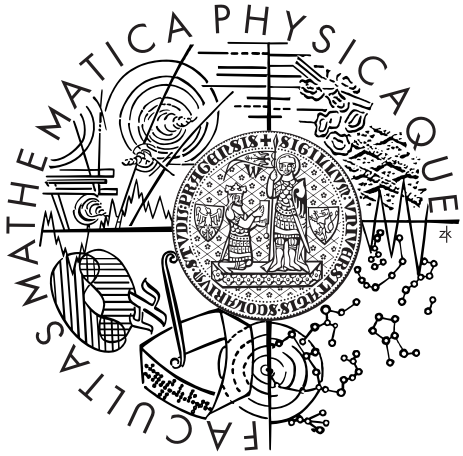
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Pelc

An IDE for C# Script Development

Department of Distributed and Dependable Systems

Supervisor:	Mgr. Pavel Ježek, Ph.D.
Consultant:	Ing. Tomáš Novotný
Study program:	Computer science
Specialization:	Software systems

Prague 2015

I would like to thank everyone who helped me with this thesis, especially my supervisor Pavel Ježek for his help with the text of this thesis and my consultant Tomáš Novotný for his support, encouragement, testing and patience. I would also like to thank Josef Závišek for his input and encouragement. Last but not least, I would like to thank my family and colleagues for their support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, April 28, 2015 Jan Pelc

Title: An IDE for C# Script Development

Author: Bc. Jan Pelc

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D.

Consultant: Ing. Tomáš Novotný

Abstract: The goal of this thesis is to explore tools for using the C# programming language to create scripts – short programs for quick and easy solving of small, usually one-time temporary tasks that usually arise during the work on larger projects. In the thesis we analyze existing tools and identify their advantages and disadvantages, formulate requirements for our own tool, and develop our own tool. The result of the thesis is a small integrated development environment (IDE) for quick and easy authoring of scripts in the C# language. The IDE offers sufficient features to allow easy authoring and debugging of programs consisting primarily of a single C# source code file. In the work we make heavy use of the NRefactory library for syntactic and semantic analysis of the C# source code.

Keywords: IDE, C#, scripting, NRefactory

Název práce: Vývojové prostředí pro vývoj skriptů v jazyce C#

Autor: Bc. Jan Pelc

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí: Mgr. Pavel Ježek, Ph.D.

Konzultant: Ing. Tomáš Novotný

Abstrakt: Tato práce se zabývá nástroji umožňujícími použití jazyka C# k tvorbě skriptů – krátkých programů určených pro rychlé vyřešení malých, typicky jednorázových úloh, které obvykle vyvstávají při práci na větších projektech. V práci zanalyzujeme existující nástroje, určíme jejich výhody a nevýhody, zformulujeme požadavky na náš vlastní nástroj a tento nástroj vytvoříme. Výsledkem práce je malé vývojové prostředí (IDE) pro rychlé a snadné psaní skriptů v jazyce C#. Prostředí nabízí dostatek funkcí pro snadnou tvorbu a ladění programů sestávajících se převážně z jediného zdrojového souboru v jazyce C#. V práci intenzivně využíváme knihovnu NRefactory pro syntaktickou a sémantickou analýzu zdrojového kódu v jazyce C#.

Klíčová slova: vývojové prostředí, C#, skripty, NRefactory

Contents

1. Introduction

The C# programming language is already over a decade a very popular tool for application development. Thanks to the Mono project [1] it can be used on other platforms than its native platform Windows and .NET, including increasingly popular mobile operating systems. During its lifetime the C# language has been extended with a lot of interesting features (e.g. lambda expressions, LINQ, or asynchronous programming support) and has attracted a broad user community, which has produced a lot of frameworks and libraries for various purposes. Furthermore, the C# language shields the developers from many low-level aspects of computing, such as pointers or memory (de)allocation. Thanks to this, it allows for a greater level of abstraction, and is also more forgiving to beginners. All this combined with the presence of advanced development tools makes the C# language a unique choice for solving a wide range of common programming problems.

However, there is one domain where the available toolset for the C# language is not sufficient. This domain is the usage of the C# language for quick and easy solving of small, often one-time temporary tasks. Because of the nature of the programs that are written to solve these tasks, we will call them *scripts*, since they are traditionally solved using specialized scripting languages (e.g. PowerShell, Bash, Python, Lua). Such specialized languages are often not very suitable for a given situation, mainly for these possible reasons:

- The nature of the problem may require users to work with the same technologies or libraries as their larger C# projects.
- The specialized language may not be available in the target environment, or it may not be practical or even possible to deploy it there.
- Users may not have the necessary knowledge to solve the problem using the specialized language, or they would be able to use the C# language to solve the problem more effectively (e.g. by taking advantage of a known language and familiar libraries, or static type checks, which are often not available in specialized scripting languages).

By the term "script" we will from now on mean a small program (usually consisting of a single source code file) created ad hoc to quickly solve a temporary task.

The situations where it is necessary to write scripts to solve such tasks arise very often in the development practice. Most common are situations where users are not necessarily required to use a specific language and there are other specialized scripting languages designed to solve the problem, but they can benefit from using the C# language for the reasons mentioned above. These may include for example:

- Small ad-hoc utilities (e.g. format conversions, downloading, extraction, data generation, password or key generation).
- Shell-like scripts (e.g. file and text manipulation, batch processing, additional build steps).

- Prototyping of more complicated algorithms and ideas.

Then there are also situations where the user needs to make quick use of the C# language itself in order to find out some information. These situations may include for example:

- API experiments – Users are not sure of the exact behavior of a certain library class or method in a specific situation, or they need to try out a code snippet, for example a non-trivial regular expression or a more complicated LINQ expression.

- Microbenchmarks – Experimental performance comparison of two different code snippets (e.g. a question of, "How much is using the `Activator` class slower when compared to direct constructor calls?").

- One-time testing (e.g. a test web service client).

Whenever such tasks need to be solved using C#, in most cases the only readily available option is to launch a new instance of a classic large IDE (integrated development environment) like Visual Studio [2] or SharpDevelop [3] and create a new project in the IDE. This is often inconvenient for quick writing of small scripts. Therefore, this work is focused on alternative options of using the C# language to create scripts. We will have a look at the various tools that have been created for this purpose so far and assess their advantages and disadvantages. Then we will specify what our own tool should look like and create it. The result of this work should be a small and easy-to-use tool for writing scripts in the C# language.

1.1 Problem statement

In the introduction we have defined what C# scripts are and stated that our goal is to look for an easy way of working with them. In this section we will look more closely at the common problems that users face when they try to use the C# language for quick solving of small tasks in a classic large IDE, and better outline the motivation behind our work.

The situations where it is necessary or helpful to quickly create small temporary programs in C# come up very often. The simplest scenario is when the user needs to immediately execute a single call, for example an API experiment. There are in general several options:

1) It is usually fastest to insert the given call into a suitable place in the currently open project, possibly place a breakpoint on it, and run the project. Such a solution is generally very quick and readily available, but it is not "clean" since it combines unrelated code and can easily lead to the pollution of the project with commented-out code snippets. Moreover, this solution may not be feasible for larger and more complicated projects whose execution flow is not trivial.

2) Another option is to execute the call during a debugging session using a debugger console. But this requires the debugger to be active, it is not very comfortable, and the debugger console usually allows only simple individual calls (there is only a limited subset of the C# language available – for example lambda expressions are usually not supported).

3) If a unit-testing framework is available, users can create and run a new test containing the given call. But this way they use something which was not meant for such a purpose, and again they pollute unrelated code, this time the unit tests.

4) The final option is to create a new project for our call and go through all the steps that such an action requires. Even though this is in principle the cleanest solution, the overhead of creating a new project is significant – users need to launch a new instance of the IDE, create a new solution and a new project, give it a meaningful name and save it into a new directory on the filesystem. Because of this overhead, users usually resort to this only as the last option.

If the code snippet is more complicated than a single call, it is often required to save it for later reference. In this case, the first three options cannot be used and the only practical and clean option is the last one – to put the code in a new project of a classic large IDE. However, doing so has many disadvantages:

- It is usually necessary to launch a new instance of the IDE. Then the user has to wait while the new instance initializes, and afterwards it uses a significant portion of system resources, especially memory. While it is possible to share a single instance and a single solution between multiple script projects, this in practice does not help much because it complicates script management and adds the inconvenience of having to switch between startup projects.

- Although the code snippet itself usually consists of no more than a single source code file, the overhead of using a classic IDE requires naming and creation of a new folder on the filesystem with a lot of additional files and subfolders in it – the solution file (`.sln`), the project file (`.csproj`), the `AssemblyInfo.cs` file, the folders `bin` and `obj` with their own specific content, the files for storing the user settings related to the environment (`.suo`, `.user`), etc.[1] For the purpose of simple scripts, all these files are redundant and such overhead only complicates the manipulation and organization of scripts, especially in larger projects where even the supporting scripts need to be organized in a repository for the purpose of version control, backups, and knowledge sharing.

[1]For example: creating, building, and running a simple console application in Microsoft Visual Studio 2010 [2] requires creating 2 folders and 5 files (excluding the `bin` and `obj` subfolders), or 8 folders and 13 files (including the `bin` and `obj` folders). SharpDevelop 4.3 [3] is only a little better in this respect, requiring 2 folders and 4 files, or 6 folders and 10 files including the `bin` and `obj` folders.

- It can be difficult to modify scripts in an environment which has limited resources or in which the IDE is not present and its installation is complicated or for any reasons (e.g. license-related, security-related or time-related) not possible. This is often the case of a production or testing environment if an immediate maintenance action is required.

Even though using a classic IDE for creating small helper programs has its disadvantages, in our experience developers who extensively use the C# language discover in time that their workspace contains many projects that were created ad hoc, without any prior planning, when they wanted to immediately solve a small task, or when they wanted just to quickly test how a certain code snippet behaved in a specific situation. Such projects are usually console applications with unpolished code (e.g. with no error handling or with hard-coded data), they usually have a very specific or one-time function, and their solution file contains a single project file with a single user-modified source code file. These programs are often typical examples of scripts as we described them in the introduction.

1.2 Goals

We have defined scripts as small programs (usually consisting of a single source code file) created ad hoc to quickly solve a temporary task. While it is possible to use classic IDEs for script development, doing so has disadvantages described in the previous section. On the other hand, when using classic IDEs for script development, users can take advantage of many features that make source code writing easier, quicker and user-friendlier, such as syntax highlighting, code completion, refactoring or debugging. What is missing is a C# tool which includes these features, but which, as opposed to classic IDEs, is also light-weight and does not require using the classic organization of source code into solutions and projects. Instead it should allow easy development of scripts.

Therefore, we will look for alternative tools (tools that are not classic IDEs) which allow using the C# language to author scripts. We will examine the various tools that have been created for this purpose so far and assess their advantages and disadvantages. Then we will specify what our own tool should look like and create it.

Summary

The primary goals of this work are:

a) Analyze the available tools that allow the usage the C# language for script development.

b) Based on this analysis, identify the advantages and disadvantages of the available tools and formulate the requirements for a new tool.

c) Analyze the requirements for the new tool and develop it.

The result of this work should be a small and easy-to-use tool for writing scripts in the C# language.

2. Problem analysis

In this chapter we analyze the relevant existing tools that allow using the C# language to write scripts as we described them in the introduction. Based on this analysis, we formulate the requirements for our own tool, further specifying the goals of our work.

2.1 Existing tools

What follows is an analysis of the existing tools that allow the usage of the C# language for script development. First, we analyze what features are of interest to us and why with respect to the scenario described in the introduction. Then, each available tool is briefly described and its relevant features, advantages, and disadvantages are stated. Finally, a conclusion based on our observation is made.

2.1.1 Features of interest

In this section we analyze the features that are of interest to us in the analysis of existing tools and explain why each feature is important and how we believe each problem should be ideally solved with respect to the aim of our work as stated in the introduction.

Ease of use

As we stated in the introduction, we are looking for a more convenient approach to writing scripts than using classic large IDEs. Therefore, we are interested in how effectively and easily the given tool can be used for script development. An ideal tool should be small, quick, easy to use, and should not strain system resources too much. We also want to be able to use it quickly in other environments than the development environment, so an ideal tool should not require any special installation and its license should allow users to deploy it to whatever machine they need.

Compatibility with the standard C# language

When users are creating small ad hoc utilities, a common scenario is that some of these utilities need to be further evolved into programs that are no longer simple scripts. In this case, users want to be able to easily transform scripts into projects of classic IDEs. Therefore, we are interested if the scripts written in the given tool can be easily used as standard compatible C# code when inserted into a project of classic IDEs, requiring only minimal changes (such as adding correct references). In an ideal tool the scripts should be compatible with the standard C# language.

Means of referencing external assemblies

The list of referenced assemblies is an information that is essential for successful compilation of any C# source code, but it is not part of the C# source code files

themselves. So the tools that we will be analyzing need to offer a way to pass this information to the compiler. Because we are looking for an easy-to-use tool, we want to avoid having to use the command line, and because the source code of a script will usually fit into a single file, we want the script files to be self-contained (without any additional information needed to use them). Therefore, in an ideal tool the additional references should be part of the source code itself, so that the script file is fully stand-alone (this should be ideally done in accordance with the previous point, so that the compatibility with the standard C# language is preserved).

Presence of a specialized C# source code editor

We are looking for an easy-to-use tool, so we want script development to be comfortable to users who are used to classic IDEs. Therefore, we are interested if the given tool includes a source code editor with features that make code authoring easier, such as syntax highlighting, error highlighting, code completion, method parameter hints, basic refactoring, or quick code navigation. An ideal tool should include an advanced C# source code editor with a rich set of features that make code authoring easier.

Debugging options

Having a debugger available is an advantage because it allows users to quickly check if the program behaves correctly or to find out why it does not. Therefore, we are interested if the given tool allows debugging of the authored scripts in a way commonly known from classic IDEs. An ideal tool should include a debugger that supports basic debugging options in a way commonly known from classic IDEs, which includes at minimum:

- An ability to pause the running program (either manually, or at a breakpoint placed into the source code, or when an exception is thrown).
- An ability to step into/over/out of called methods.
- An ability to inspect the state of the program (call stack, position within the source code, values of variables).

If the given tool does not contain dedicated debugging options, the scripts can be usually debugged by attaching a debugger from a classic IDE to the process running the script, provided that correct symbol information was generated during the compilation of the script. However, this is not a very user-friendly solution and it inherits the disadvantages of using a classic IDE for script development.

Support for code reuse

When scripts grow more numerous or complicated, it can happen that users need to share common code between them. In such a case it is useful to have an option available to reuse a piece of code between scripts, even though using multiple source code files is on the edge of our definition of scripts (it is usually better to use classic IDEs for more complicated programs that need multiple source code

files). Therefore, we are interested if the given tool allows some form of code reuse, i.e. using code from one script in another script. An ideal tool should support code reuse, for example by allowing compilation of multiple source code files into a single script, or by allowing one script to reference another. How this can be done technically is a similar question to "means of referencing external assemblies" discussed above.

Support for executable generation

When users need to execute a script on a remote machine, during batch processing, or using a task scheduler, it is more convenient if the script exists as a standalone executable file that is independent of the tool that was used to create it. Therefore, we are interested if the given tool allows generating an executable file from the script, so that it could be distributed and executed independently of the tool itself. An ideal tool should allow some form of independent executable generation.

2.1.2 Available tools

If we want to use the C# language to quickly create small temporary programs, there are some tools available already. The most known tool is:

- LINQPad [15]

By searching the Internet for tools that support scripting in C#, we were able to find:

- NScript [12]
- CSScript [13]
- ScriptCS [14]

We will also include two other tools that stand on the opposite ends of the spectrum of available tools. Even though these tools are not meant to be used specifically for scripting, they are universal and we will assess them as well for completeness:

- The C# Compiler
- A classic IDE

The last tool included in our comparison is a simple prototype of a specialized C# script editor:

- ScriptDevelop [11]

In this section we briefly describe each tool and state its relevant features, advantages, and disadvantages with respect to the features of interest discussed in the previous section.

The C# Compiler

The most primitive and bare-bones tool for compiling individual C# source files is the direct invocation of the C# compiler, which is a part of the .NET runtime environment. It allows building any application and running it. The obvious disadvantage is the difficulty and inconvenience of such an approach – it is necessary to know and enter the compiler options manually (for details see *Command-line Building With csc.exe* [18]). While this task can be made easier by using some kind of batch processing or build automation, such a solution is cumbersome and lacks flexibility.

Advantages

✓ Available as a part of the .NET runtime environment.

Disadvantages

✗ Difficult, cumbersome and inconvenient to use.

✗ External tools are needed for both code authoring and debugging.

A classic IDE

On the other end of the spectrum of available tools are classic large IDEs, such as Visual Studio [2], SharpDevelop [3] or MonoDevelop [4]. These IDEs are optimized for large projects, but they can be of course used to develop small temporary programs as well, and they offer a full-featured editor, build system and debugger. We have already stated the disadvantages of using a full-featured IDE in the first chapter, so here we will only summarize them for completeness.

Advantages

✓ Full-featured source code editor, build system and debugger.

Disadvantages

✗ It is usually necessary to install the IDE (the installation of Visual Studio is especially time-consuming)

✗ The IDE is a huge application with a lot of features not necessary for script development, and as such its system resource requirements are high.

✗ It is necessary to start a new instance of the IDE for each script.[1]

✗ The IDEs do not support compilation and running of single source code files; it is necessary to include them into projects and solutions.

[1] While it is possible to share a single instance and a single solution between multiple script projects, this in practice does not help much, complicates script management, and adds the inconvenience of having to switch between startup projects.

NScript

NScript [12] is arguably the oldest specialized tool that allows using the C# language for simple scripting. It is a small tool that compiles and runs a single stand-alone C# source code file either as a console application, or as a windowed application. It is possible to reference additional assemblies during compilation; these have to be listed in a separate accompanying file with the same name as the compiled script but with a specialized extension.

Advantages

- ✓ Automatic compilation and running of single-file scripts without the need to invoke the compiler manually.

Disadvantages

- ✗ Another file is required to list additional references.
- ✗ External tools are needed for both code authoring and debugging.

CS-Script

CS-Script [13] is another specialized tool that allows using the C# language for scripting. It is far more advanced than NScript, and can be used not only to compile and run scripts, but also as a runtime library for dynamically loaded C# code, treating the C# language as an embeddable scripting language for extension of applications by allowing them to run any user-supplied code. It even offers some interesting features in this respect, such as duck-typing (a class loaded from a script can be mapped on a new instance of the host-supplied interface without actually having to implement it formally in the code).

Another interesting feature is automatic guessing of referenced assemblies based on namespaces imported with the `using` statement. This brings the advantage of user-friendliness, but can also cause problems in case of conflicts or inability to guess the assembly correctly. Because of this, it is also possible to state additional references either on the command line, or directly in the code in form of annotations enclosed in comments.

CS-Script compensates the absence of an editor or a debugger by offering a special plugin for Visual Studio. This plugin allows using Visual Studio to edit and run CS-Script scripts, but this approach also inherits most of the downsides of using classic IDEs, except the ability to run single C# files. For debugging, CS-Script offers a tool that can convert scripts into projects and solutions that can be opened and debugged in Visual Studio.

Advantages

- ✓ Automatic compilation and running of single-file scripts.
- ✓ Reference guessing.
- ✓ Contains tools that make it easier to use Visual Studio as an editor and debugger.
- ✓ Supports code reuse.

Disadvantages

✗ External tools are needed for both code authoring and debugging.

ScriptCS

ScriptCS [14] is, much like CS-Script, another specialized tool that allows using the C# language for scripting. Unlike CS-Script it uses a different approach to the C# language itself, called REPL (Read-Evaluate-Print Loop). It allows processing of the C# language in a command-line-like manner, immediately evaluating the input statements and printing out the results, giving the language a more script-like look and feel. Because of this, it does not use the stock C# compiler, but rather makes use of Microsoft Roslyn [5], an upcoming compiler-as-a-service library from Microsoft.[2]

For specifying additional references, ScriptCS uses special directives in the source code that are not enclosed in comments. This, in addition to the REPL approach to the C# language, makes script files written for ScriptCS incompatible with the standard C# language. However, the language is compatible with Microsoft Roslyn.

For code authoring and debugging, Visual Studio with Roslyn CTP [5] can be used.

Advantages

✓ Automatic compilation and running of single-file scripts.

✓ The REPL approach gives the source code more script-like look and feel.

✓ Supports code reuse.

Disadvantages

✗ Incompatible C# code.

✗ Visual Studio (with Roslyn CTP) has to be used as an external source code editor and debugger.

LINQPad

LINQPad [15] is a popular tool which can, to a certain extent, replace applications like SQL Management Studio [19] which are used for configuring, managing, administering, developing and querying SQL databases. It is primarily focused on querying databases using the C# language and LINQ, and allows writing and executing expressions, statements, methods and classes in the context of a database connection.

Because of the presence of a code editor and some other features (e.g. the universal "dump" command that allows complex output of any object into a special window), LINQPad is often used as a small light-weight development environment for simple scripts.

[2] Microsoft Roslyn is further discussed in section 3.1.4.

One of the disadvantages of LINQPad is that it uses custom XML format for saving scripts. This format allows storing some additional information about the script, such as its type (if it is an expression, statement, etc.), imported namespaces, and referenced assemblies.

Advantages

- ✓ Small and easy to use tool.
- ✓ Allows quick running of single expressions, statements, methods, or whole scripts.
- ✓ A full-featured C# editor is present (although most of the important features are available only in the paid version).

Disadvantages

- ✗ Incompatible C# code (XML script format, single expressions or statements).
- ✗ No executable generation (the scripts are executed in a special hosting environment and cannot be used outside LINQPad).
- ✗ No support for code reuse.
- ✗ Most of the important code-editing functions are available only in the paid version.
- ✗ External tools are needed for debugging.

ScriptDevelop

ScriptDevelop [11] is a simple prototype of a specialized C# script editor that was created as a bachelor thesis on Czech Technical University (ČVUT). It is small, easy to use and supports editing, compiling and running of stand-alone single-file console applications in C#.

While the editor supports simple syntax highlighting and code completion, it does not contain any further code-aware functions like method parameter hints, reference completion, or code navigation. The code completion which is supported is very limited – it offers types and members only from a limited hard-coded set of assemblies. There is also no support for debugging or any form of code reuse (each source code file is strictly stand-alone).[3]

Advantages

- ✓ Small and easy to use tool.
- ✓ Automatic compilation and running of single-file scripts.

[3]We will further analyze ScriptDevelop in section 3.2.

Disadvantages

- ✗ No support for code reuse.
- ✗ No support for debugging.
- ✗ The editor contains no code-aware functions besides the limited code completion.

2.1.3 Summary

Table 2.1 summarizes the features of the existing tools. As can be seen from the already available solutions, using the C# language as a scripting tool is a desired functionality. During the lifetime of the C# language a variety of attempts has been made to solve this problem, because using the command-line compiler or a full-blown IDE has many downsides that the specialized tools try to address.

	Easy to install and use	Compatible with standard C#	References in the source file	Advanced source code editor	Debugging options available	Code reuse supported	Executable generation	Free of charge
The C# Compiler		✓				✓	✓	✓
A classic IDE		✓		✓	✓	✓	✓	[4]
NScript		✓						✓
CS-Script		✓	✓			✓	✓	✓
ScriptCS			✓			✓	✓	✓
LINQPad	✓		✓	✓				[5]
ScriptDevelop	✓	✓	✓				✓	✓

Table 2.1: Summary of the features of the existing tools.

None of the solutions we evaluated fully satisfy our requirements for an ideal tool for developing C# scripts. If we are interested solely in compilation and running of stand-alone C# source files, the most interesting and advanced tool seems to be CS-Script, which even contains tools that allow using Visual Studio to author and debug scripts. However, the popularity of LINQPad for solving small tasks unrelated to databases (which are the primary focus of LINQPad) is also a sign that there is a desire for a small, quick and easy to use editor for C# scripts as an alternative to classic large IDEs. Developing such an editor is the primary goal of this work.

[4] The most common C# IDE, Visual Studio, is free of charge only for limited purposes in its Express edition. SharpDevelop and MonoDevelop are free or charge and open-source.

[5] Most of the important code-editing functions in LINQPad are available only in the paid version.

2.2 Requirements

In the previous section, we have examined the existing tools that allow using the C# language for script development. We have studied their features and identified their advantages and disadvantages. In this section we look more closely at some of the points of interest stated in section 2.1.1, and based on our observations of the existing tools we formulate exact requirements for our application.

2.2.1 Non-functional requirements

We have stated several times that we aim for a "small, quick and easy to use" tool. Here we specify what exactly it means for our application in form of non-functional requirements.

Familiarity

Because the target user audience of our application is developers who routinely use classic IDEs, special effort will be made so that they would find the functions they want to use quickly and effortlessly. This includes menu structure, graphical symbols, command names, default keyboard shortcuts, etc.

- **R1.1**: When designing the user interface, we will attempt to make the application behave in an expected way for developers that are used to classic IDEs.

Installation

- **R1.2**: The application will not require any special installation. It will be deployable simply by copying it, along with all of the files in its program directory, to any machine that has a Microsoft .NET Framework [16] of the correct version installed.

Startup

- **R1.3.1**: The application's startup process will not add any extra overhead on top of what is expected of a typical .NET application. Any additional startup processing will be done in background once the user interface of the application is loaded and presented to the user.

To help better manage system resources usage in an environment where system resources are limited:

- **R1.3.2**: The application will allow nonessential components to be loaded later, after the core of the application is presented to the user and ready. Loading of nonessential components will be optional and potentially able to be turned off. Unloading of already loaded components is not required.

Stability

The application will be used to perform creative work, and so it should be protected from work loss in case of a malfunction. While malfunctions should ideally not happen, such a scenario must be taken into account. There are two main steps that will be taken to protect the application:

- **R1.4.1**: If possible, a potential unexpected exception will be caught, logged and if necessary presented to the user without terminating the entire application.
- **R1.4.2**: The application will keep backup copies of any unsaved documents, and if it is terminated unexpectedly, it will offer to restore these backups at next launch.

Reentrance

To better support organization of windows and documents:

- **R1.5**: The application will be able to run in multiple instances. These instances will not interfere with each other.

Localization

Localization support is not needed because the target user audience are developers who routinely use classic IDEs, where English is common.

- **R1.6**: The user interface of the application will be in English.

Extensibility

It is not expected that the application will be a complete, all-inclusive IDE, but rather a solid and well-usable basis that could be further extended and improved by other students and developers based on their specific requirements, ideas and needs.

- **R1.7**: The application will be designed in such a way so that it could be easily extended by other developers.

2.2.2 Editor

The application will contain an advanced C# source code editor with a rich set of features that make code authoring easier. In this section we discuss what this means in greater detail and decide what features are important for us to include.

Syntax highlighting

Syntax highlighting is an important feature of any source code editor because it makes the displayed source code easier to read and understand. We want to include advanced syntax highlighting that is not based only on simple regular expressions, but also on the real output from the syntactic and semantic analysis

of the C# source code. This way we can distinguish between identifier roles that we could not tell apart using only regular expressions, such as between type and namespace names, or we can correctly highlight contextual keywords that are not reserved in the C# language, but recognized as keywords only in certain syntactic constructs (e.g. keywords specific to LINQ expressions).

- **R2.1.1**: The editor will support syntax highlighting.
- **R2.1.2**: The syntax highlighting will use syntactic and semantic analysis of the C# source code where appropriate.

Code completion

While the user types identifier names, code completion offers a list of available types, members, variables and other constructs. It allows to choose an identifier without the need to remember its exact name or without having to type out its name in full.

- **R2.2.1**: Code completion will appear automatically when the user types an identifier, but only in places where it is expected so that it would not interfere when we want for example to name a new type, member, or variable.
- **R2.2.2**: In places where it is syntactically possible both to introduce a new identifier or to use an existing one (such as when passing a lambda delegate as a real parameter), the code completion will appear in a different mode so that if the user introduces a new identifier, the code completion will not interfere.

Because users rarely remember the exact namespace or assembly of a library type by heart, these requirements are essential to allow quick and easy code authoring:

- **R2.2.3**: In addition to all entities that can be used at the current position, the code completion will also offer types and extension methods from namespaces that are not currently imported with a `using` statement and import the namespaces accordingly when such an item is selected.
- **R2.2.4**: Code completion will also offer types and extension methods from yet unreferenced but otherwise known (or precached) assemblies, and automatically reference the correct assembly upon selection.
- **R2.2.5**: Code completion will display the relevant XML documentation of the entities where available.

Parameter hints

Parameter hint is a pop-up window that appears when the user writes a method invocation. It displays the list of invoked method's overloads, the list of parameters for the currently selected overload, and the current parameter. It allows to write a method call without the need to remember its exact overloads and parameters by heart.

- **R2.3.1**: The editor will support the parameter hint pop-up window.
- **R2.3.2**: The relevant XML documentation of the parameters will be displayed in the parameter hint pop-up window where available.

Refactoring

Most of the refactoring transformations offered by classic IDEs and other advanced tools are well suited for large projects where there is a significant benefit from using them to make the code cleaner, more organized, and easier to read. Most of these transformations are not very important in our scenario where we primarily focus on single-file scripts. There are, however, some refactoring transformations that are important enough so that we will implement them as well:

- **R2.4.1**: Renaming of symbols (types, members, variables, namespaces).
- **R2.4.2**: Automatic assembly referencing and namespace importing during code completion when a type or an extension method from yet unimported namespace or yet unreferenced assembly is selected.
- **R2.4.3**: Automatic assembly referencing and namespace importing when there is a type or an extension method already used in the source code which cannot be resolved, but is located in another namespace or assembly. In this case, a context action that will add the corresponding namespace or reference will be offered.

Navigation

- **R2.5.1**: The application will support the "Go To Definition" command known from classic IDEs. This command will navigate the user to the definition of the symbol at the cursor's position in the source code.
- **R2.5.2**: The application will support the "Find References" command known from classic IDEs. This command will highlight or list all occurrences of the symbol at the cursor's position in the source code.
- **R2.5.3**: There will be a way to quickly navigate between types and members of the current source code file.

More advanced navigation commands are not necessary in our scenario.

Tooltips

- **R2.6.1**: When the mouse cursor is moved over a type, member, or variable reference in the source code, the application will display a tooltip with the definition of the symbol.
- **R2.6.2**: The summary of the symbol's XML documentation will be displayed in the tooltip where available.

2.2.3 Compilation

- **R3.1**: The application will be able to compile and run programs consisting of a single C# source code file. There will be no need to create and manage projects or solutions as known from classic IDEs. Each source code file will be self-contained and there will be no additional information required to compile it.

Language compatibility

- **R3.2.1**: The source code written in the application will be compatible with the standard C# language.

- **R3.2.2**: When inserted into a project of classic IDEs, the source code written in the application will require only changes to the configuration of the project itself (such as adding correct references).

Compilation instructions

A single C# source code file cannot be compiled on its own. The C# compiler needs additional information, at least the list of referenced assemblies. This information is essential for successful compilation of C# source code, but it is not part of the C# source code files themselves. In classic IDEs, this information is associated with the project and stored in the project file. Since our application will not have project files, we need another way to pass this information to the compiler.

We want the script files for our application to be self-contained, so there must not be any additional information required to use them (see R3.1). Because of this, the additional compilation instructions will be part of the source code itself. Since we do not want to introduce any custom syntax that would break the compatibility with the C# language (see R3.2), the additional compilation instructions (e.g. external references) will be enclosed in C# comment blocks.

- **R3.3.1**: Any additional compilation instructions will be part of the script's source code itself.

- **R3.3.2**: Any additional compilation instructions will be enclosed in C# comment blocks.

Code reuse

Although code reuse support is not an obvious requirement for an IDE focused on single-file scripts, it is useful in case the scripts grow more numerous or complicated and we need to share common code between them. The application will support two mechanisms of code reuse:

- **R3.4.1**: One script will be able to include another. From the compiler's perspective, this will behave as if there were multiple source code files contributing to a single output assembly. If needed, this can be used to emulate a project consisting of multiple source code files.

- **R3.4.2**: One script will be able to reference another. From the compiler's perspective, this means that the referenced script will be compiled first and the resulting assembly will be used as a reference of the referencing script. If needed, this can be used to emulate a solution consisting of multiple projects.

These mechanisms will be implemented as compilation instructions the same way as described in the previous section (see R3.3).

Executable generation

Having an independent executable is more convenient when we need to execute the script on a remote machine, during batch processing, or using a task scheduler.

- **R3.5**: The application will support generating an executable file from the script, so the executable file could be distributed and executed independently of the application itself.

2.2.4 Debugging

The application will offer basic debugging options so that the users could inspect the behavior of their scripts. Based on options available in classic IDEs, this means:

- **R4.1**: Manual pause and resume of the execution.
- **R4.2**: Ability to set a breakpoint on a line.
- **R4.3**: Pause when an exception is thrown.
- **R4.4**: Highlighted position within the source code.
- **R4.5**: Stepping (step into, step out, step over).
- **R4.6**: Call stack and thread list views.
- **R4.7**: Debugger console (a command line for executing statements and queries).
- **R4.8**: Tooltips with values of variables when hovering the mouse cursor over the variables in the source code.

Complex visual object inspector is not needed; querying the current state (e.g. in the debugger console or tooltips) will use simple string representation to show the value. More complex objects can be queried using the debugger console.

3. Implementation analysis

In the previous chapter we have analyzed the existing tools that allow the usage the C# language for script development and formulated detailed requirements for our own tool. This chapter is focused on technical decisions regarding how our own tool will be implemented.

3.1 Tools and libraries

In this section we choose what tools, libraries and frameworks will be used for the development of our application. In general, we will prefer libraries that are not "black boxes", but instead:

1) We can look into their source code if the available documentation is not sufficient for us.

2) We can modify them if the library does not behave exactly as we need in some aspect or if it does not expose the exact information we want.

This usually means libraries written in C# with source code available under a permissive license. However, this is only a preference, not a strict requirement.

3.1.1 Basis

As can be seen from the requirements specified in section 2.2, we are aiming to create a special kind of development environment for the C# language. The task of creating a development environment is a very demanding one, especially when we want to include features such as an advanced source code editor or a debugger. Creating such an application from scratch would be a very large and complicated endeavor. To make the project manageable, we can approach it from two different directions:

a) Create an add-on for an existing classic IDE or use an open-source classic IDE with a permissive license and modify it to suit our goals.

b) Build our application from the ground up, but make heavy use of the existing components and libraries, many of which were specially designed to be used in IDE development.

Existing classic IDEs are large and complicated; as we stated in the introduction, we want our application to be a light-weight alternative to them. Because of this, we will use the second approach, which also gives us better control over our application's architecture.

3.1.2 Programming environment

Because we are developing a tool for the C# language and the C# language is the language of choice of the author, we will use the C# language for the development. The target platform will be *Microsoft .NET Framework 4.0* and we will use *Microsoft Visual Studio 2010*, both being the latest versions when the work on this project started.

3.1.3 User interface

Microsoft .NET Framework offers two different sets of APIs for user interface development – *Windows Forms* [20] and *Windows Presentation Foundation* (WPF) [21].

WPF is more modern, offers greater flexibility and has many features and abilities that Windows Forms lacks, for example data binding and templates, relative layout, greater customizability, or hardware acceleration. It is well established and widely supported by third-party controls and libraries. Because the author of this work would like to gain more experience with WPF, we will use it in our application.

Source code editor

A source code editor is an integral part of any IDE. It is the component the user interacts with the most. Because we want to be able to highlight various portions of the source code, it must support custom rendering.

Two most known and used libraries were considered – *AvalonEdit* [22] and *ScintillaNET* [23]. Both support a wide range of additional features that are useful for source code editors.

AvalonEdit is written in C# and WPF, whereas ScintillaNET is a managed Windows Forms wrapper around an unmanaged Scintilla [24] control. Because of this, we chose AvalonEdit. An overview of the AvalonEdit library can be found in *Using AvalonEdit* [8].

Docking manager

The user interface of an IDE is usually quite complex and must support working with not only many open documents at once, but also windows and panels with various additional information (e.g. a filesystem explorer, an error list window, or a call stack panel).

A common approach of how to present this environment to the user is called a docking manager. It is a high-level layer of an application's user interface which manages windows with open documents and panels. It creates a tabbed interface for the open documents and allows the user to move the windows around, snap them to the edges, combine them, split them, or hide them.

This docking behavior is an expected part of modern IDEs and we will also use it. It is not a standard part of the WPF framework, but since the docking behavior is quite common in applications with more complicated user interfaces, there are many available libraries that implement it. Most of the implementations are commercial, but because we prefer open source, C# and we will use WPF, we chose *AvalonDock* [25] for this purpose.

3.1.4 Syntactic and semantic analysis

One of the most important goals when creating a C# code editor is the ability to analyze and understand the source code that is being written. This ability is called syntactic and semantic analysis. There are various features of a source code editor that need source code understanding to work properly – for example code

completion, method call parameter hinting, navigation, or syntax highlighting (basically all the R2 requirements from section 2.2.2). We need not only a C# parser which would provide us with the syntax tree of the code, but also a means of semantically analyzing the code so that we would be able to resolve the *meaning* of the parsed code (e.g. know exactly what types, methods and variables are being used in the code).

The syntactic and semantic analysis of C# source code is a very large and difficult problem which requires complete understanding of the specification of the C# front-end. Therefore, we will look for a library that would help us with this problem.

There are currently two libraries that can be used for this purpose – *NRefactory* and *Microsoft Roslyn*:

- **NRefactory** [6] is a C# library that allows parsing and semantic analysis of C# source code. It also contains support for refactoring, formatting, code issue analysis and code completion. NRefactory has been created as a part of SharpDevelop [3] specifically for the purpose of supporting its code-aware functions, so it is well suited to be used in IDE development. It also contains a large library of ready-made C# refactoring transformations and code issue detectors which can be easily interfaced with. Its current version 5, which has been a complete rewrite, is stable.

- **Microsoft Roslyn** [5] is an open-source compiler and code analysis library for C# and Visual Basic .NET languages. It allows syntactic and semantic analysis of C# and Visual Basic source code, dynamic compilation to IL, and refactoring. It is currently in a beta stage and available as a preview version.

Both libraries are capable of syntactic and semantic analysis of C# code, however, NRefactory is better suited to be used in an IDE development because it also supports code formatting and completion and contains various refactoring actions and code issue detectors. Roslyn has some extra features, such as IL generation and Visual Basic support, but these are not important with respect to our goals. Moreover, NRefactory is open source (Roslyn had not been open-sourced yet when work on this project began) and stable. Roslyn is still under development and only preview versions have been released so far.

Based on these arguments, we will use NRefactory in its current version 5 for our project. The library is introduced in depth in *Using NRefactory for analyzing C# code* [7].

3.1.5 Debugging

One of the features that we will need to implement is the support for debugging the code written in our application (requirements R4 from section 2.2.2). A running CLR application can be debugged using a rather complicated `ICorDebug` API, which is a set of public COM interfaces. However, the API is very large and very little documented (some documentation can be found in *Debugging Interfaces* [26]).

Luckily, there is an open source C# library built on top of this API. It is a part of SharpDevelop and called *Debugger.Core*. It shields us from the complexity of

the `ICorDebug` API and provides a set of classes that we can use to implement debugging functionality. We will use this library in our application to implement the debugger.

The overview of the library can be found in *Internals of SharpDevelop's Debugger* [9]. The library is also used to implement debugger visualizers in *Debugger Visualizers for the SharpDevelop IDE* [10], which also describes how to work with the library.

3.1.6 Extensibility

One of the goals of our application is to make it extensible (R1.7). Because of this the application will be implemented as a set of many plugins that use one another's services. A plugin in this context will be a separate unit of functionality that extends the application or one of its parts in some way. Therefore, we will need a means of discovering, loading, and initializing these plugins at runtime. We are not looking for any inversion of control or dependency injection mechanisms, instead we need a naming and activation service. The required functionality is this:

- Discover and load all assemblies with plugins in the application's directory.
- In these assemblies, find all classes that implement a specific interface.
- Create instances of these classes and return them on demand.

There are many tools, libraries and frameworks that can be used to perform these tasks. Microsoft .NET Framework itself contains two frameworks that support extensibility – *Managed Add-in Framework* (MAF) [27] and *Managed Extensibility Framework* (MEF) [28]. MAF does not suit our needs very well because 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. MEF, on the other hand, is light-weight and covers the features we need, so we will use it in our application.

3.2 Previous work

ScriptDevelop [11] is one of the tools we described in section 2.1.2. It is a simple prototype of a specialized C# script editor that was created as a bachelor thesis on Czech Technical University (ČVUT). It supports editing, compiling and running of stand-alone single-file console applications in C#. The editor supports simple syntax highlighting and limited code completion.

The goals of ScriptDevelop are a subset of our own goals and it uses some of the libraries that we chose to use (WPF, AvalonEdit, AvalonDock), so we will consider it as a starting point that we can rewrite and extend to meet our goals.

There are many ways in which ScriptDevelop can be improved to meet our requirements as stated in section 2.2:

- It contains only simple regular-expression-based syntax highlighting. We want to use syntax highlighting which takes into account the real output

from the syntactic and semantic analysis of the C# source code (R2.1.2). This way we can distinguish between identifier roles that we could not tell apart using only regular expressions, or we can correctly highlight contextual keywords that are not reserved in the C# language but recognized as keywords only in certain syntactic constructs.

- It contains only very limited code completion:
  - Code completion is displayed only upon manual invocation. We want code completion to appear automatically when the user types an identifier (R2.2.1). This is not as straightforward as it seems, because we cannot simply display it every time an identifier is typed. Code completion must not interfere in places where a new identifier is expected, and must also support cases where it is syntactically possible both to introduce a new identifier or to use an existing one (R2.2.2).
  - Code completion offers types and members only from a limited hard-coded set of assemblies. We want code completion to automatically include all referenced assemblies.
  - Code completion does not offer items from unimported namespaces and unreferenced assemblies. We want code completion to support a mode where such items are offered and correct namespaces and references are added upon their selection (R2.2.3, R2.2.4).
  - Code completion is based on older SharpDevelop components and a previous version of NRefactory. We want to use more powerful NRefactory 5, which is a complete rewrite and has a different interface.
- It contains no debugging functionality. We want to offer basic debugging options so that the users can inspect the behavior of their scripts (R4).
- It contains no code-aware functions besides code completion. We want to offer many code-aware functions, including parameter hints, tooltips, refactoring, and navigation (R2).
- It does not allow code reuse. We want to support code reuse by allowing scripts to include or reference one another (R3.4).

We will use ScriptDevelop as a prototype starting point for our own application, but we will have to rewrite it significantly in order to meet the requirements for our application. The most important tasks that we will need to do include:

- Implement on-the-fly parsing of the source code using NRefactory 5.
- Add support for code reuse and make sure it cooperates correctly with the on-the-fly parsing from the previous point.
- Implement full code completion system according to our requirements.
- Implement other missing code-aware functions (proper syntax highlighting, parameter hints, tooltips, navigation, refactoring).
- Implement a debugger.

3.3 Application architecture

An integrated development environment is a complex application to build. To simplify its development, we can separate it into several primary components, each of which will be focused on a different functional aspect of the application. A primary component, in this sense, will be a large portion of the application, containing many classes or even assemblies. There will be these primary components:

- *Application Core* will contain the main entry point and infrastructure of the application.
- *UI Core* will implement the main window and basic user interface of the application.
- *Editor* will provide the concept of documents, their formats and editors.
- *Compiler* will contain the basic C#-related functionality that does not require code-aware insight into the source code, including commands to build and run the scripts.
- *NRefactory* will include all the code-aware features that require on-the-fly syntactic and semantic analysis of the C# source code.
- *Debugger* will implement the basic debugging functionality.

3.3.1 Plugin infrastructure

One of our requirements is to make the application extensible (R1.7). Because of this, we will implement the application as a set of many plugins that will use one another's services. This will also simplify the development of the application, allowing us to gradually extend it by adding new plugins. The core of the application will contain only the basic infrastructure for application startup and plugin management. All other functionality will be implemented as plugins.

Plugin discovery

In order to simplify adding new functionality into the application and reduce boilerplate code, plugins will be initialized automatically. All that will be required is to include them 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 also need additional metadata.

Plugin initialization

A common scenario is that a plugin will require some kind of initialization. This initialization could happen in the plugin's constructor, but plugin instances will be created by MEF, and we will need finer control over when and where the initialization takes place. Therefore, we will introduce a common base interface for plugins.

```
[InheritedExport]
public interface IPluginBase
{
    void Load();
}
```

Any plugin that implements this interface will have its `Load` method called when it is loaded into the application. Using the `Load` method for plugin initialization is preferred to its constructor for several reasons, especially if the plugin needs to interact with other parts of the application:

- The `Load` method will be called after the plugin instance has been fully composed by MEF.
- The `Load` method will be guaranteed to run on the UI thread, so its implementation can safely access UI elements.
- The `Load` method will help us better manage plugin initialization dependency.

Plugin initialization dependency

Plugins of our application will often depend on each other during initialization, which means that one plugin at some point inside the `Load` method will require that another plugin has already been loaded and is ready to be used, i.e. had its `Load` method already called. This may not generally be the case because the order in which the plugins are initialized is arbitrary. There are several ways this could be solved:

- Use dependency injection through constructor parameters. The instances passed as constructor parameters must already have their own constructors called, so we can be sure any initialization done in their constructors has already happened. However, as we stated above, we prefer to use the `Load` method for plugin initialization, so this is not a very useful solution.
- Have all public methods and properties of the plugin first check whether the plugin has been initialized (i.e. its `Load` method has been called) and if not, call it first. However, this approach introduces a lot of boilerplate code into individual plugins and introduces thread-safety concerns.
- Have the plugin infrastructure itself keep track of which plugins had their `Load` method called and which had not. The infrastructure would then call the `Load` method appropriately.

We will use the last approach. The plugin infrastructure will expose a method such as this:

```
public static void EnsurePluginLoaded(IPluginBase plugin)
```

This method will ensure that the plugin instance passed as an argument has been loaded. If it has not, it will be loaded synchronously before continuing. The method will also be implemented in such a way that it is thread-safe and executes the `Load` method on the designated thread.

The `EnsurePluginLoaded` method could be then called either by the calling plugin, or the called plugin itself, as the first call of its public methods and properties that require prior initialization (this still introduces boilerplate code, but not so much, and it also solves thread-safety problems).

Plugin initialization dependency cycles

The plugin initialization dependency resolution that will be done by the infrastructure's `EnsurePluginLoaded` method will need to correctly detect and prevent dependency cycles. Such cycles can happen for example if two plugins in their `Load` methods depend on each other already having been initialized. Without cycle detection, the execution of `Load` methods would end up in an infinite loop and eventually overflow the call stack. Instead, we will have to detect this and throw an exception with detailed description of the detected cycle so that we could better debug these situations.

We will do this in the `EnsurePluginLoaded` method by remembering which plugins are currently having their `Load` method called and detecting if it is entered twice for the same plugin. This will work trivially because the `Load` method will always be called on the same thread.

Singleton plugins

The most common type of plugin in the application will be a singleton plugin, i.e. a plugin that exists as a single shared instance. MEF supports two ways of accessing the single instance of such plugins from other plugins:

- Injecting them into a property or a field using the `Import` attribute.
- Passing them as a parameter into the constructor.

This introduces a lot of boilerplate code and works only for objects that are themselves being composed by MEF. In order to simplify the process of importing singleton plugins, allow them to be easily accessed from any class, and to make sure they are only accessed after they had been properly initialized, we will introduce a special generic base class with the following interface:

```
public abstract class SingletonPlugin<T> : IPluginBase
{
    public static T Instance { get; }
}
```

This class will allow accessing the singleton instance of a plugin of type `T`. If the `T` plugin implements `IPluginBase`, the `Instance` property will also ensure that the `Load` method has run before returning the instance (using the `EnsurePluginLoaded` method described above). This way, accessing singleton plugins via their `Instance` static property ensures that the plugin instance we access has already been initialized correctly. For example, if we define a plugin like this:

```
public class CustomPlugin : SingletonPlugin<CustomPlugin>
{
    public void DoSomething();
}
```

We can then access this plugin from anywhere in the code by simply writing:

```
CustomPlugin.Instance.DoSomething();
```

Accessing a singleton plugin like this will ensure it has been properly initialized.

Plugin dependency

The requirement R1.3.2 states that we must allow nonessential components to be loaded later. Because of this, we will need plugins to be loadable incrementally at an arbitrary point in time of the application's life cycle. In order to do this with sensible granularity, we will allow loading of plugins one assembly at a time. However, this introduces a problem with plugin dependency.

Consider the situation shown in figure 3.1. There are two singleton plugins `A` and `B` in separate assemblies. We have added plugin `A` into our MEF catalog, but not yet plugin `B`, on which plugin `A` depends. Since the assembly of plugin `A` references the assembly of plugin `B`, both assemblies have been loaded into the application, but only assembly of plugin `A` has been added to the MEF catalog. Now, if we try to access plugin `B` from `A` using the `Instance` syntax, we encounter an error, because MEF does not know about plugin `B`.

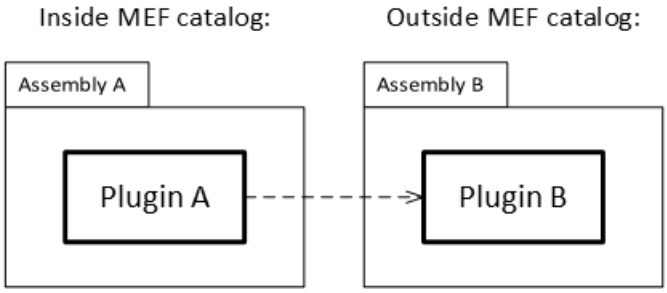


Figure 3.1: Example of the plugin dependency problem.

This could be solved when adding an assembly into the MEF catalog by going over its references and recursively adding them into the catalog as well. However, this approach would fill the catalog with a lot of assemblies without any plugins (framework assemblies, external libraries). Alternatively, we could scan each reference for plugins and add only references with plugins into the catalog, however such scanning would introduce additional overhead comparable to that of adding the assemblies into the catalog outright, since MEF internally performs a similar scan.

Instead, we will solve this simply by introducing a convention that only assemblies with a certain name (e.g. `Plugin.*.dll`) can contain plugins. When adding an assembly into the MEF catalog we will still recursively scan all its references, but we will add a specific reference to the catalog only if its name conforms to the convention.

3.3.2 Temporary storage

The application will make heavy use of temporary storage. We will need to use it for example to:

- Save new documents, so that they can be compiled and run without the need to be named and saved first by the user.
- Save results of compilation (the `.exe`, `.dll` and `.pdb` files) so that scripts can be compiled and run without cluttering the directory of their source code.
- Save modified but yet unsaved documents, so that they can be restored in case the application is unexpectedly terminated.

There are three primary requirements on the temporary storage management of our application:

1) When it is starting or exiting, the application must clean up the temporary storage so that it would not remain cluttered with files that are no longer needed.

2) Because the application must be able to run in multiple instances (requirement R1.5), special effort needs to be made to make the temporary storage management behave correctly with multiple application instances running concurrently in the same environment.

3) Because we will need to implement features like restoring lost work after a crash (requirement R1.4.2), the temporary storage management must be able to access temporary storage of application instances that were terminated unexpectedly.

This problem cannot be properly solved by simply having the multiple instances of the application use different locations for temporary storage. This is because for features like restoring lost work after a crash, the different instances have to access a common temporary storage location. On the other hand, if a common temporary storage location were used and the user launched a second instance of the application, this second instance would see the temporary storage of the first instance, which is still running, and incorrectly assume that it had been terminated unexpectedly.

Solution

We will solve the problem using the following algorithm. There will be a common temporary storage location. Each new instance of the application will:

1) Generate a unique identifier in the form of a random UUID.

2) Acquire a global mutex from the operating system with this UUID as a name.

3) Create a directory in the common temporary storage with this UUID as a name.

When the instance exits normally, it releases its global mutex and deletes its directory. If the instance in terminated unexpectedly, the directory remains, but the mutex is marked by the operating system as abandoned.

When a new instance of the application is launched, it performs a temporary storage cleanup step during its initialization. This cleanup step is protected by another (separate) global mutex to ensure that no two instances of the application perform it at the same time. During this initial cleanup step, the application looks into the common temporary storage and if it sees any directory there, it checks the global mutex with the same name as the directory. This mutex can be either:

a) Already acquired, which means that the instance the directory belongs to is still running.

b) Free or abandoned, which means that the instance the directory belongs to was terminated unexpectedly and the directory is abandoned.

If we find that the directory is abandoned, we can examine it to see if it contains anything that should be taken care of (e.g. unsaved documents to be restored) and then delete it. If the directory is not abandoned, the associated application instance is still running and we leave the directory intact.

3.3.3 Configuration and settings

The application will need to be configurable so that the user could customize some of its aspects. There are a few basic points to consider when designing how the application will be configured:

- Where the configuration will be stored and in what format.
- How the user will configure the application.
- How the configuration will be accessed in code.

User interaction

There are two opposite approaches to how the user can configure the application:

a) The application can provide detailed UI for configuration.

b) The application can provide human-readable configuration file that the user can edit.

In order to simplify initial UI design of the application, we chose the second approach, i.e. to have minimal configuration-related UI and instead use a human-readable configuration file for customizing the application. This will allow us to add the configuration UI later if needed. To make configuration a little more user-friendly, we will also:

- Allow the configuration file to be quickly opened within the application itself.

- Apply the new configuration the moment that the edited configuration file is saved.

The configuration file will use the XML format because it is both human-readable and easily manipulable in code. The file's default location will be in the same directory as the application's executable. We will keep the default configuration file embedded within the application itself and create a copy if the user decides to modify it.

Configuration vs. settings

For the purpose of implementing configuration in code, we will use two different terms:

- *Configuration* will allow the user to configure the behavior of various components of the application. It will be user-defined.
- *Settings* will allow various components of the application to persist their state between sessions. Users will not be able change this state directly.

For example, family and size of the editor's font or mapping of keyboard shortcuts to commands will be read from configuration. On the other hand, settings will persist information such as the window position and state (so that it can be restored next time), list of currently open documents (so that they can be open again next time) or list of recently open documents available from the main menu.

Access in code

In order to make manipulation with configuration and settings simple when writing the application, we will serialize and deserialize it transparently. Each configuration section will have its own XML subtree in the configuration file and will be implemented by a separate singleton plugin which will have its properties deserialized automatically upon initialization. The plugin will also contain an event that will be triggered each time the configuration file has changed so that the application can reload the modified configuration.

3.4 Compilation

We have stated our requirements for the compilation of C# scripts in section 2.2.3. In this section we discuss how we will satisfy these requirements.

3.4.1 Header sections

Since we want the script files for our application to be self-contained (requirement R3.1), we need some way to pass additional information (the information that is part of the project files in classic IDEs) to the C# compiler. In order not to introduce any custom syntax that would break the compatibility with the C# language (requirement R3.2), we have decided to put the additional compilation instructions (e.g. external references) to special C# comment blocks. We will call these special C# comment blocks with additional instructions *header sections*.

Header sections will be enclosed in block comments because block comments are usually used less in the C# language. They will need to have some kind of opening and closing tags, so that they could be distinguished from other comments. They will also need to have some kind of structure. Therefore, a header section will look like this:

```
/*HEADER**
    ...
**HEADER*/
```

To simplify parsing and make the header sections stand out in the source code, the opening and closing tags will have to be the only non-whitespace content on their respective lines in order to be recognized as header sections.

The body of a header section will contain one or more *header commands*. Again, to simplify the structure of header sections, each header command will be placed on a separate line and will have a fixed structure of a command word followed by an optional argument. Empty lines or lines beginning with double slashes will be ignored. For example, a header section could look like this:

```
/*HEADER**

    // Command without parameter
    PAUSE

    // Commands with parameter
    OUT Hello world.exe
    REF System.Core.dll

**HEADER*/
```

Available commands

We will need to offer header commands for most of the options that can be passed to the C# compiler. This includes options such as referenced assemblies, embedded resources, output assembly name and type, etc. We will also offer a command to pass any additional options to the compiler as a fallback in case some required option is not available in the form of a header command.

Because of requirement R3.4.1 (one script will be able to include another), we will also need a command to include other source code files into the resulting assembly. In this case, we will also recursively scan the included files for their own header sections, which may for example contain references or resources required by the included file.

Because of requirement R3.4.2 (one script will be able to reference another), the command for referencing assemblies will also be able to reference another script file. In this case, the referenced script file will be recursively compiled first and the resulting assembly will be used as the reference instead.

File parameters

Many header commands will expect a path to a file as a parameter. In order to make this parameter as versatile as possible, we will accept both absolute paths, or paths relative to the directory with the current source code file.

Most commands that expect a path to a file as a parameter can be applied multiple times to different files. To simplify this, we will introduce wildcards. Paths will be able to contain a wildcard character `*` in their file name. In this case, all the files found in the target directory that match the wildcard will be passed to the command. The wildcard character `*` will be also usable as the name of the last directory in the command's parameter. In this case, all subdirectories will be also recursively searched for the specified file. For example, the following command will include all files with the `csx` extension in the `Common` directory and also all its subdirectories recursively:

```
INC Common\*\*.csx
```

Sometimes it is useful to use a wildcard to include a lot of files except a select few. For this purpose, a hyphen before the parameter will indicate that the file should be excluded. For example, the following commands will include all files with the `csx` extension in the `Common` directory, except `other.csx`:

```
INC Common\*.csx
INC - Common\other.csx
```

Application-specific commands

When debugging a script, it can be useful to specify what working directory it should be run from or with what command-line arguments. Since such information is tied to individual scripts, it makes sense to include it somehow into the script files themselves. Therefore, we will also support header commands that do not control the build process itself, but instead provide information that is passed to the application, which can then use it to control how the script is launched. Such header commands will only affect scripts launched from the application itself and not scripts launched externally via the generated executable.

3.4.2 Builder

The process of compiling script files into executable binaries based on the information contained within the script files in the form of header commands is special, because it may be useful to use it outside the application itself in order to build script files externally, for example as a part of a batch script. In such cases it would be preferable to invoke just some kind of a compact command-line utility instead of using the full application. Because of this, will implement the build process in a separate library, which we will call *Builder*. This library will provide an interface that the application will call to compile script files.

Project structure

Since script files will be able to include or reference one another, we will need to handle this carefully with respect to all functionality requiring on-the-fly analysis of the source code. This means that we will need to maintain what we call the *project structure* of the currently open scripts:

- A *project* will be a single stand-alone compilation unit that consists of one or more source code files, their references, and its compilation results into a single output assembly.
- The *project structure* will then be information about what projects are open within the application, what are their external references, and how they reference one another. This information will be essential later when we implement code-aware functionality.

Since the project structure depends solely on header sections within the script files which will be parsed by the Builder library, its interface will need a means to report the project structure back to the application.

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. In order to provide error highlighting, the interface of the Builder library will need to be able to report errors back to the application so that they could be presented to the user.

Real-time information

The code-aware functionality of the application, and thus the on-the-fly analysis of the source code, will need to be done in real time and reflect the current state of the source code, which, as the user modifies it, will very often differ from the state saved on the filesystem. Such local unsaved modifications can also affect the project structure because the user can modify header commands as well. Therefore, the Builder library will need to:

- Provide a way to invoke it and supply replacement content for script files that are currently open and modified within the application.
- Provide a way to invoke it in a limited "parse" mode, which will only scan the source code files for header commands and return the project structure information. In this mode, there will be no need to invoke the compiler itself.
- Provide a way to invoke it in a limited "check" mode, which will invoke the compiler, but only in order to report any compilation errors and without actually generating the output assemblies.

Code completion

It would be convenient if code completion, which will primarily work for the C# source code itself, also worked in header sections and header commands. This would simplify usage of header commands, which could then be offered automatically and users would not need to remember them or look them up. Also, completion of various arguments of header commands would be useful, for example when specifying file paths. Therefore, the Builder library will provide interface to code completion of header commands.

3.5 NRefactory

NRefactory [6] is a C# library that allows parsing and semantic analysis of the C# source code. It also contains support for refactoring, formatting, code issue analysis and code completion. Our application will make heavy use of the NRefactory library to implement its code-aware functions.

3.5.1 Overview

NRefactory is introduced in depth in *Using NRefactory for analyzing C# code* [7]. It differs from the other libraries used in our application in the fact that it is not easy to understand and work with. Therefore, here we will outline how it is used to analyze C# source code. We will explain only its most basic principles and nomenclature so that we could later describe how it will be integrated into our application.

The basic steps of syntactic and semantic analysis of the source code are shown in figure 3.2. When C# source code needs to be analyzed, the NRefactory workflow looks like this:

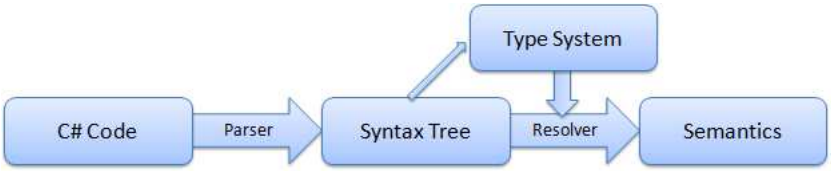


Figure 3.2: A simplified flow of source code analysis in NRefactory (from [7]).

1) A parser is invoked on the C# source code file. The result is an *abstract syntax tree*, or AST, representing the syntactic structure of the original file. The AST also contains additional information about comments, whitespace and exact position of individual syntactic elements within the original file, so that refactoring transformations can alter the AST and render it back into source code text with minimum changes. The AST is represented by the `SyntaxTree` class and its nodes by the `AstNode` abstract base class.

2) The AST of a single C# source code file is then converted into an *unresolved type system*. This process analyzes the AST and extracts information about the types and members contained within the file. The type system is called "unresolved" because it only contains basic information about the types and

members as they appear in the source code itself, without being put into context of neighbouring types, other source files, or referenced assemblies. The unresolved type system is represented by a complex hierarchy of classes (similar to the hierarchy shown in figure 3.4). The output of converting an AST of a single C# source file into an unresolved type system is represented by the `IUnresolvedFile` interface implemented by the `CSharpUnresolvedFile` class.

It is rarely needed to analyze a single C# source file on its own. The source file is almost always a part of a larger project that contains other source files, references to extrenal assemblies, or references to other projects. How this is handled by NRefactory is shown in figure 3.3. The workflow continues like this:

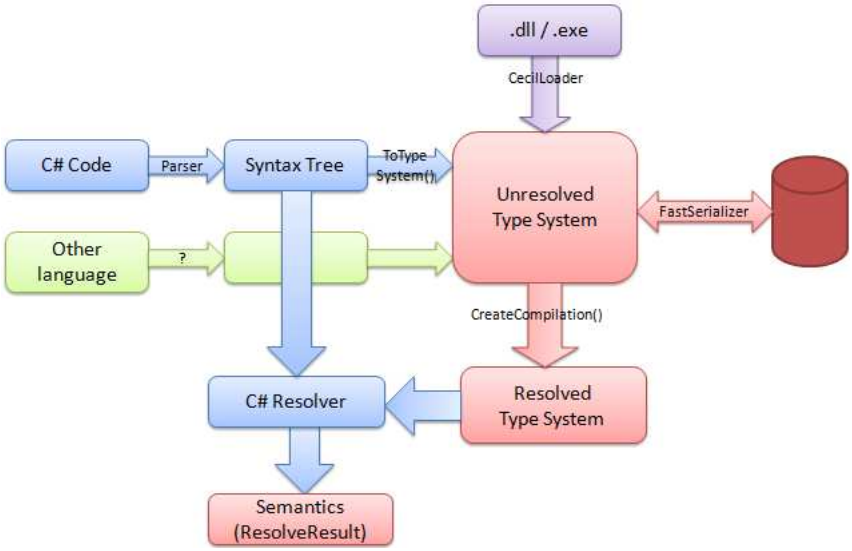


Figure 3.3: A detailed flow of source code analysis in NRefactory (from [7]).

3) We need to load information from the referenced assemblies. For this purpose the NRefactory library contains the `CecilLoader` class that uses the Cecil library [29], which is a dependency of NRefactory, to read an assembly file and create an unresolved type system representing the types and members within the loaded assembly. The result is represented by the `IUnresolvedAssembly` interface.

4) Now we need to put all the source files (`IUnresolvedFile`) and referenced assemblies (`IUnresolvedAssembly`) together so that they would represent a single project. This is done by the `CSharpProjectContent` class implementing `IProjectContent`. This class acts as a container for source files and references and represents the resulting assembly. It also implements `IUnresolvedAssembly`, so other instances of `CSharpProjectContent` can be also referenced if we have multiple projects that reference one another.

5) When we have put together an `IProjectContent` instance representing the project within which we want to analyze the source code, we call its `CreateCompilation` method. This method creates an `ICompilation` instance which represents the project as a *resolved type system*. This is a type system

that takes into account all contextual information, such as assembly references and relations that span across multiple members, types and files. The resolved type system forms a hierarchy reminiscent of the `System.Reflection` API. This hierarchy is shown in figure 3.4.

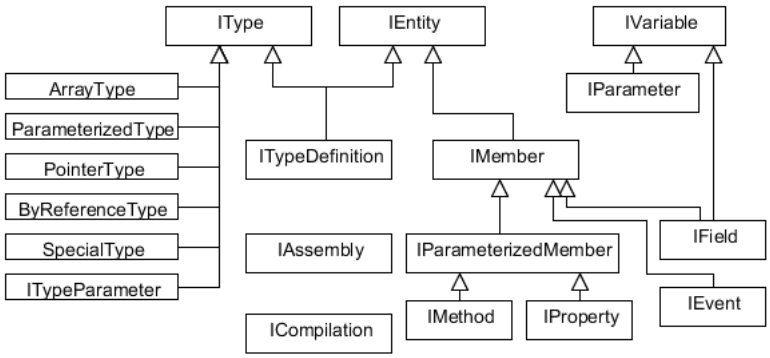


Figure 3.4: A resolved type system class hierarchy in NRefactory (from [7]).

Now that we have the `ICompilation` instance, which represents full information about the context of the C# source code files contained within the project, we can finally analyze their semantics. For this purpose we create a `CSharpAstResolver` instance, which takes the `ICompilation` instance (the context) and the `SyntaxTree` instance (the parsed source file to be analyzed). Then we can call its `Resolve` method with any node of the syntax tree as a parameter in order to find out its semantic meaning.

The semantic meaning is returned in form of a `ResolveResult` instance. For example, a single identifier (a node of the type `IdentifierExpression`) can be resolved to a local variable (`LocalResolveResult`), property (`MemberResolveResult`), type (`TypeResolveResult`) etc. Sometimes it may even mean nothing (`UnknownIdentifierResolveResult`) or several things at once (`AmbiguousTypeResolveResult`). Individual subclasses of `ResolveResult` carry the information about the entities that were identified by the resolver. The resolving process can be seen in figure 3.3 as well.

The `CSharpAstResolver` class also allows retrieving the state of the resolver at a certain position within the source file. The state is represented by a `CSharpResolver` instance and can be queried for a lot of interesting information, e.g. the current type and member declaration, the current namespace and using scopes, or name lookups.

It should also be noted that most classes in NRefactory are immutable (or at least freezable), which means that they can be safely accessed from multiple threads. This is quite surprising in case of the `CSharpProjectContent` class that acts as a container for C# source files and references. Each time a `CSharpProjectContent` needs to be modified (this happens often because it represents a project that continually evolves as the source code is edited by the user), a new instance containing the changes is created and the original instance is left unchanged.

3.5.2 Core

There are a lot of features in the application that will need the analysis provided by NRefactory (see requirements R2 in section 2.2.2). However, if every function parsed and analyzed the source code on its own, the application would be unnecessarily slow and resource intensive. Because of this, a central way of managing source code parsing and analysis is needed. We need a system that would monitor changes to the source code and keep its representation provided by NRefactory up to date, so that if any function asked for the data from NRefactory (e.g. an AST, a compilation, a resolver, etc.), it would get it right away.

The major responsibilities of this system, which we will call the NRefactory core, will be:

- Monitor the project structure of C# scripts that are open in the application.
- Keep an up-to-date `IProjectContent` instance for each project.
- Load external assemblies and source code files.
- Reload external assemblies and source code files if they change.
- Whenever a source code file changes (either externally or locally within the application), reparse it.
- Upon request, return the current `SyntaxTree` for a source code file
- Upon request, return the current `ICompilation` for a project.

Volatile resources

The syntactic and semantic representation of the source code in our application, as provided by the NRefactory library, depends on three kinds of resources:

- A source code file.
- An external assembly.
- A project assembly (an assembly representing a script built by the application itself).

Such resources have an important property – they change over time. A source code file can be edited. An external assembly can be replaced. A project assembly can be recompiled. Therefore, we will introduce a concept we shall call a *volatile resource*. A volatile resource will be a resource that changes over time and allows us to retrieve its current value and check it for updates. Thus, we will need two operations upon each volatile resource:

- A method to retrieve the resource's current state (value).
- A method to check if the resource's state (value) we retrieved is still up to date.

From the point of view of the NRefactory library, a project will consist of a set of volatile resources representing its source code files, its external references to other assemblies, and its internal references to other projects. Whenever we detect a change in any of these resources, we will know that any NRefactory structures we have parsed and analyzed for the project are out of date and need to be produced again. This will also mean that the project itself has changed, which is important because the project itself can serve as a volatile resource of other projects that reference it.

Resource loaders

Now that we have defined the concept of volatile resources, we will need a means to create them and check if they are up to date. Therefore, we will introduce *resource loaders*, which will be the central points for acquiring volatile resources for given source code files or assembly files. The loader for external assemblies will have many responsibilities:

- Because external assembly files may change, we need to check them for any changes. There are two ways this could be done – use a `FileSystemWatcher` class from the .NET framework which monitors changes to files and directories using the means of the operating system, or simply check the file's last write time.

  Since we do not expect external assembly files to change much (if at all), we will use the second approach, which is simpler and does not use operating system resources. Moreover, we will only explicitly check for update when the application returns into foreground from being inactive because changes to external files are expected to happen mostly when the user is away from the application.

- There can be XML documentation associated with the external assembly, which the NRefactory library allows to be loaded as well and associated with the assembly. Therefore, the loader will need to check for the presence of the XML documentation alongside the loaded assemblies and load it if present.

- Different projects can request the same assembly. Therefore, to avoid having to load an external assembly multiple times and thus to reduce resource usage, we will keep a weakly referenced cache of the loaded assemblies so that a single assembly is not loaded twice unnecessarily. The cache will have to be implemented in such a way that if an assembly is asked to be loaded while another thread is already in the process of loading the same assembly, we wait for it to finish and use its result.

The loader for source code files will have other responsibilities:

- Because source code files may also change externally, we need to check them for changes. We will do this in a similar way to external assemblies.

- Source code files can be opened in the application, modified by the user, and not yet saved. In this case, the user will expect all code-aware functionality of the application to reflect this current (modified) state of the source

code, as opposed to the state stored on the disk. Therefore, we will need to treat this as a change of the source code file resource and use this modified content of the source code file for NRefactory processing instead of its original content on the filesystem. This way, internal source code modifications will be included seamlessly into the version checking mechanism of volatile resources.

Project representation

One of the most important responsibilities of the NRefactory core subsystem will be monitoring the structure of the C# scripts that are open in the application and keeping an up-to-date `IProjectContent` instance for each project. A project will be identified by its path and will consist of paths to its source code files, referenced assemblies, and other referenced projects. In order to keep the `IProjectContent` instance up to date, each project also needs to have:

- For each source file its current parsed representation (`IUnresolvedFile`).
- For each external assembly reference its current representation loaded into the application (`IUnresolvedAssembly`).
- For each internal project reference its current representation (the instance of `IProjectContent` managed by the referenced project).

As the structure of a single project changes over time, modified source code files need to be parsed, external assemblies need to be loaded, and the project's `IProjectContent` needs to updated to include the current representations of these resources.

This is where we will make heavy use of the volatile resource concept described above. Each of the resources that make up a project (source code file, external assembly, another referenced project) can change over time – source code files are modified regularly, external assemblies can potentially change, and other referenced projects change every time their own source files and references are modified. Because of this, every resource that makes up a project will be implemented as a volatile resource, and so allow us to retrieve its current representation and check it for modifications. This way, when a project is queried for its NRefactory representation, we will know if we can return cached data or we must create it anew.

3.5.3 Code completion

While the user types identifier names, code completion offers a list of available types, members, variables and other constructs. It allows to choose an identifier without the need to remember its exact name or without having to type out its name in full, because names are traditionally long and descriptive in the C# language. It is also often used for quick overview of available types and their members, and it can also display snippets from the documentation. The requirements for code completion in the application (R2.2) are specified in section 2.2.2.

User interface

The editor control that is used in the application, AvalonEdit, contains basic implementation of a completion window. When we want to display it, we need to pass it the following information:

- The text segment (offset and length) of the area where the code completion happens.
  - The window is displayed beneath this area.
  - The window is automatically closed when the caret leaves this area.
  - The window filters its content by the text from the beginning of this area to the current caret position.
- A list of items to offer in the completion window. Each item has:
  - The icon that is displayed in the first column of the completion list.
  - The content of the second column of the completion list.
  - The string that is used to filter the items in the completion list.
  - The content of a tooltip that is displayed when the item is highlighted.
  - The method that is invoked when the item is selected. This method usually replaces the text in the completion segment with another text.

The example of such a completion window can be seen in figure 3.5. The text segment for completion is the word "int", its part "in" is used to filter the completion items, tooltip is shown. When any item is selected, its content replaces the word "int".

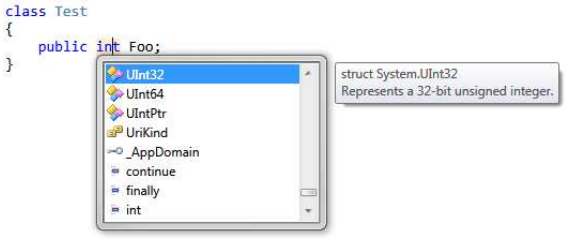


Figure 3.5: An example of the code completion window.

We will use the user interface implementation of the code completion window in AvalonEdit as the basis for our own code completion. However, we will need to extend it:

- In order to support extensibility (R1.7), we will create an extensible infrastructure for code completion data sources, so that items in the code completion list could come from plugins exported via MEF.
- According to requirements R2.2.3 and R2.2.4, code completion will have to offer types and extension methods from namespaces that are not yet imported via the `using` statement and from assemblies that are not yet referenced, and automatically add the correct `using` statement or the correct

reference accordingly upon selection. Instead of combining all the suggestions into one long list, we will instead introduce three different levels of code completion:

1) Basic completion – Offers all known symbols that can be used at the current position.
2) Import completion – Offers types and extension methods from namespaces that are not currently imported with a `using` statement and imports the namespaces accordingly when an item is selected.
3) Reference completion – Offers types, extension methods and namespaces from yet unreferenced but known assemblies, and automatically references the correct assembly when an item is selected.

The first level will be the default. Users will be able to switch to the next level by invoking the code completion again using the keyboard shortcut, which is traditionally `Ctrl+Space`.

Therefore, we will have to add support for multiple levels of code completion. If users do not find what they were looking for in the code completion list, they will be able to invoke the code completion window again and it will display a different set of items from the next level.

- To satisfy requirement R2.2.2, the code completion window will have to support a special mode so that if the user introduces a new identifier instead of selecting an item from the list, the code completion will not interfere by inserting the item's text instead.

  We will do this by introducing the concept of *active selection* and *inactive selection*. The selected (highlighted) item in the code completion list can be either active or inactive. If a character that cannot be a part of an identifier is typed (e.g. a space, comma, etc.) and the current selection is active, the selected item gets inserted. On the other hand, if the current selection is inactive, no insertion takes place.[1] The visual distinction of these two cases can be seen in figure 3.6. Optionally, the inactive selection can become active after the first character that *can* be a part of an identifier is typed.[2]

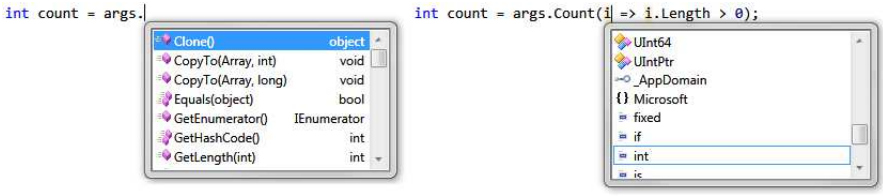


Figure 3.6: An example of active and inactive code completion window.

[1] Inactive selection is used for example in cases where both a known and a new identifier can be used, e.g. lambda method parameters in C# (see figure 3.6).

[2] This is used for example when the user typed the `new` keyword, after which a type name can follow, or a square bracket (array construction), or a curly bracket (anonymous object construction). If the user types a bracket, no completion takes place. If the user types a letter, the selection becomes active and normal completion takes place.

- We will also improve the user interface of the completion window by adding support for additional content in the right column, which can be used for displaying extra information, for example to indicate the return type of methods, as can be seen in figure 3.6. Because of this, we will also add support for automatic widening of the completion window, so that if the completion window is too narrow to fit the content of its visible items, it will be automatically widened.

Data source

Deciding what exactly to offer in the completion window at a given location in the source code is a complex task that requires syntactic and semantic analysis of the source code both preceding and following the location. Fortunately, the NRefactory library contains good support for code completion, which we will integrate with our application. We will also need a separate data source for the higher levels of code completion (import and reference completion), which means a list of all potential types in all potential assemblies.

Code completion mode

The requirements R2.2.1 and R2.2.2 require code completion to:

- Appear automatically when the user types an identifier.
- Do not appear in places where the identifier denotes a name of a new symbol instead of a reference to a previously known symbol.
- Appear in a different mode in places where it is syntactically possible both to introduce a new symbol or to use an existing one (such as when passing a lambda delegate as a real parameter). This different mode is the inactive selection mode described above.

Therefore, when the user starts typing an identifier, we must first look at the position in the source code where the typed identifier is located and decide if a new symbol can be introduced there or not:

- If a new symbol cannot be introduced at the location of the identifier, we open code completion in normal (active selection) mode.
- If a new symbol can be introduced at the location of the identifier, there are two further scenarios:
  - A known symbol can be referenced at the location of the identifier. Therefore, we open code completion in inactive selection mode.
  - No known symbol can be referenced at the location of the identifier. Therefore, there is no need to open code completion at all.

The decision process will be based on the same tools that the code completion engine from the NRefactory library uses to determine what items to offer in the completion list.

4. Implementation

The source code of the implemented application can be found on the enclosed CD in the `src` directory. To open it in Visual Studio, at least Visual Studio 2010 and .NET Framework 4.0 or newer are required. Documentation generated from the the source code is also available on the CD, in the `doc` directory.

Libraries

The application is implemented with the help of a number of external libraries. During the development of the application, it was sometimes necessary to modify some of these libraries. The changes were only minor (e.g. build configuration was tweaked or additional members were exposed), but even so, we include their full modified source code as well. Therefore, there are two subdirectories in the `src` directory:

- `ExtBrain.ScriptDevelop` – The source code of the application itself.
- `Libraries` – The source code of the libraries that the application uses.

The `readme.txt` file in the `Libraries` directory lists the modified libraries and for each library it contains the following information:

- The URL and revision of the library's Subversion repository, from which the original source code was downloaded.
- A summary of custom modifications to the library.

For each library there is also a `.patch` file generated by the Subversion client which describes what exactly has been modified in the source code of the library.

The libraries are set up to output the resulting binaries to the `ExtBrain.ScriptDevelop/Externals` directory, from which the application references them. This directory also contains two other external libraries which there was no need to modify, so only their binaries are included:

- `log4net.dll` (version 1.2.13) – A helper library for logging.
- `JetBrains.Annotations.dll` (version 6.1.37.86) – Helper attributes for the Resharper tool (which is not required to open or build the source code).

Previous work

As stated in section 3.2, we used ScriptDevelop [11] as the starting point for our application. However, we have modified it heavily to meet our goals, and very little is left of the original source code. For reference, the original source code of ScriptDevelop is included on the enclosed CD in the `previous` directory.

4.1 Overview

This section provides a high-level overview of the application's implementation. It details what components the application is composed of, what is their function and how they cooperate and depend on each other.

4.1.1 Components

The application is separated into several primary components, each of which is focused on a different functional aspect of the application. Such primary components are large portions of the application, containing many classes, plugins, or even assemblies. Each of these components will be further described in this section.

There are six primary components of the application – *Application Core*, *UI Core*, *Editor*, *Compiler*, *NRefactory*, and *Debugger*. Each of these components uses services provided by the previous ones. The six primary components can be seen in figure 4.1 as the central column.

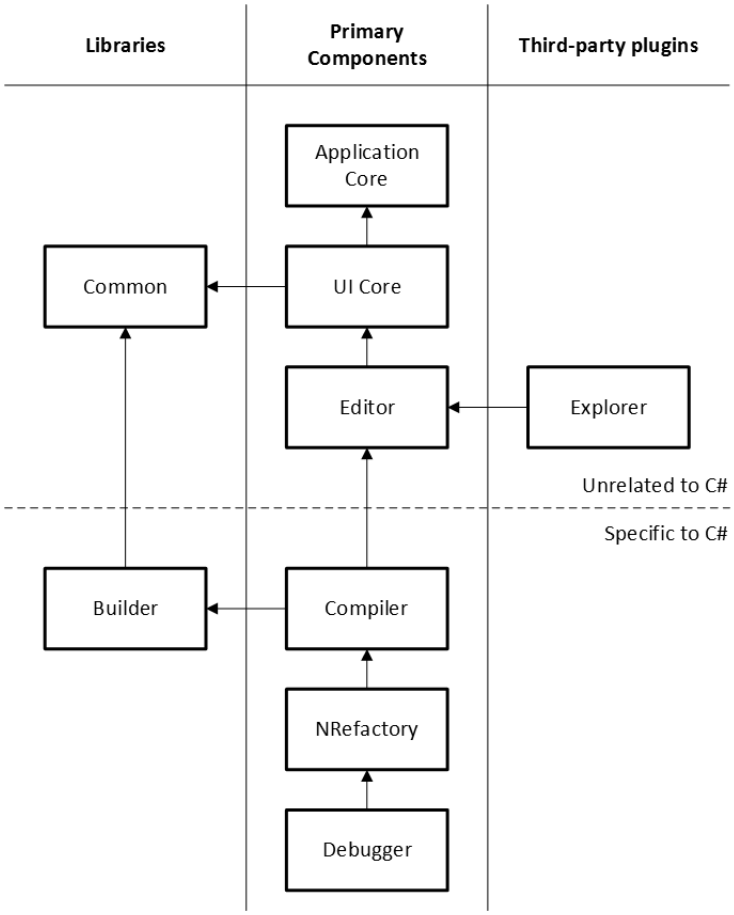


Figure 4.1: Overall structure of the application.

The purpose of the primary components is as follows:

- *Application Core* contains the main entry point of the application and offers the basic supporting services such as plugin loading, configuration or logging. It does not reference any other components – the rest of the application is loaded dynamically at runtime.
- *UI Core* contains the main window of the application and implements basic user interface – a single window with menu, toolbar, status bar, dockable panels and document tabs. This interface is universal, is not tied to any specific application type, and could be used for any other application with a similar layout.

- *Editor* provides the concept of documents, their formats and editors. It manages currently open documents and their lifecycle. It also defines a text document and its editor and offers a lot of services for them, for example highlighting, tooltips, context menus, additional margin panels, or location tracking.

- *Compiler* defines the C# document format and basic functionality that does not require code-aware insight into the source code. It contains commands to build and run scripts. It also monitors the currently open scripts and runs the build process in order to display errors and to keep the project structure information up to date.

- *NRefactory*[1] is the largest component of the application. It encompasses all code-aware features of our C# source code editor that require on-the-fly syntactic and semantic analysis of the source code, such as syntax highlighting, code completion, code issue analysis, or navigation.

- *Debugger* implements the basic debugging functionality. It defines commands for starting, pausing and stopping the debugger, for stepping through the debugged code, or for breakpoint management. It also contains panels that display call stack, exception information, or command console.

The components that stand apart from the central column in figure 4.1 have special purpose:

- *Common* is a utility library with shared code that all components use. It is a place for helper classes and extension methods that are not specific to any single part of the application.

- *Builder* is a library that the Compiler component calls to actually compile C# scripts into binary assemblies. It parses the header sections in the source code and runs the actual C# compiler accordingly. It does not depend on the rest of the application and it can be used on its own, for example it could be easily made into a separate command-line tool for building C# scripts.

- *Explorer* is an example of a third-party plugin that extends the application with additional functionality (in this case a filesystem tree explorer). It was part of the original ScriptDevelop project (see section 3.2) and it was left mostly intact during the development of our application, only with minimal changes to ensure its compatibility.

The first three primary components (Application Core, UI Core, Editor) are unrelated to C#. Application Core on its own could be used for any WPF application and provide it with the supporting infrastructure (plugins, configuration, logging). Application Core and UI Core together make up a generic WPF application with a basic user interface consisting of main menu, toolbar, status bar, dockable panels, and document tabs. Editor adds the ability to open and edit text

[1] Please note that this component is named after the underlying external library it uses, which is NRefactory 5.

files. With only these three components the application is not tied to C# in any way. The other three components (Compiler, NRefactory, Debugger) gradually extend the application into a C# IDE.

Every component of the application consists of one or more projects in the source code of the application, as described in table 4.1.

Component	**Project**
Application Core	`ExtBrain.ScriptDevelop.ApplicationCore`
UI Core	`ExtBrain.ScriptDevelop.Plugin.MainWindow`
Editor	`ExtBrain.ScriptDevelop.Plugin.Editor` `ExtBrain.ScriptDevelop.Plugin.Editor.Text`
Compiler	`ExtBrain.ScriptDevelop.Plugin.CSharp` `ExtBrain.ScriptDevelop.Plugin.Compiler`

NRefactory	`ExtBrain.ScriptDevelop.Plugin.NRefactory.Core`
	`ExtBrain.ScriptDevelop.Plugin.NRefactory.Import`
	`ExtBrain.ScriptDevelop.Plugin.NRefactory.CodeCompletion`
	`ExtBrain.ScriptDevelop.Plugin.NRefactory.Analysis`
	`ExtBrain.ScriptDevelop.Plugin.NRefactory.Analysis.Import`
	`ExtBrain.ScriptDevelop.Plugin.NRefactory.Analysis.Adapter`
	`ExtBrain.ScriptDevelop.Plugin.NRefactory.Navigation`
	`ExtBrain.ScriptDevelop.Plugin.NRefactory`

Debugger	`ExtBrain.ScriptDevelop.Plugin.Debugger`
Common	`ExtBrain.ScriptDevelop.Common`
Builder	`ExtBrain.ScriptDevelop.Builder`
Explorer	`ExtBrain.ScriptDevelop.Plugin.Explorer`

Table 4.1: Components and their corresponding projects.

The interaction between the components of the application is shown in figure 4.2 and could be summarized like this:

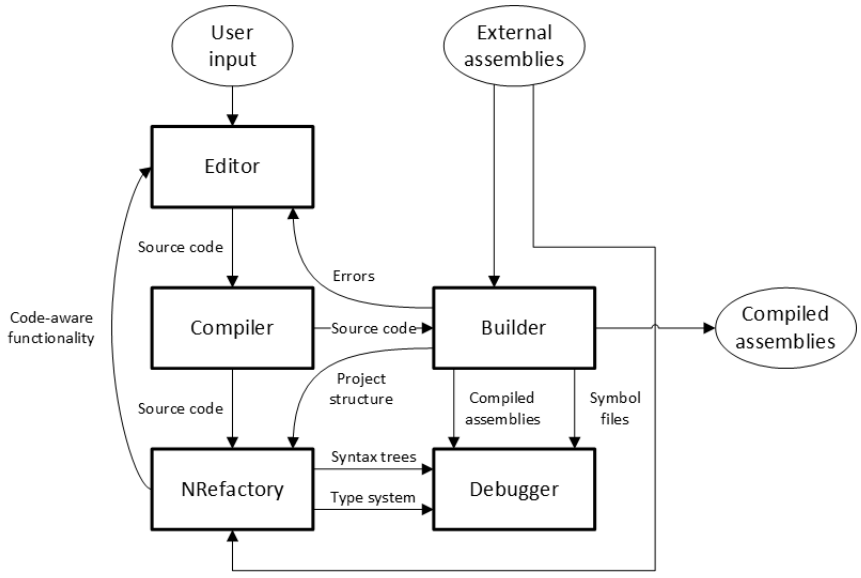


Figure 4.2: Interaction between the components of the application.

- The user types the source code in the Editor.
- The source code is processed by the Compiler, which invokes the Builder.
- The Builder parses the header sections in the source code, determines how to call the actual C# compiler, and calls it.
- If there were errors, the Builder returns their list to the Compiler, which presents it to the user and highlights the errors in the Editor.
- If there were no errors, a binary assembly is created, optionally with a symbol file for debugging.
- The Builder returns the information it parsed from the header sections in the source code back to the Compiler.
- The Compiler uses this information the keep the project structure up to date. The project structure is information about what source files make up what assemblies and how the assemblies reference each other.
- The Compiler invokes this process either in full upon a manual request from the user, or only partially whenever there has been a change in the source code in order to reparse the header sections and update the project structure.
- NRefactory processes the project structure information, parses the source code into syntax trees, loads external referenced assemblies, and builds a type system representing the source code. Syntax trees and the type system are updated whenever there has been a change in the source code.
- Subcomponents of the NRefactory component consume the syntax trees and the type system information in order to provide various code-aware functionality to the Editor.
- The Debugger utilizes the syntax trees and the type system information for evaluation and state inspection.

4.1.2 Plugins

The application is implemented as a set of many plugins that use one another's services. The main executable project, `ExtBrain.ScriptDevelop.ApplicationCore`, contains only the basic infrastructure for application startup, plugin management, and a few supporting services like configuration and logging. Without any additional plugins to load, the application would start, but it would exit immediately because there would not even be any window to be displayed. Every separate function or feature of the application, no matter how small, is implemented as a plugin and could be easily separated into an assembly of its own[2] and loaded later, or not loaded at all.

A lot of plugins can be themselves extended by other plugins. For example, the plugin that defines the main window of the application can be extended by plugins

[2]For practical reasons, a single assembly of the application usually contains many plugins.

defining individual components of the main window (main menu, toolbar, status bar). The status bar plugin can be further extended by a plugin displaying current text cursor position or the status icon. Some plugins can even extend multiple other plugins. For example, every command in the application is a plugin that can have its representation in the main menu, in the toolbar, and in the keyboard shortcut manager. An example of such a plugin structure is shown in figure 4.3.

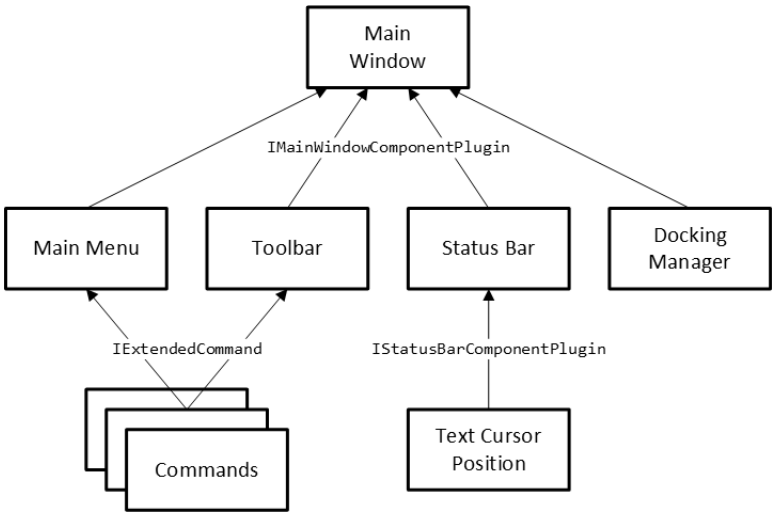


Figure 4.3: An example of a more complex plugin structure.

There is also a special kind of plugins called *extension plugins*. An extension plugin attaches an extension to every instance of its target type to modify its behavior or to provide additional services. Extension plugins are heavily used throughout the application to attach behavior to individual open documents and to extend highlighted text segments with additional functionality.

4.2 Application Core

Application Core is the main entry point of the application – it contains the `Program.Main` method and is reponsible for initialization of the entire application. It does not reference any other assemblies of the application, but instead loads the rest of the application dynamically at runtime. Therefore, the primary part of this component is the plugin infrastructure. The component also provides supporting services: logging, configuration, settings, and temporary storage.

4.2.1 Plugin infrastructure

The plugin infrastructure of the application is built on top of MEF, which is used to manage discovery and instantiation of individual plugins through its mechanism of exports and imports. In addition to this the `ExtBrain.ScriptDevelop.ApplicationCore` project contains classes and interfaces to further support plugin management in the `ExtBrain.ScriptDevelop.ApplicationCore.Infrastructure` namespace, which is illustrated in figure 4.4 and described in this section.

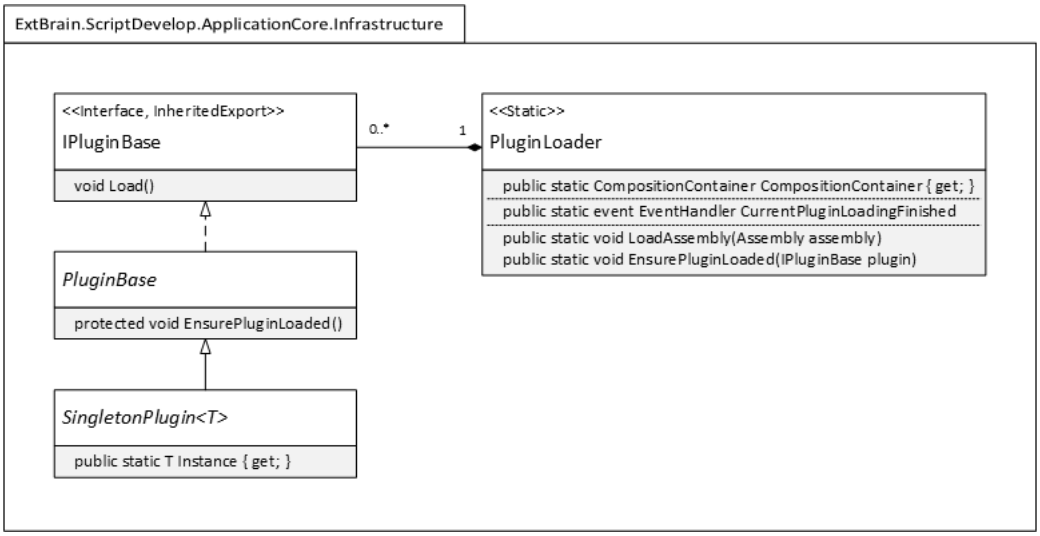


Figure 4.4: Plugin infrastructure.

Plugin loading

The loading of plugins into the application is handled by the `PluginLoader` static class using the method:

```
public static void LoadAssembly(Assembly assembly)
```

The `PluginLoader` class maintains a MEF catalog and a composition container. This method adds the given assembly into the catalog, updates the composition container, and initializes the newly added plugins. It also recursively loads and adds any assembly that was referenced by the original assembly and also contains plugins. This is to ensure that if a plugin references another plugin from a different assembly, the referenced plugin is loaded along with the referencing plugin.

Plugin initialization

A plugin that needs to perform some kind of initialization upon startup can implement the following interface:

```
[InheritedExport]
public interface IPluginBase
{
    void Load();
}
```

Any plugin that implements this interface will have its `Load` method called when it is loaded into the application. The `Load` method is guaranteed to run on the UI thread, so its implementation can safely access UI elements. However, plugins that are exported to MEF using other contracts do not necessarily need to implement this interface.

Singleton plugins

The most common type of plugin in the application is a singleton plugin, i.e. a plugin that exists as a single shared instance. To simplify accessing singleton

plugins in code and to make sure they are only accessed after they had been properly initialized, there is a special generic base class:

```
public abstract class SingletonPlugin<T> : PluginBase
{
    public static T Instance { get; }
}
```

This class allows accessing the singleton instance of the plugin of type `T`.

Extension plugins

The plugin infrastructure of the application also supports the concept of *extension plugins*. An extension plugin is a special kind of plugin that attaches an extension to an object. The object is then extended with additional functionality.

When an extensible object is created, extensions that extend this object are also created and attached to it. The extensions of the object can be accessed in code. When the lifetime of the object ends, the extensions that have been attached to it are detached and disposed. If a new extension plugin is loaded while the application is already running, the new extensions are created and attached to all extensible objects of the correct type that currently exist within the application.

The extension system consists of several interfaces[3] and classes, as shown in figure 4.5.

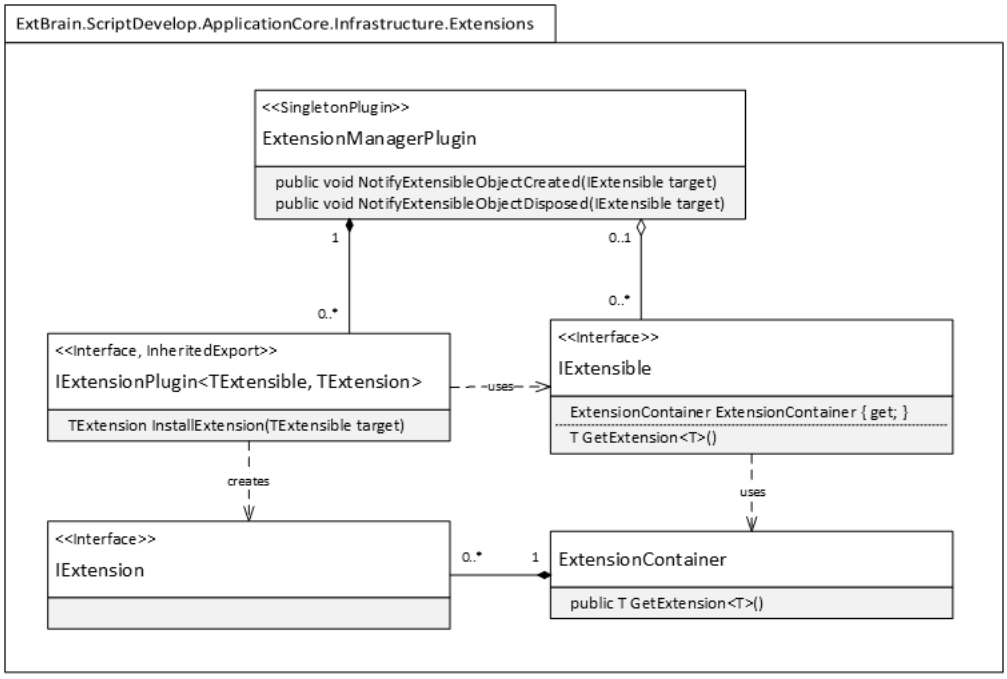


Figure 4.5: Extension infrastructure.

The `IExtension` interface marks a class the instances of which serve as extensions to an extensible object. It is only a marker interface, i.e. it contains no members.

[3]The actual definition of the interfaces is more complicated than described here for technical and type safety reasons, but the core princliple is still the same.

```
public interface IExtension
{
}
```

The `IExtensible` interface marks an extensible class, i.e. a class that can have extensions attached:

```
public interface IExtensible
{
    ExtensionContainer ExtensionContainer { get; }
    T GetExtension<T>() where T : IExtension;
}
```

The `GetExtension` method accesses the extension of type `T` associated with the `IExtensible` object. The `ExtensionContainer` property returns the extension container of the extensible object; each instance of `IExtensible` must provide its own `ExtensionContainer`, which is an internal class whose purpose is to keep track of extensions associated with a single extensible object.

The `IExtensionPlugin` interface marks a plugin that serves as a factory for extensions. It creates the corresponding `IExtension` for the given `IExtensible` object:

```
[InheritedExport]
public interface IExtensionPlugin<TExtensible, TExtension>
    where TExtensible : IExtensible
    where TExtension : IExtension
{
    TExtension InstallExtension(TExtensible target);
}
```

The lifecycle of extensions is managed by `ExtensionManagerPlugin`. This plugin exposes two methods:

```
public void NotifyExtensibleObjectCreated(IExtensible target)
public void NotifyExtensibleObjectDisposed(IExtensible target)
```

The first method is used to register a new extensible object. Upon registration, the `InstallExtension` factory method is called on all known extension plugins to create new instances of available extensions, which are then attached to the object. The second method is used to unregister the object when it is no longer alive. All extensions attached to the object are removed and the extensions that implement the `IDisposable` interface have their `Dispose` method called.

Sometimes the extensions depend on each other, i.e. the `InstallExtension` method assumes that the object being extended already contains another extension it can access using the `GetExtension` method. This situation is correctly handled by the extension manager – if `GetExtension` method is called for an extension that is not yet installed, but already in the installation queue, it is installed first and then returned to the `InstallExtension` method. This way the installation order of extensions is corrected automatically. When an extensible object is unregistered, its extensions are uninstalled in the reverse order than the order in which they were installed. This ensures that dependent extensions can access each other correctly even in their `Dispose` method.

4.2.2 Configuration and settings

Configuration and settings are one of the supporting services of the Application Core component. We described configuration and settings in section 3.3.3. Configuration allows the user to configure the behavior of various components of the application. It is user-defined and cannot be written to by the application. Settings allow various components of the application to persist their state between sessions. Users cannot change this state directly.

The classes that participate in configuration and settings management can be seen in figure 4.6.

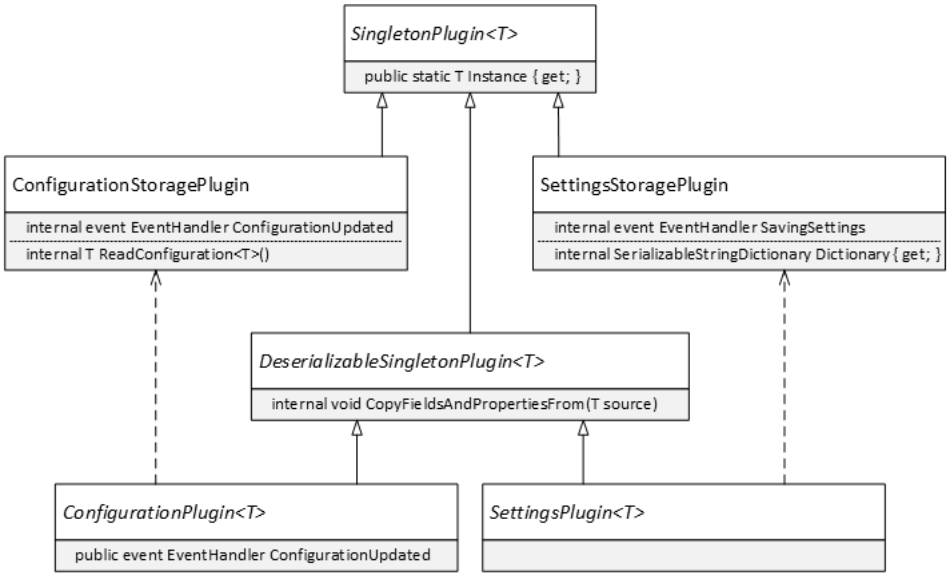


Figure 4.6: Configuration and settings infrastructure.

Configuration

The application contains no specialized configuration GUI. Instead, it is configured using an external XML file. This file can be edited directly in the application and whenever it is saved, it is reparsed and the components that can update their configuration on the fly are notified that an updated configuration is available.

The default configuration file is embedded inside the application and is used if no configuration file is found in the application's directory. If the user chooses to edit the configuration file using an option in the application, it is opened in the application's editor. If the file does not exist yet, it is created as a copy of the embedded configuration file.

The configuration file is formatted as an XML document whose root node contains one element for each separate configuration section. This is an example of a simple configuration file with two sections:

```
<Configuration>

    <KeyBindings>
        <KeyBinding Command="GoToTypeCommand" Shortcut="Ctrl+T" />
    </KeyBindings>
```

```
    <Logging>
        <Path>ScriptDevelop.txt</Path>
        <Filters>
            <Filter Prefix="*" Value="all" />
        </Filters>
    </Logging>

</Configuration>
```

In the code, each section of the configuration file is represented as a plugin derived from `ConfigurationPlugin`. When such a plugin is loaded, its public fields and properties are read from the appropriate section of the configuration file using the standard `XmlSerializer` of .NET. The name of the section by default equals to the name of the plugin's class, but can be defined separately using the `XmlRoot` attribute. If the corresponding section is not found in the configuration file or cannot be properly deserialized, the state of the plugin is left unchanged. Because of this, the constructor of the plugin should initialize it to a correct default state.

To illustrate this, the `KeyBindings` section from the above example is represented in the code like this:

```
[XmlRoot("KeyBindings")]
public sealed class KeyBindingConfiguration
    : ConfigurationPlugin<KeyBindingConfiguration>
{
    public class KeyBinding
    {
        [XmlAttribute]
        public string Command { get; set; }

        [XmlAttribute]
        public string Shortcut { get; set; }
    }

    [XmlElement(ElementName = "KeyBinding")]
    public List<KeyBinding> KeyBindingList { get; set; }

    public KeyBindingConfiguration()
    {
        KeyBindingList = new List<KeyBinding>();
    }
}
```

Since the `ConfigurationPlugin` class derives from `SingletonPlugin`, it can be easily accessed using the `Instance` static property, for example:

```
var keys = KeyBindingConfiguration.Instance.KeyBindingList;
```

The `ConfigurationPlugin` class also provides the `ConfigurationUpdated` event, which is fired every time the configuration file has been reloaded and the public fields and properties of the class have been updated with newly parsed values.

Settings

In a similar way to configuration, any plugin that derives from the `SettingsPlugin` class has its state automatically saved and restored. Saving is done at the application exit, restoring in the `SettingsPlugin.Load` method. All public fields and properties of the plugin are serialized using `XmlSerializer` and persisted using the application settings mechanism of .NET described in [30].

4.2.3 Logging

The logging infrastructure can be seen in figure 4.7. Logging is implemented as the `LoggerPlugin` singleton plugin, which contains `Get` methods for retrieving an `ILogger` instance for a given type. The `ILogger` interface contains methods for logging messages and exceptions of three different severity levels: `Debug`, `Warning`, and `Error`.

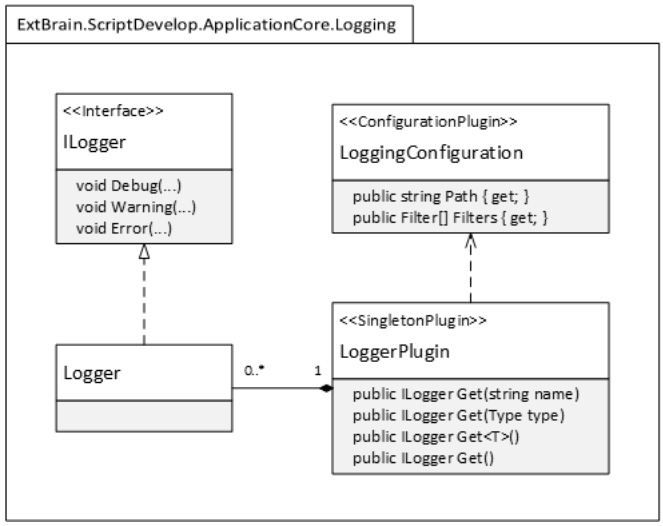


Figure 4.7: Logging infrastructure.

The logger writes messages into a text file whose path can be configured. Minimal severity levels for individual types (filters) can also be configured. Logging uses the *log4net* library [31].

4.2.4 Temporary storage

Temporary storage is described in section 3.3.2. It is implemented by the `TemporaryDirectoryPlugin`, which contains methods for accessing the shared or instance-specific temporary directories and for creating subdirectories inside them.

4.2.5 Application startup

When the application is launched, these steps are performed:

- `PluginLoader.LoadAssembly` is called on the executing assembly (`ExtBrain.ScriptDevelop.ApplicationCore`). This ensures that the plugins for the core infrastructure (described above) are initialized and ready.

- The `Loader.Initialize` method is initialized. The `Loader` class reads the configuration file to see what additional plugin assemblies are to be loaded and the `Initialize` method loads them using `PluginLoader.LoadAssembly`. Assemblies can be also configured to be loaded later, after a delay, which is also handled by the `Loader` class.

- The MEF composition container is queried for a plugin of type `IMainWindowPlugin`, and the window exposed by this plugin is displayed.

4.2.6 Stand-alone application executable

For easier manipulation and deployment, the application can package itself into a single stand-alone executable file that contains all the currently loaded plugins, the current configuration file, and all the necessary third-party libraries. The resulting single executable file can then be copied to any system which could run the original application and launched directly from the executable file, without any installation or unpacking process.

This functionality is implemented in the `Embedder` class, which contains methods to build the stand-alone executable, to query what assemblies are packaged into the current executable, and to load them.

The packaging process invokes the C# compiler in order to build the stand-alone executable. The resulting assembly contains only a short bootstrapping code that sets up the `AssemblyResolve` event of the application's `AppDomain` to look for assemblies among the resources embedded into the executable itself, and then calls the original `Program.Main` method to start the application. During the compilation, all the currently loaded and referenced assemblies are inserted into the resulting executable as embedded resource files, as well as the current configuration file, which replaces the default configuration file of the original application.

4.3 UI Core

UI Core is the primary user interface component of the application. It defines the main window and its structure – menu, toolbar, status bar, dockable panels and document tabs. It also manages commands, which are individual actions the user can perform by selecting them in the menu or toolbar or by using the respective keyboard shortcut. The UI Core component is implemented in the `ExtBrain.ScriptDevelop.Plugin.MainWindow` project and the structure of its plugins is shown in figure 4.8.

The user interface provided by this component is not tied to any specific application type and could be used for any other application with a similar layout.

4.3.1 Main window

The main window is composed and exposed by the `MainWindowPlugin` class. It imports plugins with the `IMainWindowComponentPlugin` interface. These plugins make up the main window's content. The window itself contains a single

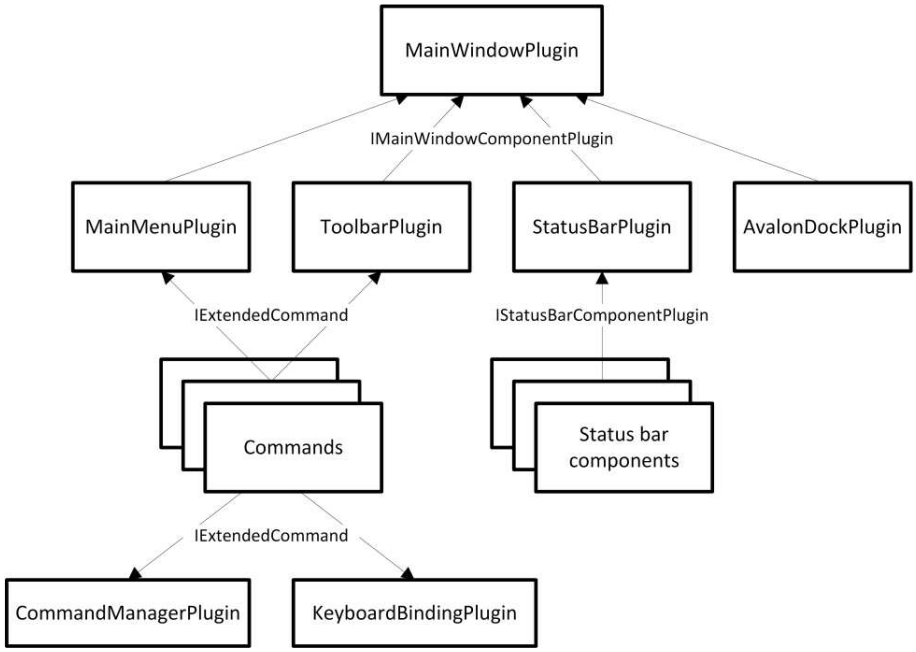


Figure 4.8: Plugin structure of the UI Core component.

`DockPanel` control that is populated with the controls exposed by the individual `IMainWindowComponentPlugin` plugins:

- `MainMenuPlugin` – Imports commands and creates the main menu composed of the items defined by the imported commands.
- `ToolbarPlugin` – Imports commands and creates the toolbar composed of the buttons defined by the imported commands.
- `StatusBarPlugin` – Imports `IStatusBarComponentPlugin` plugins and creates the status bar composed of the controls defined by the imported plugins. Also exposes methods to display textual status information.
- `AvalonDockPlugin` – Fills the central area of the window with a `DockingManager` control from the AvalonDock library and exposes it so that other components of the application can use it to display dockable panels and tabbed documents.

4.3.2 Commands

Commands are actions that the user can explicitly perform in the application. They can be represented by items in the main menu or by buttons in the toolbar. They can also have a keyboard shortcut assigned.

Definition

Individual commands are defined by classes exported via the `ExportCommand` attribute. These classes must implement the `IExtendedCommand` interface, which offers the following methods:

- `Execute` – Performs the action represented by the command.
- `CanExecute` – Returns if the command can be executed in the current state of the application. Commands that cannot are displayed as grayed-out in the menu and toolbar.
- `IsVisible` – Returns if the command is visible in the current state of the application. Commands that are not cannot be executed and are not displayed in the menu and toolbar. This is used for commands that make sense only in certain contexts (e.g. while debugging).
- `IsChecked` – Returns if the command is activated in the current state of the application. This only affects how the corresponding menu item and toolbar button are rendered. Activated commands are displayed with a check symbol or with a highlight. This is used for commands that toggle various options to offer visual feedback to the user.

The `ExportCommandAttribute` that is used to export individual commands also has several properties that define how the command is represented in the user interface:

- `Name` – The name that is displayed in the menu and toolbar.
- `Icon` – The icon that is displayed in the menu and toolbar.
- `Description` – The description that is displayed in the tooltip and status bar.
- `Shortcut` – The default keyboard shortcut that can be used to invoke the command.

Menu

In order for the command to have its representation in the menu, it must be annotated with the `MenuCommand` attribute, which is used to put the command to the correct place within the menu. The attribute has three properties:

- `Parent` – Indicates in which menu the command will be placed. It can be any class exported via the `ExportMenuCategory` attribute.
- `Group` – Indicates the name of the group in which the command will be placed. Different groups within the same parent are separated visually.
- `Order` – Indicates the order of items within the same group.

Additional entries not corresponding to any commands can be also placed in the menu. Such entries inherit from `FrameworkElement` and are exported via the `ExportMenuItem` attribute, which is used in the same way as the `MenuCommand` attribute.

Toolbar

In order for the command to have its representation in the toolbar, it must be annotated with the `ToolbarCommand` attribute, which is used to put the command to the correct place within the toolbar. The attribute has two properties:

- `Group` – Indicates the name of the group in which the command will be placed. Different groups are separated visually.
- `Order` – Indicates the order of items within the same group.

Additional entries not corresponding to any commands can be also placed in the toolbar. Such entries inherit from `FrameworkElement` and are exported via the `ExportToolbarItem` attribute, which is used in the same way as the `ToolbarCommand` attribute.

4.4 Editor

Editor is a primary component of the application that adds the ability to open and edit text files. The component is implemented in two projects:

- `ExtBrain.ScriptDevelop.Plugin.Editor` – Contains basic document management infrastructure. Provides the concept of documents, their formats and editors and it manages currently open documents.
- `ExtBrain.ScriptDevelop.Plugin.Editor.Text` – Contains functionality specific for text editors. Defines a text document and its editor and offers a lot of additional services for them, for example highlighting, tooltips, or context menus.

The user interface provided by this component is not tied to C# in any way and could be used as a generic text editor or as an editor for other programming languages as well.

4.4.1 Document infrastructure

The singleton `EditorPlugin` is the central point of the Editor component. It contains methods for creating, opening, saving and closing documents. It manages the list of currently open documents and keeps track of which document is currently active (focused) in the application.

Documents open in the application are represented by instances of the `IEditorDocument` interface. This interface provides a lot of read-only properties that expose various information about the document, for example its ID, name, path, format, version, and flags (e.g. if it is visible, active or modified). It also contains events that are invoked when some of this information is changed.

4.4.2 Extensibility

Documents in the application can be extended using the extension plugin mechanism described in section 4.2.1. The `IEditorDocument` interface inherits from `IExtension` and contains the `GetExtension` method that is used to access various extensions of the document.

There are two helper interfaces for exporting plugins that register extensions to specific document types:

- `IEditorExtensionPlugin<TFormat, TDocument, TExtension>`

  This will add an extension of type `TExtension` to all documents of type `TDocument` and format `TFormat`. The extension is created using the following factory method:

  ```
  TExtension InstallExtension(TDocument target);
  ```

  This method is called every time a document of the correct type has been added into the application. The extension returned by this method can then be accessed using the `GetExtension` method of the `IEditorDocument` interface. If `TExtension` implements `IDisposable`, its `Dispose` method is called when the document is closed.

- `IEditorExtensionPlugin<TFormat, TDocument>`

  This is a similar interface, but it is used if we only want to be notified when a document of type `TDocument` and format `TFormat` has been added into the application and we do not want to add any new extension to the document. Every time a document of the correct type is created, the following method is called:

  ```
  void InstallExtension(TDocument target);
  ```

This extension mechanism for documents is heavily used throughout the application to add functionality to individual open documents.

4.4.3 Highlights

The highlight mechanism is the most important extension of text documents. It allows highlighting certain segments of text and giving them a different look (e.g. altering its background color) or functionality (e.g. assigning it a tooltip text). This mechanism is used for example to highlight errors and issues in the edited source code.

The highlighting is provided by the `HighlightManager` document extension, which contains methods for creating, deleting and retrieving highlights. The position of highlights is automatically adjusted as the text of the document changes. If the text is changed in such a way that the highlight disappears (its length is reduced to zero), the highlight is removed from the document.

A single highlight is represented by the `IHighlight` interface, which contains properties for accessing the highlight's position, events that signal when the highlight's position is changed, and a method for deleting the highlight. The

`IHighlight` interface also inherits from the `IExtensible` interface, which means that highlights themselves can be also extended and contain the `GetExtension` method that is used to access their extensions.

Rendering

The most important function of highlights is that they can change the way the text is rendered. This is achieved using the `HighlightRenderingInfo` extension, which allows setting the highlight's renderer. The renderer is an instance of the `IHighlightRenderer` interface, which contains methods that alter the way the highlighted portion of the text is rendered. There are several `IHighlightRenderer` implementations:

- `SimpleHighlightRenderer` – Allows simple overriding of the text's color or drawing a rectangle around the highlighted text.
- `GrayOutHighlightRenderer` – Adds transparency to the highlighted text, making it appear grayed-out.
- `DottedUnderscoreHighlightRenderer` – Adds a dotted line under the text.
- `WavedUnderscoreHighlightRenderer` – Adds a waved line under the text.
- `HintHighlightRenderer` – Adds a short line under the beginning of the text.

Highlights that have no renderer associated will not be visible in the text, but can have other useful functions. The actual rendering is implemented in the `HighlightRenderingManager` class, which is a document extension that overrides how the text is rendered based on existing highlights.

Tooltip

Highlights can have a tooltip associated. This tooltip is then displayed when the user hovers the mouse cursor over the highlighted text. The tooltip of a highlight can be set using the `HighlightTooltipInfo` extension in form of an `ITooltipData` instance, which defines the content and priority of the tooltip.

Shortcut

The shortcut column is the column on the editor's right side, beyond the scroll bar. It displays small colored horizontal bars that represent highlights. The column's whole height represents the entire file, so bars at the top indicate highlights at the beginning of the file and bars at the bottom indicate highlights at the end of the file. Hovering the mouse cursor over the bar will display the highlight's tooltip, if any, and clicking the bar will scroll the text cursor to the highlight's location. The color of the highlight's bar in the shortcut column can be set using the `ShortcutBarInfo` extension.

Icon

Highlights can have an icon associated. This icon is then displayed to the left of the line on which the highlighted text begins, in the column on the editor's left side. The icon is a UI element with which the user can interact. This is used for example for breakpoint highlights that show a red circle which can be clicked. The icon can be set using the `IconBarInfo` extension.

Context actions

Highlights can have one or more context actions associated. When the text cursor is placed on a location where highlights with context actions are available, an icon is displayed at the beginning of the line. Clicking this icon (or pressing `Alt + Enter`) will open a menu with the available actions. Context actions of a highlight can be set using the `ContextActionInfo` extension in form of `IContextAction` instances, which provide the appearance and callbacks of the context actions.

4.4.4 Code completion

The Editor component also implements the core user interface and the underlying infrastructure required for code completion. The code completion UI is based on the completion window contained in the AvalonEdit library we use for the editor control, as described in section 3.5.3.

The `ExtBrain.ScriptDevelop.Plugin.Editor.Text.Completion` namespace contains classes and interfaces that allow various levels of access to the code completion functionality:

- The `CompletionWindowExtensions` class contains extension methods for AvalonEdit's `TextEditor` component. These extension methods help keep track of the code completion window associated with the editor component. At this lowest level, the completion window can be any subclass of AvalonEdit's `CompletionWindow`.

- The `CustomCompletionWindow` derives from AvalonEdit's `CompletionWindow` and adds some of the features not present in the original completion window (inactive selection, right column content, automatic widening).

- The `CompletionWindowManager` can be attached to any `TextEditor` and contains methods to open the completion window over the given text segment with the given list of items. It also allows to open the completion window asynchronously, i.e. to run a delegate which returns the list of items in the background, open the window when it is done, and ensure that the previous asynchronous task, if still running, is canceled when a new one is started.

- The `CompletionManager` is the highest level of access to the code completion window that our core implementation provides. It can be attached to any `TextEditor` and implements a pluggable infrastructure for item sources, their order (sorting), and the completion window invocation mechanisms.

This implementation of the code completion functionality is universal and can be potentially used for other purposes than C# code completion. We use it for example to implement code completion also for header sections described in 3.4.1. It can be also used with any `TextEditor` instance, not just the one that is the part of the `ITextEditorDocument`, so for example a debugger console can also offer code completion, even though it does not represent a source code document open within the application.

List items

To represent individual items for the code completion list, the original `CompletionWindow` class uses instances of the `ICompletionData` interface. The `CustomCompletionWindow` subclass uses its own `ICompletionEntry` interface and internally uses an adapter class to make it compatible with AvalonEdit's `ICompletionData`. Our own `ICompletionEntry` looks like this:

```
public interface ICompletionEntry
{
    string Identity { get; }
    string Text { get; }

    object Icon { get; }
    object LeftColumnContent { get; }
    object RightColumnContent { get; }
    object Description { get; }

    bool Visible { get; }
    bool Preselected { get; }

    bool TriggerCompletionUpon(string text,
                               char ch);

    void Complete(TextArea textArea,
                  ISegment completionSegment,
                  EventArgs insertionRequestEventArgs);
}
```

The `Identity` property uniquely identifies a completion item and is used to preselect previously used items in a new completion window. The `Text` property is used for filtering entries during typing.

The `Icon`, `LeftColumnContent`, `RightColumnContent`, and `Description` properties define the visual aspects of the completion item. `Icon` should be a WPF's `ImageSource` instance. Other three can be either a string, a WPF's `UIElement`, or a `HighlightedString`, which is our implementation of a string with additional formatting (e.g. font color or weight). `Description` is displayed in a tooltip next to the completion window when the item is selected.

The `TriggerCompletionUpon` method is called when a new character is typed to determine if we should complete the selected item first (e.g. in C# completion, this returns true for all characters that cannot be a part of an identifier). The `Complete` method is then called to perform the actual completion action.

Data source

The code completion window can be invoked using the `ShowCompletion` method of the `CompletionManager` class. When this method is called, an instance of the `CompletionInvocationContext` class is created. This instance contains various information about the circumstances of the invocation:

- The parent `CompletionManager` instance.
- The document and position within it where the completion was invoked.
- The level of the completion.
- The context object provided by the `CompletionManager` instance.
- The trigger object passed into the `ShowCompletion` method.

Then all implementations of the `ICompletionSourceProvider` instance that were imported by MEF have their `GetCompletionSources` method called:

```
[InheritedExport]
public interface ICompletionSourceProvider
{
    IEnumerable<ICompletionSource> GetCompletionSources(
        CompletionInvocationContext context);
}
```

The result is a list of `ICompletionSource` instances, which are defined like this:

```
public interface ICompletionSource
{
    IEnumerable<ICompletionEntry> GetEntries(IDocument document,
                                             int offset,
                                             out bool inactive);

    ISegment GetCompletionSegment(IDocument document, int offset);
}
```

Upon each such instance, the `GetEntries` method is called in a background task to gather items for the code completion list. When this call finishes, the `GetCompletionSegment` method is called to obtain the suggested completion text segment upon which the code completion window should be invoked. Finally, the code completion window is displayed.

Order

The order of the items in the code completion list can be modified by implementing the `ICompletionComparisonProvider` interface:

```
[InheritedExport]
public interface ICompletionComparisonProvider
{
    IEnumerable<Tuple<double, Comparison<ICompletionEntry>>>
        GetCompletionComparisons(
            CompletionInvocationContext context);
}
```

The `GetCompletionComparers` method is called on each such instance known via MEF. The result is a list of comparisons (`Comparison<ICompletionEntry>` delegates) where each comparison has a `double` priority attached to it. The comparison with the highest priority is used first when sorting, and a comparison with lower priority is used only if all the higher-priority comparisons returned zero (i.e. found the compared items equal for the purposes or sorting).

Triggers

The last aspect of the code completion that the core infrastructure takes care of is the mechanism of invoking the code completion window via so-called *triggers*. When the `CompletionManager` is created, it calls the `GetCompletionTriggers` method of every known implementation of the `ICompletionTriggerProvider` interface:

```
[InheritedExport]
public interface ICompletionTriggerProvider
{
    IEnumerable<IDisposable> GetCompletionTriggers(
        CompletionManager manager);
}
```

This call is intended to register event handlers that will trigger the code completion window. The handlers can be unregistered in the `Dispose` method of the returned objects, which is called when the `CompletionManager` itself is disposed.

The only universal trigger for code completion is the `CodeCompletionCommand`, which is by default mapped to the usual `Ctrl+Space` keyboard shortcut. If the completion window is already displayed, this command invokes it again, increasing the current code completion level. Other triggers can be implemented to invoke the code completion window upon special circumstances, for example when typing an identifier or after a dot character has been typed.

Triggers can pass additional information in a custom "trigger" object when invoking the window. This object is then available in the `Triggers` property of the `CompletionInvocationContext` instance. If multiple triggers invoke the window as a reaction to a single event, the window is invoked only once, and the `Triggers` property contains multiple objects.

4.4.5 Other services

There are other services that the Editor component provides, most importantly:

- `LocationTrackerPlugin` – A plugin that allows defining locations in text documents and notifies subscribers how these locations change when documents are opened, modified and closed. It is used for example in "find references" results window so that individual results would point to correct locations even when the source code has been modified after the search was performed.

- `TooltipManager` – A text document extension that allows registering `ITooltipProvider` instances to the document. It detects when the mouse

cursor is hovering over a portion of the text and then requests a tooltip from the registered providers. This is used for example in:

  - `ResolverTooltipPlugin` – Detects what symbol is under the mouse cursor and displays its declaration and summary.
  - `DebuggerTooltipPlugin` – Detects what variable is under the mouse cursor and displays its value when debugging.

- `MarginManager` – A text document extension that allows creating additional margins (rows and columns) around the editor control. It is used for example to create the icon column for highlights to the left of the editor, the shortcut column for highlights to the right of the editor, or the navigation bar at the top of the editor.

4.5 Builder

Builder is a library that the Compiler component calls to actually compile C# scripts into binary assemblies. It parses the header commands (see section 3.4.1) in the source code and runs the actual C# compiler accordingly. It does not depend on the rest of the application and it can be used on its own, for example it could be easily made into a separate command-line tool for building C# scripts. We have discussed the Builder library in section 3.4.2. It is implemented in the `ExtBrain.ScriptDevelop.Builder` project.

Interface

The central point of the Builder library is the `BuildService` class, which contains the following method:

```
BuildOutput Build(BuildInput input);
```

The `BuildInput` and `BuildOutput` classes represent input and output data of the build process. The `BuildInput` class has an interface like this:

```
public class BuildInput
{
    public BuildMode BuildMode { get; set; }
    public IList<ProjectInput> Projects { get; }
    public IDictionary<string, byte[]> ReplacementFiles { get; }
}
```

The input properties have the following purpose:

- `BuildMode` – One of the following values which indicate what exactly will be performed and what information will be returned or generated (see table 4.2):

  - `Parse` – Only header sections will be parsed and the project structure information will be returned without actually invoking the compiler.

- `Check` – The project will be checked for compilation errors without actually generating full output.
- `Normal` – Normal compilation will be performed.
- `Debug` – Normal compilation will be performed, but debugging information will be included to the output.

	Project structure	List of errors	Output assembly	Debug information
`BuildMode.Parse`	✓			
`BuildMode.Check`	✓	✓		
`BuildMode.Normal`	✓	✓	✓	
`BuildMode.Debug`	✓	✓	✓	✓

Table 4.2: Information returned by the Builder based on `BuildMode`.

- `Projects` – Specifies projects to be built. An input project is identified by the path to the script file.
- `ReplacementFiles` – Allows specifying replacement content for selected files. This is useful when we want to perform `Parse` or `Check` builds with files that have been modified locally but not yet saved on the filesystem.

The `BuildOutput` classes look like this:

```
public class BuildOutput
{
    public IList<BuildError> Errors { get; }
    public IList<ProjectOutput> Projects { get; }
}

public class BuildError
{
    public string Text { get; }
    public ProjectOutput Project { get; }
    public Location Location { get; }
}

public class ProjectOutput
{
    public string Path { get; }
    public ProjectInput ProjectInput { get; }

    public IList<BuildError> Errors { get; }

    public IList<string> SourceFiles { get; }
    public IList<string> ReferencedAssemblies { get; }
    public IList<string> NeighbouringAssemblies { get; }
```

```
        public IList<ProjectOutput> ReferencedProjects { get; }
        public IList<ProjectOutput> DependentProjects { get; }
        public IList<ProjectOutput> AdditionalProjects { get; }

        public string OutputAssemblyPath { get; }
        public string OutputAssemblyName { get; }

        public ITarget Target { get; }
    }
```

The most interesting of the output classes is `ProjectOutput` containing information about the project structure, which is essential later when performing syntactical and semantical analysis of the source code using NRefactory. The interesting information contained in `ProjectOutput` includes:

- `SourceFiles` – A list of source code files that this project consists of.
- `ReferencedAssemblies` – A list of external assemblies referenced by this project.
- `NeighbouringAssemblies` – A list of assemblies referenced either directly by this project, or transitively by any of its dependencies. This list indicates what assemblies must be copied to the output directory with this project's output assembly so that all dependencies are present.
- `ReferencedProjects` – A list of other projects that this project references.
- `DependentProjects` – A list of other projects referencing this project.
- `AdditionalProjects` – A list of other projects whose build process was induced by this project via a dedicated `BUILD` header command.
- `Target` – Information about this project's compiler and framework version.

Implementation

The actual build process of a single project is implemented as a series of individual steps that can be found in the `ExtBrain.ScriptDevelop.Builder.Processing.BuildSteps` namespace. These steps have a predefined order and gradually collect data or perform operations necessary to build the project. The most important steps include (in this order):

1) `ParseStep` – Iterates over all source code files of the current project and scans them for header sections. Parsing of header sections is done by the `HeaderParserService` class, which can parse a single source code file and return the result in form of a list of `IParseResult` instances. The `IParseResult` instance serves to separate the result of parsing for the purposes of better error messaging and potential in-memory caching (we can cache the list of `IParseResult` instances if the source code file has not been modified). Parsing continues recursively on all newly added source code files (which can be added by header commands) until no more such files appear.

2) `BuildReferencesStep` – Iterates over all referenced projects and recursively builds them first before continuing. This step also detects potential project dependency cycles.

3) `LocateReferencesStep` – Tries to locate the full path of assemblies that were referenced only by their name but the corresponding file could not be found. This is mostly the case of framework assemblies, e.g. `System.Core.dll`. The exact location of these assemblies depends on the current framework represented by the project's `ITarget`, which provides an instance of the `IAssemblyLocator` interface:

```
public interface IAssemblyLocator
{
    string TryLocateAssembly(string name);
}
```

Its implementation, the `AssemblyLocator` class, locates the assemblies by searching the installation directory of the .NET framework of the corresponding version.

4) `ReplaceSourcesStep` – Writes the source code files that are modified in the editor into a temporary location so that they could be handed over to the C# compiler instead of the original unmodified source code files.

5) `CheckResourcesAgeStep` – Checks last write times of the input files and compares them to the last write time of the output assembly, if it already exists, in order to find out if it is necessary to recompile.

6) `DetectNeighbouringReferencesStep` – Checks which assembly files the current project depends on (assemblies referenced either directly by this project, or transitively by any of its dependencies). This list indicates what assemblies must be copied to the output directory with this project's output assembly so that all dependencies are present. To find out what assemblies another assembly references, we use this call:

```
Assembly.ReflectionOnlyLoadFrom(path).GetReferencedAssemblies();
```

However, since this loads the assembly into our application domain and thus precludes doing the same again when the assembly changes, we must do it in a separate application domain via remoting. This is implemented in the `AssemblyReflector` class.

7) `CompileStep` – Runs the actual C# compiler. Since the .NET framework contains wrappers for the compiler, we use those wrappers instead of calling `csc.exe` manually. The compiler is invoked by a call to `CSharpCodeProvider.CompileAssemblyFromFile`.

8) `CopyNeighbouringReferencesStep` – Copies the assemblies detected during the previous `DetectNeighbouringReferencesStep` to the output directory. Only non-framework assemblies are copied (i.e. assemblies that did not have to be resolved in the `LocateReferencesStep` using `IAssemblyLocator`).

9) `OutputStep` – Prepares the `ProjectOutput` structure.

4.6 Compiler

Compiler is a primary component whose core responsibility is to integrate the rest of the application with the Builder library. It implements commands to build and run scripts, monitors the currently open scripts and runs the build process automatically in order to display errors and keep the project structure information up to date. It also defines the C# document format and implements basic C#-related functionality that does not require code-aware insight into the source code. It is implemented in the `ExtBrain.ScriptDevelop.Plugin.Compiler` project. The core interface to the Builder library is encapsulated in the `BuildPlugin`, which looks like this:

```
public class BuildPlugin
{
    event EventHandler<BuildEventArgs> BuildStarted;
    event EventHandler<BuildEventArgs> BuildStopped;

    void Parse(string path);
    void Check(string path);

    void Build(string path, bool debug, bool full,
               Action<BuildOutput> callback);

    void Cancel();
}
```

Errors

The application's error highlighting and reporting functionality is implemented in the `ExtBrain.ScriptDevelop.Plugin.Compiler.Errors` namespace. This namespace contains plugins which:

- Run the build process in the `Check` mode (see section 4.5) on the scripts open in the application in order to update the list of errors. If enabled by the user, this process is run automatically each time the source code is modified (`ErrorCheckRunningPlugin`, `ErrorCheckStatusPlugin`).
- Manage a repository of currently present errors (`ErrorRegistryPlugin`, `ActiveErrorListPlugin`).
- Present the current error check status and the list of errors to the user (`ErrorPanelPlugin`, `ErrorStatusBarIndicatorPlugin`).
- Display errors as red underlines in the editor (`ErrorHighlightManager`).

Project structure

The project structure is information about what projects are open within the application, what are their external references, and how they reference one another. This information is essential for code-aware functionality because it defines what must be included in the syntactic and semantic analysis done by NRefactory. The

project structure depends on header sections within the script files. These header sections are parsed by the Builder library and the project structure information is returned in the `ProjectOutput` objects described in section 4.5.

The project structure repository is managed by classes in the `ExtBrain.ScriptDevelop.Plugin.Compiler.ProjectStructure` namespace. It is represented by the following two interfaces:

```
public interface ISourceInfo
{
    string Path { get; }
    IProjectInfo Project { get; }
    IProjectInfo TopLevelProject { get; }
}

public interface IProjectInfo
{
    string Path { get; }
    IProjectInfo TopLevelProject { get; }

    IList<string> GetSources();
    IList<string> GetReferences();

    IList<IProjectInfo> GetReferencedProjects();
    IList<IProjectInfo> GetDependentProjects();

    ...
}
```

`ProjectStructurePlugin` is the plugin that implements the project structure repository. It has the following interface:

```
public class ProjectStructurePlugin
{

    event EventHandler<ProjectEventArgs> ProjectAdded;
    event EventHandler<ProjectEventArgs> ProjectRemoved;
    event EventHandler<ProjectUpdatedEventArgs> ProjectUpdated;

    IEnumerable<IProjectInfo> Projects { get; }

    void Update(string path, BuildOutput output);

    IProjectInfo GetProjectInfo(string projectPath);
    ISourceInfo GetSourceInfo(string sourcePath);
}
```

The `ProjectParsingPlugin` is a plugin that monitors document changes and runs the build process in the `Parse` mode to keep `ProjectStructurePlugin` up to date.

4.7 NRefactory

NRefactory [6] is a C# library that allows parsing and semantic analysis of the C# source code. It is also the name of one of the primary components of our application. The NRefactory component is responsible for all of the application's code-aware functionality. It bears the same name as the library because it uses it heavily.

4.7.1 Core

The NRefactory core is a subsystem of our application that monitors changes to the source code and keeps its representation provided by the NRefactory library up to date, so that if any other part of the application requests the data from NRefactory (e.g. an AST, a compilation, a resolver, etc.), it gets it right away.

We have analyzed the requirements for the NRefactory core subsystem in section 3.5.2. The subsystem is implemented in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.Core` project. This section describes how it is implemented.

Volatile resources

A volatile resource is a resource that changes over time and allows us to retrieve its current value and check its current version. It is represented by the following interface:

```
public interface IVolatileResource<T>
{
    bool CheckIsCurrent(object version);
    Tuple<T, object> LoadCurrent();
}
```

The `LoadCurrent` method retrieves the current value and version of the resource as a `Tuple` where the first item is the value of the resource and the second item is a version token representing the version associated with the returned value. The version token can be passed to the `CheckIsCurrent` method in order to check if the corresponding value is still up to date. If this method returns `false`, we can use `LoadCurrent` again to retrieve the updated value.

It is possible for a resource to become invalid (for example if it represents a file that has been deleted). In this case, the `LoadCurrent` method returns always `null` and `CheckIsCurrent` returns always `false`.

There is a helper class `VolatileResourceSnapshot<T>` that represents a snapshot of the volatile resource passed to its constructor. It allows easy access to the value of the resource at the time of the snapshot's creation, and contains a method to check if the snapshot is still up to date. Because NRefactory structures are immutable, this class is internally used as a base class for objects that represent the output of NRefactory based on the value of the snapshot.

Resource loaders

Resource loaders are singleton plugins that act as a factory for `IVolatileResource` instances representing source code files and assembly files.

- `AssemblyLoaderPlugin` is a single point to efficiently load external assemblies into NRefactory. It contains a single public method `GetAssemblyResource (string assemblyPath)` returning an instance of the `IVolatileResource <IUnresolvedAssembly>` interface representing the given assembly.
- `SourceLoaderPlugin` is a similar plugin for source code file loading. It contains a public method `GetSourceResource(string sourcePath)` that returns an `IVolatileResource<IDocument>` instance representing the given file. However, source code files can be opened in the application, modified by the user and not yet saved. In this case, we want to use this modified content of the source code file for NRefactory processing rather than its original content on the filesystem. For this purpose, the `SourceLoaderPlugin` contains a method `NotifySourceDocumentChanged (string sourcePath, IDocument document)` that allows substituting the content of individual files. This method is called each time a source code file is modified in the application and the updated text of the source code file is passed into it. This way, internal source code modifications are included seamlessly into the version checking mechanism of `IVolatileResource`.

Project representation

The NRefactory core subsystem monitors the structure of the C# scripts that are open in the application and keeps an up-to-date `IProjectContent` instance for each project. We have described what we need for project representation in section 3.5.2. For the purpose of the NRefactory library, a project consists of:

- A path that acts as an identifier of the project.
- A list of source code files represented as paths.
- A list of references. A reference may be either a path to an external assembly, or a path to another project.

Each of the resources that make up a project (source code file, external assembly, another referenced project) can change over time, so we use the `IVolatileResource` and `VolatileResourceSnapshot` concepts described above.

A project is represented by the `Project` class. Because `IProjectContent` is immutable and represents the state of a project at a certain time, the `Project` class implements `IVolatileResource<IProjectContent>`. The three types of resources that make up a project are represented like this:

- A source code file is identified by its path, represented as `IVolatileResource <IDocument>`, and its specific version is represented as an immutable `SourceSnapshot` class (subclass of `VolatileResourceSnapshot<IDocument>`). Upon construction of a `SourceSnapshot`, the file content is parsed and converted into an `IUnresolvedFile`, which is then used to represent the file inside the `IProjectContent` of the current project. The `SourceSnapshot` class

also exposes the source file's `SyntaxTree` (the parsed AST) and `IDocument` (the source code text), which are useful to the rest of the application.

- An assembly is identified by its path, represented as `IVolatileResource <IUnresolvedAssembly>`, and its specific version is represented as an immutable `AssemblyReference` class (subclass of `VolatileResourceSnapshot <IUnresolvedAssembly>`). The `AssemblyReference` class implements the `IAssemblyReference` interface and because of this it can be used to represent a reference inside the `IProjectContent` of the current project.

- An internal project reference is identified by its path, represented as its own `Project` class (implementing `IVolatileResource<IProjectContent>`), and its specific version is represented as an immutable `ProjectReference` class (subclass of `VolatileResourceSnapshot<IProjectContent>`). The `ProjectReference` class also implements the `IAssemblyReference` interface and because of this it can be used to represent a reference inside the `IProjectContent` of the current project.

The `Project` class keeps the original `IVolatileResource` instances for its resources, checks is they are up to date, and if they are found to be out of date, it loads their current version and updates the `IProjectContent` accordingly.

The loading process can be expensive – source code files need to be loaded and parsed, assembly files need to be loaded, and then in both cases they need to be converted into an NRefactory type system. Because of this, the loading process is done asynchronously. The `Project` class keeps track of which of its resources are scheduled for loading, and if the same resource is updated or removed while it is already scheduled for loading, the corresponding loading task is canceled.

The `IVolatileResource` does not support notifications (events) to signal that the resource has been changed. Instead it needs to be polled explicitly. When this is done depends on the type of the resource:

- If the application has been deactivated, all resources of the project are checked for modifications when the application becomes active again.

- A source code file is checked for modifications after it has been modified in the application's editor. This check always results in reparsing of the source code file, so it is performed after a slight delay which is reset each time another modification occurs. This way, the parser does not run continuously after each keystroke, but rather only when there is a pause in the user's typing.

- Each time the current project's `IProjectContent` is modified, this change needs to be reflected in all other projects that referenced the current project. Because of this, the `Project` class keeps track of other projects that reference it and notifies them whenever the current `IProjectContent` changes. The referencing projects then update their own `IProjectContent` accordingly, potentially cascading the update further down the dependency graph.

Interface

The purpose of the NRefactory core subsystem is to keep up-to-date information about the syntax trees and the type system of projects active in the application. Other components of the application access this information in order to perform their specific functionality.

When another component requests information about a certain source code file, it is retrieved in the form of the `ISourceSnapshot` instance:

```
public interface ISourceSnapshot
{
    string SourcePath { get; }
    IDocument Document { get; }
    SyntaxTree SyntaxTree { get; }
    CSharpUnresolvedFile UnresolvedFile { get; }
}
```

The instance and all its parts are immutable and represent the state of the source code file as it had been registered within the project at the time of retrieval. Internally, this interface is implemented by the `SourceSnapshot` class that has been described above. The `SyntaxTree` property is the most interesting for other components because it gets the source code file's parsed AST.

Often we also need to access the type system of the entire project, not just a single source code file. This is required for creating a resolved type system and for running the resolver on the AST in order to find out its semantic meaning. The type system for the entire project is represented by the `IProjectSnapshot` interface:

```
public interface IProjectSnapshot
{
    string ProjectPath { get; }
    IProjectContent ProjectContent { get; }
    ICompilation Compilation { get; }
}
```

As with `ISourceSnapshot`, the instance and all parts of `IProjectSnapshot` are immutable and represent the state of the project at the time of retrieval.

Very often, both the information about a source code file and its project is required. We allow retrieval of both at the same time in the form of an `ICombinedSnapshot` interface, which inherits from both `ISourceSnapshot` and `IProjectSnapshot`. It is ensured that the returned version of `IProjectContent` contains exactly the returned version of `IUnresolvedFile`.

Sometimes we also need to make sure that the returned snapshot corresponds exactly with the current state of the project. That may not always be the case – the subsystem may be waiting for the user to finish typing, or the parsing may be in progress but not yet complete. In case we need to use the most recent information, the methods for snapshot retrieval contain an optional boolean parameter `forceCurrent` that will ensure the most up-to-date information is returned. It is achieved by checking the given source code file's version first and potentially waiting until it is parsed. That means the call may be blocking if `forceCurrent` is used.

It is also useful to know when the parsed information about a certain source code file or project has been updated. This is signaled by the `ProjectUpdated` event, which offers detailed information about the update in its argument object `ProjectUpdatedEventArgs`.

The rest of the application has two options how to access the NRefactory core subsystem:

- The first option is to use the `NRefactoryPlugin` singleton, which contains all the necessary methods and events to access the information from the parser:
  - `GetSourceSnapshot`, `GetProjectSnapshot` and `GetCombinedSnapshot` methods return for the given path the snapshot interfaces described above.
  - `GetProjectSnapshots` method returns an enumerable of snapshots for all tracked projects.
  - `ProjectUpdated` event is triggered each time the information about a project is updated.
- Another option is via the `NRefactoryExtension`, which is an extension plugin of all documents the format of which is of the `CSharpDocumentFormat` type. Because this extension is always linked with a specific document, its methods do not need to be passed the path to the source code file. The `NRefactoryExtension` can be accessed like this:

  ```
  var extension = document.GetExtension<NRefactoryExtension>();
  ```

  The `NRefactoryExtension` exposes the following members:

  - `GetSourceSnapshot`, `GetProjectSnapshot` and `GetCombinedSnapshot` methods that return the snapshot interfaces for the associated document.
  - `SourceUpdated` event that is triggered whenever the information about the associated document is updated.
  - `ProjectUpdated` event that is triggered whenever the information about the project of the associated document is updated.

  The two events of this extension are particularly useful for other document extensions that depend on the parsing information. For example:

  - Code folding depends only on the AST of the given document, so it is updated in the `SourceUpdated` event.
  - Syntax highlighting depends on the whole project (because it needs to resolve types, for instance), so it is updated in the `ProjectUpdated` event.

The `NRefactoryPlugin` singleton contains a number of methods that notify it about changes in the project structure. This is because the core functionality of the NRefactory core subsystem is designed to be stand-alone and not to depend on the rest of the application. Therefore, the subsystem needs to be notified whenever there are changes that affect it. This is done by a number of simple additional plugins that observe the state of the application and notify the `NRefactoryPlugin` accordingly. These plugins reside in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.Core.Connectors` namespace.

4.7.2 Code analysis

The code analysis subsystem is a component that uses the NRefactory library to analyze source code files that are open in the application, to scan them for various issues, and to offer context-sensitive actions and refactoring transformations. The discovered issues and actions are then presented to the user, who can review the issues and invoke the actions, including actions that are aimed to fix the discovered issues.

- Code issues are presented as text highlights of various intensity (e.g. the code that is detected to be unreachable or redundant is displayed in gray).
- Code actions pertaining to the current caret position are indicated by an icon to the left of the current line. When this icon is clicked, a popup menu that lists the available actions is opened (or it can be opened by pressing `Alt+Enter`, to avoid having to use the mouse). When an action from this menu is selected, it is executed. The actions in the menu can come from two different sources:

  a) Code action analysis, which specifically looks for available code actions relevant to the current caret position and text selection.

  b) Code issue analysis, where each issue can have a list of actions associated with it. These actions are usually meant to fix the issue is some way (e.g. by removing the redundant code), and they are offered to the user when the caret position is located inside the text segment marked by the issue.

  In fact, any text highlight can have context actions associated with it that contribute to the context action menu (see section 4.4.3), but here we are interested only in actions resulting from code analysis.

Code issue analysis and code action analysis are run only on open documents. They are both triggered on a document when the type system information about the project that contains the document is updated (i.e. when any file inside the project or dependent projects is reparsed). Because code action analysis depends also on the current caret position and text selection, it is additionally triggered whenever the caret position or text selection within the document changes.

The analysis is run on a document only when the document is visible in the application. If the document is not visible and the circumstances occur that trigger the analysis, it is postponed until the document becomes visible again. This is to avoid unnecessary overhead because there is no need to analyze documents that the user cannot see.

Implementation

The core functionality of the code analysis subsystem is implemented in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.Analysis` project. The main class that manages analysis on a single document is `CodeAnalysisManager`, which is an extension of C# documents. This class monitors circumstances that trigger the analysis, performs it and processes its results.

Individual code issues and code actions are detected by classes implementing the `ICodeIssueProvider` or `ICodeActionProvider` interfaces. These interfaces are exported using MEF and queried by the `CodeAnalysisManager`. They are defined as follows:

```
public interface ICodeAnalysisProvider
{
    ICodeAnalysisProviderMetadata Metadata { get; }
}

[InheritedExport]
public interface ICodeActionProvider : ICodeAnalysisProvider
{
    IEnumerable<ICodeAction> GetActions(
        CustomRefactoringContext context
    );
}

[InheritedExport]
public interface ICodeIssueProvider : ICodeAnalysisProvider
{
    IEnumerable<ICodeIssue> GetIssues(
        CustomRefactoringContext context
    );
}
```

The parent interface `ICodeAnalysisProvider` exposes various metadata about the analysis provider (title, description, category). The only properties of the `ICodeAnalysisProviderMetadata` interface that are currently important are:

- `Identity` – Gets unique string identifier of the analysis provider (for internal usage).
- `ConfigurationNames` – Gets string identifiers that can be used to refer to this analysis provider in the configuration.

When code analysis is triggered, all currently imported instances of the `ICodeActionProvider` and `ICodeIssueProvider` interfaces are gathered and their `GetActions` and `GetIssues` methods are called in order to find code issues and code actions that are present in the current version of the document. The methods are executed in parallel, but with a limited degree of concurrency so that the task scheduler would not be overloaded with too many tasks at once. When a single method call finishes, previous code actions and issues of the corresponding provider (identified by the `Identity` metadata property) are replaced by the newly discovered code actions and issues, if there were any.

The `CustomRefactoringContext` object passed into these methods is a subclass of NRefactory's `RefactoringContext` class and encapsulates all information required to analyze the document and the type system of the corresponding project. It also provides contextual information for the correct proposal of context code actions (i.e. the current caret position and text selection).

Individual code actions are represented by the `ICodeAction` interface:

```
public interface ICodeAction
{
    ICodeActionMetadata Metadata { get; }
    string Description { get; }

    TextLocation Start { get; }
    TextLocation End { get; }

    Action<CustomScript> Run { get; }
}
```

Discovered code actions are transformed into invisible text highlights that only have the corresponding context action assigned. This means that if the user places the editor's caret in the text segment denoted by the code action's `Start` and `End` properties, the corresponding context action is offered in the context action menu and can be selected by the user.

When the user selects the action from the context action menu, we check that the current version of the document is the same as the version upon which the code analysis generated the action (the versions should be in most cases equal because any modification of the document triggers new analysis, but it can happen that the action is selected before the new analysis completes and replaces the available actions). If the versions are equal, we call the `Run` delegate to actually perform the code action.

The `Run` delegate receives a `CustomScript` instance that represents the environment for refactoring the current document. It is a subclass of the `Script` and `DocumentScript` classes defined in NRefactory. These classes offer various methods that allow and aid the refactoring of the current document.

Code issues are represented by a similar interface, `ICodeIssue`:

```
public interface ICodeIssue
{
    ICodeIssueMetadata Metadata { get; }
    string Description { get; }

    TextLocation Start { get; }
    TextLocation End { get; }

    IList<ICodeAction> Actions { get; }
}
```

The `Actions` property contains a list of code actions associated with the issue. These actions are usually meant to fix the issue in some way. If the user places the editor's caret in the text segment denoted by the code issue's `Start` and

`End` properties, the actions from the code issue are added to the list of currently available context actions.

Discovered code issues are transformed into text highlights (see section 4.4.3) of various intensity, so that the user would be visually notified that the issues exist. How these highlights look visually is determined by the `Marker` and `Severity` properties of the code issue's `Metadata` property.

There are two highlight types (two different ways a code issue can be highlighted in the editor):

- By rendering the affected code with a color underscore.
- By graying-out the affected code (this usually denotes unreachable or redundant code).

The intensity of the highlight is determined by the code issue's severity:

- `Error` – The code issue is displayed in red and counted as an error. If the highlight is of the gray-out type, the affected code is displayed in a red font instead.
- `Warning` – The code issue is displayed in orange and counted as a warning.
- `Suggestion` – The code issue is displayed in green.
- `Hint` – The code issue is displayed only as a subtle, short underscore at the beginning of the affected code, so that it would not distract but was still noticeable.
- `None` – The code issue is ignored and is not displayed in any way.

The highlights that are created to represent discovered code issues also have a tooltip and a shortcut assigned. The `Description` of the code issue is used as the tooltip for the highlight, and a shortcut marker is inserted to the right margin of the editor with the color corresponding to the code issue's severity. This way, the icon to the top right of the editor (above the shortcut margin) can be either red, orange or green according to if there are any errors or warnings among the code issues discovered by the code analysis.

Configuration

Code issue analysis highlights places in the code that are potentially dangerous and is meant to improve code quality. However, not every user considers all issues equally relevant to his or her coding style or to the given situation. Sometimes the user may want to ignore a particular issue, either globally or locally in a particular case, file or region, or lower the severity of a certain issue so that it would for example not stand out and distract so much.

Local code issue suppression is usually done directly in the source code using a special directive inside a comment that disables a particular occurrence of the issue, either between a pair of comment directives, or directly after a comment directive. However, this method of code issue suppression is not supported in the current version of the application.

Instead, global code issue suppression is supported via configuration. Using a special section of the configuration file, the user can change the severity of code issues, for example like this:

```
<CodeAnalysis>
    <OverrideSeverity Provider="RedundantUsingDirective"
                      Severity="Suggestion" />
    <OverrideSeverity Provider="RedundantThisQualifier"
                      Severity="None" />
</CodeAnalysis>
```

Such configuration elements override the severity of the specified code issues. Specifying the severity of `None` disables the corresponding analysis provider completely so that it is not even run.

The name of the provider must correspond to one of the names available through the `ConfigurationNames` property of the provider's metadata. Since these names are not used in the application's user interface, they are specially made available through the context action menu as the last item, if there are currently any code issues or code actions available.

Both code issue and code action providers can be configured in this way, however changing severities of code action providers to other values than `None` in order to turn them off does not have any useful effect.

The configuration of the code analysis is implemented in the class `CodeAnalysisConfiguration` that contains a single public method:

```
Severity? GetOverriddenSeverity(ICodeAnalysisProvider provider)
```

This method is used by the `CodeAnalysisManager` class and returns if the given provider has any severity configured.

4.7.3 Code analysis by NRefactory

The NRefactory library itself contains a vast number of ready-made code issue and code action providers. This number still grows as the library is actively developed. We have designed our code analysis subsystem specifically to be compatible with the code analysis providers of NRefactory, however, we use our own, simpler and more compact versions of the interfaces for providers, issues and actions. Our application can still use all of the code analysis providers from NRefactory by implementing an adapter that allows them to be used by our own code analysis subsystem.

The code analysis providers in the NRefactory library use two different abstract base classes, `CodeIssueProvider` and `CodeActionProvider`. Metadata for the providers are defined using special attributes, `IssueDescriptionAttribute` and `ContextActionAttribute`, that annotate the non-abstract subclasses implementing a particular code issue or action provider. These classes and attributes expose all the important information and functionality that we need in order to implement our `ICodeIssueProvider` and `ICodeActionProvider` interfaces for them.

Implementation

The adapters are implemented in the project `ExtBrain.ScriptDevelop.Plugin.NRefactory.Analysis.Adapter`:

- `NRefactoryCodeIssueProviderAdapter` – Adapts the `CodeIssueProvider` of NRefactory to use our `ICodeIssueProvider` interface.
- `NRefactoryCodeActionProviderAdapter` – Adapts the `CodeActionProvider` of NRefactory to use our `ICodeActionProvider` interface.

The project also contains the classes `NRefactoryCodeIssueProviderSource` and `NRefactoryCodeActionProviderSource`. These two classes implement the `ICodeIssueProviderSource` and `ICodeActionProviderSource` interfaces, which are a variant of `ICodeIssueProvider` and `ICodeActionProvider` and allow exposing multiple instances of these interfaces through MEF:

```
[InheritedExport]
public interface ICodeIssueProviderSource
{
    IEnumerable<ICodeIssueProvider> GetProviders();
}

[InheritedExport]
public interface ICodeActionProviderSource
{
    IEnumerable<ICodeActionProvider> GetProviders();
}
```

All the code analysis providers offered by the NRefactory library are contained in the `ICSharpCode.NRefactory.CSharp.Refactoring` assembly. The `NRefactoryCodeIssueProviderSource` class searches this assembly for all non-abstract subclasses of `CodeIssueProvider` with the `IssueDescription` attribute, instantiates them, wraps them using the adapter, and exposes them via the `GetProviders` method. The `NRefactoryCodeActionProviderSource` does the same for `CodeActionProvider` subclasses.

Limitations

Although the adapters technically allow us to use all of the code analysis providers from NRefactory, not all are supported. The `Script` object that is passed into the delegate performing a code action contains additional helper methods for more complex refactoring operations (e.g. renaming, new type creation, and other actions that potentially span multiple documents). Such methods are not implemented in the current version of the application and code actions that use them are not supported. This is because such operations are more useful for complex projects and therefore are outside the scope of a C# script editor.

4.7.4 Imports

One of the most important code-aware functions of a C# code editor is the ability to automatically add using statements and references for types that are unknown in the current context. This is mainly done by the code completion, which allows listing types from yet-unimported namespaces and even from yet-unreferenced assemblies and adding the required using statement or reference upon selection of the type. However, it is also useful to have an unknown type highlighted in red and offer context actions that will import the appropriate namespace or reference when invoked. This is especially important when copying and pasting code snippets from another source, where we can immediately see what types were not recognized in the new context and add correct namespaces and references accordingly.

Please note that "import" is not a common word in the C# nomenclature. When we say "importing an entity", we mean the process of adding the appropriate using statement or assembly reference or referring to the entity in such a way so that it would be correctly recognized. Similarly, by an "import" we mean an entity that requires a new using statement, a new assembly reference, or a special form in order to be recognized.

Not only types may need importing. Other entities that may lose their meaning if taken out of context of referenced assemblies or using statements are extension methods and namespaces.

Data source

In order to offer importing of entities (no matter if done via code completion, or as context actions correcting an unknown identifier), we need to gather all known types, extension methods and namespaces and determine if they could be used at a given position in the source code and under what circumstances.

This process is implemented in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.Import` project by the `ImportProvider` class. Upon construction this class is passed the resolver context at the given position within the source code and the syntax tree of the current file. The class contains only three public methods:

```
public IList<TypeImportResult> GetTypeImports(
    bool standAlone,
    Func<ITypeDefinition, bool> predicate = null
);

public IList<MethodImportResult> GetExtensionMethodImports(
    IType targetType,
    Func<IMethod, bool> predicate = null
);

public IList<NamespaceImportResult> GetNamespaceImports(
    Func<INamespace, bool> predicate = null
);
```

All three methods return a list of found imports, and take an optional parameter `predicate` that can be used to filter the returned entities early before more processing is performed on them. The `targetType` parameter denotes the type of the expression upon which extension methods are being invoked. The `standAlone` parameter denotes that the types are not being used after a dot operator (if they are, we can skip some processing).

The results of these calls are represented by the `ImportResult` base abstract class:

```
public abstract class ImportResult
{
    // Assembly (full path) of the imported entity
    public string Assembly { get; }

    // Namespace (full name) of the imported entity
    public string Namespace { get; }

    // Is new using statement required to use the entity?
    public bool NewUsingRequired { get; }

    // Is new reference required to use the entity?
    public bool NewReferenceRequired { get; }

    // Text that should be used to reference this entity
    public string CompletionText { get; }

    ...
}
```

If the property `CompletionText` is not null, it means that the entity cannot be referred to simply by its name and instead the text in `CompletionText` needs to be used (this is used for example when we cannot add a new using statement because of conflicts or when the type's name has a different meaning in the current context and so we must refer to the type by its full name). For extension methods, a non-null `CompletionText` means that we cannot use the instance-member-like invocation syntax but instead we must call the extension method as a static method using the string in `CompletionText`.

Three specific classes derive from this class: `TypeImportResult` for types, `MethodImportResult` for extension methods and `NamespaceImportResult` for namespaces. `NamespaceImportResult` is the simplest because namespace usage is not related to using statements and thus only `NewReferenceRequired` is used.

Conflict detection

When adding new using statements and referring to types in external namespaces, conflicts may arise. The `ImportProvider` class tries to detect the most common cases of such conflicts and return results with such parameters that their usage will not cause new errors. For example, if a new using statement cannot be added when importing a type because adding the using statement would cause ambiguity in the source code, the import's `CompletionText` is set to the type's full name instead.

There are two kinds of conflicts that are detected and prevented by the application:

- When importing a type, `ImportProvider` tests if the type can be safely accessed by its simple name. It does so by looking up the type's name in the current resolver context to see if the name already denotes another type. If yes, then we cannot use the simple name because it would be either overshadowed by the already known type, or type ambiguity would arise. In such cases, the full name needs to be used to refer to the imported type.

- When adding a new using statement, `ImportProvider` tests if such an action would not result in ambiguity in the source code.

  1) The source code is scanned for constructs that refer to a type by its simple name. If the referred type is known at that point because of a using statement, then we need to be careful, because if a namespace was imported that also contained a visible type with the same signature (simple name and type parameter count), adding such a using statement would cause type ambiguity.
  2) The source code is scanned for extension method invocations. When adding a new using statement, we resolve each of these invocations again, but using a resolver state that has been modified to contain the new namespace import. If any of these invocations result in an ambiguous call, then adding the using statement would cause extension method ambiguity.

  Successfully detecting a conflict when adding a new using statement means it cannot be done and instead the entity in question needs to be referred to without it:

  – A type needs to be referred to by its full name.
  – An extension method needs to be invoked as a static method of its containing type, which needs to be referred to by its full name.

Import assembly sources

One of the important features when considering imports and code completion is the ability to offer entities from yet unreferenced assemblies and add the references upon request. Because there can be many potential assemblies to offer entities from, we need a way to find out what assemblies to include in the selection.

For this purpose, there is an interface `IImportAssemblySource`:

```
[InheritedExport]
public interface IImportAssemblySource
{
    IEnumerable<IUnresolvedAssembly> GetImportAssemblies();
}
```

When imports are gathered by the `ImportProvider` class, all plugins implementing this interface are queried and the relevant entities from the assemblies returned by these plugins are included in the results.

There are currently three implementations of `IImportAssemblySource`:

- `ProjectAssemblySource` – Returns assemblies that represent the currently open C# script files.
- `UsedAssemblySource` – Returns assemblies that are referenced by the currently open C# script files.
- `PrecachedAssemblySource` – Returns assemblies that were explicitly marked to be precached for the purpose of importing new references. The user can explicitly choose assemblies that are precached upon the application startup and remain in the cache until the application is closed. This can be configured in the `PrecachedAssemblies` section of the configuration file.

Because `ProjectAssemblySource` can lead to importing other C# script files, the `ImportProvider` class also checks if the unreferenced assembly from which it considers offering entities is not dependent on the current assembly. If yes, such imports are not offered, because adding a new reference would in this case lead to a dependency cycle.

Import analysis

In addition to offering imports in code completion, we want to have unknown types highlighted and offer context actions that will import the corresponding types by performing the appropriate refactoring when invoked. We use the code analysis engine to achieve this by implementing code issue and code action providers. The `ExtBrain.ScriptDevelop.Plugin.NRefactory.Analysis.Import` component offers two plugins for the code analysis engine:

- `UnknownIdentifierCodeIssueProvider`, which gathers all identifiers that could not be resolved and displays them in red color.
- `UnknownIdentifierCodeActionProvider`, which offers context actions to correct unknown identifiers in case they can be corrected.

The issues do not contain the actions directly. This is so for performance reasons, because detecting if an unknown identifier can be fixed includes iterating over all accessible types from all known assemblies. We only do this for a single identifier once the user moves the caret over it.

Detecting unknown identifiers

Detecting unknown identifiers cannot be easily done by looking at the errors returned from the compiler, mainly because:

a) The error list from the compiler is meant for user reference and is not suited for parsing.

b) We need additional information about the context of unknown identifiers (for example the type upon which the unknown extension method is invoked).

Instead, whenever the source code is parsed by the NRefactory core, we look at each identifier and try to resolve it in order to find out what the identifier denotes semantically. This resolving process can return errors in case the identifier could not be resolved to any entity, which is exactly what we need in order to mark the identifier as unknown.

This way of detecting unknown identifiers has one disadvantage – the analyzed code snippet needs to be syntactically correct, or else we could not resolve it semantically. This means, for example, that we cannot immediately mark an unknown type while the user writes a member declaration because the statement is not yet complete and cannot be properly parsed. The unknown type gets recognized later, the moment the parser recognizes the declaration statement syntax and knows it should resolve the identifier as a type. However, this is not a problem since the import analysis is meant for code taken out of context (e.g. by copy & paste) and such code is usually syntactically correct. Importing entities as the user writes code is instead handled by the code completion.

Unknown identifier representation

There are two kinds of situations where an unknown identifier may be encountered – stand-alone or after a dot as an unknown member.

1) **Stand-alone identifiers.** These are represented in the AST either as `IdentifierExpression` when part of an expression, or as `SimpleType` when outside an expression. If they are unknown, both are resolved to `UnknownIdentifierResolveResult`. Such identifiers are represented in our code as `UnknownSingleIdentifier`, which does not carry any additional information.

2) **Member identifiers.** These are represented in the AST either as `MemberReferenceExpression` when part of an expression, or as `MemberType` when outside an expression. Their resolving is more complicated:

    - `UnknownMemberResolveResult` in case of simple member expressions.
    - `MethodGroupResolveResult` with zero eligible methods in case of member expressions where a method group is expected.
    - `ErrorResolveResult` in case the expression before the dot was resolved to a type or a namespace.

Unknown member identifiers are represented in our application by three separate classes, based on how the identifiers are used in the code (i.e. what kind of expression the dot before the unknown identifier expands):

- `UnknownInstanceMember` – There is an expression before the dot. Carries the type entity of the expression and the `MemberReferenceExpression` in which it was detected (which is needed when refactoring an extension method call into a static method call).

- `UnknownStaticMember` – There is a type before the dot. Carries the type entity.

- `UnknownNamespaceMember` – There is a namespace before the dot. Carries the namespace entity.

`UnknownIdentifier` is the base abstract class for all our unknown identifiers. It carries the identifier node itself, its name and its type argument count. Its property `IsAttributeTypeName` denotes that the unknown identifier is used in an attribute declaration (in which case we must take into account the possible `Attribute` suffix).

The `UnknownIdentifier` class contains the `GetUnknownIdentifierFromNode` static method which analyses the given AST node according to the rules described above and returns the correct instance of `UnknownIdentifier`, or `null` if the given node does not represent an unknown identifier. The implementation of `UnknownIdentifierCodeIssueProvider` is then a simple case of traversing the syntax tree, calling `GetUnknownIdentifierFromNode` on its every node and returning a code issue for every `UnknownIdentifier` returned.

Correcting unknown identifiers

Whenever the user moves the caret over an unknown identifier that was detected by the `UnknownIdentifierCodeIssueProvider`, we want to offer corrective context actions. That means we need to find out what the identifier could possibly mean, offer the guessed meanings in the form of context actions, and refactor the code accordingly if the context action is chosen.

The actual process of determining what imports to offer for an unknown identifier is implemented in `UnknownIdentifierCodeActionProvider`. The `ImportProvider` class described above is used to search for the entities to offer. What exactly is offered depends on the type of the unknown identifier:

- `UnknownSingleIdentifier` – We offer all accessible types and all root namespaces with the correct name. This also includes nested types because it is a common scenario to copy code referencing public nested types out of the parent class.

- `UnknownInstanceMember` – We offer all accessible extension methods that have the correct name and can be applied to the target expression based on its type.

- `UnknownNamespaceMember` – We offer all accessible types and all child namespaces that have the correct name and reside in the correct target namespace.

- `UnknownStaticMember` – We do not offer anything. Unknown static members cannot be corrected by a new using statement or a new reference because the parent type is already recognized and fully known.

The refactoring that happens when the context action is selected is implemented in the `ImportRefactoring` class and includes these actions:

- If the import has `NewUsingRequired` set to `true`, a new using statement is added to the top of the current source code file. This is implemented by NRefactory itself, in the `UsingHelper.InsertUsing` method.

- If the import has `NewReferenceRequired` set to `true`, a new reference is added to the current project. This is implemented by our own `HeaderRefactoring` class because it involves header sections that are specific to our application.

- If the import has a non-null `CompletionText` property, we cannot refer to the imported entity by its short name, but instead we need to use the text in this property.

  - For types, this simply means that we must replace the identifier with the content of `CompletionText`.
  - For extension methods, this means that we must refactor the call from the instance method form to the static method form, and use the content of `CompletionText` to identify the static method. For example, if the original code snippet was `items.Contains(x)` and `CompletionText` was `System.Linq.Enumerable.Contains`, we need to refactor the snippet to `System.Linq.Enumerable.Contains(items, x)`.

    To aid with such refactoring, the `UnknownInstanceMember` class, which represents the unknown identifier that can be an extension method, carries in its `Expression` property the original `MemberReferenceExpression` node. This way we can use NRefactory's refactoring infrastructure that allows easy AST node construction and replacement.

4.7.5 Syntax highlighting

Requirement R2.1 (advanced syntax highlighting) means we need to implement syntax highlighting that is based not solely on regular expressions, which the AvalonEdit library contains extensive support for, but also on the real output from the NRefactory processing. Such syntax highlighting is implemented in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.SyntaxHighlighting` namespace.

Syntax highlighting is implemented by the `SyntaxHighlightingManager` document extension plugin. When a new C# document is opened, the plugin assigns the corresponding editor control the correct syntax highlighting definition (defined in the `CSharp-Mode-VS-Base.xshd` file). However, this would only perform regular-expression-based highlighting. So, in addition to this, the plugin also monitors the NRefactory core and each time the source code is reparsed, it traverses the document's AST and recolors certain identifiers:

- If the identifier is represented in the AST as a token node, it is a keyword and will be recolored accordingly. This gives the keyword color even to identifiers which are not reserved in the C# language, but recognized as keywords only in certain syntactic constructs (e.g. keywords specific to LINQ expressions).

- If not, we try to resolve the identifier in order to find its semantic meaning:
  - If it resolves to a type, it is recolored as a type (unless it is `var` or `dynamic`, which are recolored as keywords). This allows us to distinguish between type and namespace names.
  - If it resolves to a local variable named `value` and we are in a property accessor body, it is recolored as a keyword.

This gives us syntax highlighting similar to that in Visual Studio. The process can be potentially easily expanded to give different colors to other language constructs based on semantics.

4.7.6 Code completion

We have analyzed how we plan to implement code completion in section 3.5.3. The general user interface for code completion is implemented as a part of the Editor component and described in section 4.4.4. In this section we describe the implementation of code completion for the C# language. The implementation is located in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.CodeCompletion` project, primarily in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.CodeCompletion.CSharp` namespace.

Engine

The NRefactory library exposes its code completion functionality through the `CSharpCompletionEngine` class. We use this class as the base and inherit from it our own `CustomCompletionEngine`, which exposes the following two methods:

```
public IEnumerable<ICompletionEntry> GetStandardCompletionData();

public IEnumerable<IImportCompletionEntry> GetImportCompletionData(
    bool includeReferencedAssemblies,
    bool includeUnreferencedAssemblies
);
```

These methods are used to implement all three levels (basic, import, reference) of C# code completion.

- `GetStandardCompletionData` uses the `GetCompletionData` method of the base `CSharpCompletionEngine`.
- `GetImportCompletionData` uses the `ImportProvider` class described in section 4.7.4 to gather `ImportResult` instances for possible unimported or unreferenced entities and returs corresponding `IImportCompletionEntry` instances for them.

There are a lot of different types of entries implementing `ICompletionEntry` that can be returned from `CustomCompletionEngine`. They form a complex hierarchy located in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.CodeCompletion.Entries` namespace.

Mode detection

As we have stated in section 3.5.3, we need a way to detect if a new symbol can be named at a particular location in the source code. If yes, we cannot automatically open the completion window at that location, or we must open it in a special mode called inactive selection.

The decision process is implemented in the `CustomCompletionEngine` by the `GetNameDefinitionStatus` method, which can return three different values:

- `NameDefinitionStatus.None` – There cannot be a new name introduced at the completion location. We can display the completion window with active selection.
- `NameDefinitionStatus.NonExclusive` – There can be a new name introduced at the completion location, but also there can be another symbol referenced at the location as well. We must display the completion window with inactive selection.
- `NameDefinitionStatus.Exclusive` – There must be a new name introduced at the completion location. We cannot display the completion window.

The `GetNameDefinitionStatus` method uses the same tools and mechanisms as the `CSharpCompletionEngine` to detect what syntactic constructs are possible at the given location.

Header sections

So far, we have been concerned only with code completion for the C# language itself. However, we have introduced special comment blocks called header sections (see section 3.4.1), which are complex enough to benefit from code completion as well. It is implemented in the Builder library in the `ExtBrain.ScriptDevelop.Builder.HeaderCompletion` namespace and integrated with our application in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.CodeCompletion.Header` namespace.

4.8 Debugger

Debugger is a primary component of the application that implements the basic debugging functionality. It provides commands for starting, pausing and stopping the debugger, for stepping through the debugged code, or for breakpoint management. It also contains panels that display call stack, exception information, or command console. It is implemented in the `ExtBrain.ScriptDevelop.Plugin.Debugger` project.

DebuggerPlugin

The basic debugger functionality is implemented in the singleton `DebuggerPlugin`, which uses the Debugger.Core library (see section 3.1.5) to debug running CLR processes. Its interface contains a lot of properties, methods and events that are used to control the instance of the `NDebugger` class from the Debugger.Core library.

```
public class DebuggerPlugin
{
    public NDebugger Debugger { get; }
    public Process Process { get; }

    public bool IsDebugging { get; }
    public bool IsRunning { get; }
    public bool IsPaused { get; }

    public event EventHandler DebuggingStarted;
    public event EventHandler DebuggingStopped;
    public event EventHandler DebuggingPaused;
    public event EventHandler DebuggingResumed;

    public void Start(ProcessStartInfo processStartInfo,
                      IEnumerable<string> sources,
                      bool breakAtBeginning);

    public void Continue();
    public void Stop();
    public void Break();

    public void StepOver();
    public void StepInto();
    public void StepOut();
}
```

Context

When the debugged process is paused, we use the concept of *current thread*, *current stack frame*, and *current statement* to define the context in which state inspection takes place:

- The debugged process may contain multiple threads, each with its own call stack. The `DebuggerPlugin` allows selecting one thread and then operates in the context of this selected (current) thread.

- The call stack of the current thread may contain multiple stack frames (invoked methods), each with its own set of local variables and current statement position. The `DebuggerPlugin` allows selecting one stack frame and then operates in the context of this selected (current) stack frame.

- Current statement is the statement in the source code that is just being executed or just about to be executed. It depends on the current stack frame (each stack frame has its own current statement) and thus also on the current thread (each thread has its own call stack).

The described concepts of current thread, current stack frame, and current next statement are exposed by the following properties and events of `DebuggerPlugin`.

```
public Thread CurrentThread { get; set; }
public StackFrame CurrentStackFrame { get; set; }
public DomRegion CurrentNextStatement { get; }

public event EventHandler CurrentThreadChanged;
public event EventHandler CurrentStackFrameChanged;
```

Evaluation

When the execution is paused, it is possible inspect the current values of variables or invoke custom commands. This can be done using `DebuggerPlugin.GetValue` method, which is used to implement the console window, the watch window, tooltips showing values of variables, and conditional breakpoints. Evaluation takes place in the context of current thread and current stack frame.

Breakpoints

Breakpoints are places in the source code where execution will be paused automatically. `DebuggerPlugin` contains the following methods for breakpoint management:

```
public Breakpoint AddBreakpoint(string path, int line);
public void RemoveBreakpoint(Breakpoint breakpoint);
```

The `Breakpoint` class extends the underlying breakpoint handle of the debugger library by implementing the following interface:

```
public class Breakpoint
{
    public bool IsEnabled { get; set; }

    public string Path { get; }
    public int Line { get; }
    public int Column { get; }

    public bool IsSet { get; }
    public DomRegion Region { get; }

    public event EventHandler IsSetChanged;
    public event EventHandler<CancelEventArgs> Hit;
}
```

The `IsSet` property is set to `true` once the debugger recognizes the position within the source code (i.e. loads the corresponding symbol files) and then the `Region` property returns the proper text segment covered by the breakpoint.

The `Hit` event is invoked when the breakpoint has been hit by the debugged process, and we can use its `CancelEventArgs` to cancel the pause if we are not interested in this particular occurrence. This is used to implement conditional breakpoints, which evaluate their condition in the `Hit` event (using `DebuggerPlugin.GetValue`) and cancel the pause if the condition is not met.

All breakpoints across all documents are managed by the `BreakpointPlugin` class. Breakpoints for a particular document, including their visual representation, are managed by the `BreakpointManager` class.

Visualizers

Several features of the debugger (value tooltips, watch panel, console panel) need to display to the user values read from the debugged process. The are many ways such values can be displayed, ranging from simple text representations to complex object inspectors that use a tree view. Because of this, the debugger offers the `IVisualizer` interface:

```
public interface IVisualizer
{
    object Visualize(Value value, Thread thread);
}
```

Classes implementing this interface inspect the given value and decide if they want to visualize it. If yes, the `Visualize` method returns an object that can be a simple string or a complex UI element presenting the value. `IVisualizer` implementations are plugged into the application using the `ExportVisualizer` attribute, which also allows specifying the priority of the visualizer (lower-priority visualizer is only used when all higher-priority visualizers return `null`).

Visualizers are managed by the singleton `VisualizerPlugin` plugin, which can be asked for a visualization of the given value using the `Visualize` method. This method iterates over all known visualizers ordered by their priority from the highest to the lowest and returns the first non-null visualization.

There are currently two simple visualizers implemented in the application:

- `PrimitiveValueVisualizer` – Converts any primitive value to string in such a way so that it would look like C# literal, using `TextWriterTokenWriter.PrintPrimitiveValue` method from NRefactory.
- `FallbackToStringVisualizer` – Converts any value to string by invoking its `ToString` method in the debugged process.

Tooltips

When the debugged process is paused, users can inspect current values of variables directly in the source code. When the mouse cursor is hovered over a variable in the source code, its current value is displayed in a tooltip. This is implemented in the `DebuggerTooltipPlugin`. First, the identifier under the mouse cursor is detected and resolved using NRefactory. Then, `DebuggerPlugin.GetValue` is called on the result to get the `Value` instance, which is then visualized by the `VisualizerPlugin` and shown in a tooltip.

Panels

The debugger contains several informational panels that the user can display while debugging. These panels are implemented in the `ExtBrain.ScriptDevelop.Plugin.Debugger.Panels` namespace.

- `ThreadsPanel` – Lists all threads of the currently debugged process, indicates which thread is currently selected, and allows switching of the currently selected thread (`DebuggerPlugin.CurrentThread`).
- `CallStackPanel` – Lists stack frames in the call stack of the currently selected thread, indicates which stack frame is currently selected, and allows switching of the currently selected stack frame (`DebuggerPlugin.CurrentStackFrame`).
- `ExceptionPanel` – Displays details of the current exception, if any. It is automatically displayed if the process has been paused because an exception has been thrown. In this case, the thread where the exception has occurred will have it associated and available for us to retrieve using `DebuggerPlugin.CurrentThreadException`.
- `WatchPanel` – Allows creating one or more watch expressions. These watch expressions are then evaluated each time the process is paused and their values are displayed in a table using `DebuggerPlugin.GetValue` and `VisualizerPlugin`.
- `ConsolePanel` – Offers a command line interface that allows evaluation of custom expressions. When an expression is entered, it is immediately evaluated in the context of the current statement. The resulting value of the expression is then printed out as text. This can be used for deeper inspection of variable values, and also to modify the state of the process since method calls and assignments can also be executed. Entered statements are evaluated using `DebuggerPlugin.GetValue` and `VisualizerPlugin`.

Code completion

The console panel, the watch panel and the breakpoint condition dialog allow entering custom expressions for evaluation. Since these expressions are C# source code fragments that should be valid in their respective contexts, we offer code completion functionality for them. This functionality is based on the general C# code completion described in section 4.7.6, but modified for specific aspects of the debugger evaluator:

- Text boxes for the expressions do not contain complete code but instead only the expression itself which is evaluated in the context of the current statement or breakpoint position. Therefore, we must provide the correct context (position in the source code) to the code completion engine.
- Users may want to use types the namespaces of which are not imported in the source code. Therefore, the code completion offers all known types and inserts their full name upon selection.

Code completion for the debugger evaluator is implemented in the `ExtBrain.ScriptDevelop.Plugin.NRefactory.CodeCompletion.Debug` namespace.

5. Conclusion

The goal of this work was to create a small and easy-to-use tool for using the C# programming language to write scripts. In the work we have analyzed the existing tools and identified their advantages and disadvantages, formulated requirements for our own tool, and developed our own tool.

5.1 Evaluation

We have created an application which satisfies all the requirements R1.1 – R4.8 we defined in section 2.2. The result is a small integrated development environment for quick and easy writing of scripts in the C# language.

The IDE offers sufficient features to allow easy authoring and debugging of programs consisting primarily of a single C# source code file. It makes heavy use of the NRefactory library for syntactic and semantic analysis of the C# source code, and thus offers many code-aware features that simple editors lack, such as correct syntax highlighting, code completion, navigation, or refactoring (most importantly reference and using statement completion). The build system of the IDE is capable of processing stand-alone source code files that include information describing additional compiler dependencies such as external references, and therefore small programs can be developed as single self-contained script files. The IDE also contains basic debugging options which can be used to inspect the execution of the scripts.

The resulting application also offers some additional features which were not originally specified, but are interesting and can be helpful to users:

- A new script file created in the application does not need to be saved first in order to run it or even debug it. This is very convenient when the user only wants to quickly test a code snippet.
- The application is capable of packaging itself into a single stand-alone executable file that contains all the currently loaded plugins, the current configuration file, and all the necessary third-party libraries. The resulting single executable file can then be copied to any system which could run the original application and launched directly from the executable file, without any prior installation or unpacking process.
- The entire application can be built in itself, which means the build system is versatile enough to handle large projects with a lot of source code files, assemblies and dependencies. While this capability was not necessary, and indeed the usage of the application for such a large project is not convenient at all, it provides a useful testing scenario.
- The application features a code analysis subsystem that scans the source code for various issues and offers context-sensitive actions and simple refactoring transformations. This capability was not present among the specified requirements because it was not very important for writing small ad-hoc programs, but since the NRefactory library offered it in an easy-to-integrate way, we have decided to include it.

5.2 Comparison with other tools

In section 2.1.2 we have analyzed the existing tools that allow the usage of the C# language for script development. None of them fully satisfied our ideas for an ideal tool. The tool that came the closest to what we were looking for was LINQPad, which is popular for solving small tasks unrelated to databases (the primary focus of LINQPad). How does our application compare to it?

- Both applications are small and easy to use.
- Both applications allow quick running of simple code snippets or scripts, but LINQPad also allows execution of stand-alone expressions and statements, whereas our application only allows execution of well-formed C# programs. However, this makes sure the scripts in our application are valid C# programs and our application offers the necessary code template with the `Main` method automatically when creating a new script.
- LINQPad uses custom XML format for saving scripts. This format allows storing some additional information about the script, such as its type (if it is an expression, statement, etc.), imported namespaces, and referenced assemblies. Our application saves scripts as valid C# source code and stores the additional information in comments.
- LINQPad's primary focus is on querying databases using the C# language and LINQ. Our application has no such functionality.
- LINQPad executes code in a custom hosting environment and offers additional runtime features, such as the universal "dump" command that allows complex output of any object into a special window. Our application executes scripts as pure stand-alone processes.
- Both applications offer a full-featured C# editor (although in LINQPad most of the important features are available only in the paid version).
- LINQPad does not support code reuse. Our application allows one script to include or reference another.
- LINQPad has no executable generation (the scripts are executed in a special hosting environment and cannot be used outside LINQPad). Our application compiles scripts as pure stand-alone executables.
- LINQPad needs external tools for debugging. Our application contains basic debugging options.

However, during the work on our application, a new version of LINQPad has been released, which added some of the previously missing features:

- A stand-alone command-line tool for execution of LINQPad scripts has been released. This tool can be used instead of executable generation to run scripts outside of LINQPad itself.
- Debugging options similar to our own were introduced, but in the highest paid variant of LINQPad.

5.3 Future work

Our aim was not to create a complete, all-inclusive IDE, but rather a solid and well-usable basis that could be further extended and improved by other students and developers based on their specific requirements, ideas and needs. The primary areas in which the application can be improved include for example:

- **Additional editor features.** The application contains only basic text-editing options. What is missing are features common from advanced text editors such as for example search and replace, navigation between cursor positions, reformatting, etc.
- **Configuration UI.** The application can be configured using a special XML configuration file. To make this process a little more user friendly, the configuration file can be edited from the application itself and is automatically reloaded each time it is saved. However, for most important configuration options a dedicated user interface would be much more convenient.
- **Debugger inspector.** The application offers basic debugging options so that the users could inspect the behavior of their scripts. When inspecting a value of a variable, only simple string representation is shown. More complex objects need to be queried using the debugger console, which is not very convenient. What is missing is a visual object inspector that would make inspecting complex values of objects and collections easier.

During the development of our application, the authors of SharpDevelop announced the final version 5.0, stating that the core team would be shifting their focus from SharpDevelop itself to the universal components that it consists of (e.g. NRefactory, AvalonEdit, ILSpy, etc.) [32]. This news impacts the future development our application since we use a lot of these components, which may now see a faster development pace.

Also, the current development around the C# language (e.g. Microsoft open-sourcing a large portion of its previously proprietary implementation of the C# runtime and libraries [33]) suggests that the popularity of the C# language, and thus tools associated with it, may be further growing in the future.

Bibliography

[1] Xamarin: *The Mono Project*, http://mono-project.com/, online 28. 4. 2015

[2] Microsoft: *Visual Studio*, http://www.visualstudio.com/, online 28. 4. 2015

[3] IC#Code: *SharpDevelop*, http://www.icsharpcode.net/OpenSource/SD/Default.aspx, online 28. 4. 2015

[4] Xamarin: *MonoDevelop*, http://monodevelop.com/, online 28. 4. 2015

[5] Microsoft: *.NET Compiler Platform ("Roslyn")*, MSDN, http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx, online 28. 4. 2015

[6] IC#Code: *Repository for NRefactory 5*, GitHub, http://github.com/icsharpcode/NRefactory, online 28. 4. 2015

[7] Grunwald D.: *Using NRefactory for analyzing C# code*, CodeProject 2012, http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code, online 28. 4. 2015

[8] Grunwald D.: *Using AvalonEdit (WPF Text Editor)*, CodeProject 2013, http://www.codeproject.com/Articles/42490/Using-AvalonEdit-WPF-Text-Editor, online 28. 4. 2015

[9] Srbecký D.: *Internals of SharpDevelop's Debugger*, http://community.sharpdevelop.net/blogs/dsrbecky/archive/2010/07/29/debugger.aspx, online 28. 4. 2015

[10] Koníček M.: *Debugger Visualizers for the SharpDevelop IDE*, 2011, Master thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Software Engineering, http://artax.karlin.mff.cuni.cz/~konim5am/thesis/MartinKonicek_DebuggerVisualizers.pdf, online 28. 4. 2015

[11] Karásek J.: *Prostředí pro snadný vývoj skriptů*, 2012, Bachelor thesis, Czech Technical University in Prague, Department of Cybernetics, http://www.extbrain.net/theses/bp-jiri-karasek.pdf, online 28. 4. 2015

[12] Vavilala, R. K.: *NScript – A script host for C#/VB.NET/JScript.NET*, CodeProject 2002, http://www.codeproject.com/Articles/3207/NScript-A-script-host-for-C-VB-NET-JScript-NET, online 28. 4. 2015

[13] Shilo O.: *CS-Script – The C# Script Engine*, http://www.csscript.net, online 28. 4. 2015

[14] Block G., Rusbatch J., Wojcieszyn F.: *scriptcs*, http://scriptcs.net/, online 28. 4. 2015

[15] Albahari J.: *LINQPad*, http://www.linqpad.net/, online 28. 4. 2015

[16] Microsoft: *Microsoft .NET Framework 4 (Standalone Installer)*, http://www.microsoft.com/en-us/download/details.aspx?id=17718, online 28. 4. 2015

[17] IC#Code: *ILSpy – .NET Decompiler*, http://ilspy.net/, online 28. 4. 2015

[18] Microsoft: *Command-line Building With csc.exe*, MSDN, http://msdn.microsoft.com/en-us/library/78f4aasd.aspx, online 28. 4. 2015

[19] Microsoft: *SQL Server Management Studio*, MSDN, http://msdn.microsoft.com/en-us/library/hh213248.aspx, online 28. 4. 2015

[20] Microsoft: *Windows Forms*, MSDN, http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx, online 28. 4. 2015

[21] Microsoft: *Windows Presentation Foundation*, MSDN, http://msdn.microsoft.com/en-us/library/ms754130.aspx, online 28. 4. 2015

[22] IC#Code: *AvalonEdit*, http://avalonedit.net/, online 28. 4. 2015

[23] Community: *ScintillaNET*, Codeplex, http://scintillanet.codeplex.com/, online 28. 4. 2015

[24] Community: *Scintilla: A free source code editing component for Win32, GTK+, and OS X*, http://scintilla.org/, online 28. 4. 2015

[25] Community: *AvalonDock*, Codeplex, http://avalondock.codeplex.com/, online 28. 4. 2015

[26] Microsoft: *Debugging Interfaces*, MSDN, http://msdn.microsoft.com/en-us/library/ms404484.aspx, online 28. 4. 2015

[27] Microsoft: *Add-ins and Extensibility*, MSDN, https://msdn.microsoft.com/en-us/library/bb384241.aspx, online 28. 4. 2015

[28] Community: *Managed Extensibility Framework*, Codeplex, http://mef.codeplex.com/, online 28. 4. 2015

[29] Xamarin: *Cecil*, http://mono-project.com/Cecil, online 28. 4. 2015

[30] Microsoft: *Application Settings for Windows Forms*, MSDN, http://msdn.microsoft.com/en-us/library/0zszyc6e.aspx, online 28. 4. 2015

[31] Apache: *Apache log4net*, http://logging.apache.org/log4net/, online 28. 4. 2015

[32] Wille, C.: *SharpDevelop 5.0 Final*, SharpDevelop Community Blog, http://community.sharpdevelop.net/blogs/christophwille/archive/2014/10/28/sharpdevelop-5-0-final.aspx, online 28. 4. 2015

[33] Landwerth, I.: *.NET Core is Open Source*, .NET Blog, MSDN, http://blogs.msdn.com/b/dotnet/archive/2014/11/12/net-core-is-open-source.aspx, online 28. 4. 2015

A. Content of the CD

The CD enclosed to this thesis contains the following directories and files:

- `bin` – Directory with the compiled executable binaries of the application (for installation instructions, see section B.1).
- `doc` – Directory with the documentation generated from the source code of the application.
- `redist` – Directory containing *Microsoft .NET Framework 4* [16], which is a prerequisite for running the application.
- `src` – Directory with the source code of the application.
  - `src/Libraries` – Source code of the modified libraries that the application uses.
  - `src/Libraries/readme.txt` – Text document describing the modified libraries (for details, see beginning of chapter 4).
  - `src/ExtBrain.ScriptDevelop` – Source code of the application itself.
  - `src/ExtBrain.ScriptDevelop/ExtBrain.ScriptDevelop.csx` – Script file that will build the application from inside the application itself, placing the resulting binaries into the `src/ExtBrain.ScriptDevelop/Bin` directory.
- `example` – Directory with a few example scripts that can be opened and run in the application.
- `previous` – Directory containing the source code of the original ScriptDevelop [11], upon which our application has been based (see section 3.2).
- `thesis` – Directory with the text of this thesis in the PDF format.
- `readme.txt` – Text document describing the content of the enclosed CD.

B. User guide

This appendix describes the application from the user's perspective. It details how to install and use the application.

B.1 Installation

The application does not require any special installation. It can be copied, along with all of the other files in its program directory, to any machine that has a Microsoft .NET Framework 4 [16] installed. On the CD attached to this thesis, you can find the binaries of the application in the `bin` directory and Microsoft .NET Framework 4 in the `redist` directory. Also please note that the application should have write access to the directory it is launched from (because of log and configuration files), so it is not recommended to launch it directly from the enclosed CD.

Aside from the executable file itself, the application contains a lot of assembly files in its directory. All these files are required for the application to function properly and must be copied when installing the application to a new location.

To simplify manipulation and deployment, the application can package itself into a single stand-alone executable file that contains all the currently loaded plugins, the current configuration file, and all the necessary third-party libraries. The resulting single executable file can then be copied to any system that could run the original application and launched directly from the executable file, without any installation or unpacking process.

This packaging process can be initiated from the main menu by selecting `File` → `Export Application`. After the application is packaged, the user will be prompted to save the resulting executable to a new location.

B.2 Quickstart guide

In order to quickly start working on a new script, select `File` → `New` → `C# Script` from the main menu. An editor with a new script appears, its source code already containing the following template:

```
using System;

/*HEADER**
    PAUSE
**HEADER*/

class Program
{
    static void Main(string[] args)
    {
    }
}
```

This is a standard C# console application with the `Main` method into which you can start writing your own code. You can immediately compile and run your script using `Build` → `Run` (or `Ctrl+F5`) without having to name and save it first. You can even debug your untitled script right away (`Debug` → `Run`, or `F5`).

Header sections

In the script template shown above, please note the comment block above the class definition. It looks like this:

```
/*HEADER**
    PAUSE
**HEADER*/
```

This comment block is called a *header section* and its purpose is to control the way the script is compiled and executed. It consists of special *header commands* (described further in this chapter), one command per line. For example, the `PAUSE` command, which is already present in the template, causes the console window with the script to pause and display "Press any key to continue..." just before it exits, provided it has been launched from the application (it does not pause if it has been launched separately via the compiled executable file). There are also similar commands for specifying working directory and command line arguments, among others.

References and using statements

The most common and useful header command you will encounter is the `REF` command, which is used to reference another assembly, including library assemblies of the framework. However, in most cases you do not need to specify it manually. For example, if you want to use methods from the LINQ library, you can just start typing the name of the method, e.g. `Any`. At this point, the word `Any` will be shown in red and code completion will offer no options, as can be seen in figure B.1.

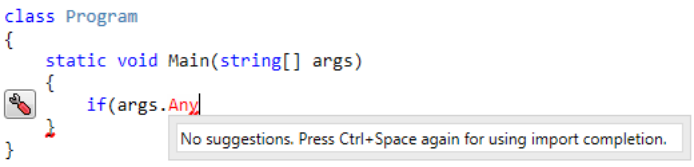


Figure B.1: Unknown identifier example.

There are now two options:

a) You can press `Ctrl+Space` two more times to open *reference completion*, which will include items from unreferenced assemblies (see figure B.2), and select the correct method from it.

b) You can click the red socket wrench icon at the beginning of the line. This icon indicates that there are some issues with possible automatic fixes on the line. Clicking it (or pressing `Alt+Enter`) will open a menu with possible fixes (see figure B.3). You can select the correct method from it.

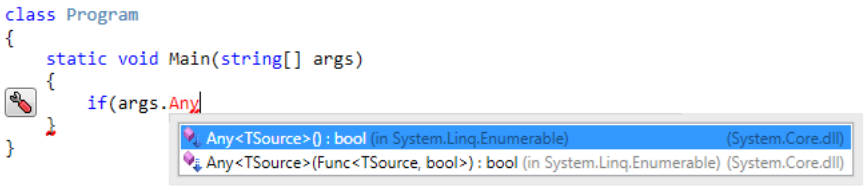


Figure B.2: Reference completion example.

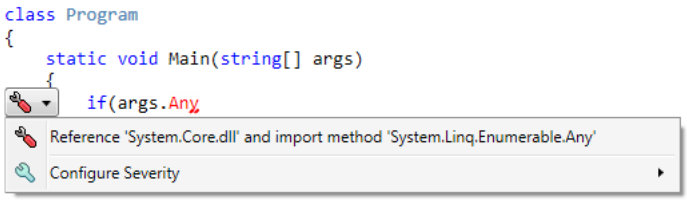


Figure B.3: Reference context action example.

Both actions will result in the correct reference and `using` statement being added to the source code, which will now become like this (please note that "`using System.Linq`" and "`REF System.Core.dll`" have been inserted):

```
using System;
using System.Linq;

/*HEADER**
    PAUSE
    REF System.Core.dll
**HEADER*/

class Program
{
    static void Main(string[] args)
    {
        if(args.Any
    }
}
```

This way you can quickly use types and extension methods that reside in namespaces which have not been imported yet with a `using` statement, or whose assembly is not yet referenced. Also please note that the application must know somehow which assemblies to offer for referencing. The list of such assemblies can be configured, as described further in this chapter.

B.3 Basics

The application is laid out like a standard text editor with a tabbed interface. The main window, which can be seen in figure B.4, consists of the main menu and toolbar at the top, informational status bar at the bottom, and a document view in the central area. Multiple documents can be open at the same time and the

user can switch between them using their tab headers at the top. The document view contains a text editor, whose exact functionality depends on the type of document that is currently open.

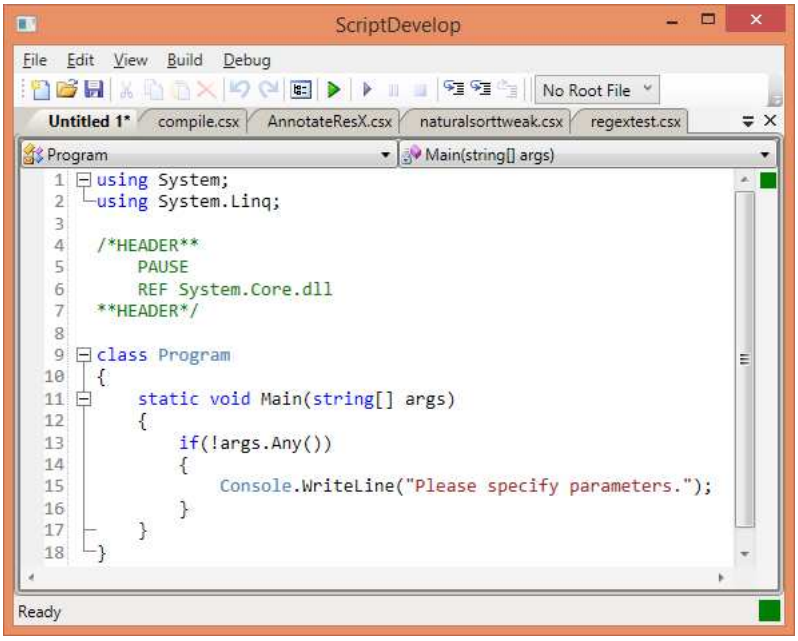


Figure B.4: The main window of the application.

There are many commands available from the main menu. If a command has a keyboard shortcut, it is displayed next to the command in the main menu. Some commands are available also as buttons in the toolbar. These commands use the same icon for the button as in the main menu and their name and keyboard shortcut are displayed in a tooltip when the mouse cursor is hovered over the button.

Not all commands are always visible. Some commands appear only in certain contexts, for example when debugging is active or when a C# document is active.

The `File` menu contains common commands for creating, opening, saving, and closing documents. Documents can be additionally opened by dragging them to the application's window or by specifying their paths in the application's command line. If the application is closed and there are documents that are still open, they will be opened again when the application is next started. If the application is terminated unexpectedly and there are documents that contain unsaved modifications, they will be restored when the application is next started.

Supported formats

The application recognizes three types of documents based on their extension:

1) *Text document* – A generic plain text document format that is used as a fallback when no other known format could be detected. The editor offers no additional functionality to text documents besides simple editing.

2) *XML document* – For `.xml`, `.xaml`, `.cfg` and `.config` extensions. The editor offers syntax highlighting and code folding (the ability to expand and collapse individual elements) to XML documents.

3) *C# document* – For `.cs` and `.csx` extensions. The editor offers full functionality to C# documents.

The application is primarily designed to work with C# source code files. Other document types are supported only for convenience. The `.csx` extension is used by default to distinguish stand-alone C# scripts created in the application from other C# source code files.

Encoding

The application uses UTF-8 encoding with byte order mark (BOM) to save documents. When opening documents, ASCII, other Unicode encodings, or the system's default encoding may also be used.

B.4 Source code editor

The application is primarily designed to work with C# source code files, for which the source code editor supports a lot of additional functionality. This section describes how to use the features specific to C#.

Basics

The `Edit` menu contains common commands for history and clipboard management. Commands specific to C# include:

- `Comment Selection`, `Uncomment Selection`, `Toggle Comment` – Comments or uncomments the selected lines using line comments (double slashes).
- `Rename` – If the text cursor is placed over a named symbol (variable, type, member, namespace), opens a window that allows specifying a new name for the symbol and renames all references of the symbol accordingly.

Code completion

While the user types identifier names, code completion offers a list of available types, members, variables and other constructs. It allows to choose an identifier without the need to remember its exact name or without having to type out its name in full.

Code completion appears automatically when the user types an identifier, or it can be invoked manually by pressing `Ctrl+Space`. It operates on three levels:

1) Basic completion – Offers all known symbols that can be used at the current position.
2) Import completion – Offers types and extension methods from namespaces that are not currently imported with a `using` statement and imports the namespaces accordingly when an item is selected.
3) Reference completion – Offers types, extension methods and namespaces from yet unreferenced but otherwise known assemblies, and automatically references the correct assembly when an item is selected.

The first level is the default. The user can switch to the next level by pressing `Ctrl+Space` again.

Parameter hints

Parameter hint is a popup window that appears when the user writes a method invocation. It displays the list of invoked method's overloads, the list of parameters for the currently selected overload, and the current parameter.

Parameter hint appears automatically when the user types a method invocation, or can be invoked manually by pressing `Ctrl+Shift+Space`. When the window is displayed, the user can switch between available overloads by pressing `Up` and `Down` keys.

Navigation

The `View` menu contains commands that allow quick navigation through the C# source code:

- `Go To Type` – Displays a list of all known types and when a type is selected, navigates to its definition.
- `Go To Member` – Displays a list of all known members and when a member is selected, navigates to its definition.
- `Go To Definition` – If the text cursor is placed over a named symbol (variable, type, member), navigates to its definition. If it is a library entity (an entity defined in an external assembly) and the path to ILSpy [17] is properly configured, displays the entity's decompiled source code in ILSpy.

Navigation bar

The navigation bar is the row on the editor's top side. It contains two selection boxes. The first one lists all types defined in the current source code file, the second one lists all members defined in the currently selected type. When the text cursor is moved within the source code, the selected values in the boxes are updated to indicate what type and member the text cursor is currently at. Selecting different values from the lists will move the text cursor to the beginning of the selected type and member in the source code, so the navigation bar can be used to quickly locate types and members in longer files.

Code issues

As the user types source code, the application continuously analyses it and detects issues (errors, warnings, suggestions). The detected issues are presented to the user by underlining or coloring the related source code. For example, source code that is detected to be redundant is displayed in gray, and unknown identifiers are displayed in red. Hovering the mouse cursor over such highlighted source code will display the issue's description in a tooltip.

Context actions

For many places in the source code, the application can offer various small refactoring transformations, and for many issues, the application can offer automatic actions to fix them. For example, source code that was marked as redundant can be quickly removed. When the text cursor is placed on a location where such actions are available, an icon is displayed at the beginning of the line. Clicking this icon (or pressing `Alt + Enter`) will open a menu with the available actions.

Shortcut column

The shortcut column is the column on the editor's right side, beyond the scroll bar. It displays small colored horizontal bars that represent issues (errors, warnings, suggestions) found in the current source code file. The column's whole height represents the entire file, so bars at the top indicate issues at the beginning of the file and bars at the bottom indicate issues at the end of the file. Hovering the mouse cursor over the bar will display the issue's description in a tooltip, clicking the bar will scroll the text cursor to the issue's location.

The small colored square at the top of the shortcut column indicates if there are any errors or warnings in the current source code file. It is red if there are errors, orange if there are warnings but no errors, and green if there are no errors or warnings. Clicking the square will cycle the text cursor over the most severe issues found in the current source code file.

Reference highlighting

When the text cursor is placed over a named symbol (variable, type, member, namespace) in the source code, all references of the same symbol are highlighted in the document for easier orientation. Alternatively, the user can manually highlight all references of the symbol under the text cursor using the `View` → `Reference Highlighting` → `Highlight References` command. There is also the `View` → `Find References` command that will list the symbol's references across multiple documents in a separate window.

B.5 Compiling and running

Scripts open in the application can be compiled using commands in the `Build` menu:

- `Run` – Compiles the script and launches it.
- `Build` – Compiles the script without launching.
- `Rebuild` – Compiles the script without launching, forcing compilation even when the output assemblies are up to date.
- `Save Assembly As` – Compiles the script and prompts the user to save the output assembly to a new location.

These commands will save the script first if it contains any unsaved modifications. The only exception to this are newly created unnamed scripts that were never saved yet. Such scripts can be run without having to save them first.

By default, scripts are compiled into a temporary directory, from which the output executable is launched. This can be changed using header commands described in the next section. Header commands can also be used to specify working directory and command line arguments of the script launched from the application.

Errors

If the compilation process fails due to errors in the source code, a window listing the errors is displayed and the errors are highlighted in the source code using a red underline. The application also offers automatic error checking which can be toggled in the `Build` → `Automatic Error Checking` menu:

- `Continuous` – Errors are checked each time the source code has been changed in the application, even when the changes were not saved yet.
- `After Save` – Errors are checked each time the script has been saved.
- `Off` – Errors are not checked automatically. A check can be invoked manually using the `Build` → `Check For Errors` command, which works even on unsaved scripts.

The colored square in the bottom-right corner of the application's main window indicates if there are any errors in the current script. It can be green (no errors), red (at least one error), or gray (error checking is in progress).

Root file

Each script open in the application is considered stand-alone by default. This means that all build-inducing commands such as `Run` are by default executed upon the currently active (focused) script. However, if an open script includes or references another open script, the application tries to detect their dependency hierarchy and act upon the top-level script (the referencing one) instead even when a lower-level script (the referenced one) is currently active.

If necessary, users can override this behavior by explicitly specifying the *root file*. All the build-inducing commands are then executed upon this root file no matter what document is currently active. The root file is specified globally for the whole application and can be set or unset using the `Build` → `Root File` menu, the selection box in the toolbar, or by toggling `Root File` in the document's context menu available by right-clicking the document's tab header.

Command line compilation

Scripts can be also compiled using the command line. Any command line argument of the application in form of `/build:`*`path`* will cause the application not to display its user interface, but instead to compile the script file denoted by *`path`*. If the compilation is successful, the application exits immediately with an exit code

0. If there were any errors, the application prints the errors to `stdout` and exits with an exit code 1. In batch files it is recommended to run the application using `start /wait` command so that the execution would pause until the compilation is finished.

B.6 Header sections

In order to correctly compile a source code file, the C# compiler needs additional information, such as the list of referenced assemblies. In classic IDEs, this information is associated with the project and stored in the project file. In our application, this information is stored in the source code files themselves using special comment blocks called *header sections*. A header section looks like this:

```
/*HEADER**
    ...
**HEADER*/
```

There can be multiple header sections in a single file. A single header section consists of any number of *header commands*. Each header command must be placed on a separate line. The first word of the line identifies the command, the rest of the line is the command's optional parameter. If the line is empty or begins with double slashes, it is ignored. For example, a header section can look like this:

```
/*HEADER**

    PAUSE
    OUT Hello world.exe

    // References
    REF System.dll
    REF System.Core.dll

**HEADER*/
```

B.6.1 Available commands

The available header commands are listed in table B.1 and described in this section.

OUT

The `OUT` command specifies the path of the output assembly. If omitted, the output assembly will be generated in a temporary directory when run from the application.

Command	Parameter	Summary
OUT	Path	Path of the output assembly.
TARGET	Target	Type of the output assembly.
VER	Version	Compiler version.
MAIN	Type	Main class.
OPT	Option	Additional compiler option.
INC	File(s)	Includes additional source files.
REF	Reference(s)	References assemblies.
IMP	Reference(s)	References and includes assemblies.
BUILD	File(s)	Compiles additional scripts.
EMB	File(s)	Includes embedded resources.
LNK	File(s)	Includes linked resources.
RES	File(s)	Includes WPF resources.
DIR	Directory	Working directory of the script.
ARGS	Arguments	Command line arguments of the script.
PAUSE	None	Pauses the script before exiting.

Table B.1: Summary of the available header commands.

TARGET

The `TARGET` command specifies the type of the output assembly (i.e. the "/target" compiler option). Expected values are:

- `EXE` – a console application (default)
- `WINEXE` – a windows application
- `LIBRARY` – a class library (dll)

VER

The `VER` command specifies which compiler version to use. Default is "v4.0".

MAIN

The `MAIN` command specifies the full name of the class with the `Main` method in the output executable. It needs to be used if there is more than one such class.

OPT

The `OPT` command adds its parameter as a separate option into the compiler's command line. It can be used to pass additional options to the compiler.

INC

The `INC` command specifies another source code file to be included when compiling the output assembly. From the compiler's perspective, this behaves as if there were multiple source code files contributing to a single output assembly. The included source code files are also scanned for header sections and the commands inside them are processed too.

REF

The `REF` command references an assembly. It can also be used to reference another script file. From the compiler's perspective, this means that the referenced script will be compiled first and the resulting assembly will be used as a reference of the current assembly.

IMP

The `IMP` command references an assembly in the same way as the `REF` command, but then embeds the referenced assembly file into the output executable and includes code that will load it correctly when needed. The referenced assembly does not need to be distributed with the output executable separately. This feature is experimental and can only be used when compiling executable binaries.

BUILD

The `BUILD` command compiles the specified script file as a post-processing step, after building of the current script is complete.

EMB

The `EMB` command includes the specified file into the output assembly as an embedded resource.

LNK

The `LNK` command includes the specified file into the output assembly as a linked resource.

RES

The `RES` command includes the specified file into the output assembly as a WPF resource.

DIR

The `DIR` command specifies the working directory in which the script will be launched. The specified directory is only used when the script is launched from the application.

ARGS

The `ARGS` command specifies the command line arguments with which the script will be launched. The specified arguments are only used when the script is launched from the application.

PAUSE

The `PAUSE` command specifies that the script should pause and display "Press any key to continue..." just before it exits. The script will be paused like this only if it is running in a console window and has been launched from the application.

B.6.2 Parameters

Many header commands expect a path to a file as a parameter. The path to the file can be expressed as an absolute path, for example:

```
INC C:\Work\Scripts\Common.csx
```

It can also be a path relative to the directory of the current file (the file with the command), for example:

```
INC Common.csx
```

Wildcards

Most commands can be applied multiple times to different files. Paths passed to these commands can contain a wildcard character `*` in their file name. In this case, all the files found in the target directory that match the mask will be passed to the command. For example, the following command will include all files with the `csx` extension in the `Common` directory:

```
INC Common\*.csx
```

The wildcard character `*` can be also used as the name of the last directory in the command's parameter. In this case, all subdirectories are also recursively searched for the specified file. For example, the following command will include all files with the `csx` extension in the `Common` directory and also all its subdirectories:

```
INC Common\*\*.csx
```

Excluding files

Sometimes it is useful to use a wildcard to include a lot of files except a select few. For this purpose, a hyphen before the parameter indicates that the file should be excluded. For example, the following commands will include all files with the `csx` extension in the `Common` directory, except `other.csx`:

```
INC Common\*.csx
INC - Common\other.csx
```

The decision process takes place after all header commands have been parsed, so the order of the commands has no significance.

References

The header commands `REF` and `IMP` expect a reference as a parameter. By default, the parameter is processed in the same way as file parameters described above and can result either in an assembly file which is used directly, or a script file which is compiled first and then the resulting assembly is used as the reference. However, if no file is found using the processing described above, the parameter value is passed to the compiler as a reference directly. This way, standard libraries can be referenced without knowing their full path.

B.7 Debugging

The application contains a debugger that behaves similarly to debuggers in classic IDEs. The debugger in the application supports breakpoints, stepping, and expression evaluation.

B.7.1 Execution control

Debugging is controlled using the commands in the `Debug` menu. The current script is compiled and started in the debugger by the `Run` command, paused by the `Break` command, resumed by the `Continue` command and terminated by the `Stop` command. Debugging can also be started by the `Step Into` command, in which case the script will be paused at the beginning, or `Run To Cursor` command, in which case the script is paused when the control reaches the statement at the text cursor's position.

Stepping

The debugger contains several commands that resume the execution in a limited way:

- `Step Over` – Executes the current statement and pauses again at the next statement in the same method.
- `Step Into` – Executes the current statement and pauses again at the next statement in the same method, or at the beginning of a method invoked while executing the current statement, if possible.
- `Step Out` – Continues execution of the current method and pauses again at the next statement in the method that invoked the current method.
- `Run To Cursor` – Continues execution until a statement at the text cursor's position is reached.

Breakpoints

Breakpoints are places in the source code where execution will be paused automatically. They can be placed on the current line using the `Debug` → `Toggle Breakpoint` command or by clicking in the left margin of the editor. If a breakpoint already exists at the selected line, it will be removed. The breakpoint's context menu (available by right-clicking on the breakpoint in the source code or on its icon in the left margin of the editor) contains commands to delete, disable or enable the breakpoint, and allows to set the breakpoint's condition, which is a boolean expression that is evaluated each time the breakpoint is reached to see if the script should be paused there or not.

B.7.2 State inspection

When the script is paused, its current state can be inspected as described in this section.

Current thread and stack frame

The debugged script process may contain multiple threads, each with its own call stack. The debugger allows the user to select one thread and then operates in the context of this selected thread. Similarly, the current thread's call stack may contain multiple stack frames (invoked methods), each with its own set of local variables and current statement position. The debugger allows the user to select one stack frame and then operates in the context of this selected stack frame.

Current statement

The current statement is the statement in the source code that is just being executed or just about to be executed. It depends on the current stack frame (each stack frame has its own current statement) and it is highlighted in the source code in different colors:

- Yellow – The highlighted statement has not been executed yet, but its execution is just about to start. There is no stack frame above the current one in the call stack of the current thread.

- Green – The highlighted statement is just being executed, but its execution has not finished yet. There is at least one more stack frame above the current one in the call stack of the current thread.

Variable values

When the script is paused, users can inspect current values of variables. When the mouse cursor is hovered over a variable in the source code, its current value is displayed in a tooltip. More complex inspections can be performed using the watch and console windows described in the following section.

B.7.3 Windows

There are several windows in the application that are available only when the debugger is active. These windows can be displayed using respective commands in the `Debug` menu or using the corresponding toolbar buttons.

Threads

The `Threads` window lists all threads of the currently debugged script, indicates which thread is currently selected, and allows switching of the currently selected thread.

Call Stack

The `Call Stack` window lists stack frames (method calls) in the call stack of the currently selected thread, indicates which stack frame is currently selected, and allows switching of the currently selected stack frame.

Exception

The `Exception` window contains details of the current exception. It is automatically displayed if the script was paused because an exception has been thrown.

Watch

The `Watch` window allows creating one or more watch expressions. These watch expressions are then evaluated each time the script is paused and their values are displayed in a table.

Console

The `Console` window offers a command line interface that allows evaluation of custom expressions. When an expression is entered, it is immediately evaluated in the context of the current statement. The resulting value of the expression is then printed out as text. This can be used for deeper inspection of variable values, and also to modify the state of the script since method calls and assignments can also be executed.

B.8 Configuration

The application is configured using an external XML configuration file. The configuration file must be placed in the same directory as the application's executable file and it must have the same name, but the `.cfg` extension instead of `.exe`. If there is no configuration file, the application uses the default configuration file stored internally. This internal configuration file contains the default configuration of the application, and so users are not required to create or edit the configuration file if they are satisfied with the current configuration.

The application contains no specialized user interface for modifying configuration. However, the configuration file can be edited in the application itself the same way as any other XML file. The application detects changes to the configuration file and applies them as soon as the modified file is saved.

The current configuration file can be opened in the editor using the `Edit` → `Preferences` command. If the file does not exist yet, it is created as a copy of the default configuration file that is stored internally.

Also note that the current configuration file will be embedded into the application if the user packages the application into a stand-alone executable, as described in section B.1.

B.8.1 Configuration sections

The configuration file is formatted as an XML document whose root node contains one element for each separate configuration section. The exact format of these elements depends on the section and is described in the following text.

Logging configuration

The `Logging` configuration section allows configuring the path where the application writes its log file. It looks like this:

```
<Logging>
    <Path>log.txt</Path>
</Logging>
```

By default, the application writes the log to a file with the same path as the application's executable file, but the `.log` extension instead of `.exe`. Alternatively, the `Path` element can be left empty, in which case the log is not written anywhere.

Text editor configuration

The `TextEditor` configuration section allows configuring the editor's font. It looks like this:

```
<TextEditor>
    <FontFamily>Consolas</FontFamily>
    <FontSize>14</FontSize>
</TextEditor>
```

Plugin configuration

The `Plugins` configuration section specifies what plugins will be loaded at the application's startup. It looks like this:

```
<Plugins>
    <Plugin Name="ExtBrain.ScriptDevelop.Plugin.*.dll" Delay="0" />
</Plugins>
```

The `Delay` attribute specifies time in milliseconds after the application's startup that the plugins specified in the element will be loaded.

Precached assemblies configuration

The `PrecachedAssemblies` configuration section specifies assemblies that will be loaded in advance at the application's startup. It looks like this:

```
<PrecachedAssemblies>
    <Assembly>System.dll</Assembly>
    <Assembly>System.Core.dll</Assembly>
    <Assembly>System.Xml.dll</Assembly>
    <Assembly>System.Xml.Linq.dll</Assembly>
    <Assembly>System.Drawing.dll</Assembly>
</PrecachedAssemblies>
```

Types, extension methods and namespaces from these assemblies will be offered in code completion and context actions even when their respective assemblies are not referenced yet in the script.

Keyboard shortcuts configuration

The `KeyBindings` configuration section allows overriding the keyboard shortcut of any command in the application. It looks like this:

```
<KeyBindings>
    <KeyBinding Command="GoToTypeCommand" Shortcut="Ctrl+T" />
</KeyBindings>
```

The `Command` attribute specifies the short or full name of the class that implements the command. The `Shortcut` attribute specifies the overridden keyboard shortcut for the command. The shortcut is parsed using the `KeyGestureConverter` class of WPF, so it follows the same rules. Multiple shortcuts can be specified and separated by semicolons.

Code analysis configuration

The `CodeAnalysis` configuration section allows overriding the default severity of any code analysis provider in the application. It looks like this:

```
<CodeAnalysis>
    <OverrideSeverity Provider="RedundantUsingDirective"
                      Severity="Suggestion" />
    <OverrideSeverity Provider="InvokeAsExtensionMethod"
                      Severity="None" />
    <OverrideSeverity Provider="InconsistentNamingIssue"
                      Severity="None" />
</CodeAnalysis>
```

The `Severity` attribute can be set to one of the following values:

- `None` – The provider will be disabled.
- `Hint` – The issues found by the provider will be indicated only by a subtle short underline and will not be shown in the shortcut column.
- `Suggestion` – The issues found by the provider will be indicated by a green color.
- `Warning` – The issues found by the provider will be indicated by an orange color and will count as warnings in the shortcut column.
- `Error` – The issues found by the provider will be indicated by a red color and will count as errors in the shortcut column.

The `Provider` attribute specifies the name of the code analysis provider. The provider names can be found out in the application by placing the text cursor upon an issue and choosing `Configure Severity` from the context actions menu. This will open the configuration file and copy the respective `OverrideSeverity` XML element into the clipboard.

Code formatting configuration

The `CSharp` configuration section allows overriding the default formatting of the C# source code that is used when performing refactoring transformations. It looks like this:

```
<CSharp>
    <CSharpFormattingOptionsPreset>
        Allman
    </CSharpFormattingOptionsPreset>
</CSharp>
```

The value of the `CSharpFormattingOptionsPreset` element can be one of the following values: `Allman`, `GNU`, `KRStyle`, `Mono`, `SharpDevelop`, or `Whitesmiths`. These values correspond with the presets defined in the `FormattingOptionsFactory` class of the NRefactory library. Alternatively, `TextEditorOptions` and `CSharpFormattingOptions` elements can be used instead of the `CSharpFormattingOptionsPreset` element to specify custom formatting. The structure of these elements is defined by the `TextEditorOptions` and `CSharpFormattingOptions` classes of the NRefactory library (the standard `XmlSerializer` is used to deserialize them).

ILSpy configuration

The `ILSpy` configuration section allows specifying the path to the ILSpy [17] executable. It looks like this:

```
<ILSpy>
    <Path>ILSpy\ILSpy.exe<Path>
</ILSpy>
```

The path can be either absolute, or relative to the application's executable. If ILSpy is properly configured, the `Go To Definition` command can be used to navigate to an external entity's decompiled source code in ILSpy.

C. Extension points

This appendix provides an overview of the most useful extension points in the application. These extension points allow quickly extending the functionality of the application without having to modify the existing source code base.

Plugin

To add a new plugin that needs to be initialized upon startup to the application, implement the `IPluginBase` interface. For details, see section 4.2.1.

Command

To add a new command into the application, implement the `IExtendedCommand` interface and export it using the `ExportCommand` attribute. Commands can be also added to the main menu, to the toolbar, and to various context menus using the `MenuCommand` and `ToolbarCommand` attributes. For details, see section 4.3.2.

There are several helper `IExtendedCommand` implementations that can be derived from, for example:

- `BaseCommand` – Basic implementation, also implements `IPluginBase` for initialization.
- `SimpleCommand` – Basic implementation, exposes methods without the `parameter` parameter.
- `DockableContentCommand<T>` – Implements a command for showing and hiding a dockable panel.
- `DocumentCommand<TFormat, TDocument>` – Implements a command that acts upon a document of a specific format and type.

Document extension

A new functionality can be added to every document open in the application by implementing the `IEditorExtensionPlugin` interface. For details, see section 4.4.2.

Dockable panel

A new dockable panel can be added to the application by inheriting from the `DockableContentPlugin` class.

Configuration section

A new section can be added to the configuration file by inheriting from the `ConfigurationPlugin` class. For details, see section 4.2.2.

Persisted settings

A new persisted settings object can be added to the application by inheriting from the `SettingsPlugin` class. For details, see section 4.2.2.

Main window component

A new component can be added to the main window by implementing the `IMainWindowComponentPlugin` interface.

Status bar component

A new component can be added to the status bar by implementing the `IStatusBarComponentPlugin` interface.

Code analysis provider

A new code issue or code action provider can be added to the application by implementing the `ICodeIssueProvider` or `ICodeActionProvider` interface. For details, see section 4.7.2.

Code completion provider

New entries to the code completion window can be added by implementing the `ICompletionSourceProvider` interface. For details, see section 4.4.4.

Debugger visualizer

New debugger visualizers can be added by implementing the `IVisualizer` interface and exporting it using the `ExportVisualizer` attribute. For details, see section 4.8.