Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

DSL-specific IDE generation

Bachelor thesis

Author: Pavel Pakhomov
Supervisor: Jan Trávníček

Prague, 2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Pakhomov Pavel**	Personal ID number:	**487587**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Software Engineering and Technology**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

DSL-specific IDE generation

Bachelor’s thesis title in Czech:

Generování IDE specifické pro DSL

Guidelines:

1. Analyze the Xtext as an Eclipse framework for the development of domain-specific languages (DSL), and the Xtext grammar language as a method for describing DSL.
2. Analyze Grammar-Kit as an IntelliJ IDEA plugin for generating parser support for languages in Backus-Naur form (BNF).
3. Analyze a way to develop plugins for IntelliJ platform-based IDEs.
4. Using the Grammar-Kit plugin, design an IntelliJ IDEA plugin as a tool to generate a DSL-specific IDE for an Xtext compatible language description.
5. Implement the designed IntelliJ IDEA plugin.
6. Select examples of DSL languages and test the plugin on those.
7. Compare the behaviour of the environment generated by the implemented plugin with the one generated by Eclipse Xtext for the same input.

Bibliography / sources:

1. "Domain Specific Languages" by Martin Fowler, with Rebecca Parsons 2010
2. "Implementing Domain-Specific Languages with Xtext and Xtend" - Second Edition by Lorenzo Bettini 2016

Name and workplace of bachelor’s thesis supervisor:

Ing. Jan Trávníček, Ph.D., Department of Theoretical Computer Science, FIT

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **26.02.2021** Deadline for bachelor thesis submission: **13.08.2021**

Assignment valid until: **19.02.2023**

Ing. Jan Trávníček, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Declaration

I hereby declare I have written this bachelor thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, 2021

....
Pavel Pakhomov

Acknowledgements

I would like to thank Ing. Jan Trávníček, Ph.D. for being the project's supervisor. Additionally I would like to thank Michail Golubev the, in-firm supervisor, and Ing. Jiří Šebek, the faculty guarantor, for the support during the project.

Abstract

This bachelor thesis describes the design and development of an IntelliJ IDEA plugin which generates language support infrastructure, integrating new programming languages to the IDE based on the language grammar specification. The goal of the theoretical part of the project is to study formal language theory and language grammar definition methods. The Xtext framework from the Eclipse IDE is discussed and its functionality and interface is replicated in the IntelliJ IDEA. This theoretical knowledge is subsequently applied in the practical section, which describes the entire implementation process. This part includes enriching the IntlliJ IDEA with the Xtext grammar language support logic, creating a Syntax Tree analyzer component, implementation of file generators and the mechanism that facilitates the translation between two different tree structures. The concluding part of the thesis presents the evaluation of the newly created plugin by testing the behaviour of the language environments it generates.

Keywords: IDE, DSL, grammar specification, parser, lexer, Abstract Syntax Tree, Grammar-Kit, Ecore model.

Abstract

Cílem této bakalářské práce je navrhnout a vyvinout plugin Intellij IDEA, který bude generovat infrastrukturu jazykové podpory a integrovat nové programovací jazyky do IDE na základě specifikace jazykové gramatiky. Cílem teoretické části projektu je zkoumaní teorie formálního jazyka a metod definice jazykové gramatiky. Je také analyzován Xtext framework z Eclipse IDE, jehož funkčnost a rozhraní byli replikováné v IntelliJ IDEA. Tyto teoretické znalosti jsou následně aplikovány v praktické části, která popisuje celý proces implementace. Implementační část zahrnuje obohacení IntlliJ IDEA o podporu jazyka Xtext, vytvoření komponenty analyzátoru syntaktického stromu, implementaci generátorů souborů a mechanismu realizujícího překlad mezi dvěma různými stromovými strukturami. Závěrečná část práce obsahuje zhodnocení vytvořeného pluginu prostřednictvím testování chování jazykových prostředí, které se generují.

Kličova slova: IDE, DSL, specifikace gramatiky, parser, lexer, Abstract Syntax Tree, Grammar-Kit, Ecore model.

List of Figures

Contents

Chapter 1

Introduction

In this current age of digital revolution, many companies are expected to write their own software. Domain-specific languages (DSLs) are increasingly vital in software development. DSLs can help domain experts participate in the software development process and can significantly improve communication between stakeholders. Building a DSL is usually a complex task that requires specific technical knowledge. Fortunately, there are several available useful tools that can help developers. The most powerful among them is Xtext (among java programmers) - a widely used framework in the Eclipse IDE. With Xtext, users can quickly create new DSL, defining it using powerful grammar language. With the help of Xtext, the user receives a full environment, including rich IDEs and text editors to support the new DSL.

Importantly, Eclipse is not the most progressive IDE and fewer and fewer users today are inclined to use it and prefer products from JetBrains instead. Currently, DSL developers familiar with Xtext do not have the possibility to transfer to a modern IDE without significantly changing their workflow. At the same time IntelliJ IDEA does not have such a powerful tool that DSL developers can use, so they have to spend a lot of time and effort to create the new language environment that Xtext users do with 1 click.

The main goal of this project is to design and implement a plugin in IntelliJ IDEA that will repeat the Xtext plugin functionality. The thesis will cover all sides from analysing two different approaches of language support creation to software implementation and testing.

Chapter 2

Language definition

2.1 Introduction

The thesis covers the specific topic of development of programming languages, therefore the reader is invited to get acquainted with the notion of formal languages and methods of their representation. There will be no deep analysis of *Formal Languages and Automata Theory* presented in the work, but the familiarity with basic concepts is required for better understanding of further material.

2.2 Basic concepts

According to [1]:

- **Alphabet** is a non-empty finite set often denoted as Σ. Members of Σ are called symbols. (e.g $\Sigma = \{a, b, c\}$ or $\Sigma = \{0, 1, 2, ..., 9\}$)
- A **string** or word over an alphabet Σ is a finite sequence of symbols of . (e.g. let $\Sigma = a, b$ be an alphabet; then *aa, ab, bba, baaba,...* are some examples of strings over Σ).
  The **empty string** is denoted by ε.
  The **set of all strings** over an alphabet is denoted by $\Sigma*$. (e.g. $\Sigma = \{0, 1\}$, then $\Sigma* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$).
- **Formal language** (further just **language**) over an alphabet Σ is a subset of $\Sigma*$ (the collection of strings over the alphabet). If this subset is finite, then the language is called **finite**, otherwise it is **infinite**.
  Languages also can be divided into **natural languages** (any language that has

evolved naturally in humans) and **computer languages** (a method of communication with a computer).

2.3 Finite Representation

Proficiency in a language does not require one to know all the sentences (strings) of the language; rather with some limited information one should be able to come up with all possible sentences of the language. This limited information describing the language is called a finite representation of the language. In the case of finite languages, the representation can just be an enumeration of all its strings. On the other hand, the representation of an infinite language(e.g. all natural and computer languages) is a more complex task. There are different ways to represent an infinite language, such as using mathematical expressions, automata, or the more commonly used method representation - **grammar**[2].

2.4 Grammar

Simply put, the grammar describes how to form strings of a language.
The classic formalization of grammars proposed by Noam Chomsky is a quadruple: $G = (N, \Sigma, P, S)$, where:

- N is a finite set of **non-terminal** symbols that is disjoint with the strings formed from G.
- Σ is a finite set of **terminal** symbols that is disjoint from N.
- P is a finite set of **production rules**.
- $S \in N$ is the **start symbol**.

Every **production rule** (also named a rewriting or derivation rule) has the following form: $(\Sigma \cup N) * N(\Sigma \cup N)* \longrightarrow (\Sigma \cup N)*$ [1]. That is, each rule maps one string of symbols to another. Or in other words, a string to the left can be replaced by a string to the right. This replacement is called **derivation** (denoted as $\Rightarrow$).

Non-terminal symbols can be understood as references to another production rule, while **terminal** symbols are those that form the final valid string of a language.

The **valid string of a language** is one that consists exclusively of terminal symbols and can be derived from S in any number of steps. The set of all valid strings forms the

language **generated** by grammar G, denoted as $L(G) = \{w \in \Sigma^* | S \Rightarrow {}^*w\}$(i.e. language generated by grammar G is all strings that can be derived from G‘s start symbol).
The only constraint put on production rules is that the left side must contain at least one non-terminal symbol.

Context-free grammar is a special type of grammar in the Chomsky hierarchy. It is a grammar in which the left-hand side of each production rule consists only of a single non-terminal symbol. So the production rule form is simplified to $N \longrightarrow (\Sigma \cup N)*$. Further, only context-free grammars will be considered.

2.4.1 Example of a grammar

Consider the grammar G , where $N = \{S, B\}, \Sigma = \{a, b, c\}$, S is the start symbol, P consists of the following rules:

$S \longrightarrow aBSc$
$S \longrightarrow abc$
$B \longrightarrow cB$
$B \longrightarrow bb$

Examples of derivations of strings in *L(G)*:

- $S \Rightarrow abs$
- $S \Rightarrow aBSc \Rightarrow abbSc \Rightarrow abbabcc$
- $S \Rightarrow aBSc \Rightarrow aBaBScc \Rightarrow acBaBScc \Rightarrow acbbaBScc \Rightarrow acbbaBabcccacbbabbabccc$
  ...

2.5 BNF and EBNF

Backus–Naur form (**BNF**) is a metasyntax notation for context-free grammars proposed by John Backus, a programming language designer at IBM [7]. It is used to express the syntax of formal languages - typically computer programming languages. Let’s start by rewriting the grammar defined as an example [2.4.1] with in BNF:

$< S >$::= *“a”* $< B >< S >$ *“c”*|*“abc”*
$< B >$::= *“c”* $< B >$ |*“bb”*

Sequences of characters enclosed in the brackets <>represent metalinguistic variables whose values are sequences of symbols. In other words <...>represents a nonterminal symbol. Characters enclosed in commas are just terminal symbols. The "::=" symbol means "can be replaced with", the same meaning as "⟶" in the Chomsky formalization. The vertical bar "|" can be read as "or" and denotes the choice between two sequences of symbols. There are many variants and extensions of BNF. The most famous is probably the EBNF [7]. An extended Backus–Naur form (EBNF) consists of terminal symbols and non-terminal production rules which are the restrictions governing how terminal symbols can be combined into a legal sequence. The common features of the EBNF [8] are listed below:

Notation	**Usages**
=	definition
,	concatenation
;	termination
|	alternation
[...]	optional
. . .	repetition
(. . .)	grouping
"..." or '...'	terminal string
? . . . ?	special symbol
-	exception

Table 2.1: EBNF syntax features

These metasyntax notations and their further extensions are used by parser generators (compiler generators) that will be discussed in further chapters.

2.5.1 Example of EBNF usage

Firstly the example [2.4.1] written in EBNF will look as follows:

$$S = \text{“}a\text{”}, B, S, \text{“}c\text{”} \,|\, \text{“}abc\text{”}$$
$$B = \text{“}c\text{”}, B | \text{“}bb\text{”}$$

This example doesn't fully show all of EBNF features, so the a more complex illustration is needed.

EBNF is metasyntax for describing computer languages. Consider the definition of a Pascal-like programming language that allows only assignments written in EBNF:

```
program = 'PROGRAM', white_space, identifier, white_space,
           'BEGIN', white_space,
           { assignment, ";", white_space },
           'END.' ;
identifier = alphabetic_character, { alphabetic_character | digit } ;
number = [ "-" ], digit, { digit } ;
string = '"' , { all_characters - '"' }, '"' ;
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic_character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white_space = ? white_space characters ? ;
all_characters = ? all visible characters ? ;
```

Figure 2.1: Pascal-like language grammar in EBNF

Then the syntactically correct program of defined language could be:

```
PROGRAM DEMO1
BEGIN
  A:=3;
  B:=45;
  H:=-100023;
  C:=A;
  D123:=B34A;
  BABOON:=GIRAFFE;
  TEXT:="Hello world!";
END.
```

Figure 2.2: Example of valid program in Pascal-like language

Chapter 3

Domain Specific Languages

3.1 Introduction to DSL

In theory of computer languages, a domain-specific language (DSL) is a language meant for use in the context of a particular domain [4]. This is in contrast to a general-purpose language (e.g. Java, C++), which is broadly applicable across domains.

Domain is a term used in software engineering which means the targeted subject area of a computer program. The domain could be such familiar things like database querying or document markup. Additionally it could be any specific business context such banking, insurance, automotive production or smart home systems.

3.2 DSL examples

Query languages:

- **SQL**
- **XPath**

Data storage languages:

- **XML**
- **YAML**

Document formatting languages:

- **LaTex**
- **HTML**

- **CSS**

Languages that cover more specific domains:

- **DOT**.
  A DSL to define graphs

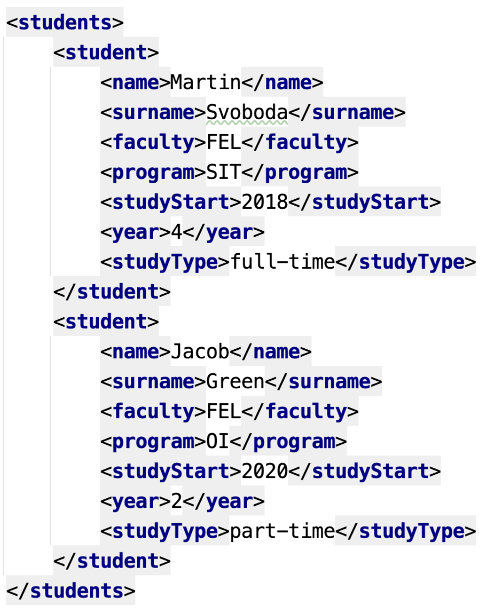
- **PlantUML**.
  A DSL to draw UML diagrams

It is possible to list many examples of DSL, but many of them are developed and used by a small group of people; for example, a language that exists within a single organization.

Consider a scenario in which a car manufacturer develops its own language for expressing car configurations. The syntax and features of this DSL meet the requirements of this particular company only. Such a language is unlikely to go beyond its limits, and most likely will not be suitable for other users due to the domain specific design for a particular company.

3.3 Need for a new language

The main goal of this theses is the creation of a plugin for DSL developers. One may wonder why people need to develop new languages to describe specific data and models at all. There are known tools to describe data in both a machine and human-readable form, the most popular of which is XML. An example can be some a university that needs to store information about their students. The XML file may look as follows:



```
<students>
    <student>
        <name>Martin</name>
        <surname>Svoboda</surname>
        <faculty>FEL</faculty>
        <program>SIT</program>
        <studyStart>2018</studyStart>
        <year>4</year>
        <studyType>full-time</studyType>
    </student>
    <student>
        <name>Jacob</name>
        <surname>Green</surname>
        <faculty>FEL</faculty>
        <program>OI</program>
        <studyStart>2020</studyStart>
        <year>2</year>
        <studyType>part-time</studyType>
    </student>
</students>
```

Figure 3.1: Students data stored in XML

The XML representation fills the document with too much additional syntax noise due to all of the tags. It is not straightforward for a human to grasp the actual information about a student from such a specification.
The same information can be described in a custom DSL:

```
Martin Svoboda [FEL SIT] 4(2018) full-time,
Jacob Green [FEL OI] 2(2020) part-time
```

Figure 3.2: Students data stored in DSL

This specification is more compact and contains much less noise. Also the editor of the DSL can provide many useful features such as a rich validation system that will not allow the user to write invalid information. For example the editor will help to complete the keyword `part-time` as well as faculty/program names. It may also forbid improper year/study start date values.

Chapter 4

Implementing a DSL

4.1 Introduction

Implementing a DSL means developing a program that is able to read text written in that DSL, parse it, process it, and then possibly interpret it or generate code in another language [13]. Also, to ensure that the end user can effectively and comfortably work with a new language, the implementation process should include the building of a rich environment that provides features modern developers are used to, such as: validation, highlighting etc.

4.2 Parsing

First and foremost, while reading a program written in a specific DSL, the implementation must ensure that the program follows the language's syntax.

To accomplish this, the program must be divided into tokens. Each token is a single atomic element of the language. An analogy between program tokens and terminal symbols [2.2] can be drawn here. The following are possible program tokens:

1. **Keywords**
   Such as `class` or `private` in Java.

2. **Identifiers**
   Such as Java class (or class member) name.

3. **Literals**
   Examples of literals can be string literals, typically surrounded by quotes (*"Hello world"*), integer literals (`204`) or boolean literals (`true` or `false`).

4. **Separators**

   Such as parentheses or terminating semicolons.

5. **Spaces**

   Language elements such as whitespaces or newlines characters.

For instance, in the figure 3.2 `Martin` is an example of a string literal, `2018` is an integer literal, square and round brackets are separators and **`full`** is a keyword.

The process of converting a sequence of characters into a sequence of tokens is called **lexical analysis**, and the program or procedure that performs such analysis is called a **lexer** [9]. This analysis is usually implemented by using the syntax of regular expressions.

Having a sequence of tokens from the input file is not enough, since this sequence should form a valid statement in the language. This phase is called parsing or **syntactic analysis**. The program or procedure that performs such analysis is called a **parser**. In fact, the parser asks the lexer for tokens and tries to build a valid statement of the language [2.4].

Writing the parser and the lexer by hand requires a lot of effort. In practice, special tools are utilized to do this work for a human. These tools are called **parser generators**.

A **parser generator** is a programming tool that creates a parser, interpreter, or compiler from some form of formal description of a programming language. Its input is a grammar file, typically written in extended Backus–Naur form (EBNF) [2.5.1] and its output is the source code of a parser for the programming language. In the Java world, the most well-known parser generator is ANTLR (ANother Tool for Language Recognition) [6].

4.3 Semantic analysis

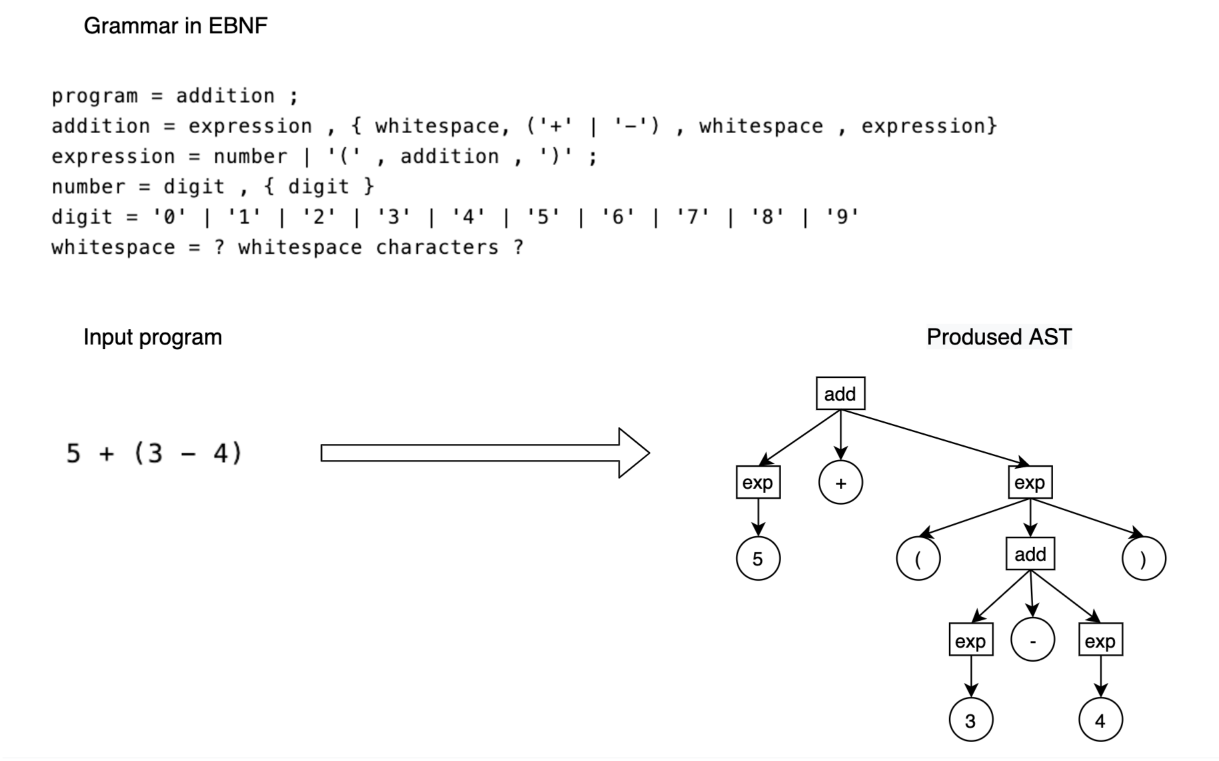
Parsing a program is only the first stage in implementing a programming language. Usually it is not possible to check the overall validity of a program by only performing syntactic analysis. For example **type checking** cannot be performed during parsing (e.g. assigning a string value to an integer variable is not possible in Java). Validations like this are part of **semantic analysis** and are handled by a different section of the implementation program.

For these reasons, a representation of the parsed program should be constructed and stored in memory. In this way the implementation program can perform semantic analysis on the memory representation instead of parsing the same text repeatedly.

4.3.1 Abstract Syntax Tree

A tree structure called the **Abstract Syntax Tree** (**AST**) is a convenient representation of a program in memory [10].

The **Abstract Syntax Tree** may also be thought of as a representation of the abstract syntactic structure of source code written in a programming language. The following example of a parsed program in a simple “Addition expressions language” will review the previous topics and illustrate the principle of AST construction:



Grammar in EBNF

```
program = addition ;
addition = expression , { whitespace, ('+' | '-') , whitespace , expression}
expression = number | '(' , addition , ')' ;
number = digit , { digit }
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
whitespace = ? whitespace characters ?
```

Input program

```
5 + (3 - 4)
```

Figure 4.1: Example of AST built from input program of particular language

The tree diagram shows that its leaf elements are tokens in the context of **syntactic analysis** (terminals in the context of grammar). Inner nodes are associated with the languages’s grammar non-terminal symbols.

Since the AST nodes may differ in types and properties, different code is required to represent them. Therefore, the language implementation should include Java classes for each type of nodes. Often parser generators are responsible for generating code for AST nodes. In this way, using ANTLR, a programmer should specify names of Java classes directly in the grammar file to map elements of the language to tree nodes types that will be built in the future [3].

4.4 IDE integration

Even if the DSL implementation is able to read, parse and semantically validate the programs written in a new language, the work cannot really be considered finished. Nowadays, many programmers are accustomed to using powerful IDEs, which assist programmers to use learn and maintain programming languages.
Every IDE should provide the following features:

- **Syntax highlighting**
  This provides users with immediate feedback on the syntactic validity of what they are writing by coloring and formatting various DSL elements in various styles.

- **Error markers**
  If the development environment is able to mark incorrect parts of the program and provide explanatory messages directly in the editor, the programmer does not have to go to the console to discover such errors and will easily spot the parts of the program that need to be fixed.

- **Completion assist**
  This feature automatically, or on demand, provides suggestions on how to complete a given statement that the programmer just typed. The feature not only speeds up the programming process but also tells the user what will make sense in that specific program context, allowing them to avoid consulting the documentation or inspecting the code.

- **Hyperlinking**
  Hyperlinking is a feature that makes it possible to navigate between references in a program. If the DSL provides declarations of any sort, then the IDE should also provide the possibility to directly jump from an identifier token to the corresponding declaration.

- **Quickfixes**
  The ability of IDE to fix the mistake made by user if the DSL implementation is able to do so. When a programmer gets stuck, this feature could be quite handy.

Chapter 5

Xtext plugin in Eclipse

5.1 Introduction

Xtext is an Eclipse framework for implementing programming languages and DSLs [14]. It lets a developer implement languages quickly, and most of all, it covers all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a complete Eclipse IDE integration with all the typical IDE features that were discussed previously [4.4].

Xtext generates the complete language infrastructure in one stage. The only thing the framework requires from the developer is the grammar specification file written in powerful language developed by the Xtext team.

5.2 Xtext grammar language

The language used in the Xtext framework to specify DSL grammars is appropriately named "Xtext." Because the Xtext language is an extension of EBNF, the reader will be familiar with its syntax. However, it has a wide range of features that allow it to express all parts of language infrastructure in one file using friendly syntax.

5.2.1 Basic concepts

1. **Interconnection between language structures and Ecore model objects**
   The need of AST(in-memory representation) of a parsed program was discussed in the previous chapter. In the Xtext world, the notion of AST is often referred to as the Ecore model because its nodes are objects of some Ecore model. Here a brief introduction to the EMF framework is needed.

EMF (Eclipse Modeling Framework) is a framework in Eclipse for modeling, generating code and presenting object models [12]. The main elements of EMF are:

- **Ecore model** is model built with EMF, the name comes from Ecore language in which models are declared.
- **EClass** is a Java class of a the Ecore model.
- **EObject** is an instance of an EClass.
- **EDataType** is an EMF representation of a Java primitive type.
- **EPackage** is a set of EClasses.

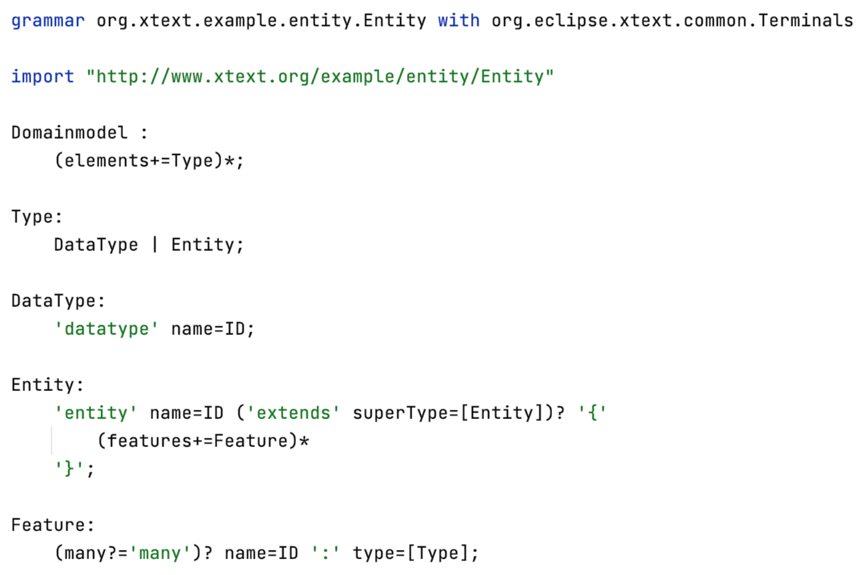
The parser generated by Xtext is responsible for building such object graphs. Therefore, the developer should specify how certain parts of the language will be mapped to certain EObjects (quite similar to the ANTLR approach because the Xtext framework uses it under the hood). The user has the possibility to map not only the whole rule to the EObject but parts of the rule to EObject's properties called **features**.

2. **References system**
   A unique feature of the Xtext grammar language is called **Cross References**. This allows the user to specify how one named part of the DSL can be referenced from another in the grammar specification file.

3. **Reuse of grammars**
   The DSL developer is able to declare the existing Xtext grammar for reuse(referenced as **grammar mixin**). In this way, a developer can make use of the declared grammar's rules in the language that he or she develops.

5.2.2 Syntax description

First of all, the example of a grammar language syntax should be shown. This is the grammar file defining the DSL called “Entity” that the developers of Xtext provide in their official framework documentation[3]:



```
grammar org.xtext.example.entity.Entity with org.eclipse.xtext.common.Terminals

import "http://www.xtext.org/example/entity/Entity"

Domainmodel :
    (elements+=Type)*;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=[Type];
```

Figure 5.1: Entity language grammar defined by Xtext grammar language

Now let’s discuss the syntax and features based on this example.
The whole grammar file consists of two parts: the header and the set of grammar rules.
The header includes:

1. **Language Declaration**
   `grammar` `org.xtext.example.entity.Entity`
   `with` `org.eclipse.xtext.common.Terminals`
   This line declares the fully qualified name (**FQN**) of the language.
   The FQN that follows the keyword **`with`** indicates the **grammar mixin**.

2. **EPackage Declarations**
   `import` `"http://www.xtext.org/example/entity/Entity"`
   This line defines the Ecore model, which enables the use of their EObjects in the grammar. The imported model’s EObjects are then utilized to create an AST of DSL expressions.

There are two types of rules in the Xtext grammar language: **Parser rules** and **Terminal rules**.

5.2.2.1 Parser rules

A parser rule is an Xtext interpretation of a **non-terminal production rule** [2.4]. The syntax of each parser rule is quite similar to EBNF:

```
Domainmodel :(elements+=Type)*;
```

Here `Domainmodel` is the rule‘s name, colon (`:`) is a definition sign. The semicolon (`;`) is used to separate rules. Xtext syntax uses four types of cardinalities similar to regular expressions operators: exactly one (the default, no operator), zero or one (operator `?`), zero or more (operator `*`) and one or more (operator `+`).
The first parser rule defined in the grammar file is considered as the **root rule** (an analogy of **start symbol** [2.4])

The parser rule features:

- **Return types**

  ```
  RuleA returns TypeA: ...  ;
  ```

  The developer specifies the type of EObject that will represent the DSL expression matched by the parser rule in the AST using the rule’s return type. If the return type is omitted, the EClass of the same name as the parser rule is considered. In this way the ”Domainmodel” rule can be rewritten as:

  ```
  Domainmodel returns Domainmodel:(elements+=Type)*;
  ```

- **Assignments**

  ```
  elements+=Type
  ```

  Assignments are used to assign parts of the rule to a **feature** (the Xtext notation of EObjets‘s property) of the returned object. The type of the assigned feature is inferred from the right hand side of the assignment.

  There are three different assignment operators, each with different semantics.

  1. The simple equal sign = is the straightforward assignment, and is used for features which take only one element.
  2. The += sign (the add operator) expects a multi-valued feature and adds the value on the right hand side to the list feature.
  3. The ?= sign (boolean assignment operator) expects a feature of type EBoolean and sets it to true if the right hand side was parsed.

For instance in the rule

`DataType:‘datatype’ name=ID;`

the EObject of type DataType will have a feature called “name” and the object returned by the rule ”ID” will be assigned to it.

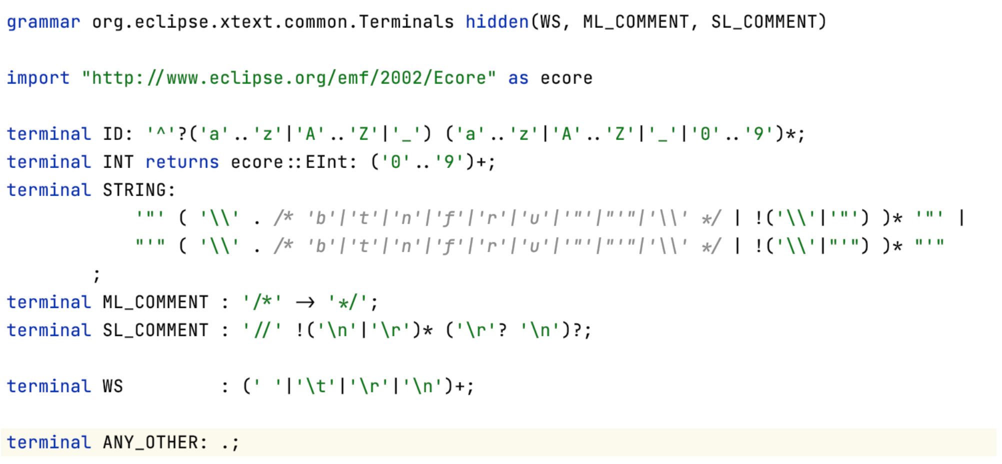
- **Cross-References**

  `superType=[Entity]`

  Cross-Reference denotes reference by name and the parser will expect the ID token here, not the Entity rule. This ID should be the name of an existing Entity. The type of supertype feature will be the EClass that the Entity rule returns.

5.2.2.2 Terminal rules

There are no terminal rules presented in the Entity language grammar (figure 5.1) so one should take a look at the grammar it reuses. `org.eclipse.xtext.common.Terminals` grammar is predefined grammar in Xtext that consists exclusively of terminal rules:



```
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID: '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING:
            '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"' |
            "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|"'") )* "'"
        ;
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS         : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER: .;
```

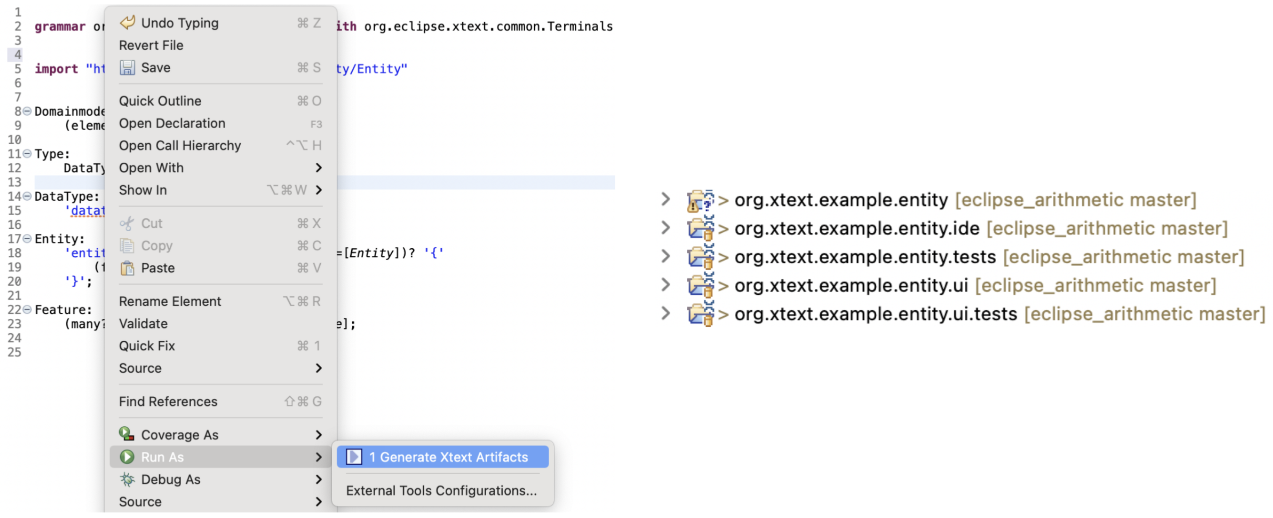
Figure 5.2: Terminals grammar file

Terminal rules are also known as token rules or lexer rules since they represent atomic elements of the language. Every terminal rule starts with the keyword **`terminal`**. The return type of the terminal rule should be **EDataType**. If the return type is omitted, the type **EString** is considered.

The features of the language do not end there. Unmentioned things such as tools for more detailed designing of AST called **Actions** will be discussed in future implementation chapters [9].

5.3 How it works

When the user finishes with the grammar, the next step is to run an action on a grammar description file. This action generates the parser, text editor, and the other additional infrastructure code. Simply put, it will generate all the things you need to start using the new DSL.



```
grammar o                ith org.eclipse.xtext.common.Terminals

import "ht               ty/Entity"

Domainmode
    (eleme

Type:
    DataTy

DataType:
    'datat

Entity:
    'entit
        (                =[Entity])? '{'
    '}';

Feature:
    (many                e];
```

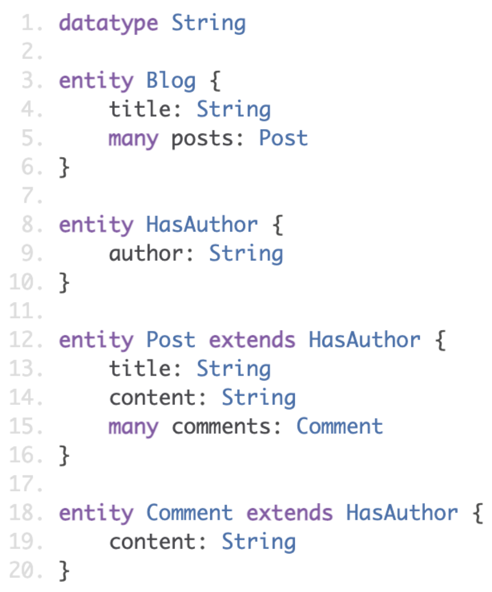
```
> org.xtext.example.entity [eclipse_arithmetic master]
> org.xtext.example.entity.ide [eclipse_arithmetic master]
> org.xtext.example.entity.tests [eclipse_arithmetic master]
> org.xtext.example.entity.ui [eclipse_arithmetic master]
> org.xtext.example.entity.ui.tests [eclipse_arithmetic master]
```

Figure 5.3: The output Xtext framework generates

After the Xtext finishes its work, the user will see 5 Xtext projects were generated. The developer can now launch a new instance of Eclipse, which will include the new language plugins and provide functionality for code completion, syntax highlighting, syntactic validation, linking errors, formatting, hyperlinking, find references, folding, rename refactoring, etc.

Furthermore, the programmers can easily enrich the generated code with their own logic. An example might be configuring editor-support files(e.g to add one's own validation rules) or integrating the DSL with java.

Ones the new instance of Eclipse is launched the IDE will provide full infrastructure for the new language. Thus, the following code can be typed and successfully parsed by the IDE:



```
datatype String

entity Blog {
    title: String
    many posts: Post
}

entity HasAuthor {
    author: String
}

entity Post extends HasAuthor {
    title: String
    content: String
    many comments: Comment
}

entity Comment extends HasAuthor {
    content: String
}
```

Figure 5.4: The example of program in Entity language

Chapter 6

IntelliJ IDEA approach

6.1 Introduction

IntelliJ IDEA is an IDE built on the IntelliJ Platform, which is a powerful platform for building development tools targeting any language [15]. Similarly to Eclipse IDE the implementation of a new language means creating a plugin which will contain all of the needed components described in the chapter 4, starting from parser and ending with various editor support features.

In contrast to Eclipse, there is no such powerful framework in the IntelliJ IDEA that could generate an entire programming language infrastructure in one click. The DSL developer is forced to spend more time and effort to implement the language support plugin in IDEA. Fortunately, however, this does not mean the developer must code everything from scratch. The IntelliJ platform provides a rich language-independent code base that allows developers to implement a particular IDE feature by just implementing a few interfaces. Moreover, the programmer is not obligated to write the parser and lexer code as far as it can take full advantage of the Grammar-Kit plugin.

6.2 Grammar-Kit

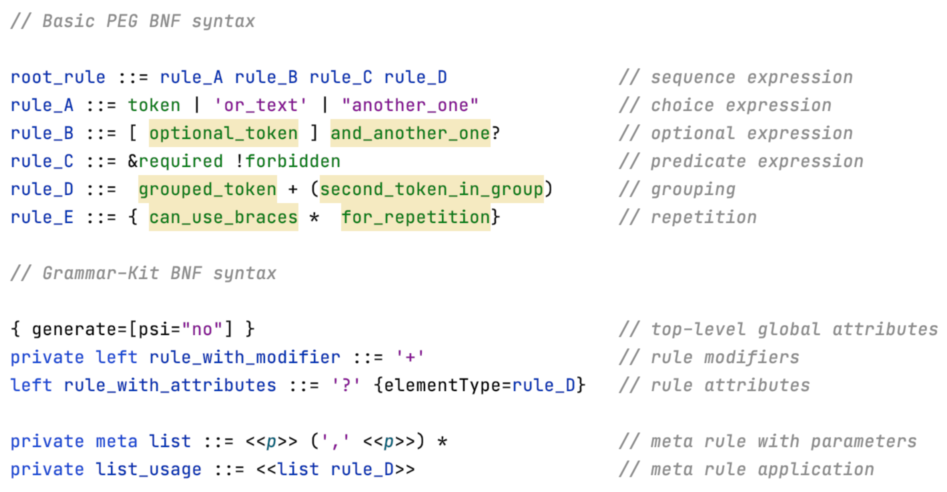
Grammar-Kit is an IntelliJ IDEA plugin for creating language support plugins [18]. It adds parser, lexer and PSI (an extension over AST in IntelliJ platform, described in more detail below) generators to IDEA.

With the help of Grammar-Kit the programmer can describe the language‘s grammar using BNF-like specification [2.5.1]. However, the plugin doesn't allow the programmer to declare everything in one file like Xtext in Eclipse does. In order to define **tokens** (terminal symbols) the developer should declare them in the **JFlex** (lexer generator) specification file, using special syntax [22]. Thus Grammar-Kit plugin adds BNF and JFlex file editing support.

The plugin works with IntelliJ **PSI**. The Program Structure Interface(PSI), is the layer in the IntelliJ Platform responsible for parsing files and creating the syntactic and semantic code model that powers many of the platform's features [19]. Therefore, the parser created by Grammar-Kit builds a PSI tree whose nodes are subclasses of `PsiElement` (the common base interface for all of the elements of the PSI tree).

6.2.1 Grammar-Kit BNF

Grammar-Kit uses PEG (parsing expression grammar) with EBNF notations.
The syntax of grammar language looks as follows:



```
// Basic PEG BNF syntax

root_rule ::= rule_A rule_B rule_C rule_D               // sequence expression
rule_A ::= token | 'or_text' | "another_one"            // choice expression
rule_B ::= [ optional_token ] and_another_one?          // optional expression
rule_C ::= &required !forbidden                         // predicate expression
rule_D ::=  grouped_token + (second_token_in_group)     // grouping
rule_E ::= { can_use_braces *  for_repetition}          // repetition

// Grammar-Kit BNF syntax

{ generate=[psi="no"] }                                 // top-level global attributes
private left rule_with_modifier ::= '+'                 // rule modifiers
left rule_with_attributes ::= '?' {elementType=rule_D}  // rule attributes

private meta list ::= <<p>> (',' <<p>>) *               // meta rule with parameters
private list_usage ::= <<list rule_D>>                  // meta rule application
```

Figure 6.1: Grammar-Kit BNF syntax

In other words, Grammar-Kit BNF uses basic EBNF syntax, but it is extended with **global attributes**, **rule attributes** and **rule modifiers**.

Global attributes are information for the code generator. There could be specified paths where the parser should be generated, the prefix of generated classes, or paths to special "Util" classes. Rule attributes contain additional information for the code generator, but separately for each rule. It could be a prefix attribute such as `private`, `left`, `inner`, etc. Rule attributes (enclosed by curly brackets) could be used to specify the interface for the PSI class to be generated. It could also be used to change the parser logic for a particular rule (`pin` or `recoverWhile` attributes).

6.2.2 Entity DSL example

The Entity language, which was used as an example in the previous chapter [5.2.2], can be implemented with Grammar-Kit plugin. Its syntax description written in Grammar-Kit BNF will look like this:

```
Domainmodel ::= (Type)*

Type ::= DataType | Entity

DataType ::= 'datatype' ID

Entity ::= 'entity' ID ('extends' ID)? '{' (Feature)* '}'

Feature ::= ('many')? ID ':' ID
```

Figure 6.2: Entity DSL grammar defined using Grammar-Kit BNF

Tokens are described using a separate file (hereafter called Flex), which is specification for the **JFlex generator**. Terminal Rules from "Terminals" grammar (figure 5.2) have the following form in Flex:

```
ID =\^?(([a-z]|[A-Z]|_))(([a-z]|[A-Z]|_|[0-9]))*
INT =([0-9])+
STRING =(\"((\\.|[^\\\"]))*\"|'((\\.|[^\\']))*')
ML_COMMENT =\/\*([^"*"]*("*"+[^"*""/"])?)*("*"+"/")?\*\/
SL_COMMENT =\/\/([^\n\r])*(\r?\n)?
WS =((" "|\t|\r|\n))+≤
ANY_OTHER =.
```

Figure 6.3: Token elements definition using Flex

Specification like this is only used as input for the Grammar-Kit parser generator and doesn't describe IDE behaviour or add editor features. To do so in IntelliJ IDEA, the developer should perform a number of actions that will be discussed in detail in a later chapter [9.2].

Chapter 7

Summary and analysis

This intermediate chapter aims to review the thesis goals and how they relate to the covered theoretical material. It analyzes the differences between the procedures discussed earlier which will be important in the implementation stage.

- The development of programming languages is the main topic of the work.
- The DSL is a programming language to describe things and make expressions which belong to a specific domain.
- The DSL developer's job consists of the following parts:
  1. Inventing of the language itself, defining its syntax and rules of how sentences should be formed. The developer achieves this by describing the language's grammar using some variation of BNF metasyntax.
  2. Creation of the environment that will support the usage of the DSL. Usually this environment is built on the base of some existing IDE - in this case, a plugin that adds DSL-specific components(parsers, editor support logic) to that IDE.
- The developer is not required to write complex IDE components (such as the parser or lexer) since tools called parser generators are used for that purpose.
- There is an "Xtext" framework in the Eclipse IDE, that is much more powerful than a typical parser generator. The main advantages of Xtext are:
  1. Xtext generates the entire ready-to-use environment in one click, requiring only one grammar specification file as its input.
  2. The Xtext framework uses its own BNF for describing a DSL's grammar. This grammar language allows developers to flexibly design a mapping between grammar structures and domain instances. It also allows the programmers

to define references by name and token elements and handle the AST creation process in one file.

3. The Xtex framework relies on an object modeling framework EMF that is ideal solution for describing and working with specific domains.

- The main objective of this bachelor thesis is to "transfer" the Xtext features from the Eclipse IDE to the IntelliJ IDEA IDE. In other words, this project encompassed the creation of the IDEA plugin that extends all of the Xtext functionality to the IntelliJ platform.

- This "transferring" is a quite complex task since Eclipse and IDEA are completely different platforms with various approaches to the creation of language support plugins.

- Two major differences between these two IDEs are:

  1. **AST creation**.
     In the IntelliJ IDEA the PSI layer is responsible for parsing files and creating the semantic model. In the case of a successfully parsed program(in any language) the PSI tree is created.

     In Eclipse the process of AST creation is hidden from user. Using Xtext, the DSL developers receives a semantic model constructed from EObjects that they specified in the grammar description.

     The approach with the tree structure common for all language formats is definitely more convenient, but due to the fact that Xtext in Eclipse uses Ecore models throughout its code and predefined grammars, the implementation of Xtext in IntelliJ IDEA should be able to build an EObjects tree out of a PSI tree. That also will allow Xtext users who were working with ECore models in their projects to transfer to IDEA with ease.

  2. **Parser generators**.
     The syntax of the Xtext grammar language and Grammar-Kit BNF was described in previous chapters. Although Grammar-Kit provides many possibilities to fine-tune parser and PSI tree creation, some of the features that Xtext provides have no direct analogues in the Grammar-Kit world.

- The implementation can be considered successful if: for the same input (grammar specification in Xtext format) the implemented plugin will generate an environment similar to one that the Xtext framework in Eclipse generates (e.g. the semantic models for the same DSL input should be equal).

Chapter 8

Design plan

The implementation of the project is divided into several stages:

1. **Implementation of the Xtext grammar language support plugin in IDEA**
   In the final plugin users should be able to describe grammars in the Xtext grammar language. To achieve this the IDEA plugin that adds support for this language will be written manually.

2. **Implementation of the mechanism to "translate" Xtext grammars to Grammar-Kit grammars**
   The method of generating a Grammar-Kit BNF file and a Flex file for a given Xtext grammar should be devised. At this stage, only language syntax support is considered because Xtext grammar features like Cross-References cannot be defined in the Grammar-Kit BNF.

3. **Implementation of the language-defining files generator**
   The DSL developer is required to write a significant number of classes which are part of the the language support plugin in IntelliJ platform. These classes should be generated together with the Grammar-Kit BNF/Flex files, thus the base IDEA plugin supporting the particular DSL could be generated from the input Xtext grammar specification file. There are still no Xtext features or editor support that will be implemented at this stage.

4. **Implementation of the editor support files generator**
   According to [4.4] the generated IDE should provide base support to help programmers in their work, such as syntax highlighting or keyword completion. The generator of classes which realize this support is implemented in this stage.

5. **Regeneration of the Xtext language support plugin created in point 1.**
   Using generators created in points 2-4, the plugin supporting Xtext grammar lan-

guage will be generated. This is a "bootstrapping" stage in the sense that the project component will be built by this project itself.

6. **Implementation of "Bridge" between PSI tree and EObjects tree**
   The "Bridge" is the name chosen during the implementation process for the project component which is responsible for the creation of semantic models identical to those created in Eclipse.
   This is the most complex and time-consuming stage. Therefore it should be divided into substages:
   a) "Bridge" algorithm implementation for one test language.
   b) Generalisation of the algorithm for every language.
   c) Implementation of the "Bridge" generator.
7. **Implementation of validation rules**
   Adding custom validation rules is the most common practice in DSL development after the base language implementation is completed. The Xtext framework offers a user-friendly API for these purposes. This API will be reproduced in the IntelliJ IDEA at this stage.
8. **Implementation of a new project wizard**
   The new project wizard is a tool that assists users in creating new projects with specific structures by requesting them to provide some initial inputs.

The next chapter covers the entire implementation process. The material will be narrated in the order corresponding to the design plan.

Chapter 9

Implementation

9.1 Technologies used

Programming languages:

- **Java**
  Java is the central language of the thesis topic. All of the IntelliJ platform classes are implemented (and generated) in it.

- **Kotlin**
  Most of the code is written in Kotlin [20]. It is fully interpretable with Java and provides useful features that help to implement classes containing difficult logic, generators and IntelliJ UI classes.

- **Xtend**
  This Java dialect is used to configure Xtext plugins in Eclipse [21]. In this project it was used at the "Bridge" testing stage for the implementation of the unit test methods in Eclipse.

Products and technologies:

- **IntelliJ IDEA Community IDE**
  Target platform of the created plugin - final product of this project.

- **Gradle**
  Assembling tool used in the IntelliJ platform for the creation of language support plugins [16].

- **Grammar-Kit**
  Plugin in IDEA which provides parser generator functionality.

- **PsiViewer**
  Plugin in IDEA for representation of PSI trees.
- **Eclipse IDE**
  The IDE which was used to learn, analyze and test the Xtext framework. Ecore models are also built and exported from it.
- **Xtext framework**
  The framework this project is aimed to replicate.
- **EMF framework**
  Eclipse framework which provides tools for creating Ecore models.
- **Enterprise Architect**
  This product from the Sparx Systems company [17] was used for design purposes during the implementation process. Moreover, all of the diagrams presented in the thesis were drawn using Enterprise Architect.

9.2 Xtext language support plugin implementation

The first thing the implementation should start with is the creation of the plugin supporting Xtext grammar language. After the successful implementation of this stage, IDEA will be expanded with the possibility to parse Xtext grammars and build PSI trees of parsed files. These trees then will be used as the main input parameter by other project components(e.g. generators).

The process of implementing plugins of this type can be logically divided into two phases:

1. Defining the language grammar and generating the parser/lexer/PSI classes (using Grammar-Kit plugin)

2. Writing classes that integrate the new language to the IntelliJ IDEA and add editor support.

9.2.1 Defining language grammar

The Xtext grammar language is described by itself. At this point the task is to translate the grammar from the Xtext BNF to the Grammar-Kit BNF.

Some grammar elements will remain unchanged, whereas others will be substantially altered or ignored at all. The following table represents Xtext grammar features of **Parser rules** that should be modified in some way while being rewritten in the Grammar-Kit BNF:

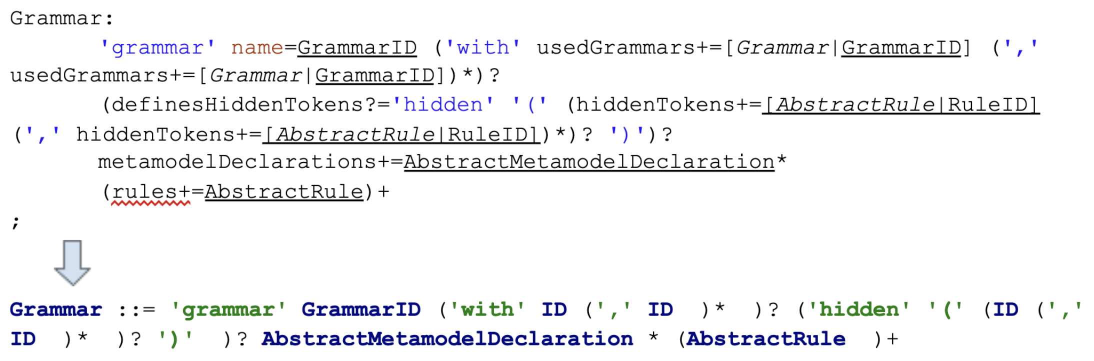
Xtext grammar feature	Xtext notation	Grammar-Kit BNF solution
Rule return type	`Alternatives returns AbstractElement: ...`	`Alternatives ::= ...`
Assignments	`... name=GrammarID ...`	`... GrammarID ...`
Cross-References	`... supertype=[Entity|STRING] ...`	`... STRING ...`
	`... classifier=[EClassifier] ...`	`... ID ...`
Unordered groups	`... 'static' & 'final' & Visiability ...`	-
Simple Actions and **Assigned Actions**	`... {Group} ...`	-
	`... {Alternatives.elements+=current} ...`	-
Syntactic Predicates	`... -> 'else' ...`	-

Table 9.1: Grammar-Kit replacements of Xtext grammar features

Dashes in the third column of the table mean that the corresponding feature is ignored at this stage of implementation. Some of skipped features like **Actions** or **Syntactic**

Predicates will be implemented in future "Bridge" implementation stage [9.4], when building EObject models and AST modification operations will make more sense. However, the **Unordered groups** feature will not be implemented in this project because Grammar-Kit BNF doesn't offer a mechanism to express such language structures.

Thus, the root rule of Xtext grammar language is translated to Grammar-Kit BNF as shown in the figure:

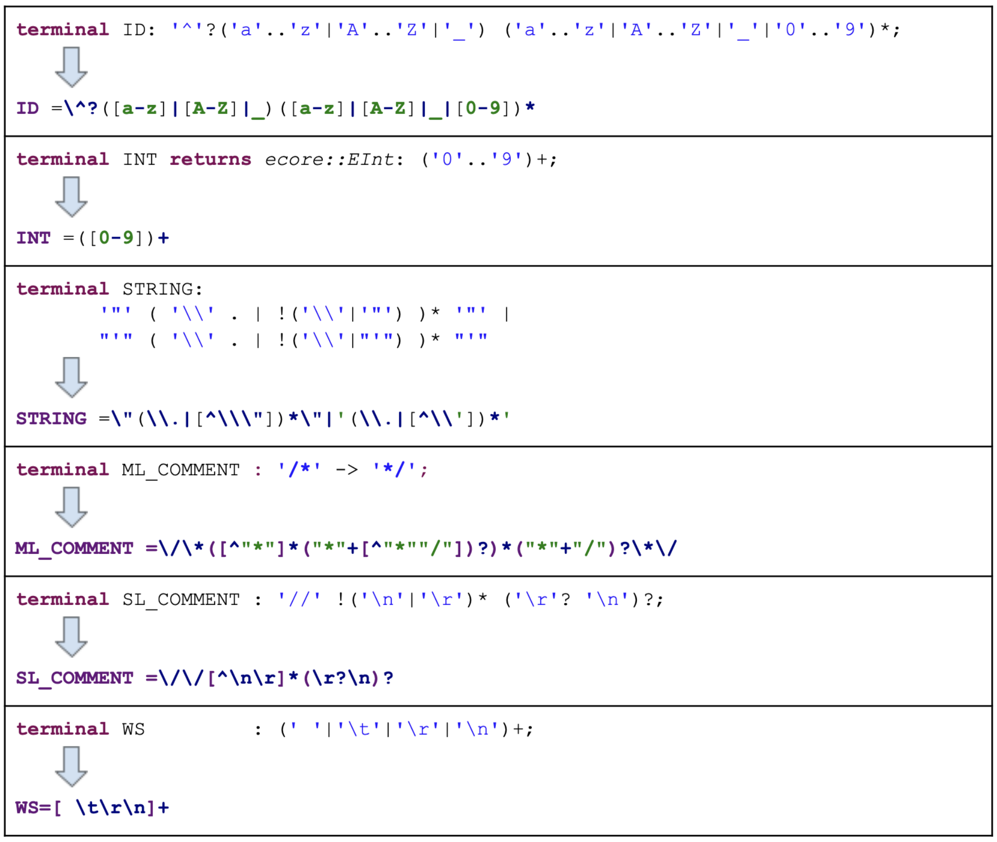


```
Grammar:
      'grammar' name=GrammarID ('with' usedGrammars+=[Grammar|GrammarID] (','
usedGrammars+=[Grammar|GrammarID])*)?
      (definesHiddenTokens?='hidden' '(' (hiddenTokens+=[AbstractRule|RuleID]
(',' hiddenTokens+=[AbstractRule|RuleID])*)? ')')?
      metamodelDeclarations+=AbstractMetamodelDeclaration*
      (rules+=AbstractRule)+
;
```

```
Grammar ::= 'grammar' GrammarID ('with' ID (',' ID  )*  )? ('hidden' '(' (ID (','
ID  )*  )? ')'  )? AbstractMetamodelDeclaration * (AbstractRule  )+
```

Figure 9.1: Xtext grammar language root rule translated to Grammar-Kit BNF

In a similar way Xtext **Terminal rules** should be translated. Since Grammar-Kit uses JFlex generator for Lexer generation, all terminals should be defined in a Flex file that has a fairly intuitive syntax similar to regular expressions. The following figure displays how Terminals rules are translated to the `.flex` format:



```
terminal ID: '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

ID =\^?([a-z]|[A-Z]|_)([a-z]|[A-Z]|_|[0-9])*
```

```
terminal INT returns ecore::EInt: ('0'..'9')+;

INT =([0-9])+
```

```
terminal STRING:
       '"' ( '\\' . | !('\\'|'"') )* '"' |
       "'" ( '\\' . | !('\\'|"'") )* "'"

STRING =\"(\\.|[^\\\"])*\"|'(\\.|[^\\'])*'
```

```
terminal ML_COMMENT : '/*' -> '*/';

ML_COMMENT =\/\*([^"*"]*("*"+[^"*""/"])?)*("*"+"/")?\*\/
```

```
terminal SL_COMMENT : '//' !('\n'|'\r')* ('\r'? '\n')?;

SL_COMMENT =\/\/[^\n\r]*(\r?\n)?
```

```
terminal WS          : (' '|'\t'|'\r'|'\n')+;

WS=[ \t\r\n]+
```

Figure 9.2: Xtext Terminal Rules translated to Flex format

9.2.2 Base plugin architecture

The “base version” of plugin is created at first. This simplified version will only contain the code mandatory for base language integration without optional editor support logic.

The technique for creating such types of plugins is described in [19] and is common for any language in the IntelliJ platform. The **generator** component of the project (whose implementation is discussed in future chapters) follows all of the steps outlined at this stage.

The architecture of the created plugin appears as follows:

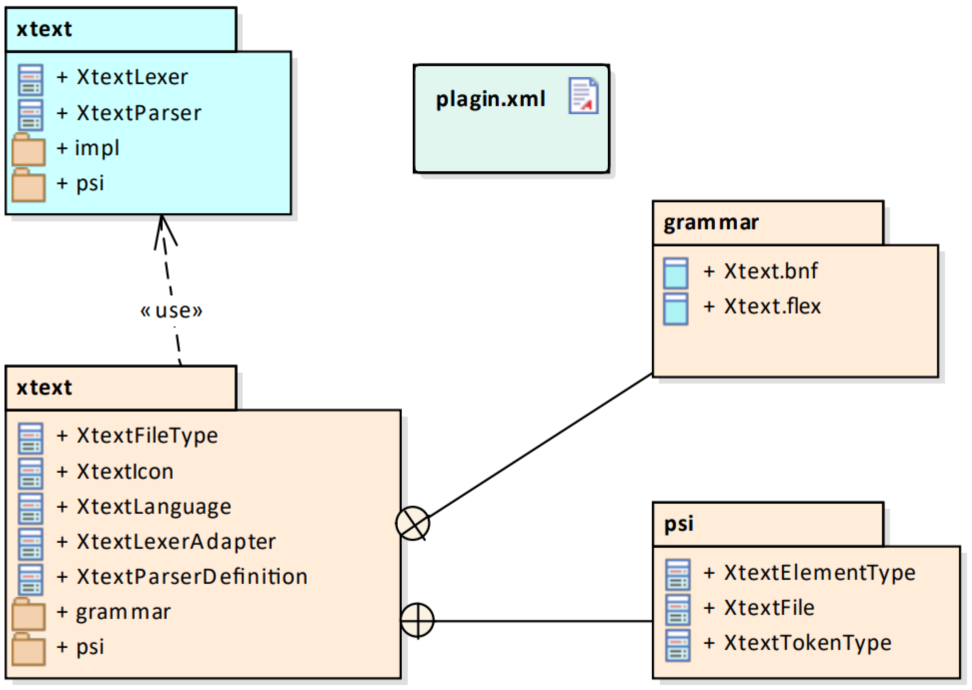


Figure 9.3: Xtext language support plugin architecture

The plugin consists of the following components:

1. The **`xtext`**(blue) package contains classes generated by Grammar-Kit.
   This component include base elements of DSL implentatation process discussed in [4.2], namely **lexer**, **parser** and **AST classes**. The IntelliJ platform semantic model is being built from PSI elements. The `psi` package contains PSI interfaces and their implementations that were generated into the `impl` package. The input files for generator are located in the **`grammar`** package.

2. The **`xtext`**(orange) package contains base classes needed to integrate a new language to the IntelliJ platform.

   - `XtextFileType` and `XtextLanguage` are required to make the IDE able to determine the type of Xtext files.
   - `XtextLexerAdapter` allows the `XtextLexer` generated by the JFlex generator to be adapted to the IntelliJ Platform Lexer API.
   - The **`psi`** package consists of classes which represent types of AST nodes that each language should have in the IntelliJ Platform: `XtextElementType` for non-terminal nodes, `XtextTokenType` for terminal nodes and `XtextFile` for root element of the file.

  - `XtextParserDefinition` class aggregates all the logic implemented above and provides parser, lexer, tokens and other important language plugin components to the IntelliJ Platform.

3. The **`plugin.xml`** is a configuration file for plugins in IntelliJ IDEA. Within it all language plugin components should be registered.

9.2.3 Final plugin architecture

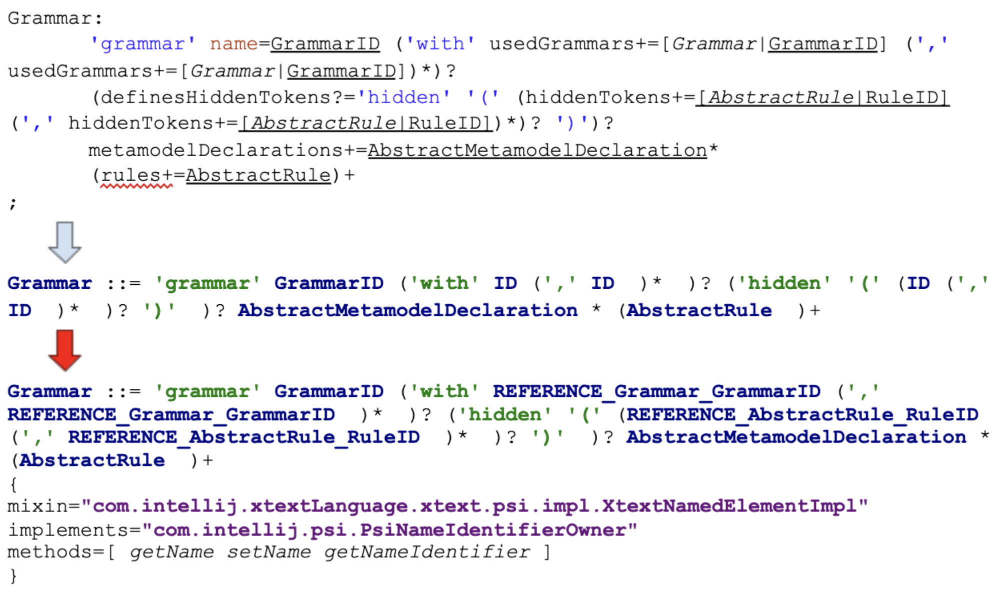
At this stage, the previously created plugin will be extended with editor support.
There are several IDE features that should be added:

- Syntax highlighting
- Completion support
- Hyperlinking
- Usage finder
- Renaming support

The first two features are not hard to implement. It requires adding a couple of classes that implement the corresponding platform's interfaces with small amount of language-specific logic.

The implementation of the references mechanism and usage finder, though is not a simple task. It requires adding some PSI helper and "util" classes. Additionally changes in the grammar file should be made.

Resolving references means the ability to go from the usage of an element to its declaration. To achieve this, the grammar elements that refer somewhere should be distinguished from other identifiers(Cross-References in Xtext notation). The solution was devised to declare new grammar rules for each referencing `ID` element in the grammar. Thus, the **`Grammar`** rule was modified as follows:



```
Grammar:
        'grammar' name=GrammarID ('with' usedGrammars+=[Grammar|GrammarID] (','
usedGrammars+=[Grammar|GrammarID])*)?
        (definesHiddenTokens?='hidden' '(' (hiddenTokens+=[AbstractRule|RuleID]
(',' hiddenTokens+=[AbstractRule|RuleID])*)? ')')?
        metamodelDeclarations+=AbstractMetamodelDeclaration*
        (rules+=AbstractRule)+
;

Grammar ::= 'grammar' GrammarID ('with' ID (',' ID  )*  )? ('hidden' '(' (ID (','
ID  )*  )? ')'  )? AbstractMetamodelDeclaration * (AbstractRule  )+

Grammar ::= 'grammar' GrammarID ('with' REFERENCE_Grammar_GrammarID (','
REFERENCE_Grammar_GrammarID  )*  )? ('hidden' '(' (REFERENCE_AbstractRule_RuleID
(',' REFERENCE_AbstractRule_RuleID  )*  )? ')'  )? AbstractMetamodelDeclaration *
(AbstractRule  )+
{
mixin="com.intellij.xtextLanguage.xtext.psi.impl.XtextNamedElementImpl"
implements="com.intellij.psi.PsiNameIdentifierOwner"
methods=[ getName setName getNameIdentifier ]
}
```

Figure 9.4: "Grammar" rule modifications

Not only ID elements have undergone changes, some rule attributes were added since the **Grammar** rule itself is a referential element of the language. These attributes modify generated PSI classes making them extend the specified classes and implement specified methods. This enables grammar files to provide their name when asked.

The final Xtext grammar language support plugin then appears as follows in the diagram below:

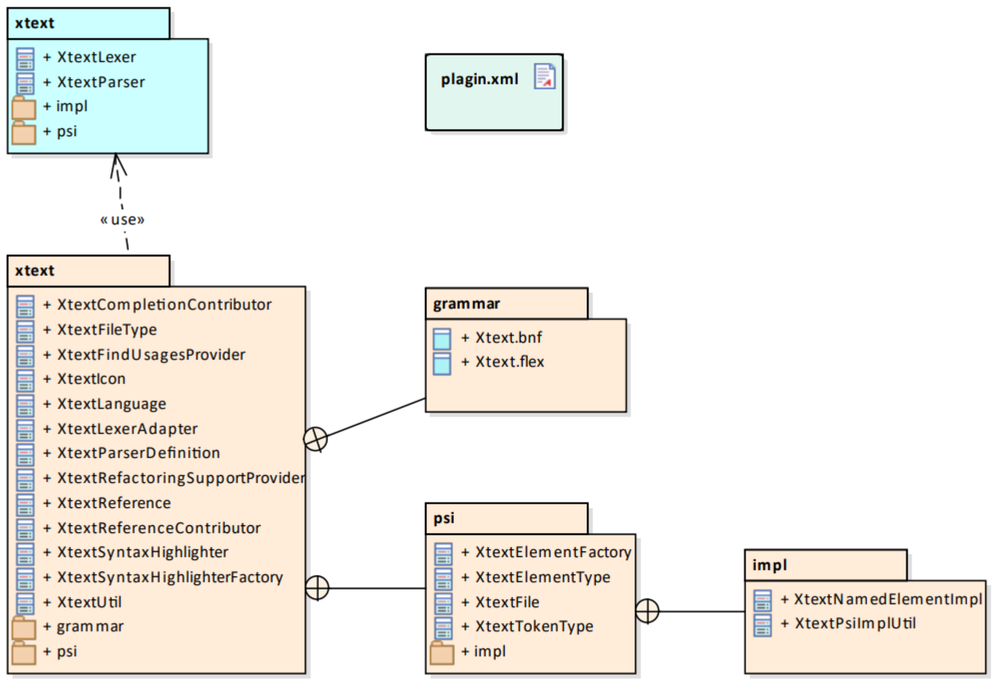
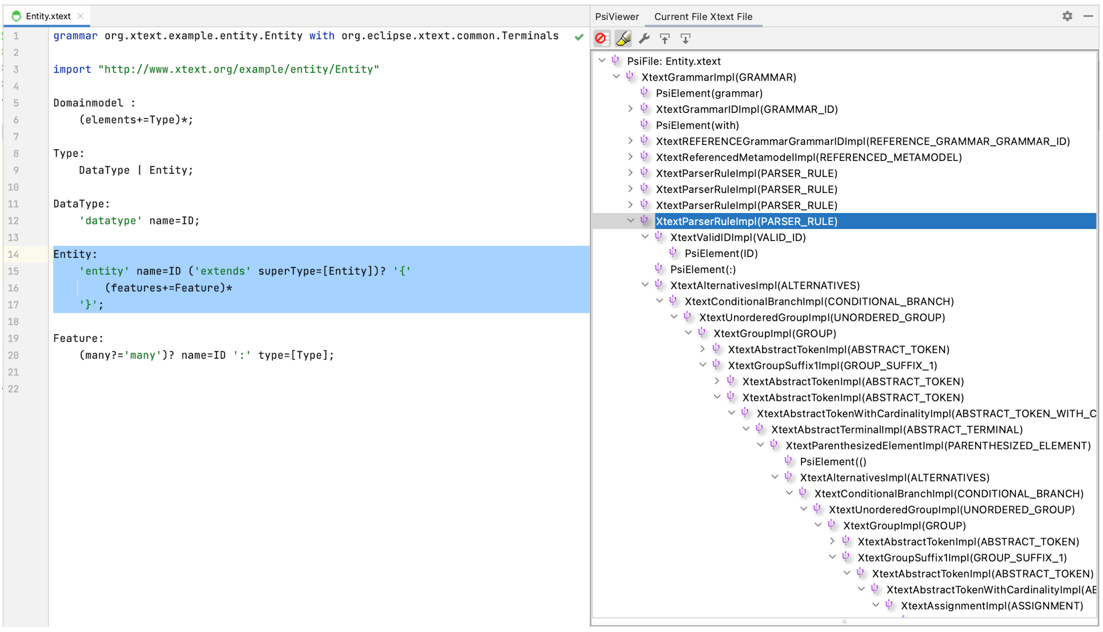


Figure 9.5: Xtext language support plugin architecture(final version)

9.2.4 Plugin testing

The validity of the plugin work has been evaluated by numerous unit tests. However, the best way to prove that this stage of the project is finished and working correctly is to examine the IDE behaviour:



```
grammar org.xtext.example.entity.Entity with org.eclipse.xtext.common.Terminals

import "http://www.xtext.org/example/entity/Entity"

Domainmodel :
    (elements+=Type)*;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=[Type];
```

```
PsiFile: Entity.xtext
  XtextGrammarImpl(GRAMMAR)
    PsiElement(grammar)
    XtextGrammarIDImpl(GRAMMAR_ID)
    PsiElement(with)
    XtextREFERENCEGrammarGrammarIDImpl(REFERENCE_GRAMMAR_GRAMMAR_ID)
    XtextReferencedMetamodelImpl(REFERENCED_METAMODEL)
    XtextParserRuleImpl(PARSER_RULE)
    XtextParserRuleImpl(PARSER_RULE)
    XtextParserRuleImpl(PARSER_RULE)
    XtextParserRuleImpl(PARSER_RULE)
      XtextValidIDImpl(VALID_ID)
        PsiElement(ID)
      PsiElement(:)
      XtextAlternativesImpl(ALTERNATIVES)
        XtextConditionalBranchImpl(CONDITIONAL_BRANCH)
          XtextUnorderedGroupImpl(UNORDERED_GROUP)
            XtextGroupImpl(GROUP)
              XtextAbstractTokenImpl(ABSTRACT_TOKEN)
              XtextGroupSuffix1Impl(GROUP_SUFFIX_1)
                XtextAbstractTokenImpl(ABSTRACT_TOKEN)
                XtextAbstractTokenImpl(ABSTRACT_TOKEN)
                  XtextAbstractTokenWithCardinalityImpl(ABSTRACT_TOKEN_WITH_C
                    XtextAbstractTerminalImpl(ABSTRACT_TERMINAL)
                      XtextParenthesizedElementImpl(PARENTHESIZED_ELEMENT)
                        PsiElement(()
                        XtextAlternativesImpl(ALTERNATIVES)
                          XtextConditionalBranchImpl(CONDITIONAL_BRANCH)
                            XtextUnorderedGroupImpl(UNORDERED_GROUP)
                              XtextGroupImpl(GROUP)
                                XtextAbstractTokenImpl(ABSTRACT_TOKEN)
                                XtextGroupSuffix1Impl(GROUP_SUFFIX_1)
                                  XtextAbstractTokenImpl(ABSTRACT_TOKEN)
                                    XtextAbstractTokenWithCardinalityImpl(AE
                                      XtextAssignmentImpl(ASSIGNMENT)
```

Figure 9.6: Entity language grammar(Xtext format) in IDEA editor

The Entity language and its grammar is taken as an example. The IntelliJ IDEA editor highlights important Xtext grammar language elements. Completion, references and renaming features are working as well. Most significantly, the PSI tree on the right is appropriately constructed.

9.3 Language support plugins generation

This subchapter will cover phases of implementation which correspond to points 2-4 of the Design plan.

The main idea is to automate the process of the language support plugin implementation. The component that will be created at this stage should be able to perform all of the steps mentioned in the previous section, including generating all language defining and editor support files, as well as translating grammar from Xtext to the Grammar-Kit format.

9.3.1 Grammar-Kit BNF and Flex files generator

There is always an input Xtext grammar file. With the plugin created in the previous section, it is now a possible to parse such files and work with them programmatically.

The first task is to create the grammar-translating mechanism. This mechanism could be implemented using knowledge, such as how Xtext grammar expressions should look in the Grammar-Kit BNF/Flex files [9.2.1]. This translation, however, will not be straightforward. With regard to the fact that some of the Xtext grammar expressions should be simplified or renamed, an **inter-structure** is a needed. The name that was chosen for this structure is **Meta model**. The Meta model will be especially useful in the "Bridge" implementation stage(remember that Xtext features that are still ignored, e.g. Assignments).

The following component diagram describes how the generating process is organized:

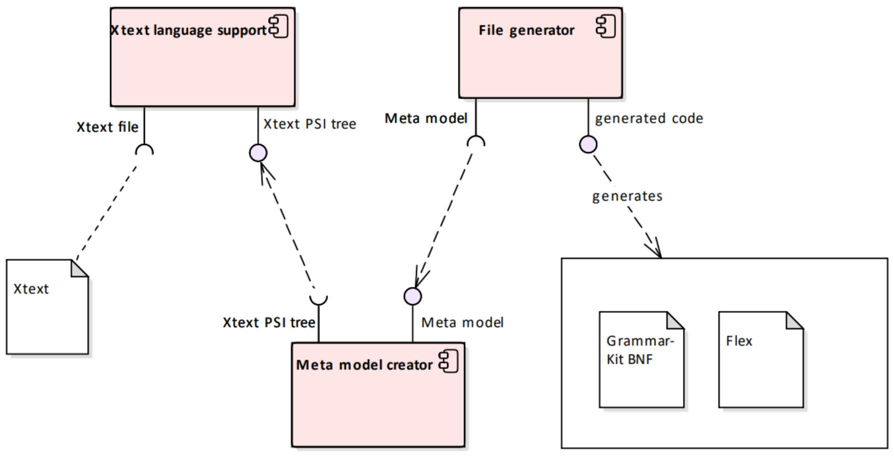


Figure 9.7: Grammar files generator component diagram

There are three distinct stages:

1. Parsing the Xtext file and building its PSI tree. The **Xtext language plugin** component created in the previous stage is responsible for this.

2. Building the **Meta model**. This is the most important and complicated to implement phase. Here the **Meta model creator** goes through the Xtext PSI tree, finds Xtext PSI elements which describe the grammar rules and creates **Meta rules** in accordance to each of them.

   Meta rule is a tree structure which carries all the information that was mentioned in the original Xtext rule. Meta rule child nodes provide strings for generators describing how they should look in BNF/Flex formats. The Xtext details such as **Return Types**, **Assignments** and **Actions** are also saved in tree nodes for future purposes. The final Meta model is actually a "forest" of Meta rules.

3. **File generator** receives the Meta model and generates content of Grammar-Kit grammar files guided by simple logic.

9.3.2 Plugin files generator

The **File generator** component should now be extended with the generator that will create all of the language-defining and editor support files. Despite the fact there were a lot of classes written in first implementation stage [9.2], none of them contains difficult logic. This generator requires the following input information:

- The language **name** is defined in Xtext grammar file, therefore is saved in the Meta model.

- Language **extension** is not part of Xtext grammar file. In Eclipse this string value is specified by the user in the wizard window when he/she creates new project. The same will be done in this project, but for now, the extension will be hardcoded.

- **Token set**: the generator will receive the token set from the Meta model by finding Meta rules built from the Xtext terminal rules.

- **Keywords** will be found using a simple filtering function through the Meta rules set.

- **Referenced elements** are Meta rules with a special mark.

- **Referential identifiers** are represented by a special type of nodes in the Meta rules.

In this way, all of the input information is prepared for the generator. It only thing that remains is to organize the Meta information correctly, generate Java files and register all providers in the `pugin.xml` file.

At this point, the implemented part of the project is able to create the full language support plugin(i.e. ready-to-use programming language environment) given an Xtext grammar file.

9.3.3 Bootstrapping

This stage is not mandatory for the implementation process. It is here only for test purposes. The fact of having the component of the program generated by the program itself is a good practice in the sphere of SDK development(e.g. ANTLR version 3 is written using a parser developed in ANTLR version 2 [6]).

The Xtext language support plugin(implemented in [9.2]) was successfully regenerated by the implemented generator. It was achieved by starting the plugin-generation process on the `Xtext.xtext` - Xtext language grammar file.

9.4 “Bridge”

In this section, the meaning of the “Bridge” component will be explained. Then the implementation of the “Bridge” component generator will be discussed.

At this point, the implemented part of the project creates language support plugins that work perfectly, but lack the component that could create a semantic model from EObjects, in the similar way to the Xtext parser in the Eclipse IDE. This component should act in a runtime, when files of the particular DSL are parsed and their PSI trees are created. Thus, the component’s primary function is to construct EObject trees out of PSI trees, realizing transition between the two tree structures. This is why the component was named “Bridge”.

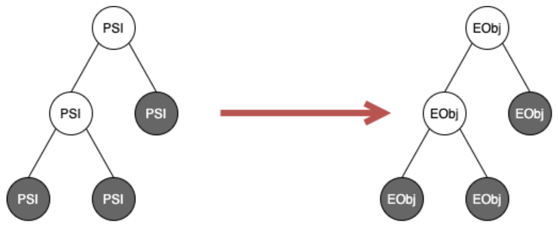


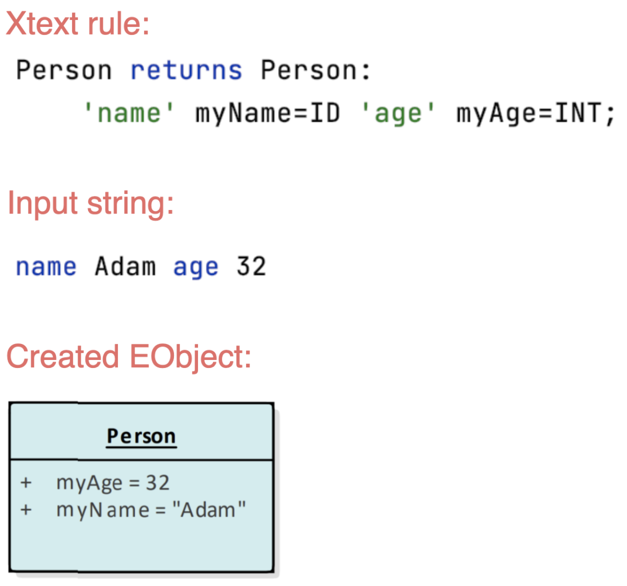
Figure 9.8: Translating PSI tree to EObjects tree

9.4.1 Advanced Xtext grammar features

To realize the “Bridge” is quite a complex task. The difficulties are caused by some Xtext grammar features that were ignored and now will be finally taken into account.
They are:

- **Return Type**
- **Assignments**
- **Simple Actions**
- **Assigned Actions**

Return Types and **Assignments** were partially explained in [5.2.2]. The Parser Rule is always associated with some EClass. The object of a given EClass is created if the string (which is matched by that rule) is parsed by the Xtext parser. Assignments bind parts of the Parser Rule to EClass properties. For example:



```
Person returns Person:
    'name' myName=ID 'age' myAge=INT;
```

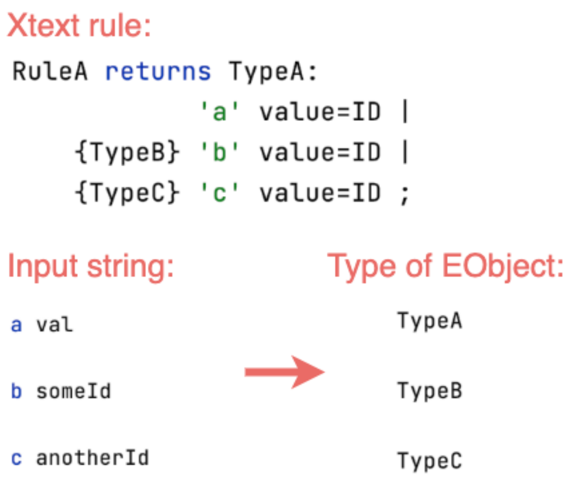
```
name Adam age 32
```

```
Person
+  myAge = 32
+  myName = "Adam"
```

Figure 9.9: Example of created EObject

Actions are another tool that operate with types of objects in the semantic model.

Simple Actions are used to specify type of EObject to be created, similar to Return Types, but are more flexible. The return type can vary depending on which part of the rule describes the string of the language. For example:

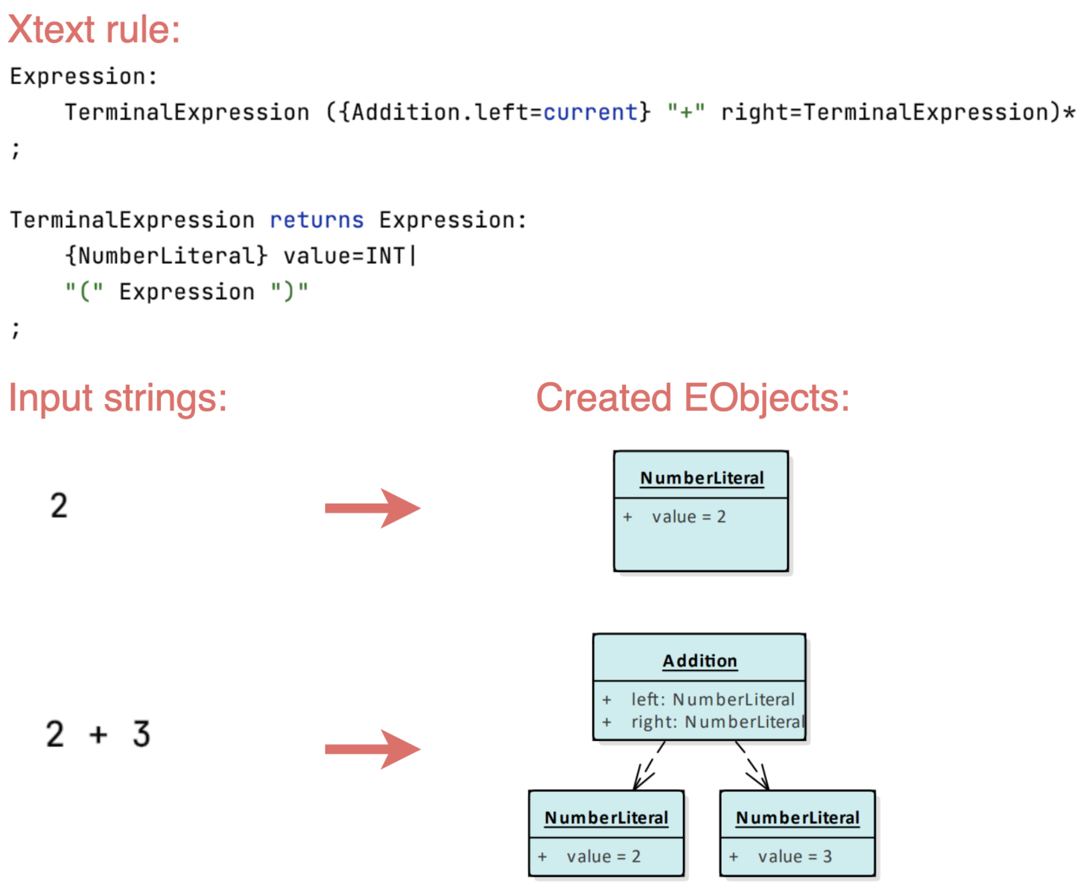


```
RuleA returns TypeA:
              'a' value=ID |
     {TypeB} 'b' value=ID |
     {TypeC} 'c' value=ID ;
```

```
a val            TypeA
b someId    →    TypeB
c anotherId      TypeC
```

Figure 9.10: Simple Actions example

Assigned Actions are used mainly for one purpose: to deal with the left recursion. Left recursion is a problematic situation for a top-down parser (LL parser [5]). It means that the grammar rule starts with a reference to itself and usually appears in grammars which describe some expressions(`Addition : Addition '+' ...;`). Consider the Assigned Action usage example:



```
Expression:
    TerminalExpression ({Addition.left=current} "+" right=TerminalExpression)*
;

TerminalExpression returns Expression:
    {NumberLiteral} value=INT|
    "(" Expression ")"
;
```

Figure 9.11: Assigned Actions example

The Assigned Action {`Addition.left=current`} is an action that rewrites the tree. This forces the parser to create an instance of the EClass Addition and assign the current object to-be-returned to the `Addition` expression's feature left.

9.4.2 "Bridge" design

Test implementation

The first version of "Bridge" was implemented for the Entity language support plugin. The process of translating PSI elements to EObjects was organized by the means of a visitor pattern. A class which contains methods for each language rule was written. Despite the fact that Entity language grammar does not even contain difficult Xtext features, the logic of these methods was convoluted and therefore difficult to generate. Therefore, the decision was made to devise more general method of "Bridge" translation.

Final implementation

The “heart” of the final “Bridge” implementation is a recursive algorithm that traverses the given PSI tree and builds the model of EObjects. This algorithm is common for all languages. It takes a PSI element, iterates over element’s children and builds EObjects in parallel.

It is important to note that every non-terminal PSI element represents particular Xtext Parser Rule, and that terminal PSI elements are representations of either keywords or Xtext Terminal rules. Accordingly, PSI elements can be discussed as if they were elements of Xtext grammar. The following statement uses this analogy:
Each PSI element could be in one of the following **states**:

1. The element is **assigned** to the feature.
2. The element is preceded by the **Simple Action** or by the **Assigned Action**.
3. The combination of states 1 and 2.
4. No feature is applied to the element.

The algorithm should now be able to decide how to proceed with the PSI element’s child. Namely, whether to assign it to the feature of currently producing EObject or perform a tree rewrite action. The solution is to write a class for each grammar rule which can provide all of the information about itself and its children. These classes implement the following interface:

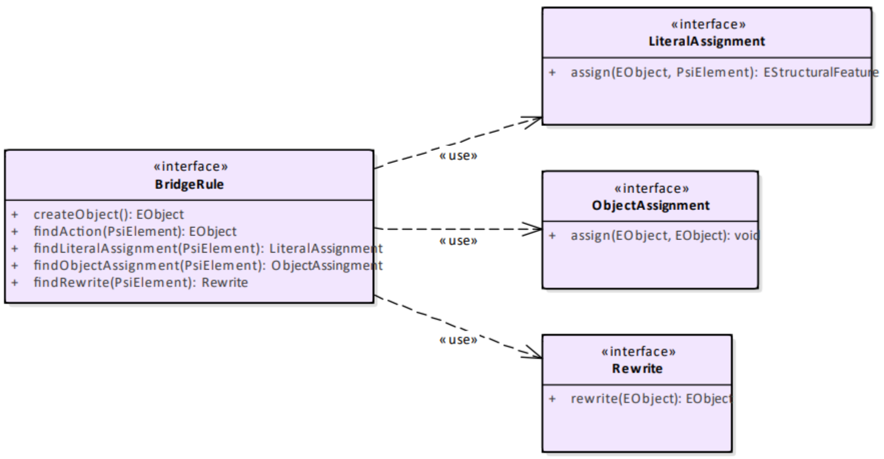

«interface»
LiteralAssignment
+ assign(EObject, PsiElement): EStructuralFeature
«use»
«interface»
BridgeRule
+ createObject(): EObject
+ findAction(PsiElement): EObject
+ findLiteralAssignment(PsiElement): LiteralAssignment
+ findObjectAssignment(PsiElement): ObjectAssingment
+ findRewrite(PsiElement): Rewrite
«use»
«interface»
ObjectAssignment
+ assign(EObject, EObject): void
«use»
«interface»
Rewrite
+ rewrite(EObject): EObject

Figure 9.12: **`BridgeRule`** class diagram

There is BridgeRule interface and 3 helping functional interfaces displayed on the diagram. Note that BridgeRule takes PSI elements as parameters and provides EOjects or operations under EOjects. The method `createObject` returns the instance of rule's return type. Other 4 methods have the similar logic: depending on the type of taken PsiElement provide information about its state [9.4.2].

Consider an example from figure 9.9 and `BridgeRule` for **`Person`** rule. After passing the PSI element of **`ID`** ("Adam") to the `findLiteralAssignment`, the method will return the function that assigns the PsiElement's text to the Person's feature "myName". If the PSI element of the keyword **`name`** will be passed, `null` will be returned.

Here is the pseudocode of the traversing algorithm:

```
Method visitElement(element)
BEGIN
   bridgeRule ← getBridgeRuleForPsiElement(element)
   current ← null
   FOR child IN element.children DO
      newObject ← bridgeRule.findAction(child)
      IF newObject != null THEN
         current ← newObject
      END IF
      rewrite ← bridgeRule.findRewrite(child)
      IF rewrite != null THEN
         IF current == null THEN
            current ← bridgeRule.createObject()
         END IF
         current ← rewrite.rewrite(current)
      END IF
      literalAssignment ← bridgeRule.findLiteralAssignment(child)
      IF literalAssignment != null THEN
         IF current == null THEN
            current ← bridgeRule.createObject()
         END IF
         literalAssignment.assign(current, child)
      ELSE
         newObject ← visitElement(child)
         IF newObject != null THEN
            assigment ← bridgeRule.findObjectAssignment(child)
            IF assigment != null THEN
               IF current == null THEN
                  current ← bridgeRule.createObject()
               END IF
               assigment.assign(current, newObject)
            ELSE
               current ← newObject
            END IF
         ELSE IF isCrossReference(child) THEN
            createCrossReference(child, current)
         END IF
      END IF
   END FOR
   RETURN current
END
```

Figure 9.13: Pseudocode of the "Bridge" alorithm

One thing that has not yet been mentioned is processing **Cross References**.
The `createCrossReference` method saves referencing identifiers without resolving. They should be resolved after the first traversing through the tree, when all the names are

known. The class that connects all discussed parts together is called `BridgeCreator`. It knows how to map a certain PSI type to its `BridgeRule`, how to save unresolved references, etc.

The **`Bridge`** component is a part of every DSL support plugin. The final architecture of the "Bridge" component with relation on previously created component is shown in the diagram below:

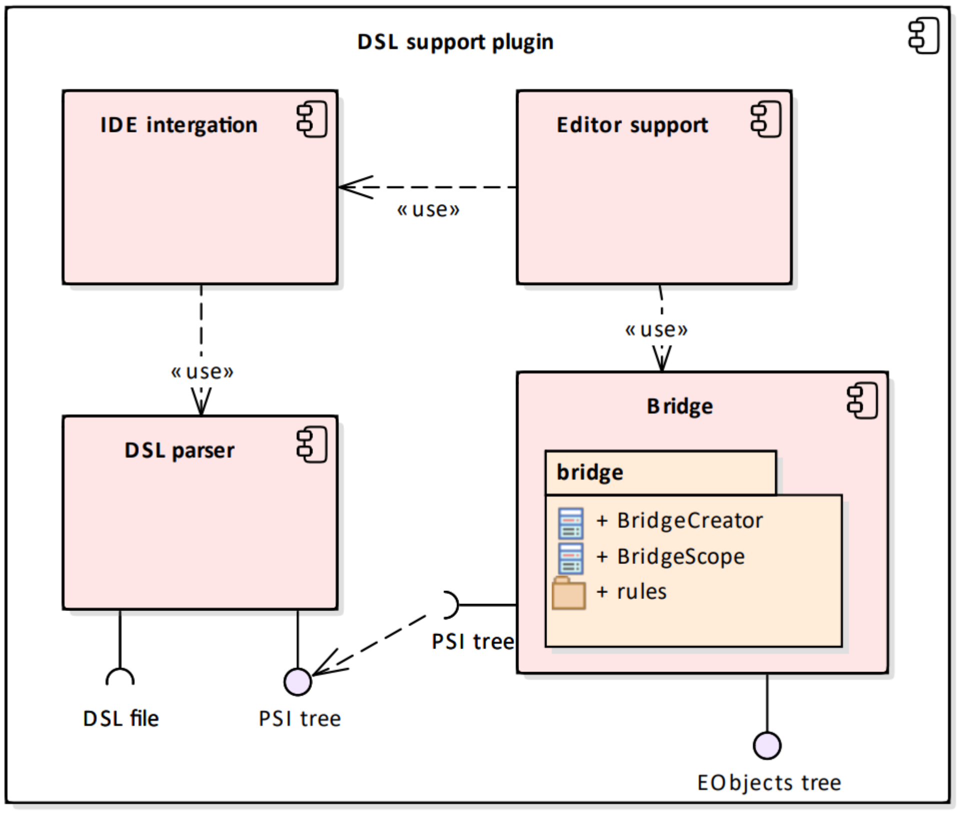


Figure 9.14: Architecture of DSL support plugin with **`Bridge`** component

9.4.3 "Bridge" generator

The next step is the implementation of the "Bridge" generator. The **File generator** component that was created in [9.3] should now be extended with the ability to generate **`Bridge`** component.

The discussed Bridge component contains quite complex logic. At the same time, the generator logic should remain simple.

The input structure for **File generator** component is **Meta model** described earlier. Therefore all of the required information about Ecore types, assignments and actions is conveniently stored in its tree's nodes.

The implementation of this stage is then divided into two parts:

1. Making changes into **Meta model creator** component.

2. Implementation of “Bridge” file generators.

The **first** part is more complex. The fact that EObjects model building is a two-step process in the IntelliJ IDEA(neither one-step like in the Eclipse IDE) causes some difficulties. In Eclipse, the Xtext parser builds the EObjects model on the run, assigning incoming tokens to the particular features directly. The “Bridge” in IDEA, on the other hand, is required to work with the PSI structure that is received from the Grammar-Kit parser.

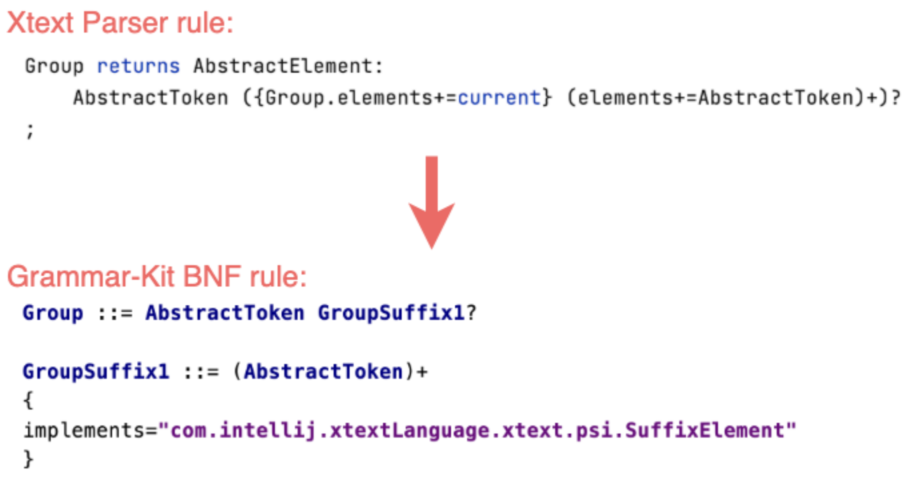
Since `BridgeRules` provide information about a given rule’s children by examining their types, identical language elements with different roles in the semantic model should be presented by `PsiElements` of different types - otherwise they will be indistinguishable.

Therefore the PSI tree should be prepared correctly. This preparation should take place at the stage of Meta model building. Some grammar elements should be modified in Grammar-Kit BNF. These modifications are usually fall into two categories:

- Creating a copy of the rule with a different name and replacing initial rule usages with usages of a created copy. This ensures the uniqueness of types within one rule.

- Moving a certain part of a rule to a separate rule, thereby producing additional nodes (`SuffixElements`) in the AST. Although the AST complication might not be considered “good practice”, in some cases there is not possible to implement Xtext features like **Actions** without these “suffixes”.

For example, here is a modification of the `Group` rule of Xtext grammar language:



```
Group returns AbstractElement:
    AbstractToken ({Group.elements+=current} (elements+=AbstractToken)+)?
;
```

```
Group ::= AbstractToken GroupSuffix1?

GroupSuffix1 ::= (AbstractToken)+
{
implements="com.intellij.xtextLanguage.xtext.psi.SuffixElement"
}
```

Figure 9.15: Rule modification example

At this stage, the **Meta model creator** component was greatly complicated and extended with the logic of analyzing and modifying **Meta trees**.

The **second** part involves implementing `BridgeCreator` and `BridgeRules` file generators. This stage is not even nearly as difficult as the previous one. All troublesome spots of the grammar are avoided and all Ecore model-related data is stored in the **Meta model**. The generators only need to iterate through the **Meta rules** and form strings based on information they provide.

After this is done, the plugin is able to generate the **`Bridge`** component of the **DSL support plugins** (figure 9.14). The diagram from the figure 9.7 can be updated:

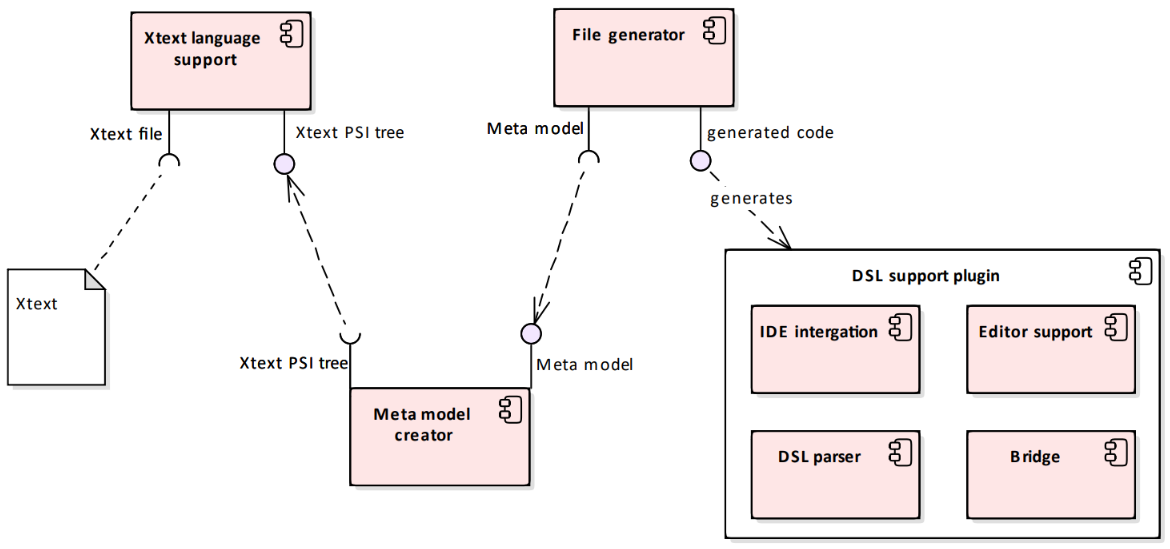
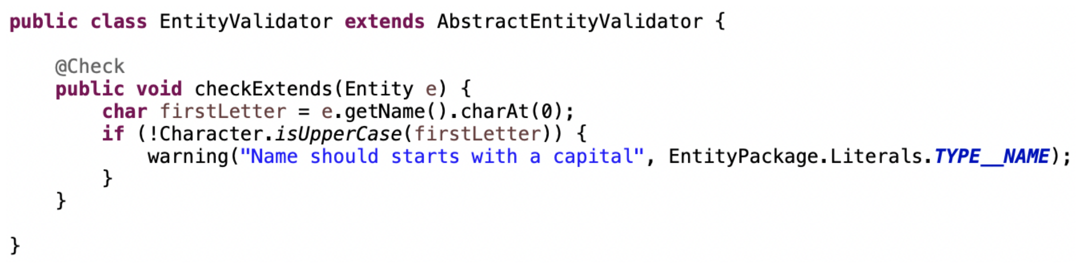


Figure 9.16: Updated file generator component diagram

9.5 Validation Rules

One of the main advantages of DSLs is the possibility to statically validate domain-specific constraints. Xtext framework provides a special API-hook which allows the user to define custom validation rules with ease:



```
public class EntityValidator extends AbstractEntityValidator {

    @Check
    public void checkExtends(Entity e) {
        char firstLetter = e.getName().charAt(0);
        if (!Character.isUpperCase(firstLetter)) {
            warning("Name should starts with a capital", EntityPackage.Literals.TYPE__NAME);
        }
    }

}
```

Figure 9.17: Example of Xtext validation rule

One single method adds a new validation rule to the IDE. After this the validation mechanism will check all Entities in Entity DSL files if their name starts with a capital letter. `warning()` method takes two parameters: the message to be displayed, and the feature to be highlighted. There are also `error()` and `info()` methods available which differ in the way they highlight the code.

The current implementation stage's purpose is to implement Xtext the Validation Rules mechanism, copying its API. Of course, the IntelliJ platform provides developers with the opportunity to define custom validation logic. This logic applies to PSI elements. Xtext Validation Rules however "live" in the EObjects world. Fortunately the "Bridge" solves this inconsistency, allowing two-side communication between PSI elements and corresponding EObjects.

9.5.1 Adapter design

To achieve complete API replication, a kind of **adapter** was created. The adapter allows the developers to write validation rules methods in the Xtext manner, describing what EObject's features should be highlighted in what cases. Most importantly, these rules cause the highlighting of corresponding PSI elements. The following diagram shows, how the whole process is organized:

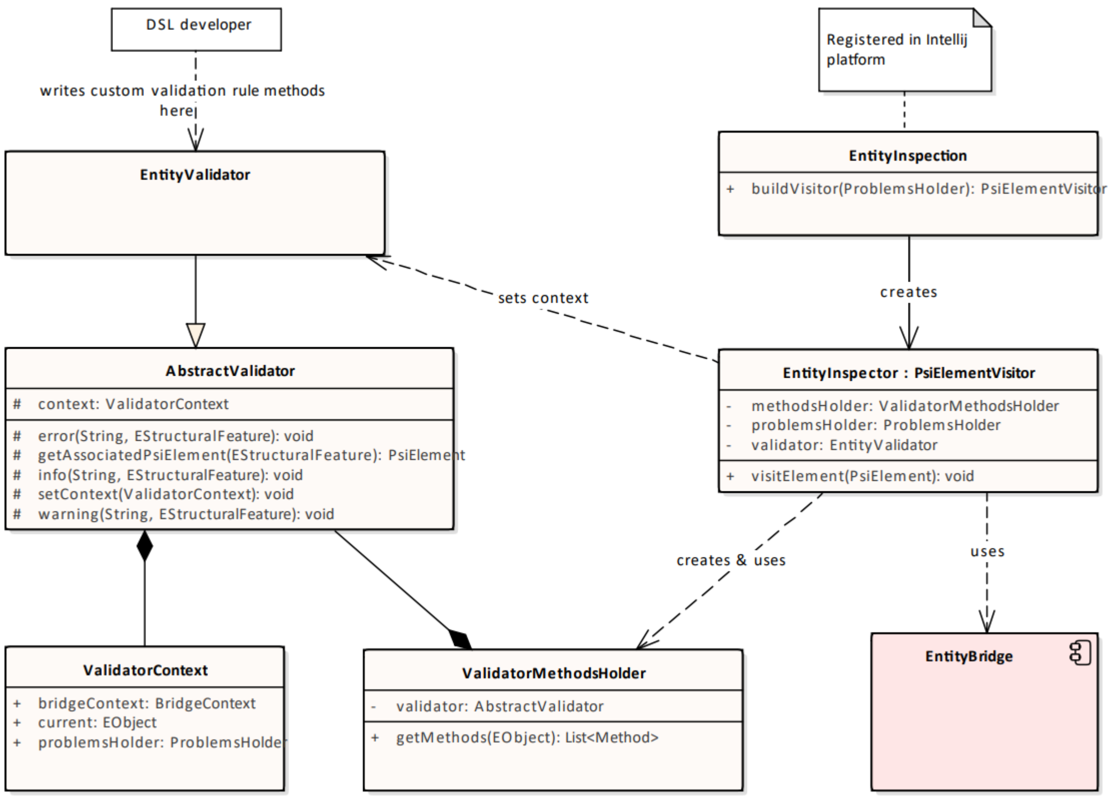

DSL developer
writes custom validation rule methods here
EntityValidator
Registered in Intellij platform
EntityInspection
+ buildVisitor(ProblemsHolder): PsiElementVisitor
sets context
creates
AbstractValidator
# context: ValidatorContext
# error(String, EStructuralFeature): void
# getAssociatedPsiElement(EStructuralFeature): PsiElement
# info(String, EStructuralFeature): void
# setContext(ValidatorContext): void
# warning(String, EStructuralFeature): void
EntityInspector : PsiElementVisitor
- methodsHolder: ValidatorMethodsHolder
- problemsHolder: ProblemsHolder
- validator: EntityValidator
+ visitElement(PsiElement): void
creates & uses
uses
ValidatorContext
+ bridgeContext: BridgeContext
+ current: EObject
+ problemsHolder: ProblemsHolder
ValidatorMethodsHolder
- validator: AbstractValidator
+ getMethods(EObject): List<Method>
EntityBridge

Figure 9.18: Class diagram of the Validator component

The diagram uses the Entity DSL as an example, but the structure applies to any language. The `EntityValidator` and its parent `AbstractValidator` classes provide the Xtext Validation Rules API which means that the developer can write validation methods like he used to do in Eclipse. The `EntityInspection` and `EntityInspector` classes are parts of the IntelliJ platform. The `EntityInspection` class is registered as a tool for local inpections, and therefore will be called by IDE editor at the syntax highlighting stage. The `EntityInspector` is responsible for inspecting and highlighting. Under normal circumstances it should perform the following logic:

1. Visit a particular PSI element.
2. Register problems via `ProblemsHolder` if needed.

In order to replicate the Xtext API, the order of events is changed:

1. Visit a particular PSI element.
2. Ask the `EntityBridge` to create an EObjects model of a PSI element's containing file(and cache it in order to not perform model building every time).
3. Get an EObject that corresponds to the PSI element.
4. Set context of `EntityValidator`.
5. Using `ValidatorMethodsHolder` object, find methods in `EntityValidator` that work with current EObject's type.
6. Invoke received methods via Java reflection.
7. Each of these methods ends in one of `warning()`, `error()` or `info()` methods. In those, by the means of provided context, the reverse communication between two structures happens. The PSI element that corresponds to the specified `EStructuralFeature` is found, and, again, using the `ProblemsHolder` the problem to be highlighted is registered.

The structure described above forms the next **`Validation`** component of language plugins. It means that generator must create it. Fortunately, this component doesn't differ from one DSL to another, so the Validation component generator was implemented without much difficulty and there are no interesting details to pay attention to.

9.6 Wizard

The first thing that users of an IDE encounter is a wizard that helps them to create a new project, whatever type of project it is. The IntelliJ IDEA plugin developed within this bachelor thesis adds a completely new type of projects to IDE - Xtext projects. Therefore the presence of a “new Xtext project wizard” is required.

9.6.1 Wizard design

The wizard will contain 3 steps:

1. SDK selection step.
2. Xtext-related information configuration step.
3. Project name and location configuration step.

While the first and the last steps are common for many projects in IDEA, the second step is unique for Xtext projects and deserves a closer look.

The wizard will have 2 modes:

1. Creation of a new Xtext project.
2. Creation of a project from existing Xtext grammar file.

The first mode is quite primitive, it only asks the user to provide **language name** and **language extension** values. If this mode is chosen, a nearly empty project is created.

The second mode is more interesting and has more usages in practice. The specified grammar file will serve as a starting point for generators and the user will receive the ready-to-use environment after the wizard will end its work. However, before that, the wizard needs to make sure whether all of the required information is provided. A single grammar file will not be sufficient, since every grammar uses at least one Ecore model and often depends on another grammars. As a result, the wizard will not let the user go further to the final step until he/she provides all dependencies.

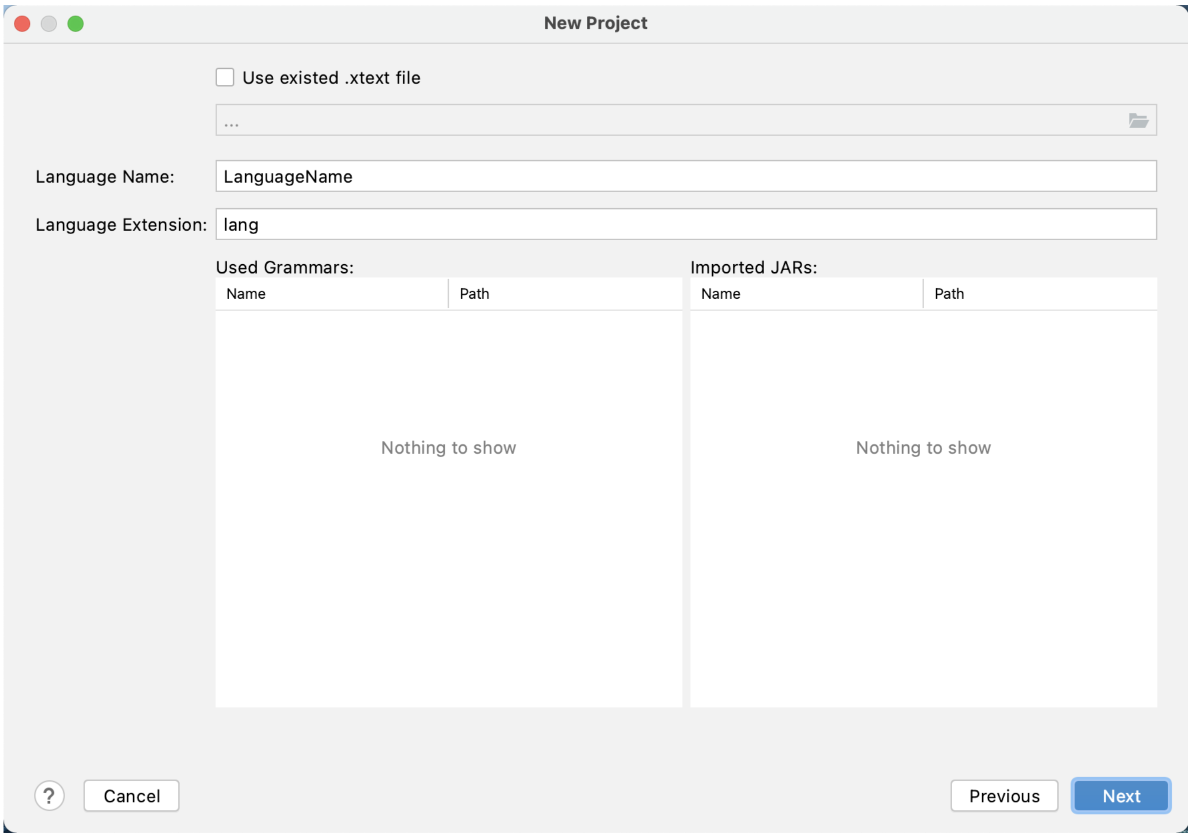
The wizard should also analyze provided additional grammars recursively since they will have dependencies too. In this way, all inputs for Xtext project generation are follows:

1. Language name : String value.
2. Language extension : String value.
3. Used Xtext grammars files(including the language grammar) : list of `*.xtext` files.

4. Used Ecore models : list of JARs.

The wizard implementation process will not be discussed in detail here. It consists mostly of defining IntelliJ UI components classes that aren't particularly interesting and hardly related to the topic of the thesis.

After the work is completed, the wizard's second step appears as follows:



Language Name: LanguageName

Language Extension: lang

Figure 9.19: The Xtext new project wizard, second step

The wizard window has fields for each input type. These fields are protected by validation mechanisms. Thus, for example, it is not possible to pass incorrect JAR to the field that expexts Ecore model of the particular URI. An example of the this step with the filled values will be shown in the next chapter.

9.6.2 Project generation

The wizard that has been implemented is only a part of the **Module builder** component. This component is responsible for the construction and configuration of new projects. The generated project is a language-support plugin, thus it uses Gradle as the assembling tool. Therefore the Xtext module builder should extend Gradle module builder, which provides all module configuration logic(e.g. defining source roots).

The module builder then should perform a number of steps in the correct order to ensure the project is built correctly:

1. Receive all of the input data from the **Wizard**.
2. Copy all JARs and Xtext grammar files to the new project.
3. Using **Xtext File Generaror** component generate all Xtext languge plugin files into the new project.
4. Configure the **Gradle build script**, defining source directories and paths to JARs.
5. Delegate further configuration of the project to **Gradle module builder**.
6. Wait until the previous step is completed.
7. Register all extensions in `plugin.xml`.
8. Invoke **Grammar-Kit** parser and lexer generation actions on the corresponding files.

After the **Xtext module builder** is implemented, the work can be considered as done. The Xtext plugin JAR could be created and used as IntelliJ IDEA plugin. The final component diagram could be found in the appendix section [A].

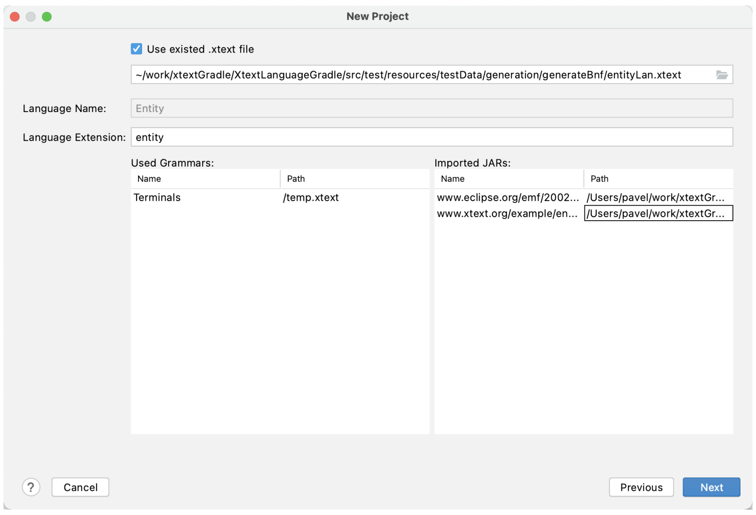
Chapter 10

Testing and evaluation

Let's now evaluate the results of the work and test the implemented plugin functionality. The testing will be performed manually and the IDE behaviour will be illustrated in the screenshots.

10.1 Creating new Xtext project

The new Xtext project is created in the same way as all other projects in IDEA are: by choosing `File > New > Project...` in the file menu bar. Choosing the Xtext project type from the list on the left will show the Xtext project wizard. The already-familiar Entity DSL is used as an example in this chapter. The second step of the wizard with all required data provided then looks as follows:



[✓] Use existed .xtext file

~/work/xtextGradle/XtextLanguageGradle/src/test/resources/testData/generation/generateBnf/entityLan.xtext

Language Name: Entity

Language Extension: entity

Used Grammars:

Name	Path
Terminals	/temp.xtext

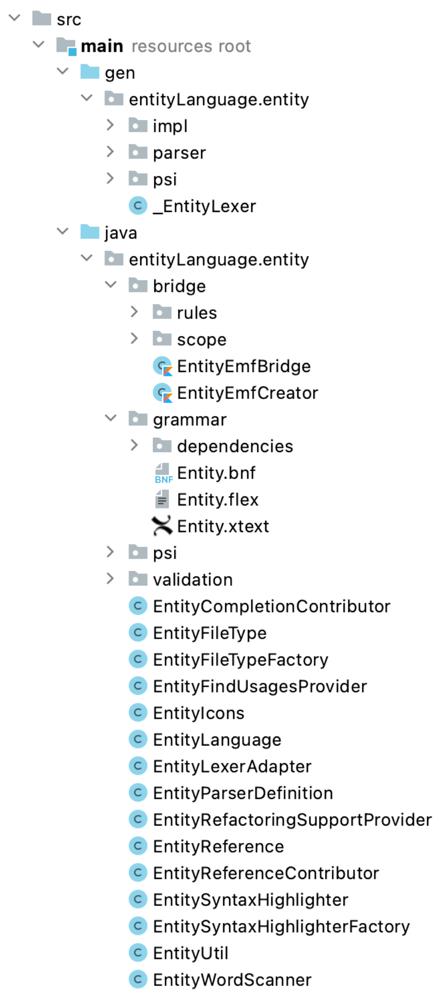
Imported JARs:

Name	Path
www.eclipse.org/emf/2002...	/Users/pavel/work/xtextGr...
www.xtext.org/example/en...	/Users/pavel/work/xtextGr...

Figure 10.1: The Xtext new project wizard, second step with the input data filled

Defining the project name and location on the last step and clicking on the "Finish" button will start the project creation process.

After a few seconds the user will see the generated project with the following structure:

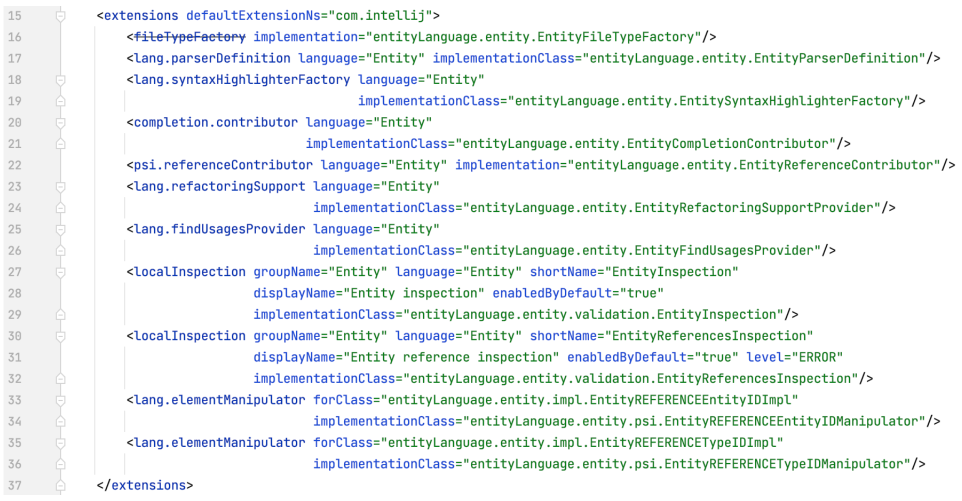


```
src
  main resources root
    gen
      entityLanguage.entity
        impl
        parser
        psi
        _EntityLexer
    java
      entityLanguage.entity
        bridge
          rules
          scope
          EntityEmfBridge
          EntityEmfCreator
        grammar
          dependencies
          Entity.bnf
          Entity.flex
          Entity.xtext
        psi
        validation
        EntityCompletionContributor
        EntityFileType
        EntityFileTypeFactory
        EntityFindUsagesProvider
        EntityIcons
        EntityLanguage
        EntityLexerAdapter
        EntityParserDefinition
        EntityRefactoringSupportProvider
        EntityReference
        EntityReferenceContributor
        EntitySyntaxHighlighter
        EntitySyntaxHighlighterFactory
        EntityUtil
        EntityWordScanner
```

Figure 10.2: The plugin structure generated by plugin. Entity DSL

The generated project represents the plugin that adds the support of the Entity DSL to the IntelliJ IDEA IDE. The parsing support files(**parser**, **lexer**, **PSI classes**) are located in the `gen` package. Files that integrate the new language to the IntelliJ platform and provides editor support are those that rooted in the `java.enityLanguage.entity` package and in the nested `java.enityLanguage.entity.psi` package. The **"Bridge"** files are located in the corresponding `bridge` directory. **Validation rules** files were generated to the `validation` package. The `grammar` folder contains language grammar specification files: `Entity.bnf`, `Entity.flex` - specifications for Grammar-Kit plugin and `Entity.xtext` - the DSL grammar in Xtext format which is the main file the user should work with.

All IDE extensions are registered correctly in the `plugin.xml` file:

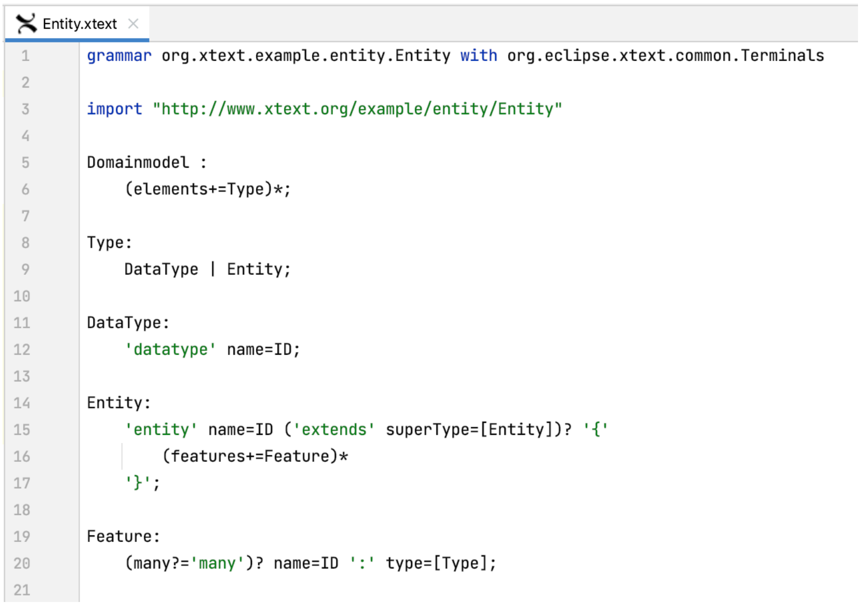


```
<extensions defaultExtensionNs="com.intellij">
    <~~fileTypeFactory~~ implementation="entityLanguage.entity.EntityFileTypeFactory"/>
    <lang.parserDefinition language="Entity" implementationClass="entityLanguage.entity.EntityParserDefinition"/>
    <lang.syntaxHighlighterFactory language="Entity"
                                   implementationClass="entityLanguage.entity.EntitySyntaxHighlighterFactory"/>
    <completion.contributor language="Entity"
                            implementationClass="entityLanguage.entity.EntityCompletionContributor"/>
    <psi.referenceContributor language="Entity" implementation="entityLanguage.entity.EntityReferenceContributor"/>
    <lang.refactoringSupport language="Entity"
                             implementationClass="entityLanguage.entity.EntityRefactoringSupportProvider"/>
    <lang.findUsagesProvider language="Entity"
                             implementationClass="entityLanguage.entity.EntityFindUsagesProvider"/>
    <localInspection groupName="Entity" language="Entity" shortName="EntityInspection"
                     displayName="Entity inspection" enabledByDefault="true"
                     implementationClass="entityLanguage.entity.validation.EntityInspection"/>
    <localInspection groupName="Entity" language="Entity" shortName="EntityReferencesInspection"
                     displayName="Entity reference inspection" enabledByDefault="true" level="ERROR"
                     implementationClass="entityLanguage.entity.validation.EntityReferencesInspection"/>
    <lang.elementManipulator forClass="entityLanguage.entity.impl.EntityREFERENCEEntityIDImpl"
                             implementationClass="entityLanguage.entity.psi.EntityREFERENCEEntityIDManipulator"/>
    <lang.elementManipulator forClass="entityLanguage.entity.impl.EntityREFERENCETypeIDImpl"
                             implementationClass="entityLanguage.entity.psi.EntityREFERENCETypeIDManipulator"/>
</extensions>
```

Figure 10.3: Generated plugin.xml file, extensions tag

10.2 Xtext grammar language editor

Let's look at the `Entity.xtext` closer:



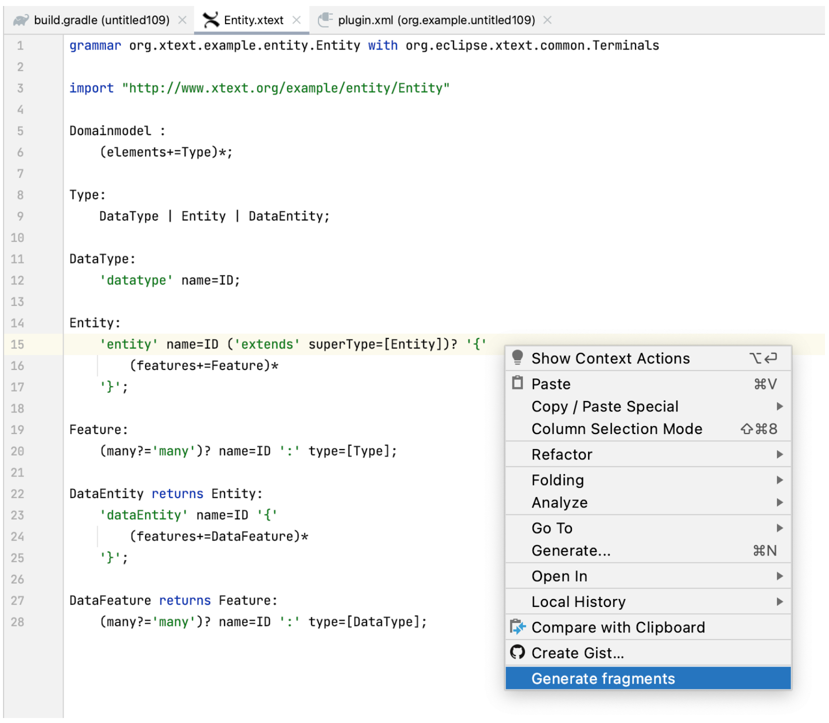
```
grammar org.xtext.example.entity.Entity with org.eclipse.xtext.common.Terminals

import "http://www.xtext.org/example/entity/Entity"

Domainmodel :
    (elements+=Type)*;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=[Type];

```

Figure 10.4: Entity language grammar in Xtext format

The user can enjoy all editor features the plugin provides. Among them are general features like syntax highlighting, error marking, completion support and more complex: support of references from Rule Calls to rule declarations, references to Ecore models and classes and renaming feature.

Let's edit the Entity grammar by adding a few new rules. Since every rule in Xtext should be mapped to the concrete Ecore type and the current project has only one(Entity)

Ecore model imported, it is impossible for now to go beyond the domain and declare a fundamentally new language structure. To do so, the developer is required to connect a new Ecore model to the project or edit Entity model(using EMF in Eclipse). Thus the new rules will be modifications of `Entity` and `Feature`. Here is the updated Entity grammar:



```
grammar org.xtext.example.entity.Entity with org.eclipse.xtext.common.Terminals

import "http://www.xtext.org/example/entity/Entity"

Domainmodel :
    (elements+=Type)*;

Type:
    DataType | Entity | DataEntity;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=[Type];

DataEntity returns Entity:
    'dataEntity' name=ID '{'
        (features+=DataFeature)*
    '}';

DataFeature returns Feature:
    (many?='many')? name=ID ':' type=[DataType];
```

Figure 10.5: Updated Entity grammar

The grammar is now extended by **DataFeature** rule - a Feature that can only accept DataTypes(neither other Entities) and **DataEntity** rule - a simplified version of Entity that can only contain DataFeatures.

After changes to the grammar file are completed, the entire plugin can be regenerated. It is possible to so in one click by calling the **"Generate fragments" action** which is located in the editor's context menu.

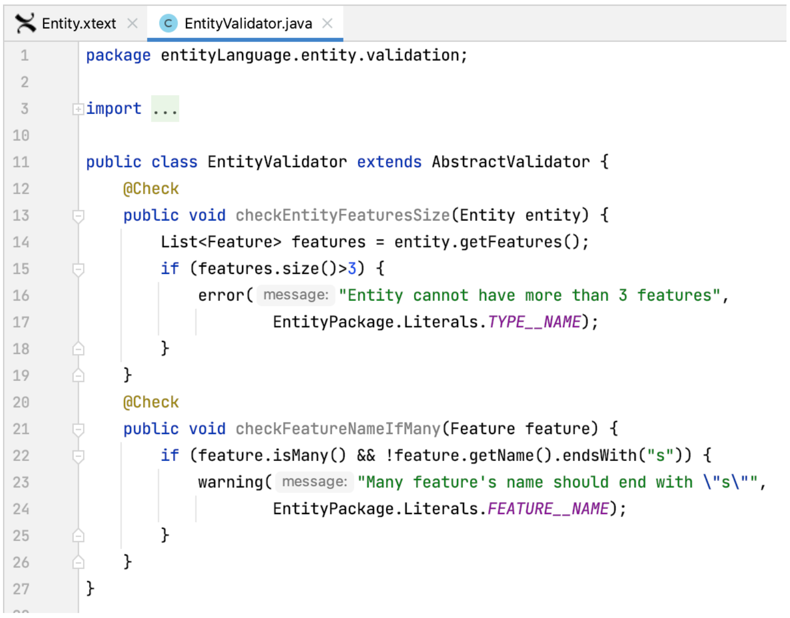
10.3 Defining Validation rules

The next functionality that should be covered in this chapter is defining Validation rules. Let's define 2 rules of the following logic:

1. Each Entity cannot have more than 3 features. This rule will be assigned the **error** severity level.

2. The Feature, marked by keyword `many`, should have a name ending with “s” letter. This rule will have the **warning** severity level.

The `EntityValidator` class will contain 2 corresponding methods:



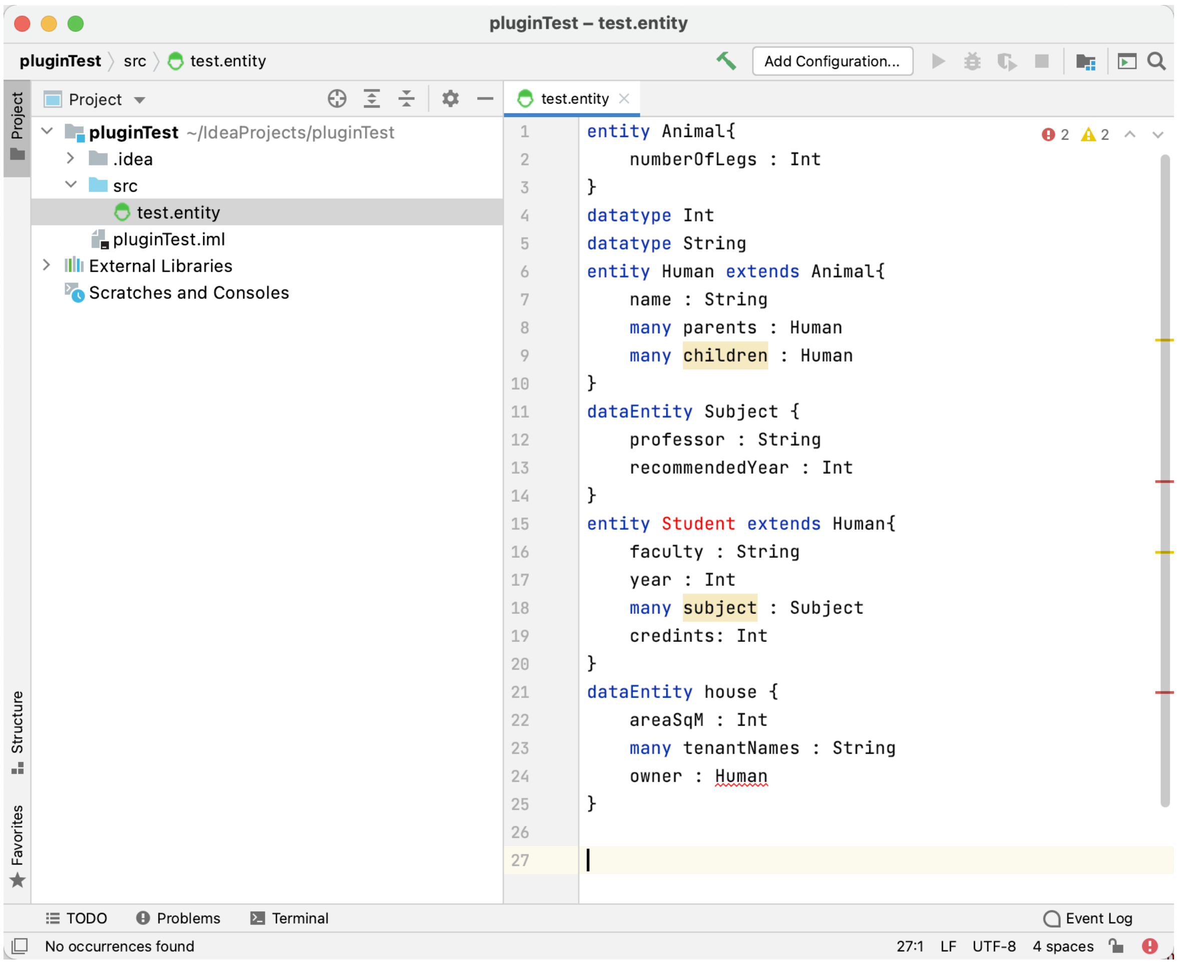
```java
package entityLanguage.entity.validation;

import ...

public class EntityValidator extends AbstractValidator {
    @Check
    public void checkEntityFeaturesSize(Entity entity) {
        List<Feature> features = entity.getFeatures();
        if (features.size()>3) {
            error( message: "Entity cannot have more than 3 features",
                    EntityPackage.Literals.TYPE__NAME);
        }
    }
    @Check
    public void checkFeatureNameIfMany(Feature feature) {
        if (feature.isMany() && !feature.getName().endsWith("s")) {
            warning( message: "Many feature's name should end with \"s\"",
                    EntityPackage.Literals.FEATURE__NAME);
        }
    }
}
```

Figure 10.6: EntityValidator class

The result of the actions described here is shown in the next section.

10.4 Launching new instance of IDEA with included language support

Now it is time to run **runIde** gradle task. This will build the current project and launch the new instance of IntelliJ IDEA with current plugin imported. The started IDEA is provided with Entity language support logic, so let’s open new project, create a file with **entity** extension in it and write some expressions in Entity DSL:



```
entity Animal{
    numberOfLegs : Int
}
datatype Int
datatype String
entity Human extends Animal{
    name : String
    many parents : Human
    many children : Human
}
dataEntity Subject {
    professor : String
    recommendedYear : Int
}
entity Student extends Human{
    faculty : String
    year : Int
    many subject : Subject
    credints: Int
}
dataEntity house {
    areaSqM : Int
    many tenantNames : String
    owner : Human
}

```

Figure 10.7: Entity DSL IDE test

The `test.entity` file is parsed correctly. All keywords are highlighted. Keywords/-names completion mechanism works perfectly. References between declaration and usages behave properly as well as renaming feature. References inspector underlined the word `Human`(line 24) correctly, in order to the rule described in grammar - `DataFeature` in `DataEntity` can be only of `DataType` type.

The result of Validation rules work can be also evaluated. The Features `children` and `subject` are highlighted yellow which corresponds to the rule `checkFeatureNameIfMany`. The Entity `Student` has more than 3 features and is therefore highlighted in red.

The entire testing process took no more than 5 minutes but covered the whole procedure of creating a programming environment for the new language. Under usual circumstances, the user might spend a week to learn how to implement language plugins and write all necessary files registering them in IntelliJ platform. And that's not even considering the possibility to define grammars with Xtext grammar language and use Ecore models in the DSL development which IntelliJ IDEA now provides.

Chapter 11

Conclusion

Summary

The primary task of the project was to design and implement a plugin for the IntelliJ IDEA IDE in order to streamline the work of DSL developers. The second important aspect of the thesis is that plugin is based upon an existing multifunctional software product called Xtext, which was developed for the popular Eclipse IDE. Accordingly, the resulting product is a type of "plugin-adapter" which allows the user of one IDE to use all of the functionality of a framework from a different IDE in the exact same manner.

Within this project, the author studied Programming Language Theory and formal description methods of programming languages. Additionally, the the principles of implementation of development environments were analyzed and helping tools like parser generators were discussed. The author utilized existing software models to avoid completely building the project from scratch. In particular, the IntelliJ platform was studied in details and comparing analysis with Eclipse IDE was performed. In fact, the author was engaged in metaprogramming since the developed plugin's main functionality is the creation of other IDEA plugins.

The project goals are considered accomplished, although a market-ready product is definitely not finished yet. Nevertheless, the plugin at the current stage will be instrumental to many other programmers working in this area of expertise. Among the benefits of the work are the author's significant advancement in programming skills and knowledge gained in the sphere of IDE plugin development and computer languages implementation.

Future plans and development

The implemented plugin lacks some features the Xtext framework in Eclipse provides and needs further development. First in line is the improvement of the Xtext language editor. For example, the inspection logic preventing the user from writing invalid grammar structures should be added. After that, Xtext instruments for generator modification, mapping to Java and testing will be transferred in IDEA.

Appendix A

Component diagram

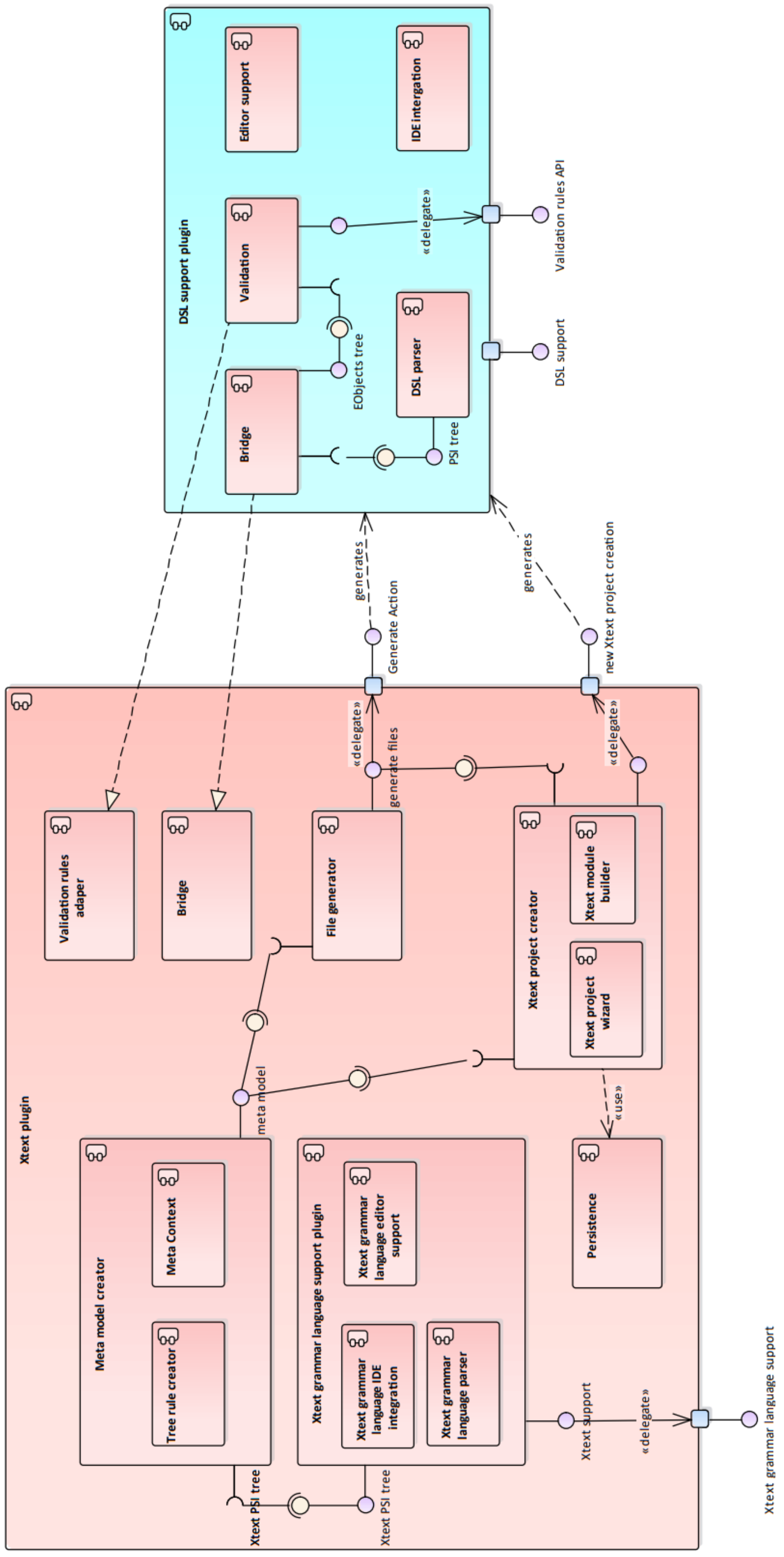


Figure A.1: The final component diagram

Appendix B

Electronic attachment description

The archive attached to the work contains two main elements:

1. **`XtextPlugin-0.0.1.zip`** archive.
   This archive is used as the IntelliJ IDEA plugin. The JAR `IntelliJ Platform Xtext Plugin-0.0.1.jar`, which contains all of the source code is located in it.

2. **`testing`** folder.
   This folder consists of `testingScenario.txt` file, which describes how to launch IDE with created plugin and test its functionality. The folder also includes testing resources.

B.1 Source code description

The source folder of the project contains the following packages:

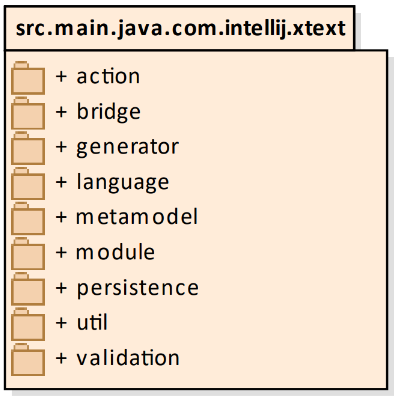


Figure B.1: The source directory of the project

- **`action`** - contains the "Generate fragments" action class.

- **`bridge`** - contains classes of the **`Bridge`** component.
- **`generator`** - contains classes of the **`File generator`** component.
- **`language`** - represents the **`Xtext language support plugin`** component.
- **`metamodel`** - contains classes of the **`Meta model creator`** component.
- **`module`** - contains classes of the **`Xtext project creator`** component.
- **`persistence`** - contains classes for persisting the project-related data.
- **`util`** - contains all helping classes.
- **`validation`** - contains classes of the **`Validation rules adapter`** component.

Bibliography

[1] D. Goswami, K. V. Krishna, *Formal Languages and Automata Theory*, 11.2010.

[2] P. Linz, *An Introduction to FORMAL LANGUAGES and AUTOMATA Fourth Edition*, 2006, ISBN 0763737984.

[3] A. Deursen, P. Klint, *Domain-Specific Language Design Requires Feature Descriptions*, 2002.

[4] M. Fowler, R. Parsons, *Domain-Specific Languages*, 2010, ISBN 9780321712943.

[5] R. Frost, R. Hafiz, P. Callaghan, *Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars*, 2007.

[6] T. Parr, *The Definitive ANTLR 4 Reference*, 2013, ISBN 9781934356999.

[7] L. Garshol, *BNF and EBNF: What are they and how do they work?*, 2008, https://www.garshol.priv.no/download/text/bnf.html, [online].

[8] R. Feynman, *EBNF: A Notation to Describe Syntax.*

[9] C. Trim, *The Art of Tokenization*, 2013.

[10] J. Jones, *Abstract Syntax Tree Implementation Idioms*, 2013.

[11] Eclipse Foundation Inc., *Eclipse*, 2020, https://www.eclipse.org/, [online].

[12] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, *EMF: Eclipse Modeling Framework*, 2008, ISBN 0321331885.

[13] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend. Second edition*, 2016, ISBN 9781786464965.

[14] Eclipse Foundation Inc., *Xtext*, 2020, https://www.eclipse.org/Xtext/index.html, [online].

[15] JetBrains s.r.o., *IntelliJ IDEA*, 2021, https://www.jetbrains.com/idea/, [online].

[16] Gradle Inc., *Gradle*, 2021, https://gradle.org/, [online].

[17] Sparx Systems, *Enterprise Architect*, 2013, https://sparxsystems.com/, [online].

[18] JetBrains s.r.o., *Grammar-Kit*, 2020, https://github.com/JetBrains/Grammar-Kit#readme, [online].

[19] JetBrains s.r.o., *IntelliJ Platform Plugin SDK*, 2020, https://plugins.jetbrains.com/docs/intellij/welcome.html, [online].

[20] D. Jemerov, S. Isakova, *Kotlin in Action*, 201ý, ISBN 9781617293290.

[21] Eclipse Foundation Inc., *Xtend*, 2020, https://www.eclipse.org/xtend/index.html, [online].

[22] G. Klein, *JFlex User's Manual*, 2001, http://www.di.uevora.pt/~pp/2002-s1/compil/jflex-man/manual.html, [online].