emftext

USER GUIDE

Contents

1	Ove	rview 1							
	1.1	Generation features							
	1.2	Specification features							
	1.3	Editor features							
	1.4	Other features							
2	Get	Setting Started with EMFText 3							
	2.1	Specifying a Language Metamodel							
	2.2	Specifying the Language's Concrete Syntax							
	2.3	Generating the Language Tooling							
		2.3.1 Generating Resource Plug-ins in Eclipse							
		2.3.2 Generating Resource Plug-ins with ANT							
	2.4	Optionally Customising the Language Tooling							
3	Con	crete Syntax Specification Language (CS)							
	3.1	Configuration Block							
		3.1.1 Required General Information							
		3.1.2 Importing other Metamodels and Syntax Specifications							
		3.1.3 Code Generation Options							
	3.2	Tokens							
		3.2.1 Defining Custom Tokens							
		3.2.2 Composed Tokens							
		3.2.3 Token Priorities							
	3.3	Token Styles							
	3.4	Syntax Rules							
		3.4.1 Simple Syntax							
		3.4.2 Syntax for EAttributes							
		3.4.3 Syntax for EReferences							
		3.4.4 Syntax for Printing Instructions							
		3.4.5 Syntax for Expressions							
		3.4.6 Overriding Imported Rules							
	3.5	Suppressing Warnings							
4	DSL	. Customization 25							
	4.1	Customization Techniques							
		4.1.1 Overriding Generated Artifacts							
		4.1.2 Overriding Meta Information Classes							
		4.1.3 Using Generated Extension Points							
	4.2	Concrete Customizations							

Contents

Bil	Bibliography 6						
	A3	Syntax	Dependent Artifacts	64			
	A2		of Warnings	63			
	A1	Code (Generation Options	41			
Α	Арр	endix		41			
		4.2.14	Working with Layout Information	37			
		4.2.13	Enabling Debug Support	37			
		4.2.12	Implementing Launch Support	37			
		4.2.11	Providing Custom Content for Projects created by the New Project Wizard	36			
		4.2.10	Providing Custom Content for Files created by the New File Wizard	36			
		4.2.9	Adding Context Menu Items to the Outline View	35			
		4.2.8	Customizing Code Completion Proposals	35			
		4.2.7	Customizing Text Hovers	34			
		4.2.6	Implementing Interpreters	33			
		4.2.5	Implementing Builders	32			
		4.2.4	Implementing Quick Fixes	31			
		4.2.3	Implementing Post Processors	29			
		4.2.2	Customizing Reference Resolving	28			
		4.2.1	Customizing Token Resolving	26			

1 Overview

EMFText is a tool for defining textual syntax for Ecore-based metamodels. It enables developers to define their own textual languages—be it domain specific languages (e.g., a language for describing forms) or general purpose languages (e.g., Java)—and generates accompanying tool support for these languages. It provides a domain specific language (DSL) for syntax specification from which it generates a full-fledged Eclipse editor and components to load and store model instances.

To give a quick overview, some of the most compelling features of EMFText are outlined in the following.

1.1 Generation features

EMFText uses a generative approach where all artifacts that form the tooling for a textual language are generated. This includes a parser for loading textual models, a printer for storing model instances and the editor with all its customizable components.

Generation of independent code The code that is generated by EMFText does not contain dependencies to EMFText and is fully customizable. This implies that generated language tooling can be deployed in environments where EMFText is not available and that future compatibility issues are completely avoided.

Generation of default syntaxes With EMFText, initial syntaxes for the textual DSL can be generated in one step for any Ecore-based metamodel. One can generate a syntax that conforms to the HUTN standard [Obj02], a Java-like syntax, or a custom syntax configured by using the custom syntax wizard. In all cases, the initial, generated specification of the syntax can be further tailored towards specific needs (cf. Section 2.2).

Highly customizable code generation EMFText provides many options for tailoring its code generation process to specific needs. For example, manually modified code can be preserved by disabling its generation or custom license headers can be provided if needed (cf. Appendix A1).

1.2 Specification features

EMFText comes with a simple but rich syntax specification language—the Concrete Syntax Specification Language (CS). It is based on EBNF and follows the concept of convention over configuration. This allows for very compact and intuitive syntax specifications, but still supports tweaking specifics where needed (cf. Chapter 3).

Modular specification EMFText provides an import mechanism that not only supports specification of a single text syntax for multiple related Ecore models, but also allows for modularization and extension of CS specifications (cf. Section 3.1.2).

- **Default reference resolving mechanisms** A default name resolution mechanism for models with globally unique names is available out of the box for any syntax. Also, external references are resolved automatically, if URIs are used to point to the referenced elements. More complex resolution mechanisms can be realized by implementing generated resolving methods (cf. Section 4.2.2).
- **Comprehensive syntax analysis** A number of analyses of CS specifications inform the developer about potential errors in the syntax—like missing syntax for certain metaclasses (cf. Appendix A2).

1.3 Editor features

Editors generated by EMFText provide many advanced features that are known from, e.g., the Eclipse Java editor. This includes code completion (with customizable completion proposals cf. Section 4.2.2 and Section 4.2.8), customizable syntax and occurrence highlighting via preference pages, advanced bracket handling, code folding, hyperlinks and text hovers for quick navigation, an outline view and instant error reporting.

1.4 Other features

EMFText provides numerous other interesting features, some of them outlined below.

- **ANT support** Dedicated ANT tasks are provided to allow the generation of text syntax plugins in build scripts (cf. Section 2.3.2).
- **Support for post processors** By default, registered post processors are called by the tooling after parsing. These post processors can be customized to check consistency of models or perform necessary modifications after parsing (cf. Section 4.2.3).
- **Generation of builder stubs** EMFText generates a builder stub that can be used to process model instances on changes and to automatically produce derived resources when needed (cf. Section 4.2.5).
- **Generation of interpreter stubs** Similarly, interpreters are used to execute model instances (cf. Section 4.2.6).
- **Quick fixes** Quick fixes provide actions that can automatically solve problems found during analysis of model instances. EMFText provides means to attach quick fixes to reported problems which then can be fixed by the developer in a convenient way (cf. Section 4.2.4).

2 Getting Started with EMFText

Generating an advanced Eclipse editor for a new language with EMFText just requires a few specifications and a generation step. Basically, a language specification for EMFText consists of a language metamodel and a concrete syntax specification. Taking these specifications the EMFText generator derives an advanced textual editor, that uses a likewise generated parser and printer to parse language expressions to EMF models or to print EMF models to languages expressions respectively.



Figure 2.1: Iterative EMFText language development process.

The basic language development process with EMFText is depicted in Fig. 2.1. It is an iterative process that can be passed several times and consists of the following basic tasks:

- (1) Specifying a Language Metamodel,
- (2) Specifying the Language's Concrete Syntax,
- (3) Generating the Language Tooling,
- (4) Optionally Customising the Language Tooling.

Each of the these tasks will be explained and exemplified in the subsequent sections:

To kick-start the development of a new language you can use the EMFText project wizard. Select File > New > Other... > EMFText Project. In the Wizard (cf. Fig. 2.2) you just enter the language name and EMFText will initialise a new EMFText Project containing a metamodel folder that holds an initial metamodel and syntax specification.

2.1 Specifying a Language Metamodel

As EMFText is tightly integrated with the Eclipse Modeling Framework (EMF) [SBPM08] language metamodels are specified using the Ecore Metamodelling Language. The metamodel



Figure 2.2: EMFText Project wizard.

specifies the abstract syntax of a new language. It can be build from classes with attributes that are related using references. References are further distinguished into containment references and non-containment references. It is important to notice this difference, as both reference types have different semantics in EMF and are also handled differently in EMFText. Containment references are used to relate a parent model element and a child model element that is declared in the context of the parent element. An example which can be found for instance in object-oriented programming languages is the declaration of a method within the body of a class declaration. Non-containment references are used to relate a model element with an element that is declared elsewhere (not as one of its children). An example for programming languages is a method call (declared in a statement in the body of a method declaration) that relates to the method that it calls using a non-containment reference. The referenced method, however, is declared elsewhere: In a class the method relates to with a containment reference.

Example. To define a metamodel for a language, we have to consider the concepts this language deals with, how they interrelate and what attributes they have. In the following we discuss the concepts of an exemplary language to specify forms and how they are represented in a forms metamodel.

- A Form (class) has a caption (attribute) and contains (containment reference) a number of question Groups (class).
- Each Group has a name (attribute) and contains (containment reference) a number of question Items (class).

- Each Item has a question text (attribute) and an explanation (attribute).
- There are various Types (class) of question items with regard to the answer values they expect: e.g., Text questions (subclass), Date questions (subclass), Number questions (subclass), Choices (subclass), or Decisions (subclass).
- Choices and Decisions declare (containment reference) a number of selection Options (class).
- There may be question Items that are dependent of (non-containment reference) the selection of a particular Option in another Item, e.g., a question that asks for the age of your children, only if you previously selected that you have some.

The subsequent listing depicts a textual representation of the according EMF metamodel. Besides the mapping of forms concepts to Ecore it also refines the multiplicities and types. A new text.ecore metamodel is created by selecting File > New > Other... > EMFText.text.ecore file. For a detailed introduction on the basics of Ecore metamodelling we refer to [SBPM08].

```
package forms // this is the package name
        forms // this is the namespace prefix
        "http://org.emftext/language/forms.ecore" // the namespace URI
        class Form {
                attribute EString caption (0..1);
                containment reference Group groups (1..-1) ;
        }
        class Group {
                attribute EString name (1..1);
                containment reference Item items (1..-1);
        }
        class Item {
                attribute EString text (0..1);
                attribute EString explanation (0..1);
                containment reference ItemType itemType (1..1);
                reference Option dependentOf (0..-1);
        }
        abstract class ItemType {}
        class FreeText extends ItemType {}
        class Date extends ItemType {}
        class Number extends ItemType {}
        class Choice extends ItemType {
                attribute EBoolean multiple (0..1);
```

Each Ecore metamodel is accompanied by an .genmodel. You can create the .genmodel by selecting File > New > Other... > EMF Generator Model. The generator model is used to configure various options for EMF code generation (e.g., the targeted Java runtime). From the root element of the .genmodel you can now start the generation of Java code implementing your metamodel specification. By default the generated files can be found in the src folder of the metamodel plug-in, but this can also be configured in the .genmodel. We suggest to change the code generation folder to src-gen to better separate generated code from hand-written.

2.2 Specifying the Language's Concrete Syntax

After defining a metamodel, we can start specifying our concrete syntax. The concrete syntax specification defines the textual representation of all metamodel concepts. For that purpose, EMFText provides the cs-language. As a starting point, EMFText provides a syntax generator that can automatically create a cs specification conforming to HUTN (Human-Useable Textual Notation) [Obj02] from the language metamodel. To manually specify the concrete syntax create a new syntax specification by selecting File > New > Other... > EMFText.cs file.

The listing at the end of this section depicts a syntax specification for the forms language. It consists of five sections:

- In the first section the language file extension is defined, the syntax specification is bound to the metamodel, and the syntax start symbol is defined.
- In the second section various EMFText code generation options can be configured.
- In the third section basic token types used by the language lexer to tokenise language expressions are defined. If no token definitions are given, default token types are used.
- In the fourth section token styles are defined that customise syntax highlighting for specific token types in the generated editor.
- In the fifth section the syntax rules for the language are specified.

The syntax specification rules used in the cs-language are derived from the EBNF syntax specification language to support arbitrary context-free languages. They are meant to define syntax for EMF-based metamodels and, thus, are specifically related to the Ecore metamodelling concepts. Therefore, it provides Ecore-specific specialisations of classical EBNF constructs like terminals, and non terminals. This specialisation enables EMFText to provide advanced support during syntax specification, e.g., errors and warnings if the syntax specifica-

tion is inconsistent with the metamodel. Furthermore, it enables the EMFText parser generator to derive an parser that directly instantiates EMF models from language expressions.

In the following we conclude the most important syntax specification constructs found in the cs-language and their relation to EBNF and Ecore metamodels. For an extensive overview on the syntax specification language we refer to Sect. 3. Each syntax construct is also related to examples taken from the listing at the end of this section.

Rule An cs rule is always related (by its name) to a specific class from the metamodel. It defines the syntactic representation of this metaclass, its attributes and references. All syntax rules are collected in the rules section of the cs file. Within syntax rules various constructs like keywords, syntax terminals, non-terminals, and EBNF operators as multiplicities (?, +, *), alternative (|), or rounded brackets to nest sub-rules can be used.

Examples:

```
Form ::= ...;, Group ::= ...;
```

Keywords Keywords are purely syntactic elements that are mainly used to structure and markup particular language expressions.

Examples:

```
"FORM", "GROUP", "ONLY" "IF"
```

Attribute Terminal Attribute terminals are used in rule bodies to specify the syntactic representation for attributes of the according meta class. They can be recognised by the attribute name that is followed by square brackets. Within these square brackets a token that specifies the syntax allowed for attribute values, or a prefix and a suffix that must surround attribute values can be given. If nothing is given a default text token is assumed.

Examples:

```
name[], mulitple[MULTIPLE], name['"','"']
```

Containment Reference Non-Terminals Containment reference non-terminals are used in rule bodies to specify the syntactic representation for containment references of the according metaclass. They use the reference name and are not followed by brackets. Containment reference non-terminals are derived from EBNF non-terminals, which means that during parsing the parser descends in the syntax rule specified for the class the containment reference points to. This is in line with the semantics of containment references as used in metamodels.

Examples:

groups, questions

Non-Containment Reference Terminals Non-containment reference terminals are used in rule bodies to specify the syntactic representation for non-containment references of the according metaclass. They use the reference name that is followed by square brackets. Within these square brackets a token can be given that specifies the syntax allowed for expressions in the concrete syntax that identify the element the non-containment reference relates to. This symbolic reference is later resolved to the actual element (cf. Sect. 4.2.2). If no token is given, again the default text token is used.

Examples:

dependentOf[]

Printing Markup Printing Markup is used to customise the behaviour of the generated printer. This is useful to achieve a particular standard layout for printed language expressions. Two forms of printing markup are supported:

- whitespace markup, prints a given number of whitespaces: #<n>
- linebreak markup, introduces a linebreak followed by a given number of tab characters:!<n>

```
SYNTAXDEF forms
FOR <http://www.emftext.org/language/forms>
START Form
OPTIONS {
        overrideBuilder = "false";
        additionalDependencies = "org.emftext.language.forms.generator";
}
TOKENS {
        DEFINE MULTIPLE $'multiple'|'MULTIPLE'$;
TOKENSTYLES {
        "TEXT" COLOR #da0000;
        "FORM" COLOR #000000, BOLD;
        "ITEM" COLOR #000000, BOLD;
        "CHOICE" COLOR #000000, BOLD;
        "ONLY" COLOR #da0000, BOLD;
        "IF" COLOR #da0000, BOLD;
        "DATE" COLOR #000000, BOLD;
        "FREETEXT" COLOR #000000, BOLD;
        "NUMBER" COLOR #000000, BOLD;
        "DECISION" COLOR #000000, BOLD;
        "GROUP" COLOR #000000, BOLD;
RULES {
        Form ::= "FORM" caption['"','"'] !1 groups*;
        Group ::= !0 "GROUP" name['"','"'] !0 items*;
        Item ::= "ITEM" text['"','"'] ( explanation['"','"'] )?
                                ("ONLY" "IF" dependentOf[])? ":" itemType !0;
        Choice ::= "CHOICE" (multiple[MULTIPLE])?
                                "(" options ("," options)* ")";
        Option ::= ( id[] ":")? text['"','"'];
        Date ::= "DATE";
        FreeText ::= "FREETEXT";
```

```
Number ::= "NUMBER";
Decision ::= "DECISION" "(" options "," options ")";
}
```

2.3 Generating the Language Tooling

Given a complete syntax specification the EMFText code generator can be used to derive an advanced textual editor and an accompanying customisable language infrastructure. There are two alternative ways to use the code generator: Manually within Eclipse or from an Apache Ant script.

2.3.1 Generating Resource Plug-ins in Eclipse

Manual code generation can be triggered from the context menu of the concrete syntax specification. Therefore, right click the cs file and select *Generate Text Resource*. This starts the EMF code generator that produces a number of plug-ins. Fig. 2.3 depicts the plug-ins generated for our exemplary forms language. In the following we shortly discuss their purpose:

- org.emftext.commons.antrl3_2_0 This project contributes the ANTLR parsing runtime that the generated parser for the forms language depends on. As EMFText is meant to be runtime free the ANTLR runtime is generated for every new language, if no runtime is found in the current workspace.
- org.emftext.language.forms This is the basic plug-in of the language. In the folder metamodel it contains the Ecore metamodel and the cs specification defined previously. The src-gen folder contains the Java-based implementation that was generated using the .genmodel. This plug-in will not be altered by the EMFText code generator.
- org.emftext.language.forms.resources.forms This is the plug-in that contains the generated parser, printer and various infrastructure for the forms language. The project contains two source folders (src and src-gen). The contents of src-gen is overridden by every run of EMFText code generation. It is, thus not meant to contain manually customised code. The contents of the src folder contains implementation classes that are meant for manual customisation. By default it only contains reference resolvers that are used to resolve symbolic names of non-containment references to the model element actually meant. For details on reference resolving we refer to Sect. 4.2.2. Various cs options are available to tailor what language implementation artifacts shall be customised and, therefore, put into the src folder. For a detailed discussion of such options we refer to Sect. 4.1.2 and Appendix A1.
 - Besides the files implementing the language tooling, a number of extension points specific for the language are generated to the **schema** folder. They can be used to further customise language tooling. For details we refer to Sect. 4.1.3.
- org.emftext.language.forms.resources.forms.ui This plug-in contains all generated classes related to the Eclipse-based User Interface (UI). Such separation of implementation classes belonging to the UI or not, enables the application of the language backend detached from the Eclipse UI.

```
Sorg.emftext.commons.antlr3_2_0
▶ # src-gen
 ► MIRE System Library [J2SE-1.5]
  ▶ ➡ Plug-in Dependencies
  ▶ META-INF
  metamodel
   n build.properties
   plugin.properties
   🙀 plugin.xml
▶ # src
 ▶ 进 src–gen
 ▶ MIRE System Library [J2SE-1.5]
  Plug-in Dependencies
  ▶ AMETA-INF
  ▶   schema
   n build.properties
   🙀 plugin.xml
▼ 📂 org.emftext.language.forms.resource.forms.ui
   ) src
  ▶ # src-gen
  ▶ March JRE System Library [J2SE-1.5]
 ▶ ➡ Plug-in Dependencies
 icons
  ▶ ETA-INF
   abuild.properties
   plugin.xml
```

Figure 2.3: Projects generated by EMFText to implement language tooling.

2.3.2 Generating Resource Plug-ins with ANT

A second way of starting the EMFText code generator is using Apache Ant scripts. Therefore EMFText contributes a number of tasks for Apache Ant, which are automatically registered to the Eclipse platform using the naming scheme: *emftext.taskName*. The following task are shipped with EMFText:

GenerateTextResource This task can be used to generate all language implementation plugins. The following listing exemplifies the application of this task and its obligatory parameters:

Further parameters are generateANTLRPlugin="[true|false]", which specifies whether the additional plug-in containing the ANTLR parsing runtime should be generated, and pre-processor="[qualified class name]" referring to an implementation of the org.emftext.sdk.-ant.SyntaxProcessor interface, which is provided for realising Java-based syntax specification preprocessors.

RegisterEcoreResourceFactory This task registers an Ecore model's resource factory for a certain type. This is especially useful for testing purposes without a running Eclipse platform. The following listing exemplifies its application:

RegisterURIMapping This task adds an URI mapping to the EMF URI map, which is useful for mapping symbolic namespace URIs to physical locations, i.e., for locating ecore models. The following listing exemplifies its application:

RemoveURIMapping This task removes an URI mapping from the EMF's URI map, which is useful for removing unwanted symbolic URI mappings from the URI map. The following listing exemplifies its application:

To execute an Ant script that uses EMFText tasks from within your Eclipse runtime, you have to adjust the script's run configuration. Therefore, select $Run > External\ Tools > External\ Tools\ Configurations...$ and select your Ant script's run configuration. In the JRE tab you have to activate the option Run in the same JRE as the workspace to make the EMFText tasks available to the script.

2.4 Optionally Customising the Language Tooling

The previous steps are mandatory to generate an initial implementation of basic tooling for your language. The generated text editor already comes with a number of advanced editing features that help editing language expressions a lot. However, there are various ways to make your language tooling more useful. EMFText helps you in customising your language tooling

with a number of additional functions ranging from semantic validation of language expressions, language compilation, language interpretation, or editor functions like folding, custom quickfixes, extended code completion, refactoring and more. To discover the full spectrum of possibilities please consider Sect. 4.

3 Concrete Syntax Specification Language (CS)

An EMFText syntax specification must be contained in a file with the extension .cs and consists of four main blocks:

- 1. A mandatory configuration block, which specifies the name of the syntax (i.e., the file extension), the generator model where to find the metaclasses, and the root metaclass (start symbol). Optionally, other syntaxes and metamodels can be imported and code generation options can be specified.
- 2. An (optional) TOKENS section. Here, token types like identifiers, numbers etc. for the lexical analyser can be specified.
- 3. An (optional) TOKENSTYLES section. Here, the default style (i.e., color and font style) for tokens and keywords can be specified.
- 4. A RULES section, which defines the syntax for concrete metaclasses.

In the following sections, these four main blocks will be explained in more detail.

3.1 Configuration Block

3.1.1 Required General Information

The first required piece of information is the file extension that shall be used for the files, which will contain your models:

SYNTAXDEF yourFileExtension

Note: The file extension must not contain the dot character.

Second, EMFText needs to know the EMF generator model (.genmodel) that contains the metaclasses for which the syntax is specified. EMFText does use the generator model rather than the Ecore model, because it requires information about the code generated from the Ecore model (e.g., the fully qualified names of the classes generated by the EMF). The genmodel can be referred to by its namespace URI:

FOR <vourGenModelNamespaceURI>

To find the generator model with the given namespace URI, EMFText tries to load it from the generator model registry. If it is not registered, EMFText looks for a .genmodel file with the same name as the syntax definition. For example, if the syntax specification is contained in a file yourdsl.cs, EMFText looks for a file called yourdsl.genmodel in the same folder.

If your genmodel is not contained in the same folder or is called differently from the syntax file name or if you do not want to use the one in the registry, the optional parameter yourGenmodelLocation can be used:

FOR <yourGenModelNamespaceURI> <yourGenmodelLocation>

The value of **yourGenmodelLocation** must be an URI pointing to the generator model. The URI can be absolute or relative to the syntax specification folder.

Third, the root element (start symbol) must be given. The root element must be a metaclass from the metamodel:

START YourRootMetaClassName

A CS specification can also have multiple root elements, which must be separated by a comma:

START RootMetaClass1, RootMetaClass2, RootMetaClass3

Typical candidates for root elements are metaclasses that do not have incoming containment edges.

Altogether a typical header for a .cs file looks something like:

```
SYNTAXDEF yourFileExtension
FOR <yourGenModelNamespaceURI> <yourGenmodelLocation>
START YourRootMetaClassName
```

3.1.2 Importing other Metamodels and Syntax Specifications

Sometimes it is required to import additional metamodels, e.g., if they are only referenced in the current one and a syntax for some or all of its concepts needs to be specified or reused. Metamodels and syntax specifications can be imported in a dedicated import section, which must follow after the start symbols:

```
IMPORTS {
    // imports go here
}
```

The list of imports must contain at least one entry. If no imports are needed the whole section must be left out. An import entry consists of a prefix, which can be used to refer to imported elements in rules, the metamodel namespace URI and optionally the name of a concrete syntax defined for that metamodel. If a syntax is imported, all its rules are reused and need not to be specified in the current **cs** specification. Importing syntax rules is optional. One can also just import the metamodel contained in the generator model.

```
prefix : <genModelURI> <locationOfTheGenmodel>
    // next line is optional
WITH SYNTAX syntaxURI <locationOfTheSyntax>
```

The two locations are again optional. For resolving the generator model the same rules as for the "main" generator model (declared after the FOR keyword) apply. For locating the syntax, EMFText looks up the registry of registered syntax specifications. If no registered syntax is found, locationOfTheSyntax is used to find the .cs file to import. Again, locationOfTheSyntax must be a relative or absolute URI.

3.1.3 Code Generation Options

EMFText's code generation can be configured using various options. These are specified in a dedicated optional OPTIONS section:

```
OPTIONS {
    // options go here in the following form:
    optionName = "optionValue";
}
```

The list of valid options and their documentation can be found in Appendix A1.

3.2 Tokens

EMFText allows to specify custom tokens. Each token type has a name and is defined by a regular expression. This expression is used to convert characters from the DSL files to form groups (i.e., tokens). Tokens are the smallest unit processed by the generated parser. By default, EMFText implicitly uses a set of predefined standard tokens, namingly:

```
TEXT: ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-')+,
LINEBREAK: ('\r\n'|'\r'|'\n'),
WHITESPACE: (''|'\t'|'\f').
```

The predefined tokens can be explicitly excluded by using the usePredefinedTokens option:

```
OPTIONS {
    usePredefinedTokens = "false";
}
```

3.2.1 Defining Custom Tokens

To define custom tokens, a TOKENS section must be added to the .cs file. This section has the following form:

```
TOKENS {
    // token definitions go here in the form:
    DEFINE YOUR_TOKEN_NAME $yourRegularExpression$;
}
```

Every token name has to start with a capital letter. A regular expression must conform to the ANTLRv3 syntax for regular expressions (without semantic annotations). However, don't worry: EMFText will complain if there is a problem with your regular expressions, such as typos or overlaps of regular expressions.

3.2.2 Composed Tokens

Sometimes, regular expressions are quite repetitive and one wants to reuse simple expressions to compose them to more complex ones. To do so, one can refer to other token definitions by their name. For example:

```
TOKENS {
    // simple token
    DEFINE CHAR $('a'...'z'|'A'...'Z')$;
    // simple token
    DEFINE DIGIT $('0'...'9')$;
    // composed token
    DEFINE IDENTIFIER CHAR + $($ + CHAR + $|$ + DIGIT + $)*$;
}
```

If token definitions are merely used as "helper" tokens, they can be tagged as FRAGMENT. This means the helper token itself is used in other token definitions, but not anywhere else in the syntax specification:

```
TOKENS {
    // simple token
    DEFINE CHAR $('a'...'z'|'A'...'Z')$;
    // helper token - not used on its own
    DEFINE FRAGMENT DIGIT $('0'...'9')$;
    // composed token
    DEFINE IDENTIFIER CHAR + $($ + CHAR + $|$ + DIGIT + $)*$;
}
```

The regular expressions are composed the same way strings are composed in Java programs. Therefore, make sure to put parenthesis around expressions where it is needed.

3.2.3 Token Priorities

EMFText does automatically sort token definitions. However, sometimes token definitions might be ambiguous (i.e., the regular expressions defined for two different tokens are not disjoint). In such cases EMFText will always prefer more specific tokens over more general tokens. That is, if one token definition includes another one, the latter is preferred over the former. If the automatic token sorting fails, EMFText will report an error. In this case one must turn off the automatic sorting using the <code>disableTokenSorting</code> option and sort the tokens manually. If automatic token sorting is turned off, one can give a higher priority to imported tokens by using the following directive:

```
TOKENS {
         PRIORITIZE NameOfImportedToken;
         DEFINE SOME_CUSTOM_TOKEN $someCustomRegularExpression$;
}
```

The PRIORITIZE directive can also be used with the predefined tokens TEXT, LINEBREAK and WHITESPACE.

3.3 Token Styles

To define the default syntax highlighting for a language, a special section TOKENSTYLES can be used. For each token or keyword the color and style (BOLD, ITALIC, STRIKETHROUGH, UNDERLINE) can be specified as follows:

```
TOKENSTYLES {
    // show YOUR_TOKEN in black
    "YOUR_TOKEN" COLOR #000000;
    // show keyword 'public' in red and bold font face
    "public" COLOR #FF0000, BOLD;
}
```

The default highlighting can still be customized at runtime by using the generated preference pages.

3.4 Syntax Rules

For each concrete metaclass you can define a syntax rule. The rule specifies what the text that represents instances of the class looks like. Rules have two sides—a left and right-hand side. The left side denotes the name of the meta class, while the right-hand side defines the syntax elements. If you have imported additional metamodels you can refer to their metaclasses using the prefix you've defined in the import statement. For example pre.MetaClassA refers to MetaClassA from the metamodel with the prefix pre.

3.4.1 Simple Syntax

The most basic form of a syntax rule is:

```
YourMetaClass ::= "someKeyword" ;
```

This rule states that whenever the text **someKeyword** is found, an instance of **YourMetaClass** must be created. Besides text elements that are expected "as is", parts of the syntax can be optional or repeating. For example the syntax rule:

```
YourMetaClassWithOptionalSyntax ::= ("#")? "someKeyword" ;
```

states that instances of YourMetaClassWithOptionalSyntax can be represented both by #someKeyword and someKeyword. Similar behavior can be defined using a star instead of a question mark. The syntax enclosed in the parenthesis can then be repeated. For example,

```
YourMetaClassWithRepeatingSyntax ::= ("#")* "someKeyword" ;
```

allows to represent instances of metaclass YourMetaClassWithRepeatingSyntax by writing someKeyword, #someKeyword, #someKeyword, or any other number of hash symbols followed by someKeyword. One can also use a plus sign instead of a star or question mark. In this case, the syntax enclosed in the parenthesis can be repeated, but must appear at least once.

3.4.2 Syntax for EAttributes

Syntax for EAttributes Having an Arbitrary Type

If metaclasses have attributes, we can also specify syntax for their values. To do so, simply add brackets after the name of the attribute:

```
YourMetaClassWithAttribute ::= yourAttribute[];
```

Optionally, one can specify the name of a token inside the brackets. For example:

```
YourMetaClassWithAttribute ::= yourAttribute[MY_TOKEN] ;
```

If the token name is omitted, as in the first example, EMFText uses the predefined token TEXT, which includes alphanumeric characters (see Sect. 3.2). The found text is automatically converted to the type of the attribute. If this conversion is not successfull, an error is raised when opening a file containing wrong syntax. For details on customizing the conversion of tokens, see Sect. 4.2.1.

Another possibility to specify the token definition that shall be used to match the text for the attribute value is do it inline. For example

```
YourMetaClassWithAttribute ::= yourAttribute['(',')'] ;
```

can be used to express that the text for the value of the attribute yourAttribute must be enclosed in parenthesis. Between the parenthesis arbitrary characters (except the closing parenthesis) are allowed. Other characters can be used as prefix and suffix here as well.

By default, the suffix character (in the example above this was the closing parenthesis) can not be part of the text for the attribute value. To allow this, an escape character needs to be supplied:

```
YourMetaClassWithAttribute ::= yourAttribute['(',')','\\'];
```

Here the backslash can be used inside the parenthesis to escape the closing parenthesis. It must then also be used to escape itself. That is, one must write two backslash characters to represent one.

To give an example on how escaping works, consider the following text: (text(more)). After parsing, this yields the attribute value text(more). The character sequence \) is replaced by). Note that the opening parenthesis does not need to be escaped.

Syntax for EAttributes of Type EBoolean

For boolean attributes, EMFText provides a special feature to ease syntax specification. All that is required is to give the two strings that represent true and false. To give an example consider the following syntax rule:

```
YourMetaClassWithAttribute ::= yourAttribute["yes" : "no"] ;
```

This rule states that **yes** represents the **true** value and **no** represents **false**. You can also use the empty string for one of the values:

```
YourMetaClassWithAttribute ::= yourAttribute["set" : ""] ;
```

This way, the attribute is set to false by default and set to true in the text set is found.

Syntax for EAttributes of Type EEnum

For enumeration attributes, EMFText does also provide a special feature to ease syntax specification. For each literal of the enumeration, the corresponding string representation must be given. For example, consider the following syntax rule:

```
YourMetaClassWithAttribute ::=
yourAttribute[red : "r", green : "g", blue : "b"];
```

This rule states that **r** represents the literal **red**, **g** represents the literal **green** and **b** represents the literal **blue**. The literals of the enumeration are identified by their name. You can also use the empty string for one of the values:

```
YourMetaClassWithAttribute ::=
yourAttribute[red : "r", green : "g", blue : ""] ;
```

This way, the attribute is set to blue by default.

3.4.3 Syntax for EReferences

Metaclasses can have references and consequently there is a way to specify syntax for these. EMF distinguishes between *containment* and *non-containment* references. In an EMF model, the elements that are referenced with the former type are contained in the parent elements. EMFText thus expects the text for the contained elements (children) to be also contained in the parent's text.

The latter (non-containment) references are referenced only and are contained in another (parent) element. Thus, EMFText does not expect text that represents the referenced element, but a symbolic identifier that refers to the element. This is very similar to the declaration and use of variables in Java. The declaration of a variable consists of the complete text that is required to describe a variable (e.g., its type). In contrast, when the variable is used at some other place it is simply referred to by its name. Non-containment references are similar to uses of variables.

Syntax for Containment References

A basic example for defining a rule for a meta class that has a containment reference looks like this:

```
YourContainerMetaClass ::= "CONTAINER" yourContainmentReference ;
```

It allows to represent instances of YourContainerMetaClass using the keyword CONTAINER followed by one instance of the type that yourContainmentReference points to. If multiple children need to be contained the following rule can be used:

```
YourContainerMetaClass ::= "CONTAINER" yourContainmentReference*;
```

In addition, each containment reference can be restricted to allow only certain types, for example:

```
YourContainerMetaClass ::= "CONTAINER"

yourContainmentReference : SubClass ;
```

does allow only instances of **SubClass** after the keyword **CONTAINER** even though the reference **yourContainmentReference** may have a more general type. One can also add multiple subclass restrictions, which must then be separated by a comma:

```
YourContainerMetaClass ::= "CONTAINER"

yourContainmentReference : SubClassA, SubClassB;
```

Syntax for Non-Containment References

A basic example for defining a rule for a metaclass that has a non-containment reference looks like this:

```
YourPointerMetaClass ::= "POINTER" yourNonContainmentReference[] ;
```

The rule is very similar to the one for containment references, but uses the additional brackets after the name of the reference. Within the brackets the token that the symbolic name must match can be defined. In the case above, the default token TEXT is used. Therefore, the syntax for an example instance of class YourPointerMetaClass can be POINTER a.

Since a is just a symbolic name that must be resolved to an actual model element, EMF-Text generates a Java class that resolves a to a target model element. This class be customized to specify how symbolic names are resolved to model elements. The default implementation of the resolver looks for all model elements that have the correct type (the type of yourNonContainmentReference) and that have a name or id attribute that matches the symbolic name. For details on how to customize the resolving of references, see Sect. 4.2.2.

3.4.4 Syntax for Printing Instructions

By default, EMFText can print all kinds of models. It does also preserve the layout of the textual representation when models are parsed and printed later on. However, to print models that have been created in memory, additional information can be passed to EMFText to cutomize the print result. This (optional) information includes the number of whitespaces and line breaks to be inserted between keywords, attribute values, references and contained elements. If you do not want to print models to text, printing instructions are not needed in your .cs file.

Syntax for Printing Whitespace

To explicitly print whitespace characters, the # operator can be used on the right side of syntax rules:

```
YourMetaclass ::= "keyword" #2 attribute[];
```

It is followed by a number that determines the number of whitespaces to be printed. In the example above, two whitespace characters are printed between the keyword and the attribute value.

Syntax for Printing Line Breaks

To explicitly print line breaks, the ! operator can be used on the right side of syntax rules:

```
YourMetaclass ::= "keyword" !0 attribute[];
```

It is followed by a number that determines the number of tab characters that shall be printed after the line break. In the example above, a line break is printed after **keyword**. The number of tabs refers to the current model element (i.e., **EObject**), which is printed. To print contained objects with an indendation of one tab, you can use a rule like this:

```
YourMetaclass ::= "keyword" "{" (!1 containmentRef)* !0 "}";
```

Here, the first line break operator (!1) makes sure that all the contained objects appear on a new line and that they are preceded by one tab character. The second line break operator (!0) tells EMFText to print the closing parenthesis (}) also on a new line, but without a leading tab.

3.4.5 Syntax for Expressions

When defining syntax for an expression language (e.g., arithmetic expressions) EMFText's standard mechanisms for specifying syntax can lead to structures that can not be optimally handled by an interpreter or evaluator. Furthermore, the underlying parser generator technology used by EMFText causes problems if left recursive rules are required to build an optimal expression tree, which is the case for all expression languages with left-associative binary operators (e.g., -). Therefore, EMFText provides a special feature called operator precendence annotations (@Operator). These annotations can be added to all rules, which refer to expression metaclasses with a common superclass. For example, the rule:

```
@Operator(type="binary_left_associative", weight="1", superclass="Expression")
Additive ::= left "+" right;
```

defines syntax for a metaclass Additive. The references left and right must be containment references and have the type Expression, which is the abstract supertype for all metaclasses of the expression metamodel.

The type attribute specifies the kind of expression at hand, which can be binary (either left_associative or right_associative), unary_prefix, unary_postfix or primitive.

The weight attribute specifies the priority of one expression type over another. For example, if a second rule:

```
@Operator(type="binary_left_associative", weight="2", superclass="Expression")
Multiplicative ::= left "*" right;
```

is present, EMFText will create an expression tree, where Multiplicative nodes are created last (i.e., multiplicative expressions take precedence over additive expressions).

Unary expressions can be defined as follows:

```
@Operator(type="unary_prefix", weight="4", superclass="Expression")
Negation ::= "-" body;
```

There is also the option to define unary_postfix rules.

Primitive expressions can be defined as follows:

```
@Operator(type="primitive", weight="5", superclass="Expression")
IntegerLiteralExp ::= intValue[INTEGER_LITERAL];
```

They should be used for literals (e.g., numbers, constants or variables).

One can certainly mix syntax rules that use the <code>@Operator</code> annotation with ones that do not in the same CS specification. However, one must be careful with the inheritance hierarchy in the metamodel in this case. All rules that use the <code>@Operator</code> annotation must refer to a metaclass that extends the metaclass specified with the <code>superclass</code> attribute. For subclasses of this superclass there must not be other non-<code>@Operator</code> rules. One could say that subtrees of the metaclass hierarchy must be either consistently specified as <code>@Operator</code> rules or not. Mixing is not possible.

For examples how to use @Operator annotations see the SimpleMath language in the EMF-Text Syntax Zoo¹ and the ThreeValuedLogic DSL². These do also come with an interpreter which shows how expression trees can be evaluated.

3.4.6 Overriding Imported Rules

EMFText supports to reuse syntax definitions partially by importing them and overriding rules. Rules can be redefined in the importing syntax by adding an <code>@Override</code> annotation to the overriding rule. You can also remove imported rules by using <code>@Override(remove="true")</code>.

Examples are:

```
// original rule in imported.cs
MetaClass ::= "originalSyntax";

// overriden rule in importing.cs
@Override
importPrefix.MetaClass ::= "newSyntax";

// use this to remove imported syntax rules
@Override(remove="true")
importPrefix.MetaClass ::= "does_not_matter";
```

Please replace **importPrefix** with the prefix that you have assigned to the imported syntax in the import statement (see Sect. 3.1.2).

http://www.emftext.org/language/simplemath

 $^{^2 \}verb|http://www.emftext.org/language/threevaluedlogic|$

3.5 Suppressing Warnings

To suppress warnings issued by EMFText in .cs files one can use the @SuppressWarnings annotation. This annotation can be added to rules, token definitions or complete syntax definitions. One can either suppress all warnings or just specific types. To suppress all warnings for a syntax use the following syntax:

```
@SuppressWarnings
YourMetaClass ::= "someKeyword";
```

A list of all warning types can be found in Appendix A2. For example, to suppress warnings about features without syntax, you may use:

```
@SuppressWarnings(featureWithoutSyntax)
YourMetaClassWithAttribute ::= "someKeyword";
```

4 DSL Customization

4.1 Customization Techniques

To adjust DSL plug-ins generated by EMFText to specific needs, there are three different customization techniques. Each of the subsequent sections describes one of them.

4.1.1 Overriding Generated Artifacts

The most simple way to customize generated artifacts is to tell EMFText that it must not override a specific class or file, which needs to be changed. For all artifacts that are generated by EMFText there is a **override** option, which can be set to **false** to preserve such manual changes (see Appendix A1 for a complete list). For example, to customize the hover text shown when the mouse arrow points at an element in the editor, the **overrideHoverTextProvider** must be set to **false**.

For all files that do not depend on the rules defined in the .cs file, this customization technique is fine. These files do not change, if new rules are added or existing ones are changed. Thus, manual changes will not cause conflicts if the syntax evolves. Only when EMFText is updated and the code generators are replaced, one may want to compare the manually adjusted files with the ones generated by the new EMFText version to see whether all customizations are still correct. This does particularly apply to generated manifest files and plug-in descriptors. A list of all classes that are syntax dependent can be found in Appendix A3.

4.1.2 Overriding Meta Information Classes

For all files that do depend on the rules defined in the .cs file, another customization technique is more appropriate. Instead of setting the override option to false for the artifact that needs to be changed, one can set the override option for the meta information classes to false.

Each of the two generated resource plug-ins contains a meta information class. These are called XyzMetaInformation and XyzUIMetaInformation. Both classes provide factory methods to create instances of some important classes (e.g., createParser() or createPrinter()). To customize these classes (e.g., the printer) one can change the create methods to return instances of subclasses of the original classes. By using subclasses instead of overriding the classes directly, one can regenerate the resource plug-ins and thereby obtain new up-to-date classes, but still make customizations by overriding individual methods.

4.1.3 Using Generated Extension Points

In addition to overriding generated classes—either directly or using the meta information factory methods—one can use the extension points that are generated by EMFText for all DSLs.

Currently EMFText generates two extension points for each DSL—default_load_options and additional_extension_parser.

The former can be used to customize how resources are loaded. For example, post processors can be registered which apply changes to the models that are created from their textual representation (see Sect. 4.2.3). Also, pre processors can be registered to process the input before it is actually passed to the parser. This is particularly useful to handle unicode characters (see the JaMoPP implementation¹ for an example how to use it).

The latter extension point can be used to register additional parsers which can handle a particular file extension. EMF on its own does map one file extension to one resource factory, but sometimes it is useful to have multiple resource types for the same file extension. An example for how to use this extension point can be found in the textual syntax for Ecore².

4.2 Concrete Customizations

4.2.1 Customizing Token Resolving

To create models from their textual representation, it is necessary to convert the plain text found in Domain-specific Language (DSL) documents to attribute values (i.e., data types). For example, if the string "123" is found in a text file and shall be used as value for an attribute which has type EInt, the string needs to be converted to an int. Basic conversions, such as the one just mentioned, are handled by the generated class XyzDefaultTokenResolver (assuming the file extension of your DSL is xyz). However, if you want to use custom data types in your metamodels, or if you need to customize the default conversion, there are two ways to change the conversion of text to data types.

Customizing TokenResolver Classes

The first option to customize the conversion of text, is to change the generated token resolver classes. EMFText generates one of these classes for each token that is defined in the .cs file. All classes end up in a package called analysis in the src folder of the generated resource plug-in.

Each token resolver class has two methods—resolve() and deResolve(). The first one is used to convert text to data types. The second one is used to perform the other way around. Consequently, resolve() is used when models are parsed, while deResolve() is used to print models to text.

The default implementation for both methods delegates calls to a default token resolver. However, this call can be replaced by custom code implementing different behavior. The code in the resolve() method must convert the text (given by the parameter lexem) to an object of the data type. This object must be set using result.setResolvedToken(). The deResolve() must implement the opposite behavior by returning a string representation of the object.

In the following a custom token resolver class is shown, which converts TEXT tokens to java.util.Date objects:

¹http://www.jamopp.org

 $^{^2} http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Ecore$

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EStructuralFeature;
import org.emftext.language.xyz.resource.xyz.IXyzTokenResolveResult;
import org.emftext.language.xyz.resource.xyz.IXyzTokenResolver;
public class XyzTEXTTokenResolver implements IXyzTokenResolver {
  private SimpleDateFormat format = new SimpleDateFormat("dd.MM.yyyy");
  public String deResolve(Object value, EStructuralFeature feature,
      EObject container) {
   return format.format(value);
  }
  public void resolve(String lexem, EStructuralFeature feature,
      IXyzTokenResolveResult result) {
    try {
      Date date = format.parse(lexem);
      result.setResolvedToken(date);
    } catch (ParseException e) {
      result.setErrorMessage(lexem + "_is_not_a_valid_date.");
   }
  }
  public void setOptions(Map<?,?> options) {
    // can be left empty
  }
}
```

The difference between this kind of customization and the one below, is that the implemented conversion is local w.r.t. the textual syntax of the DSL . If you have multiple syntax definitions for your DSL , each can use completely different algorithms to convert data types.

Customizing the EMF Data Type Handling

Alternatively, you can customize the data type handling that is built into EMF. To do so, you need to define a custom data type in the metamodel (e.g., JavaDate). Then, the instance type name must be set to the actual Java class, which shall be used to represent instances of the data type (e.g., java.util.Date). When running the EMF code generation, the FactoryImpl class

will contain two methods—createJavaDateFromString() and convertJavaDateToString(). These need to be customized similar to the token resolver class before.

The following code is a snippet from the XyzFactoryImpl class and shows how to implement the same behavior as above using EMF's own data type handling facilities.

```
private SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
public Date createJavaDateFromString(EDataType eDataType,
    String initialValue) {
 try {
    return format.parse(initialValue);
  } catch (ParseException e) {
    // ignore
  return (Date)super.createFromString(eDataType, initialValue);
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
 */
public String convertJavaDateToString(EDataType eDataType,
    Object instanceValue) {
 return format.format(instanceValue);
}
```

4.2.2 Customizing Reference Resolving

If metamodels expose non-containment references (i.e., EReferences where the containment attribute is set to false), EMFText needs to resolve these references. This basically means that symbolic identifiers, which are used to reference other EObjects must be replaced by actual references to the respective objects.

Thus, EMFText generates one reference resolver class for each non-containment reference that is found in the metamodel of your DSL and that is actually used in the concrete syntax definition. All reference resolver classes end up in a package called **analysis** in the **src** folder of the generated resource plug-in.

The default implementation delegates calls to the **DefaultResolverDelegate** class. This class uses the following strategy to find objects that are referenced by identifiers:

- 1. the resource is searched for objects that have the correct type (i.e., the type of the non-containment reference)
- 2. if the objects having the correct type have an ID attribute, or a name attribute, or a single attribute of type EString, the value of this attribute is compared to the symbolic identifier. If the identifier matches the value of the attribute, the object is considered to be referenced.
- 3. if no matching object is found and the symbolic identifier is a valid URI, EMFText tries to load the resource at the URI. If the resource contains a root object with the correct type, this object is assumed to be referenced.

In cases, where this default resolving strategy is not sufficient, you can customize the resolver classes by changing the bodies of the methods resolve() and deResolve(). These methods are similar to the ones generated for the token resolver classes (see Sect. 4.2.1). The first one is used to find the object referenced by an identifier. The second one does the opposite—it creates a symbolic identifier for a referenced object. Again, the former is used after parsing. The latter is called when printing models.

The resolve() method must call result.addMapping(identifier, object) to set the referenced object, if one is found. The deResolve() method can simply return the textual representation of the referenced object as string.

To enable code completion for references, the resolve() method must be extended to take care of the resolveFuzzy parameter. If this parameter is true, the resolver class is used for code completion and must add all referenceable object to the result. Thus, instead of checking, whether identifier actually references an object, resolve() can simply add all objects that have the correct type to the result by calling result.addMapping(). However, in this case, the first argument, which is passed to addMapping() should not be identifier, but rather the string representation of the object.

4.2.3 Implementing Post Processors

Another quite common customization task is to implement post processors. Post processors basically provide the possibility to modify the model that is created when text is parsed. This way one can add default elements which are not represented in the model's textual representation or normalize models if multiple concrete syntax is allowed for the same DSL concept.

Registered post processors are automatically called by the generated DSL tooling whenever a model is created from text. This does also include the case where the editor parses text in the background to show errors immediately. Thus, some special restrictions apply to post processors. First, post processors must be able to deal with partial models or explicitly abort their execution if errors (e.g., syntactical problems) have been detected beforehand. Second, post processors must be sufficiently fast to make sure that editing is not disturbed. In particular attaching large amounts of errors to resources can slow down Eclipse a lot. In such cases, it is recommended to attach a maximum number of errors only.

Post processors should not be used to solely implement semantic checks (i.e., to validate models). This should rather be done using the EMF Validation Framework as this allows checks to be available in all editors rather than a single one that was generated by EMFText.

To register a post processor for your DSL, the generated <code>default_load_options</code> extension point must be used. This extension point allows to register classes that provide default load options, which are used whenever resources are loaded by the generated DSL tooling. Such classes must implement the <code>IXyzOptionProvider</code> interface, which has one method—<code>getOptions()</code>. To register a post processor, this method must return a map that has an entry where the key is <code>IXyzOptions.RESOURCE_POSTPROCESSOR_PROVIDER</code> and the value is an instance of a class that implements the <code>IXyzResourcePostProcessorProvider</code> interface. The latter object is used by the generated DSL tooling to instantiate post processors by calling <code>getResourcePostProcessor()</code>.

To illustrate this procedure consider the case where you want to add some default model elements to all models that are created from text. To do so, you need a plugin.xml which registers the option provider using the following code snippet:

This plugin.xml can be part of a separate plug-in (i.e., it does not need to be part of the generated resource plug-ins). The respective post processor class can be as follows.

```
package org.emftext.language.xyz.post;
import java.util.Collections;
import java.util.Map;
import org.eclipse.emf.ecore.EObject;
import org.emftext.language.xyz.resource.xyz.IXyzOptionProvider;
import org.emftext.language.xyz.resource.xyz.IXyzOptions;
import org.emftext.language.xyz.resource.xyz.IXyzResourcePostProcessor;
import org.emftext.language.xyz.resource.xyz.IXyzResourcePostProcessorProvider;
import org.emftext.language.xyz.resource.xyz.mopp.XyzResource;
public class PostProcessorExample implements IXyzOptionProvider,
  IXyzResourcePostProcessorProvider,
  IXyzResourcePostProcessor {
  public Map<?, ?> getOptions() {
    return Collections.singletonMap(
      IXyzOptions.RESOURCE_POSTPROCESSOR_PROVIDER,
      this
    );
  }
```

```
public IXyzResourcePostProcessor getResourcePostProcessor() {
   return this;
}

public void process(XyzResource resource) {
   EObject root = resource.getContents().get(0);
   // perform model modifications here
}
```

One can see that this class implements all three interfaces that are required to register a post processor. The actual post processing must be implemented in the **process()** method. Here, the model can be modified in arbitrary ways. However, one must be aware that any modification will yield different textual representations when the model is printed.

4.2.4 Implementing Quick Fixes

If a problem is added to a resource (e.g., by a post processor, cf. Section 4.2.3), problem markers are automatically created in the editor. Markers are a convenient way to inspect the cause of the problem directly from the editor. By providing an instance of <code>IXyzQuickFix</code> while creating an <code>IXyzProblem</code>, actions are specified that can automatically solve the reported problem.

To implement a custom quick fix CustomQuickFix for a specific problem, XyzQuickFix must be subclassed. Normally, the context object (i.e., the object where the action is applied to) is provided as a parameter to the constructor of CustomQuickFix. The method applyChanges() performs the actual fix of the problem on the context object.

This context object is also passed to the constructor of XyzQuickFix along with a brief description of the quick fix and an image key that references an image for the quick fix. The image key is used by the XyzImageProvider and can reference images in multiple ways.

First, the key can be one of the standard Eclipse images. That is, the key can be a field from class org.eclipse.ui.ISharedImages, for example, IMG_ELCL_REMOVE. Second, the key can denote a path relative to the UI project. Or, the third option is to use an arbitrary URL as key. For example, the URL platform:/plugin/com.xyz.plugin/icons/xyzIcon.png indicates that the file xyzIcon.png is used for the quick fix.

In case a more sophisticated means for providing images is needed, the XyzImageProvider can be manually extended after the option overrideImageProvider has been set to false.

The following listing shows a simple quick fix, which removes a given element from the resource.

```
public class RemoveElementQuickFix extends XyzQuickFix
   implements IXyzQuickFix {
   private EObject objectToRemove;

public RemoveElementQuickFix(String message, EObject objectToRemove) {
    super(message, "IMG_ETOOL_DELETE", objectToRemove);
}
```

```
this.objectToRemove = objectToRemove;
}

@Override
public void applyChanges() {
    EcoreUtil.delete(objectToRemove);
}
```

4.2.5 Implementing Builders

To implement a custom builder for your DSL , you can basically set the code generation option overrideBuilder to false:

```
OPTIONS {
    overrideBuilder = "false";
}
```

After regenerating the resource plug-ins (see Sect. 2.3), you will find a new class XyzBuilder in the src folder of the generated resource plug-in (assuming the file extension of your DSL is xyz). If you face compilation errors, make sure to delete the XyzBuilder class from the src-gen folder.

The generated builder class contains two methods—isBuildingNeeded() and build(). The first one is called to let the builder decide, which resources need to be included in the build process. The default implementation returns false to avoid unnecessary loading of resources. To include all textual resources that contain models of your DSL, change the method to return true.

The second method is called whenever the content of a resource changes. You can implement arbitrary behavior here. Usually, builders create some kind of derived artifact, for example a transformed or compiled version of the DSL model. Since build() retrieves the resource as method parameter, you can easily access the contents of the resource. To save the derived artifact it is good practice to use the URI of the original resource to derive a new URI. This can for example be done by removing segments and adding new ones.

The following listing shows a simple builder, which copies the contents of the resource to a new resource without making any changes.

```
import java.io.IOException;
import java.util.Collection;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.resource.Resource;
```

```
import org.eclipse.emf.ecore.util.EcoreUtil;
import org.emftext.language.xyz.resource.xyz.IXyzBuilder;
public class XyzBuilder implements IXyzBuilder {
  public boolean isBuildingNeeded(URI uri) {
    return true:
  }
  public IStatus build(XyzResource resource, IProgressMonitor monitor) {
    // get contents and create copy
    EList<EObject> contents = resource.getContents();
    Collection<EObject> contentsCopy = EcoreUtil.copyAll(contents);
    // create new resource with different name
    URI newUri = URI.createURI("copy.xyz").resolve(resource.getURI());
    Resource newResource = resource.getResourceSet().createResource(newUri);
    // add copy of original content to new resource
    newResource.getContents().addAll(contentsCopy);
    // save new resource
    try {
      newResource.save(null);
    } catch (IOException e) {
      // handle exception
    return Status.OK_STATUS;
  }
}
```

Alternatively, you can also register builders for your DSL in other plug-ins.

4.2.6 Implementing Interpreters

To ease the implementation of interpreters for your DSL, EMFText generates an interpreter stub. Assuming the file extension of your DSL is xyz, the abstract stub class will be named AbstractXyzInterpreter. To implement concrete interpreters, you can create subclasses of this stub class.

For each metaclass found in the metamodel of your DSL, the interpreter stub contains a <code>interprete_Classname</code> method. These methods can be overridden in concrete interpreter classes to implement the desired interpretation for the objects of each type.

After implementing the methods for the classes which shall be interpreted, the interpreter can be used in different modes. First, models can be interpret using a stack. In this case, the interprete_Classname methods must perform the interpretation, but should not call other interprete methods. This is automatically performed by the interpreter. One can put objects

on the interpretation stack by calling addObjectToInterprete() and then start interpretation by calling interprete(). Interpretation ends when all objects from the stack are consumed.

Second, the interpretation can be performed without using the stack. In this case, the interprete_Classname methods call other interprete methods to continue interpretation. The traversal of the model is more explicit than using the interpreter with the stack in this mode.

The first, stack-based interpretation mode is useful to traverse models in a bottom-up fashion. One can simply put all models elements (using eallContents() on the model root element) on the stack and then start interpretation. The second, stack-independent interpretation mode is useful to traverse models top-down.

The interpreter stub class has two type parameters—ResultType and ContextType, which concrete subclasses must bind. The former parameter (i.e., ResultType) specifies the return type of the interprete methods. The latter parameter (i.e., ContextType) defines the type of the parameter that is passed to the interprete methods. By binding the type parameters one can use arbitrary classes to pass interpretation results.

Examples for interpreters can be found in the EMFText Syntax Zoo. Both SimpleMath³ and the ThreeValuedLogic DSL⁴ use the generated interpreter stubs.

4.2.7 Customizing Text Hovers

To implement custom text hovers for your DSL , basically set the code generation option overrideHoverTextProvider to false:

```
OPTIONS {
    overrideHoverTextProvider = "false";
}
```

After regenerating the resource plug-ins (see Sect. 2.3), a new class XyzHoverTextProvider can be found in the src folder of the generated resource UI plug-in (assuming the file extension of your DSL is xyz). If you face compilation errors, make sure to delete the XyzHoverTextProvider class from the src-gen folder.

The generated hover text provider class contains one method—getHoverText(). The default implementation of this method delegates calls to a default provider. To customize the hover text you can inspect the EObject passed to the method and return arbitrary HTML code. The following listing shows a simple customized provider, which returns the type of the EObject.

```
import org.eclipse.emf.ecore.EObject;
import org.emftext.language.xyz.resource.xyz.IXyzHoverTextProvider;

public class XyzHoverTextProvider implements IXyzHoverTextProvider {
    public String getHoverText(EObject object) {
        return "An_object_of_type_" + object.eClass().getName();
    }
}
```

³http://www.emftext.org/language/simplemath

 $^{^4}$ http://www.emftext.org/language/threevaluedlogic

```
}
}
```

4.2.8 Customizing Code Completion Proposals

The DSL tooling generated by EMFText does equip the DSL editor with default code completion facilities. If you find the completion proposals to be not sufficient, or you want to adjust them w.r.t. the text that is displayed for specific proposals or the icons that are shown, you can customize the proposals. To do so, set the overrideProposalPostProcessor option to false. After regenerating the resource plug-ins, you will find a class XyzProposalPostProcessor in the src folder of the UI resource plug-in. The class with the same name in the src-gen folder can then be deleted.

The default implementation of the only method in this class (i.e., process()) does return the list of proposals as they are. However, you can make arbitrary changes to this list. For example, you can remove proposals if you find them not useful or modify proposals if you want to change the displayed string or icon. You can also add new proposals if needed.

The proposals that are passed to the process() method provide information such as which structural feature they complete (getStructuralFeature()), which image they are associated with (getImage()) or which text is inserted if the respective proposal is selected by a user (getInsertString()). To modify proposals, new instances of the XyzCompletionProposal must be created, because this class is immutable.

4.2.9 Adding Context Menu Items to the Outline View

To add new items to the context menu of the generated outline view, you can use the standard mechanism to register menu items. All that is needed, is to register a menu item (using the org.eclipse.ui.menus extension point), a command that is created when the menu item is selected (using the org.eclipse.ui.commands extension point) and a handler that can process your command (using the org.eclipse.ui.handlers extension point). For example, the following fragment of a plugin.xml file registers a new menu item that can be invoked on objects of type org.emftext.language.xyz.XyzClass.

```
</extension>
<extension point="org.eclipse.ui.menus">
  <menuContribution
  locationURI="popup:org.emftext.language.xyz.resource.xyz.ui.outlinecontext?
           ....after=additions">
   <command
    commandId="org.emftext.language.xyz.resource.xyz.ui.command"
    label="Sample_action_registered_for_XYZ_outline_view">
    <visibleWhen checkEnabled="false">
      <iterate>
      <adapt type="org.emftext.language.xyz.XyzClass" />
      </iterate>
    </visibleWhen>
  </command>
  </menuContribution>
</extension>
```

Note that you will also need a class CommandHandler, which must implement the interface org.eclipse.core.commands.IHandler.

4.2.10 Providing Custom Content for Files created by the New File Wizard

By default, EMFText generates a New File Wizard for your DSL. The content of the file that is created by this wizard, is automatically obtained by creating a minimal model that conforms to your metamodel and by printing this minimal model. However, if you would like to provide a different content for the files that are created by this wizard, you can simply create a text file with a different content. Assuming your syntax definition is stored in a file called xyzsyntax.cs and the file extension of your syntax if xyz, this text file must be called xyzsyntax.newfile.xyz and be stored in the folder that contains your syntax definition. When generating the text resource plug-ins, EMFText will read the content of this file and embed its content in the generated code.

If you would like to provide dynamic content for the files that are created by the New File Wizard, you can also set the overrideNewFileContentProvider option to false and change the NewFileContentProvider class according to your needs.

4.2.11 Providing Custom Content for Projects created by the New Project Wizard

By default, EMFText generates a New Project Wizard for your DSL. The content of the project that is created by this wizard, is contained in the newProject.zip file located in the generated resource UI plug-in. When the user creates a new project with the wizard, the content of this file is extracted into the new project. By default, the newProject.zip contains only a placeholder file, which is replaced by a new file (see above) after extraction. To customize this behaviour, replace the generated newProject.zip with your own newProject.zip

that contains, for example, initial folder structures, example files or documentation. Custom newProject.zip files are not replaced by the generator.

If you would like to make adjustments to the New Project Wizard itself, you can set the **overrideNewProjectWizard** option to **false** and change the **NewProjectWizard** class according to your needs.

4.2.12 Implementing Launch Support

For executable DSLs it is convenient to provide users with support for launch configurations. EMFText does generate all the code that is required to obtain a custom launch configuration type and basic user interface to configure launch parameters. Also, shortcuts are registered that allow DSL users to execute (i.e., run or debug) DSL models directly from the project explorer or the active editor. Basically invoke Run as... or Debug as... on one of your DSL files to check out the default launch support.

To specify how your DSL is executed set the overrideLaunchConfigurationDelegate option to false and implement the launch method. The example code that is generated by EMFText shows how to obtain the root element of the model that shall be executed.

For an example on how to implement a launch configuration consider the SimpleMath language in the EMFText Syntax Zoo⁵. If you do not need launch support for your DSL and you want to get rid of the generated code for this feature, set the option disableLaunchSupport to true.

4.2.13 Enabling Debug Support

EMFText generates all the infrastructure code that is required to obtain a full fledged debugger for executable DSLs that use a stack-based interpreter. In Sect. 4.2.6, details on the implementation of such interpreters can be found. To enable debugging you can simply adjust the LaunchConfigurationDelegate class as described in Sect. 4.2.12 to tell EMFText which interpreter must be used when running or debugging DSL models. Here, the LaunchConfigurationHelper class provides a method launchInterpreter() that allows to run and debug custom interpreters.

The WebTest⁶ and the Pico⁷ language in the EMFText Syntax Zoo contain custom interpreters and launch configurations. They can serve as a starting point to see how debugging support can be realized.

4.2.14 Working with Layout Information

Layout information is the information that is usually discarded when text is parsed into a model. It usually includes whitespaces, linebreaks and often comments. That is, all the tokens (cf. Section 3.2) that are not used in grammar rules.

⁵http://www.emftext.org/language/simplemath

 $^{^6}$ http://www.emftext.org/language/webtest

⁷http://www.emftext.org/language/pico

By default, layout information is collected by EMFText in the background and used during printing to restore the original layout of model elements. However, sometimes it is also useful to have access to this information in the model and, if applicable, modify this information. This can for example be used to access and modify comments.

For this purpose, EMFText offers a dedicated layout metamodel layout.ecore (with the namespace http://www.emftext.org/commons/layout). It is located in the plugin org.emftext. commons.layout. If you import the layout metamodel in your metamodel, you can obtain access to layout information. For this, you need to add references with multiplicity * to the EClass LayoutInformation to each of your EClasses. (The preferred way of doing this is to have a comment super class for all your EClasses that contains this reference.) EMFText will then fill this reference with layout information during parsing. An example for using the layout metamodel is JaMoPP's Java metamodel.⁸

If you do not use layout information and want to discard it during parsing (to optmise memory usage) you may use the following option:

IXyzOptions.DISABLE_LAYOUT_INFORMATION_RECORDING

⁸http://www.jamopp.org

List of Figures

2.1	Iterative EMFText language development process	3
2.2	EMFText Project wizard	4
2.3	Projects generated by EMFText to implement language tooling	10

A Appendix

A1 Code Generation Options

EMFText currently supports 293 code generation options. However, most of them (250) are only used to specify which generated artifacts shall be customized. Subsequently, a list of all options and their description can be found.

additionalDependencies

A list of comma separated plug-in IDs, which will be added to the manifest of the generated resource plug-in. The default value for this option is an empty list.

additionalExports

A list of comma separated packages, which will be added as exports to the manifest of the generated resource plug-in. The default value for this option is an empty list.

additionalImportedPackages

A list of comma separated packages, which will be added as imported packages to the manifest of the generated resource plug-in. The default value for this option is an empty list.

additionalLibraries

A list of comma separated libraries, which will be added to the manifest file, the .classpath file and the build.properties file of the generated resource plug-in. The default value for this option is an empty list.

additionalUIDependencies

A list of comma separated plug-in IDs, which will be added to the manifest of the generated resource UI plug-in. The default value for this option is an empty list.

additionalUIExports

A list of comma separated packages, which will be added as exports to the manifest of the generated resource UI plug-in. The default value for this option is an empty list.

additionalUIImportedPackages

A list of comma separated packages, which will be added as imported packages to the manifest of the generated resource UI plug-in. The default value for this option is an empty list.

antlrPluginID

Sets the ID for the generated common ANTLR runtime plug-in. The default value for this option is org.emftext.commons.antlr3_4_0.

autofixSimpleLeftrecursion

If set to true, EMFText will try to fix rules that contain simple left recursion. The default value for this option is false. This is a non-standard option, which might be removed in future releases of EMFText.

backtracking

If set to false, the ANTLR-backtracking is deactivated for parser generation. The default value for this option is true.

basePackage

The name of the base package EMFText shall store the generated classes or the resource plug-in in. If this option is not set, the default value is determined by adding the suffix resource.FILE_EXTENSION to the base package of the generator model.

baseResourcePlugin

The plug-in containing the resource implementation for the DSL (if different from the generated resource plug-in). By default this option is not set, which means that the generated resource plug-in provides the resource implementation.

defaultTokenName

This option can be used to specify the name of the token that is used when no token is given for attributes or non-containment references in syntax rules. Declarations like featureX[] in CS rules will automatically be expanded to featureX[TOKEN_Y] if the value of this option is TOKEN_Y. The default value for this option is TEXT, which makes the predefined token TEXT the default token.

disableBuilder

If set to **true**, the builder that is generated and registered by default will not be registered anymore. The default value for this option is **false**.

disableDebugSupport

If set to **true**, code that is required to support debugging of DSL models is not generated. The default value for this option is **false**.

disableEMFValidationConstraints

If set to **true**, constraint validation using the EMF Validation Framework is disabled. The default value for this option is **false**.

disableEValidators

If set to false, constraint validation using registered EValidators will be enabled. The default value for this option is true.

disableLaunchSupport

If set to **true**, code that is required to support launching of DSL models is not generated. The default value for this option is **false**.

disableNewProjectWizard

If set to true, the new NewProjectWizard is not registered. Use this option if there is already an existing wizard. The default value for this option is false.

disableTokenSorting

Disables the automatic sorting of tokens. The default value for this option is false. editorName

The default editor name can be changed with this option. Default name is 'EMFText <yourSyntaxExtension> Editor'

forceEOF

If set to false, EMFText will generate a parser that does not expect an EOF signal at the end of the input stream. The default value for this option is true.

generateCodeFromGeneratorModel

If set to true, EMFText automatically generates the model code using the generator model referenced in the CS specification. The default value for this option is false.

generateTestAction

If set to true, EMFText generates a UI action that can be used to test parsing and printing of files containing textual syntax. The default value for this option is false. This is a non-standard option, which might be removed in future releases of EMFText.

generateUIPlugin

If set to false, EMFText will not generate the resource UI plug-in. The default value for this option is true.

ignoreTypeRestrictionsForPrinting

If set to true, the modern printer will not consider type restrictions for containment references that are defined in CS specifications. The classic printer does ignore these restrictions in any case. The default value for this option is false.

licenceHeader

A URI pointing to a text file that contains a header which shall be added to all generated Java files. This option is useful to include copyright statements in the generated classes. If this option is not set, a default (empty) header is added to all generated Java classes.

memoize

If set to false, the ANTLR-memoize is deactivated for parser generation. The default value for this option is true.

overrideAbstractDebuggable

If set to **false**, the AbstractDebuggable class will not be overridden. The default value for this option is **true**.

overrideAbstractExpectedElement

If set to **false**, the AbstractExpectedElement class will not be overridden. The default value for this option is **true**.

overrideAbstractInterpreter

If set to **false**, the AbstractInterpreter class will not be overridden. The default value for this option is **true**.

overrideAbstractOutlinePageAction

If set to false, the AbstractOutlinePageAction class will not be overridden. The default value for this option is true.

override A dapter Factory

If set to false, the AdapterFactory class will not be overridden. The default value for this option is true.

override Additional Extension Parser Extension Point Schema

If set to **false**, the extension point schema for additional parsers is not overridden. The default value for this option is **true**.

overrideAnnotationModel

If set to false, the AnnotationModel class will not be overridden. The default value for this option is true.

overrideAnnotationModelFactory

If set to false, AnnotationModelFactory class will not be overridden. The default value for this option is true.

overrideAntlrPlugin

If set to false, no ANTLR common runtime plug-in is generated. The default value for this option is true.

overrideAntlrTextToken

If set to false, the ANTLRTextToken class will not be overridden. The default value for this option is true.

overrideAntlrTokenHelper

If set to false, the AntlrTokenHelper class will not be overridden. The default value for this option is true.

overrideAttributeValueProvider

If set to false, the AttributeValueProvider class will not be overridden. The default value for this option is true.

overrideBooleanTerminal

If set to false, the BooleanTerminal class will not be overridden. The default value for this option is true.

overrideBracketInformationProvider

If set to false, the BracketInformationProvider class will not be overridden. The default value for this option is true.

overrideBracketPreferencePage

If set to false, the BracketPreferencePage class will not be overridden. The default value for this option is true.

overrideBracketSet

If set to false, the BracketSet class will not be overridden. The default value for this option is true.

overrideBrowserInformationControl

If set to false, the BrowserInformationControl class will not be overridden. The default value for this option is true.

overrideBuildProperties

If set to false, the build properties file will not be overridden. The default value for this option is true.

overrideBuilder

If set to false, the Builder class will not be overridden. The default value for this option is true.

overrideBuilderAdapter

If set to false, the BuilderAdapter class will not be overridden. The default value for this option is true.

overrideCardinality

If set to false, the Cardinality class will not be overridden. The default value for this option is true.

overrideCastUtil

If set to false, the CastUtil class will not be overridden. The default value for this option is true.

overrideChangeReferenceQuickFix

If set to **false**, the ChangeReferenceQuickFix class will not be overridden. The default value for this option is **true**.

overrideChoice

If set to false, the Choice class will not be overridden. The default value for this option is true.

overrideClasspath

If set to false, the .classpath file of the resource plug-in will not be overridden. The default value for this option is true.

override Code Completion Helper

If set to false, the CodeCompletionHelper class will not be overridden. The default value for this option is true.

overrideCodeFoldingManager

If set to **false**, the CodeFoldingManager class will not be overridden. The default value for this option is **true**.

overrideColorManager

If set to false, the ColorManager class will not be overridden. The default value for this option is true.

overrideCompletionProcessor

If set to **false**, the CompletionProcessor class will not be overridden. The default value for this option is **true**.

overrideCompletionProposal

If set to false, the CompletionProposal class will not be overridden. The default value for this option is true.

overrideCompound

If set to false, the Compound class will not be overridden. The default value for this option is true.

overrideContainedFeature

If set to false, the ContainedFeature class will not be overridden. The default value for this option is true.

overrideContainment

If set to false, the Containment class will not be overridden. The default value for this option is true.

overrideContainmentTrace

If set to false, the ContainmentTrace class will not be overridden. The default value for this option is true.

overrideContextDependentURIFragment

If set to false, the ContextDependentUriFragment class will not be overridden. The default value for this option is true.

overrideContextDependentURIFragmentFactory

If set to false, the ContextDependentUriFragmentFactory class will not be overridden. The default value for this option is true.

overrideCopiedEList

If set to false, the CopiedEList class will not be overridden. The default value for this option is true.

overrideCopiedEObjectInternalEList

If set to false, the CopiedEObjectInternalEList class will not be overridden. The default value for this option is true.

override Debug Communication Helper

If set to false, the DebugCommunicationHandler class will not be overridden. The default value for this option is true.

overrideDebugElement

If set to false, the DebugElement class will not be overridden. The default value for this option is true.

overrideDebugMessage

If set to false, the DebugMessage class will not be overridden. The default value for this option is true.

overrideDebugModelPresentation

If set to false, the DebugModelPresentation class will not be overridden. The default value for this option is true.

overrideDebugProcess

If set to false, the DebugProcess class will not be overridden. The default value for this option is true.

overrideDebugProxy

If set to false, the DebugProxy class will not be overridden. The default value for this option is true.

overrideDebugTarget

If set to false, the DebugTarget class will not be overridden. The default value for this option is true.

overrideDebugThread

If set to false, the DebugThread class will not be overridden. The default value for this option is true.

overrideDebugValue

If set to false, the DebugValue class will not be overridden. The default value for this option is true.

overrideDebugVariable

If set to false, the DebugVariable class will not be overridden. The default value for this option is true.

overrideDebuggableInterpreter

If set to false, the DebuggableInterpreter class will not be overridden. The default value for this option is true.

overrideDebuggerListener

If set to **false**, the DebuggerListener class will not be overridden. The default value for this option is **true**.

overrideDefaultHoverTextProvider

If set to false, the DefaultHoverTextProvider class will not be overridden. The default value for this option is true.

overrideDefaultLoadOptionsExtensionPointSchema

If set to **false**, the extension point schema for default load options is not overridden. The default value for this option is **true**.

overrideDefaultNameProvider

If set to false, the DefaultNameProvider class will not be overridden. The default value for this option is true.

overrideDefaultResolverDelegate

If set to **false**, the default resolver class will not be overridden. The default value for this option is **true**.

overrideDefaultTokenResolver

If set to false, the DefaultTokenResolver class will not be overridden. The default value for this option is true.

overrideDelegatingResolveResult

If set to false, the DelegatingResolveResult class will not be overridden. The default value for this option is true.

$override Dev {\tt NullLocation} {\tt Map}$

If set to **false**, the DevNullLocationMap class will not be overridden. The default value for this option is **true**.

override Doc Browser Information Control Input

If set to false, the DocBrowserInformationControlInput class will not be overridden. The default value for this option is true.

overrideDummyEObject

If set to **false**, the DummyEObject class will not be overridden. The default value for this option is **true**.

overrideDynamicTokenStyler

If set to false, the DynamicTokenStyler class will not be overridden. The default value for this option is true.

overrideEClassUtil

If set to false, the EClassUtil class will not be overridden. The default value for this option is true.

overrideEDebugMessageTypes

If set to false, the EDebugMessage enumeration will not be overridden. The default value for this option is true.

overrideEObjectSelection

If set to false, the EObjectSelection class will not be overridden. The default value for this option is true.

overrideEObjectUtil

If set to false, the EObjectUtil class will not be overridden. The default value for this option is true.

overrideEProblemSeverity

If set to false, the EProblemSeverity class will not be overridden. The default value for this option is true.

overrideEProblemType

If set to false, the EProblemType class will not be overridden. The default value for this option is true.

overrideEclipeProxy

If set to false, the EclipseProxy class will not be overridden. The default value for this option is true.

overrideEditor

If set to false, the Editor class will not be overridden. The default value for this option is true.

overrideEditorConfiguration

If set to false, the EditorConfiguration class will not be overridden. The default value for this option is true.

overrideElementMapping

If set to false, the ElementMapping class will not be overridden. The default value for this option is true.

overrideEnumerationTerminal

If set to false, the EnumerationTerminal class will not be overridden. The default value for this option is true.

overrideExpectationConstants

If set to false, the ExpectationConstants class will not be overridden. The default value for this option is true.

overrideExpectedBooleanTerminal

If set to **false**, the ExpectedBooleanTerminal class will not be overridden. The default value for this option is **true**.

overrideExpectedCsString

If set to false, the ExpectedCsString class will not be overridden. The default value for this option is true.

overrideExpectedEnumerationTerminal

If set to false, the ExpectedEnumerationTerminal class will not be overridden. The default value for this option is true.

overrideExpectedStructuralFeature

If set to false, the ExpectedStructuralFeature class will not be overridden. The default value for this option is true.

overrideExpectedTerminal

If set to false, the ExpectedTerminal class will not be overridden. The default value for this option is true.

override Folding Information Provider

If set to false, the FoldingInformationProvider class will not be overridden. The default value for this option is true.

overrideFollowSetProvider

If set to false, the FollowSetProvider class will not be overridden. The default value for this option is true.

overrideFormattingElement

If set to false, the FormattingElement class will not be overridden. The default value for this option is true.

overrideFuzzyResolveResult

If set to **false**, the FuzzyResolveResult class will not be overridden. The default value for this option is **true**.

overrideGrammarInformationProvider

If set to false, the GrammarInformationProvider class will not be overridden. The default value for this option is true.

overrideHTMLPrinter

If set to false, the HtmlPrinter class will not be overridden. The default value for this option is true.

overrideHighlighting

If set to false, the Highlighting class will not be overridden. The default value for this option is true.

overrideHoverTextProvider

If set to false, the HoverTextProvider class will not be overridden. The default value for this option is true.

overrideHyperlink

If set to false, the Hyperlink class will not be overridden. The default value for this option is true.

overrideHyperlinkDetector

If set to false, the HyperlinkDetector class will not be overridden. The default value for this option is true.

overrideIAnnotationModelProvider

If set to false, the IAnnotationModelProvider class will not be overridden. The default value for this option is true.

overrideIBackgroundParsingListener

If set to false, the IBackgroundParsingListener class will not be overridden. The default value for this option is true.

overrideIBracketHandler

If set to false, the IBracketHandler class will not be overridden. The default value for this option is true.

overrideIBracketHandlerProvider

If set to false, the IBracketHandlerProvider class will not be overridden. The default value for this option is true.

overrideIBracketPair

If set to **false**, the IBracketPair class will not be overridden. The default value for this option is **true**.

overrideIBuilder

If set to false, the IBuilder class will not be overridden. The default value for this option is true.

overrideICommand

If set to false, the ICommand class will not be overridden. The default value for this option is true.

overrideIConfigurable

If set to false, the IConfigurable class will not be overridden. The default value for this option is true.

overrideIContextDependentURIFragment

If set to false, the IContextDependentUriFragment class will not be overridden. The default value for this option is true.

overrideIContextDependentURIFragmentFactory

If set to false, the IContextDependentUriFragmentFactory class will not be overridden. The default value for this option is true.

overrideIDebugEventListener

If set to false, the IDebugEventListener interface will not be overridden. The default value for this option is true.

overrideIDelegatingReferenceResolver

If set to false, the IDelegatingReferenceResolver interface will not be overridden. The default value for this option is true.

overrideIElementMapping

If set to false, the IElementMapping class will not be overridden. The default value for this option is true.

overrideIExpectedElement

If set to false, the IExpectedElement class will not be overridden. The default value for this option is true.

overrideIFunction1

If set to false, the IFunction1 interface will not be overridden. The default value for this option is true.

overrideIHoverTextProvider

If set to false, the IHoverTextProvider class will not be overridden. The default value for this option is true.

overrideIInputStreamProcessorProvider

If set to false, the IInputStreamProcessorProvider class will not be overridden. The default value for this option is true.

overrideIInterpreterListener

If set to false, the IInterpreterListener interface will not be overridden. The default value for this option is true.

overrideILocationMap

If set to false, the ILocationMap class will not be overridden. The default value for this option is true.

overrideIMetaInformation

If set to **false**, the IMetaInformation class will not be overridden. The default value for this option is **true**.

overrideINameProvider

If set to false, the INameProvider class will not be overridden. The default value for this option is true.

overrideIOptionProvider

If set to false, the IOptionProvider class will not be overridden. The default value for this option is true.

overrideIOptions

If set to false, the IOptions class will not be overridden. The default value for this option is true.

overrideIParseResult

If set to false, the IParseResult class will not be overridden. The default value for this option is true.

overrideIProblem

If set to false, the IProblem class will not be overridden. The default value for this option is true.

overrideIQuickFix

If set to false, the IQuickFix class will not be overridden. The default value for this option is true.

overrideIReferenceCache

If set to false, the IReferenceCache class will not be overridden. The default value for this option is true.

overrideIReferenceMapping

If set to false, the IReferenceMapping class will not be overridden. The default value for this option is true.

overrideIReferenceResolveResult

If set to false, the IReferenceResolveResult class will not be overridden. The default value for this option is true.

overrideIReferenceResolver

If set to **false**, the IReferenceResolver class will not be overridden. The default value for this option is **true**.

overrideIReferenceResolverSwitch

If set to false, the IReferenceResolverSwitch class will not be overridden. The default value for this option is true.

overrideIResourcePostProcessor

If set to false, the IResourcePostProcessor class will not be overridden. The default value for this option is true.

overrideIResourcePostProcessorProvider

If set to false, the IResourcePostProcessorProvider class will not be overridden. The default value for this option is true.

overrideIResourceProvider

If set to false, the IResourceProvider class will not be overridden. The default value for this option is true.

overrideITextDiagnostic

If set to false, the ITextDiagnostic class will not be overridden. The default value for this option is true.

overrideITextParser

If set to false, the ITextParser class will not be overridden. The default value for this option is true.

overrideITextPrinter

If set to false, the ITextPrinter class will not be overridden. The default value for this option is true.

overrideITextResource

If set to false, the ITextResource class will not be overridden. The default value for this option is true.

overrideITextResourcePluginPart

If set to false, the ITextResourcePluginPart class will not be overridden. The default value for this option is true.

overrideITextScanner

If set to false, the ITextScanner class will not be overridden. The default value for this option is true.

overrideITextToken

If set to false, the ITextToken class will not be overridden. The default value for this option is true.

overrideITokenResolveResult

If set to false, the ITokenResolveResult class will not be overridden. The default value for this option is true.

overrideITokenResolver

If set to false, the ITokenResolver class will not be overridden. The default value for this option is true.

overrideITokenResolverFactory

If set to false, the ITokenResolverFactory class will not be overridden. The default value for this option is true.

overrideITokenStyle

If set to false, the ITokenStyle class will not be overridden. The default value for this option is true.

overrideIURIMapping

If set to false, the IUriMapping class will not be overridden. The default value for this option is true.

overrideIgnoredWordsFilter

If set to **false**, the IgnoredWordsFilter class will not be overridden. The default value for this option is **true**.

overrideImageProvider

If set to false, the ImageProvider class will not be overridden. The default value for this option is true.

overrideInputStreamProcessor

If set to false, the InputStreamProcessor class will not be overridden. The default value for this option is true.

overrideKeyword

If set to false, the Keyword class will not be overridden. The default value for this option is true.

$over {\tt rideLaunchConfigurationDelegate}$

If set to false, the LaunchConfigurationDelegate class will not be overridden. The default value for this option is true.

overrideLaunchConfigurationHelper

If set to false, the LaunchConfigurationHelper class will not be overridden. The default value for this option is true.

$override Launch Configuration {\tt MainTab}$

If set to false, the LaunchConfigurationMainTab class will not be overridden. The default value for this option is true.

overrideLaunchConfigurationTabGroup

If set to false, the LaunchConfigurationTabGroup class will not be overridden. The default value for this option is true.

overrideLaunchShortcut

If set to false, the LaunchShortcurt class will not be overridden. The default value for this option is true.

overrideLayoutInformation

If set to false, the LayoutInformation class will not be overridden. The default value for this option is true.

overrideLayoutInformationAdapter

If set to false, the LayoutInformationAdapter class will not be overridden. The default value for this option is true.

overrideLayoutUtil

If set to false, the LayoutUtil class will not be overridden. The default value for this option is true.

overrideLineBreak

If set to false, the LineBreak class will not be overridden. The default value for this option is true.

overrideLineBreakpoint

If set to false, the LinebreakPoint class will not be overridden. The default value for this option is true.

overrideLineBreakpointAdapter

If set to false, the LineBreakpointAdapter class will not be overridden. The default value for this option is true.

overrideListUtil

If set to false, the ListUtil class will not be overridden. The default value for this option is true.

overrideLocationMap

If set to false, the LocationMap class will not be overridden. The default value for this option is true.

overrideManifest

If set to **false**, the manifest of the resource plug-in will not be overridden. The default value for this option is **true**.

overrideMapUtil

If set to false, the MapUtil class will not be overridden. The default value for this option is true.

overrideMarkerAnnotation

If set to false, the MarkerAnnotation class will not be overridden. The default value for this option is true.

overrideMarkerHelper

If set to false, the MarkerHelper class will not be overridden. The default value for this option is true.

overrideMarkerResolutionGenerator

If set to false, the MarkerResolutionGenerator class will not be overridden. The default value for this option is true.

overrideMetaInformation

If set to false, the MetaInformation class will not be overridden. The default value for this option is true.

overrideMinimalModelHelper

If set to false, the MinimalModelHelper class will not be overridden. The default value for this option is true.

overrideNature

If set to false, the Nature class will not be overridden. The default value for this option is true

overrideNewFileContentProvider

If set to false, the NewFileContentProvider class will not be overridden. The default value for this option is true.

overrideNewFileWizard

If set to false, the new file wizard class will not be overridden. The default value for this option is true.

overrideNewFileWizardPage

If set to false, the NewFileWizardPage class will not be overridden. The default value for this option is true.

overrideNewProjectWizard

If set to false, the NewProjectWizard class will not be overridden. The default value for this option is true.

overrideOccurrence

If set to false, the Occurrence class will not be overridden. The default value for this option is true.

overrideOccurrencePreferencePage

If set to false, the OccurrencePreferencePage class will not be overridden. The default value for this option is true.

overrideOptionProvider

If set to false, the OptionProvider class will not be overridden. The default value for this option is true.

overrideOutlinePage

If set to false, the OutlinePage class will not be overridden. The default value for this option is true.

overrideOutlinePageActionProvider

If set to false, the OutlinePageActionProvider class will not be overridden. The default value for this option is true.

override Outline Page Auto Expand Action

If set to false, the OutlinePageAutoExpandAction class will not be overridden. The default value for this option is true.

overrideOutlinePageCollapseAllAction

If set to false, the OutlinePageCollapseAllAction class will not be overridden. The default value for this option is true.

overrideOutlinePageExpandAllAction

If set to false, the OutlinePageExpandAllAction class will not be overridden. The default value for this option is true.

overrideOutlinePageGroupTypesAction

If set to false, the OutlinePageGroupTypesAction class will not be overridden. The default value for this option is true.

override Outline Page Lexical Sorting Action

If set to false, the OutlinePageLexicalSortingAction class will not be overridden. The default value for this option is true.

overrideOutlinePageLinkWithEditorAction

If set to false, the OutlinePageLinkWithEditorAction class will not be overridden. The default value for this option is true.

overrideOutlinePageTreeViewer

If set to false, the OutlinePageTreeViewer class will not be overridden. The default value for this option is true.

overrideOutlinePageTreeViewerComparator

If set to false, the OutlinePageTreeViewerComparator class will not be overridden. The default value for this option is true.

overridePair

If set to false, the Pair class will not be overridden. The default value for this option is true.

overrideParseResult

If set to false, the ParseResult class will not be overridden. The default value for this option is true.

overrideParser

If set to false, the Parser class will not be overridden. The default value for this option is true.

overrideParsingStrategy

If set to false, the ParsingStrategy class will not be overridden. The default value for this option is true.

overridePixelConverter

If set to false, the PixelConverter class will not be overridden. The default value for this option is true.

overridePlaceholder

If set to false, the Placeholder class will not be overridden. The default value for this option is true.

overridePluginActivator

If set to false, the PluginActivator class will not be overridden. The default value for this option is true.

overridePluginXML

If set to **true**, the plugin.xml file will be overridden. The default value for this option is **true**.

overridePositionCategory

If set to false, the PositionCategory class will not be overridden. The default value for this option is true.

overridePositionHelper

If set to false, the PositionHelper class will not be overridden. The default value for this option is true.

overridePreferenceConstants

If set to false, the PreferenceConstants class will not be overridden. The default value for this option is true.

overridePreferenceInitializer

If set to **false**, the PreferenceInitializer class will not be overridden. The default value for this option is **true**.

overridePreferencePage

If set to **false**, the PreferencePage class will not be overridden. The default value for this option is **true**.

overridePrinter

If set to false, the printer will not be overridden. The default value for this option is true.

overridePrinter2

If set to false, the Printer2 class will not be overridden. The default value for this option is true.

overrideProblemClass

If set to false, the problem class will not be overridden. The default value for this option is true.

overrideProjectFile

If set to **false**, the .project file of the resource plug-in will not be overridden. The default value for this option is **true**.

overridePropertySheetPage

If set to false, the PropertySheetPage class will not be overridden. The default value for this option is true.

overridePropertyTester

If set to false, the PropertyTester class will not be overridden. The default value for this option is true.

overrideProposalPostProcessor

If set to false, the ProposalPostProcessor class will not be overridden. The default value for this option is true.

overrideQuickAssistAssistant

If set to false, the QuickAssistAssistant class will not be overridden. The default value for this option is true.

overrideQuickAssistProcessor

If set to false, the QuickAssistProcessor class will not be overridden. The default value for this option is true.

overrideQuickFix

If set to false, the QuickFix class will not be overridden. The default value for this option is true.

overrideReferenceCache

If set to false, the ReferenceCache class will not be overridden. The default value for this option is true.

overrideReferenceResolveResult

If set to false, the ReferenceResolveResult class will not be overridden. The default value for this option is true.

overrideReferenceResolverSwitch

If set to false, the reference resolver switch will not be overridden. The default value for this option is true.

overrideReferenceResolvers

If set to **true**, the reference resolver classes will be overridden. The default value for this option is **false**.

overrideResourceFactory

If set to false, the resource factory class will not be overridden. The default value for this option is true.

overrideResourceFactoryDelegator

If set to false, the ResourceFactoryDelegator class will not be overridden. The default value for this option is true.

overrideResourcePostProcessor

If set to false, the ResourcePostProcessor class will not be overridden. The default value for this option is true.

overrideResourceUtil

If set to false, the ResourceUtil class will not be overridden. The default value for this option is true.

overrideRule

If set to false, the Rule class will not be overridden. The default value for this option is true.

overrideRuntimeUtil

If set to false, the RuntimeUtil class will not be overridden. The default value for this option is true.

overrideScanner

If set to false, the Scanner class will not be overridden. The default value for this option is true.

overrideSequence

If set to false, the Sequence class will not be overridden. The default value for this option is true.

overrideSourceLocator

If set to false, the SourceLocator class will not be overridden. The default value for this option is true.

overrideSourceLookupParticipant

If set to **false**, the SourceLookupParticipant class will not be overridden. The default value for this option is **true**.

overrideSourcePathComputerDelegate

If set to false, the SourcePathComputerDelegate class will not be overridden. The default value for this option is true.

overrideSourceViewerConfiguration

If set to false, the SourceViewerConfiguration class will not be overridden. The default value for this option is true.

overrideStackFrame

If set to false, the StackFrame class will not be overridden. The default value for this option is true.

overrideStreamUtil

If set to false, the StreamUtil class will not be overridden. The default value for this option is true.

overrideStringUtil

If set to false, the StringUtil class will not be overridden. The default value for this option is true.

overrideSyntaxColoringHelper

If set to false, the SyntaxColoringHelper class will not be overridden. The default value for this option is true.

overrideSyntaxColoringPreferencePage

If set to false, the SyntaxColoringPreferencePage class will not be overridden. The default value for this option is true.

overrideSyntaxCoverageInformationProvider

If set to false, the SyntaxCoverageInformationProvider class will not be overridden. The default value for this option is true.

overrideSyntaxElement

If set to false, the SyntaxElement class will not be overridden. The default value for this option is true.

overrideSyntaxElementDecorator

If set to false, the SyntaxElementDecorator class will not be overridden. The default value for this option is true.

overrideTaskItem

If set to false, the TaskItem class will not be overridden. The default value for this option is true.

overrideTaskItemBuilder

If set to false, the TaskItemBuilder class will not be overridden. The default value for this option is true.

overrideTaskItemDetector

If set to false, the TaskItemDetector class will not be overridden. The default value for this option is true.

overrideTerminal

If set to false, the Terminal class will not be overridden. The default value for this option is true.

overrideTerminateParsingException

If set to false, the TerminateParsingException class will not be overridden. The default value for this option is true.

overrideTextHover

If set to false, the TextHover class will not be overridden. The default value for this option is true.

overrideTextResource

If set to false, the text resource class will not be overridden. The default value for this option is true.

overrideTextResourceUtil

If set to false, the TextResourceUtil class will not be overridden. The default value for this option is true.

overrideTextToken

If set to false, the TextToken class will not be overridden. The default value for this option is true.

overrideTokenResolveResult

If set to false, the TokenResolveResult class will not be overridden. The default value for this option is true.

overrideTokenResolverFactory

If set to false, the token resolver factory class will not be overridden. The default value for this option is true.

overrideTokenResolvers

If set to true, the token resolver classes will be overridden. The default value for this option is false.

overrideTokenScanner

If set to false, the TokenScanner class will not be overridden. The default value for this option is true.

overrideTokenStyle

If set to false, the TokenStyle class will not be overridden. The default value for this option is true.

override Token Style Information Provider

If set to false, the TokenStyleInformationProvider class will not be overridden. The default value for this option is true.

overrideUIAntlrTokenHelper

If set to false, the AntlrTokenHelper class in the UI plug-in will not be overridden. The default value for this option is true.

overrideUIBuildProperties

If set to false, the build properties file of the resource UI plug-in will not be overridden. The default value for this option is true.

overrideUIDotClasspath

If set to false, the .classpath file of the resource UI plug-in will not be overridden. The default value for this option is true.

overrideUIDotProject

If set to false, the .project file of the resource UI plug-in will not be overridden. The default value for this option is true.

overrideUIManifest

If set to false, the manifest of the resource UI plug-in will not be overridden. The default value for this option is true.

overrideUIMetaInformation

If set to **false**, the MetaInformation class of the resource UI plug-in will not be overridden. The default value for this option is **true**.

overrideUIPluginActivator

If set to false, the plug-in activator class of the resource UI plug-in will not be overridden. The default value for this option is true.

overrideUIPluginXML

If set to **false**, the plugin.xml file of the resource UI plug-in will not be overridden. The default value for this option is **true**.

overrideURIMapping

If set to false, the UriMapping class will not be overridden. The default value for this option is true.

overrideURIUtil

If set to false, the URIUtil class will not be overridden. The default value for this option is true.

override Unexpected Content Type Exception

If set to false, the UnexpectedContentTypeException class will not be overridden. The default value for this option is true.

overrideUnicodeConverter

If set to false, the UnicodeConverter class will not be overridden. The default value for this option is true.

overrideWhiteSpace

If set to false, the WhiteSpace class will not be overridden. The default value for this option is true.

parserGenerator

The name of the parser generator to use. The default value for this option is antlr, which is also the only valid value. This is a non-standard option, which might be removed in future releases of EMFText.

reloadGeneratorModel

If set to true, EMFText reloads the generator model before loading it. This is particular useful, when the meta model (i.e., the Ecore file) is changing a lot during language development. The default value for this option is false.

removeEclipseDependentCode

If set to **true**, the generated resource plug-in will not contain any dependencies to Eclipse. The default value for this option is **false**.

resolveProxyElementsAfterParsing

If set to false, the generated resource class will not resolve references after parsing. The default value for this option is true.

resourcePluginID

The ID of the generated resource plug-in. The resource plug-in is stored in a folder that is equal to this ID.

resourceUIPluginID

The ID of the generated resource UI plug-in. The resource UI plug-in is stored in a folder that is equal to this ID.

saveChangedResourcesOnly

If set to true, the generated EMF resource will save only resource when their content (text) has actually changed. The default value for this option is false.

srcFolder

The name of the folder where EMFText shall store the customizable classes of the resource plug-in in. All classes for which the **override** option is set to **false** will be stored in this folder.

srcGenFolder

The name of the folder where EMFText shall store the generated classes of the resource plug-in in. All classes for which the **override** option is set to **true** will be stored in this folder.

tokenspace

The (numerical) value of this option defines how many whitespace should be printed between tokens if no whitespace information is given in CS rules. This option should

only be used with the classic printer. The default value of this option is 1 if the classic printer is used (see option useClassicPrinter) and automatic otherwise.

uiBasePackage

The package where to store all classes of the resource UI plug-in in. If this option is not set, the default value is determined by adding the suffix resource.FILE_EXTENSION.ui to the base package of the generator model.

uiSrcFolder

The name of the folder where EMFText shall store the customizable classes of the resource UI plug-in in. All classes for which the **override** option is set to **false** will be stored in this folder.

uiSrcGenFolder

The name of the folder EMFText shall store the generated classes of the resource UI plug-in in. All classes for which the **override** option is set to **true** will be stored in this folder.

useClassicPrinter

If set to true, the classic printer (i.e., the one used before EMFText 1.3.0) will be used. Otherwise the new printer implementation is used. In any case both printers are generated, but only one is used. The default value for this option is false.

usePredefinedTokens

If set to false, EMFText does not automatically provide predefined tokens (TEXT, WHITESPACE, LINEBREAK). The default value for this option is true.

A2 Types of Warnings

- abstractSyntaxHasStartSymbols
- collectInTokenUsedInRule
- duplicateOptionWithSameValue
- duplicateTokenStyle
- explicitSyntaxChoice
- featureWithoutSyntax
- leftRecursiveRule
- licenceHeaderNotFound
- maxOccurenceMismatch
- minOccurenceMismatch
- multipleFeatureUse
- noRuleForMetaClass
- nonContainmentOpposite
- nonStandardOption
- oppositeFeatureWithoutSyntax
- optionalKeyword

- referenceToAbstractClassWithoutConcreteSubtypesInAbstractSyntax
- styleReferenceToNonExistingToken
- tokenOverlapping
- tokenPriorizationUselessWhenTokenSortingEnabled
- unreachableRule
- unusedResolverClass
- unusedToken

A3 Syntax Dependent Artifacts

The following artifacts depend on the .cs specification. Overriding them does therefore require special care.

- ANTLR grammar
- AbstractInterpreter
- BracketInformationProvider
- Builder
- BuilderAdapter
- EclipseProxy
- FoldingInformationProvider
- GrammarInformationProvider
- IBuilder
- MarkerHelper
- MetaInformation
- Nature
- NewFileContentProvider
- Plugin
- Printer
- Printer2
- ReferenceResolverSwitch
- Resource
- ResourceFactoryDelegator
- ResourceUtil
- RuntimeUtil
- ScannerlessParser
- SyntaxCoverageInformationProvider
- TaskItemBuilder
- TextResourceUtil
- TokenResolverFactory
- TokenStyleInformationProvider

Bibliography

- [Obj02] Object Management Group. Human Usable Textual Notation (HUTN) Specification. Final Adopted Specification ptc/02-12-01, 2002.
- [SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework, 2nd Edition*. Pearson Education, 2008.