FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

C# Base Language for MPS

(CS4MPS)

Documentation

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

One of general recommendations for modern software development is to use as high-level abstractions and tools as possible to reduce development effort and increase efficiency of delivering business value. One of the most known abstractions is programming language.

The taxonomy of programming languages consists of many kinds of classifications. One of them speaks about domain-specific languages. Their main characteristic is their concentration on a given application domain, allowing the developer to ignore programming-specific details.

To be able to use a domain-specific language, one must be equipped with the language's 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 code in a domain-specific language that 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 project context in a greater level of detail.

1.1 Domain-Specific Languages

The definition of *domain-specific languages* (DSL) is rather blurry. Generally speaking, a DSL 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.

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 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 or MATLAB are representatives of DSLs. Some might even consider shell scripting languages to be DSLs as well.

DSLs are usually used to help people express a domain-specific algorithm in an easy and quick way without necessity of 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 the development, it is annoying for the developers and it is error-prone. This is the place where a DSL could do these repetitive activities implicitly, without any action from the developers. This would obviously increase efficiency of the development, increase the 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.

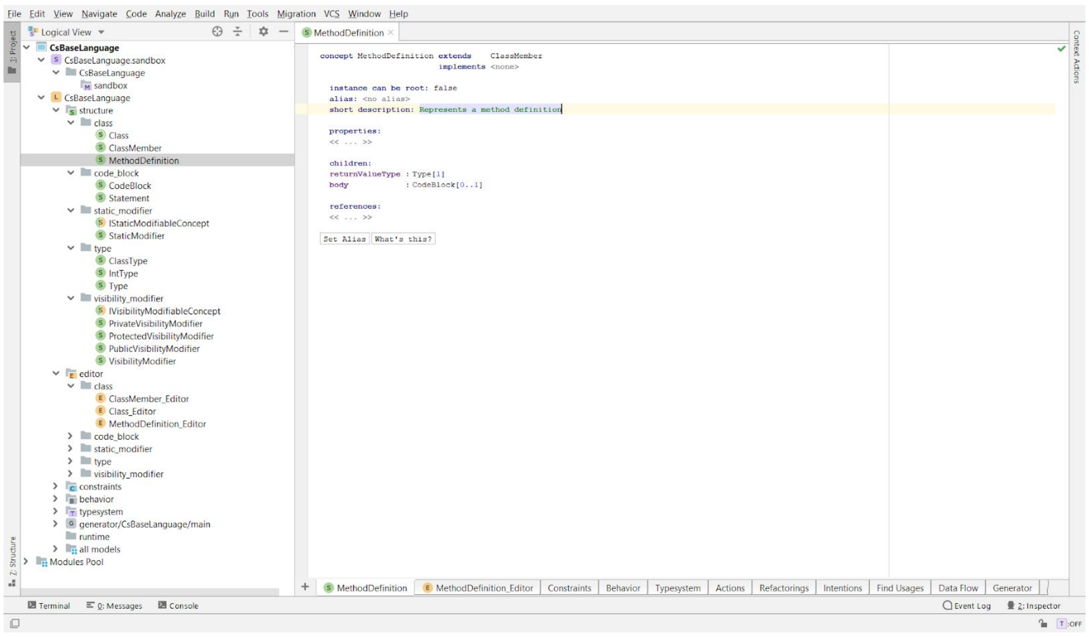
As can be seen from the examples above, DSLs can be very helpful. However, how can one get a DSL, write code in it and execute the code? There is no standardized way to do that.

Martin Fowler, a famous British software engineer, introduced the term *language workbench* that 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 users to create a custom DSL, IDE for it and provides a solution for execution of the DSL code.

The main GUI window of the MPS tool is captured in Figure 1.1. The right area serves as the editor and the left area as the project explorer, similarly to well-known IDEs.



```
concept MethodDefinition extends    ClassMember
                         implements <none>

  instance can be root: false
  alias: <no alias>
  short description: Represents a method definition

  properties:
  << ... >>

  children:
  returnValueType : Type[1]
  body            : CodeBlock[0..1]

  references:
  << ... >>

Set Alias  What's this?
```

Figure 1.1: The main window 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 into the work of end-users and then let’s introduce 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 that 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 structure of the code elements. The transformation actions may look like writing code into text-cells, connecting figures together, using shortcuts or other actions.

This design of projectional editing of code has several advantages:

- Alice can design the language so that it has no text form. Bob might be allowed to use tables in the code, to connect figures into diagrams or anything what is considered to be user-friendly for the given domain. This is probably the greatest advantage.
- The language may provide multiple alternative notations, allowing Bob to choose a notation that best fits the task at hand - implementation, testing, debugging, 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, which is a non-trivial task, and concentrate more on other value of the software.

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 the authors of MPS put a lot of effort into making the projectional editing user-friendly, and to a great degree they 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 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 code.

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*. For example:

- *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), or both.

More details about Alice's work are left for Section *1.2 JetBrains 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 and only 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 do 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, physical units or other.

The current version of MPS supports the following base languages (some of them are not programming languages):

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

Now, when the context of our project has been introduced, it is the right time to introduce the project itself.

1.3 This Project

The main goal of our project was to extend the set of supported base languages in JetBrains MPS by a base language mirroring the C# general-purpose language.

This makes the MPS tool accessible to users who prefer the C# ecosystem over the Java one. There are two basic use-cases of our project:

- it allows language designers to make their current and new DSLs transformable into the C# general-purpose language,
- it allows end-users to use the C# (base) language directly in their programs.

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

The goal of this project was not to provide a C# base language corresponding to full and latest C# version but rather a base language corresponding to the most commonly used part of the C# general-purpose language.

Detailed specification of the project's goals will be given in Section *3.1 Project Goals*.

1.4 This Document

Since it might be confusing what we mean by saying C#, we strictly distinguish two terms in the following text of this document. By *C#* we mean the traditional, well-known programming language and by *C# base language* we mean the the base language designed in the JetBrains MPS tool mirroring the traditional C#.

This document serves as a detailed documentation of our 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 JetBrains MPS tool, which forms the environment in which this project has been developed, from a point of view of language designers.

The following, third, chapter provides more detailed description of this project itself covering the implementation details and alternative solution discussions.

In the following fourth chapter, the reader can find several tutorials about the deployment and usage of our product.

The fifth chapter introduces some simple examples of the usage of the final product.

In the sixth chapter, we evaluate the project. We mainly discuss satisfaction of the original goals and list ideas for future work on this project.

The last chapter contains details about the process of the project development, listing interesting techniques that we used.

And finally, in the appendix, one can find a glossary of context-specific terms, links to the project’s components and external resources.

2 JetBrains MPS

This chapter introduces the tool JetBrains MPS in further details, mainly from the point of view of a language designer.

2.1 Workflow

The basic principles of the JetBrains MPS tool were covered by Section *1.2 JetBrains MPS*. The basic idea is that end-users create their programs via a projectional editor, directly as abstract syntax trees, not in textual form. These ASTs are then transformed by model-to-model and model-to-text transformations into general-purpose language source code.

This is basically the end-user's point of view of the MPS tool. Language designers, however, sees the tools in a different view. They use a set of tools and MPS-specific languages by which they can create a DSL environment. When creating a DSL, the first thing language designers do is defining the DSL by describing its grammar. Then they specify how the DSL abstract syntax trees can be viewed and edited. And after that, they describe the model-to-model and/or model-to-text transformations. Each of these aspects of the language designer's work requires usage of a different tool and a different MPS-specific language. JetBrains MPS uses a term for these toolset - *language aspect*. Different language aspects will be described in Section *2.3 Language Aspects* below.

But first, in the following section, let us introduce some terminology around project organizational units in the JetBrains MPS.

2.2 Structural Units

As any other IDE on the market, JetBrains MPS structures the programs into different-level organizational or structural units.

The top-level unit is a *project* which then consists of *modules*. There are three kinds of modules: *solutions*, *languages* and *devkits*.

The devkit module is not important for our project and therefore we skip its description.

The language module is a set of language aspects, as introduced above, which cover the DSL definition.

The solution module is basically a program (or more generally, a program module). It consists of *models*[1]. A model is an analogy of packages in Java or namespaces in C# or

[1] Actually, even the other kinds of modules consists of models but they are not so important there.

C++, i.e. it contains a set of tightly-related program elements. Each model then consists of a set of *root nodes* which are roots of abstract-syntax trees (e.g. a root node in the Base Language can be a Java class). The abstract syntax trees are formed from *nodes*.

Each node is an instance of a *concept*. A concept models how a node should look like and what are its relations to other nodes. One can see nodes and concepts as instances and classes in object-oriented programming.

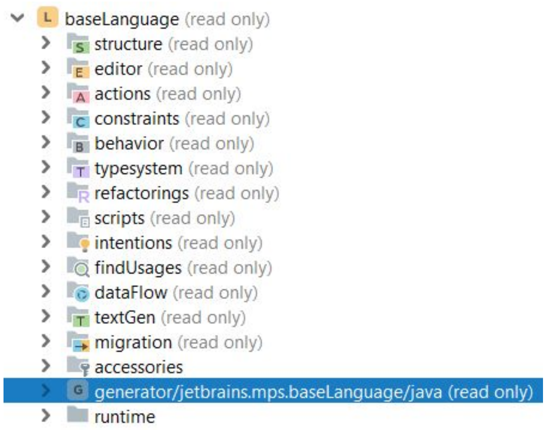
2.3 Language Aspects

In Section *2.1 Workflow*, language aspects have been introduced as toolsets for defining different aspects of a DSL. Each aspect uses a different MPS-specific language, which is a DSL again[2] and as such it is edited via a projectional editor.

Another way of understanding language aspects is as parts of a DSL definition, i.e. products made by the toolsets mentioned above (e.g. an aspect of the grammar definition or an aspect of model-to-model transformations). Some aspects are essential and the DSL would not work without them and some only provide support for the other aspects or make the language more user-friendly for the end-user.

To get an idea of the possible aspects, Figure 2.1 shows different aspects used in the Base Language displayed in the project explorer of the MPS tool.

The following subsections describe the most important language aspects.



```
baseLanguage (read only)
  structure (read only)
  editor (read only)
  actions (read only)
  constraints (read only)
  behavior (read only)
  typesystem (read only)
  refactorings (read only)
  scripts (read only)
  intentions (read only)
  findUsages (read only)
  dataFlow (read only)
  textGen (read only)
  migration (read only)
  accessories
  generator/jetbrains.mps.baseLanguage/java (read only)
  runtime
```

Figure 2.1: Language aspects of the Base Language

2.3.1 Structure

The Structure aspect is the most important aspect of a DSL. It defines the grammar of the language, or more generally, 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.

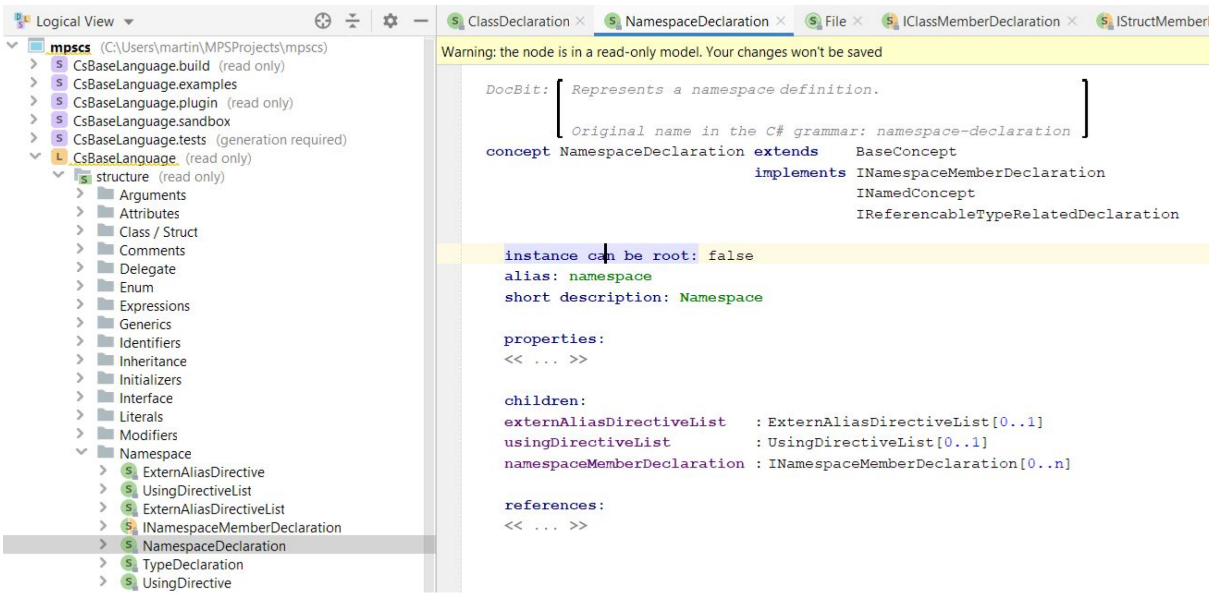
[2] Partly consisting of the Base Language DSL so some parts look like Java.

The language elements are defined via concepts, which have been already mentioned in Section *2.2 Structural Units*.

A concept mainly consists of:

- IS-A relations to ancestor 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 classes have methods.
- Properties. A property can be of a basic type (e.g. a name of the instantiated node) or of a type of a reference to another AST node (a link across the AST, e.g. from a variable assignment to a declaration of the variable).

Figure 2.2 shows an example of a concept definition, concretely the definition of a concept corresponding to the namespace language element in our C# base language.



```
mpscs (C:\Users\martin\MPSProjects\mpscs)
  CsBaseLanguage.build (read only)
  CsBaseLanguage.examples
  CsBaseLanguage.plugin (read only)
  CsBaseLanguage.sandbox
  CsBaseLanguage.tests (generation required)
  CsBaseLanguage (read only)
    structure (read only)
      Arguments
      Attributes
      Class / Struct
      Comments
      Delegate
      Enum
      Expressions
      Generics
      Identifiers
      Inheritance
      Initializers
      Interface
      Literals
      Modifiers
      Namespace
        ExternAliasDirective
        UsingDirectiveList
        ExternAliasDirectiveList
        INamespaceMemberDeclaration
        NamespaceDeclaration
        TypeDeclaration
        UsingDirective
```

Warning: the node is in a read-only model. Your changes won't be saved

```
DocBit: [ Represents a namespace definition.

          Original name in the C# grammar: namespace-declaration ]
concept NamespaceDeclaration extends    BaseConcept
                             implements INamespaceMemberDeclaration
                                        INamedConcept
                                        IReferencableTypeRelatedDeclaration

  instance can be root: false
  alias: namespace
  short description: Namespace

  properties:
  << ... >>

  children:
  externAliasDirectiveList    : ExternAliasDirectiveList[0..1]
  usingDirectiveList          : UsingDirectiveList[0..1]
  namespaceMemberDeclaration  : INamespaceMemberDeclaration[0..n]

  references:
  << ... >>
```

Figure 2.2: The Structure aspect illustration (C# base language)

2.3.2 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.

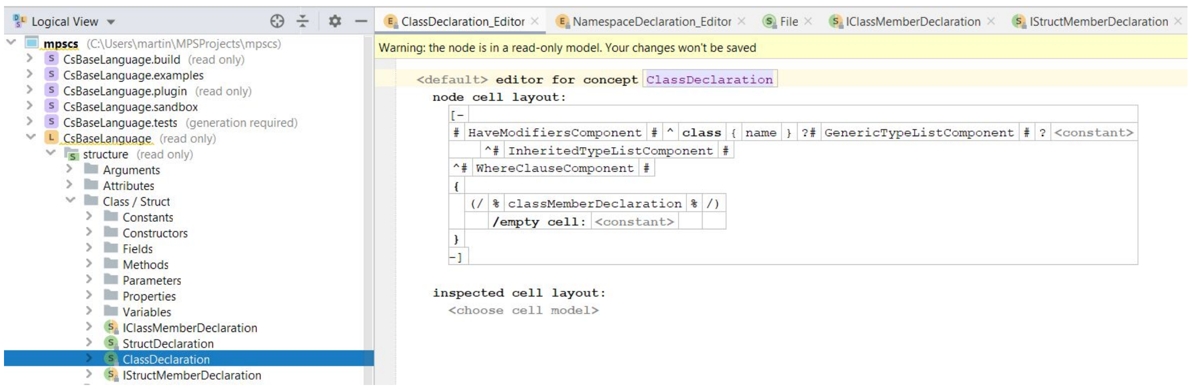
Each editor consists of *editor cells*, which display constants, individual children or properties related the concept.

Figure 2.3 illustrates an editor for the class concept in our C# base language. Figure 2.4 shows advanced editor settings for one editor cell of the editor from Figure 2.3. One can notice that MPS uses quite a complex language and options for the Editor aspect so that it can provide the language designers with different layouts of text, visibility settings, 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 2.5 illustrates how an empty class would look like in the projectional editor if the class concept did not have the editor defined.

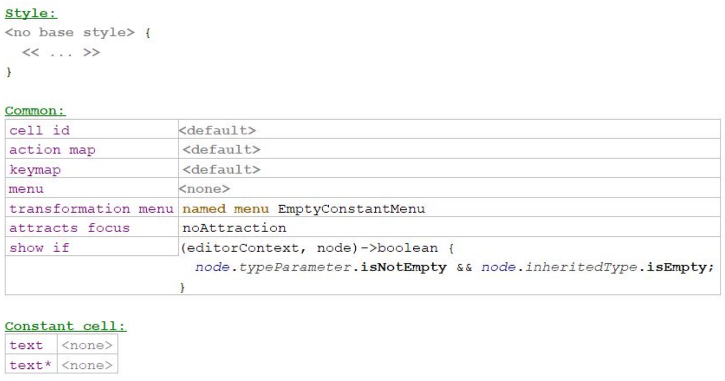
The Editor aspect also offers ways how to make DSL-code-writing more user friendly through different auto-completion menus, auto-transformation actions and similar features.



Warning: the node is in a read-only model. Your changes won't be saved

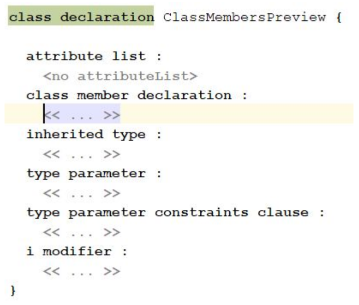
```
<default> editor for concept ClassDeclaration
  node cell layout:
    [-
    # HaveModifiersComponent # ^ class { name } ?# GenericTypeListComponent # ? <constant>
        ^# InheritedTypeListComponent #
    ^# WhereClauseComponent #
    {
      (/ % classMemberDeclaration % /)
         /empty cell: <constant>
    }
    -]

  inspected cell layout:
    <choose cell model>
```

Figure 2.3: The Editor aspect illustration (C# base language)



```
Style:
<no base style> {
  << ... >>
}

Common:
cell id               <default>
action map            <default>
keymap                <default>
menu                  <none>
transformation menu   named menu EmptyConstantMenu
attracts focus        noAttraction
show if               (editorContext, node)->boolean {
                        node.typeParameter.isNotEmpty && node.inheritedType.isEmpty;
                      }

Constant cell:
text   <none>
text*  <none>
```

Figure 2.4: The advanced settings of one of the editor cells from Figure 2.3



```
class declaration ClassMembersPreview {

  attribute list :
    <no attributeList>
  class member declaration :
    << ... >>
  inherited type :
    << ... >>
  type parameter :
    << ... >>
  type parameter constraints clause :
    << ... >>
  i modifier :
    << ... >>
}
```

Figure 2.5: The default editor for an empty class from our C# base language

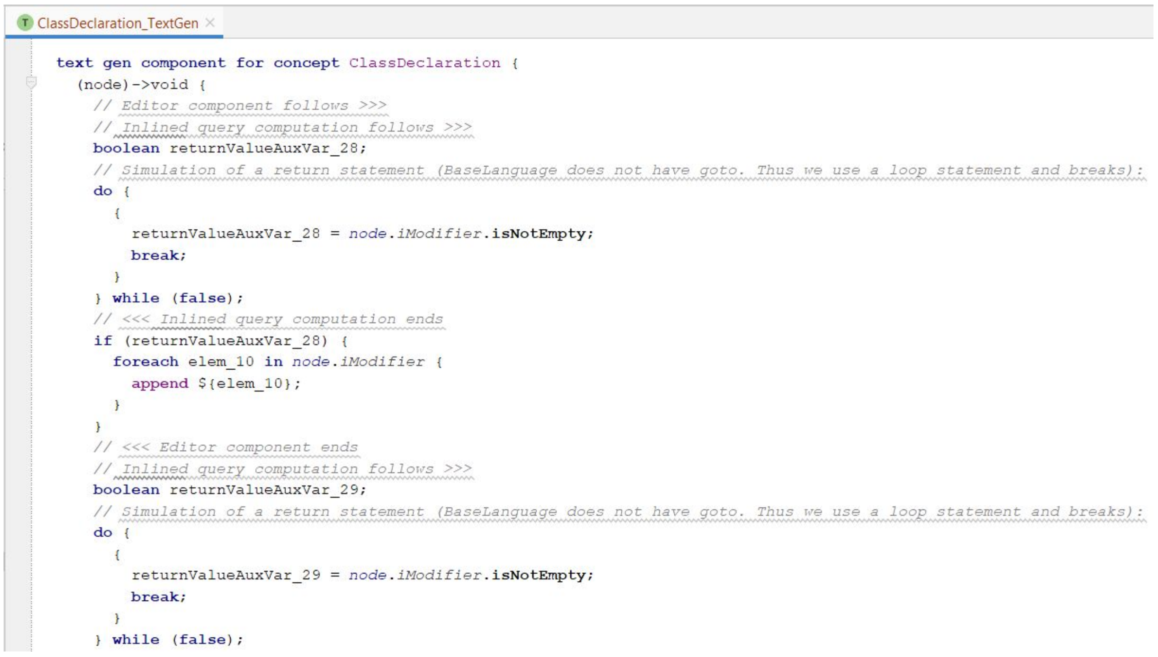
2.3.3 Generator and TextGen

As described earlier, the process of execution of programs developed in an MPS DSL consists of optional model-to-model transformations, which transform the program from one DSL to another DSL, and a mandatory model-to-text transformation, which transforms a DSL program into a GPL program. The Generator aspect defines the model-to-model transformation for a given DSL into a selected another DSL while the TextGen aspect defines the model-to-text transformation for a given DSL into a selected GPL.

Standard DSLs, which are used by most of the MPS end-users, have only the Generator aspect defined, which allows them to be transformable into another DSLs and finally into a base language, which is, as mentioned earlier, also a DSL. The base languages, however, have not the Generator aspect defined, whereas they utilize the TextGen aspect. This allows the base language programs to be transformable into a GPL source code which can then be executed by standard tools of that particular GPL.

Even though it may seem surprising, both the aspects have quite similar character. Each of them defines how to transform a given source chunk of code into a target chunk of code. The TextGen aspect transforms AST nodes into target GPL source code and the generator aspect transforms AST nodes into target DSL code (i.e. target DSL AST subtree).

Figure 2.6 shows an example of a TextGen definition for the class concept of our C# base language[3]. The language of the TextGen aspect is characteristic by *append* commands, which append text to a buffer which gets inserted into the resulting source code file at the place that corresponds to the transformed AST node.

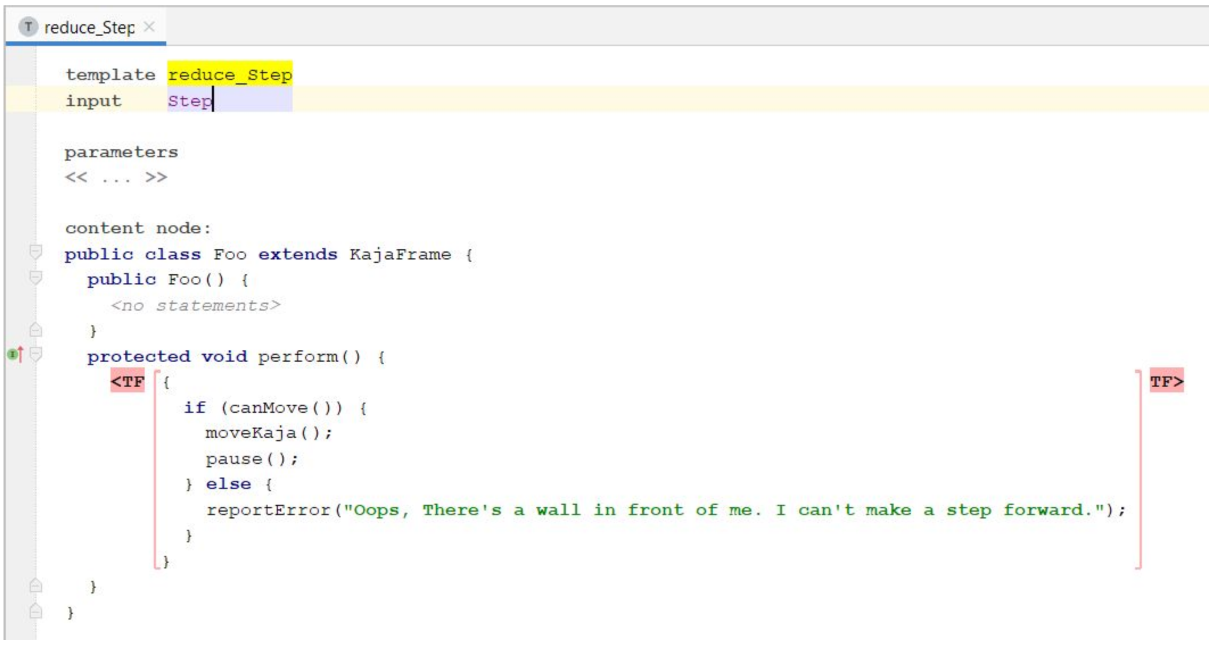


```
text gen component for concept ClassDeclaration {
  (node)->void {
    // Editor component follows >>>
    // Inlined query computation follows >>>
    boolean returnValueAuxVar_28;
    // Simulation of a return statement (BaseLanguage does not have goto. Thus we use a loop statement and breaks):
    do {
      {
        returnValueAuxVar_28 = node.iModifier.isNotEmpty;
        break;
      }
    } while (false);
    // <<< Inlined query computation ends
    if (returnValueAuxVar_28) {
      foreach elem_10 in node.iModifier {
        append ${elem_10};
      }
    }
    // <<< Editor component ends
    // Inlined query computation follows >>>
    boolean returnValueAuxVar_29;
    // Simulation of a return statement (BaseLanguage does not have goto. Thus we use a loop statement and breaks):
    do {
      {
        returnValueAuxVar_29 = node.iModifier.isNotEmpty;
        break;
      }
    } while (false);
```

Figure 2.6: The TextGen aspect illustration (C# base language)

[3] As will be discussed later, the TextGen aspect of our language is auto-generated. Therefore the code constructions in Figure 2.6 might seem strange.

Figure 2.7 illustrates a Generator definition for one of the concepts in one of the example DSL languages that come bundled with the Jetbrains MPS tool. One can see a transformation definition for an AST node of concept *Step* into an AST subtree in Base Language.



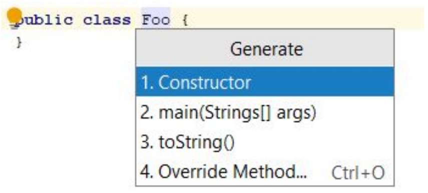
```
reduce_Step

template reduce_Step
input    Step

parameters
<< ... >>

content node:
public class Foo extends KajaFrame {
  public Foo() {
    <no statements>
  }
  protected void perform() {
    <TF [{
      if (canMove()) {
        moveKaja();
        pause();
      } else {
        reportError("Oops, There's a wall in front of me. I can't make a step forward.");
      }
    }] TF>
  }
}
```

Figure 2.7: The Generator aspect illustration (Robot Kaja language)

2.3.4 Intentions

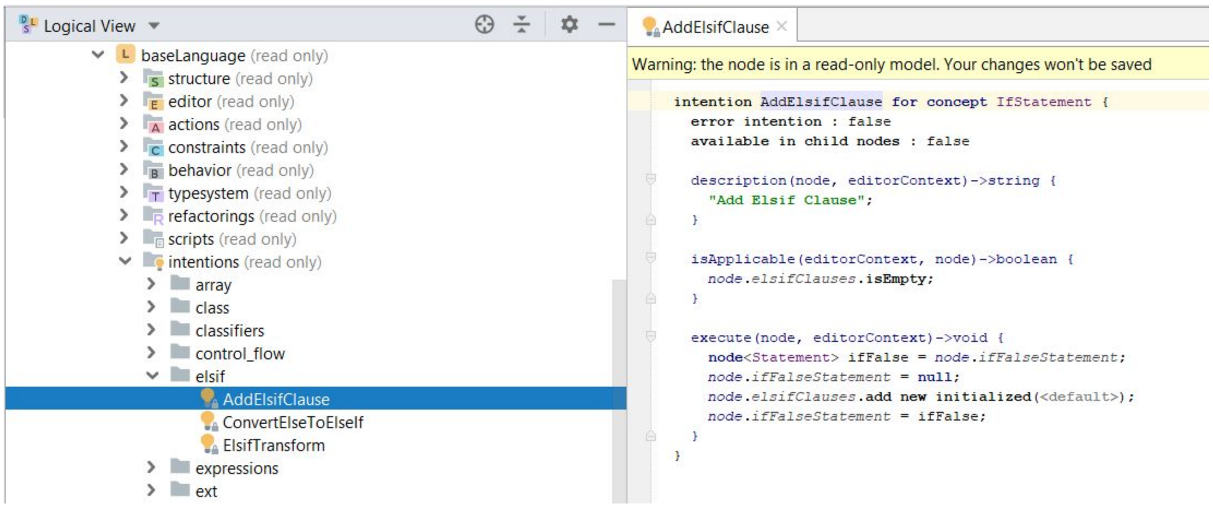
The Intentions aspect defines quick actions which the end-user can initiate when he or she positions the cursor at some location in code. An example of these actions is illustrated in Figure 2.8. Intentions serve to improve the usability of the IDE providing shortcuts to accomplishing desired code modifications.

Figure 2.9 shows the Intentions aspect definition for an intention of adding an *else-if* clause to an *if* statement.



```
public class Foo {
}
```

Figure 2.8: Example of an intentions dropdown menu (Base Language)



Warning: the node is in a read-only model. Your changes won't be saved

```
intention AddElsifClause for concept IfStatement {
  error intention : false
  available in child nodes : false

  description(node, editorContext)->string {
    "Add Elsif Clause";
  }

  isApplicable(editorContext, node)->boolean {
    node.elsifClauses.isEmpty;
  }

  execute(node, editorContext)->void {
    node<Statement> ifFalse = node.ifFalseStatement;
    node.ifFalseStatement = null;
    node.elsifClauses.add new initialized(<default>);
    node.ifFalseStatement = ifFalse;
  }
}
```

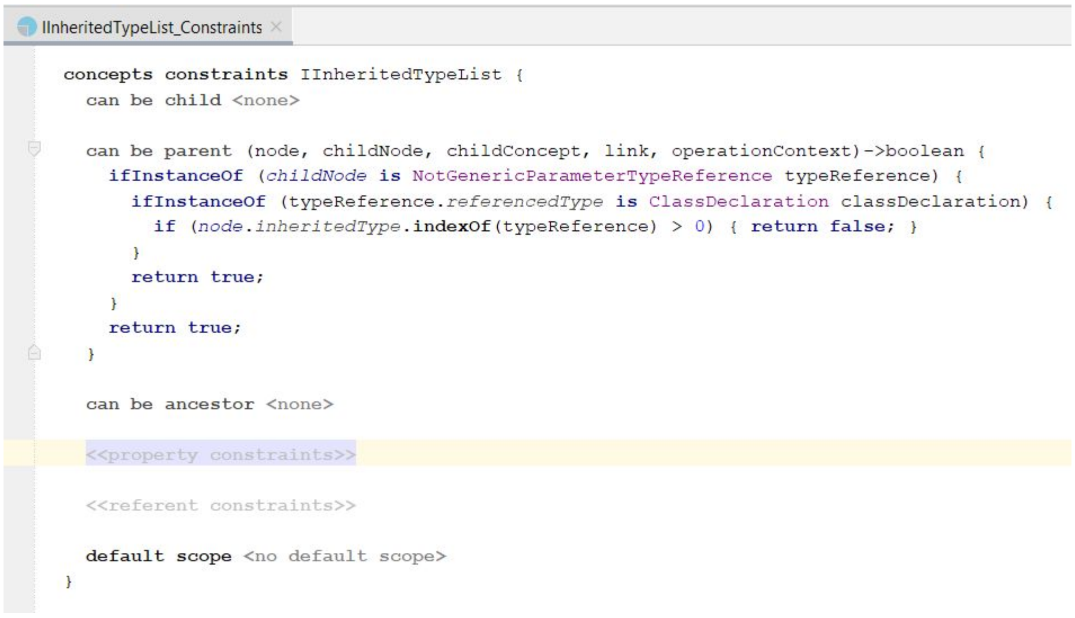
Figure 2.9: The Intentions aspect illustration (Base Language)

2.3.5 Constraints

The Constraints aspect provides the language designer with better control of how AST nodes can be composed together. The Structure aspect defines only basic rules such as what AST nodes can be parents or children of what AST nodes. The Constraints aspect can set more detailed conditions based on the exact, final AST structure in a program.

For example, one can forbid some order of children of some AST node. This can be for example applied when we want to force inherited classes to appear as first in an inheritance list, before any implemented interfaces.

Figure 2.10 illustrates a definition of the Constraints aspect which controls the order of inherited classes and interfaces, exactly as suggested in the previous paragraph.



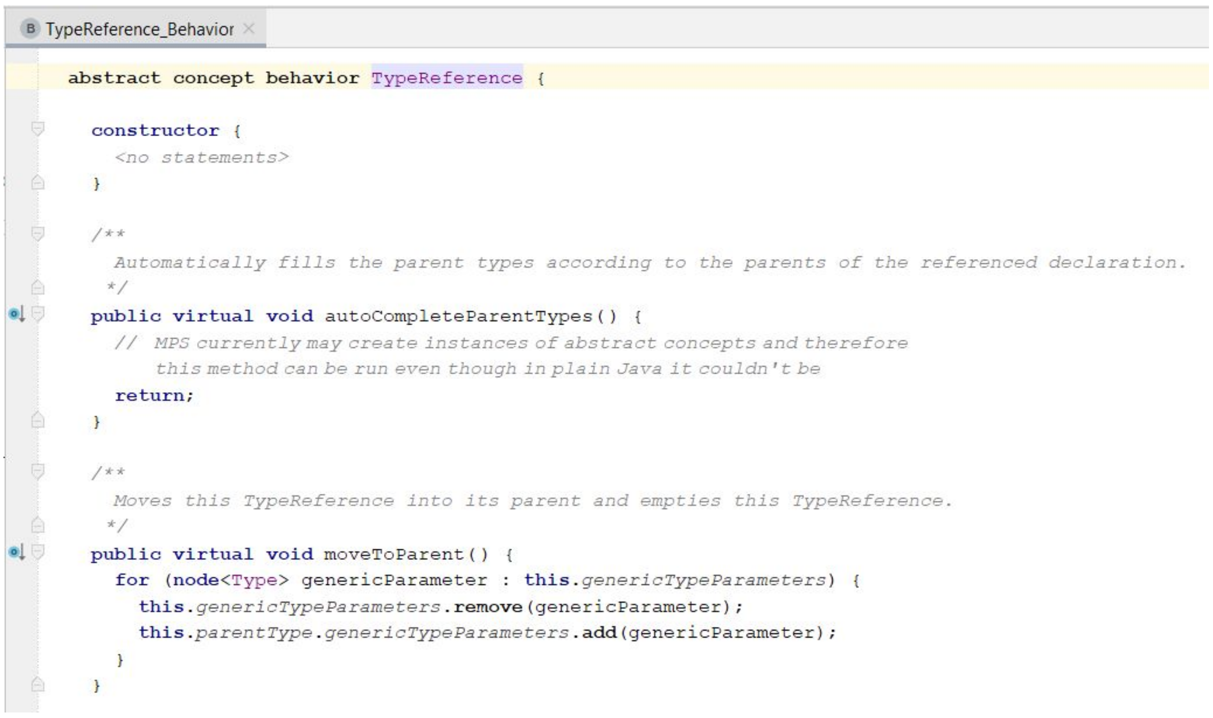
```
concepts constraints IInheritedTypeList {
  can be child <none>

  can be parent (node, childNode, childConcept, link, operationContext)->boolean {
    ifInstanceOf (childNode is NotGenericParameterTypeReference typeReference) {
      ifInstanceOf (typeReference.referencedType is ClassDeclaration classDeclaration) {
        if (node.inheritedType.indexOf(typeReference) > 0) { return false; }
      }
      return true;
    }
    return true;
  }

  can be ancestor <none>

  <<property constraints>>

  <<referent constraints>>

  default scope <no default scope>
}
```

Figure 2.10: The Constraints aspect illustration (C# base language)

2.3.6 Behavior

The Behavior aspect allows defining methods for concepts from the Structure aspect. These methods can be used in other language aspects providing the language designer with a way of how to encapsulate language definition related code, use polymorphism or other features of the object-oriented programming.

Figure 2.11 shows an example of a Behavior definition for a concept in our C# base language.


TypeReference_Behavior

```
abstract concept behavior TypeReference {

  constructor {
    <no statements>
  }

  /**
    Automatically fills the parent types according to the parents of the referenced declaration.
   */
  public virtual void autoCompleteParentTypes() {
    // MPS currently may create instances of abstract concepts and therefore
       this method can be run even though in plain Java it couldn't be
    return;
  }

  /**
    Moves this TypeReference into its parent and empties this TypeReference.
   */
  public virtual void moveToParent() {
    for (node<Type> genericParameter : this.genericTypeParameters) {
      this.genericTypeParameters.remove(genericParameter);
      this.parentType.genericTypeParameters.add(genericParameter);
    }
  }
```

Figure 2.11: The Behavior aspect illustration (C# base language)

2.3.7 Other aspects

The aspects introduced in the previous sections are only a selection of the most important aspects. There are another language aspects controlling for example type correctness (the Type System aspect[4]), data flow and/or other features.

2.4 Stubs

To be able to develop programs efficiently, a developer needs libraries. The most used, and usually necessary to use, is the standard library for the selected programming language. Libraries usually relate to general-purpose languages. There are often no libraries for domain-specific languages, although there are exceptions.

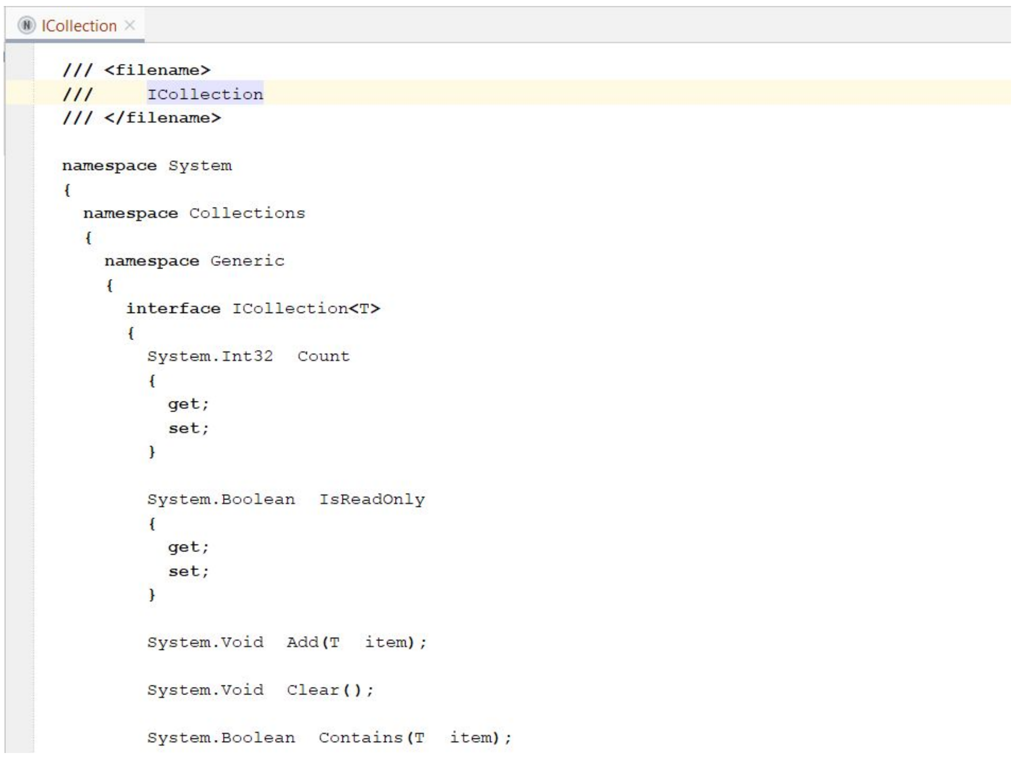
[4] Developing a Type System aspect for a complex language is difficult. For example, in case of Base Language, it is not fully implemented even though it has been developed for several years.

In order to be able to use a library, one needs a description of the library's API[5]. In C++, for instance, this description are the header files corresponding to that library. Basically, the point is to describe what procedures the library's user can call and what structures he or she can use.

The MPS, expectedly, also supports usage of libraries, in a similar form to the building process of C++. The end-user of the DSL is provided with so-called stubs. These form the description of the library's API discussed above. They are simple abstract syntax trees which are simply programs that contain just declarations and no definitions. For example, in case of our C# base language, they are only classes, structures or methods without any implementation, which can be called or used. They are casual ASTs just like in case of their real implementation, but without definitions of the methods.

The end-users are able to call or use these stubs simply, in the same way as they would develop these stubs themselves. When they complete their DSL program, they can generate the GPL source code files for the program which may then be executed. The stubs are not used during this process, they do not appear in the generated GPL source code files at all. They were only needed for development of the DSL program inside MPS to be able to appropriately and correctly construct references across the code (e.g. from a method call to the method's declaration). As mentioned before, the entire process is really close to the process in C++.

In Figure 2.12, one can see an example of a stub for the *ICollection* interface in our C# base language.



```
/// <filename>
///     ICollection
/// </filename>

namespace System
{
  namespace Collections
  {
    namespace Generic
    {
      interface ICollection<T>
      {
        System.Int32  Count
        {
          get;
          set;
        }

        System.Boolean  IsReadOnly
        {
          get;
          set;
        }

        System.Void  Add(T  item);

        System.Void  Clear();

        System.Boolean  Contains(T  item);
```

Figure 2.12: Stub example

[5] In fact, we need to know the Application Binary Interface (ABI), but that it too much detail for us now.

3 C# base Language

This chapter describes what has been done in the project, analyses particular problems which occurred during the development and presents the solutions of those problems.

3.1 Project Goals

The main goal of our project was to deliver a C# base language, which could serve those who want their MPS solutions to be compilable for the .NET platform.

The primary intended purpose of our base language is to be a target base language of model-to-model transformations for MPS DSLs so that the DSL programs can be compilable into C# source code.

This means that the base language must have supported the basic C# functionality so that the language designers can express the C# code chunks they need, even though that the editing might be a little less user-friendly than in case of text-based C# IDEs. At the same time, we needed to allow users to write C# program directly in our base language. This has similar requirements.

This goal comprised development of the main language aspects enlisted in the following list. Other language aspects were used to support them.

- The Structure aspect. Our base language supports commonly used features of the C# general-purpose language[6]. The language structure is extensible as new features will be probably needed in the future.
- The TextGen aspect, which produces pretty-formatted C# source code.
- The Editor aspect, which ensures formatting of the C# base language code similarly to C# code standards.

Fulfilling of the project's goal also required to implement user-friendly editing of the code. This feature is, in the further text, called as *fluent editing*.

An indirect part of the main goal was also delivering a solution of how to interconnect C# (standard) libraries with the C# base language code.

As a secondary goal, we delivered a proper documentation of our solution and its development.

There were also other requirements for the project which do not have a character of goals:

- The solution must have been compatible with the latest MPS version.

[6] Features that we decided to support are not listed here but it is discussed further in the text as the list is too long.

- All to-be-published documents shall have been provided in English.
- The main parts of the solution shall have been released under the Apache 2.0 license, consistently with the MPS standard.

3.2 Solution Overview

In this section, we provide an overview of the solution parts. The details around individual parts are discussed later in the text.

The core of the solution is, of course, the C# base language itself, distributed as an MPS plugin. It consists of several language aspects, namely Structure, Editor, TextGen, Intentions, Actions, Constraints and Behavior.

Since the TextGen aspect for this kind of base languages is very similar to the Editor aspect, it is convenient to automatically generate it from the Editor. This idea is behind a plugin that generates the TextGen aspect from the Editor aspect. When we started our work on the project, a basic version of this plugin existed. However, it supported only the most basic Editor features. As a part of our solution, we significantly improved the plugin's functionality so that it fully supports such complex generation as we needed for our base language.

The language contains also tests created in an MPS built-in tool. These tests should ease the work of future maintainers of the project.

Furthermore, in order to support usage of the C# standard library in C# base language code, we needed to implement also the following:

- A parser of DLL libraries containing C# assemblies, which creates an XML file containing a specification of the library declarations.
- Two MPS plugins parsing the XML file from the DLL parser and creating MPS library stubs.
- An MPS plugin wrapping the standard library so that it can be easily distributed to users.

The project also contains a set of examples of our product usage, which are particularly important for users searching for help about how to deploy our solution.

Last but not least, we also delivered an extensive documentation of all the parts of the solution, which is important for future maintenance of the language and development of another base languages.

3.3 Language

This section presents the developed C# base language. We specify the supported features, describe the inspirational resources, describe the language solution parts, the distribution format and discuss some alternative solutions.

3.3.1 Supported C# Features

There are multiple versions of the C# programming language, which differ in a nontrivial manner. In this section, we discuss which version we decided to use for our base language and/or what features we decided to support.

3.3.1.1 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 were not considered to be appropriate 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 was expected to correspond to a C# version for non-deprecated platforms.

The version 5.0 of C# is the newest version that has been, at the time of development, officially standardized by ECMA and ISO. This is a great advantage of this version as its formal description was 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, which would have caused problems during our development.

The last version 8.0 of C# is currently in development and not released yet. This makes it not an appropriate candidate for the 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. This helped us a lot during the language definition process. Furthermore, there was no risk that the specification would change during implementation of the base language and our language definition would be invalidated.
- Implementation of support of newer versions is possible. It is left primarily for community around the MPS tool. It should not be as difficult as implementing the C# base language from scratch because the future versions of the C# base language are expected to just extend our version.
- Furthermore, 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 support only its most important features. The following sub-section lists the supported parts of the C# 5.0 language.

3.3.1.2 Supported Features

This subsection contains a list of features that are included in almost every C# program. They are the core features that had to 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,
    - Object.

- Literals (full syntax support)
- Variables:
    - Static,
    - Instance,
    - Local,
    - Arguments (value, output, reference).
- Generics (definition and usage)
- Comments
- Expressions
    - Method calls,
    - Assignment,
    - Operators.
- 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,
    - Partial support of checked/unchecked statements,
    - Partial support of lock statements,
    - Partial support of try statements.
- Delegate + modifier
- Partial types (only syntactically)

3.3.2 Language Structure Inspiration Resources

In this section, we discuss which resources we used in order to design the C# base language well and to ensure that the structure satisfies all promised features. We also present other possible resources that could have been used for this purpose and the reasons why we did not use them.

The main resource that we used to design the language structure was the freely-available grammar of C# 5.0. We transformed a lot of the grammar symbols into MPS Structure concepts almost one-to-one. The other grammar symbols were either too low-level and the MPS supports them automatically, or were skipped as we do not support the corresponding features, or were designed differently. On the contrary, some concepts are not in the grammar as well as those features were designed purely for purposes of fitting the language into MPS.

A lot of inspiration also came from the solutions used in the Base Language (i.e. the Java base language). However, it has no documentation and some of its parts are deprecated - thus, in many cases, it is not a good resource of inspiration as one is not able to figure out exactly the idea behind. Furthermore, Java and C# are not the same languages and there are parts which are quite different.

There are some other possible inspiration resources:

- There is an existing diploma thesis *Grammar to JetBrains MPS Convertor*[7] which generates the Structure aspect from a grammar of the language. We decided not to base 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.
- There are also the specification of C# 5.0 and similar resources from Microsoft. However, they are rather designed to provide information about special cases than about the language hierarchical structure.

3.3.3 Language Aspects

The C# base language consists of the following language aspects:

- Structure,
- Editor,
- TextGen,
- Intentions,
- Actions,
- Constraints,
- Behavior.

There are also other language aspects which could be implemented by a base language. A good example representing what aspects can be used for a base language is presented in Figure 2.1 in Section *2.3 Language Aspects*. The most significant language aspect we did not implement is Type System. Full implementation of this aspect is out of scope of this project as it is not fully implemented even for the Base Language, which has been developed over several years.

The core aspect of any language is the Structure aspect which more-or-less represents the language's grammar. The closely-related Editor aspect is necessary to provide the user with friendly editing of the C# base language code. These two aspects are the largest aspects in our base language. The triple of the most important aspects is completed by the TextGen aspect. As has been mentioned before, we generate TextGen from the Editor aspect.

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

The other listed aspects have only supporting character. Being diverse in their size, the least is the Intentions aspect while the largest of them is probably Behavior.

In the sub-sections below, we selected and described the most important areas and design principles of our base language.

3.3.3.1 Constraining the Language Structure

When designing an MPS language, one must control what the language's user will and will not be able to do. The goal of this effort is not to restrict the user's freedom but it is prevention from coding errors. Ideally, these errors should be discovered and reported to the user as soon as possible to make his work efficient.

There are three basic ways of how to control what combinations of language entities can the user create:

- First, the language designer may concentrate (almost) all the restrictions into the Structure aspect by making the Structure aspect very tight. For example, in the list of inherited entities for a class, we do not have just a list of references to those entities (i.e. list of references of one type), but we differentiate between whether the reference goes to a class or to an interface (i.e. list of references of two types). As in this example, in this design, there are more Structure concepts needed to differentiate various use cases of language entities and the relationships between the language concepts are strictly specified. The advantage of this approach is that the language designers do not have to pay so much attention to the supporting aspects, and that they can leave a lot of work to already functioning automation of MPS, such as auto completion, et cetera.
- Another option is the opposite. The language designer can make the Structure aspect free and control everything by the supporting aspects, mainly by the Constraints aspect. There he needs to specify different rules like restrictions of what concepts, or more precisely AST nodes can be parents or children of what. The advantage of this approach is that the Structure aspect is simple and thus the language is more maintainable.
- The third approach is, as you may have expected, a combination of the two approaches. This may be seen as a good option as one gets advantages of both the previous approaches but some might see it as a bad idea of creating a dirty result.

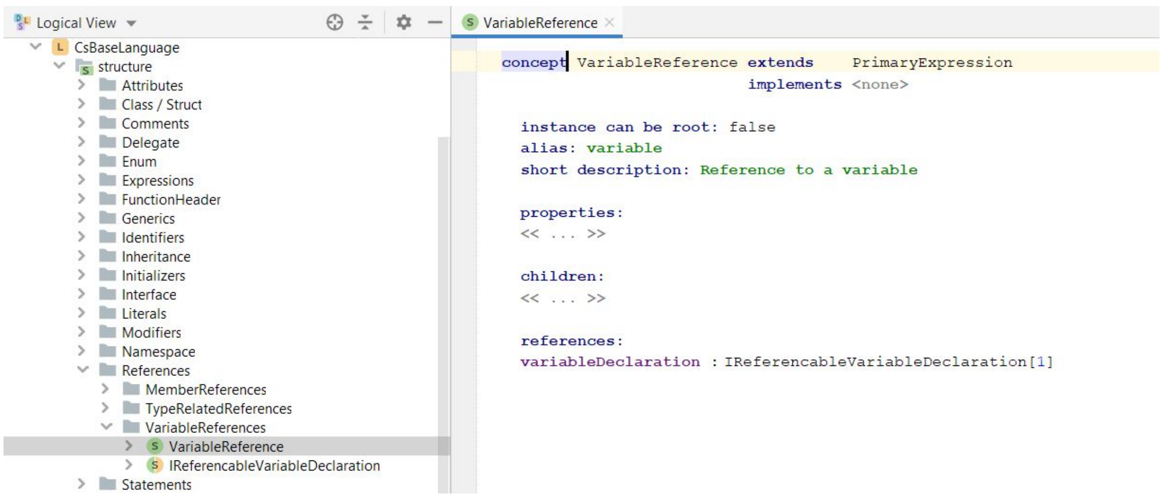
We decided to use the third approach as we believe it is the advantages we gained by this combination, not the bad things.

3.3.3.2 Structure

The Structure aspect design usually follows the official grammar of C# 5.0. At the beginning, we were more-or-less following the freely-available grammar of C# 5.0 and constructing the base language's Structure aspect in the same way.

During this process, we also documented the Structure aspect concepts so that future maintainers of the base language or other developers of other base languages can more easily understand our concepts. As a part of this documentation, at many concepts, we stated the name of the original grammar entity from which the concept originated.

Later on, we needed to adapt the concepts a little bit or create entirely new concepts that have no corresponding entity in the grammar. This was needed to either fit the language into the MPS environment or to complete the language (the available grammar is not complete, it is only a presentation of the grammar). One good example of completely new concepts, which were not precisely covered by the grammar, is implementation of references, e.g. references to type or references to local variables (Figure 3.1). Another example can be concepts that encapsulate some properties or behavior of multiple other concepts, e.g. a concept encapsulating inheritance list from which concepts such as class or interface inherit.

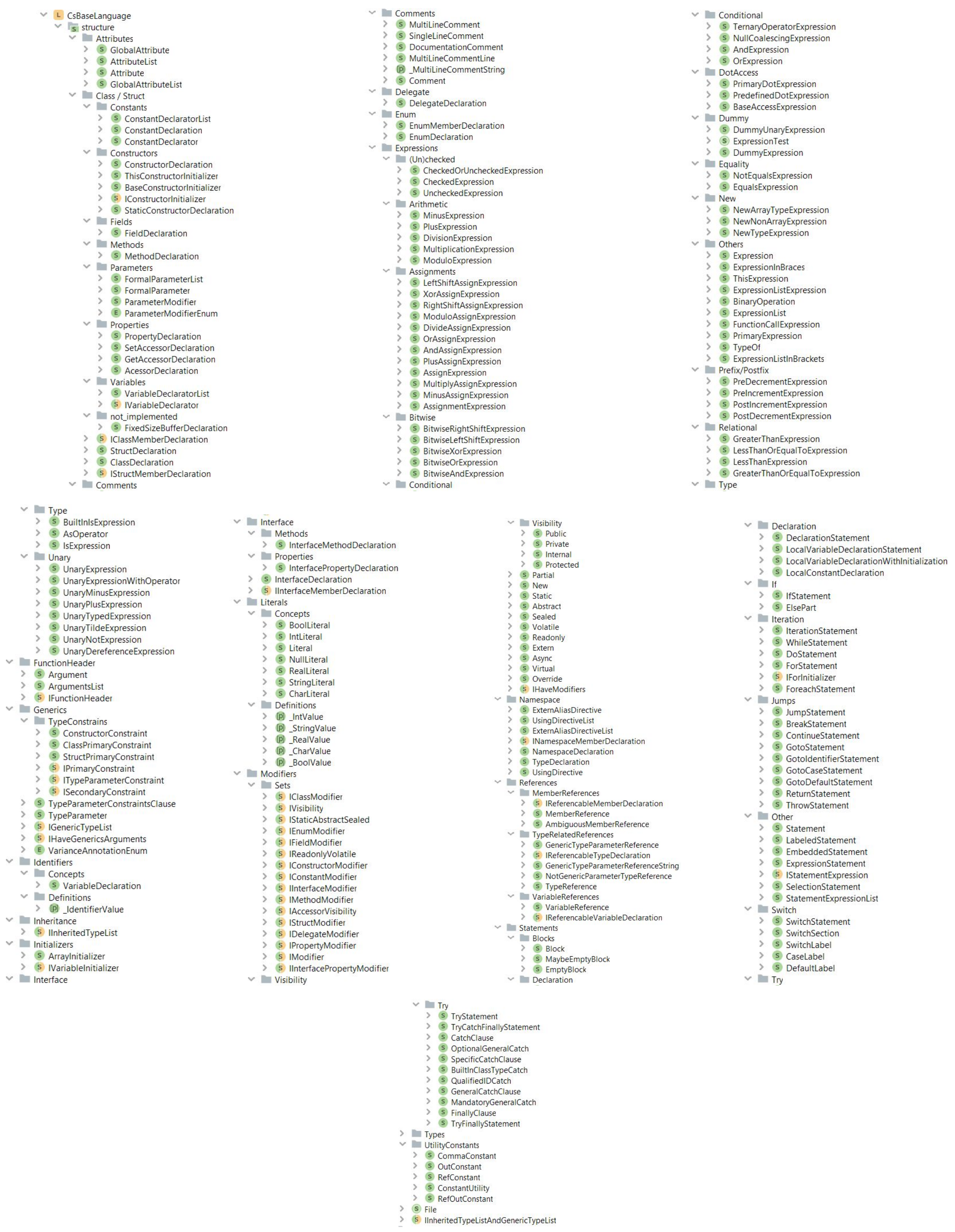


```
concept VariableReference extends     PrimaryExpression
                          implements <none>

  instance can be root: false
  alias: variable
  short description: Reference to a variable

  properties:
  << ... >>

  children:
  << ... >>

  references:
  variableDeclaration : IReferencableVariableDeclaration[1]
```

Figure 3.1: An example of a concept that has no correspondence in the grammar

In Figure 3.2, you can see an overview of concepts that we implemented to cover the promised language. As can be seen, the number of concepts is rather large. Along with the fact that the Structure aspect is only a part of the language, it shows that base languages mirroring the mainstream traditional programming languages are quite non-trivial to implement.

Every MPS language has a set of concepts marked as *root concepts*, i.e. concepts whose AST nodes can become the root nodes. We decided to use only one root concept: *File*. As can be guessed from the name, we were trying to reflect the traditional C# code organization. A *File* AST node can contain classes, namespaces, interfaces, etc.



```
CsBaseLanguage
  structure
    Attributes
      GlobalAttribute
      AttributeList
      Attribute
      GlobalAttributeList
    Class / Struct
      Constants
        ConstantDeclaratorList
        ConstantDeclaration
        ConstantDeclarator
      Constructors
        ConstructorDeclaration
        ThisConstructorInitializer
        BaseConstructorInitializer
        IConstructorInitializer
        StaticConstructorDeclaration
      Fields
        FieldDeclaration
      Methods
        MethodDeclaration
      Parameters
        FormalParameterList
        FormalParameter
        ParameterModifier
        ParameterModifierEnum
      Properties
        PropertyDeclaration
        SetAccessorDeclaration
        GetAccessorDeclaration
        AcessorDeclaration
      Variables
        VariableDeclaratorList
        IVariableDeclarator
      not_implemented
        FixedSizeBufferDeclaration
      IClassMemberDeclaration
      StructDeclaration
      ClassDeclaration
      IStructMemberDeclaration
    Comments
      MultiLineComment
      SingleLineComment
      DocumentationComment
      MultiLineCommentLine
      _MultiLineCommentString
      Comment
    Delegate
      DelegateDeclaration
    Enum
      EnumMemberDeclaration
      EnumDeclaration
    Expressions
      (Un)checked
        CheckedOrUncheckedExpression
        CheckedExpression
        UncheckedExpression
      Arithmetic
        MinusExpression
        PlusExpression
        DivisionExpression
        MultiplicationExpression
        ModuloExpression
      Assignments
        LeftShiftAssignExpression
        XorAssignExpression
        RightShiftAssignExpression
        ModuloAssignExpression
        DivideAssignExpression
        OrAssignExpression
        AndAssignExpression
        PlusAssignExpression
        AssignExpression
        MultiplyAssignExpression
        MinusAssignExpression
        AssignmentExpression
      Bitwise
        BitwiseRightShiftExpression
        BitwiseLeftShiftExpression
        BitwiseXorExpression
        BitwiseOrExpression
        BitwiseAndExpression
      Conditional
        TernaryOperatorExpression
        NullCoalescingExpression
        AndExpression
        OrExpression
      DotAccess
        PrimaryDotExpression
        PredefinedDotExpression
        BaseAccessExpression
      Dummy
        DummyUnaryExpression
        ExpressionTest
        DummyExpression
      Equality
        NotEqualsExpression
        EqualsExpression
      New
        NewArrayTypeExpression
        NewNonArrayExpression
        NewTypeExpression
      Others
        Expression
        ExpressionInBraces
        ThisExpression
        ExpressionListExpression
        BinaryOperation
        ExpressionList
        FunctionCallExpression
        PrimaryExpression
        TypeOf
        ExpressionListInBrackets
      Prefix/Postfix
        PreDecrementExpression
        PreIncrementExpression
        PostIncrementExpression
        PostDecrementExpression
      Relational
        GreaterThanExpression
        LessThanOrEqualToExpression
        LessThanExpression
        GreaterThanOrEqualToExpression
      Type
        BuiltInIsExpression
        AsOperator
        IsExpression
      Unary
        UnaryExpression
        UnaryExpressionWithOperator
        UnaryMinusExpression
        UnaryPlusExpression
        UnaryTypedExpression
        UnaryTildeExpression
        UnaryNotExpression
        UnaryDereferenceExpression
    FunctionHeader
      Argument
      ArgumentsList
      IFunctionHeader
    Generics
      TypeConstrains
        ConstructorConstraint
        ClassPrimaryConstraint
        StructPrimaryConstraint
        IPrimaryConstraint
        ITypeParameterConstraint
        ISecondaryConstraint
      TypeParameterConstraintsClause
      TypeParameter
      IGenericTypeList
      IHaveGenericsArguments
      VarianceAnnotationEnum
    Identifiers
      Concepts
        VariableDeclaration
      Definitions
        _IdentifierValue
    Inheritance
      IInheritedTypeList
    Initializers
      ArrayInitializer
      IVariableInitializer
    Interface
      Methods
        InterfaceMethodDeclaration
      Properties
        InterfacePropertyDeclaration
      InterfaceDeclaration
      IInterfaceMemberDeclaration
    Literals
      Concepts
        BoolLiteral
        IntLiteral
        Literal
        NullLiteral
        RealLiteral
        StringLiteral
        CharLiteral
      Definitions
        _IntValue
        _StringValue
        _RealValue
        _CharValue
        _BoolValue
    Modifiers
      Sets
        IClassModifier
        IVisibility
        IStaticAbstractSealed
        IEnumModifier
        IFieldModifier
        IReadonlyVolatile
        IConstructorModifier
        IConstantModifier
        IInterfaceModifier
        IMethodModifier
        IAccessorVisibility
        IStructModifier
        IDelegateModifier
        IPropertyModifier
        IModifier
        IInterfacePropertyModifier
      Visibility
        Public
        Private
        Internal
        Protected
      Partial
      New
      Static
      Abstract
      Sealed
      Volatile
      Readonly
      Extern
      Async
      Virtual
      Override
      IHaveModifiers
    Namespace
      ExternAliasDirective
      UsingDirectiveList
      ExternAliasDirectiveList
      INamespaceMemberDeclaration
      NamespaceDeclaration
      TypeDeclaration
      UsingDirective
    References
      MemberReferences
        IReferencableMemberDeclaration
        MemberReference
        AmbiguousMemberReference
      TypeRelatedReferences
        GenericTypeParameterReference
        IReferencableTypeDeclaration
        GenericTypeParameterReferenceString
        NotGenericParameterTypeReference
        TypeReference
      VariableReferences
        VariableReference
        IReferencableVariableDeclaration
    Statements
      Blocks
        Block
        MaybeEmptyBlock
        EmptyBlock
      Declaration
        DeclarationStatement
        LocalVariableDeclarationStatement
        LocalVariableDeclarationWithInitialization
        LocalConstantDeclaration
      If
        IfStatement
        ElsePart
      Iteration
        IterationStatement
        WhileStatement
        DoStatement
        ForStatement
        IForInitializer
        ForeachStatement
      Jumps
        JumpStatement
        BreakStatement
        ContinueStatement
        GotoStatement
        GotoIdentifierStatement
        GotoCaseStatement
        GotoDefaultStatement
        ReturnStatement
        ThrowStatement
      Other
        Statement
        LabeledStatement
        EmbeddedStatement
        ExpressionStatement
        IStatementExpression
        SelectionStatement
        StatementExpressionList
      Switch
        SwitchStatement
        SwitchSection
        SwitchLabel
        CaseLabel
        DefaultLabel
      Try
        TryStatement
        TryCatchFinallyStatement
        CatchClause
        OptionalGeneralCatch
        SpecificCatchClause
        BuiltInClassTypeCatch
        QualifiedIDCatch
        GeneralCatchClause
        MandatoryGeneralCatch
        FinallyClause
        TryFinallyStatement
    Types
    UtilityConstants
      CommaConstant
      OutConstant
      RefConstant
      ConstantUtility
      RefOutConstant
    File
    IInheritedTypeListAndGenericTypeList
```

Figure 3.2: Overview of the Structure aspect

3.3.3.3 Editor

The most important principle in our Editor aspect is having a defined editor for each non-abstract concept. Although this is not a world-breaking principle, it is really important as it is crucial for the coding experience. Even though the MPS tool automatically generates default editors for all concepts, these are very far from what C# code looks like.

One of the most difficult things in our project was trying to get as close coding experience to the traditional C# as possible. Unfortunately, this task is very complicated as projectional editing is very far from traditional text editing. To achieve fluent editing of code, one must use different combinations of features of the Editor aspect. The most important features we used are:

- ***Transformation menus***. These provide the language designer with a way how to allow smart transformations of the AST node or its neighborhood when a user manipulates with an AST node by making simple actions, most commonly by typing some character or a text.

  We used transformation menus mostly for cases when the user adds some text from the left or from the right side to some entity. E.g. when the user adds a colon to a class name, the inheritance list is created (Figure 3.3).

  Transformation menus are primarily used in the high-level part of the language, i.e. for entities such as classes, methods etc.

- ***Substitution menus***. They are quite similar to transformation menus but they are more powerful. As can be expected, it is easier to use transformation menus if is is possible.

  Substitution menus are used primarily at the part of our base language that is related to statements or expressions.

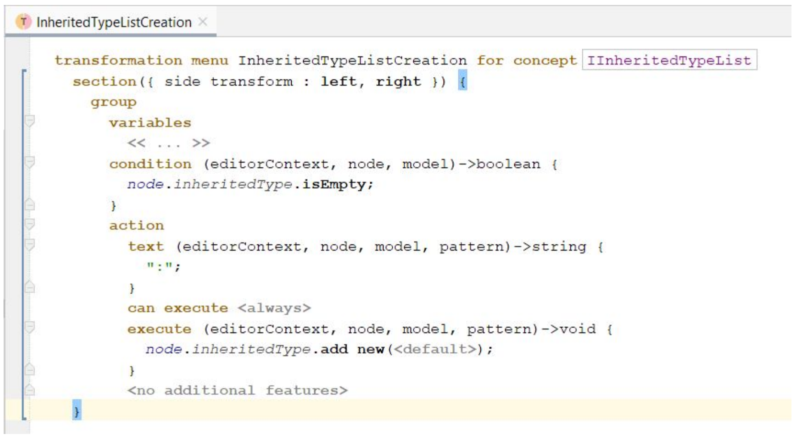
  An example from our base language can be seen in Figure 3.4.

- ***Key maps***. This feature allows the language designer to register some actions under different keyboard shortcuts. Individually for particular concept, not for the whole MPS environment.

  We do not use key maps often in the language. Most of the time, transformation menus with one-character long pattern are more appropriate than a keymap with that character as the trigger.

- ***Action maps***. These are quite similar to key maps, but the trigger is not a key shortcut but an action such as auto-completion of an AST node, deletion of a node etc. As these actions are usually performable via keyboard shortcuts, a lot of things can be done using action maps as well as key maps.

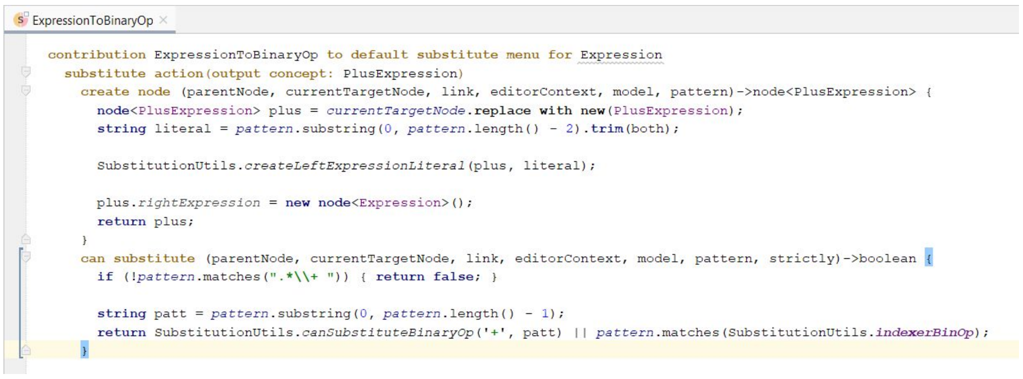
  We do not use action maps much in our base language.



InheritedTypeListCreation

```
transformation menu InheritedTypeListCreation for concept IInheritedTypeList
  section({ side transform : left, right }) {
    group
      variables
        << ... >>
      condition (editorContext, node, model)->boolean {
        node.inheritedType.isEmpty;
      }
      action
        text (editorContext, node, model, pattern)->string {
          ":";
        }
        can execute <always>
        execute (editorContext, node, model, pattern)->void {
          node.inheritedType.add new(<default>);
        }
        <no additional features>
  }
```

Figure 3.3: Transformation menu for addition of list of inherited types (to a class)



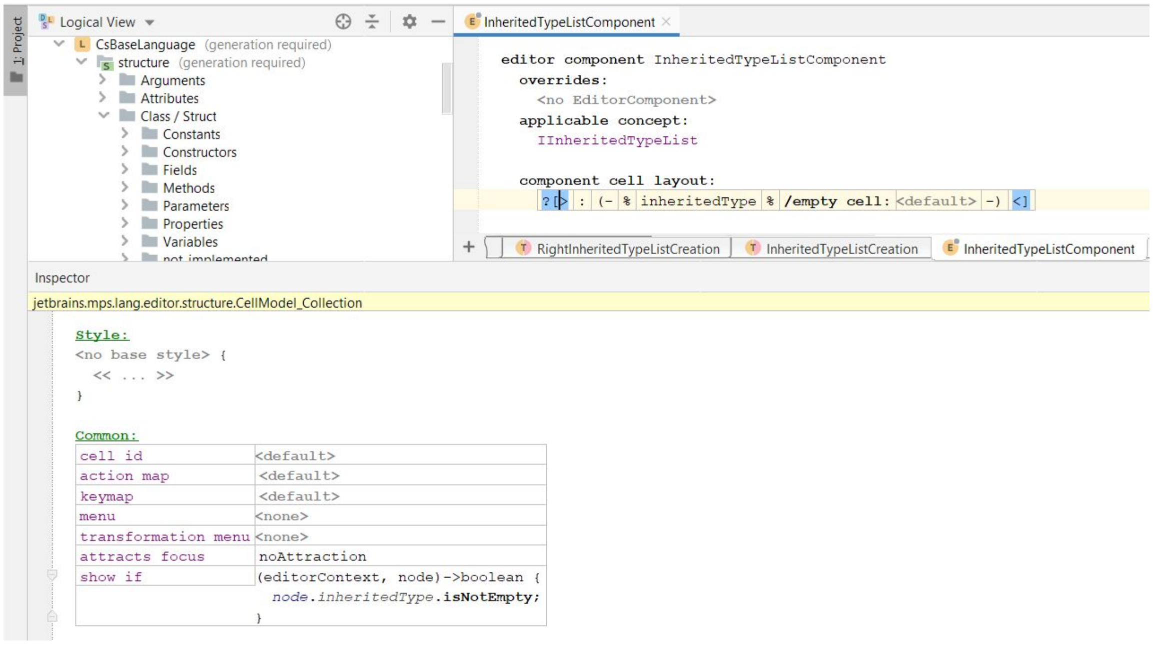
ExpressionToBinaryOp

```
contribution ExpressionToBinaryOp to default substitute menu for Expression
  substitute action(output concept: PlusExpression)
    create node (parentNode, currentTargetNode, link, editorContext, model, pattern)->node<PlusExpression> {
      node<PlusExpression> plus = currentTargetNode.replace with new(PlusExpression);
      string literal = pattern.substring(0, pattern.length() - 2).trim(both);

      SubstitutionUtils.createLeftExpressionLiteral(plus, literal);

      plus.rightExpression = new node<Expression>();
      return plus;
    }
    can substitute (parentNode, currentTargetNode, link, editorContext, model, pattern, strictly)->boolean {
      if (!pattern.matches(".*\\+ ")) { return false; }

      string patt = pattern.substring(0, pattern.length() - 1);
      return SubstitutionUtils.canSubstituteBinaryOp('+', patt) || pattern.matches(SubstitutionUtils.indexerBinOp);
    }
```

Figure 3.4: Substitution menu for creation of an addition AST node

Furthermore, to display the AST similarly to what C# code looks like, one must make use of:

- Empty-cell editors, i.e. features that allow displaying special output for empty or, in other words, missing AST nodes.
- Features that allow displaying an AST node only under specific circumstances. This feature can be found in the Inspector for a cell of an editor under keyword *show-if* (Figure 3.5). We use this very often.
- Styling functionality of editor cells that allows the language designer to specify where spaces between displayed AST nodes should be omitted or what AST nodes should not be selectable by a cursor or editable. This is necessary to use to display the code in a way expected by the user from the traditional C#.


Logical View
1: Project
CsBaseLanguage (generation required)
structure (generation required)
Arguments
Attributes
Class / Struct
Constants
Constructors
Fields
Methods
Parameters
Properties
Variables
not implemented
InheritedTypeListComponent
RightInheritedTypeListCreation
InheritedTypeListCreation
InheritedTypeListComponent
Inspector

```
editor component InheritedTypeListComponent
  overrides:
    <no EditorComponent>
  applicable concept:
    IInheritedTypeList

  component cell layout:
    ?[> : (- % inheritedType % /empty cell: <default> -) <]
```

jetbrains.mps.lang.editor.structure.CellModel_Collection

```
Style:
<no base style> {
  << ... >>
}

Common:
```

cell id	<default>
action map	<default>
keymap	<default>
menu	<none>
transformation menu	<none>
attracts focus	noAttraction

show if	(editorContext, node)->boolean {
	  node.inheritedType.isNotEmpty;
	}

Figure 3.5: The show-if feature illustration

Then, to encapsulate some Editor aspect parts and not to duplicate them, which could result in a hard maintainable base language, we make use of *Editor Components*, i.e. reusable parts of editors.

The next section discusses a specific topic related to the Editor aspect: *fluent editing* of code, i.e. editing of code in a projectional editor that reminds editing in a text editor.

3.3.3.4 Fluent Editing

The nature of MPS is projectional editing. Even though it has several advantages, which were discussed in Section *1.2 JetBrains MPS*, a user used to IDEs with textual editing of code expects some specific kind of behavior of code editor and that is quite different what MPS projectional editor behaves like in its basic form. In order to make these users comfortable with code writing in MPS, the language designer must define non-trivial parts of the Editor aspect to minimize the gap between projectional editing and text editing. Because every language is different, MPS cannot do this work automatically.

Projectional editing in its basic form requires the user to edit the code in a way that reflects what is happening under the covers in the AST. E.g. in order to write an assignment statement such as `i = 1 + 2`, the user needs to

1. First, create an assignment AST node.
2. Then fill in the left side of the assignment.
3. Go to the right side of the assignment.
4. There, create an addition AST node.
5. Fill in the left side of the addition.
6. Go to the right side of the addition.
7. Fill in the right side of the addition as well.

This approach to writing of code is quite tedious and not very user-friendly, especially when considering that most of the code consists of expressions which contain another expressions as their part.

In order to make editing of the code simpler, MPS provides the language designer with some ways how to make code editing more text-like (they were introduced in the previous section). Even though MPS provides some tools for that, it is a very hard part of language design and the ideal fluent editing is practically impossible to achieve because the tools are very basic and as such they need to be used at all possible kinds of places in an AST where the language designer wants to achieve fluent editing. Furthermore, at these places, the tools need to be appropriately selected and then adapted to the exact context of the AST part, individually.

For purposes of fluent editing in our language, we decided to use mostly MPS features called transformation menus and substitution menus, which were introduced in the previous section. Fluent editing of the top-level concepts like namespaces, classes or methods is implemented using mostly *side transformation menus*. For statements and expressions, we decided to use substitution menus as they were the most straight-forward option.

Because of the character of the language designer's work, i.e. that all kinds of places in an AST needs to be handled individually, we did not achieve the state of ideal fluent editing. This would require a lot of more work, which would be out of the time constraints on the project.

3.3.4 Tests

As tests are important for any software project for many reasons, we also implemented tests for our C# base language.

3.3.4.1 MPS Tests

In MPS, the language designer can implement tests targeting different language aspects. We decided to implement tests targeting primarily the Editor aspect as this should cover the user's work complexly (typing, using different kinds of actions, intentions and so on).

An Editor test consists of three parts described in the following list. An example can be seen in Figure 3.6.

- Pre-conditions, i.e. the state of the AST tree and the user's cursor position before the test
- A sequence of actions of the user executed in the test, such as typing, pressing keys, using Intention actions, invocating auto-completion, etc.
- Post-conditions, i.e. the state of the AST tree and optionally the user's cursor position after the test

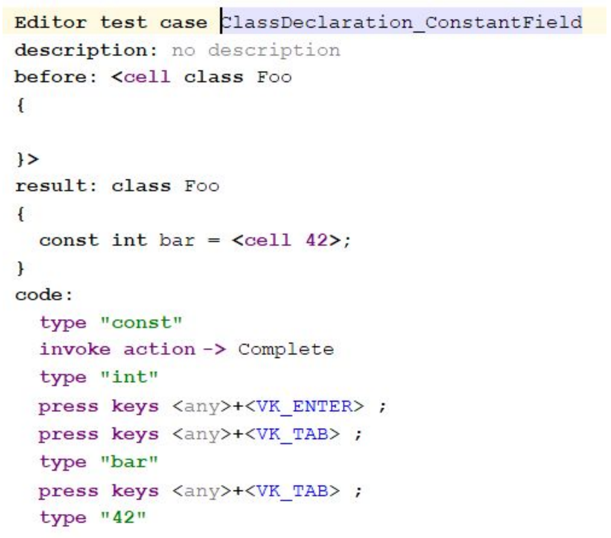

```
Editor test case ClassDeclaration_ConstantField
description: no description
before: <cell class Foo
{

}>
result: class Foo
{
  const int bar = <cell 42>;
}
code:
  type "const"
  invoke action -> Complete
  type "int"
  press keys <any>+<VK_ENTER> ;
  press keys <any>+<VK_TAB> ;
  type "bar"
  press keys <any>+<VK_TAB> ;
  type "42"
```

Figure 3.6: An example of a test

In Figure 3.6, we can see that the pre-conditions are located under the *before* section. There we assume that the user starts with an empty class Foo and a cursor located in it (the cursor location cannot be seen in Figure 3.6). The *code* section lists the user's actions in the order in which they are executed in the test. As can be seen in the figure, the user creates a constant with an initialization using the illustrated set of steps. The post-conditions, i.e. the code state at the end, are visible under the *result* section.

If the post-conditions are not equal to the result of the user's actions applied to the pre-conditions, the test fails and reports a mismatch.

Even though we implemented dozens of tests, they do not cover the whole language. That would require almost the same amount of effort as to develop the language again. We primarily focused on the high-level parts of the language.

3.3.4.2 Execution of MPS Tests

The MPS tests can be run very easily. Right-click the *CsBaseLanguage.tests* solution and trigger *Run Tests in CsBaseLanguage.tests*.
An illustration is provided by Figure 3.7.

3.3.4.3 Other tests

We also tested the language by implementing example programs based on our base language. These tests covered the whole process of the program development, i.e. writing the code utilizing the language aspects in combination, using the standard libraries and generating and running the C# source code.

More about this topic can be found in *Chapter 5*.

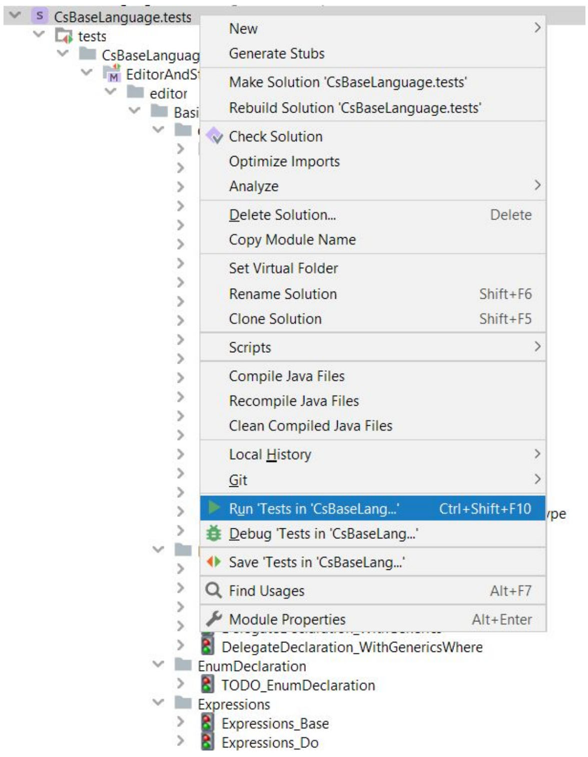


Figure 3.7: Running tests

3.3.5 Distribution

The C# base language is distributed as an MPS plugin, which can be installed via Jetbrains MPS Marketplace. This is the most optimal way of how to distribute an MPS language or any other utility. It is very easy for a user to install it and use it, with just a few clicks.

Detailed guides about how to use our language are located in *Chapter 4*.

3.4 TextGen Generation

As mentioned before in Section *3.2*, we decided to generate the TextGen aspect from the Editor aspect as, in the case of a base language, that is an appropriate way of creating the TextGen aspect.

When we started our project, there was an existing plugin called *TextGenGen* for this purpose. However, its functionality was restricted only to really basic Editor features and as such it was practically unusable for such a complex Editors as that of our base language. Therefore, we evaluated that the best possible solution is to improve that plugin by extending its functionality so that it can be used even for complex Editor aspects.

When improving the plugin, we discovered that the design was quite prepared for functionality extensions, although not very well documented. Later on, we improved the design even further, significantly refactored the code and documented it so we believe that now the plugin is in a good shape and more extendable.

Here is a list of features that we added to the plugin’s functionality:

- **Support of the *show-if* Editor feature.**

  This feature can be found in the Inspector window for an editor cell. An editor cell represents a unit of code, e.g. a keyword or an AST node property). An Inspector window for an editor cell using the *show-if* feature is illustrated in Figure 3.5 in Section *3.3.3.3* (see especially the bottom of the figure).

  The *show-if* feature allows showing the editor cell if some conditions are satisfied. These conditions are usually evaluated at run-time when the user writes the code. This feature is really a core feature and is used very often. For example, we use it for hiding inheritance lists if there are not inherited entities, hiding modifiers if they are not used, etc.

- **Support of the *empty-cell* Editor feature.**

  This feature is used when the language designer wants to define what should be displayed instead of a child AST node (or a list of child AST nodes) if that node (or that list) is empty.

- **Support of Editor components.**

  Editor components encapsulate Editor parts so that these parts do not have to be duplicated.

  One case of an Editor component, which had to be handled in the plugin specially, is the *alias* component which serves as a reference to the Structure aspect, to the *alias* name of the concept. We often use this component for keywords.

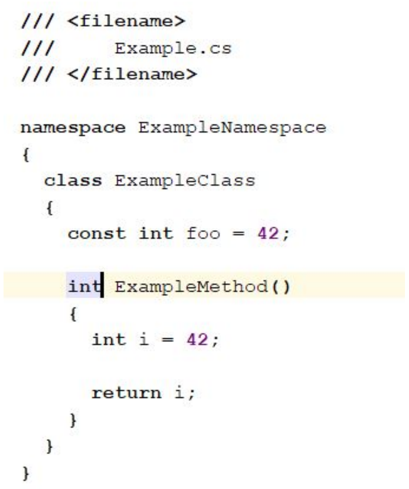
- **Significantly improved the appearance of the generated GPL source code.**

  The indentation and the spacing between the elements were at a basic level. We had to reimplement this functionality almost from scratch.

  Now, the generated GPL source code looks almost exactly the same as the AST displayed by the Editor. An example for our base language is illustrated in Figures 3.8 and 3.9. Figure 3.8 shows an AST displayed by the Editor aspect and Figure 3.9 shows the corresponding generated GPL source code.

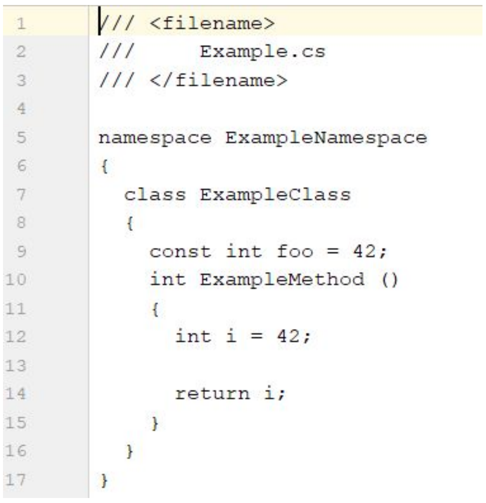
- **The usability of the plugin improved as well: a new TextGen generation now automatically removes the old TextGen.**

  The old version of the plugin did not manipulate with the old TextGen so there were collisions when a new TextGen was generated.



```
/// <filename>
///     Example.cs
/// </filename>

namespace ExampleNamespace
{
  class ExampleClass
  {
    const int foo = 42;

    int ExampleMethod()
    {
      int i = 42;

      return i;
    }
  }
}
```

Figure 3.8: Example program rendered by Editor



```
/// <filename>
///     Example.cs
/// </filename>

namespace ExampleNamespace
{
  class ExampleClass
  {
    const int foo = 42;
    int ExampleMethod ()
    {
      int i = 42;

      return i;
    }
  }
}
```

Figure 3.9: Example program generated by TextGen

We released our work as a new version of the plugin. The source code can be found on GitHub (see links in *Appendix B*) and the binary package is distributed via the JetBrains MPS Marketplace.

3.5 Stubs

In this section, we present the solution for generation of library stubs, which were introduced in Section 2.4.

The primary goal of this part of our project is to provide stubs for the C# standard library, which are practically necessary for any serious C# development[8]. Because this library changes over time and it is too large for manual creation, we needed to implement automated generation of stubs from given DLL files.

In the following subsections we describe the architectural design of the solution, its usage and deployment, involved data file formats, and we also discuss some problems that appeared during the development.

3.5.1 Design

While designing our solution, we identified that its non-trivial part is independent of the selected base language. Therefore we decided to provide an implementation that may be partly reused also for other base languages, current or future.

We noticed that a stubs generator for any base language can be decomposed into three parts:

1. A parser of binary libraries for which we intend to create stubs.

[8] The stubs generation solution can be used for other libraries as well but we focus on the standard library.

2. An orchestrator controlling the order of stubs generation and organization of stubs into MPS models.
3. Generator of concrete AST nodes for the selected base language based on requests of the orchestrator.

The second part of the decomposition is language-independent. It does not have to know neither what concepts the language contains nor the structure of binary libraries. The first and the third part can be understood as adapters of the orchestrator to the particular selected programming language.

With this idea in mind, we created a solution consisting of three modules, one for each of the listed parts above.

The first one is a library parser written in the C# general-purpose language. This is an appropriate solution as the binary libraries can be relatively easily introspected via C# Reflection. It is described in detail in Section *3.5.2*.

The second and third modules are formed as plugins to JetBrains MPS and are written in the Base Language. More about them can be found in Section *3.5.4*.

As for data flow, the first module creates an XML file describing the library structure. It does not contact the other modules in any way, the second module is given this file via an action of the user[9]. The second and third modules communicate via direct method calls inside MPS.

3.5.2 DLL Parser

The first part of the stubs generation process is handled by a parser of C# binary libraries. As we needed to primarily deliver stubs for the standard library, which is distributed as DLL files, and parsing of source code is hard, we chose to create a DLL parser, not a source code parser. This limits the usage of our solution only to libraries which are distributed as DLLs and it cannot be directly used for example for parsing C# source code[10].

For a given DLL, the DLL parser produces an XML document containing description of declarations of the library entities. In our case, it contains for example namespaces, classes, interfaces or methods. It does not contain any implementation of the methods as they are not needed for library stubs. The detailed description of the produced XML file is left for Section *3.5.3.*

The DLL parser is a console-based program utilizing the C# Reflection functionality to analyze the given DLLs.

[9] The main reason for this is that we are not able to contact MPS from outside. However, it is also appropriate for the users who may want to manipulate the stubs generation process as they may modify the XML file or create the XML file themselves.
[10] Although the solution can be extended so that parsing of raw C# source code is implementable.

We use the DLL parser to extract API of the C# standard library, restricted to a supported subset of features in the C# 5.0 specification so that the constructed library stubs are as compatible as possible with our C# base language. The API is based on .NET Framework, its reference source code, .NET Core and .NET Standard.

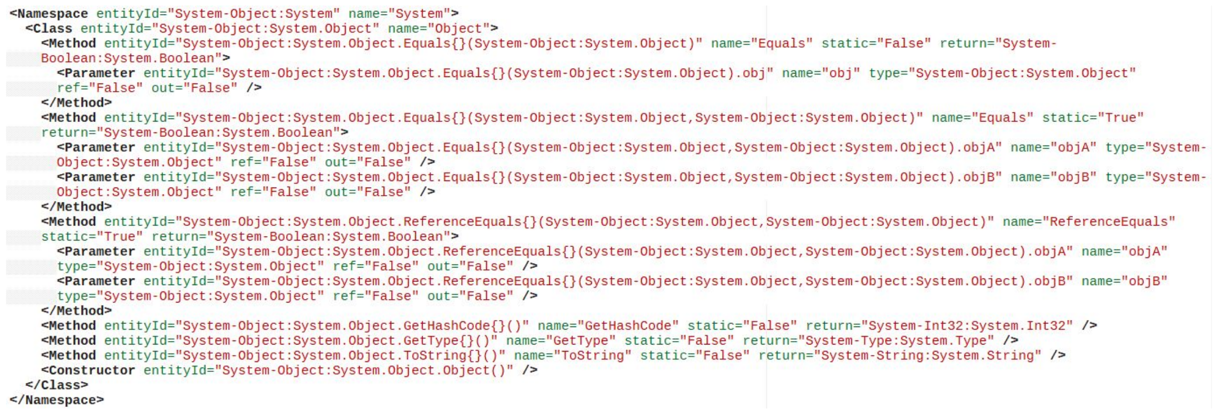
3.5.3 XML Document

The XML file created by the DLL parser describes the library in a hierarchical structure of individual library entity specifications. Each library entity results in an XML element which describes this entity via attributes and child XML elements (we call this description as *entity specification*). For example, a class entity results in an XML element of name *Class* and with attributes such as the name of the class or the inherited base class and with child XML elements such as the methods of that class.

Each XML element has a mandatory attribute called *entity ID* which uniquely identifies the XML element in the XML file. As mentioned before, an XML element represents a library entity (such as a class, method, parameter etc.) and therefore the entity ID is a unique identifier of the XML element as well as of the library entity.

Here is an illustration of a chunk of the XML file which contains a class *MyClass* inside a namespace *Example*. The class contains a field *Foo* and a one-parameter method *Bar*. Note that this example is really only an illustration. A real XML file is much more sophisticated. An example of a real XML file chunk can be seen in Figure 3.10.

```xml
<Namespace entityId="Example" name="Example">
  <Class entityId="Example.MyClass" name="MyClass" BaseClass="X.Y">
    <Field entityId="Example.MyClass.Foo" name="Foo" type="System.Object" />
    <Method entityId="Example.MyClass.Bar{}(System.Char)" name="Bar" return="void">
      <Parameter entityId="Example.MyClass.Bar{}(System.Char).c" name="c"
                 type="System.Char" />
    </Method>
  </Class>
</Namespace>
```



```xml
<Namespace entityId="System-Object:System" name="System">
  <Class entityId="System-Object:System.Object" name="Object">
    <Method entityId="System-Object:System.Object.Equals{}(System-Object:System.Object)" name="Equals" static="False" return="System-
    Boolean:System.Boolean">
      <Parameter entityId="System-Object:System.Object.Equals{}(System-Object:System.Object).obj" name="obj" type="System-Object:System.Object"
      ref="False" out="False" />
    </Method>
    <Method entityId="System-Object:System.Object.Equals{}(System-Object:System.Object,System-Object:System.Object)" name="Equals" static="True"
    return="System-Boolean:System.Boolean">
      <Parameter entityId="System-Object:System.Object.Equals{}(System-Object:System.Object,System-Object:System.Object).objA" name="objA" type="System-
      Object:System.Object" ref="False" out="False" />
      <Parameter entityId="System-Object:System.Object.Equals{}(System-Object:System.Object,System-Object:System.Object).objB" name="objB" type="System-
      Object:System.Object" ref="False" out="False" />
    </Method>
    <Method entityId="System-Object:System.Object.ReferenceEquals{}(System-Object:System.Object,System-Object:System.Object)" name="ReferenceEquals"
    static="True" return="System-Boolean:System.Boolean">
      <Parameter entityId="System-Object:System.Object.ReferenceEquals{}(System-Object:System.Object,System-Object:System.Object).objA" name="objA"
      type="System-Object:System.Object" ref="False" out="False" />
      <Parameter entityId="System-Object:System.Object.ReferenceEquals{}(System-Object:System.Object,System-Object:System.Object).objB" name="objB"
      type="System-Object:System.Object" ref="False" out="False" />
    </Method>
    <Method entityId="System-Object:System.Object.GetHashCode{}()" name="GetHashCode" static="False" return="System-Int32:System.Int32" />
    <Method entityId="System-Object:System.Object.GetType{}()" name="GetType" static="False" return="System-Type:System.Type" />
    <Method entityId="System-Object:System.Object.ToString{}()" name="ToString" static="False" return="System-String:System.String" />
    <Constructor entityId="System-Object:System.Object.Object()" />
  </Class>
</Namespace>
```

Figure 3.10: Example of a chunk of an XML stub specification file

The modules which handle the rest of stubs generation process, and which are described in the following Section *3.5.4*, have several set of requirements for this file.

The most important one is that each XML element (except the root XML element) has a unique entity ID, which has been already mentioned. This entity ID is used for cases when some XML element needs to reference some other XML element, for example in case of a return type of a method.

Another requirement is that entity IDs of child elements of another XML element must be prefixed by the entity ID of this parent XML element. This serves for purposes of an easy search of an XML element according to a given entity ID.

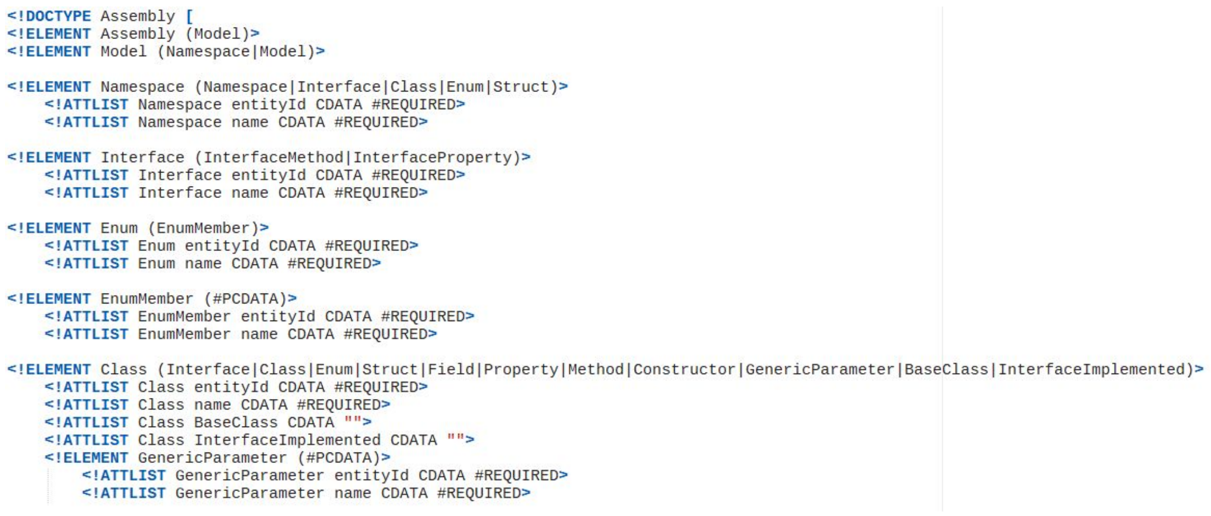
Last of the important requirements is that the XML must contain a DTD specification of the XML structure so that it can be easily parsed by the other modules of the stubs generation process (DTD example is illustrated by Figure 3.11).

As you can see, the XML file format is quite strict at some aspects and must be agreed by the DLL parser and the other modules from the next section.

As for the XML file that we used for the standard library stubs generation, the file size is about 18.5 MB and it contains about 95 thousand lines of text[11]. The contained XML elements are of several kinds:

- *Model*, which represents an MPS model. It wraps library entities which should be generated into the same MPS model,
- Namespace,
- Class,
- Struct,
- Interface,
- Delegate,
- Enum and EnumMember,
- Field,
- Property and InterfaceProperty, which are distinguished because the language structure aspect distinguishes them,
- Method and InterfaceMethod, which are distinguished from the same reasons as properties,
- Constructor.

[11] As for the execution time of our stubs generator solution for this standard library XML file, it takes about several seconds to generate the stubs.



```
<!DOCTYPE Assembly [
<!ELEMENT Assembly (Model)>
<!ELEMENT Model (Namespace|Model)>

<!ELEMENT Namespace (Namespace|Interface|Class|Enum|Struct)>
    <!ATTLIST Namespace entityId CDATA #REQUIRED>
    <!ATTLIST Namespace name CDATA #REQUIRED>

<!ELEMENT Interface (InterfaceMethod|InterfaceProperty)>
    <!ATTLIST Interface entityId CDATA #REQUIRED>
    <!ATTLIST Interface name CDATA #REQUIRED>

<!ELEMENT Enum (EnumMember)>
    <!ATTLIST Enum entityId CDATA #REQUIRED>
    <!ATTLIST Enum name CDATA #REQUIRED>

<!ELEMENT EnumMember (#PCDATA)>
    <!ATTLIST EnumMember entityId CDATA #REQUIRED>
    <!ATTLIST EnumMember name CDATA #REQUIRED>

<!ELEMENT Class (Interface|Class|Enum|Struct|Field|Property|Method|Constructor|GenericParameter|BaseClass|InterfaceImplemented)>
    <!ATTLIST Class entityId CDATA #REQUIRED>
    <!ATTLIST Class name CDATA #REQUIRED>
    <!ATTLIST Class BaseClass CDATA "">
    <!ATTLIST Class InterfaceImplemented CDATA "">
    <!ELEMENT GenericParameter (#PCDATA)>
        <!ATTLIST GenericParameter entityId CDATA #REQUIRED>
        <!ATTLIST GenericParameter name CDATA #REQUIRED>
```

Figure 3.11: DTD part example

3.5.4 Stubs Creation

The solution of stubs creation from the XML file consists of two JetBrains MPS plugins. The first plugin, called *StubsGenerator*, is language-independent and orchestrates the stubs generation process. It handles the order of the stubs generation and organizes the stubs into MPS models. The second plugin, called *CsStubsGenerator*, creates a concrete concept instances (AST nodes) for specified library entities.

StubsGenerator forms the core of the stubs generation and controls the overall process. It requests *CsStubsGenerator* only for generation of language-specific AST nodes or for execution of small language-specific tasks.

StubsGenerator parses a given XML file and creates a tree structure mirroring the XML tree in the file. Note that it is not an XML tree representation from an XML parsing framework (which is also used, but only for parsing the XML file) but a custom tree representation that consists of nodes called *MpsEntitySpec*. Each MpsEntitySpec represents a specification of one library entity and contains a map of property names to property values and list of child library entity specification nodes of the tree. Properties are typically the XML attributes. This custom structure eases manipulation with the library entity specifications and encapsulates the plugins.

After parsing of the XML file, *StubsGenerator* starts generation of stubs in a post-order depth-first-search way. For each node of the tree, i.e. for each library entity specification, it calls *CsStubsGenerator* to get a generated AST node representing the stub corresponding to that library entity specification. Sometimes it happens that another library stub must be generated before the library stub corresponding to the currently visited library stub specification node. In this case, *CsStubsGenerator* uses a call-back of the *StubsGenerator* and asks for out-of-order generation of the needed library stub.

After this process, *StubsGenerator* inserts the library stubs into specified MPS models and satisfies the dependencies of the models on each other and on the base language.

Although it might seem that *CsStubsGenerator* does all the work and *StubsGenerator* is quite simple and only calls *CsStubsGenerator*, it is not true. In fact, the work and functionality provided by *StubsGenerator* is very helpful when developing the other, language-specific plugin (*CsStubsGenerator* in our case) and significantly simplifies its code. This is the power of the selected design - it offers a really helpful plugin for development of other base languages.

3.5.5 Usage

In this section, we present a few how-to tutorials which should guide a user through expected use-cases of our library stubs generation solution.

3.5.5.1 Generation of library stubs

This tutorial will guide you through generation of library stubs from a DLL file to final library stubs in MPS. It is assumed that you have JetBrains MPS installed and the DLL file prepared.

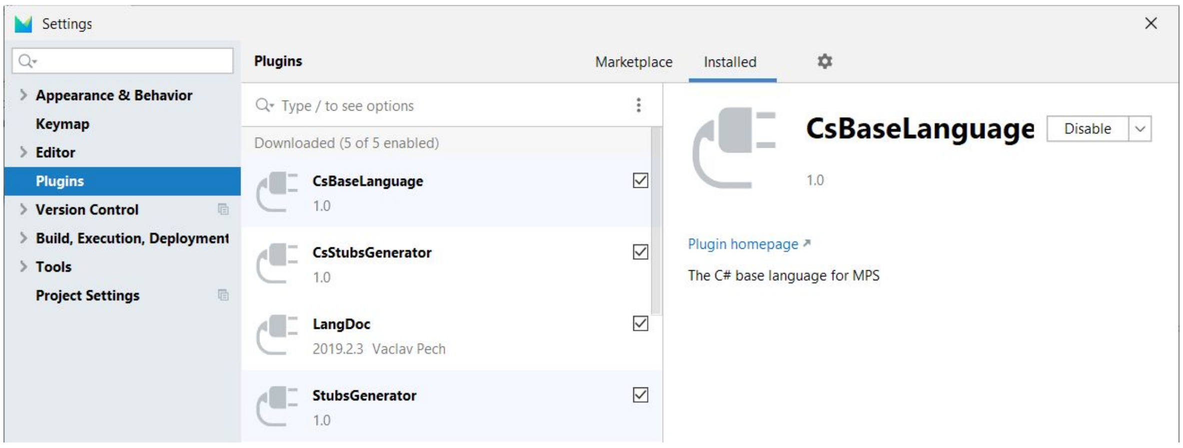
First, get our DLL parser from GitHub and compile it. You can use Visual Studio 2017, or newer, for this purpose. Then run the program from the command line like this:

```
DotNetFrameworkDllExporter.exe <dll-file-path> [-o <output-file-path>]
```

As a result of running this command, you should have an XML file, which describes the library stubs. With this done, you can continue to another phase of the process - using our plugins for stubs generation.

First, you should have these plugins installed. You can do it by navigating through *File* to *Settings* to *Plugins* to *Marketplace* where you can search for the plugin *StubsGenerator* and install it. Then, if you want for example to generate stubs for C# libraries, install also plugins *LangDoc*, *CsBaseLanguage* and *CsStubsGenerator*, in this order.

After MPS restart, you should end up with something similar to what is illustrated in Figure 3.12.



Downloaded (5 of 5 enabled)

CsBaseLanguage 1.0

CsStubsGenerator 1.0

LangDoc 2019.2.3 Vaclav Pech

StubsGenerator 1.0

CsBaseLanguage
1.0
Plugin homepage ↗
The C# base language for MPS

Figure 3.12: Installed plugins for stubs generation

When the previous is ready, create a new solution-like project (via *File*, *New, Project, Solution project*).

When right-clicking the created solution in the created project, you should see a pop-up menu shown in Figure 3.13. Selecting the *Generate Stubs* option should result in a file-choosing dialog which you utilize for selection of the XML file you have generated by the DLL parser.

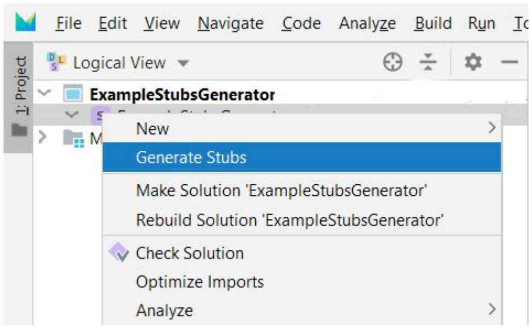

File Edit View Navigate Code Analyze Build Run
1: Project
Logical View
ExampleStubsGenerator
New
Generate Stubs
Make Solution 'ExampleStubsGenerator'
Rebuild Solution 'ExampleStubsGenerator'
Check Solution
Optimize Imports
Analyze

Figure 3.13: Pop-up menu shown during stubs generation

When the XML file is selected, automatic generation starts and on its end, you should see the generated stubs under the selected solution. Note that this process might take a while when the library is large (e.g. the stubs generation for the C# standard library lasts several seconds).

Now you can rebuild the solution in order to be able to use the stubs in your models. Note that, when using our C# base language and creating a model in which you intend to use the generated stubs, you should set up a dependency on the *CsBaseLanguage* (see Figures 3.14 and 3.15) and on the models containing the generated stubs you desire to use. You cannot set up a dependency on the whole solution with the stubs, you must select individual models (see Figures 3.16 and 3.17 where we set up a dependency on the *System* model of the C# standard library, which is distributed as the *CsStdLibrary* plugin).

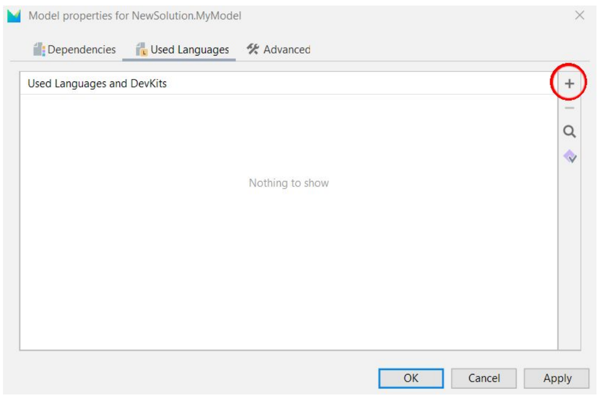

Model properties for NewSolution.MyModel
Dependencies
Used Languages
Advanced
Used Languages and DevKits
Nothing to show
OK
Cancel
Apply

Figure 3.14: Adding a model dependency

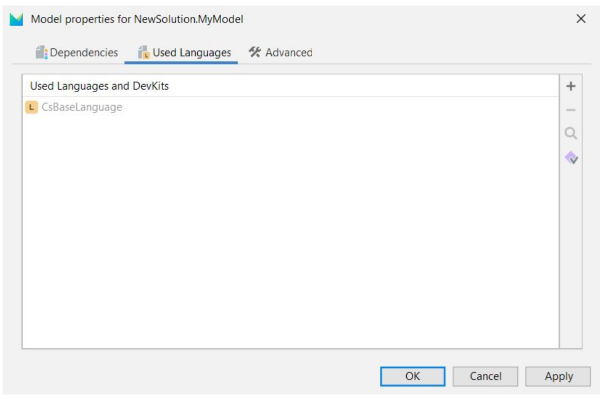
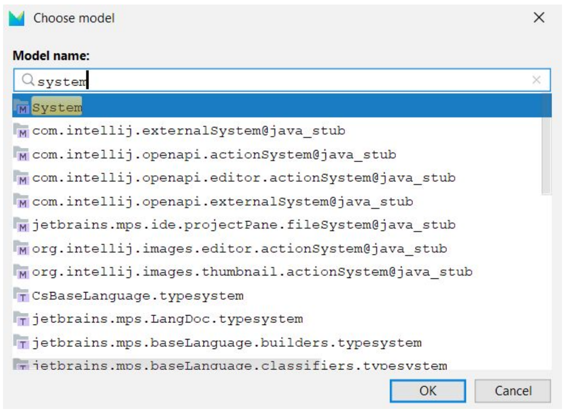

Model properties for NewSolution.MyModel
Dependencies
Used Languages
Advanced
Used Languages and DevKits
CsBaseLanguage
OK
Cancel
Apply

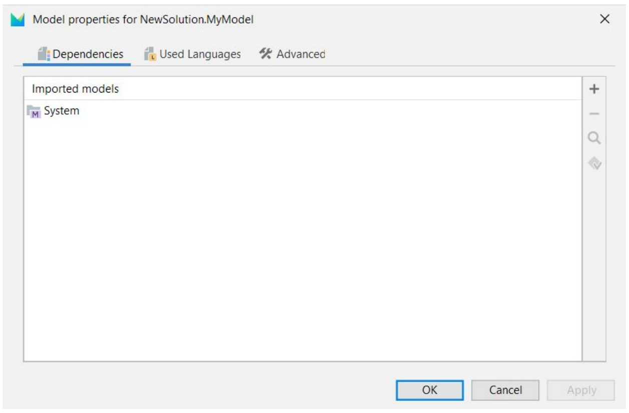
Figure 3.15: Model dependency on the C# base language



system

System
com.intellij.externalSystem@java_stub
com.intellij.openapi.actionSystem@java_stub
com.intellij.openapi.editor.actionSystem@java_stub
com.intellij.openapi.externalSystem@java_stub
jetbrains.mps.ide.projectPane.fileSystem@java_stub
org.intellij.images.editor.actionSystem@java_stub
org.intellij.images.thumbnail.actionSystem@java_stub
CsBaseLanguage.typesystem
jetbrains.mps.LangDoc.typesystem
jetbrains.mps.baseLanguage.builders.typesystem
jetbrains.mps.baseLanguage.classifiers.typesystem

Figure 3.16: Searching for the System model of the C# standard library



Imported models
System

Figure 3.17: Model dependency on the System model of the C# standard library

3.5.5.2 Creating a Custom Stubs Generator

If you desire to create either your own stubs generator for the C# base language, you want to modify *CsStubsGenerator* or you plan to utilize *StubsGenerator* to develop a stubs generator for your base language, you can make use of this guide, where we describe an implementation of a stubs generator based on the *StubsGenerator* plugin from scratch.

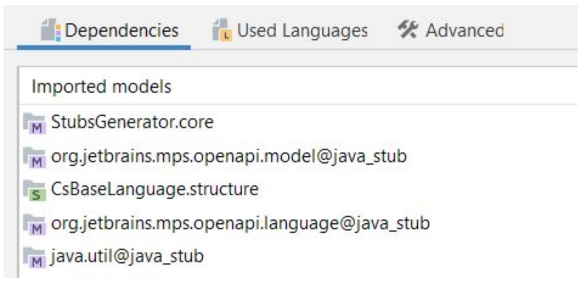
Since you plan to do a quite advanced thing, we assume that we do not need to explain the process into too much detail and we therefore we keep the guide rather brief and fast.

First, you should make sure that you have the *StubsGenerator* plugin installed. You can use JetBrains MPS Marketplace for that.

Then, create a solution project. It will contain your plugin that will be called by *StubsGenerator* in order to create AST nodes for concepts from your base language.

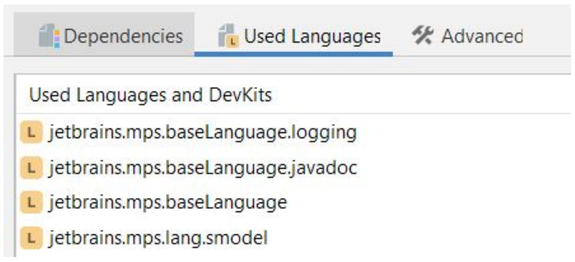
Create a plugin solution (right-click the project, select *New* and then *Plugin Solution*) and then create a model *core* inside.

Set the dependencies of the *core* model to contain the *StubsGenerator.core* model, the Structure model of your base language and the Base Language. In Figures 3.18 and 3.19, you can see a full example of dependencies of *CsStubsGenerator*.



Imported models
StubsGenerator.core
org.jetbrains.mps.openapi.model@java_stub
CsBaseLanguage.structure
org.jetbrains.mps.openapi.language@java_stub
java.util@java_stub

Figure 3.18: Model dependencies of the model CsStubsGenerator.core

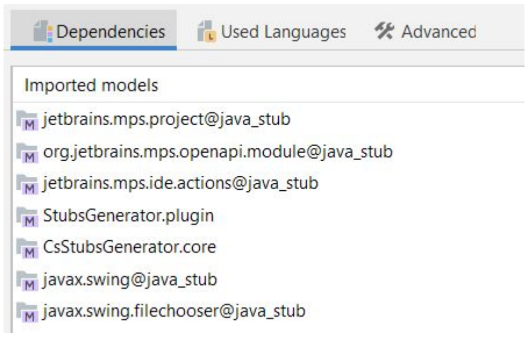

Dependencies | Used Languages | Advanced

Used Languages and DevKits

- jetbrains.mps.baseLanguage.logging
- jetbrains.mps.baseLanguage.javadoc
- jetbrains.mps.baseLanguage
- jetbrains.mps.lang.smodel

Figure 3.19: Language dependencies of the model CsStubsGenerator.core

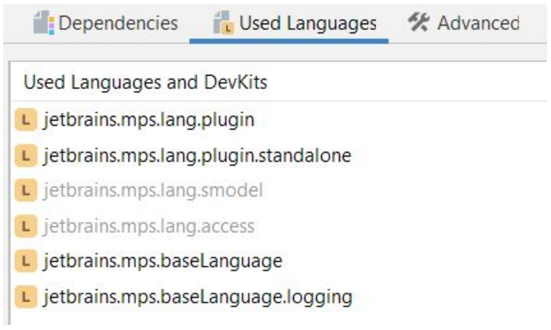
The dependencies of the *plugin* model should primarily contain a dependencies on your plugin's *core* model, on the *plugin* model of *StubsGenerator* and on the Base Language. Again, we present a full example of dependencies of *CsStubsGenerator* in Figures 3.20 and 3.21.


Dependencies | Used Languages | Advanced

Imported models

- jetbrains.mps.project@java_stub
- org.jetbrains.mps.openapi.module@java_stub
- jetbrains.mps.ide.actions@java_stub
- StubsGenerator.plugin
- CsStubsGenerator.core
- javax.swing@java_stub
- javax.swing.filechooser@java_stub

Figure 3.20: Model dependencies of the model CsStubsGenerator.plugin

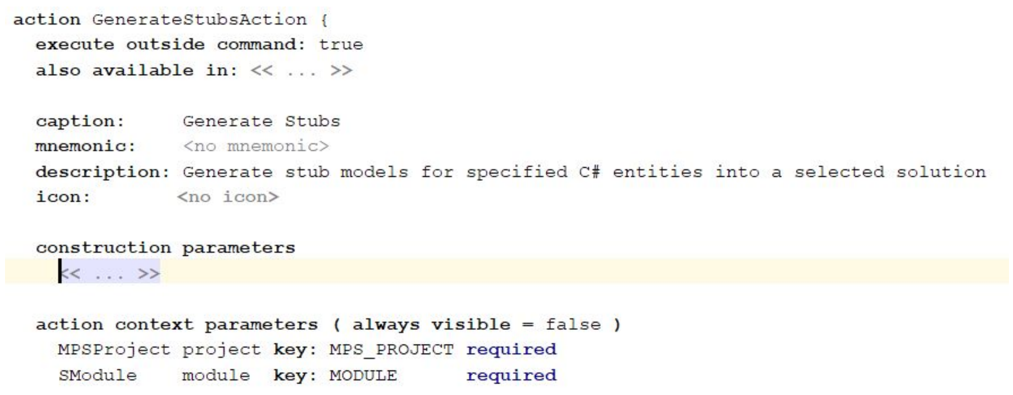

Dependencies | Used Languages | Advanced

Used Languages and DevKits

- jetbrains.mps.lang.plugin
- jetbrains.mps.lang.plugin.standalone
- jetbrains.mps.lang.smodel
- jetbrains.mps.lang.access
- jetbrains.mps.baseLanguage
- jetbrains.mps.baseLanguage.logging

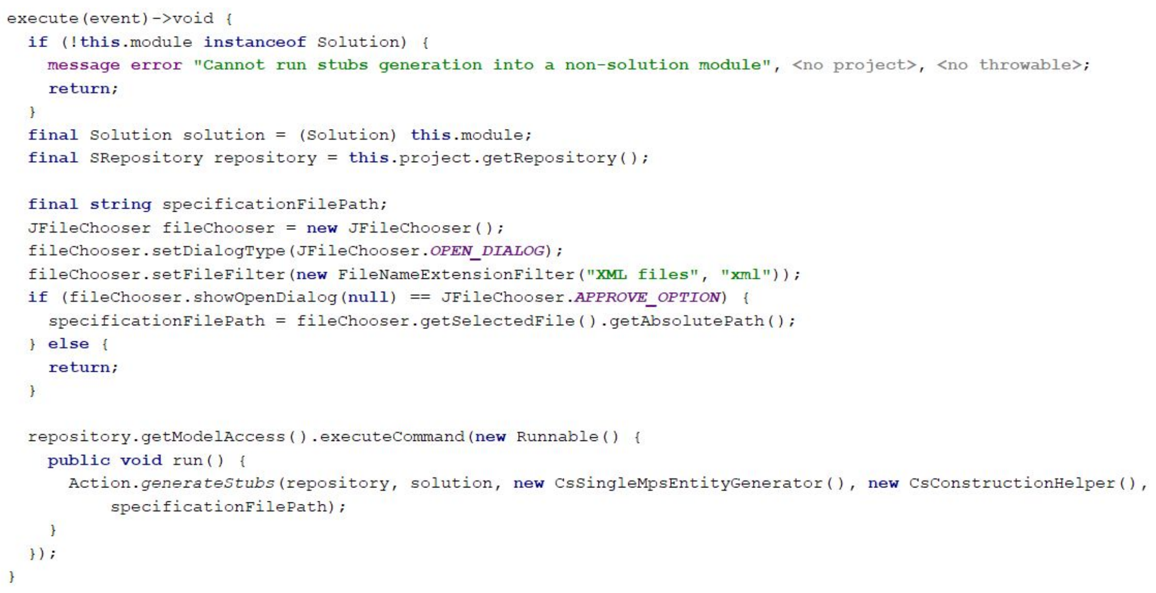
Figure 3.21: Language dependencies of the model CsStubsGenerator.plugin

Now you can create an action that will be displayed after right-clicking a solution module and that will offer your plugin's generation functionality. Right-click the *plugin* model, select *New*, *j.m.lang.plugin*, *Action*. In Figure 3.22, there is a header part of this action. Notice particularly the flag *execute outside command* set to true. Figure 3.23 then illustrates a possible implementation of this action. The most important part is at the end where we call the *StubsGenerator's* prepared static method which handles calling of *StubsGenerator* properly.



```
action GenerateStubsAction {
  execute outside command: true
  also available in: << ... >>

  caption:     Generate Stubs
  mnemonic:    <no mnemonic>
  description: Generate stub models for specified C# entities into a selected solution
  icon:        <no icon>

  construction parameters
    << ... >>

  action context parameters ( always visible = false )
    MPSProject project key: MPS_PROJECT required
    SModule    module  key: MODULE      required
```

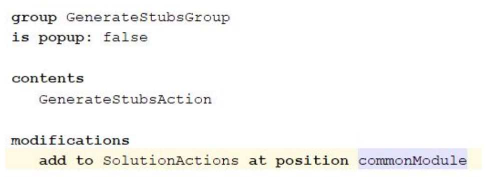
Figure 3.22: Stubs generation action header



```
execute(event)->void {
  if (!this.module instanceof Solution) {
    message error "Cannot run stubs generation into a non-solution module", <no project>, <no throwable>;
    return;
  }
  final Solution solution = (Solution) this.module;
  final SRepository repository = this.project.getRepository();

  final string specificationFilePath;
  JFileChooser fileChooser = new JFileChooser();
  fileChooser.setDialogType(JFileChooser.OPEN_DIALOG);
  fileChooser.setFileFilter(new FileNameExtensionFilter("XML files", "xml"));
  if (fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
    specificationFilePath = fileChooser.getSelectedFile().getAbsolutePath();
  } else {
    return;
  }

  repository.getModelAccess().executeCommand(new Runnable() {
    public void run() {
      Action.generateStubs(repository, solution, new CsSingleMpsEntityGenerator(), new CsConstructionHelper(),
          specificationFilePath);
    }
  });
}
```

Figure 3.23: Stubs generation action implementation

Now you can create an action group via right-clicking the *plugin* model and selecting *New*, *j.m.lang.plugin*, *Group*. It should have a content similar to what you can see in Figure 3.24.

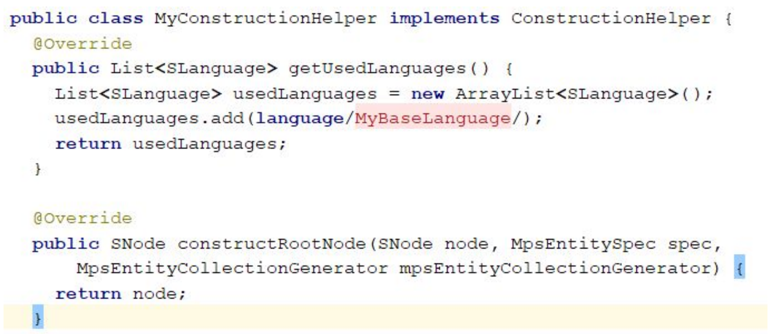


```
group GenerateStubsGroup
is popup: false

contents
   GenerateStubsAction

modifications
   add to SolutionActions at position commonModule
```

Figure 3.24: Stubs generation action group

Here comes finally the interesting part. You will create two classes that get called by *StubsGenerator* in order to obtain AST nodes of concepts from your selected base language. These classes might be very simple but they tend to get complex if your base language is complex. For example, in case of the C# base language the code gets to high hundreds of lines. But, in this tutorial, we can keep it simple because the idea is, fortunately, visible even then.

First, let’s create a class called construction helper. This class provides the stubs generation with meta-information or secondary routines. Create it and implement the *ConstructionHelper* interface from *StubsGenerator*. You should override the method *getUsedLanguages*, which returns language dependencies that should be set to the models containing the generated stubs, and the method *constructRootNode*, whose task is to wrap a given AST node into an AST node whose concept has the *instance-can-be-root* flag set to true. Examples can be found in Figure 3.25. The *constructRootNode* method may only return the given AST node without any change - it is up to design of your plugin, the design of your base language and the design of the XML file. For example, we, in our *CsStubsGenerator*, used this method.

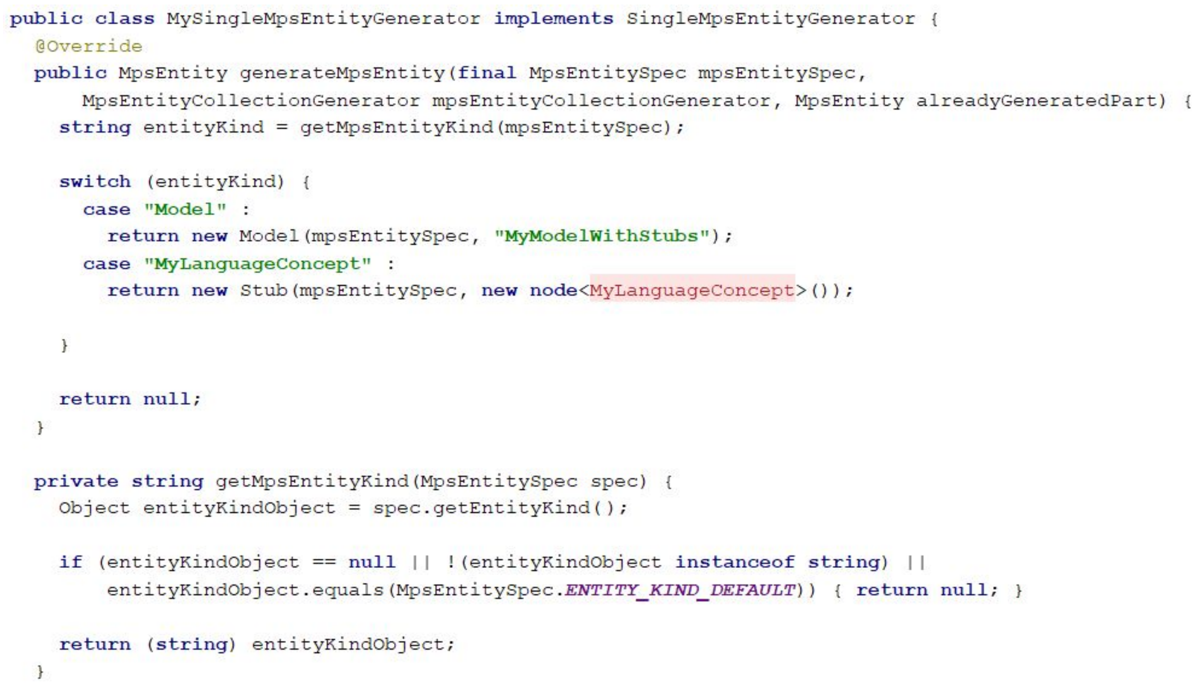


```
public class MyConstructionHelper implements ConstructionHelper {
  @Override
  public List<SLanguage> getUsedLanguages() {
    List<SLanguage> usedLanguages = new ArrayList<SLanguage>();
    usedLanguages.add(language/MyBaseLanguage/);
    return usedLanguages;
  }

  @Override
  public SNode constructRootNode(SNode node, MpsEntitySpec spec,
      MpsEntityCollectionGenerator mpsEntityCollectionGenerator) {
    return node;
  }
```

Figure 3.25: Example of a construction helper

Approaching the end, we need to create the second class, called a *single MPS entity generator*, according to the interface *SingleMpsEntityGenerator* which it implements. This class gets called from *StubsGenerator* when it is needed to generate a stub based on the specification coming from the XML file. The basic use case of this class, encapsulated in the *generateMpsEntity* method, is that you get an object *MpsEntitySpec* that represents an XML element, contains a set of properties (mostly the XML attributes) and references to *MpsEntitySpec* objects corresponding to the child XML elements. Based on this specification you should generate an AST node representing the specified stub. In Figure 3.26, a really simple implementation of this class’s *generateMpsEntity* method is presented.

As you can see, the *generateMpsEntity* method decides according to a mandatory property *entityKind* of the *MpsEntitySpec* object what stub should be generated. This property is the name of the corresponding XML element. The method assumes that there are two kinds of XML elements in the XML file. First, *Model*, results in a generated MPS model containing the stubs corresponding to inner XML elements of this *Model* element (this is handled automatically by StubsGenerator). And second, *MyLanguageConcept*, that results in an AST node of concept *MyLanguageConcept*, which has no special properties, children or references (otherwise we would have to set them manually in this method to the AST node).

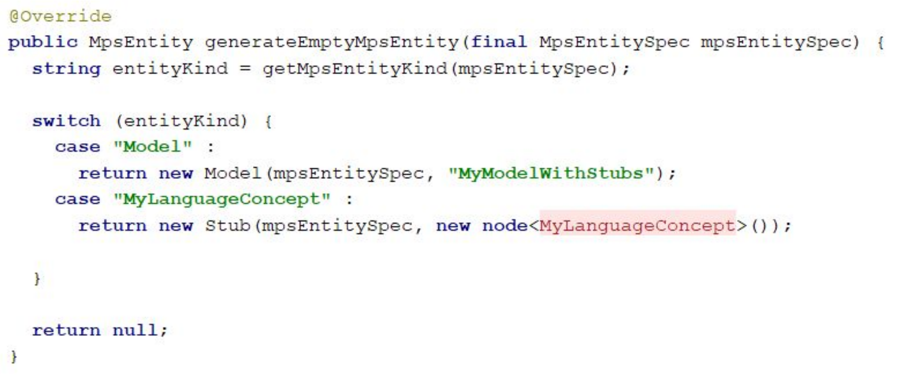


```
public class MySingleMpsEntityGenerator implements SingleMpsEntityGenerator {
  @Override
  public MpsEntity generateMpsEntity(final MpsEntitySpec mpsEntitySpec,
      MpsEntityCollectionGenerator mpsEntityCollectionGenerator, MpsEntity alreadyGeneratedPart) {
    string entityKind = getMpsEntityKind(mpsEntitySpec);

    switch (entityKind) {
      case "Model" :
        return new Model(mpsEntitySpec, "MyModelWithStubs");
      case "MyLanguageConcept" :
        return new Stub(mpsEntitySpec, new node<MyLanguageConcept>());

    }

    return null;
  }

  private string getMpsEntityKind(MpsEntitySpec spec) {
    Object entityKindObject = spec.getEntityKind();

    if (entityKindObject == null || !(entityKindObject instanceof string) ||
        entityKindObject.equals(MpsEntitySpec.ENTITY_KIND_DEFAULT)) { return null; }

    return (string) entityKindObject;
  }
```

Figure 3.26: Example of the generateMpsEntity method

If you noticed the *mpsEntityCollectionGenerator* parameter, it is a way how to call-back *StubsGenerator* to give you a reference to another stub (possibly generating it out of order immediately, when it has not been generated yet). It is useful when you, for example, need to reference an inherited class or a return type of a method.

And the last parameter, *alreadyGeneratedPart*, is bound to another method of the class. Sometimes, *StubsGenerator* needs to prepare a stub before its actual generation (it relates to the out-of-order generation mentioned before). This stub can be generated empty but needs to exist. For this purpose, *StubsGenerator* calls your class to get an empty AST node of a concept specified by an *MpsEntitySpec*, and later on, when the actual generation of this AST nodes takes place, it passes this unfinished AST node via the *alreadyGeneratedPart* parameter. Figure 3.27 illustrates an implementation of this second method corresponding to the example in Figure 3.26.



```
@Override
public MpsEntity generateEmptyMpsEntity(final MpsEntitySpec mpsEntitySpec) {
  string entityKind = getMpsEntityKind(mpsEntitySpec);

  switch (entityKind) {
    case "Model" :
      return new Model(mpsEntitySpec, "MyModelWithStubs");
    case "MyLanguageConcept" :
      return new Stub(mpsEntitySpec, new node<MyLanguageConcept>());

  }

  return null;
}
```

Figure 3.27: Example of the generateEmptyMpsEntity method

It actually looks the same as the first method but it would not if the concept of our example language had some children, references or properties. Then the *generateMpsEntity* would be more complex and different from the *generateEmptyMpsEntity* method.

With this knowledge, you now see that our example implementation of the *generateEmptyMpsEntity* method is actually wrong as we do not take the *alreadyGeneratedPart* into account. We should check if it is null and if not, we should use it for our stub generation.

This is the end of this tutorial, this is more-or-less all you need know to create a simple stubs generator for your base language. If you had any problem, look into the documentation of *StubsGenerator*, into its in-source comments, or into the *CsStubsGenerator* implementation.

3.5.5.3 Stubs Generation For Library Tree

If you want to generate stubs for a library that depends on another libraries (including the standard library), the work is more complicated.

Since our implementation of generation of stubs does not support finding of already existing stubs (it is not so trivial), you cannot use the first-idea approach of generating first the libraries which your library depends on and then your library. This will not work.

You have to generate XML files using the DLL parser for each of the libraries separately and then merge them by hand. That is, just concatenate the XML files. If you do not know structuring of XML files or you are not sure about what we mean by concatenating the files, do the following:

1. Take all the content of one of the XML files and put it into a new XML file. This will be the resulting file.
2. Then, for each of the other XML files, take the whole content of the *Assembly* element (without the *Assembly* element itself) and put it into the *Assembly* element of the resulting file, just before the ending tag.

Now, run the plugin stubs generation for this resulting file. The steps are the same as in case of generation of stubs for one library.

3.5.6 Distribution

The constructed stubs for the C# standard library are encapsulated in a separate plugin. The sources can be found on GitHub (see links in the *Appendix B*) and the binary plugin can be obtained via the JetBrains MPS Plugin Marketplace.

The DLL parser is a standalone C# program whose source code is located on GitHub (the link is again in *Appendix B*). No binary form of this program is distributed but it can be easily compiled from the provided source code.

StubsGenerator and *CsStubsGenerator* are either distributed as plugins accessible via the JetBrains MPS Plugin Marketplace or they can be built from sources which are accessible at GitHub (links are listed in *Appendix B*).

3.5.7 Encountered Problems

The generation of library stubs for the C# standard library appeared to be much more difficult than estimated at the beginning and during the development. There were several major challenges:

- Finding a way how to generate AST trees inside MPS programmatically. As there is not much MPS documentation about this and a quite large part of the existing documentation is deprecated, it was quite hard to get even a least working prototype.
- Constructing type references properly. There are many places in the library stubs that reference a type (e.g. return types of methods), another class (e.g. inheritance lists), etc. These references might be, and in the C# standard library they are, very complex. They contain the wildest combinations of generic types or array structures, classes are declared in a large number of outer classes or namespaces, there are naming collisions if the namespaces are ignored and so on. Getting a proper working solution cost us a lot of time.
- The DLL parser and the *CsStubsGenerator* modules are quite tightly coupled through the XML file content. Because these modules were developed by different team members, a lot of communication was needed. This necessarily caused that there was quite a great amount of waiting time and several cases of misunderstanding of the requirements asked by the other party.

Some particular minor problems include that C# Reflection does not behave in a way we needed. A lot of data restructuring was necessary. One representative for all, open types were a problem as obtaining their full name returns a null value and therefore we must have constructed the *entity IDs* in a more complicated way for these types.

We also had to introduce a black list into our solution as we needed to restrict the libraries to supported subset of the C# standard library API (we support only subset of the C# 5.0 specification).

Also the generation of libraries that depend on other libraries is not particularly user-friendly, although possible.

These problems caused that the DLL parser and the *CsStubsGenerator* are not optimally implemented. Some problems impacted even the C# base language itself. For example the references to generic type parameters in the library are raw strings and not true references to the generic type parameter declarations.

However, the implementation is not bad. But we would do it slightly differently if we would do it again, especially in case of the type references.

3.6 Documentation

An important part of our project is the provided documentation. As the Base Language has practically no documentation, neither about its current state nor about its development, our project intended to provide a (relatively) large amount of documentation that can be used by developers of future base languages. Furthermore, a proper documentation should help to keep our base language well-maintained.

The documentation of our project consists of several parts:

- The most important documentation is this file. It contains a detailed, overall picture of our project and its parts. However, its problem is that it will get outdated over the time of maintenance of the project. Therefore we also delivered the following kind of documentation:
- GitHub documentation of our project's parts contains detailed design overviews and manuals how to get everything working. It should be easier to keep it up-to-date.
- The GitHub projects contain also in-source documentation with low-level information regarding the particular projects.
- In-source documentation inside the Structure aspect of our language contains some low-level documentation about the concepts of the language. It also contains names of grammar entities from the grammar of C# 5.0 corresponding to our concepts.

4 Deployment and Usage

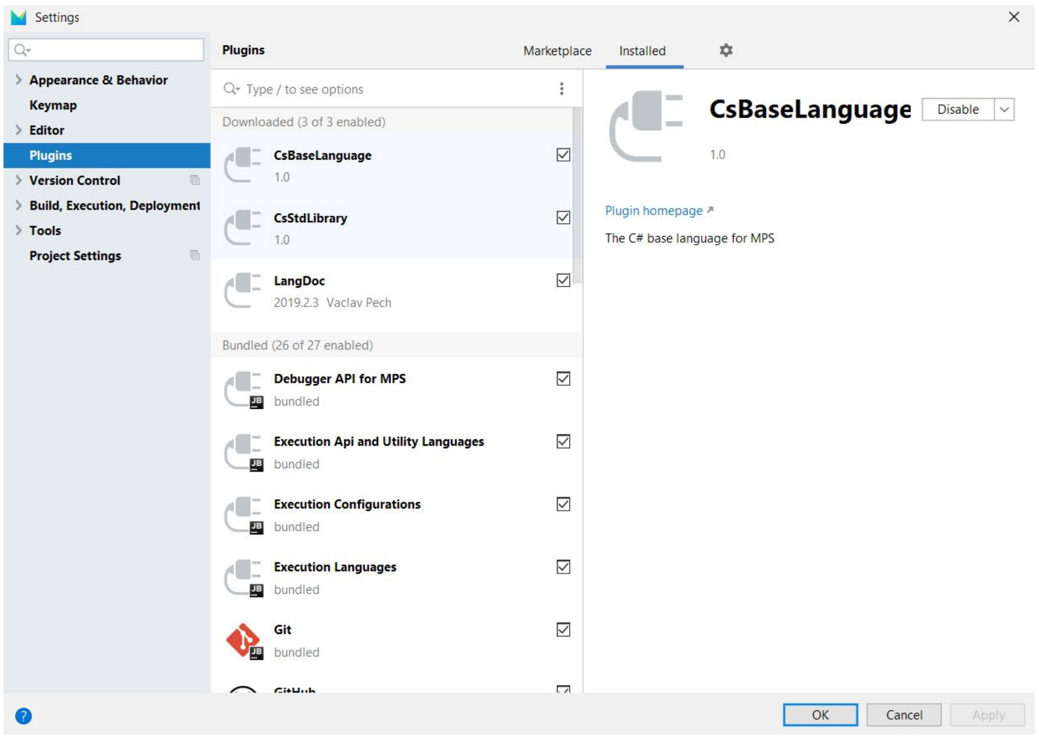
This chapter consists of several tutorials related to deployment and utilization of our language and other parts of our solution. You should find here everything you need to be able to make use of our work.

4.1 Creating C# Based MPS Program

In this tutorial, you will create an MPS solution project with C# base language code.

First, you will need to install a set of plugins in order to be able to link your project to the C# base language and to the C# standard libraries.

Please, go to *File*, *Setting*, *Plugins*, *Marketplace*. Use the search field to find a plugin called *LangDoc*. Install it. Do the same for plugins *CsBaseLanguage* and *CsStdLibrary*. Then, restart the MPS. Now, when you return to *File*, *Settings*, *Plugins*, *Installed*, you should see what is illustrated in Figure 4.1.



Downloaded (3 of 3 enabled)

CsBaseLanguage 1.0 ✓
CsStdLibrary 1.0 ✓
LangDoc 2019.2.3 Vaclav Pech ✓

Bundled (26 of 27 enabled)

Debugger API for MPS bundled ✓
Execution Api and Utility Languages bundled ✓
Execution Configurations bundled ✓
Execution Languages bundled ✓
Git bundled ✓

CsBaseLanguage Disable
1.0
Plugin homepage ↗
The C# base language for MPS

Figure 4.1: Plugins needed for creating a C# base language program

After closing the *Settings* window, please, select *File*, *New*, *Project*, as in Figure 4.2.

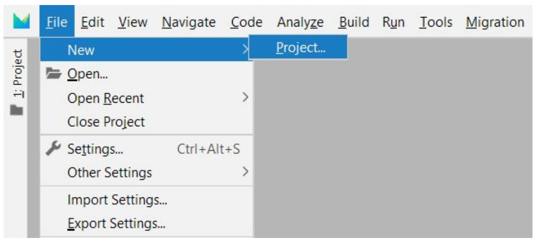
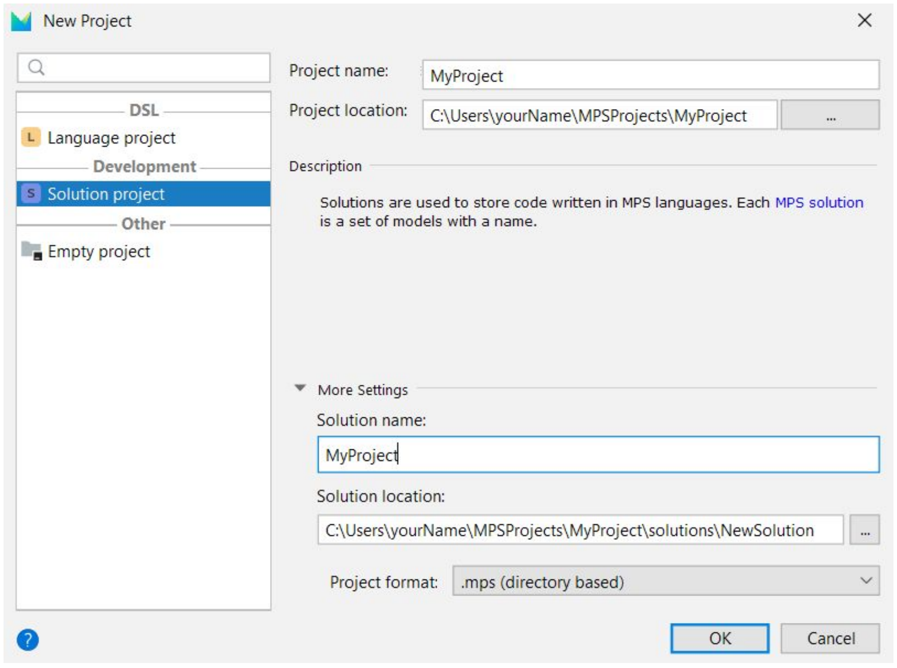


Figure 4.2: Creating a project

Select *Solution Project* and type in the project's and the solution's names. This is illustrated in Figure 4.3. Push the *OK* button.


New Project

DSL
Language project
Development
Solution project
Other
Empty project

Project name: MyProject
Project location: C:\Users\yourName\MPSProjects\MyProject

Description

Solutions are used to store code written in MPS languages. Each MPS solution is a set of models with a name.

More Settings
Solution name: MyProject
Solution location: C:\Users\yourName\MPSProjects\MyProject\solutions\NewSolution
Project format: .mps (directory based)

OK Cancel

Figure 4.3: Creating a solution project

Then, create a new model (Figure 4.4). You will be asked to fill in the model's name. In this step, do not delete the auto-filled-in prefix and just append your model's name, as in Figure 4.5.

After doing so, you can add the model dependencies in the newly opened sub-window or you can close it and do it later. This tutorial will guide you the way with closing the sub-window and adding the dependencies from scratch, so that you can learn how to add them later than when you are creating a model.

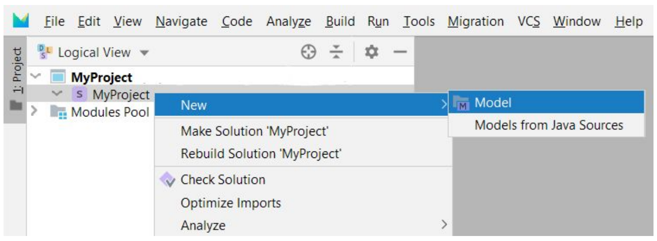
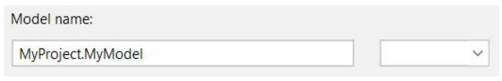


Figure 4.4: Creating a model

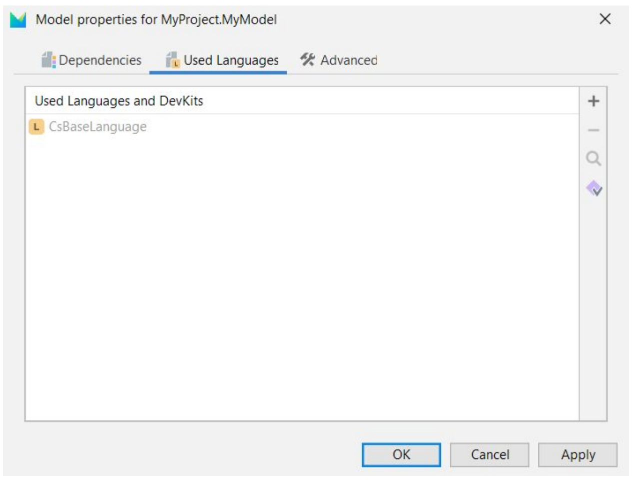

Model name:

MyProject.MyModel

Figure 4.5: Naming a model

To add the dependencies after closing the sub-window, right-click your model and select *Model Properties*.

Under the tab *Used languages*, you can add a dependency on the C# base language. Use the plus button on the right, type in *CsBaseLanguage* and select the corresponding item. The result is illustrated in Figure 4.6.


Model properties for MyProject.MyModel

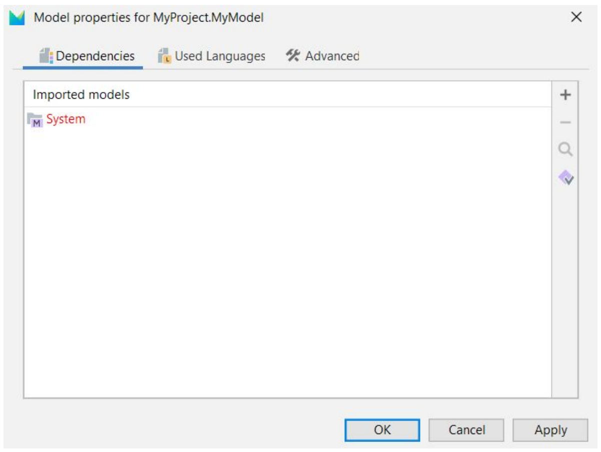
Used Languages and DevKits

CsBaseLanguage

Figure 4.6: Dependency on the C# base language

Under the tab *Dependencies*, you can add dependencies on libraries (more precisely library stubs). For example, we will add a dependency on the *System* namespace of the C# standard library. Hit the plus button on the right and type *System*. Select the *System* item. You should see what is presented in Figure 4.7. Note that you cannot select a dependency on the whole *CsStdLibrary*, you must select particular models (top-most namespaces) in it.

Close the sub-window with the dependency settings.


Model properties for MyProject.MyModel

Imported models

System

Figure 4.7: Dependency on the System model

Now, add a new root AST node. There is only one root AST node in our base language, called *File*. Right-click the model, select *New* and then *File* (Figure 4.8).

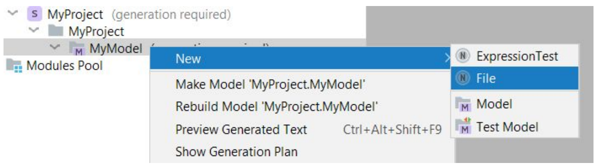
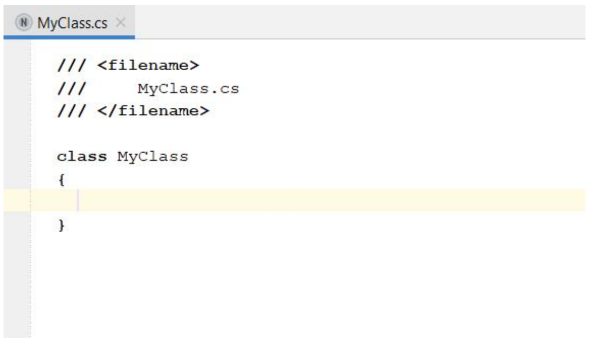

MyProject (generation required)
MyProject
MyModel
Modules Pool
New
Make Model 'MyProject.MyModel'
Rebuild Model 'MyProject.MyModel'
Preview Generated Text Ctrl+Alt+Shift+F9
Show Generation Plan
ExpressionTest
File
Model
Test Model

Figure 4.8: Creating a new File AST root node

In the created root AST node, you can write your code. For example, your first class (Figure 4.9). Importantly note that whenever you will experience some difficulties when writing the code, you will probably be able to proceed using the keyboard shortcut *CTRL+Space*. This shortcut offers you all possible AST nodes that can be created in the cursor's location. It is very often used when writing code in MPS because not always is the projectional editing of any selected language fluent as in a text editor.



MyClass.cs

```
/// <filename>
///     MyClass.cs
/// </filename>

class MyClass
{

}
```

Figure 4.9: Creating the user's first class

If you wish to generate C# source code for your just implemented *File*, right-click the item corresponding to your *File* in the project explorer and trigger *Preview Generated Text* (as in Figure 4.10).

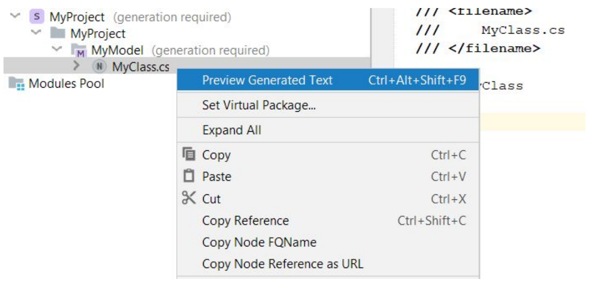
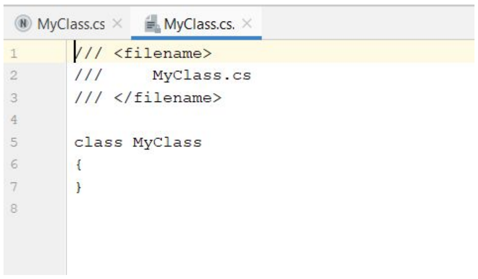

MyProject (generation required)
MyProject
MyModel (generation required)
MyClass.cs
Modules Pool
Preview Generated Text Ctrl+Alt+Shift+F9
Set Virtual Package...
Expand All
Copy Ctrl+C
Paste Ctrl+V
Cut Ctrl+X
Copy Reference Ctrl+Shift+C
Copy Node FQName
Copy Node Reference as URL

Figure 4.10: Generating the source code

Then you should see that MPS opened another tab with the source code for your *File*. For example, for our program from Figure 4.9, it would look like what is presented in Figure 4.11.



```
/// <filename>
///     MyClass.cs
/// </filename>

class MyClass
{
}
```

Figure 4.11: Source code for the created user's first class

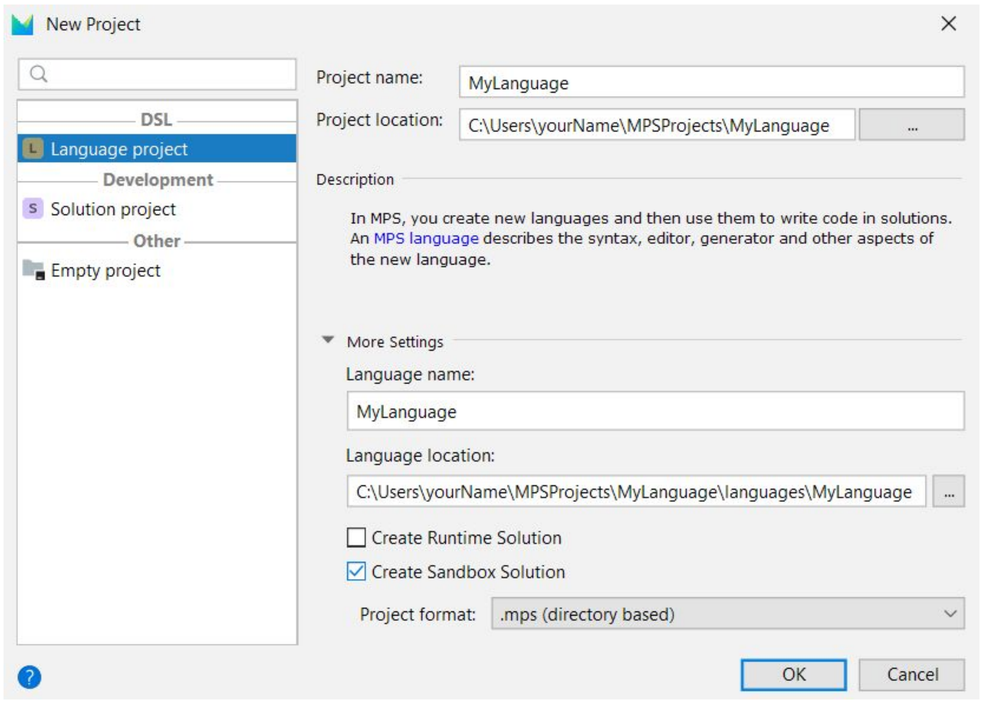
From this point on, you are on your own. For more help, remember the keyboard shortcut *CTRL+Space* and see the other tutorials in this document. You will probably also need to see the JetBrains MPS documentation for more general idea of the MPS functionality.

4.2 Creating C# Based MPS DSL

In this tutorial, you will create a very simple DSL based on the C# base language.

Please, follow the same initial steps as in the tutorial presented in Section 4.1 until you get to the *New Project* window and, according to that tutorial, you should create a solution project. This is the point from where this tutorial goes its own way.

In *New Project* window, you should select *Language project* and type in your project's and language's name, as in Figure 4.12. You may also want to check the *Create Sandbox Solution* checkbox if you want to experiment with your language while designing it. This is really recommended. When done, push the *OK* button.



Project name: MyLanguage
Project location: C:\Users\yourName\MPSProjects\MyLanguage

Description

In MPS, you create new languages and then use them to write code in solutions. An MPS language describes the syntax, editor, generator and other aspects of the new language.

Language name: MyLanguage
Language location: C:\Users\yourName\MPSProjects\MyLanguage\languages\MyLanguage

☐ Create Runtime Solution
☑ Create Sandbox Solution

Project format: .mps (directory based)

Figure 4.12: Creating a language project

Go to the project explorer and expand the item *MyLanguage* with the yellow icon with the capital letter *L*. Then, right-click the *Structure* item, select *New* and *Concept*. As illustrated by Figure 4.13. This will create a concept, i.e. the modelling description for AST nodes that users of your language will create.

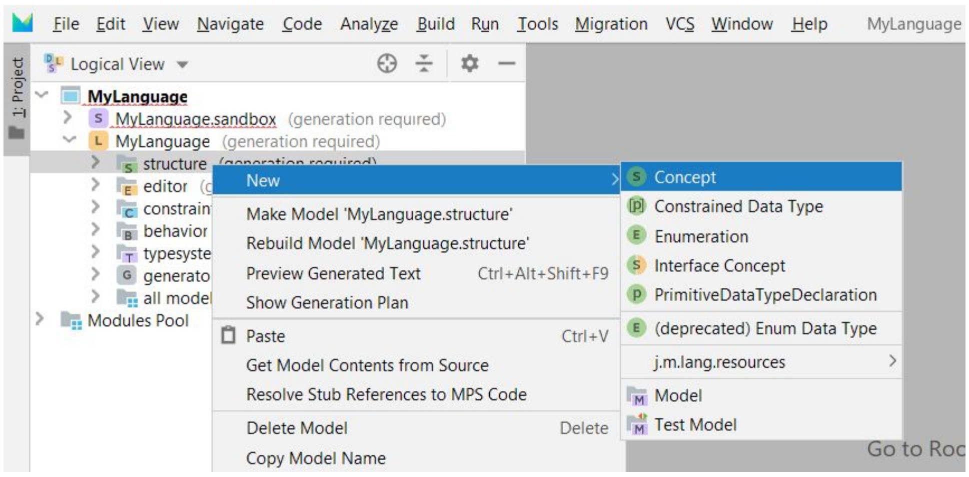
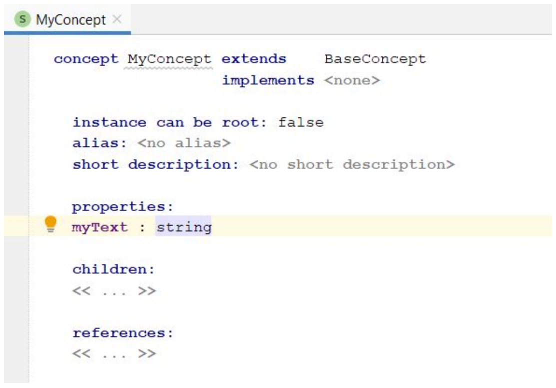

File Edit View Navigate Code Analyze Build Run Tools Migration VCS Window Help MyLanguage
1: Project
Logical View
MyLanguage
MyLanguage.sandbox (generation required)
MyLanguage (generation required)
structure
editor
constrain
behavior
typesyste
generato
all model
Modules Pool
New
Make Model 'MyLanguage.structure'
Rebuild Model 'MyLanguage.structure'
Preview Generated Text Ctrl+Alt+Shift+F9
Show Generation Plan
Paste Ctrl+V
Get Model Contents from Source
Resolve Stub References to MPS Code
Delete Model Delete
Copy Model Name
Concept
Constrained Data Type
Enumeration
Interface Concept
PrimitiveDataTypeDeclaration
(deprecated) Enum Data Type
j.m.lang.resources
Model
Test Model
Go to Roo

Figure 4.13: Creating a language concept

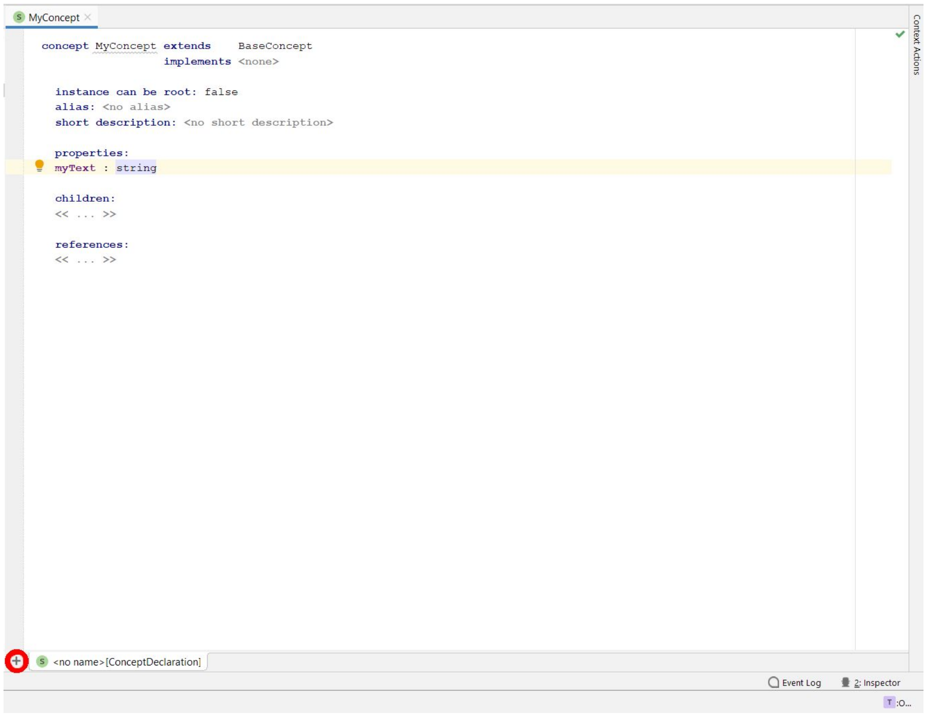
Now, fill in the name of the concept and, for example, add a string property for it. You can inspire by Figure 4.14.



```
concept MyConcept extends    BaseConcept
                  implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  myText : string

  children:
  << ... >>

  references:
  << ... >>
```

Figure 4.14: Initializing the created concept

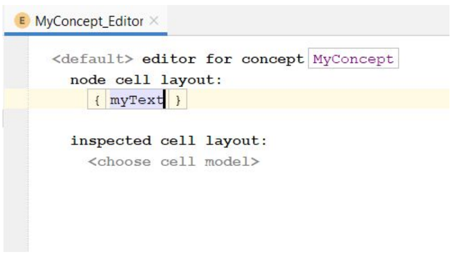
Then, you should create an Editor for your concept so that it can be nicely displayed to your users. Push the plus button on the bottom of the window (Figure 4.15) and select *Editor* and then *Concept Editor*.



```
concept MyConcept extends    BaseConcept
                  implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  myText : string

  children:
  << ... >>

  references:
  << ... >>
```

Figure 4.15: Adding a new aspect for a concept

In the opened tab, go with your cursor to the red rectangle and hit *CTRL+Space* shortcut and select *{myText}*, just as in Figure 4.16.



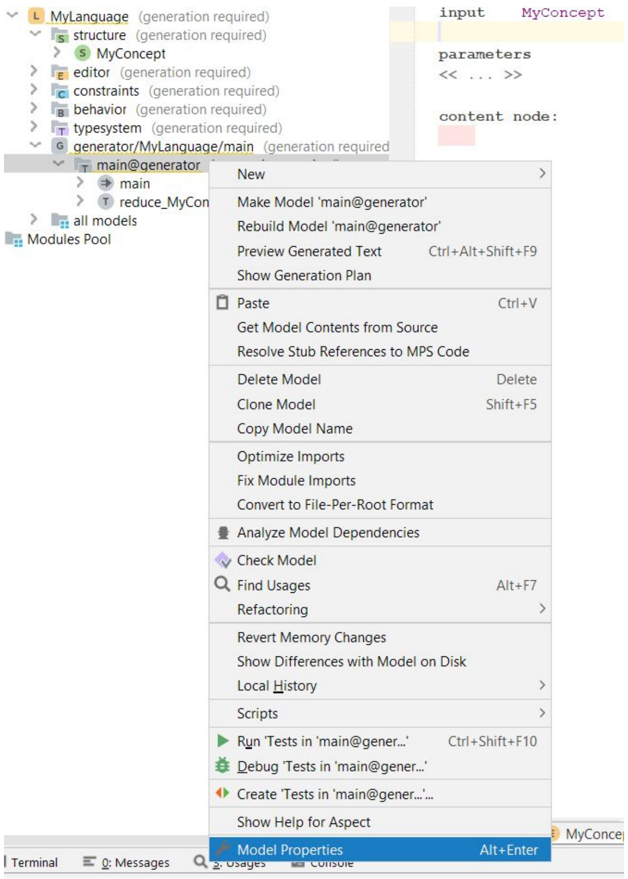
```
<default> editor for concept MyConcept
  node cell layout:
    [ myText ]

  inspected cell layout:
    <choose cell model>
```

Figure 4.16: Initializing the first concept's editor

Then, you will create the Generator aspect for your concept. This will serve to transform AST nodes of your concept into chunks of code of the C# base language. Push the plus button again, select *Generator* and the *Template Declaration*.

Now, you will add a dependency on the C# base language so that you will be able to use it for the above-mentioned chunks of code.

Expand the *generator/MyLanguage/main* item in the project explorer, right-click the *main@generator* item and select *Model Properties* (see Figure 4.17).


MyLanguage (generation required)
structure (generation required)
MyConcept
editor (generation required)
constraints (generation required)
behavior (generation required)
typesystem (generation required)
generator/MyLanguage/main (generation required
main@generator
main
reduce_MyCon
all models
Modules Pool
New
Make Model 'main@generator'
Rebuild Model 'main@generator'
Preview Generated Text Ctrl+Alt+Shift+F9
Show Generation Plan
Paste Ctrl+V
Get Model Contents from Source
Resolve Stub References to MPS Code
Delete Model Delete
Clone Model Shift+F5
Copy Model Name
Optimize Imports
Fix Module Imports
Convert to File-Per-Root Format
Analyze Model Dependencies
Check Model
Find Usages Alt+F7
Refactoring
Revert Memory Changes
Show Differences with Model on Disk
Local History
Scripts
Run 'Tests in 'main@gener...' Ctrl+Shift+F10
Debug 'Tests in 'main@gener...'
Create 'Tests in 'main@gener...'...
Show Help for Aspect
Model Properties Alt+Enter
MyConce
Terminal
0: Messages
3: Usages
Console

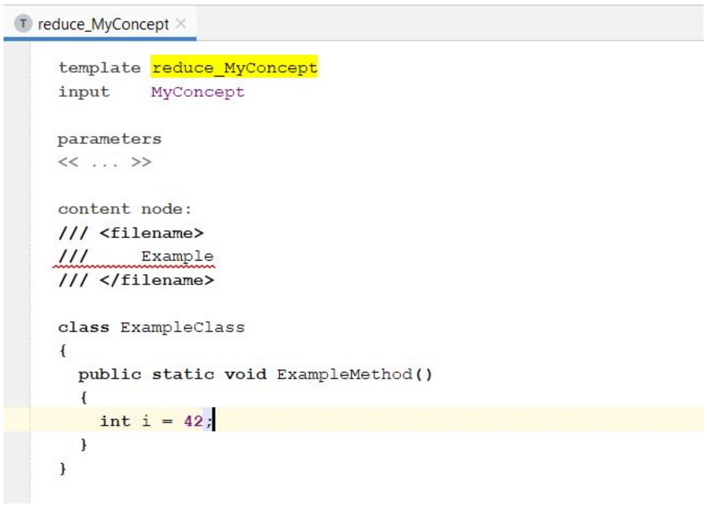
```
input    MyConcept

parameters
<< ... >>

content node:
```

Figure 4.17: Setting dependencies for Generator

In the opened window, under the tab *Used languages*, push the plus button on the right. Type in *CsBaseLanguage* and hit *Enter*. You should get the result presented by Figure 4.6 from the previous Section *4.1*. By this, you have added the C# base language among the languages that you can use in the Generator aspect.

Then, under the tab *Dependencies*, push the plus button again, type in *System* and select the found item. Then you should see what is illustrated by Figure 4.7 in the previous section *4.1*. This has made available the *System* namespace of the C# standard library.

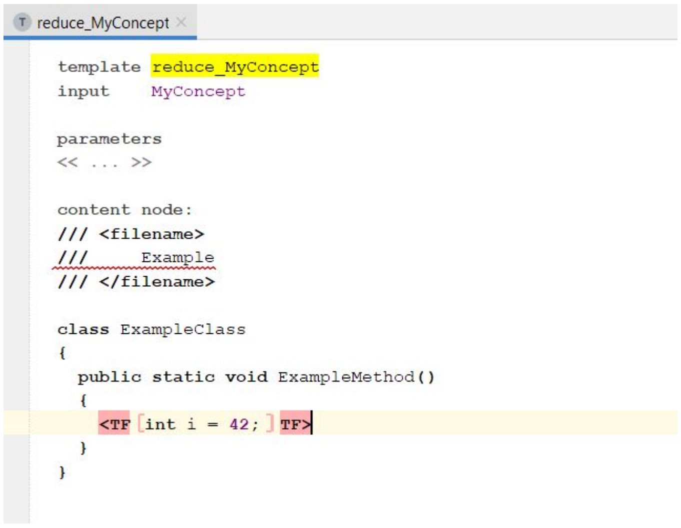
Now, return to your Generator for your concept. Put cursor in the red rectangle and start creating a C# base language program. Start with a *File* AST node and then for example, create what is illustrated in Figure 4.18. Do not worry about the red underlined text, you will fix it in a while.



```
template reduce_MyConcept
input    MyConcept

parameters
<< ... >>

content node:
/// <filename>
///      Example
/// </filename>

class ExampleClass
{
  public static void ExampleMethod()
  {
    int i = 42;
  }
}
```

Figure 4.18: Basics of the created Generator

Put your cursor as in Figure 4.18, hit *CTRL* together with the up arrow twice. Then hit *Alt+Enter* and select *Create Template Fragment*. You should obtain what is shown in Figure 4.19.



```
template reduce_MyConcept
input    MyConcept

parameters
<< ... >>

content node:
/// <filename>
///      Example
/// </filename>

class ExampleClass
{
  public static void ExampleMethod()
  {
    <TF [int i = 42; ] TF>
  }
}
```

Figure 4.19: Template Fragment

The code wrapped in the template fragment, i.e. in the marked area, is the chunk of C# base language code that will be used whenever an AST node of your concept will be transformed into C# base language. If you define such Generators for all the concept in your DSL in such a way that they will fit together, the result of the transformation of a program written in your DSL into C# base language will be a valid C# base language program.

Of course, the DSL created in this tutorial is unusable, but the information you learned here should be enough to get you started using our C# base language for DSL development. To develop more complex DSLs, you will probably have to dive into the MPS documentation. We strongly recommend to start first with the tutorial in Section *4.1*.

4.3 Building C# Base Language

This tutorial will guide you through building the plugin with our C# base language from scratch.

First, you will need to install the *LangDoc* plugin since our base language depends on it. Go to *File*, *Settings*, *Plugins*, *Marketplace* and search for the *LangDoc* term. You should see something similar to what is presented in Figure 4.20. Now, install the found *LangDoc* plugin. You will be probably asked for restarting the JetBrains MPS. Please, do so.

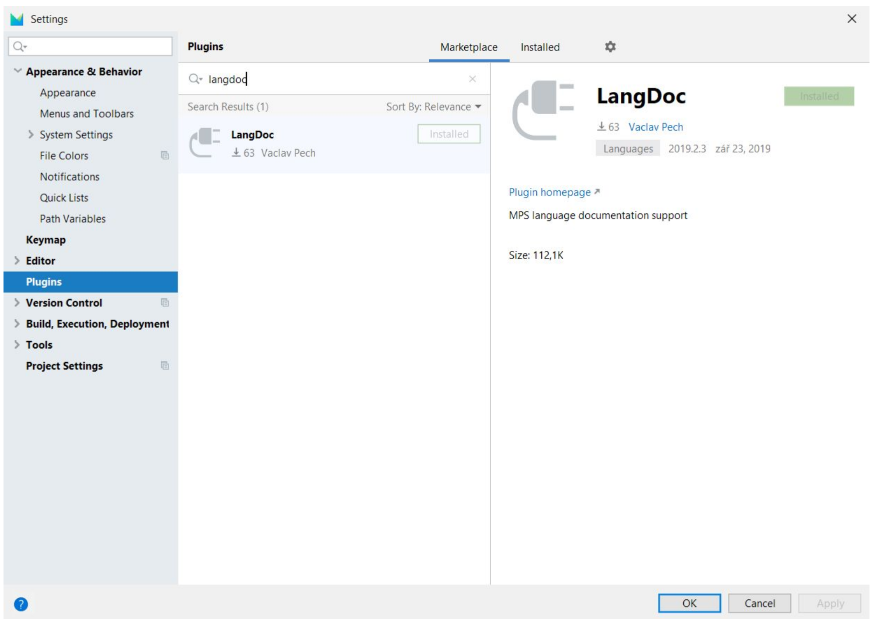


Figure 4.20: Searching for the LangDoc plugin

Then, get our source code of the C# base language. It is available on GitHub (see the link in *Appendix B*), you can either clone the project or download an archive containing the source code.

When you are done, open the project with JetBrains MPS. You can do that via *File*, *Open* and selecting the directory *mpscs* that you have just cloned or downloaded from GitHub.

Now, right-click the project in the project explorer and trigger *Rebuild Project*[12], as in Figure 4.21. This will compile the language and all related parts in this project.

[12] If you do not want to rebuild the whole project, you may rebuild just the solutions *CsBaseLanguage*, *CsBaseLanguage.plugin* and *CsBaseLanguage.build*.

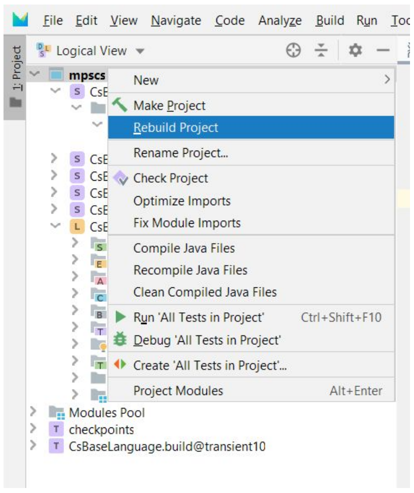

File Edit View Navigate Code Analyze Build Run Too
Logical View
mpscs
New
Make Project
Rebuild Project
Rename Project...
Check Project
Optimize Imports
Fix Module Imports
Compile Java Files
Recompile Java Files
Clean Compiled Java Files
Run 'All Tests in Project' Ctrl+Shift+F10
Debug 'All Tests in Project'
Create 'All Tests in Project'...
Project Modules Alt+Enter
Modules Pool
checkpoints
CsBaseLanguage.build@transient10

Figure 4.21: Rebuilding the mpscs project

After that, you should generate the TextGen aspect. For this, you will need the *TextGenGen* plugin. Please, visit the Marketplace, search for the plugin and install it. The procedure is the same as in case of the *LangDoc* plugin.

When you have the *TextGenGen* plugin installed and your MPS restarted, right-click the *CsBaseLanguage* item with the yellow icon, which you can see in the project explorer, and select *Generate TextGen*. See Figure 4.22 if you are unsure.

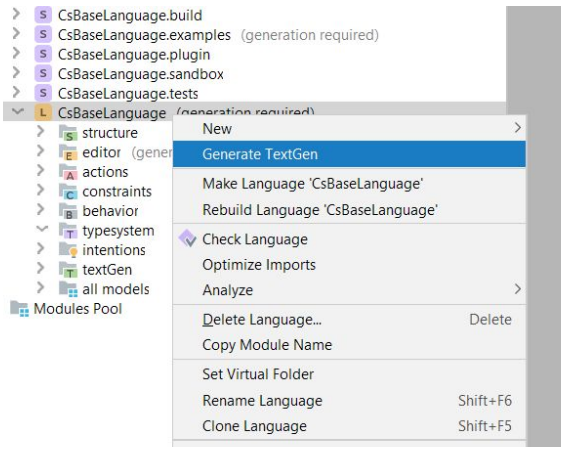


Figure 4.22: Generating TextGen

After generation of the TextGen aspect, you should rebuild the project once more. Right-click the project and select *Rebuild Project*.

Now you are ready to build the plugin. Please, expand the *CsBaseLanguage.build* item in the project explorer, right-click the shown *CsBaseLanguage* item with an icon of a spider and select *Run CsBaseLanguage* (see Figure 4.23). After a while, you should see a console output ending with a *BUILD SUCCESSFUL* message. This means that the plugin has been successfully constructed.

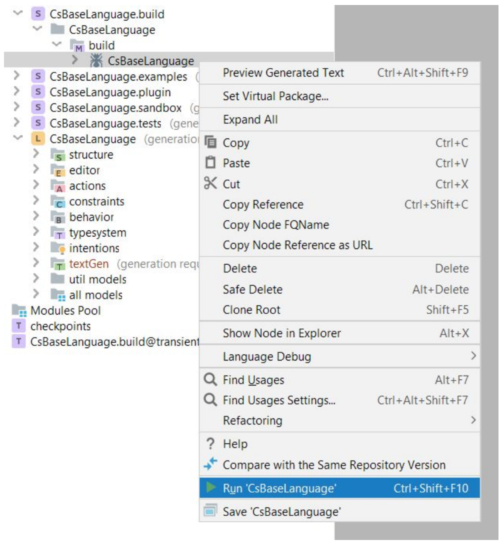


Figure 4.23: Building the C# base language plugin

You can find the plugin in the directory with the C# base language source code. Go to that directory and then to directories *CsBaseLanguage*, *build*, *artifacts* and *CsBaseLanguage*. There should be a ZIP archive called *CsBaseLanguage.zip* - this is the built plugin. Do not extract the archive, the plugin is the whole archive.

Now you should have a complete plugin, similar to what is distributed via the JetBrains MPS Marketplace.

4.4 Other Tutorials

Some tutorial have been already presented in *Chapter 3*:

- The tutorial of how to build library stubs is presented in Sections *3.5.5.1* and *3.5.5.3*.
- The tutorial about running language tests is located in Section *3.3.4.2*.

5 Examples

To test our product from the point of view of a real user, we decided to prepare a set of examples which make use of our C# base language.

There are two kinds of expected usage scenarios of the C# base language:

- Most of the users will be probably language designers, who will create MPS domain-specific languages transformable into the C# base language via the Generator aspect (or in other words, via a model-to-model transformation).
- A small set of users are expected to use our C# base language also for regular programs that are written directly in the C# base language. This might seem strange and you can ask the question of why these users should use MPS and not traditional C# IDEs for a such task. It can be convenient when a program consists of a combination of C# code and some other DSL code.

Both of these kinds of usage scenarios have more-or-less similar requirements. The C# base language must allow creation of common constructs in C# (more precisely C# 5.0, as we discussed in Section *3.3.1*) and it should allow the user to write them in a user-friendly way. Since the majority of usage scenarios will be implementations of DSLs based upon our C# base language and since the language designers do not need completely user-friendly editing of code (as language designers usually create only small chunks of code and only once), editing of C# base language code does not have to be completely fluent. A reasonable level suffices.

To test that these requirements were satisfied, we decided to create an example DSL and several example programs. Even though it may seem that higher number of example DSLs should be prefered over higher number of example programs, since DSLs are the number-one expected usage scenario, we thought that implementing more programs than DSLs would test the C# base language better. We had two reasons:

- Programs typically contain more various constructs than what is usually used in the Generator aspect of a DSL.
- It is faster to create a program than a DSL and therefore we could deliver more tests in a shortening time before the project's deadline.

Here is a list of example programs that we implemented:

- **Hello World**. The well-known program of printing a message to the world.
- **Quicksort**. A program implementing the popular *Quicksort* algorithm. It tests mainly coding of expressions.
- **Sort Comparison**. A program checking whether algorithms *Bubble Sort* and *Selection Sort* return the same results. This example tests mainly whether a code split into multiple files can be implemented in our C# base language.
- **Word Count**. A program counting words in a file. This tests primarily whether the C# standard library can be properly used in our product.

The implemented example DSL, called **Calculator**, is inspired by a similar example DSL for the Base Language (i.e. the Java base language bundled as the main base language in MPS). It is a very simple DSL allowing creation of parameterized expressions. Even though it is so simple, it fully demonstrates the possibility of using our C# base language for implementation of DSLs.

To test the example DSL we also delivered a small example program written in it.

All the examples are located in a separate GitHub repository. The link can be found in *Appendix B*.

6 Evaluation

In this chapter, we conclude the CS4MPS project. We would like to evaluate the work that was done and compare it with our goals and expectations from the beginning of the project.

We also list some possible future extensions of our work.

6.1 Goals Satisfaction

The result of the project is a C# base language with the specification from Section *3.3.1*. A C# base language has been requested by a portion of MPS community that wanted their DSLs to be compilable for the .NET platform. We delivered a full-blown MPS language, including a port of the C# standard library, that can be used exactly for this purpose. This means that language designers are able to provide a model-to-model transformations of their DSLs into our C# base language, which makes the DSL programs transformable into a C# source code that is ready for compilation.

The C# base language can also be used for development of standalone C# programs in MPS, although that is not expected to be a major use-case.

The delivered solution also contains a large portion of documentation, which is helpful for future maintainers of our base language and for developers of other future base languages.

In the specification document, we described what C# features should have been supported, what should have been supported if the time would permit and what features would not be implemented.

In the end, we did not implement these promised features:

- The *using* keyword. Since the dependencies in MPS work in a totally different fashion than is common in case of the traditional C# (not via *using* keywords but via setting dependencies using MPS GUI) and implementation of the using keyword functionality would be really non-trivial, we decided to adapt to this MPS mechanism and not to implement the functionality around the *using* keyword.
- The *dynamic* built-in type. We decided not to implement it at this stage of development of the language as its currently possible form may be a problem when the Type System aspect will be created.

On the contrary, we implemented quite a few features that we did not promise, only if the time would permit. Namely:

- Delegate + modifier,
- Partial types (syntactically),
- Greater support of expression variability,
- Full syntax support for literals,

- Partial support of checked/unchecked statements,
- Partial support of lock statements,
- Partial support of try statements.

6.2 Quality of Work

In this section, we discuss the quality of the delivered solution in a few independent sub-sections that target some highlights or problems of our work.

6.2.1 Structure

The Structure aspect of our language is designed in a way that leads to easy extensibility as we took an advantage of the abstract concepts and interfaces that are both features of MPS. This approach revealed itself as very effective later in the development when we needed to slightly modify the language. It was very easy to do so because the concepts are, thank to this approach, loosely coupled.

While working on the language structure, we were able to match the C# grammar almost one to one. That means that almost each concept of our base language corresponds to a symbol of the C# 5.0 grammar. This should be helpful when the language shall be extended to support further versions of C# or a larger subset of the C# 5.0 features.

These facts make us believe that in the future, when the language shall be extended, it will not be too difficult to do so.

6.2.2 TextGenGen

During our work we improved a third-party plugin intended for TextGen aspect generation from the Editor aspect. The made modifications significantly improved the plugin and therefore it can be now utilized for very serious use cases. That could really help the MPS community.

6.2.3 Library Stubs

The solution around generation of library stubs has one highlight and one limitation.

The highlight is that we delivered a language-independent plugin that significantly eases creation of library stubs generators. This could really help future base language designers.

However, we can now, after solving the encountered problems, see that the design of this plugin could be slightly better. It is not a big limitation and the plugin is surely well-usable but it could be further improved (it could require less effort to use it and the data processing could be simpler). Furthermore, the related modules functioning as adapters of this plugin to our base language have this limitation as well, but maybe in a slightly greater level. Still, we think that it is quite fine.

6.2.4 Scoping

A limitation of our solution that can be seen at the first sight is that the auto-completion does not take scoping into an account. This is sometimes problematic but we unfortunately did not have time to improve this.

Although unpleasant, it is not something that we have promised in the specification document.

6.3 Possible extensions

Although our solution is fully functioning, there are still some things that can be done to further improve it. The following sections form a list of improvement suggestions for the future maintainers of the C# base language.

6.3.1 Language structure

The C# language is still evolving and new features are being added all the time. There are also some features of C# 5.0 we did not implement due to the time constraints. In the future, these features might become more important than they are right now and they might be necessary to deliver.

6.3.2 Fluent editing

Although we implemented a reasonable level of fluent editing and completed our goal for this project, the editing is not as good as in the case of the Base Language, which has been implemented for several years. The greatest difference is in creation of not fully-formed structures, e.g. writing a method header from left to right.

Until fluent editing of the C# base language reaches the level of Base Language's fluent editing, there will be always some work to be done.

6.3.3 Scoping

Part of code editing is the ability to provide the user with auto-completion suggestions while writing the code. This is achieved by using MPS smart references. We implemented the simple mechanism that serves this purpose. However, we did not implemented it in such a way that would recognize the difference between different scopes. In our version, all possible entities in the program are always suggested when auto-completion is triggered.

6.3.4 Generic Type Parameter References

The Structure concepts around type references are sub-optimal. We needed to use dummy references to generic type parameters because of complications in the generation mechanism of library stubs.

The optimal solution is described in the in-source comments at the concepts around type references.

6.3.5 Stubs Generation For Library Tree

As visible in Section *3.5.5.3*, generation of libraries that depend on another libraries is quite complicated.

If the plugin StubsGenerator was improved so that it would check whether some generated stubs already exist, the generation of such libraries would be more straight-forward. However, this modification of the plugin is not trivial. This is not because of the design of the plugin, but because of non-trivial querying of MPS and retrieval of information.

6.3.6 Type System

The language has no support for checking of type correctness. This is not necessary for serious development but it is a helpful feature for discovering type errors.

Implementing the complete Type System aspect is, however, very non-trivial and it is not even done fully for the Base Language.

7 Development

This chapter describes how we organized the development of the project and our gained experience.

7.1 Development organization

At the beginning, we identified the high-level development tasks that were needed to do in order to successfully finish the project:

- Create the Structure aspect,
- Construct editors for the Structure concepts,
- Create the TextGen aspect,
- Ensure fluent editing,
- Provide solution for generation of standard C# library stubs,
- Create tests,
- Create example usage scenarios of our base language,
- Polish our base language.

We intended to organize the development according to the Waterfall model. Since we needed to alter the Structure aspect during the whole process iteratively and the fluent editing stretched almost throughout the whole development, it was not a pure waterfall development in the end.

At the beginning, we split the team into two sub-teams. One focused on the Structure and the Editor aspects around the high-level part of the base language, which consists of concepts such as classes, namespaces or methods. And the other sub-team was responsible for similar work around the low-level part of the base language around expressions and statements. This work was more-or-less mechanical as we were mirroring the C# 5.0 grammar. Both sub-teams could work at their own pace and there was no delay caused by waiting for the work of someone else. The sub-teams utilized dual programming during this phase in order to deliver a well-designed core of the language.

During the second phase of the project, the second team continued working together on fluent editing, more-or-less until the end of the whole development. The first team split up into individuals who worked on different parts such as the TextGen aspect, library imports and tests.

At the end, we all focused on developing examples of usage of our base language and final polishing of the project.

7.2 Experience

We discovered that the Waterfall model worked quite well for this kind of project as there were no changing requirements over time.

We have seen that dual programming is an appropriate technique for achieving a good design of the important parts.

And we think that spliting the team into two sub-teams was also a good move as it was much easier to organize meetings of the sub-teams as there were less schedules to intersect. Frequent meetings were crucial for the dual programming technique.

Appendix

A. 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 code during the final stage of the source-code-building process
- **Base Language** = a base language representing Java
- **Concept** = definition of a DSL entity in the Structure language aspect. In other words, specification of inner structure of AST nodes
- **CsStubsGenerator** = our plugin for generation of stubs for the C# standard library
- **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
- **Generator** = a definition of a model-to-model transformation of one DSL to another
- **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
- **Model** = an MPS organizational unit of code similar to C# namespace, C++ namespace or Java package
- **Module** = an MPS organizational unit of code. It can be either a Solution, a Language (i.e. MPS DSL) or a Devkit (which is not important for our project)
- **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
- **Solution** = an MPS organizational unit of code representing a standalone program or a program module
- **Structure** = a definition of a language’s hierarchical structure in MPS
- **Stub** = an MPS representation of an external library entity
- **StubsGenerator** = our plugin for generation of stubs for external libraries
- **TextGen** = a model-to-text transformator used to generate source files from an AST
- **TextGenGen** = a plugin for generation of the TextGen aspect from the Editor aspect

B. Links to Project Repositories

Here are links to GitHub repositories containing source code of individual parts of our product. For more information, please, read the text in this document.

- The main repository with the C# base language:
  https://github.com/vaclav/mpscs
- Stubs for the C# standard library:
  https://github.com/wirthma/CsStdLibrary
- Example programs and an example DSL based upon the C# base language:
  https://github.com/Zeman-Dalibor/mpscs-examples
- The TextGenGen plugin:
  https://github.com/Kripner/textGenGen
- The StubsGenerator plugin:
  https://github.com/wirthma/StubsGenerator
- The CsStubsGenerator plugin:
  https://github.com/wirthma/CsStubsGenerator
- The DLL parser related to library stubs generation:
  https://github.com/Zeman-Dalibor/DotNetLibraryExporter

C. 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