

Introduction à Xtext

Par Alain BERNARD 

Date de publication : 28 octobre 2012

Dernière mise à jour : 13 décembre 2012

Cet article est le premier d'une série de tutoriels liés à l'utilisation d'Xtext et à la création d'un éditeur pour un DSL personnalisé. Dans ce premier article, nous verrons comment créer une grammaire Xtext et les différentes possibilités qui sont offertes pour rendre l'éditeur généré par Xtext le plus convivial possible, au sein même de la grammaire. Les concepts gravitant autour de la personnalisation de l'IDE non directement liés à la grammaire seront traités dans un autre article.

Les sources de l'exemple sont disponibles à l'adresse suivante : **FTP** ou **HTTP**.

Pour toute remarque ou question sur ce tutoriel, profitez de cette discussion : **Commentez**

I - Introduction.....	3
II - Le framework Xtext.....	3
III - Présentation de notre DSL.....	3
IV - Création de la grammaire Xtext.....	6
IV-A - Installation et démarrage de Xtext.....	6
IV-B - Écriture de la grammaire Xtext.....	8
IV-B-1 - « Hidden terminals ».....	10
IV-B-2 - Terminaux.....	10
IV-B-3 - DataTypes.....	11
IV-B-4 - Types énumérés.....	11
IV-B-5 - Assignations.....	11
IV-B-6 - Références.....	12
IV-B-7 - Imports de fichiers.....	13
IV-B-8 - Règles de grammaire.....	13
IV-C - Génération de l'infrastructure associée à la grammaire.....	13
IV-C-1 - Le fichier MWE2.....	13
IV-C-2 - Lancement de la génération.....	14
IV-C-3 - Résoudre les erreurs : LL-recursivity.....	15
IV-D - Test de l'éditeur.....	15
IV-D-1 - Créer la configuration.....	15
IV-D-2 - Testons notre grammaire !.....	16
V - Un outil bien pratique : ANTLRWorks.....	17
VI - Conclusion.....	18
VII - Liens utiles.....	18
VIII - Remerciements.....	18

I - Introduction

Le but de cet article est de présenter le fonctionnement d'Xtext au travers de la mise en place de la grammaire. Tout d'abord nous verrons quels sont les enjeux d'Xtext, puis nous présenterons notre exemple avant de se lancer dans la mise en place de la grammaire. Nous allons aborder dans cet article des notions avancées de gestion de la grammaire Xtext. Pour un tutoriel plus court qui se focalise plus sur les concepts de base, je vous invite à consulter le tutoriel de Georges **Kemayo** : [Introduction à Xtext : créer son propre DSL](#). D'autre part, seuls les concepts qui ont trait à la grammaire Xtext seront abordés dans cet article. Ceux liés à la personnalisation de l'IDE seront présentés dans un prochain article.

Pour suivre cet article, il est recommandé d'avoir des connaissances en [développement de plugins sous Eclipse](#), ainsi qu'en [grammaire des langages](#). Cet article est basé sur Eclipse 4.2 (Juno).

II - Le framework Xtext

Xtext est un framework pour le développement de langages de programmation et de DSL (Domain-Specific programming Language). Un DSL est un langage propre à un métier ou à une société qui suit un formalisme défini et compact. On peut ensuite faire le pont entre ce langage simple assimilable par tous sans prérequis informatique et un langage de programmation générique, tel que du C ou du Java. Pour plus d'informations reportez-vous à la [page Wikipédia](#). Le framework Xtext s'appuie sur une grammaire générée ANTLR ainsi que sur le framework de modélisation EMF. Ainsi, si vous utilisez déjà une grammaire ANTLR, le bénéfice d'une migration vers Xtext est « limité » à la génération d'un IDE pour votre langage.

Xtext couvre tous les aspects d'un IDE moderne : parseur, compilateur, interpréteur et intégration complète dans l'environnement de développement Eclipse. À ce titre, il permet de couvrir les aspects et les fonctionnalités principales d'un IDE classique.

Xtext est donc adapté aux développeurs désireux de fournir un environnement convivial aux utilisateurs d'un DSL. Il leur permet de mettre en place facilement tous les outils énumérés au paragraphe précédent à partir d'un unique fichier contenant la description de la grammaire du langage.

Parmi les fonctionnalités d'Xtext, on peut mentionner :

- coloration syntaxique : suivant les éléments de la grammaire, Xtext propose une coloration syntaxique entièrement personnalisable pour les éditeurs de votre DSL ;
- autocomplétion : Xtext propose directement l'autocomplétion sur les éléments du langage ;
- validation et « Quick fixes » : Xtext valide le contenu de l'éditeur « à la volée », produisant ainsi un retour direct à l'utilisateur en cas d'erreur de syntaxe. Il permet en plus de proposer, à la manière d'Eclipse, des corrections sur les erreurs détectées ;
- intégration avec d'autres composants Eclipse : en fournissant une API riche pour la manipulation des ressources, Xtext permet de créer facilement des vues graphiques pour la gestion de votre DSL ;
- et plus encore de fonctionnalités pour l'IDE : personnalisation de la vue Outline, recherche des références, hyperliens sont autant d'éléments qui rendent le développement de votre DSL encore plus facile.

Pour aborder tous ces concepts, nous nous baserons sur un exemple simple, décrit dans le paragraphe suivant. Cet exemple (et donc ce tutoriel) met de côté le lien entre la génération d'un DSL et la transformation du DSL en un langage de programmation générique.

III - Présentation de notre DSL

Le but de notre DSL est de créer des fichiers contenant des informations sur des lignes aériennes. Chaque ligne aérienne porte un nom et est définie par :

- un aéroport de départ ;
- un aéroport d'arrivée ;
- un avion ;
- une compagnie ;
- une durée de vol ;

- un paramètre indiquant s'il s'agit d'une ligne régulière.

Un avion est défini par :

- son nom ;
- le nombre de passagers qu'il peut transporter ;
- le nom de la compagnie propriétaire ;
- un paramètre indiquant la motorisation (hélices ou propulseurs).

Un aéroport est défini par :

- son nom ;
- un code OACI ;
- un pays de localisation ;
- un nombre de pistes.

Un exemple de fichier est donné ci-dessous :

```
//Inclusions
%include <avions.air>;
%include <aeroports.air>;

/* ***** *
   Definition des avions
   ***** */
Avion A380:
Passagers: 525;
Motorisation: PROPULSEURS;
End.

Avion ATR42:
Passagers: 48;
Motorisation: HELICES;
End.

/* ***** *
   Definition des aeroports
   ***** */
Aeroport "Marseille-Provence":
OACI: LFML;
Pays: "France";
Pistes: 2;
End.

Aeroport "Roissy-CDG":
OACI: LFPG;
Pays: "France";
Pistes: 4;
End.

Aeroport "JFK-Airport":
OACI: KJFK;
Pays: "USA";
Pistes: 4;
End.

Aeroport "Toulouse-Blagnac":
OACI: LFBO;
Pays: "France";
Pistes: 2;
End.

/* ***** *
   Definition des lignes
   ***** */
Ligne "Toulouse-Marseille":
Avion: ATR42;
Duree: 01h00m;
Compagnie: "Airlinair";
Depart: LFBO;
Arrivee: LFML;
End.

Ligne "NYC-Paris":
Avion: A380;
Duree: 08h15m;
Compagnie: "Air France";
Depart: LFPG;
Arrivee: KJFK;
End.
```

Fichier d'exemple de notre DSL

Comme on peut le voir, le fichier est découpé en quatre blocs.

- Les inclusions : cette section va nous permettre d'inclure des fichiers de notre DSL contenant certaines déclarations. Elle va nous permettre d'alléger le contenu du fichier de déclaration des lignes en déclarant avions et aéroports ailleurs que dans le fichier.
- Les avions : ici sont déclarés les avions. Chaque bloc se termine par « End. », chaque ligne par un ';' et les mots-clés sont séparés des valeurs par ':'. Remarquez que le nom des avions n'est pas entre guillemets. Ceci

pour la bonne raison que ces mots ne seront pas considérés comme de simples chaînes de caractères mais bien comme des identifiants de variables.

- Les aéroports : le fonctionnement est le même que pour les avions. De même que pour le nom des avions, c'est ici le code OACI qui va servir de référence.
- Les lignes : dans les déclarations de lignes aériennes, les identifiants des avions et le code OACI des aéroports seront utilisés pour faire les liens entre les variables. Remarquez aussi le format de l'heure, que nous allons spécifier précisément dans notre grammaire.

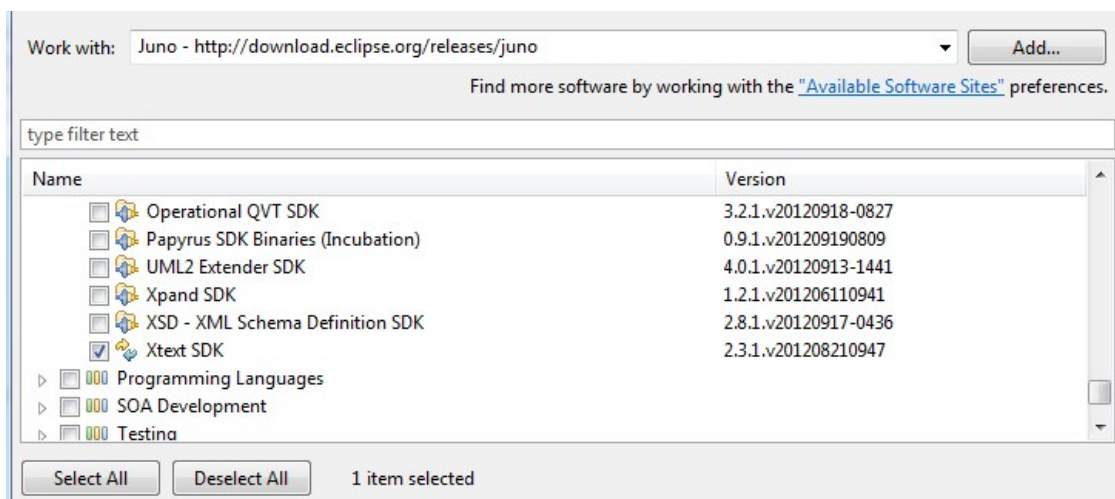
Pour raffiner notre grammaire, nous partirons du postulat que des avions et des aéroports sont déclarés dans le désordre dans un fichier et qu'il se peut que le fichier n'en contienne pas (ils sont alors contenus dans les fichiers inclus). Par contre, les lignes aériennes sont forcément déclarées après les avions et les aéroports. Un fichier peut aussi être vide !

Il est temps de créer notre grammaire Xtext !

IV - Création de la grammaire Xtext

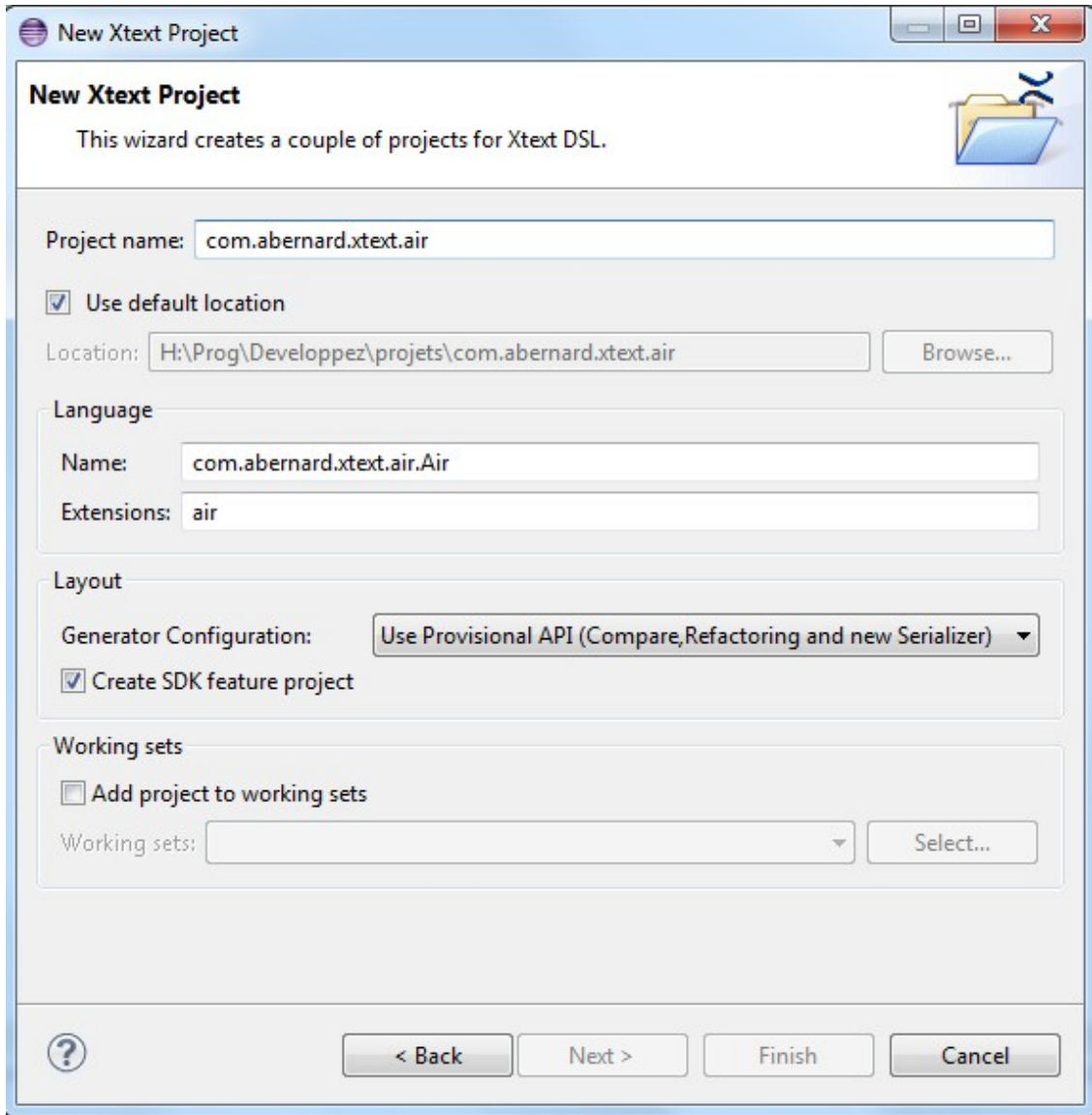
IV-A - Installation et démarrage de Xtext

Le plus simple pour commencer à utiliser Xtext est de télécharger la distribution d'Eclipse correspondante sur le [site officiel](http://download.eclipse.org/releases/juno). Cette distribution contient déjà tous les plugins nécessaires au bon fonctionnement d'Xtext (EMF notamment). Néanmoins, si vous souhaitez ajouter Xtext sur une version d'Eclipse existante, parcourez le repository correspondant à votre version d'Eclipse (pour Juno : <http://download.eclipse.org/releases/juno>) et sélectionnez « Xtext SDK » :



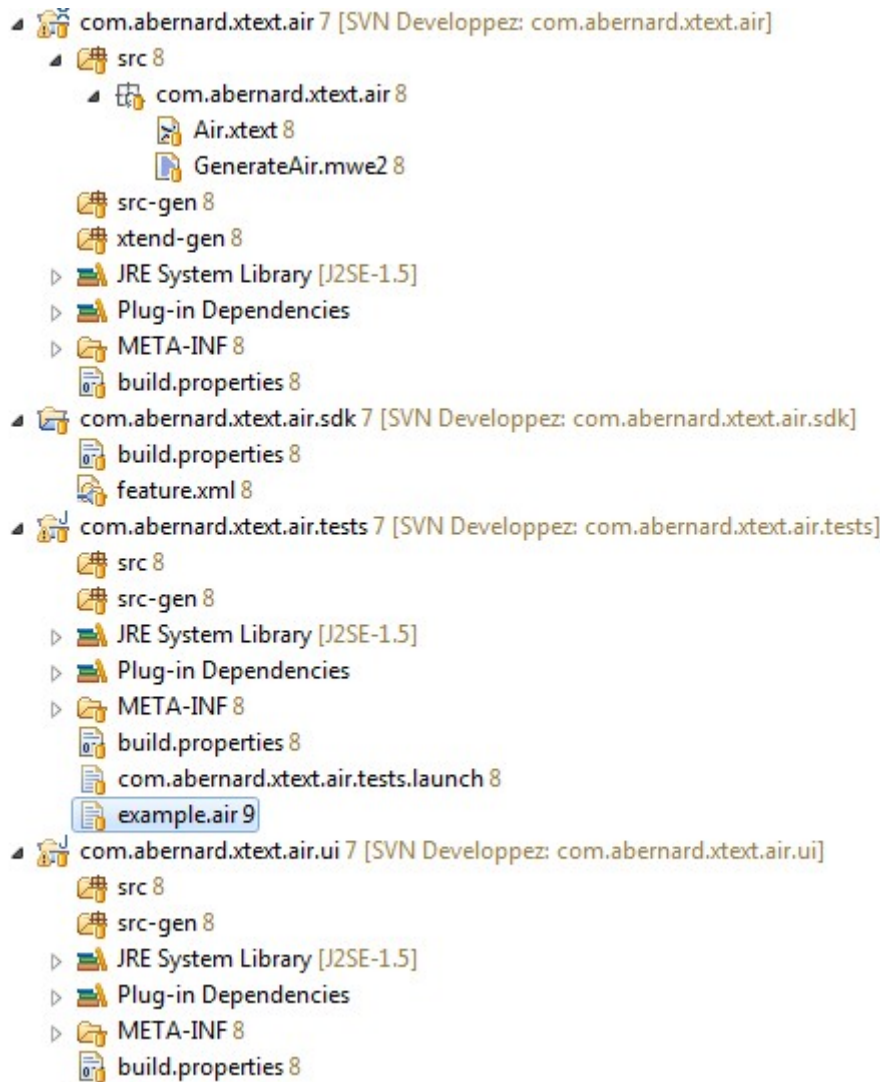
Installation de Xtext

Créez ensuite le projet Xtext via « File > New > Other > Xtext > Xtext project ». Un assistant s'ouvre, vous demandant de renseigner le nom de votre projet mais surtout l'extension de fichier correspondant à votre DSL et le nom de votre DSL. Dans notre cas, nos fichiers porteront l'extension « .air ».



Xtext wizard

L'assistant nous permet aussi de créer directement un projet de type « feature » qui nous permettra de déployer plus facilement nos plugins de DSL. À la fin de l'assistant, quatre nouveaux projets sont créés dans notre workspace Eclipse :



Hiérarchie des projets Xtext

Chacun de ces projets a un but distinct.

- `com.abernard.Xtext.air` : contient le cœur de notre DSL. Il est pour l'instant vide, hormis deux fichiers : le fichier « `Air.Xtext` » et le fichier « `GenerateAir.mwe2` ». Ces fichiers vont nous permettre respectivement de décrire notre projet et de générer automatiquement toutes les classes de gestion du langage et de l'IDE. Hormis ces deux fichiers, nous ne serons pas amenés à modifier nous-mêmes les fichiers qui seront contenus dans les packages « `src` », « `src-gen` » et « `xtend-gen` ».
- `com.abernard.Xtext.sdk` : « `feature` » qui permet de générer facilement les plugins de notre DSL afin de les incorporer dans une distribution Eclipse.
- `com.abernard.xtest.air.tests` : projet qui peut nous permettre de créer des tests JUnit sur notre grammaire.
- `com.abernard.Xtext.air.ui` : contiendra l'interface de notre IDE et toutes les classes propres à son fonctionnement. Les packages « `src` » et « `src-gen` » seront remplis à la première génération avec des fichiers permettant à Xtext de nous apporter les éléments nécessaires à la personnalisation de notre IDE. Encore une fois, nous ne modifierons pas les fichiers présents dans « `src-gen` ».

Nos projets étant créés, il nous faut maintenant créer la grammaire Xtext dans le fichier « `Air.Xtext` ».

IV-B - Écriture de la grammaire Xtext

Le contenu du fichier « `Air.Xtext` » est le suivant pour notre grammaire.

Air.xtext

```

1.
2. grammar com.abernard.xtext.air.Air with org.eclipse.xtext.common.Terminals
3.
4. generate air "http://www.abernard.com/xtext/air/Air"
5.
6. // R1 : Racine du fichier
7. Model:
8.     (includes+=Includes)*(planes+=Plane | airports+=Airport)*(airlines+=Airline)*;
9.
10. // R2 : Declaration des inclusions
11. Includes:
12.     '%include' '<'importURI=INCLUDE'>';';
13.
14. // R3 : Declaration des avions
15. Plane:
16.     'Avion' name=ID ':'
17.     ('Passagers' ':' passengers=INT ';'
18. & 'Motorisation' ':' motorisation=Engines';')
19.     'End.';
20.
21. // R4 : Declaration des aeroports
22. Airport:
23.     'Aéroport' title=STRING ':'
24.     ('OACI' ':' name=CodeOACI';'
25. & 'Pays' ':' country=STRING';'
26. & 'Pistes' ':' runways=INT';')
27.     'End.';
28.
29. // R5 : Declaration des lignes aeriennes
30. Airline:
31.     'Ligne' name=STRING ':'
32.     ('Avion' ':' plane=[Plane]';'
33. & 'Compagnie' ':' company=STRING';'
34. & 'Depart' ':' departure=[CodeOACI|OACI]';'
35. & 'Arrivee' ':' arrival=[CodeOACI|OACI]';'
36. & 'Duree' ':' duration=DURATION';'
37. & regular?='REGULIERE';?)
38.     'End.';
39.
40.
41. // R6 : Enumeration des motorisations
42. enum Engines:
43.     PROPELLER ='HELICES' | ENGINE = 'PROPULSEURS' ;
44.
45. // R7 : Duree de vol
46. terminal DURATION : (('0'..'2' '0'..'9') 'h')? ('0'..'5' '0'..'9') 'm';
47.
48. // R8 : Declaration du code OACI
49. CodeOACI : name=OACI;
50.
51. terminal OACI : ('A'..'Z') ('A'..'Z') ('A'..'Z') ('A'..'Z');
52.
53. // R9 : Reconnaissance des fichiers inclus
54. terminal INCLUDE: ID('air');
  
```

Nous allons détailler les points particuliers de cette grammaire. Tout d'abord intéressons-nous à la première ligne, qui contient :

```
[...] with org.eclipse.xtext.common.Terminals
```

Le fichier décrit ici contient un ensemble d'éléments génériques de langage qu'Xtext fournit de manière à simplifier l'écriture de notre grammaire. Ce fichier peut-être ouvert grâce à un Ctrl-Clic.

```

/*****
 * Copyright (c) 2008 itemis AG and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *****/
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

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

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

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

terminal ANY_OTHER: .;
  
```

Xtext Terminals

IV-B-1 - « Hidden terminals »

La première ligne contient le texte :

```
hidden(WS, ML_COMMENT, SL_COMMENT)
```

Cela énumère les éléments de la grammaire qu'Xtext ne prendra pas en compte dans les analyses syntaxiques et sémantiques. Typiquement, les commentaires sur une ligne (SL_COMMENT) et sur plusieurs lignes (ML_COMMENT). De même les espaces et les retours chariot (WS) n'ont pas d'importance sémantique.

i La règle `SL_COMMENT` a une importance toute particulière : en s'assurant de sa présence dans votre grammaire, vous débloquentez automatiquement la fonctionnalité Eclipse pour commenter/décommenter un bloc, grâce au raccourci `Ctrl+Maj+C` ou à un clic droit puis « *Toggle Comment* » dans l'éditeur.

IV-B-2 - Terminaux

Xtext définit aussi une manière générique de décrire un identifiant (de variable par exemple) grâce à la règle ID. Il propose aussi une définition pour les entiers (INT) et pour les chaînes de caractères (STRING). Tous ces éléments sont des éléments dits « terminaux ». Leur nom est systématiquement en majuscules et ils ne peuvent contenir d'affectation. Ce sont des valeurs brutes caractérisées par une suite de caractères ou un intervalle délimité par des expressions (caractérisé par `'->'`, dans `ML_COMMENT`). On peut par contre spécifier le type de données qu'auront ces terminaux dans le modèle généré. Regardons par exemple :

```
terminal INT returns ecore::EInt:
```

Cette instruction signifie que les terminaux INT auront une valeur de type `EInt`, élément définissant les entiers dans le métamodèle `ECore`. L'utilisation de `ECore` est permise par l'instruction :

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

On peut aussi si on le souhaite inclure encore dans notre propre grammaire afin d'utiliser les types de données proposés. Notons que les cardinalités des différents éléments dans ces terminaux sont données par des expressions de type EBNF : Extended Backus-Naur Form. De fait les cardinalités suivantes peuvent être exprimées :

- exactement un (par défaut) ;
- un ou zéro (opérateur '?') ;
- indéfini (opérateur '*') ;
- au moins un (opérateur '+').

Au sein des terminaux, on peut utiliser différents mécanismes tels que les intervalles de caractères ('0'..'9'), les intervalles de texte (grâce au caractère '->'), les « wildcards » ('f.o' va reconnaître fAo, fOo, f_o...) et les négations exprimées grâce à '!'. **Il est important de noter que l'ordre des terminaux est essentiel.** En effet lors de la lecture du fichier, Xtext s'arrête dès qu'il a trouvé un terminal correspondant au texte lu. Dès lors, les terminaux de la grammaire peuvent se masquer les uns les autres. Il est donc recommandé si possible d'utiliser des éléments de type DataType.

IV-B-3 - DataTypes

Ces éléments sont décrits comme des règles standards sauf que, comme les terminaux, aucune assignation ne peut être effectuée. Voici par exemple la déclaration d'un flottant utilisant cette technique :

```
Double returns ecore::EDouble : INT('INT')?;
```

Notons que si aucun type de retour n'est indiqué, Xtext retournera par défaut un ecore::EString.

IV-B-4 - Types énumérés

Il est possible de construire des énumérations. Par exemple, la visibilité d'un package en Java serait décrite sous la forme d'une énumération (trois états : private, public, protected). Dans notre grammaire, le type de motorisation se prête bien à une énumération : regardons la règle R6 des motorisations.

```
// R6 : Enumeration des motorisations
enum Engines:
    PROPELLER = 'HELICES' | ENGINE = 'PROPULSEURS'
;
```

Nous définissons ici une énumération à deux éléments, PROPELLER ou ENGINE. L'utilisation dans le modèle de ces éléments sera effective suivant le texte rentré par l'utilisateur, respectivement 'HELICES' ou 'PROPULSEURS'.

IV-B-5 - Assignations

Lors de la construction des règles de notre grammaire, nous pouvons assigner les règles et les valeurs entrées dans l'éditeur à des variables qui seront présentes dans notre modèle. Ces assignations peuvent être de trois sortes :

- affectation simple : du type `maVariable = Regle` : ici l'objet recevra simplement la valeur retournée par la règle. Dans notre grammaire, cela est visible par exemple dans la règle R3 : « Avion name=ID ». Ici la variable « name » aura la valeur de l'ID entré dans l'éditeur. Notons à ce sujet que les variables nommées « name » ont une signification particulière dans Xtext : le framework assimilera automatiquement cette variable au nom de l'objet, qui sera affiché dans la vue Outline par exemple ;
- affectation multiple : du type `maListe += Regle` : ici l'objet `maListe` sera une liste de `Regle`. Autant de fois la règle apparaîtra dans le document, autant il y aura d'éléments dans la liste. Par exemple, dans la règle R1 « includes+=Includes », la variable « includes » contiendra la liste de tous les fichiers déclarés comme inclus dans ce fichier ;

- affectation conditionnelle : du type `monBooleen ?= 'TEXTE'` : ici le booléen recevra vrai ou faux suivant la présence ou non du texte indiqué à droite du signe '='. Par exemple dans la règle R5 « regular?=`'REGULIERE'`;`'` », la variable « regular » recevra « true » si l'utilisateur entre 'REGULIERE' dans son fichier.

Il est aussi possible d'écrire des règles sans assignation. Par exemple :

```
Regle : RegleA | RegleB | RegleC;
```

Il est parfois nécessaire de forcer l'instanciation d'un objet du modèle. Par exemple, si vous entrez la règle suivante :

```
Regle : (regle=Element)?;
```

Xtext vous affichera alors l'avertissement suivant :

```
The rule 'Regle' may be consumed without object instantiation. Add an action to ensure object creation, e.g. '{Regle}'.
```

Il suffit alors comme suggéré d'ajouter « {Regle} » juste après les '.', ce qui permettra d'avoir toujours une instance de Regle créée, même si aucun texte n'a été entré pour cette règle :

```
Regle : {Regle}(regle=Element)?;
```

IV-B-6 - Références

Xtext est capable de créer directement les hyperliens entre la déclaration des variables de votre DSL et leur utilisation dans les fichiers. Pour cela, un mécanisme de déclaration de références est utilisé, signalé par le positionnement de l'objet à lire entre crochets. Il est important de retenir que :

- seule la référence est entre crochets, pas la déclaration elle-même ;
- l'élément entre crochets fait référence à un objet de type EClass et non à une règle grammaticale ! ;
- par défaut, le terminal de l'élément entre crochets est considéré comme ID. Le terminal indique en fait le texte qui va être concrètement remplacé par le lien. On peut spécifier un autre terminal en ajoutant un pipe ('|') derrière l'élément, suivi du terminal désiré.

Dans notre grammaire, deux types de références existent : les références aux codes OACI des aéroports et aux noms des avions. Observons d'abord le cas le plus simple : les avions. La règle R3 « Plane » va implicitement créer le type Plane qui comporte un identifiant de type ID. Nous pouvons alors dans la déclaration des lignes aériennes créer une référence vers le type Plane grâce à l'instruction « `'Avion':plane=[Plane];` ». Comme mentionné précédemment, inutile de spécifier un type particulier car par défaut le type cherché est 'ID'.

Dans le cas des aéroports, le terminal n'est pas ID mais de type OACI. Il faut donc préciser le type de terminal qu'Xtext doit rechercher afin de créer les liens entre données, sous la forme « `airport=[CodeOACI|OACI];` » :

```
// R5 : Declaration des lignes aerienues
Airline:
    'Ligne' name=STRING ':'
    ('Avion':plane=[Plane]);
    & 'Compagnie':company=STRING;
    & 'Depart':departure=[CodeOACI|OACI];
    & 'Arrivee':arrival=[CodeOACI|OACI];
    & 'Duree':duration=DURATION;
    & regular?='REGULIERE';
    'End.'
;
```

Déclaration des références

IV-B-7 - Imports de fichiers

Xtext fournit des outils pour gérer les imports de fichiers. À ce titre, le framework est capable de gérer les références que nous venons de décrire au travers de plusieurs fichiers, grâce à un mécanisme d'import de fichiers par leur URI. Pour indiquer à Xtext quels éléments correspondent à des imports, il suffit de les marquer avec l'attribut « importURI », comme dans notre règle R2 « '%include'<'importURI=INCLUDE>,' ». Parfois ce mécanisme nécessite d'être affiné mais cela sera étudié dans le prochain article.

IV-B-8 - Règles de grammaire


D'autres règles existent, en plus de celles citées dans les paragraphes précédents. Tout d'abord, on peut spécifier que l'ordre des règles est indéfini, mais que les règles doivent apparaître au moins une fois. Dans l'ensemble de nos déclarations par exemple, la liste des paramètres n'a pas un ordre défini, mais tous doivent être présents. Cela est réalisé en séparant les règles par le symbole '&'. Un autre exemple simple est celui des préfixes de méthodes et d'attributs en Java : on peut écrire 'public static final' ou 'final public static'.

Enfin, on peut aussi forcer la reconnaissance d'une règle pour lever par exemple une indétermination. En effet, dans le cas de conditionnelles imbriquées telles que :

```
if (condition())
    if (autreCondition())
        maMethode();
    else
        autreMethode();
```

Dans un cas tel que celui-ci, impossible de prédire avec précision à quel 'if' appartient le 'else'. Xtext propose de spécifier au parser quelle règle doit être consommée en priorité, comme le montre l'exemple ci-dessous :

```
Condition:
    'if' '(' condition=BooleanExpression ')'
    then=Expression
    (=>'else' else=Expression)?
;
```

Nous avons vu dans cette partie les éléments essentiels à la construction d'une grammaire Xtext assez fournie et complexe. Pour toute référence supplémentaire, je ne peux qu'à vous enjoindre de consulter directement la  **documentation officielle**, plus fournie. Une fois notre fichier créé, il est temps de passer à la génération de la grammaire.

IV-C - Génération de l'infrastructure associée à la grammaire

IV-C-1 - Le fichier MWE2

Le fichier « GenerateAir.mwe2 » situé dans le même dossier que le fichier « Air.Xtext » définit la liste des fragments qui vont être utilisés par Xtext pour la génération de l'éditeur. Le tableau suivant récapitule certains fragments et leur fonction :

Fragment	Éléments générés
EcoreGeneratorFragment	Code EMF pour les modèles générés
XtextAntlrGeneratorFragment	Grammaire ANTLR, parser, lexer, services associés et modèle Ecore
GrammarAccessFragment	Accès à la grammaire dans le code
ResourceFactoryFragment	Constructeur de ressources EMF
ParseTreeConstructorFragment	Sérialisation modèle vers texte
ImportNamespacesScopingFragment	Gestion du scope des imports
JavaValidatorFragment	Validation de modèle
FormatterFragment	Formateur de code
LabelProviderFragment	Label Provider
OutlineTreeProviderFragment	Construction de la vue Outline
JavaBasedContentAssistFragment	Autocomplétion basée sur Java
XtextAntlrUiGeneratorFragment	Autocomplétion basée sur ANTLR

On peut choisir les fragments que l'on souhaite générer. D'autres fragments sont présents, par exemple pour le support des tests JUnit, le refactoring ou la génération d'un wizard de création de projets pour nos fichiers. Plus particulièrement, les sections qui vont nous intéresser sont les sections de gestion des imports et des liens.

```
// scoping and exporting API
// fragment = scoping.ImportURIScopingFragment {}
// fragment = exporting.SimpleNamesFragment {}

// scoping and exporting API
fragment = scoping.ImportNamespacesScopingFragment {}
fragment = exporting.QualifiedNamesFragment {}
fragment = builder.BuilderIntegrationFragment {}
```

Valeurs par défaut

Par défaut, les liens sont recherchés à travers tous les fichiers existants, ce qui pose problème pour une gestion claire et propre des fichiers inclus. Nous allons donc modifier cette section de la manière suivante, afin que la création des liens se fasse au sein d'un unique fichier et de ses imports :

```
// scoping and exporting API
fragment = scoping.ImportURIScopingFragment {}
// fragment = exporting.SimpleNamesFragment {}

// scoping and exporting API
// fragment = scoping.ImportNamespacesScopingFragment {}
fragment = exporting.QualifiedNamesFragment {}
fragment = builder.BuilderIntegrationFragment {}
```

Nouvelles valeurs

IV-C-2 - Lancement de la génération

Une fois ces deux fichiers préparés, nous pouvons lancer la génération de notre grammaire. Ceci peut se faire de deux façons :

- soit par clic droit sur le fichier Xtext > « Run As > Generate Xtext Artifacts »
- soit par clic droit sur le fichier MWE2 > « Run As > MWE2 Workflow »

Cette action va lancer la génération de tous les éléments nécessaires à notre éditeur. Cette étape peut être longue, suivant la taille de la grammaire. La première fois qu'on lance la génération, le message suivant apparaît dans la console Eclipse :

ATTENTION It is recommended to use the ANTLR 3 parser generator (BSD licence - <http://www.antlr.org/license.html>). Do you agree to download it (size 1MB) from '<http://download.itemis.com/antlr-generator-3.2.0.jar>'? (type 'y' or 'n' and hit enter)

Il suffit de répondre 'y' et la génération continue. Ce JAR ne peut pas être inclus par défaut avec Xtext pour des problèmes de licence : le code généré par ANTLR et utilisé par Xtext n'est pas contraint en licence. Il n'utilise pas une « librairie/runtime » (par exemple) qui pourrait avoir une licence contaminante. À la fin de la génération, tous les dossiers « src-gen » de vos plugins devraient être fournis avec un certain nombre de classes. Dans le plugin « ui », le dossier « src » contient le package « org.Xtext.ui » qui nous permettra de personnaliser notre IDE.

IV-C-3 - Résoudre les erreurs : LL-recursivity

Lorsque Xtext rencontre un problème lors de la génération de la grammaire, c'est quasiment systématiquement suite à la vérification de celle-ci par le parser ANTLR. Si la plupart des erreurs sont compréhensibles et donc assez faciles à corriger, il en est une qui revient très couramment mais pourtant n'est pas explicite. Cette erreur apparaît sous cette forme :

[fatal] rule MaRegle has non-LL(*) decision due to recursive rule invocations reachable from alts x,y. Resolve by left-factoring or using syntactic predicates or using backtrack=true option.

i Les nouvelles versions d'Xtext savent détecter cette erreur dans l'éditeur directement. Ce n'était pas le cas avec des versions plus anciennes, auquel cas ce paragraphe pourra vous dépanner !

Cette erreur est levée par une expression comme celle-ci :

```
Expression :
    Expression '+' Expression |
    '(' Expression ')' |
    INT
;
```

Pour corriger cette erreur, il suffit de factoriser la règle de cette manière :

```
Expression :
    TerminalExpression ('+' TerminalExpression)?
;
TerminalExpression :
    '(' Expression ')' |
    INT
;
```

IV-D - Test de l'éditeur

IV-D-1 - Créer la configuration

Une fois notre grammaire générée, il est temps de la tester au sein d'un éditeur. Pour cela, une méthode simple est de faire un clic droit sur le projet « *.ui » et de choisir « Run As > Launch Runtime Eclipse ». Cette option va lancer une copie exacte de votre instance Eclipse en cours d'exécution, ce qui peut entraîner un temps de chargement assez long pour de simples tests. L'autre solution consiste à définir soi-même une configuration au sein du panneau « Run

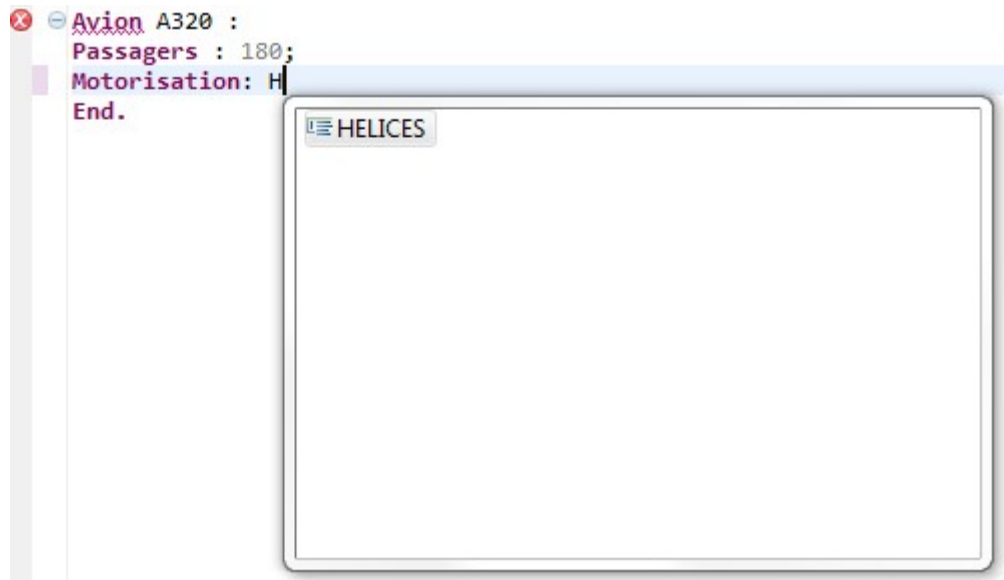
configurations » d'Eclipse et de ne sélectionner que les plugins nécessaires à l'exécution du plugin « *.ui » (bouton « Add required plugins »). Dans l'onglet « Main », l'option choisie dans la section « Program to Run » est « Run a product : org.eclipse.sdk.ide »



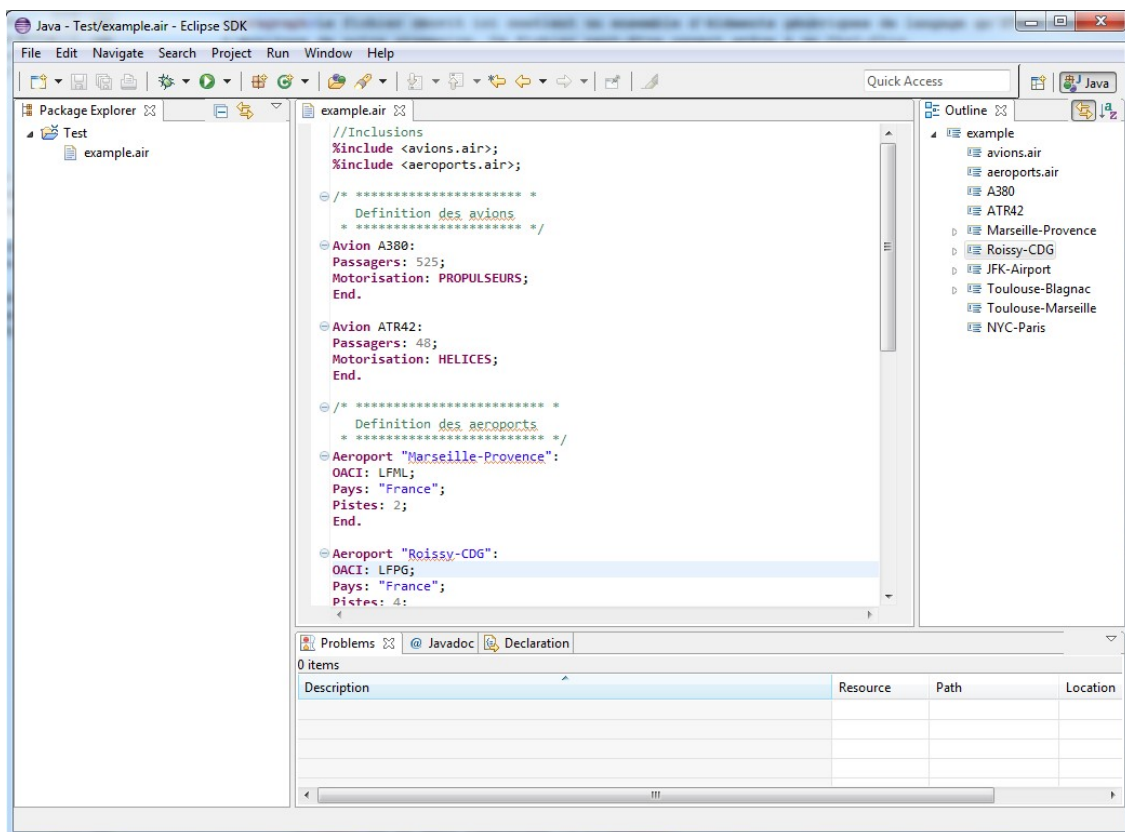
IV-D-2 - Testons notre grammaire !

Une fois notre instance de test lancée, il suffit de créer un nouveau projet vide « File > New > Project », et de créer à l'intérieur de ce projet un nouveau fichier « .air ». Xtext vous proposera alors de configurer votre projet pour être utilisé avec Xtext :

Vous pourrez donc profiter de l'autocomplétion et des autres outils associés pour créer vos fichiers.



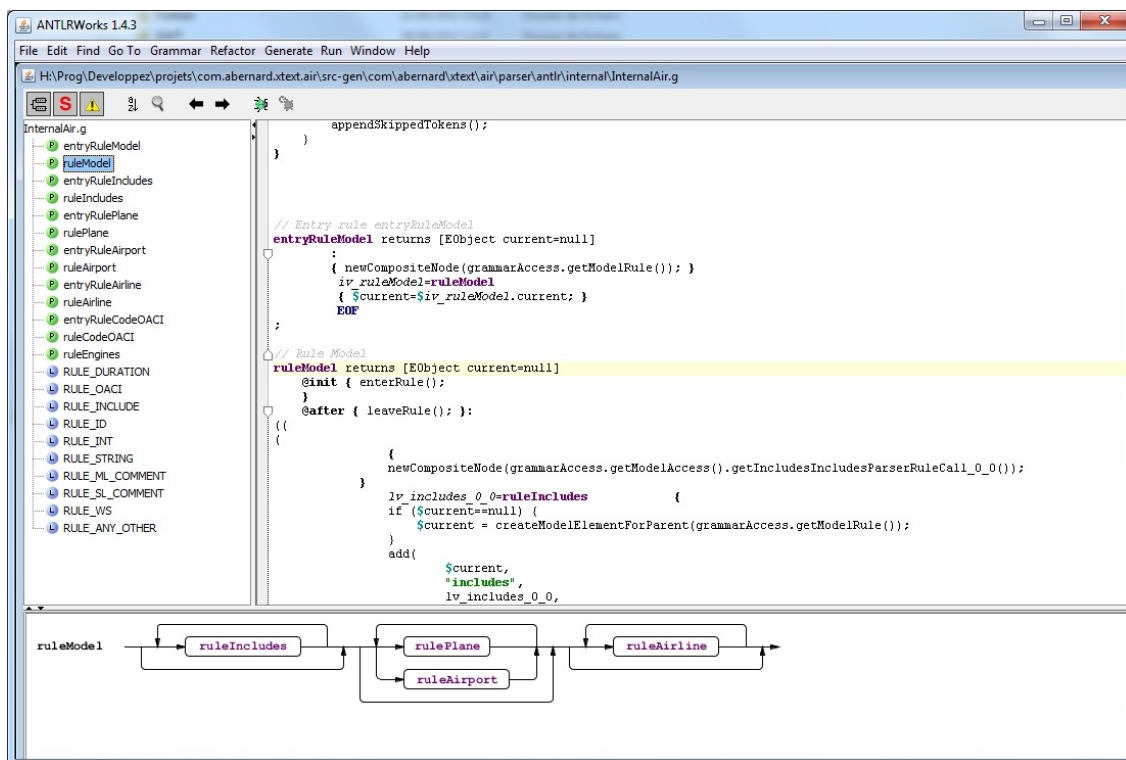
On peut d'ailleurs observer le résultat avec notre fichier d'exemple présenté en début de cet article. Xtext nous fournit bien un environnement convivial de travail, bien plus qu'un simple éditeur classique. Notez la présence de la Outline déjà configurée avec les attributs de la grammaire nommés « name » !



Notre IDE

V - Un outil bien pratique : ANTLRWorks

Il est parfois difficile de corriger des erreurs remontées lors de la génération de la grammaire, plus particulièrement celles exprimant un non-déterminisme de la grammaire (i.e. lorsque le parser ne peut pas « choisir » entre deux règles à suivre). L'outil ANTLRWorks permet de visualiser la grammaire sous forme de graphe et permet donc de corriger les erreurs plus facilement. ANTLRWorks est [téléchargeable ici](#). Une fois lancé, il suffit d'ouvrir le fichier « .g » qui a été généré dans Eclipse. Il se trouve dans le package « src-gen/%votreprojet%.parser antlr.internal ». On peut alors naviguer au sein des différentes règles de la grammaire et éventuellement voir au sein des graphes affichés si une règle comporte un non-déterminisme. Pour notre projet par exemple, on peut afficher la règle de base :



ANTLRWorks

L'intérêt est aussi de prendre un élément concret du langage et de pouvoir voir en quoi il est parsé.

VI - Conclusion

Dans cet article nous aurons étudié les principales règles qui régissent la construction d'une grammaire sous Xtext. Nous avons cependant évoqué uniquement les plus grands principes et la richesse de la grammaire Xtext ne s'arrête pas là. La documentation officielle vous fournira éventuellement les réponses aux questions que vous pourriez encore vous poser.

Nous avons pu aussi constater qu'à partir d'un seul fichier de définition, le framework est capable de proposer un environnement de développement complet et convivial. En plus de proposer les outils nécessaires à l'exploitation efficace des fichiers d'un DSL, cet environnement est hautement personnalisable et ce sera le sujet d'un prochain article, focalisé sur la personnalisation de l'IDE.

VII - Liens utiles

-  [Site officiel Xtext](#)

Articles de Georges KEMAYO sur developpez.com :

- **Introduction à Xtext** : présentation rapide de la mise en place d'une grammaire Xtext pour un DSL.
- **Développement des plugins associés à Xtext** : présentation de la personnalisation de l'IDE généré par Xtext.
- **Manipulation d'un DSL basé sur Xtext** : manipulation du modèle objet d'Xtext pour effectuer des opérations.

VIII - Remerciements

Je tiens à remercier toute l'équipe Eclipse de developpez.com de m'avoir conseillé dans l'écriture de cet article, tout particulièrement **Mickaël BARON** et Sylvain CAMBON. Je remercie aussi **Claude LELOUP** et **_Max_** pour leur relecture orthographique attentive.