# DSL Development That Does Not Hurt

### Development of a data type specification language

Author: Antoine Logean

## Abstract

## The development of a textual external domain specific language with Xtext …

## Introduction

Model driven engineering is often wrongly associated with the use of the Unified Modeling Language (UML). If UML is a defacto standard as modeling language, there are many cases where it is not well suited. Most of the time its use is forced by the management that believes that the use of some tooling’s or standard notations will lead to a better design. This is a big mistake. Indeed UML is certainly as complex as a big and cryptic programming language. Most of the time somebody without experience will get lost in the tooling and notations, forgetting the most important: the design of a business specific model. Worst is the fact that in many cases the model produced is not formal and will be not "executed". UML should only be used as high level sketching language that help to communicate a design. It should not be used as formal modeling languages.

So how can we define our model so that it will have i) a high abstraction level (platform independent) and ii) a formal aspect (can be interpreted by a computer)? A very elegant way is to develop a dedicated programming language, a so called Domain Specific Language (DSL). A few years ago the technical development of a DSL was a very difficult and time consuming task. Today there are emerging frameworks that make possible the development of DSLs in a couple of hours. I will show in this article how by using Xtext from openArchitectureWare (oAW) you can implement your own business driven programming language. We will first design and implement the grammar of the language, generate the DSL tooling (the DSL-parser and DSL-editor) and finally writte a DSL generator.

## Terminology

Prior to start we have to define some important concepts. (see figure 1) that we will use along this article.

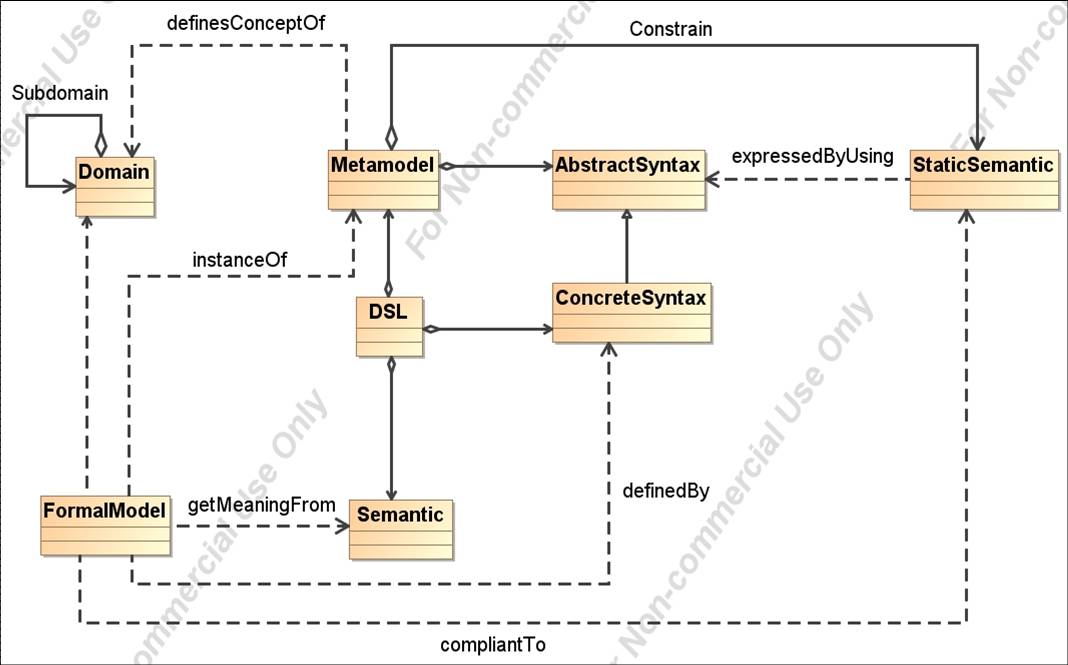


Figure 1: Overview of important concepts associated with the development of DSLs

By opposition to general purpose language (GPL) like Java, a DSL focus on one specific **domain**. Domains can be business driven (private banking, aircraft,...) or technical driven (representing a view point a an software architecture). Our aim is to describe formally a domain. The formal description of a domain is called **metamodel**. A metamodel is made of an **abstract syntax** and of a **static semantic**: the abstract syntax defines the different concepts and their relations. The static semantic defines the rules (constrains) that a model has to fulfill in order to be well formed. They are based on the abstract syntax and are thus dependant of them. In Java for example are these constraints are used by the compiler to give error messages like "variable have to be declared" or "casting error". The **concrete syntax** describes how the language look like: for example the concept of Class is represented in UML as a rectangle and in Java as the key world Class. The DSL programs are written by using this concrete syntax. There is a parser or an “instantiator” that reads the concrete syntax and instantiate the corresponding classes of the abstract syntax (the metamodel). The following processing of the DSL program is made on this object network (also called parser tree).

A DSL can be seen as a programming language for a specific domain. This means that any concept of the language should have a precise business meaning or **semantic**.

A **formal model** is simply a program that is written with the DSL. Formal means here that it can be read or executed by a computer with the help of an interpreter or a code generator.

## Development Process

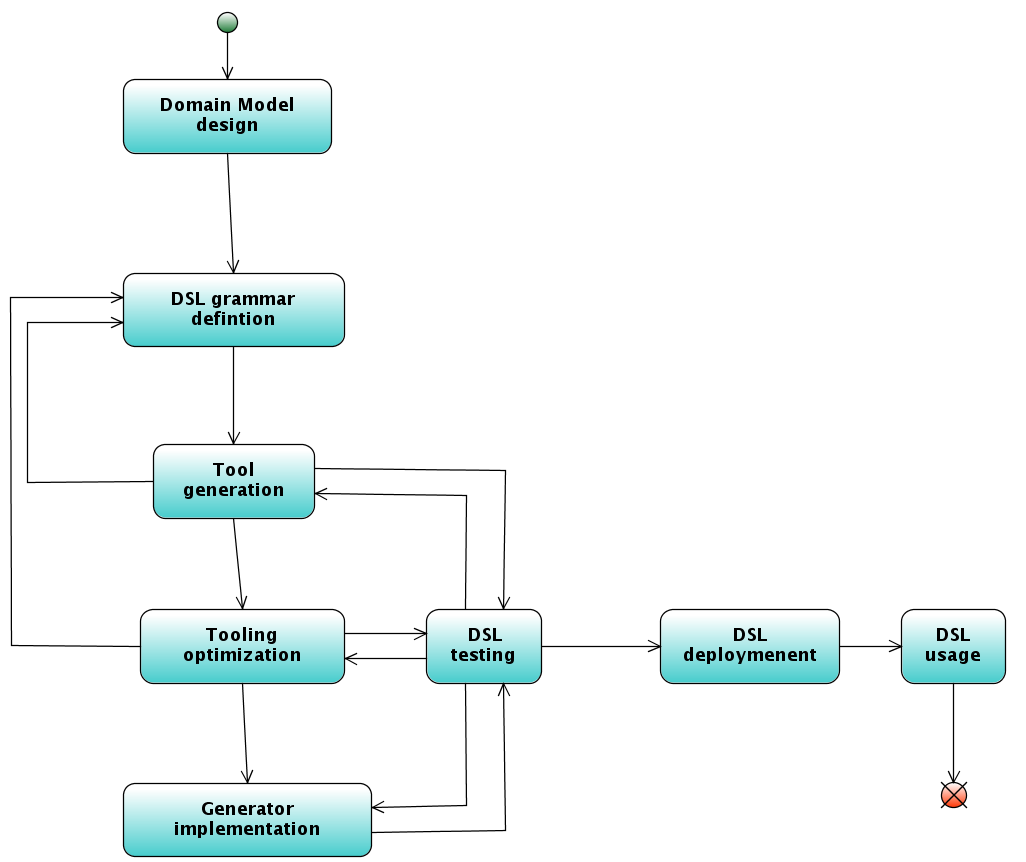
**

Figure 2: different steps involved in the development of a DSL with Xtext.

1. Starting from a domain model one has to define the grammar of the language
2. From this grammar generates Xtext an EMF-based metamodel, a DSL parser, and a DSL editor
3. The tooling is then optimized: addition of Check constrains, optimization of the DSL editor (code completion, outline view, syntax highlighting, ...)
4. Development of the DSL generator
5. Testing and deployment of the DSL.

Let us have a look more closely to each step

## Domain Model Design

Prior to start the development of a DSL we need to define the domain model. This will be the base of the abstract syntax of our DSL. This part is not specific to the development of a DSL. This is a very important step where the business logic has to be specify.

The problem domain we address in this article is the specification of business specific datatypes. The DSL will be used by domain architects. Figure 2 shows a class diagram representing this domain model.

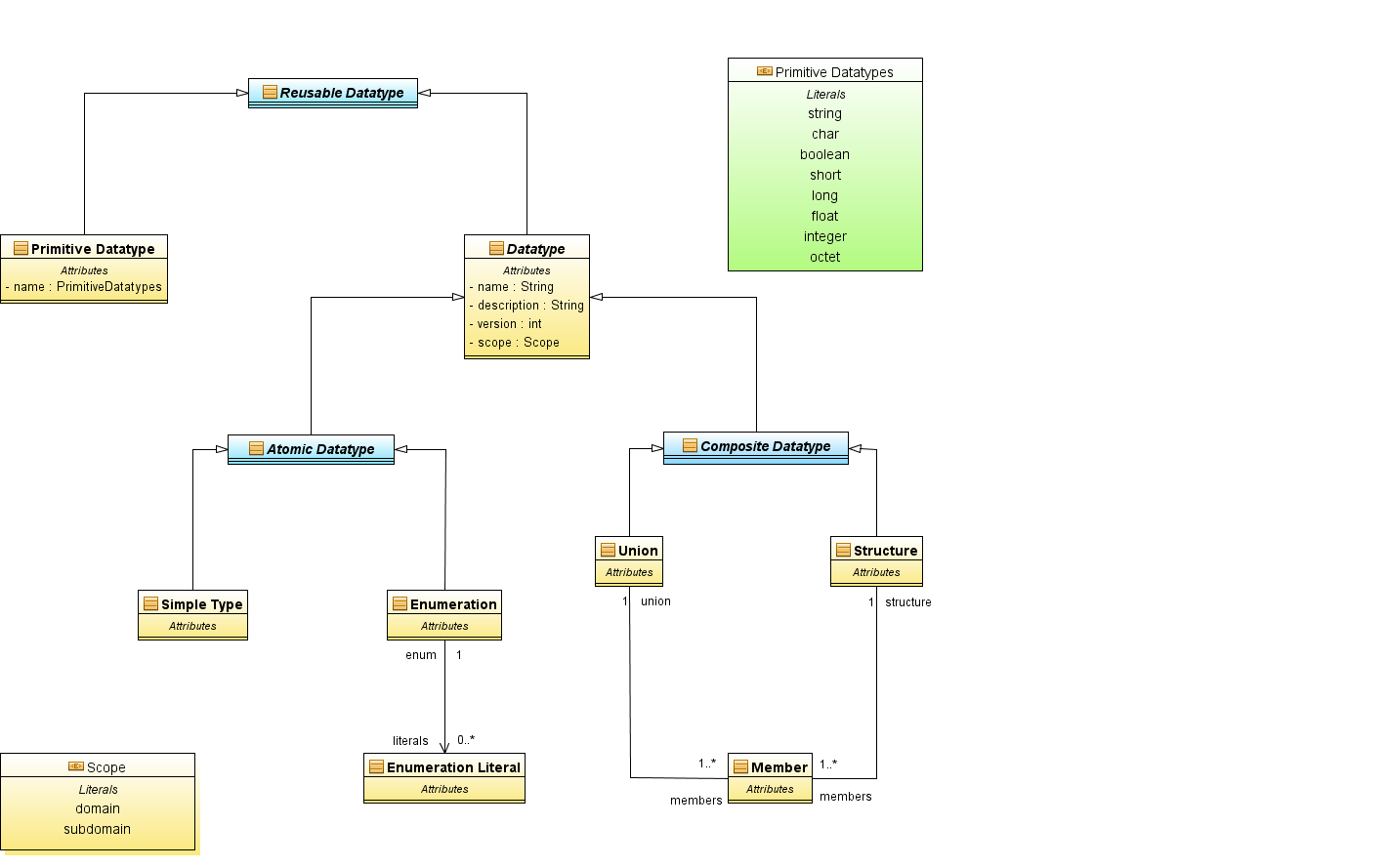
**

Figure 3: Domain model associated with our DSL.

On the top we have an abstract entity **Reusable Datatype**. This abstract class has two concrete subclasses: **Primitive Type** and **Datatype**. **Primitive Type** corresponds to types that can be directly mapped with types defined by the target language/platform (Java/XML schema/Ruby/...). They are not versioned and have no name. Then we have **DataType**. This is again an abstract class. It defines types that have a name, a version. a state, etc... Datatypes can be subdivided into two abstract sub-classes: **Atomic Type** and **Composite Type. Atomic Type** can be either a **Simple type** or a literal of an **Enumeration**. **Composite Type** are made by **Union** or **Structure**.

## The oAW tooling

Having completed the design of the domain model the development of the textual DSL with oAW/Xtext can start. Before we look to the grammar definition let me give you an overview of the oAW tooling chain.

The first step is the creation of an Xtext project by using the Xtext project wizard (see Figure 4).

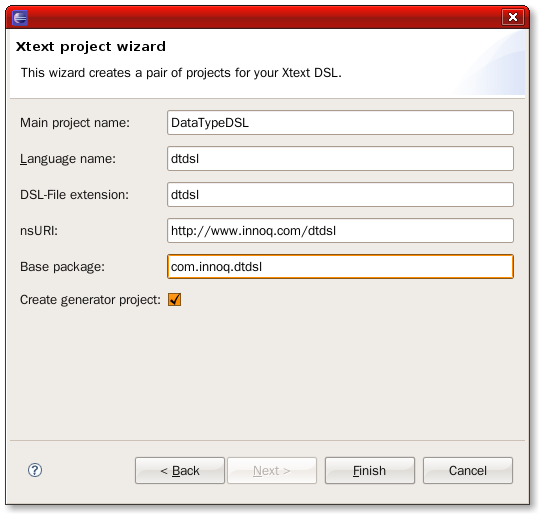
**

Figure 4: The Xtext project wizard ensures an easy start by setting all the needed stuff.

Xtext has created 3 Eclipse projects (see Figure 5) :

1. A project containing the DSL grammar, the generated abstract syntax tree, the DSL parser, and the DSL constrains
2. A project containing the DSL editor,
3. A project containing the DSL generator composed of different Xpand templates.

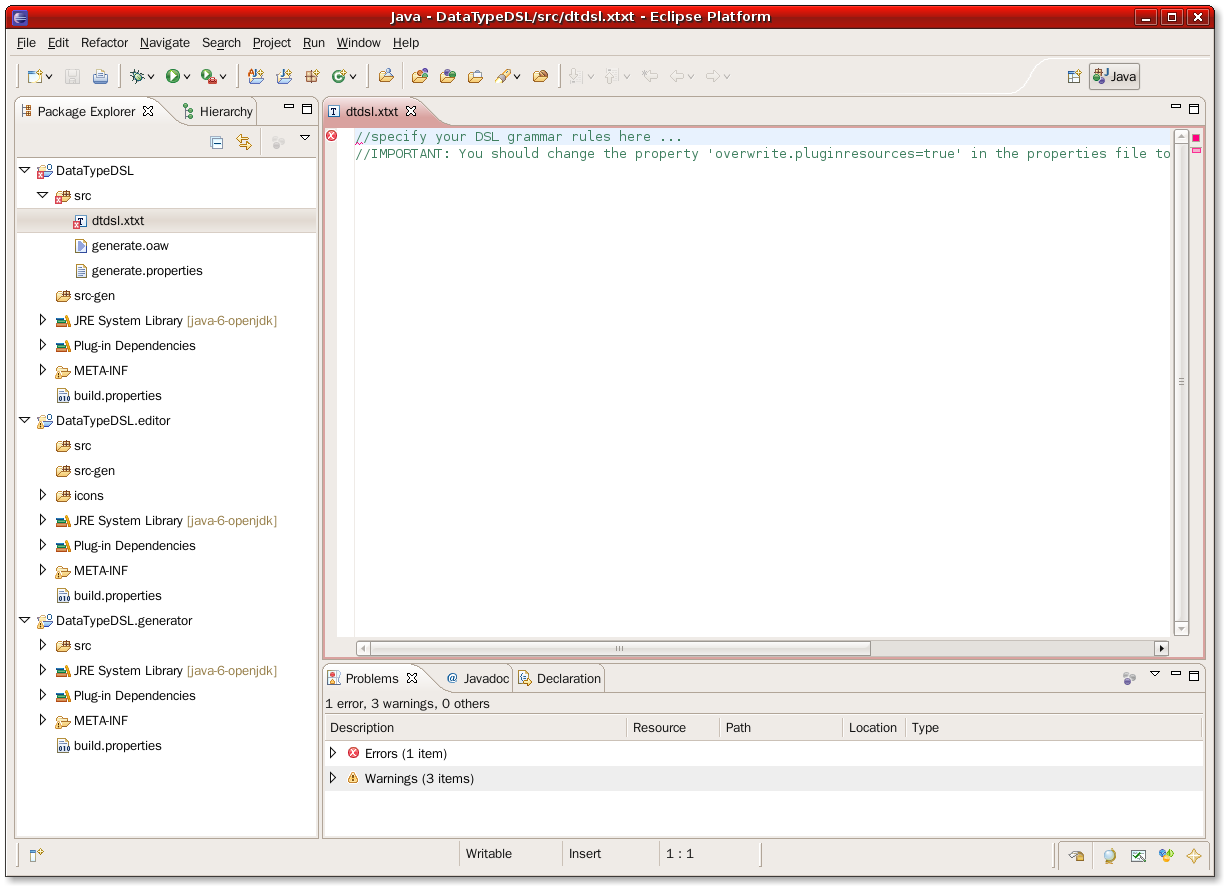
**

Figure 5: The definition of a textual DSL with Xtext is based on three Eclipse plugin projects: one for the language definition self, one for the language editor and one for the language generator.

After the wizard has completed the creation of the three projects, we can start to define the grammar by editing the file /DataTypeDSL/src/dtdsl.xtxt. Xtext provides an editor with code completion and constraint checking for the grammar themselves. The language used is similar to an Extended Backus-Naur-Form (EBNF) with a particularity: it describes the abstract syntax as well as the concrete syntax. The grammar file will contains a list of “rules”. A rule start with their name followed by “:” ending with “;”, and contains a list of tokens.

Before entering in the details of the grammar syntax let us make a complete work-around in order to better understand the architecture of the Xtext tooling. Enter in the grammar (dtdsl.xtxt) the following code:

Datatype:

"datatype" name=**ID** "{"

"@description" description=**STRING**

"@version" version=**INT**

"}"

;

Listing 2: The file dtdsl.xtxt contains the grammar definition of our DSL.

This is a very simple grammar that contains only one rule. Starting from this grammar will oAW/Xtext generate many things. To trigger the generation, right click on the /DataTypeDSL/src/generate.oaw Run as > oAW workflow.

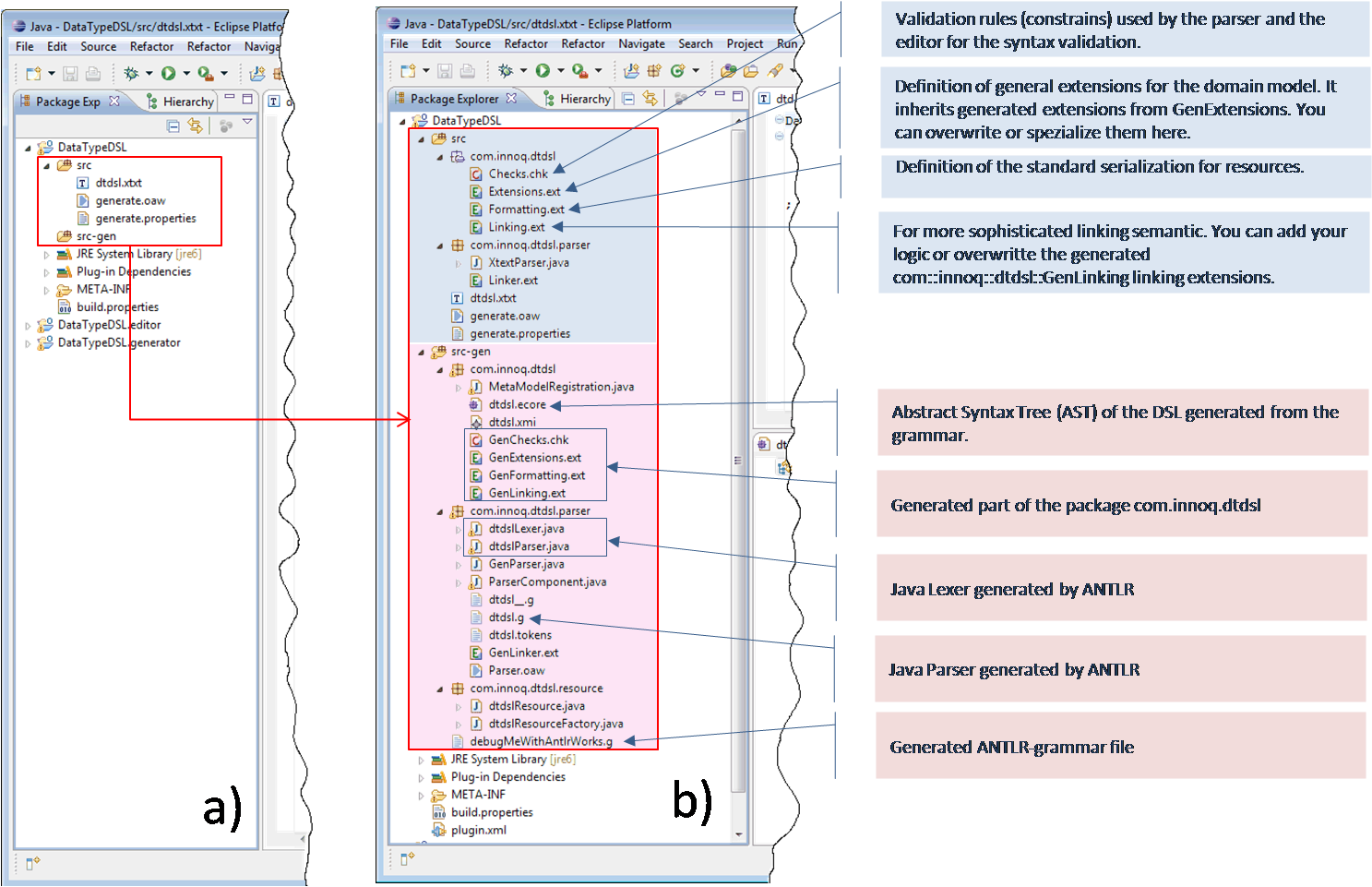


Figure 6: After the definition of the grammar the first generation can be laughed. The figure shows all the file that are created for the DataTypeDSL project.

After the generation (see Figure 6 a) 🡪 b)) is completed, we can have a look to what has been generated. Let’s start with the main Projekt DataTypeDSL. We can see that the sources are divided in two directories /src-gen and /src.

The first one, /src-gen, contains pure generated code that is not intended to be modified. It will be overwritten for each new generation. It contains three subpackages:

/src-gen/com/innoq/dtdst containing :

* the Abstact Syntax Tree (AST) corresponding to our grammar (dtdsl.ecore and its serialized form dtdsl.xml)
* validation rules (GenChecks.cks) used by the DSL editor and the parser for the syntax validation,
* definitions of general extensions (GenExtensions.ext)
* definition of the standard serialization for resources (GenFormatting.ext)
* linking semantic (GenLinking.ext)

/src-gen/com/innoq/dtdst/parser containing :

* the java lexer and parser generated by ANTLR from the grammar.

## The Grammar Definition

*Syntactic analysis = analyzing the original text into its constituent grammatical parts = lexical analysis + parsing*

*This task is traditionally broken down into*

*lexical analysis (done by a lexical analyzer, a tokenizer, or a scanner) which breaks the text down into the smallest useful atomic units, known as tokens, while throwing away (or at least, putting to one side) extraneous information, such as white space and comments—*

*and parsing—which operates on tokens and groups them into useful grammatical structures.*

*We’re going to start with lexical analysis. The part of a compiler that performs this task is called a In brief outline, Regular expressions can describe a variety of languages (sets of strings), including the set of atomic symbols of a typical programming language.*

*• Finite-state automata (FSAs) are abstract machines that also recognize languages.*

*• Deterministic ﬁnite-state automata (DFAs) are a subset of ﬁnite-state automata that are easily converted into programs.*

*• There exists a translation from regular expressions into FSAs.*

*• There exists a translation from FSAs that happen to be nondeterministic into DFAs (and hence into programs).*

*The total process of conversion from regular expression to program is automatable. In fact, we’ll be using a couple of handy programs: Flex (for producing scanners written in C or C++) and JFlex (for producing scanners written in Java). These programs are really compilers themselves, translating succinct descriptions of programming-language syntax (a piece of it, anyway) into programs that “execute” these descriptions to extract tokens from the input. 2*

*Tokens = smallest unit of program text that it is convenient when describing the syntax of a language.*

After having defined the domain model of our language we have to define its grammar. The grammar defines an abstract and a concrete syntaxes.

**Abstract syntax**: for this simple example the domain model will represent the abstract syntax tree of our language. This is of course in many cases not possible.

**Concrete syntax**: here we have to specify how the language has to look like. Do we use {} to delimitate a block ? Do we use “this” as keywords, … A good habit is to sit with the people who will use the language and define how the language should look like. In our example the user of our datatype specification language are domain architects. They are not programmers. The language should be simple and only contains business specific concepts. After a few workshops we agree to the following concrete syntax (see Listing 1).

datatype **Customer** {

@version 1

@description “Description of a customer”

@scope domain

@visibility public

firstname Name 2

@description “Firstname of a customer”

lastname Name 2

@description “Lastename of a customer”

}

primitive **Name** {

@version 2

@description “Description of a customer”

@scope domain

@visibility public

@type String

}

Listing 1: An example of the specification DSL that serves as model for the definition of the concrete syntax.

Having defined the abstract syntax (the domain model) and the concrete syntax we can start to implement our DSL.

Xtext defines different types of rules. In order to better understand what follow we have understand how the DSL parsing works. Parsing, or, more formally, syntactic analysis, is the process of analyzing a sequence of [tokens](http://en.wikipedia.org/wiki/Lexical_analysis#Token) (for example, words) to determine their grammatical structure with respect to a given (more or less) [formal grammar](http://en.wikipedia.org/wiki/Formal_grammar). The parser checks for correct syntax and builds a [data structure](http://en.wikipedia.org/wiki/Data_structure) (often some kind of [parse tree](http://en.wikipedia.org/wiki/Parse_tree), [abstract syntax tree](http://en.wikipedia.org/wiki/Abstract_syntax_tree) or other hierarchical structure) implicit in the input tokens. The parser often uses lexer rules to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be semi-automatically generated (in some programming language) by a tool (such as [Yacc](http://en.wikipedia.org/wiki/Yet_Another_Compiler_Compiler)) from a grammar written in [Backus-Naur form](http://en.wikipedia.org/wiki/Backus-Naur_form).

One can define two main category of rules (see Figure 6).

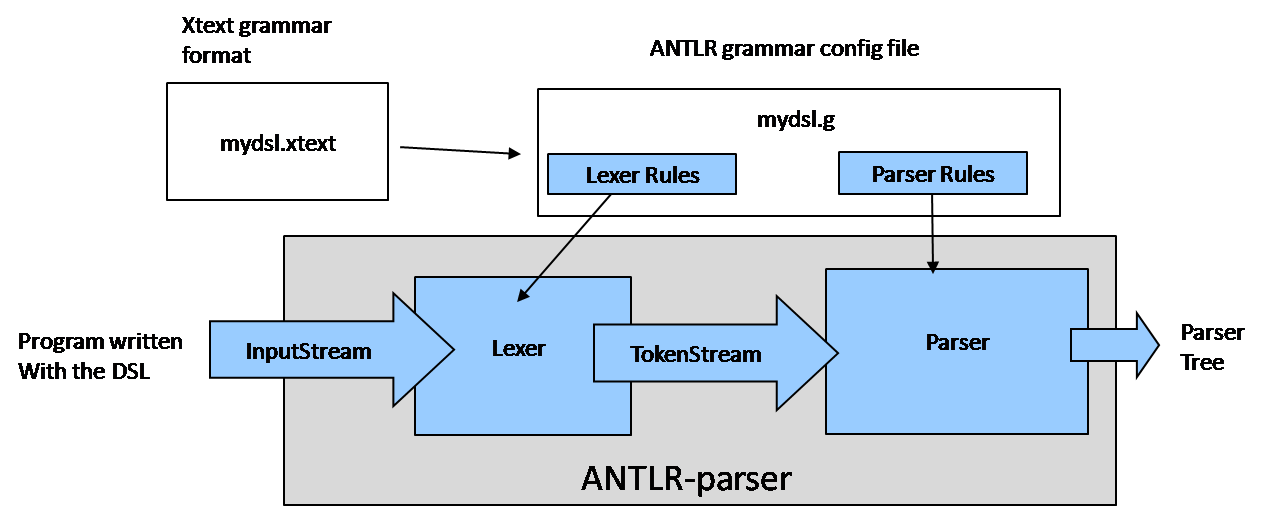


Figure 7: From the Xtext grammar generates Xtext an ANTLR grammar config file used by an ANTLR-parser.

Let start by representing the in our grammar the three top entities of our domain model: ReusableDatatype, PrimitiveDatatype and Datatype as shown in the listing 1.

The first group of rules is the so called *Type Rules*. In listing 1 ReusableDatatype, PrimitiveDatatype and Datatype are Type Rules. Datatype for example is both the name of the rule and the name of the metatype corresponding to this rule.

* Keyword tokens : “{“, “datatype”, “}”
* Assignment 1 (name=ID)
* name = property of the metatype
* Build-in token ID (a lexer rule)
* Assignment 2 (features+=Feature)+
* ( )+ : one or more features
* Points to an other rule (Feature)
* += list of Feature
* Type (Parser) Rules:
  + for each rule Xtext creates a class in the metamodel,each rule property results in a property of the metaclass
  + may contain keywords (using string literal syntax)‏ +
  + The property type is derived from the called rule
  + There are different kinds of properties
    - = single assign
    - += (multiple assign/add)‏
    - ?= (boolean assign)
  + There are different property cardinalities
    - ? (0..1)
    - \* (0..n)
    - + (1 n)
    - nothing (1..1)
* Reference Rule: parsers construct parse tree not graphs, no cross-references possible, usually solved by using a third task: the linking
  + Xtext solution: specification of the linking within the grammar [ ], by default the ID is used as reference
* Lexer Rules (return value = String)
  + Keyword tokens defined with string literal syntax
  + Build-in Lexer Types
  + The ID token ('^')?('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'\_'|'0'..'9')\*
  + The STRING token
  + The INT token
  + The URI token
  + Single and multi-line Comments
  + // this a single line comment
  + /\* this is a multiline comment \*/
* Abstract Type Rules (metatype (multi-) inheritance)
  + A Type rule can be abstract
  + collection of OR-ed alternatives
  + mapped to an abstract metaclass
  + The OR-ed alternative become concrete subclasses
  + Multiple inheritance : the same rule can be or-ed in several abstract rules.
* Enumeration rule
  + Definition of a limited set of defined alternative
  + map to an Enum in the metatmodel
  + Enum keyword
  + list of enum litteral
  + enum litteral has a token name and a string representation

## Tooling generation

Figure 5: Overview of the different components of the Xtext tooling chain.

## Tooling Optimization

1. Syntaxic and semantic validation
2. Code completion
3. Outline
4. Navigation

## DSL Generator Implementation

Generation of SQL statements

## Discussion

(Disadvantages / Advantages of a DSL)

Let start with the "classical" list of advantages associated with the use of DSL (source [the paper of Deuren/Klint/Visser](https://www.innoq.com/svn/Sandbox/al/adsl/DSLs_Survey.pdf" \t "_top)):

* DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.
* DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes.
* DSLs enhance productivity, reliability, maintainability, and portability.
* DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.
* DSLs allow validation and optimization at the domain level.
* DSLs improve testability

And the disadvantages of the use of a DSL (from the same source as previously):

* The costs of designing, implementing and maintaining a DSL.
* The costs of education for DSL users.
* The limited availability of DSLs.
* The difficulty of finding the proper scope for a DSL.
* The difficulty of balancing between domain-specificity and general-purpose programming language constructs.
* The potential loss of efficiency when compared with hand-coded software.

(DSL Classification)

(Executabitlity of DSLs)

(DSL development process)

* Decision
* Analysis
* Design
* Implementation
* Deployment

## References

* [When and how to develop domain-specific languages" M. Mernik, J. Heering, A.M. Sloane, Report SEN-E0517, 2005](http://portal.acm.org/citation.cfm?id=1118890.1118892" \t "_top)