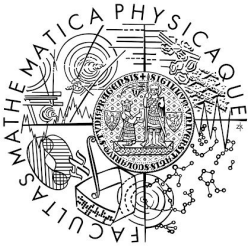

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

C# Base Language for MPS

(CS4MPS)

Specification

Team members:
Tomáš Eliáš, Roman Firment, Jakub Saksa, Martin Wirth, Dalibor Zeman
Faculty of Mathematics and Physics
Charles University

Supervisor:
RNDr. Pavel Parízek, Ph.D.
Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University

Consultant:
Mgr. Václav Pech
JetBrains

2019

Table of contents

1 Introduction

In modern software development, it is generally preferred to use as high-level instruments or abstractions as possible. One of the most used abstractions is a programming language. The taxonomy of programming languages consists of many kinds of classifications. One of them speaks about Domain-Specific Languages (DSL). Their main characteristic is that they concentrate on a given application domain, allowing the developer to ignore programming-specific details.

To be able to use a DSL, one must be equipped with the language definition, a code editor and a way to compile code or to interpret it. There is no standardized approach for these aspects. However, the JetBrains MPS tool provides a unified solution which targets all the aspects, in a single environment.

In the MPS tool, the end-user can write DSL code which is then automatically transformed into code of a well-known general-purpose programming language like Java. Our project aims to provide support for a transformation into the C# programming language.

The following sections introduce the context in a greater level of detail.

1.1 Domain-Specific Languages

A domain-specific language is a language which is intended to be used for a defined application domain. A DSL is usually easy to use for that given domain whereas hard or impossible to use for other domains. This is the main difference from a general-purpose language (GPL), which can be used for more-or-less any domain. The division line between both kinds of languages is a little blurry: deciding whether a language is a DSL or a GPL is often a matter of a viewpoint.

In this project we understand the term language as a programming language even though there are other kinds of languages which are also related to this topic but out of scope of this project.

To help the reader better understand the term DSL, we present a set of examples of both, DSLs and GPLs. Languages like Java, C#, C++ or Python are representatives of GPLs. One can use them to develop almost any program. On the contrary, languages for educating children in programming and MATLAB are representatives of DSLs. Some might even consider shell scripting languages to be DSLs too.

DSLs are usually used to help people express a domain-specific algorithm in an easy and quick way without the necessary knowledge of the art of programming. This primarily allows domain experts without any programming background to develop algorithms to control

processes in their domain. This is used for example in the domain of banking or insurance companies.

DSLs may also help programmers who develop programs for a certain domain. In this case, a lot of repetitive code or processes might be necessary. This slows down development, it is annoying for the developers and it is error-prone. In this case, a DSL could do these repetitive activities implicitly, without any action from the developers. This would obviously increase efficiency of the development, increase safety of the code and decrease the cost. For example, one could use a DSL in the domain of space engineering and decrease the risk of code errors causing a destruction of a spacecraft.

As can be seen from the examples above, DSLs can be very helpful. However, how can one get a DSL, write code in it and execute the code? There is no standardized way to do that. Martin Fowler, a famous British software engineer, introduced the term *language workbench* which denotes a tool used for defining and composing DSLs, together with creation of integrated development environments (IDE) for them and solving the DSL code execution. The following section introduces one of language workbenches, the JetBrains MPS tool.

1.2 JetBrains MPS

JetBrains Meta Programming System (MPS) is an open-source language workbench. As described in the previous section, it allows the user to create a custom DSL, IDE for the language and MPS then provides a solution for execution of the DSL code.

The main window of the MPS tool is captured in Figure 1. The right area serves as the editor and the left area as the project explorer.

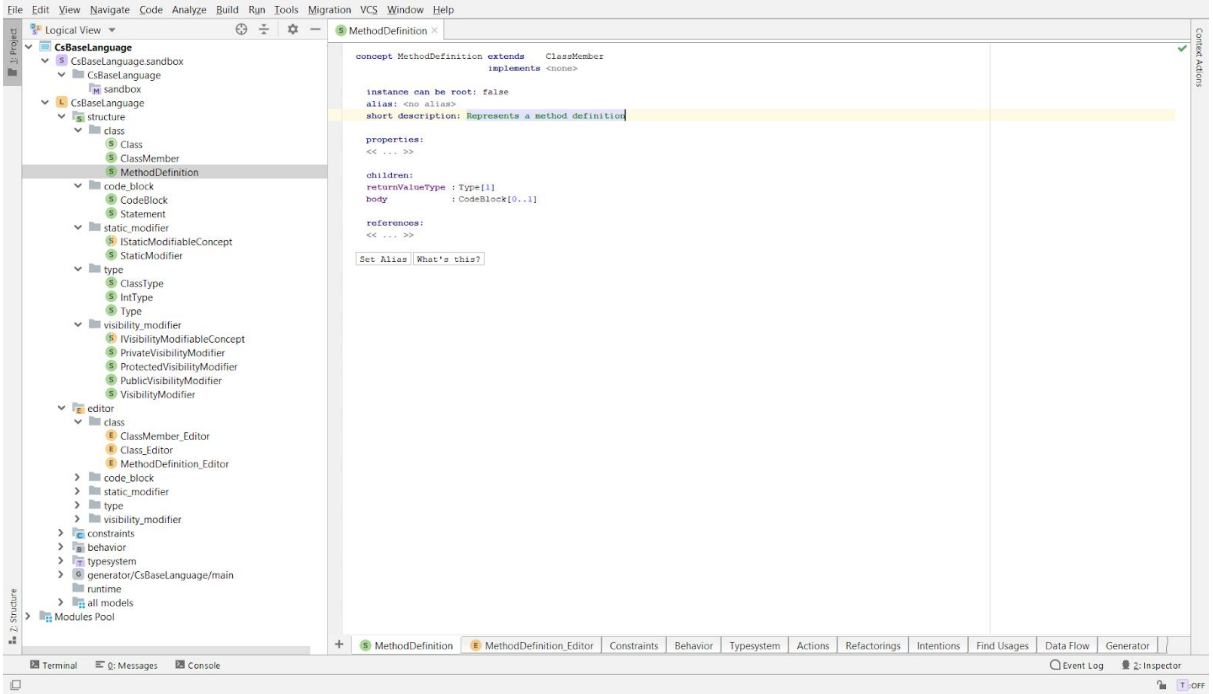

Figure 1: The environment of the MPS tool

There are two kinds of users of the MPS tool. First, there are language designers who develop DSLs. Second, there are end-users who utilize these DSLs to develop actual programs.

First, let's briefly look onto the work of end-users, what is going on behind the covers in MPS and after that, let's explain the basics of defining a DSL. To simplify the text, we use two representative users: Alice, the language designer, and Bob, the end-user.

The first thing Bob has to do to create a DSL program is to select the DSL language for it. Then, he develops the code in an integrated development environment which was provided by the MPS tool and flavorized by Alice.

The main characteristic of this environment is the projectional editor. Bob does not write the source code in a text form, as he would in a common editor. Instead, he executes transformation actions that directly edit the abstract syntax tree (AST), which is a tree representation of a program's code capturing the hierarchy of the code elements. The transformation actions may look like writing code, using shortcuts and often they have a form of smart, user-friendly creation of text-cells in the editor, which represent the AST nodes.

This has several advantages:

- Alice can design the language so that it has no text form. Bob might be allowed to use tables or diagrams in the code. This is probably the greatest advantage.
- The language may provide multiple alternative notations, allowing Bob to choose a notation that fits best the task at hands - implementation, testing, merging, debugging, requirement tracking, etc.
- Bob can combine as many DSLs as he desires in a single program without any explicit indication of what part of code is in what DSL.
- Authors of MPS do not have to parse textual form of code.

The main drawback is that Bob has to get used to a different style of writing and editing code than in a common text editor, even though authors of MPS put a lot of effort into user-friendly way of using the projectional editor and to a great degree succeeded.

After writing the code, Bob just has his code transformed into source codes of a GPL language, currently, in most cases, Java. This process is only a sequence of model-to-model transformations, where the models are the DSLs, followed by a single model-to-text transformation. In other words, the segments of the Bob's code are transformed into equivalent segments of a code in a different DSL, then again and again until they are transformed into equivalent chunks of a GPL.

As one can expect, the intermediate steps of model-to-model transformations into other DSLs are not necessary and in many cases are not even present. But the last step of the model-to-text transformation into the GPL source code is mandatory (if text is the desired output).

These magical transformations are possible due to Alice's work. Without Alice, MPS would not know how the transformation should look like.

Alice, who develops DSLs in MPS, needs to understand several areas of language definition in MPS, or, as called in MPS, *language aspects*:

- *Structure*: the hierarchical definition of the DSL which contains all elements of the language and specifies the relations between them.
- *Editor*: the way how the language elements will be displayed in the projectional editor
- *Generator*: the definition of a model-to-model transformation into another DSL. Elements in the defined DSL are mapped onto elements in another DSL.
- *TextGen*: the definition of a model-to-text transformation into a GPL. Elements in the defined DSL are mapped onto source chunks in a GPL.
- Many other supportive aspects, primarily intended to ease the work of language designers and to make the projectional editor more user-friendly. E.g. *Type System*, *Intentions* or *Behavior*.

To develop a DSL, Alice does not have to define both Generator and TextGen. She can choose whether she wants her DSL to be transformable into another DSL (e.g. in case when she wants to add some little improvements into another DSL) or whether she wants her DSL to be transformable into a GPL (e.g. in case when she defines a brand new DSL).

More details about Alice's work are left for the section *DSL definition in MPS*.

Even though a DSL can be transformed directly into a source code of a GPL, DSLs are usually transformed into a special intermediate language which is basically the same as the given GPL but is defined as any other DSL in MPS. In other words, this special language is another DSL into which other DSLs are transformed but has the same elements as the corresponding GPL. These special languages are called *base languages*.

For example, there is Java base language (often called only *Base Language* as it was the first base language in MPS). DSL code is usually transformed into this Base Language and the code in this Base Language is then transformed into Java.

This might seem confusing for a beginner. Why we need this step? Actually, we do not. But:

- it simplifies the work of language designers as they do not have to usually understand the TextGen aspect but only the Generator aspect,
- it allows the end-users to use a base language (e.g. Java) directly in their DSL code, which would not be possible without base languages,
- it allows to create simple extensions of a GPL, like decision tables.

The current version of MPS supports the following base languages (some of them are not intended for creating programs):

- Base Language: represents Java 6 and some extensions from Java 7 and Java 8,
- C,
- JavaScript,
- Bash,

- R,
- Ant,
- XML.

Now that we understand what MPS is, how it works and what are base languages, it is the right time to introduce this software project.

1.3 Motivation for This Project

The main goal of this project is to extend the set of supported base languages in MPS by the C# base language.

This will make the MPS tool accessible to users who prefer the C# ecosystem:

- it will allow language designers to make their current and new DSLs to be transformable into C#,
- it will allow end-users to use C# directly in their programs.

Furthemore, since the Java base language has very poor documentation, our goal is also to provide an extensive documentation of the new base language and its development. This documentation will primarily serve for further maintenance and extending of the C# base language and developing other base languages.

The goal of this project is not to provide a base language corresponding to full and latest C# version but rather a base language corresponding to the most commonly used part of C#. The supported subset of C# will be discussed in the *Analysis* chapter of this specification.

1.4 This Document

This document serves as a detailed specification of a project developed for the purposes of the Software Project course at the Faculty of Mathematics and Physics of the Charles University.

The first chapter of this document located above this section introduces the reader to the problem which the project aims to solve and its context.

The second chapter provides detailed description of the MPS tool from a point of view of language designers.

The following, third chapter provides more detailed description of this project itself, especially its goals.

The fourth chapter presents a discussion about some key problems in the project development and tries to provide the answers for them.

In the fifth chapter, a high-level plan of the software process for this project can be found.

The sixth chapter contains a glossary allowing the reader to quickly find definitions of some key terms used in this document.

2 DSL definition in MPS

The following sections describe MPS from the viewpoint of a language designer in further details than in the *Introduction* chapter of this specification.

2.1 Language Aspects

A definition of a DSL in MPS consists of several parts called *language aspects*. Some of them are essential and the language would not work without them (e.g. *Structure*) and some provide support for the other aspects or make the language more user-friendly for the end-user (e.g. *Intentions*).

Figure 2 illustrates different aspects used in the Base Language as displayed in the project explorer of the MPS tool.

Each aspect is defined in a specific language the MPS tool uses for that particular aspect. Each of them is a DSL again and as such its code is developed in a projectional editor.

Each of the following sections describes one of the language aspects. They do not cover all of them, but only those which are important for this project.

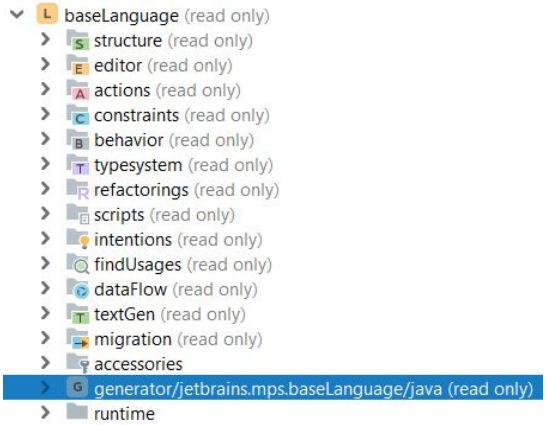

Figure 2: Language aspects of Base Language

2.2 Structure

The Structure aspect is the most important aspect of a DSL. It defines the hierarchical structure of the language: the elements, their IS-A and HAS-A relations and properties of

both. Unless a language element is defined in the Structure aspect, the end-user cannot use it.

Each language element is defined by so-called *Concept*. When the end-user creates a program, he or she edits an AST. The AST consists of nodes where each of them is an instance of a concept defined in the DSL. Thus, one can view the concepts as classes from the object-oriented programming and nodes as their instances.

A concept mainly consists of:

- IS-A relations to parent concepts. They are slightly different from the inheritance in the terms of object-oriented programming. Concepts are more about identity than about behavior.
- HAS-A relations to children concepts. They specify the hierarchy of the language concepts, e.g. that class has methods.
- Properties. A property can be of a basic type (e.g. a name of a node in an AST) or of a type of a reference to an AST node (a link across an AST, e.g. from a variable assignment to a declaration of the variable).

Figures 3 and 4 show an example of a concept definition. Figure 3 illustrates the Base Language definition of a concept corresponding to the class language element. Figure 4 shows a concept corresponding to a method definition in our current version of the C# base language.

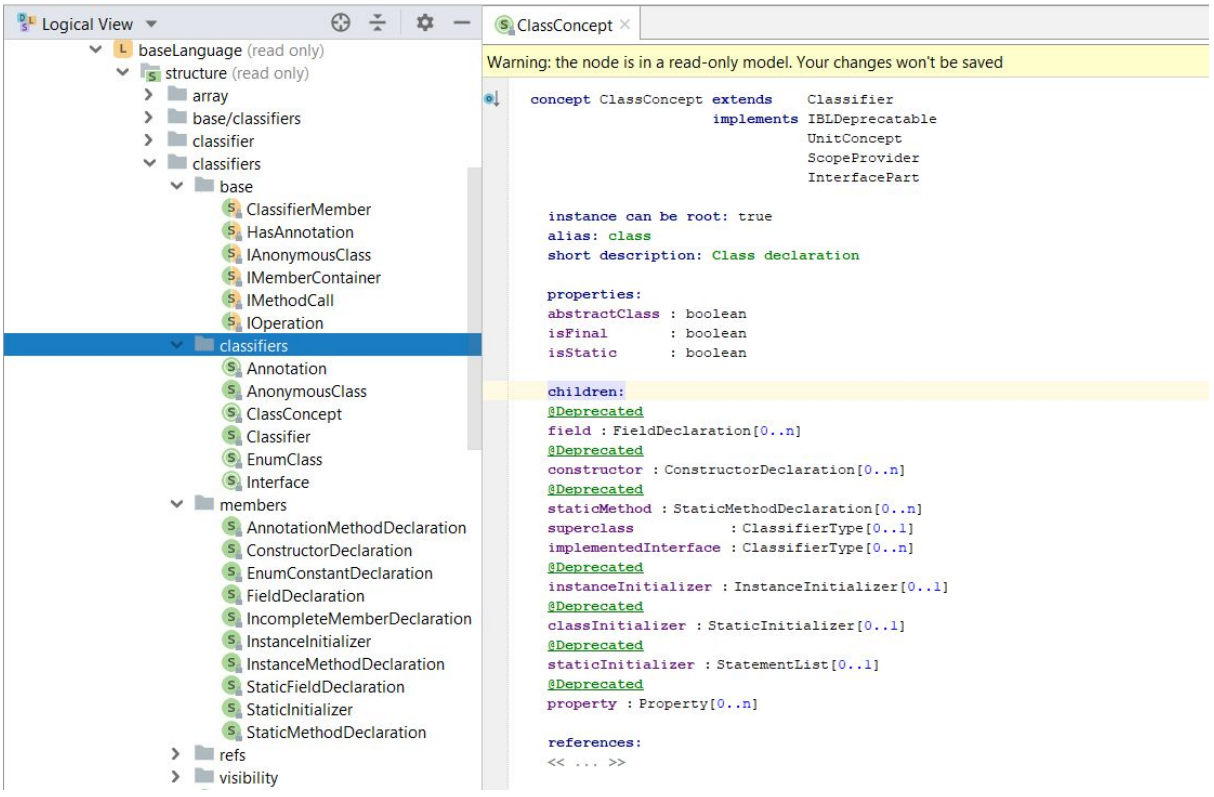

Figure 3: The Structure aspect of the Base Language class concept

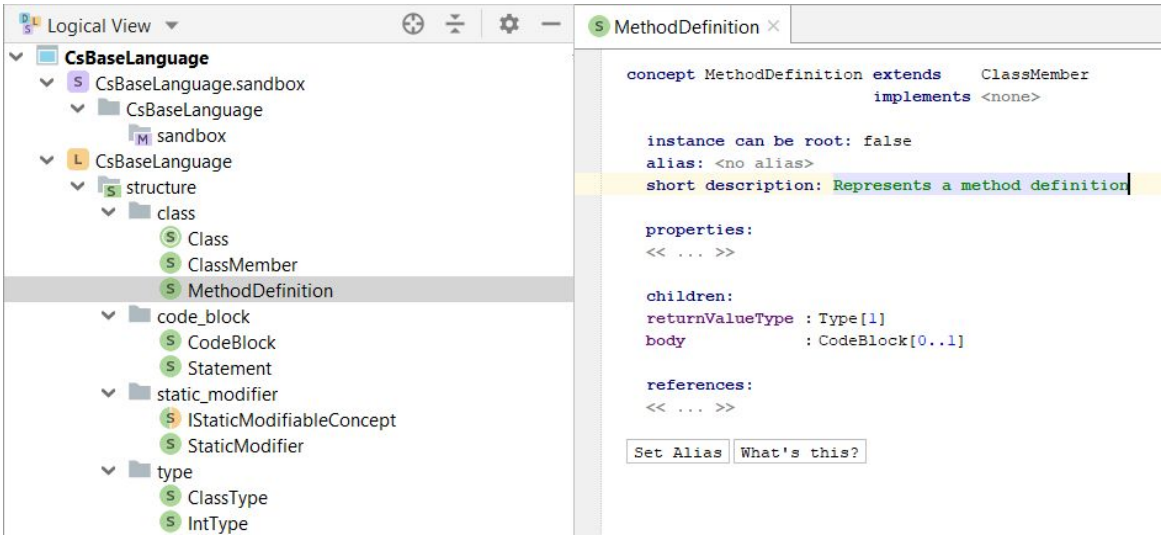

Figure 4: The Structure aspect of the method definition concept in our current base language

2.3 Editor

The Editor aspect defines a so-called *editor* for each concept from the Structure aspect. A concept's editor specifies how an AST node of that concept will be displayed to the end-user in the projectional editor.

Figures 5 and 6 illustrate an Editor aspect definitions for the method definition concept in our current version of C# base language and for the class concept in Base Language. One can notice that MPS uses quite a complex language for the Editor aspect so that it can provide the language designers with different layouts of text, etc.

The language designer does not have to define any editor for a concept, in which case MPS uses a default editor which prints all information about the AST node in a structured text format.

Even though the Editor aspect is not necessary to define, it is crucial for the DSL to be user-friendly to the end-user.

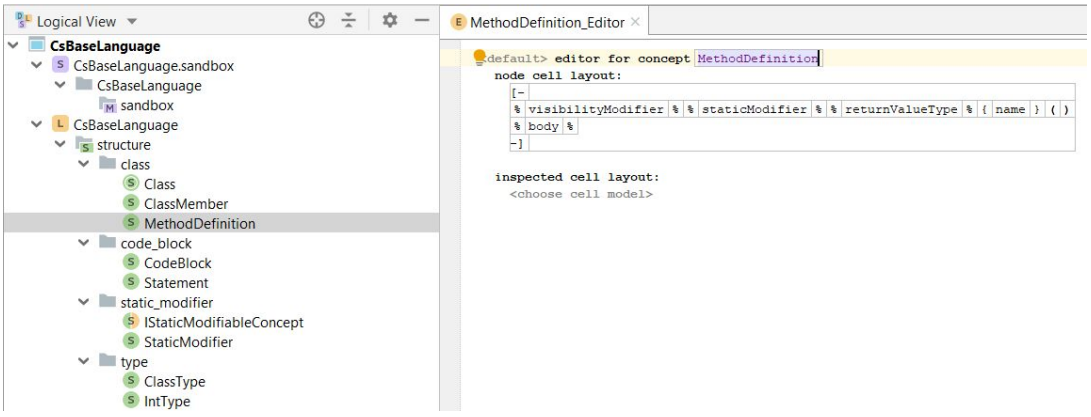

Figure 5: The Editor aspect for the method definition concept of our current base language

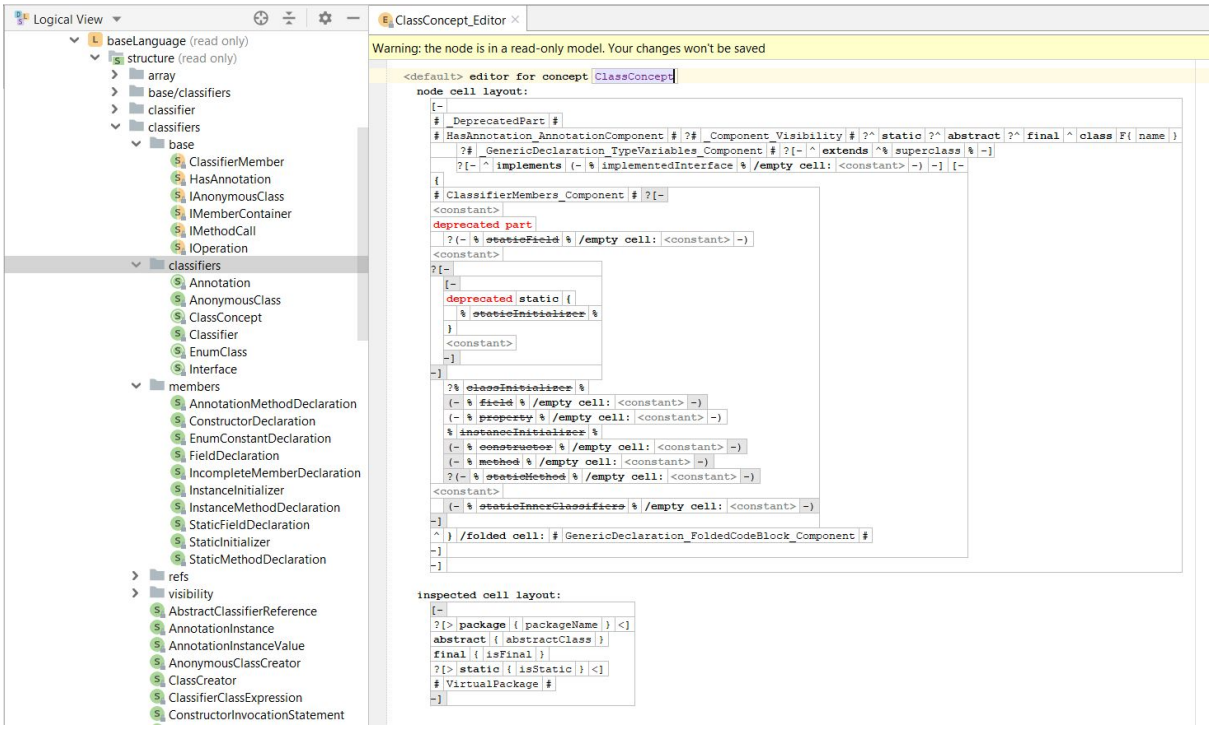

Figure 6: The Editor aspect for the Base Language class concept

2.4 Generator

The Generator aspect is used to define a model-to-model transformation of a DSL into another DSL. It consists of a set of concept-mapping rules.

It is the second most important language aspect of a DSL definition as without it the DSL couldn't work. However, the Generator aspect is not usually a part of a base language definition.

2.5 TextGen

This aspect is counterpart of the Generator aspect in the area of base languages. It defines model-to-text transformation of a base language into a GPL. As such, it is the second most important language aspect for base languages.

It consists of rules which transforms an AST node into a source code.

Figure 7 shows the definition of the TextGen aspect for the class concept of the Base Language. The language of the TextGen aspect is characteristic by many *append* commands, which append text to a buffer which gets inserted into the resulting source file, at the right place that corresponds to the AST node of the given concept.

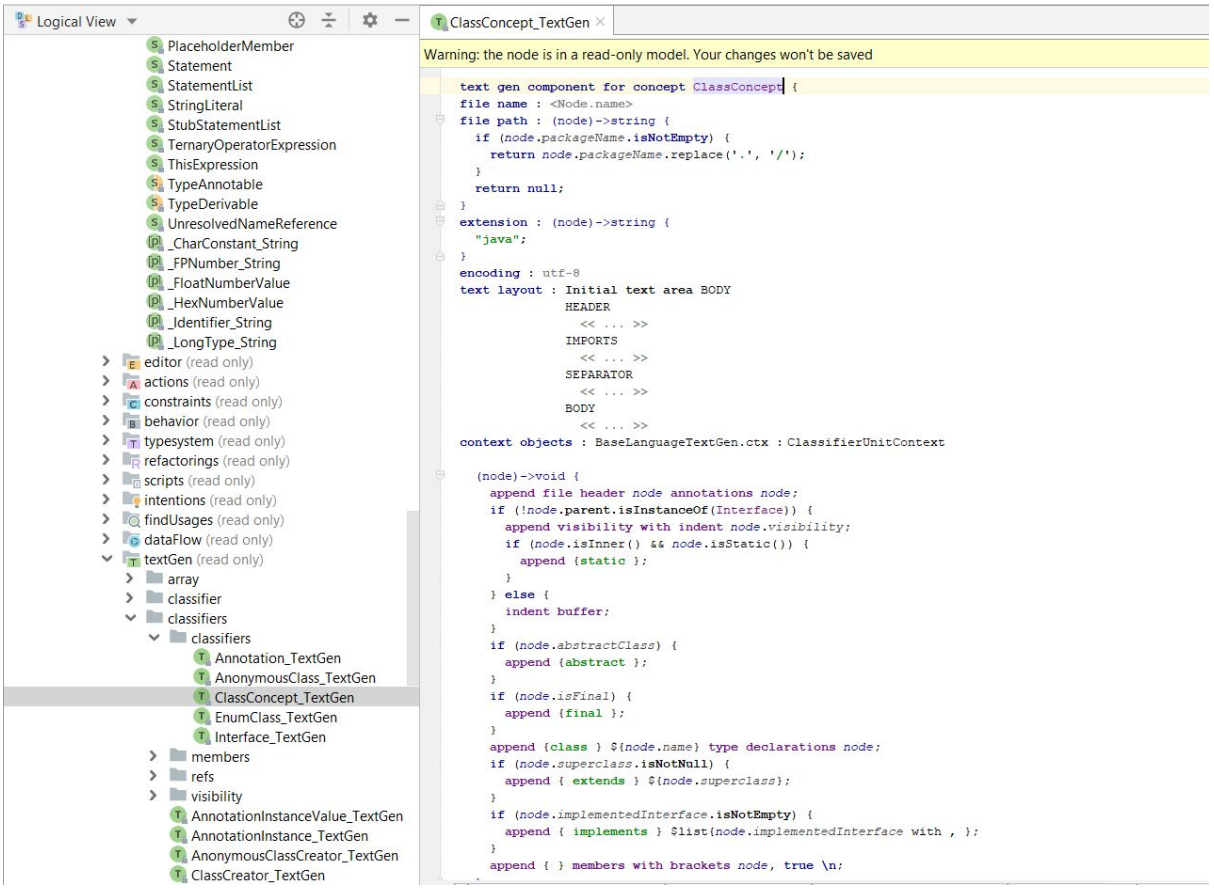

Figure 7: The TextGen aspect of the Base Language class concept

2.6 Type System

The Type System aspect ensures type correctness of a DSL code, warning the end-user about incompatible type usage and similar coding errors.

Developing a type system for a complex language is difficult. For example, in case of Base Language, it is not fully implemented even though it has been developed for several years.

2.7 Intentions

The Intentions aspect defines quick actions which the end-user can initiate when he or she positions the cursor at some location in code. An example is illustrated in Figure 8.

Intentions serve to improve the usability of the IDE providing shortcuts to accomplishing desired code modifications.

Figure 9 shows the Intentions aspect definition for an intention of adding an *else-if* clause to an if statement.

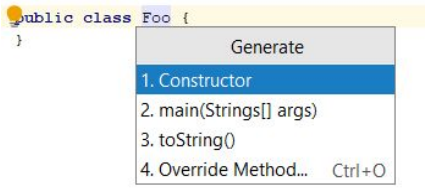

Figure 8: Example of an intentions dropdown menu in an end-user's program

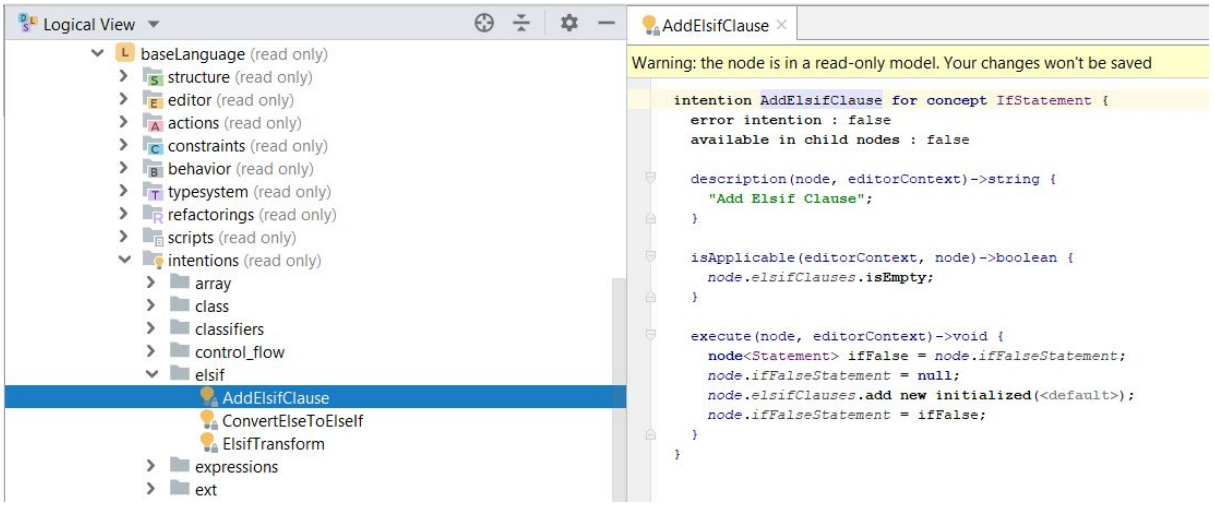

Figure 9: The Intentions aspect of the Base Language if statement concept

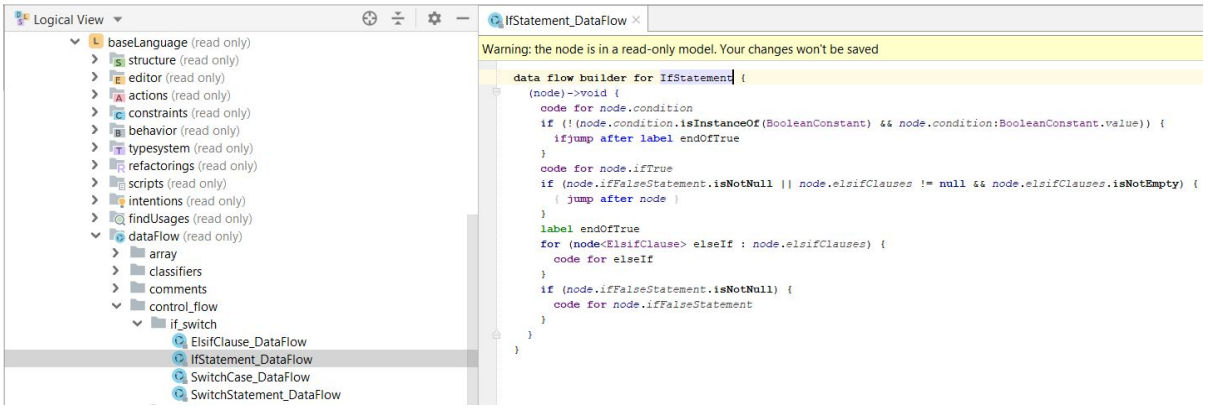

Figure 10: The Data Flow aspect of the Base Language if statement concept

2.8 Data Flow

The Data Flow aspect computes control and data flow through the user code. It may detect important measures like code reachability or read-before-write errors and warns the user about such issues.

Figure 10 illustrates the definition of the Data Flow aspect for the if-statement concept.

2.9 Behavior

The Behavior aspect allows defining methods for a concept from the Structure aspect. These methods can then be used in other language aspects.

Figure 11 shows such a definition for the if-statement concept.

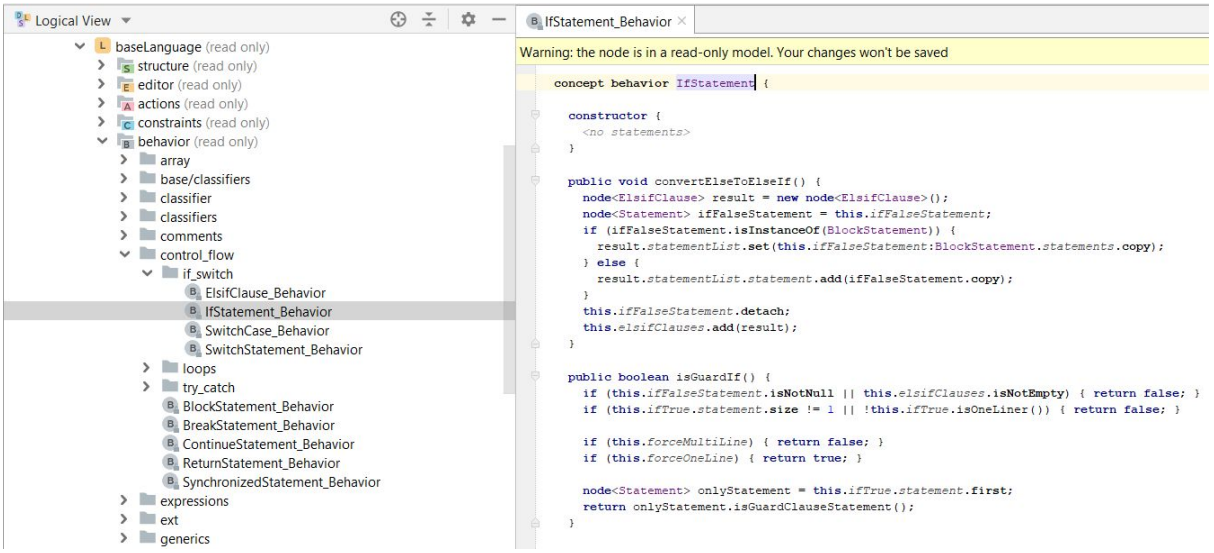

Figure 11: The Behavior aspect of the Base Language if statement concept

3 Project

This chapter of the specification describes this project in a greater level of detail, in the context of the previous sections.

3.1 Overview

Our project aims to develop a C# base language for the MPS tool. This comprises development of at least these language aspects:

- Structure,
- TextGen,
- Editor,
- aspects which make projectional editing user-friendly, on a level comparable to the Base Language

The supported subset of C# is described in the *Analysis* chapter of this specification, in the section *Supported subset of C#*.

We intend to deliver more than in the list above. The discussion about possible extensions is in the *Conclusion* section. We aim to deliver at least one of the advanced features listed in the *What will be delivered if time permits* subsection.

The developed C# base language must be extensible to allow future improvements and development of new features such as support of the rest of C# language together with the new and upcoming C# features, both as chapter of the base language itself and as its language extensions.

On top of the previous, we will deliver an extensive documentation of the developed base language and the development itself.

3.2 Expected Usage

Implementing the C# base language will extend the current functionality of JetBrains MPS in several ways. In principle, there are three parties which are expected to use this project: MPS language designers, MPS end-users and MPS community developers (MPS maintainers). Below, we illustrate the expected usage of the project for each of them.

Language designers may use it as a base language for their DSL so that the end-users will be able to generate C# source files for their programs developed in this DSL. For this purpose, the C# base language should allow creation of basic C# programs.

End-users might develop a program directly in the C# base language instead of using a DSL. For this purpose, it should be user-friendly to develop code in the created C# base

language and it should be possible to develop slightly complex C# programs in it. Some DSLs are even only a thin extension over the base language, e.g. they add only decision tables or physical units to numbers. This use-case also requires the base language to be usable for slightly complex programs. Very complex programs are not expected to be developed using the base language, as one would probably use rather directly the GPL itself.

MPS community developers and MPS maintainers are expected to use the documentation of the C# base language to maintain and extend the language definition. Furthermore, they might use it for development of other base languages. Therefore, it is important to deliver a well-documented and extendable solution.

3.3 Goals and Requirements

In this section we describe the goals for this project as well as requirements set by JetBrains. The requirements must be met to provide full compatibility and consistency with the existing MPS environment and to allow future extensions by the community.

Goals

- Implement the support for C# to MPS:
    - Support for the specific subset of C# language features (see the section *Supported subset of C#*).
    - The Structure aspect
        - Besides the support of the promised subset of the C# language, the aim is to provide easily extendable solution to allow future extensions.
    - The Editor aspect
        - We will focus on user-friendly editing of C# code in the projectional editor. The result should be as close as possible to editing of C# code in a standard text-based editor. In other words, the AST transformation actions should be easy to do and as intuitive as possible.
    - The TextGen aspect
    - The Intentions aspect
        - Further extension of the user-friendly editing of code.
- Provide the means for importing libraries, mainly the standard C# libraries.
    - Without these, the users would have very limited usage of our base language.
- We would like to implement some of the advanced features (see the subsection *What will be delivered if time permits* in the *Conclusion* section), but this cannot be guaranteed.
- Provide an extensive documentation for the language and the decisions made along its development.
    - Since Base Language has very limited documentation, it will be an immense help for future development of base languages.

Requirements

- The solution must be compatible with the latest MPS version.
    - JetBrains releases three MPS versions every year, version 2019.3 is expected to be released within the time frame of this project - in December 2019.
- All to-be-published documents shall be provided in English.
- The project shall be released under the Apache 2.0 license to provide consistency with the MPS standard.
    - The project will remain open source under the JetBrains repository and is expected to be further extended by the community and JetBrains.

4 Analysis

This chapter of the specification discusses some of the main problematic areas of the C# base language development and tries to provide answers for the discovered questions.

4.1 Supported Subset of C#

There are multiple versions of the C# programming language, which differ in a nontrivial manner. Our base language should correspond to one of them. In the following paragraphs, we discuss what version should we use for this purpose.

Version Selection

The versions 1.x of C# are old and dramatically different from the newer versions and thus they are not widely used. As such, they are not appropriate candidate specifications for our C# base language.

The versions 2.0, 3.0 and 4.0 are still old, but are much closer to the modern versions of C#. They are the last versions supported by the Windows XP operating system. Even though this system has been deprecated and has no further support from Microsoft, it is still used world-wide. This may be a reason to consider these versions as candidate specifications for our base language. However, we decided not to use them because the first C# base language supported by the MPS tool should probably correspond to a C# version for non-deprecated platforms.

The version 5.0 of C# is the newest version that has been officially standardized by ECMA and ISO. This is a great advantage of this version as its formal description can be very helpful during the development of the C# base language.

Thanks to open sourcing of .NET Core, Roslyn and other parts of the .NET ecosystem, together with involving the community into the development, C# has rapidly developed during last years, introducing the versions 6.0, 7.0, 7.1, 7.2 and 7.3. The features they bring are, however, only new syntactic sugar which is not so important to support. Furthermore, official standards for these versions do not exist yet.

The last version 8.0 of C# is currently in development and not released yet. This makes it not an appropriate candidate for our C# base language.

We decided to base our base language upon the version C# 5.0:

- The main advantage is that there is an official specification from ECMA and ISO which will help us when we find ourselves in doubt during the language definition process. Furthemore, there is no risk that the specification will change in the future and our language definition would be invalidated.

- The support of newer versions will be possible to add and it will be left primarily for community around the MPS tool. It should not be as difficult as implementing the first C# base language because the future versions of the C# base language may just extend our version.
- Furthemore, many people are not yet used to the newest versions of C# and their features.
- It is the first version where the standard libraries and runtime environment is covered by the .NET standard.

As the C# 5.0 language is too big and complex for the scope of this project, we divide its features into three categories:

- features that we will deliver,
- features that we will deliver if the restricted time for the project allows us to do so,
- and features that we expect not to deliver.

The following subsections present these three categories.

What Will Be Supported

This subsection contains a list of features that are included in almost every C# program. They are the core features that must be implemented:

- Members:
    - namespace,
    - struct,
    - enumeration,
    - class,
    - interface,
    - array.
- Class/Struct members:
    - constructors,
    - methods,
    - fields,
    - properties,
    - constants.
- Modifiers for the following members:
    - class,
    - constant,
    - field,
    - method,
    - property,
    - accessor,
    - constructor,
    - struct,
    - interface,
    - enum.
- Inheritance:

    - class inheritance,
    - interface implementation.
- Built-in types:
    - integral types (8,16,32,64 bits, signed and unsigned),
    - floating-point types (float, double),
    - decimal,
    - bool,
    - string, char,
    - dynamic,
    - object.
- Decimal integer literal
- Variables:
    - static,
    - instance,
    - local,
    - arguments (value, output, reference).
- Generics (definition and usage)
- Comments
- Expressions
    - method calls,
    - assignment,
    - operators.
- Simple using namespace
- Statements:
    - Iteration statements:
        - while,
        - do,
        - for,
        - foreach.
    - Jump statements:
        - break,
        - continue,
        - goto,
        - return,
        - throw.
    - Selection statements (conditional-jump statements):
        - if-else,
        - switch.
    - Blocks of statements

What Will Be Supported If Time Permits

The following list contains less frequently used features of C# 5.0, which are not so important. Most of them can be replaced by combinations of the previously listed features. On the other hand, programmers sometimes use them. We decided to support their syntax if time given for this project permits us to do so:

- Delegate + modifier
- Partial types (only syntactically)
- Async functions
- Anonymous functions
- Nullable types
- Events + modifier
- Indexers + modifier
- Operators overloads
- Anonymous functions
- Namespace aliases and their other advanced usage
- Extension methods
- Finalizers
- Explicit interface implementation
- Exceptions
- Expression (full syntax support)
- Literals (full syntax support)
- Attributes
- Preprocessing directives
- Statements:
    - Checked/Unchecked statement,
    - Lock statement,
    - Using statement,
    - Try statement,
    - Yield statement.

What Is Not Expected to Be Supported

This subsection defines features that are used only occasionally or that are too complex to implement them or which relate to other than syntactic part of the language. Their support is expected to be out of scope of this project:

- Query expression - LINQ syntax
- Unsafe code in general
- Checks for partial types

4.2 Inspiration and Used Resources

In this section, we discuss which resources we could reuse and which we could use as an inspiration when designing the language in MPS.

There is an existing diploma thesis *Grammar to JetBrains MPS Convertor*[1] which analyses generation of an MPS language from a grammar of the language. We decided not to base

[1] Diploma thesis of Přemysl Vysoký at the Faculty of Mathematics and Physics, Charles University

our solution on the language generated by the tool developed in this thesis because it is rather not human-readable and that would prevent further redesigning of the language, for example in order to get better usability and user-friendliness inside MPS. Instead, we decided to develop the whole language ourselves to really understand its structure and to be able to efficiently redesign it when necessary.

To design the C# base language well and fast we intend to use some inspiration resources:

- Base Language is currently the number-one base language for MPS. As such, it should be a good resource for inspiration. On the other hand, it has no documentation and some of its parts are deprecated, which could leave us guessing which parts do what and whether they are not a bad historical design. Furthermore, Java and C# are not the same languages and therefore an elegant design in one might not be good in the other.
- The grammar of C# 5.0 is freely available and should also be a good candidate for inspiration. However, it may be too complicated as it covers whole C# 5.0, whereas we do not intend to support all the features. Too complicated language design might result in worse future maintainability of the developed language.
- The specification of C# 5.0 and similar resources from Microsoft are rather designed to provide information about special cases than about the language hierarchical structure. Thus it may be hard to retrieve that information from it.

We decided not to use just one of these resources and copy the language structure from it but rather design the language structure ourselves with combining inspiration from all of these resources. This should result in our better understanding of the language structure and developing a language design which is simple and well-maintainable.

Furthemore, there are some existing tools which ease development of an MPS language. For example, a tool which generates the TextGen aspect from the Editor aspect of a language. We decided to consider their usage later in the development.

4.3 MPS Language Aspects

This section analyzes individual language aspects of the C# base language, their importance and proposed approach for their development.

Language Structure

The first step of implementing a language in MPS is creating the Structure aspect. It is the most important part of the language design.

The following paragraphs discuss minor problems within the Structure aspect.

Root Concepts

Root concepts represent the root of the AST tree. Constructs represented by these concepts can be written in the projectional editor without being encapsulated inside another construct.

By studying the C# grammar we have identified these root concepts: namespace, interface, class, struct and enum.

Abstract Concepts

Sometimes, there might be multiple concepts in the role of a member of another concept. For example, the class concept has the method concept, the field concept and other concepts used as members. There are two ways how to implement it.

First, we can list all the individual concepts (method, field, etc.) as members of the super concept (class). However, this design would force the user of the base language to define all the methods together in one place, all the fields together and so on.

Or we can try a second approach. We create an abstract concept (member) and make it a single member of the super concept (class). The original member concepts (method, field, etc.) will inherit this abstract concept (member). This design will be more flexible. We do not force the user to define the methods altogether or fields altogether and furthermore we can more easily add another member concept (e.g. an inner class definition).

Editor

MPS generates a default editor for every concept in the Structure aspect. However, the behavior of the default editor is not what we are looking for. We aim to create an editor that resembles native C# syntax. This means that the code displayed in the projectional editor should look exactly like C# code. For instance, if we used default editor generated by MPS, the visibility modifier would be displayed as if it was inside the class, instead of being a part of the class declaration.

One of challenging parts of creating a user-friendly editor is to properly display user defined types as the method return types, variable declarations, etc. This can be achieved by providing concepts with references to other AST nodes.

TextGen

This language aspect cannot be underestimated because without the ability to translate the C# base language to C# source code, the user will not be able to compile and run programs. This fact makes the generator the second most important part of the language.

4.4 Support for External C# Libraries

As almost every C# program uses standard C# libraries, the support for their usage in the C# base language is crucial. Furthermore, non-standard libraries should be supported too.

To add the support of external C# libraries, they have to be parsed and a list of high-level entities (e.g. classes, methods, fields) must be extracted and imported into MPS.

In MPS, the user uses only the high-level entities such as classes or their members, not their implementation code. Therefore it is enough to extract only the high-level entities in a library. For the same reason, support of external libraries in MPS is provided through creation of stubs representing the high-level entities. These stubs have an empty implementation.

A stub of the C# *System* class created for the Base Language is presented in Figure 12. One can see that the methods have an empty implementation, as discussed above.

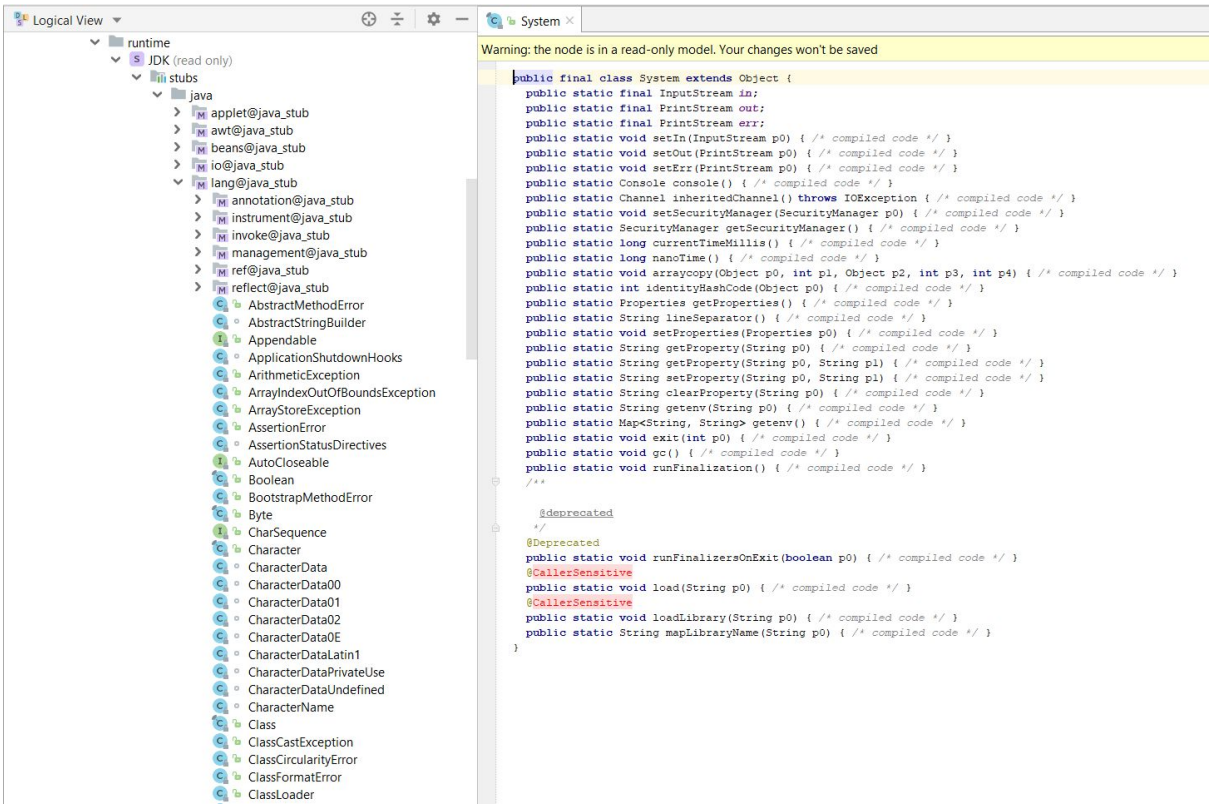

Figure 12: The stubs generated for the Base Language

4.5 Testing

The implemented C# base language should be properly tested for obvious reasons.

Two of the most important tests include creation of a DSL based upon our base language and creation of a stand-alone program in our base language. Also, some kind of automatic tests, such as testing of correct C# source code generation or correct behavior of the projectional editing of code, will be developed.

4.6 Documentation

The documentation is an important part of the delivered solution as it will be used for further development of the C# base language and maybe for development of new base languages.

The documentation should be as close to the code as possible because that way it is more likely that it will stay updated when the code is changed. The MPS tool allows to put documentation right into the language definition. We decided to use this feature for low-level documentation of concepts in the Structure aspect.

Furthermore, a documentation describing the language from a high-level view is also needed because it is important for new-coming community members and JetBrains developers maintaining the C# base language and it might be useful for developers who are designing a brand new base language. For the purpose of this high-level documentation, we decided to use a hierarchical documentation in form of interlinked Markdown-formatted files. The Markdown format is readable as-is but to improve readability further, we will use the MkDocs tool to generate an HTML project documentation from the Markdown files.

4.7 Conclusion

Here is a short conclusion of the previous sections, describing what will be part of the final product and what will be omitted and why.

What Will Be Certainly Delivered

The final solution must include all the necessary aspects of an MPS base language, namely the Structure aspect, the Editor aspect and the TextGen aspect. They should be implemented in an appropriate extent so that the user will be able to write reasonable C# code. Without any of the previously stated aspects, the C# base language is considered incomplete.

As for the concrete subset of C# language that will be certainly covered by our base language, see the section *Supported subset of C#*.

Another part of the product is providing the user with support for C# external libraries, without which the base language would be almost unusable.

Last but not least, one of the reasons why the C# base language is needed is the ability to create DSLs in MPS which can be generated as C# source code. This means that our base language must be available in the form of an MPS plugin, so language designers can use it in MPS projects. Also, it will be available as an open source project under the Apache 2 license.

Part of the final product will, of course, be the documentation, for more details see *Documentation* section.

What Will Be Delivered If Time Permits

Here we list some features of the C# base language which are not crucial for proper usage of the language. If we find ourselves ahead of the schedule at some point during the

development, we will pick some of these features. Note that this is not by any means guaranteed, as we may not be ahead of the schedule at any point during the development.

- Advanced C# language constructs (see the *Supported subset of C#* section),
- The Data Flow aspect of the language.

What Is Not Expected to Be Delivered

We decided that these functionalities will not be supported as their complexity is well beyond the scope of this project:

- The Type System aspect (e.g. checking types of variables and expressions while writing the code) is commonly supported feature in modern IDEs. However, implementation of the language Type System is too vast topic and out of scope of this project (even the MPS Base Language Type System, after many years of development, does not fully handle all the corner cases of the Java type system). The C# base language will be fully working even without the Type System aspect.
- Import of external libraries whose API is not easy to parse is considered out of scope of this project. Absence of this feature should not be a problem for a common user of the language as reasonably assembled libraries will be supported.

5 Project Plan

In this chapter of the specification, we describe proposed work decomposition, its organization and analyze the risks for this project.

5.1 Implementation Decomposition

As we have described before, implementing a DSL in the MPS tool consists of three basic parts: the Structure aspect, the Editor aspect and the TextGen aspect.

Two of these parts, the Structure and the Editor, are tightly coupled. This is given by the way MPS works as the Editor is defined for every concept from the Structure aspect. This coupling also makes sense from the testing viewpoint, because by implementing the Editor for a concept we can immediately see how this concept will look in the projectional editor.

The last part, TextGen, can be implemented later as generation of C# source code is possible only after the Structure is provided.

Parsing of the API of the C# standard library can be implemented after we complete implementation of the basic part of our base language.

The Intentions aspect and making the Editor user-friendly is not a core feature and as such it will be left for later.

Reasoning from the previous paragraphs leads us to a decomposition where the language will be iteratively extended. We intend to start from the root concepts (e.g. class, interface, etc.) and move down the language structure. Each iteration will have the following schedule:

1. Implementation of the Structure aspect corresponding to the iteration, together with the Editor aspect.
2. Implementation of the TextGen aspect for the implemented concepts in the iteration.

After we implement the core of the language structure, we intend to start working on STL API support.

At the end, we will focus on the user-friendly editing in the projectional editor.

Along the way, we will document the solution.

It the time permits, we intend to develop some of the advanced features of the C# base language (see the *What will be delivered if time permits* subsection in the *Conclusion* section).

5.2 Team

The team consists of five team members:

- Tomáš Eliáš
- Roman Firment
- Jakub Saksa
- Martin Wirth
- Dalibor Zeman

The team is supervised by:

- RNDr. Pavel Parízek, Ph.D. from the Faculty of Mathematics and Physics, Charles University
- Mgr. Václav Pech from the JetBrains company

5.3 Work Organisation

We decided to organise our work into short iterations similar to sprints in the Scrum methodology. Each iteration will be 2 or 3 weeks long. Before each iteration we will have a meeting where we will discuss what to do in the following iteration and distribute the work to individual members of the team. At the end of the iteration we will have to deliver what we have agreed on at its beginning. This approach ensures that each team member will be working on the project, that the project will be developed continuously and that the tasks assigned to a team member will be small and well-understandable.

At the meeting, every member of the team will be assigned a set of tasks to deliver at the end of the iteration. The tasks will typically have size of multiple language structure concepts including the Editor aspect for that concepts.

The Structure aspect will be divided into development units corresponding to subtrees of the language structure hierarchy tree. Within an iteration, each member will be assigned to develop such a subtree. This distribution of work decrease dependencies among work of different team members because if two members worked on the same subtree, there would be a high probability that the work of one member would depend on the work of the other and that would cause delays.

Besides implementation of the functionality, a part of a team member's work will be also reviewing of another team member's work. The reason for the reviews is to enforce every member to follow coding conventions we have agreed on beforehand (e.g. naming conventions, documentation comments, etc).

For issue tracking we will use the JetBrains YouTrack tool.

5.4 Risk Analysis

Below is an overview of risks which could influence the schedule of this project:

Risk	Impact on the project	Probability of occurrence
Bad design of the Structure aspect at the beginning of the project	High	Medium
Problematic creation of stubs for entities inside an external C# library	Medium	Medium
Problems with ensuring user-friendly editing of C# base language code in the projectional editor	Medium	Medium
Very complicated parsing of C# libraries	Medium	Low
Problematic cooperation inside the team	Medium	Low
Quit of a team member	Medium	Low

The solution of the first four risks is just simply finding a way how to solve the problems. As the C# base language would not be usable without those features, they just have to be implemented.

The solution of the last two risks lies in redistribution of the work among the working team members.

Occurrence of a risk will increase the probability that no advanced features of the C# base language, listed in the *What will be delivered if time permits* subsection of the *Conclusion* section, will be delivered. In the worst case, it could, of course, even endanger delivering of the basic solution on time.

6 Glossary

Here is the list of the most important domain-specific terms used in the document, in alphabetical order:

- **Abstract Syntax Tree** = see AST
- **AST** = Abstract Syntax Tree, a tree representation of a program's code. If you are confused, you can think of it as just the inner representation of a program in the MPS tool.
- **Base language** = a language defined in MPS upon which other languages are based and which is being directly transformed into textual source files during the final stage of the source-building process
- **Base Language** = a base language representing Java
- **Concept** = definition of an element of a DSL inside the Structure language aspect
- **Domain-Specific Language** = see DSL
- **DSL** = Domain-Specific Language, a language intended to be used for a specific domain
- **Editor** = a definition of how an AST node will be displayed to the user in MPS
- **End-user of MPS** = a user who utilizes a language defined in MPS to develop programs
- **General-Purpose Language** = see GPL
- **GPL** = General-Purpose Language, a language which can be used for more-or-less any program
- **Intentions** = definition of quick actions offered at some location in code. E.g. negating of a condition in an if statement.
- **Language aspect** = a part of DSL definition in the MPS tool
- **Language designer** = a user of JetBrains MPS who designs and defines domain-specific languages in it
- **Language workbench =** a tool used for defining and composing DSLs, together with creation of integrated development environments for them and solving the DSL code execution
- **MPS** = a language workbench developed by JetBrains
- **Projectional editor** = an editor where the user edits directly the AST of the program, not the text representation of the program
- **Root concept** = a concept which can be used for a root node of an AST
- **Structure** = a definition of a language's hierarchical structure in MPS
- **TextGen** = a model-to-text transformator used to generate source files from an AST

Appendix

A: Links to C# Standards, Drafts and Proposals:

- ECMA standard ECMA-334 (2017-12):
  https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf
- ISO/IEC 23270 (2018-12):
  https://standards.iso.org/ittf/PubliclyAvailableStandards/c075178_ISO_IEC_23270_2018.zip
- Unofficial draft of C# 6:
  https://github.com/dotnet/csharplang/tree/master/spec
- Proposals of C# 7.0, 7.1, 7.2, 7.3:
  https://github.com/dotnet/csharplang/tree/master/proposals
- Proposal of C# 8.0:
  https://github.com/dotnet/csharplang/tree/master/proposals/csharp-8.0

B: List of Figures