

# Notice

## ecore to hutn in sintaks

Michel Hassenforder  
12 rue des frères Lumière  
68093 Mulhouse  
FRANCE



# Table of contents

1	How to install.....	1
2	Requirements.....	1
3	Processes .....	1
4	Hints .....	2
4.1	StartSymbol .....	2
4.2	Adjectives .....	2
4.3	Protection .....	3
5	Transformation process.....	3
5.1	Relevant classes .....	3
5.2	Main concept around classes and features .....	3
5.3	Relevant features .....	3
6	Conventions used in this document.....	4
6.1	Model conventions used in this document.....	4
6.2	Grammar conventions used in this document.....	4
7	Transformations about an abstract class.....	4
7.1	About the abstract class.....	4
7.2	About its features.....	5
8	Transformations about a concrete class.....	5
8.1	About its features.....	5
8.2	Short or long rendering.....	5
8.3	Rendering of an IDfeature.....	6
8.4	Rendering of an adjective before of an adjective after .....	6
8.5	Rendering of a Boolean value adjective .....	6
8.6	Rendering of an attribute .....	6
8.7	Rendering of a container.....	7
8.8	Rendering of a reference.....	7
8.9	Rendering of a shared feature .....	7
8.10	About the concrete class .....	8
9	Syntax genmodel or not.....	8
9.1	Mandatory cases .....	8
9.2	Optional cases.....	8
9.3	Sintaks genmodel model .....	8
10	Expression examples.....	9
11	TinyJava exemple.....	11

## ecore to hutn in sintaks

### 1 How to install

Installation procedure belongs to the distribution used. Either you can use a site update or drop the plugins in the plugin folder (or in the dropins one). Feel free to look at the <http://www.kermeta.org> web site about sintaks.

### 2 Requirements

You need a java 5 JRE, an eclipse distribution either europa (was the development distribution) or ganymede (was tested, seems working).

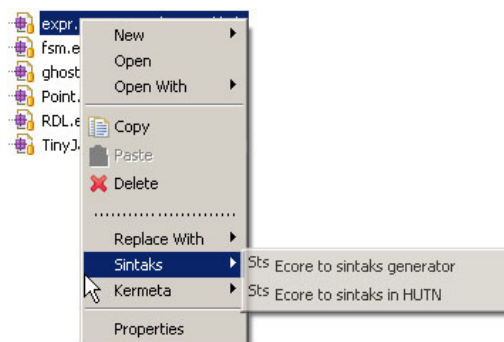
You should have at least in the plugin directory three plugins:

- fr.uha.mips.sintaks.ecore2hutn
- fr.uha.mips.sintaks.stsgen
- org.kermeta.sintaks.model

Of course if you want use and edit the results you need additional plugins such as :

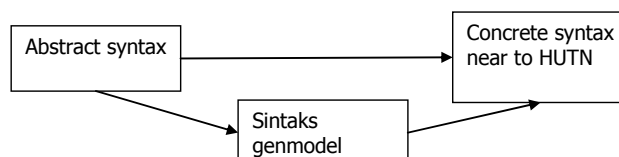
- fr.uha.mips.sintaks.stsgen.edit
- fr.uha.mips.sintaks.stsgen.editor
- org.kermeta.sintaks.model.edit
- org.kermeta.sintaks.model.editor
- org.kermeta.sintaks.ui
- org.kermeta.sintaks
- fr.uha.mips.sintaks.trace

If the plugins are correctly activated, you can see a new menu entry in the contextual menu of an ecore file.



### 3 Processes

This plugin proposes now two ways to develop a concrete syntax near from HUTN for an abstract syntax modelled using EMF (Figure below).



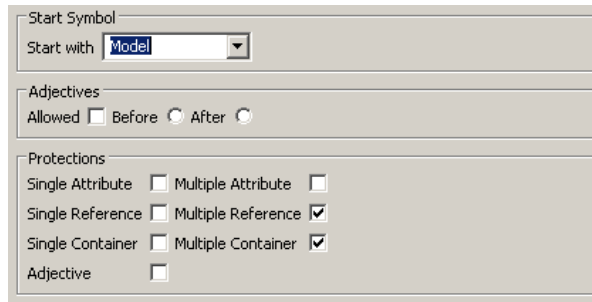
The first (historic) solution is a quick and dirty way to create a concrete syntax close to HUTN (written in sintaks). Given an abstract sintaks and some hints we generate a concrete syntax. Later you can customize the generated file to suit your concrete syntax requirements.

The second solution creates first a sintaks gen model intermediate representation. Such an intermediate representation can later be corrected to add some new information not present in

the abstract syntax (ID, better position, protections, etc). Later you can use such sintaks genmodel to create the targeted concrete syntax.

## 4 Hints

The concrete syntax generation process requires some hints to help in the generation process. The hints are here to balance between security and verbosity or between light and heavy concrete syntaxes. All hints are accessible in the main wizard page after selection of one of two processes to apply (figure below).



A hint is global and acts for all items in the abstract syntax. So you have to choose the more frequent way to represent items. Later, you can correct either the sintaks or the sintaks genmodel file.

### 4.1 StartSymbol

You have to choose the start symbol (concrete EClass) used to start the analysis. The drop menu gives all concrete classes you can use.

### 4.2 Adjectives

You have to choose if adjectives are allowed or not. Adjectives are some single value attribute you can put either before or after the name/ID of a concept.

The example below is generated if 'Allowed' and 'Before' are checked. GetInstance seems to be the ID of a feature, and public, static and void are adjectives put before this ID.

```
public static void getInstance { };
```

The example below is generated if 'Allowed' and 'After' are checked. GetInstance seems to be the ID of a feature, and public, static and void are adjectives put after the ID and enclosed by parenthesis.

```
getInstance ( public static void ) { };
```

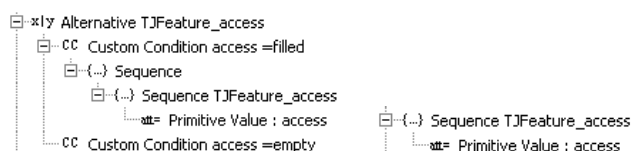
The example below is generated if 'Allowed' is not checked. The attributes are enclosed in braces in an ugly way.

```
getInstance {
    visibility = public;
    access = static;
    returnType = void;
}
```

In brief, adjective proposes nicer concrete syntax and easier to read. Before position is like in HUTN, but a parser can be confused by the conflicting attribute (let think about class and interface were the attribute are before the name... in LL Parser the rules are ambiguous.). After position is better as the ID should create a non ambiguous rule and the enclosing parenthesis enforce non ambiguity; the result is a heavier concrete syntax. Of course, if adjective is not allowed we propose a 'soviet' syntax.

### 4.3 Protection

You have to choose if protections have to be applied to given kind of features. A protection does not have a visual effect on the concrete syntax but it could protect the sintaks engine about empty feature. If a protection is activated, a heavy pattern is copied in the target sintaks file. This pattern is harder to read but safer to execute. Cancel the pattern means that the sintaks file is easy to read (change) but really not safe if the feature is empty. Below, the left hand-side figure shows the protect pattern applied to a single value feature and the right hand-side figure show the same feature without a protection pattern. The pros and the cons of the protection hint should clear



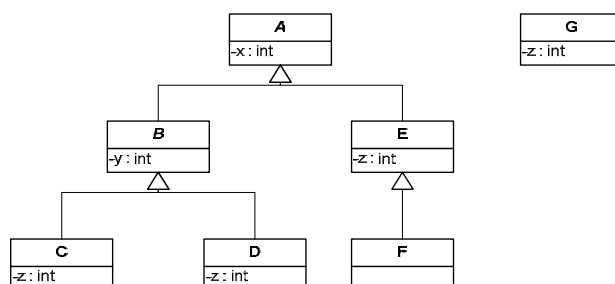
## 5 Transformation process

### 5.1 Relevant classes

According to the ecore metamodel, several concepts can be found in an ecore file. Many of them (enumeration, datatype, ...) are relevant to the value carried by other concepts. As sintaks relies on the factory generated by EMF, we don't care about them. Packages are not processed as each class carried itself the owning package. So, we have only to process EClasses. In fact, two kinds of classes can be found in an ecore model:

- abstract classes
- concrete classes

We assume that an abstract class is an intermediate class in a hierarchy of classes; a concrete class is always a standalone class or a leaf in a hierarchy. In the below diagram, the paths ABC or ABD should be well processed instead of the path AEF where E is concrete and should be abstract...



Warning: concrete intermediate classes are not well processed by our engine (class E).

### 5.2 Main concept around classes and features

In a model you can organize classes and features to share some information such as in the previous figure (from the C and D classes, x and y are shared but not z from the C and D classes). We assume that the concrete syntax should maintain such architecture. In such a case, the feature x in the mother class A should share its concrete representation with the sub classes C, D, E and F, the feature y in the class B have to be shared by the class C and D, but the feature z in the subclass C, D, E and G do not have to share concrete syntax even if they share the name. But later you can choose to share a concrete representation editing either the sintaks genmodel or the sintaks file.

### 5.3 Relevant features

Ecore proposes many kinds of features: in our concrete syntax we take into account only 'real' plain value features.

A derived, transient, not settable feature is automatically removed from the set of available features.

An EReference with an opposite containment is also removed from the set of available features.

An inherited feature in an abstract class is also removed from the set of available features.

## 6 Conventions used in this document

### 6.1 Model conventions used in this document

We call **ID** a single value attribute which is tagged ID in the ecore model or a feature (single value) tagged ID in the sintaks genmodel. An ID attribute is useful to give a 'name' for an EClass or mandatory to query the model about a reference (in case of EReferences).

We call **adjective** a single value attribute which has to be placed before or after the name or the ID of an EClass.

We call **attribute** an ecore attribute (single or multi values) which has to be place in the content section of the concrete syntax of an EClass.

We call **containment** an ecore reference (single or many values) with a container relationship.

We call **reference** an ecore reference without containment relationship from both ways (normal and opposite and single or many values).

Warning: two opposite references are treated by two different rules (one for each end). This leads to two descriptions for the same relationship (automatically updated by EMF). You have to remove one manually or you will have too many references.

### 6.2 Grammar conventions used in this document

Grammar is expressed in an EBNF like format.

"xxx"	a constant (keyword).
<xxx>	a constant (keyword) extracted from the xxx concept.
text	something associated to a feature of the model under construction
*	0 or more
+	1 or more
?	0 or 1
(xxx)	grammar rule grouping

## 7 Transformations about an abstract class

The transformation consists in creating several fragments: one for the abstract class and one per shared feature in the abstract class.

### 7.1 About the abstract class

The most important one is about the abstract class itself. It is not possible to create an abstract class ☺ but an abstract class can be represented by a sub class. In this case we have to create a rule where a kind of abstract class can be created. In sintaks we use a combination of an alternative and several polymorphic conditions composed over the list of all concrete subclasses.

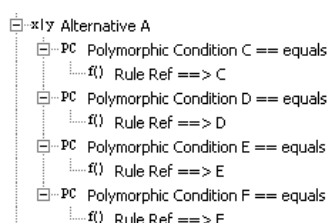
If we try to translate it into a grammar, we obtain expressed in EBNF something like:

```

Abstract_A ::= Concrete_C
             | Concrete_D
             | Concrete_E
             | Concrete_F

```

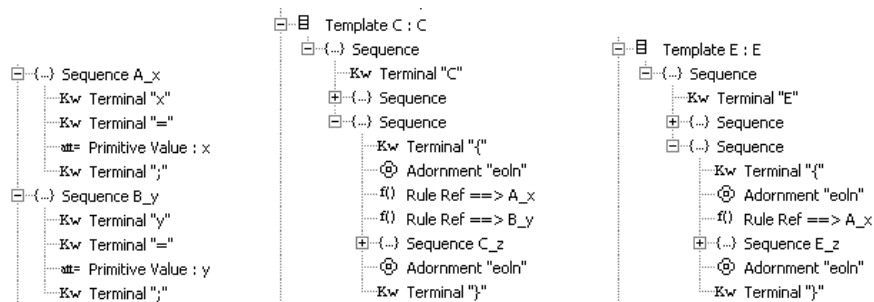
In the case of the running example, we obtain the following description (in sintaks):



## 7.2 About its features

For each feature owned by an abstract class we create an associated fragment (sintaks rule owned by the root concept of sintaks) composed by rules (similar as those describes if they where in a concrete class). For a concrete class, child of this abstract class, we generate a ruleref to the shared rule. We remove the generation about features belonging to the mother class in an abstract sub class.

In the case of the running example, we obtain the following description. The left hand side shows the shared features, the right hand side figures show two classes.



Notice on the left hand side figure, the two conventional sequences of key/value pair (one for x in A, one for y in B).

In the C template we have two rule references to the corresponding sequences (Rule Ref ==> ...) . And in the E template we have only the rule reference to the sequence about the x attribute.

## 8 Transformations about a concrete class

The transformation consists in creating a sintaks template handling the class. Inside the template we have a set of rules handling all the features we have in a concrete class.

### 8.1 About its features

According to specification we classify features into 6 kinds of features: IDfeature, adjective before, adjective after, attribute, container and reference. Look at the section 6.1 to understand the differences. Each kind of feature will be rendered differently in the concrete syntax either in the position or in the style.

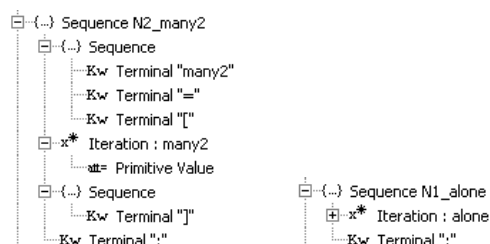
### 8.2 Short or long rendering

Faced with a set of attributes, containers or references, we can have three cases: empty, one element, several elements. If the set is empty, we skip the rendering process. If the set contains one instance we use a short pattern else the long pattern.

A long pattern is non ambiguous and safe but a bit verbose (many nice decorations). In contrary, the short pattern removes all the optional decorations as the value is enough to represent the feature... There should be no ambiguity as there is only one feature in the set. The representation is shorter and perhaps easier to read. See below the EBNF rule and its sintaks representations for a multi value attribute.

```

notAlone    ::= beforeDecoration value ( ',' value ) * afterDecoration ';'
beforeDecoration ::= <eAttribute.name> '=' '['
afterDecoration  ::= ']'
alone          ::= value ( ',' value ) * ';'
  
```

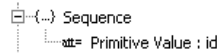


### 8.3 Rendering of an IDfeature

An IDfeature is rendered just with its value and, as an ID should always exist, without any kind of protection and as it is ever alone there is no short or long rendering. The corresponding ENBF rule is below.

```
idFeature ::= value
```

The associated sintaks sequence is below.



### 8.4 Rendering of an adjective before of an adjective after

An adjective before or after is rendered just with its value it can be protected using the pattern in section 4.3. Even if the adjective can be alone or not, there is no short or long rendering, we hope that adjectives are handled as unambiguous short hands to some features. In real world, an adjective should not be protected.

The corresponding ENBF rule is below.

```
idFeature ::= value
```

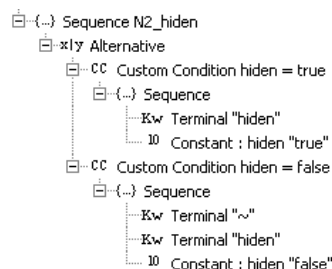
The associated sintaks sequences are below, left unprotected and right protected.



### 8.5 Rendering of a Boolean value adjective

In case of a Boolean adjective we have to shorten the representation as much as possible. Writing true or false is really not a nice way. We choose to follow the HUTN idea. The name of the attribute means a true value and the name preceded by the tilda symbol (~) means false. This leads to the following grammar and its representation in sintaks. Now the sintaks representation seems a little clumsy but the concrete representation is so nice.

```
boolean ::= <eAttribute.name>
          | ~ <eAttribute.name>
```



It seems a complex rule but it is only two alternatives where we look about a given terminal (keyword) preceded or not by the tilda symbol and we use the Constant sintaks rule to set a value to a given attribute.

### 8.6 Rendering of an attribute

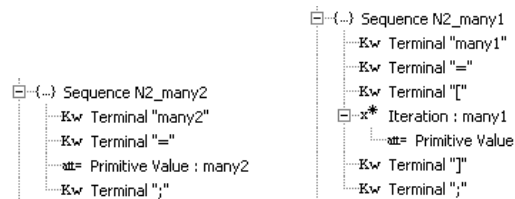
An attribute is rendered using up to eight kinds of pattern depending on three properties: alone in the set of feature, it is a single or a multi valued attribute, it should be protected or not. Protection is applied using the pattern in section 4.3 and property alone or not is rendered using the pattern in section 8.2. The core of the pattern is proposed below in its EBNF notation or in the sintaks one.



```

singleAttribute ::= <eAtt.name> '=' value ';'
multiAttribute  ::= <eAtt.name> '=' '[' value ( ',' value ) * ']' ';'

```



Note: a single value attribute is represented using the name of the attribute, an equal sign, a primitive value and a semicolon. In case of multi value, square brackets enclose an iteration over the collection of values and the colon is removed.

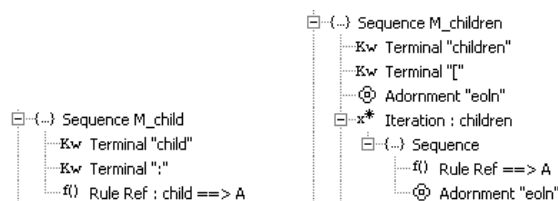
## 8.7 Rendering of a container

A container is rendered using up to eight kinds of pattern depending on three properties: alone in the set of feature, it is a single or a multi valued container, it should be protected or not. Protection is applied using the pattern in section 4.3 and property alone or not is rendered using the pattern in section 8.2. The core of the pattern is proposed below in its EBNF notation or in the sintaks one.

```

singleContainer ::= <eRef.name> ':' eClass.name
multiContainer  ::= <eRef.name> '[' eClass.name * ']' ';'

```



Note: a single value container is represented using the name of the container, a colon and a rule reference to the rule handling the target class. In case of multi value, square brackets enclose an iteration over the rule reference.

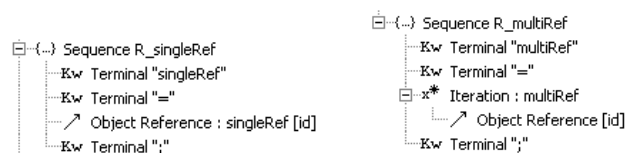
## 8.8 Rendering of a reference

A reference is rendered using up to eight kinds of pattern depending on three properties: alone in the set of feature, it is a single or a multi valued container, it should be protected or not. Protection is applied using the pattern in section 4.3 and property alone or not is rendered using the pattern in section 8.2. The core of the pattern is proposed below in its EBNF notation or in the sintaks one.

```

singleReference ::= <eRef.name> '=' OR(key) ';'
multiReference  ::= <eRef.name> '=' '[' OR(key) ( ',' OR(key) ) * ']' ';'

```



Note: a single reference is represented using the name of the container, an equal sign and an object reference to an EObject identified by the key feature in the model. You should have a feature with the ID property set in the target class of a reference. In case of multi value, square brackets enclose an iteration over the collection of object reference.

## 8.9 Rendering of a shared feature

A shared feature (an inherited feature in a concrete class) is rendered as a rule reference to the target rule generated by the abstract class. See section 7.2 about the look of the generated rules.

### 8.10 About the concrete class

The transformation consists in creating a sintaks template handling the class. Let say that the current class is represented by the name eClass, we propose the EBNF rules below to describe the produced template.

```
Template      ::=  adjectivesBefore?
                  <eClass.name> IDFeature?
                  adjectivesAfter?
                  body
                  "eoln"
```

The rule adjectivesBefore is just the set of adjectives put before the name or ID without any additional decorations:

```
adjectivesBefore ::=  eClass.adjective *
```

The rule adjectivesAfter is just the set of adjectives put after the name or ID but enclosed by round parenthesis:

```
adjectivesAfter  ::=  "(" eClass.adjective * ")"
```

The rule body states that if additional content have to be described, we produce it one per line (to enhance readability), enclosed by curly braces else we just close the class description using a semi colon.

```
body            ::=  "{" "eoln" ( content "eoln" ) + "}"
                    |  ";"
content         ::=  attribute | reference | container | shared
```

## 9 Sintax genmodel or not

This section is devoted to the question: "Do I need a sintaks genmodel ?" as the roots are just the same.

The sintaks genmodel is helpful in several cases and mandatory in several others.

### 9.1 Mandatory cases

Let says that an ecore model is available and not modifiable (eg ecore.ecore).

1) If the ID property is not correctly filled by the engineer, it will be tiring enough to promote for each class an attribute to the ID feature. Don't forget that every thing is forgotten when we generate a new sintaks file.

2) If the ID property is not correctly filled by the engineer, it will be just not possible to create a reference from a class to a class without ID attribute. The Object reference has his key feature empty and will not be able to operate correctly.

### 9.2 Optional cases

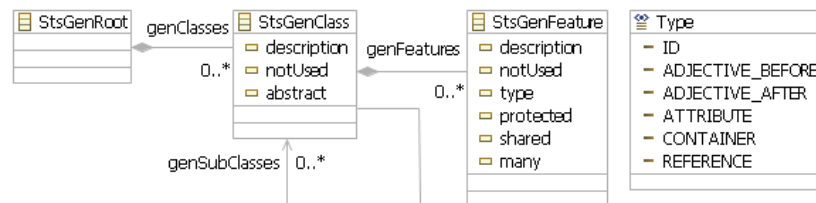
If the ecore file is really a big one (like ecore). The general hints can be really just not enough. Half of the features are well defined but not the other half. Corrections can take a lot of time.

Customizing the sintaks genmodel can be a long work easy to do and to try. As the sintaks genmodel is kept in a file, the designer is free to change its idea until he finds the best representation. Of course several representations can be created.

The idea behind the sintaks genmodel is to capture the basic concepts of a concrete syntax. Later the concrete syntax is modelled using sintaks. If it is straight forward you can skip this way, but...

### 9.3 Sintaks genmodel model

Below, we have the model of the sintaks genmodel. Basically it's a collection of classes to handle and each classes handle either subclasses or a collection of features.



A StsGenClass has a description. It is a derived attribute (label easy to understand in the tree editor).

A StsGenClass can be abstract. It is a derived attribute, a proxy of the abstract attribute of the target EClass. In case of an abstract class the subclasses reference carry relevant information.

A StsGenClass can be tagged notUsed in the generation process. The purpose of this attribute is to hide some classes not relevant for the designed concrete syntax but keeping them in the file.

A StsGenClass has a collection of StsGenFeature treated by the generation process.

A StsGenClass could have a collection of StsGenClass (subclasses) only treated by the generation process if the class is abstract. In this case the collection represents the set of usable concrete classes.

A StsGenFeature has a description. It is a derived attribute (label easy to understand in the tree editor).

A StsGenFeature can be tagged notUsed in the generation process. The purpose of this attribute is to hide some feature not relevant for the designed concrete syntax but keeping them in the file.

A StsGenFeature has a type, one in the Type enumeration. The purpose is to select the best way to represent a feature.

A StsGenFeature can be tagged protected in the generation process. The purpose of this attribute is to protect the access to the value using a special sintaks pattern.

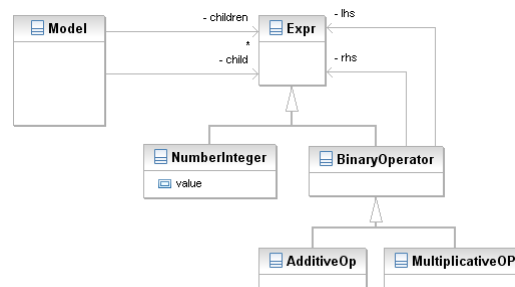
A StsGenFeature can be tagged shared. A feature is shared if it is owned by the mother class. Later it means that a shared feature will share a concrete syntax.

A StsGenFeature can be many. It is a derived attribute, a proxy of the many attribute of the target EStructuralFeature.

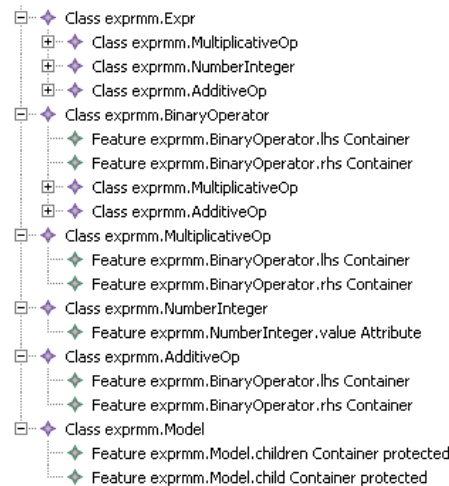
The Type enumeration seems straight forward.

## 10 Expression examples

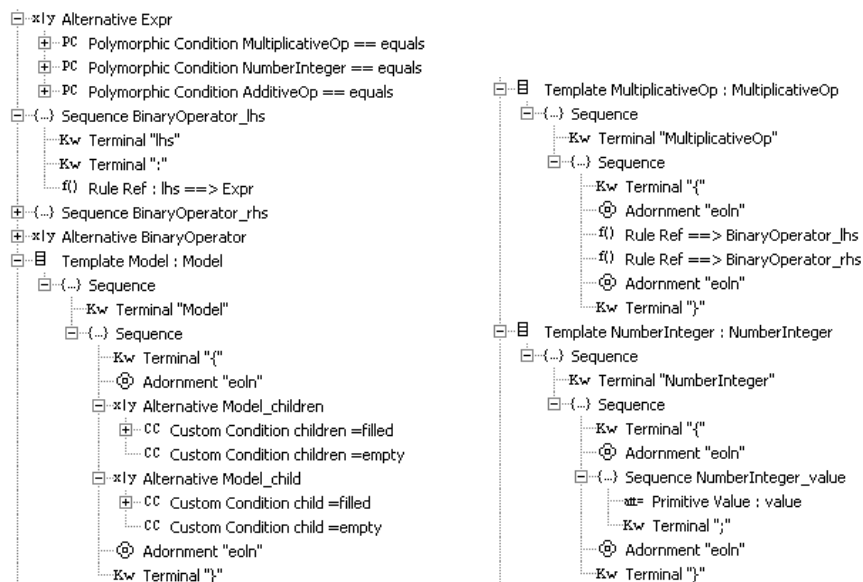
We propose the following metamodel :



If we generate the sintaks genmodel using the following hints: the startClass is Model, adjectives are not allowed and without any protection. Later, we edit the stsgen file and add a protection to the child and children container. We obtain the following stsgen file.



The descriptions are enough understandable so we need no additional comment. Now we can generate the sintaks file.



Some comments from the top to the bottom of the generated file. Alternative Expr try to choose the best concrete class. Sequence BinaryOperator\_lhs expresses the representation of a shared unprotected container (lhs). The rule reference targets the Alternative Expr (which targets the real concrete class). Sequence BinaryOperator\_rhs has the same representation as Sequence BinaryOperator\_lhs. The Alternative BinaryOperator is not relevant and can be safely removed. The template Model concerns the Model class which owns two protected container child and children. The template MultiplicativeOp just calls the shared features (lhs and rhs). The template NumberInteger uses a Primitive Value to extract the value from a file.

If we create a very simple model such as "11 \* 12 + 13" in the common way to represent expressions, we obtain the following text (Ok, I added some spaces and returns to format it a little):

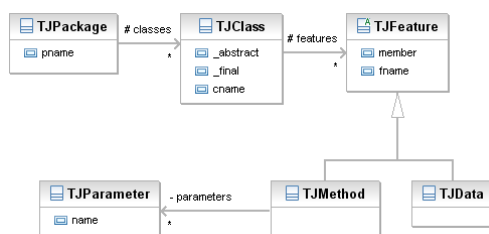
```
Model {
  child : MultiplicativeOp {
    lhs : NumberInteger { 11 ; }
    rhs : AdditiveOp {
      lhs : NumberInteger { 12 ; }
      rhs : NumberInteger { 13 ; }
    }
  }
}
```

```
}

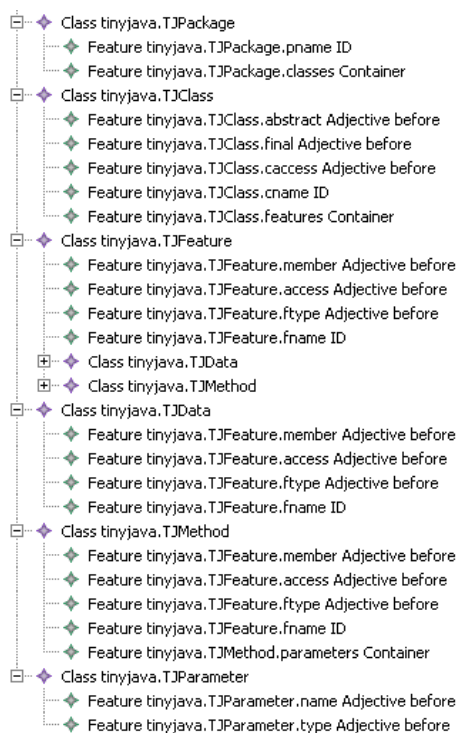
```

Later you can edit the sintaks file to adjust the representation. For example, simplify the NumberInteger to use only the primitive value. Of course, adapt this file to work on infixed language will be hard but it is a good starting point.

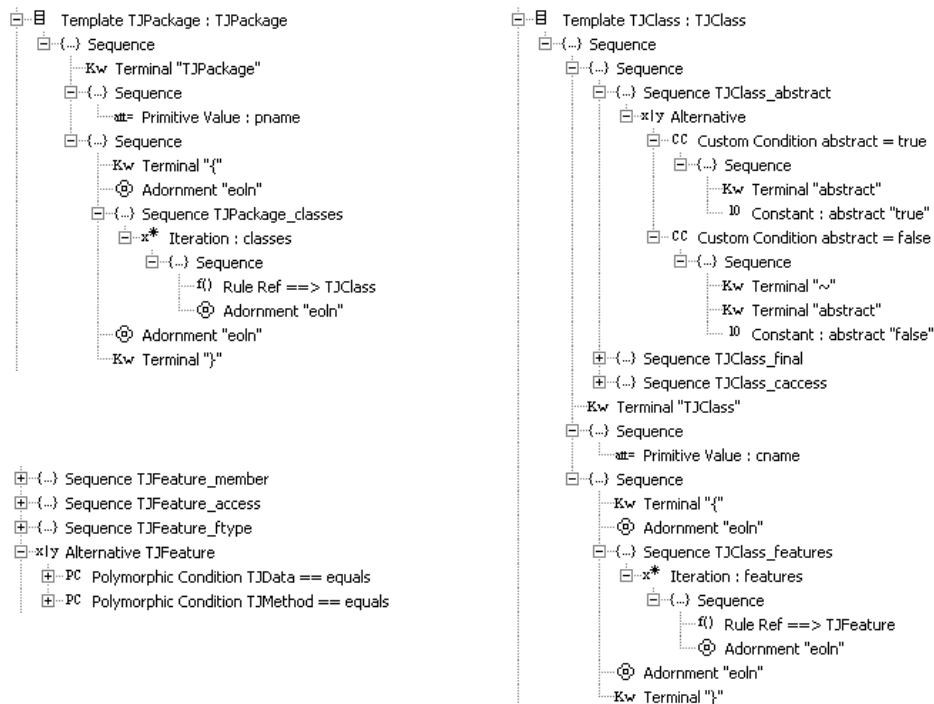
## 11 TinyJava exemple



In the tinyJava example, we propose the above meta model. You should notice that the different names are tagged as ID (not TJParameter). If we generate the sintaks genmodel using the following hints: the startClass is TJPackge, adjectives are allowed before and without any protection. We obtain the following stsgen file.



The descriptions are enough understandable so we need no additional comment. Now we can generate the sintaks file. Below, some excerpt from the file.

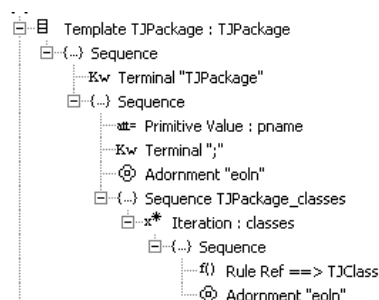


If I create a very simple TJClass Point, with a private empty constructor, the traditional get/set, a move and a public class method named create, we obtain the following concrete syntax.

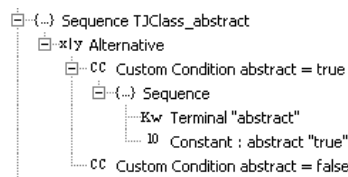
```
TJPackage hassen {
  ~ abstract ~ final public TJClass Point {
    member private Integer TJData x ;
    member private Integer TJData y ;
    member private void TJMethod Point { }
    member public void TJMethod setX {
      x Integer TJParameter ;
    }
    member public Integer TJMethod getX { }
    member public void TJMethod move {
      x Integer TJParameter ;
      y Integer TJParameter ;
    }
    ~ member public String TJMethod create {
      x Integer TJParameter ;
      y Integer TJParameter ;
    }
  }
}
```

Ok, it is not really nice, but it works. Now it is time to change the look.

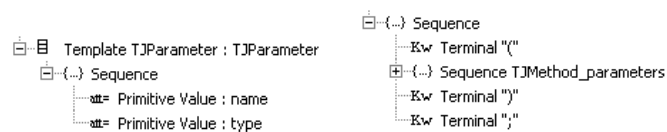
First: the TJPackage, I propose to use the java convention. So I remove the brackets etc... Now the template looks like the template below.



Second: the booleans are written using the HUTN way. Well, I switch to the standard java way either a keyword or nothing to indicate the corresponding feature is false. Example with the abstract keyword:



Third: the TJParameter is ugly; ok I use only the name and the type. The iteration is not nice, why not change it to a comma separated iteration enclosed into parenthesis. Now the TJParameter template look like this one on the left hand side figure, and the iteration is like the right hand side figure:



Last: in real world java do not require to tag differently data or method, the semicolon or the braces are enough to distinguish them. We can do the same changes. In our case, just remove the terminals.

Now the generated file with the same example looks like:

```
TJPackage  hassen ;

public TJClass  Point {
    private  Integer  x ;
    private  Integer  y ;
    private  void  Point () ;
    public  void  setX ( x  Integer ) ;
    public  Integer  getX () ;
    public  void  setY ( y  Integer ) ;
    public  Integer  getY () ;
    public  void  move ( x  Integer , y  Integer ) ;
    static  public  String  create ( x  Integer , y  Integer ) ;
}
```