Atelier 5 :

« Xtext : Un DSL pour la création des entités »

Faculté des sciences - Centre d'Excellence IT (Filière : IL - S3)

Module : MDE

Pr: **Y. AIT LAHCEN**

Objectifs :

— Implémenter un DSL pour la modélisation des entités et des propriétés.

Environnement d'exécution :

— Eclipse

— xtext plug-in correspondant à votre plateforme et que vous pouvez télécharger sous le lien suivant :

— https://eclipse.dev/Xtext/download.html ou

— https://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/

1 Concepts les plus importants du langage de grammaire Xtext

1.1 Création d'un nouveau projet

À partir d'Eclipse, créez un nouveau projet :

(a) Menu **File** → **New** → **Project...**→ **Xtext Project**.

(b) Proposez un nom à votre projet, à votre langage et aux extensions. Exemple :

— `Projet name : org.xtext.example.entity.`

— `Language name : org.xtext.example.entity.Entity.`

— `DSL File extension : entity.`

Cliquer sur **Finish** pour permettre à Xtext de créer les projets associé à votre DSL à savoir :

TABLE 1 – Projets générés automatiquement par Xtext

Projet	Description
`org.xtext.example.entity`	Contient la grammaire et toutes les composantes du langage (parser, lexer, linker, validation, etc.)
`org.xtext.example.entity.tests`	Fournit les unités de test pour le langage.
`org.xtext.example.entity.ide`	Regroupe les fonctionnalités principales liées à l'IDE.
`org.xtext.example.entity.ui`	Contient l'éditeur Eclipse et d'autres fonctionnalités d'intégration.
`org.xtext.example.entity.ui.tests`	Fournit les unités de test pour l'éditeur Eclipse.

1.2 Définition de la grammaire de notre DSL

Comme vous avez remarqué, l'assistant de création de projet Xtext a automatiquement ouvert le fichier *Entity.xtext* qui contient une simple grammaire de *Hello World*.

Entity.xtext

```
1  grammar org.xtext.example.entity.Entity with org.eclipse.xtext.common.Terminals
2
3  generate entity "http://www.xtext.org/example/entity/Entity"
4
5  Model:
6      greetings+=Greeting*;
7
8  Greeting:
9      'Hello' name=ID '!';
```

Maintenant on va supprimer le contenu de ce fichier pour définir la grammaire de notre DSL.

entrer les deux lignes suivantes :

```
grammar org.xtext.example.entity.Entity with org.eclipse.xtext
.common.Terminals
generate entity "http://www.xtext.org/example/entity/Entity"
```

La première ligne indique une nouvelle grammaire *Entity* dirivée de la grammaire *Terminals* qui définit quelques régles basiques (STRING, ID, INT, etc.) la seconde définit le nom et le namespace URI de notre grammaire.
Continuons à difinir les déférentes règles de notre grammaire :

(a) La première règle est toujours la règle de démarrage.
Model :

```
1    (types+=Type)*
```

Cela signifie que `Model` contient un nombre arbitraire (*) de `Type` qui sont ajoutés (`+=`) à l'attribut `types`.

(b) La règle Type.
Type :

```
1    TypeDef | Entity;
```

La règle `Type` peut être soit un `TypeDef`, soit une `Entity` (notion d'héritage).

(c) La règle TypeDef
TypeDef :

```
1    "typedef" name=ID ("mapsto" mappedType=JAVAID)?;
```

Cette règle commence par le mot-clé `"typedef"`, suivi d'une règle `ID` affectée à l'attribut `name`. La règle `ID` est définie dans la grammaire *org.eclipse.xtext.common.Terminals*. Après l'attribut `name`, on peut facultativement (le `?`) ajouter la clause `mapsto`. Le fragment `mappedType=JAVAID` spécifie que `TypeDef` peut avoir un attribut nommé `mappedType` de type `JAVAID`.

(d) Définition de la règle JAVAID
JAVAID :

```
1    name=ID("." ID)*
```

D'après sa définition, la règle `JAVAID` est une séquence d'`ID` et de points, permettant ainsi la définition des types Java, par exemple : `java.util.Date`.

(e) Définition de la règle Entity
Entity :

```
1    "entity" name=ID ("extends" superEntity=[Entity])?
2    (attributes+=Attribute)*
3    "}";
```

Cette règle commence par le mot-clé `"entity"` suivi d'un `ID` pour `name`. Elle peut facultativement (`?`) hériter d'une autre entité (`superEntity`) déjà existante. Ce référencement est exprimé par les deux crochets `[ ]`. Ensuite, elle peut contenir zéro ou plusieurs attributs (`attributes+=Attribute`).

(f) Définition de la régle Attribute.

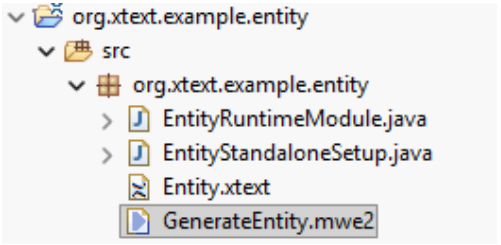
```
1    Attribute:
2    (many ?= "*")? name=ID ":" type=[Type];
```

Un `Attribute` peut avoir `(?=` qui signifie un type booléen), un indicateur de multiplicité `(*)`, un nom (`name`) et un type qui fait référence à un `Type` (qui peut être un `TypeDef` ou une `Entity`).

À présent, votre grammaire utilise les concepts les plus importants du langage de grammaire Xtext.

1.3 Compilation du DSL

A ce stade on va se contenter à la compilation de notre grammaire. Pour ce faire, Xtext créera les composants suivants : un parseur, un éditeur de texte, un sérialiser et bien d'autres encore.
Sélectionner le fichier :
- *org.xtext.example.entity/src/org/xtext/example/entity/GenerateEntity.mwe2*
- et faire un clique-droit, puis sélectionner *Run As*→ MWE Workflow.



1.4 Exéction du plug-in Eclipse

Testons maintenant l'éditeur de notre DSL. Sélectionner le projet `org.xtext.example.entity`, puis choisir :

— **Run As → Eclipse Application**

Une nouvelle instance d'Eclipse s'ouvrira.

Dans cette nouvelle instance :

(a) Créer un nouveau projet :
File → New → Other... → General → Project .
(b) Dans le dossier `src`, créer un nouveau fichier avec l'extension `.entity` définie dans la grammaire.

Cela permettra le lancement de l'éditeur correspondant à notre DSL.

Découvrir les différentes fonctionnalités de votre éditeur :

— l'auto-complétion
— la coloration syntaxique
— la validation syntaxique
— la gestion des références

Créer maintenant le modèle suivant en utilisant la syntaxe de votre DSL :

— **Un typedef** nommé `String`
— **Un typedef** nommé `Integer`
— **Un typedef** nommé `Date` et qui fait référence au type Java `java.util.Date`
— **Une entité** `Person` avec les attributs :
  — `name` de type `String`
  — `surName` de type `String`
  — `birthDay` de type `Date`
  — `home` de type `Address`
  — `work` de type `Address`
— **Une entité** `Chef` qui étend `Person` avec l'attribut :
  — Plusieurs employees qui font référence à PersontextttPerson
— **Une entité** `Address` avec les attributs :
  — `street` de type `String`
  — `number` de type `String`
  — `city` de type `String`
  — `ZIP` de type `String`

2 Autres notions

2.1 Ajout de packages et Imports

Dans cette partie, nous allons améliorer notre DSL. En effet, notre langage doit supporter la notion de *package* afin :

— d'éviter les conflits de noms,
— de mieux structurer les modèles,
— de faciliter la génération de code pour différentes cibles (Java, C++, etc.).

Un **package** peut contenir des `Type` et/ou d'autres packages. Pour pouvoir utiliser le contenu d'un package, il est nécessaire de déclarer un **import**.

Afin de rendre notre modèle plus modulaire, nous souhaitons redéfinir le modèle précédent en le répartissant sur **trois fichiers**.

main.entity
Une entité Chef qui étend Person avec l'attribut :
— Plusieurs employees qui font référence à **Person**

commons.entity
attributs : Une entite Person avec les :
— name de type String
— surName de type String
— birthDay de type Date
— home de type Adress
— work de type Adress

datatypes.entity
— Un typedef nommé String
— Un typedef nommé Integer
— Un typedef nommé Date et qui fait reference au type Java java.util.Date

Pour ce faire, on commencera à enrichir notre DSL.

(a) Maintenant notre modèle doit contenir, en plus des `types`, des `packages`. Donc la première règle de notre DSL doit être modifiée. Ainsi, un super type pour `Packages` et `types` sera défini : `AbstractElement`.

```
1    Model:
2        (types+=AbstractElement)*;
3
4    AbstractElement:
5        PackageDeclaration | Type;
```

(b) Le `PackageDeclaration` contient un nombre arbitraire d'`Imports` et d'`AbstractElements`. On mettra à jour la règle `AbstractElement`.

```
PackageDeclaration:
    "package" name=QualifiedName
    "{"
        (types+=AbstractElement)*
    "}";
```

```
AbstractElement:
    PackageDeclaration | Type | Import;

QualifiedName:
    ID ("." ID)*;
```

La règle `QualifiedName` est un peu spéciale. Elle ne contient aucune affectation. Par conséquent, elle sert comme une règle qui retourne un `String`.

(c) Définition de la règle Import

```
Import:
    "import" importedNamespace =
       QualifiedNameWithWildcard;
```

```
QualifiedNameWithWildcard:
    QualifiedName ".*"? ;
```

La règle `QualifiedNameWithWildcard` est similaire à la règle `QualifiedName` et retourne aussi une chaîne de caractères.

(d) La dernière étape consiste à permettre un référencement au `QualifiedName`. Sinon, on ne peut pas se référer à une entité sans l'emploi de l'instruction `import`.

```
Entity:
    "entity" name=ID
    ("extends" superType=[Entity | QualifiedName])
       ?
    "{"
        (attributes+=Attribute)*
    "}" ;

Attribute:
    (many ?= "*")? name=ID ":" type=[Type |
       QualifiedName];
```

C'est tout ce qu'il faut pour votre grammaire. Reste maintenant à recompiler votre DSL et à tester les nouvelles modifications. Refaire le modèle précédent en le décomposant en trois fichiers, comme décrit en début de cette partie.

Annexe : →

Code 1 – Entity DSL Model

```
package datatypes {
    typedef String
    typedef Integer
    typedef Date mapsto java.util.Date
}
package commons{
    import datatypes.*
    entity Address {
        street : String
        number : String
        city : String
        ZIP : String
    }
    entity Person {
    name : String
    surName : String
    birthDay : Date
    home : Address
    work : Address
    }
}
package main {
    import commons.*
    entity Chef extends Person {
        * employees : Person
    }
}
package datatypes {
    typedef String
    typedef Integer
    typedef Date mapsto java.util.Date
}
```

Fin de l'atelier : Xtext : Un DSL pour la création des entités.