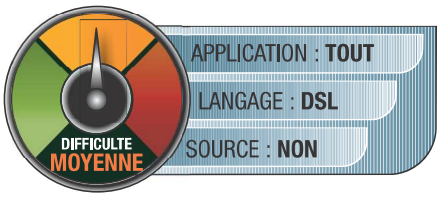
Créez votre propre langage et son éditeur Eclipse avec Xtext

1re partie

On ne crée pas de nouveau langage tous les jours. C'est une tâche fastidieuse, qui requiert des compétences très spécifiques et beaucoup de temps devant soi. Les outils traditionnels de création d'analyseurs lexicaux et de parseurs comme Lex ou Yacc sont d'ailleurs des noms qui font froid dans le dos et les outils plus récents comme Antlr, bien que beaucoup plus abordables restent complexes à apprivoiser.



De surcroît, créer un langage n'est déjà pas simple, mais lui créer un éditeur est un investissement bien plus lourd encore, réservé aux plus téméraires. Or sans un éditeur approprié, un langage a peu de chance de satisfaire ses utilisateurs. Xtext est un nouveau framework de création de DSL textuelle (Domain Specific Language ou langage dédié) qui change la donne. Ce framework révolutionnaire vous permet de créer un nouveau langage et son éditeur riche pour Eclipse, très simplement. L'éditeur résultant est, de surcroît, doté de la plupart des fonctionnalités avancées que l'on trouve uniquement dans des éditeurs sophistiqués comme l'éditeur Java du JDT Eclipse : coloration syntaxique, coloration sémantique, validation à la saisie, complétion contextuelle, liaison croisée, indexation continue, multi-fichiers, espaces de nommages, reprise sur erreur syntaxique, formatage automatique et propositions de templates, pour ne mentionner que les fonctionnalités essentielles. Ce premier article d'une série sur Xtext et Eclipse RCP vise à vous mettre le pied à l'étrier par la pratique en vous guidant dans l'installation de l'environnement et dans la création de votre premier langage. Dans les articles suivants, nous irons plus loin dans les fonctionnalités avancées de Xtext.

INSTALLATION DE L'ENVIRONNEMENT DE CRÉATION DE DSL

L'environnement Xtext est disponible sur le site http://www.xtext.org. Le plus simple pour commencer est de télécharger une distribution préconfigurée, mais les habitués d'Eclipse préféreront sans doute installer Xtext depuis l'update-site.

CRÉATION DE VOTRE PREMIER PROJET

Après avoir démarré la distribution, créez un nouveau projet Xtext en lançant l'assistant de création de projet Xtext.
Cet assistant vous demande le nom du projet principal, le nom du langage et l'extension que vous souhaitez utiliser pour votre éditeur. Il vous demande aussi si vous souhaitez créer un projet de génération de code [Fig.1]. Le projet Xtext matérialisé dans le workspace Eclipse est immédiatement fonctionnel.

LA GRAMMAIRE, LA PIERRE ANGULAIRE DE XTEXT

Le projet généré contient un exemple de grammaire traitant la problématique classique de modélisation d'entités métiers. À partir de cette seule grammaire, Xtext va générer pour vous toute l'infrastructure technique qui compose votre langage et son éditeur. Voici un aperçu de ce que l'éditeur généré nous permettra de saisir :

```
type string

entity Node {
    property name : string
}

entity Folder extends Node {
    property items : Node []
}
```

Pour comprendre comment obtenir un tel éditeur, penchons-nous sur notre grammaire en commençant par l'en-tête :

```
grammar com.proxiad.articles.xtextseries.IntroDsl
    with org.eclipse.xtext.common.Terminals
```

L'en-tête de notre grammaire est purement technique. Nous retrouvons tout d'abord le nom de notre grammaire renseigné au préalable dans l'assistant de création de projet. Le mot clé `with` spécifie que notre grammaire hérite d'une autre grammaire : `org.eclipse.xtext.common.Terminals`, mise à disposition par Xtext.
L'en-tête définit aussi un nom unique qui permet d'identifier notre DSL. Ce nom est distinct du nom de la grammaire :

```
generate introDsl
    "http://www.proxiad.com/articles/xtextseries/IntroDsl"
```

Inutile pour le moment de nous attarder sur cet en-tête, ces informations régissent simplement l'infrastructure générée par Xtext. Poursuivons en nous intéressant à la première règle définie dans notre grammaire, `Model` :

```
Model :
    (imports+=Import)*
    (elements+=Type)*;
```

Cette règle tout comme chaque règle Xtext définit deux points de vue : d'une part la structure du modèle, d'autre part la syntaxe du langage.
Ici, la règle `Model` spécifie que sur le plan structurel, une classe `Model` est composée de deux attributs : une liste d'`Import` nommée `imports` et une liste de `Type` nommée `elements`. Sur le plan syntaxique, cette règle précise qu'un fichier peut commencer par plusieurs imports suivis de plusieurs éléments. Ici, il ne sera pas possible de placer des types avant des imports. Poursuivons précisément avec la définition de la règle `Import` :

```
Import :
    'import' importURI=STRING;
```

Sur le plan lexical, la règle `Import` indique que la classe `Import` sera instanciée lorsque le mot clé `'import'` sera rencontré. Sur le plan structurel, la règle `Import` indique que la classe Import contient un

attribut `importURI` dont le type est déterminé par le terminal `STRING`. Un terminal est une règle dite lexicale. Contrairement à la règle `Import` qui spécifie une classe complexe `Import` constituée d'attributs, une règle lexicale définit une correspondance entre une séquence de caractères (exprimée avec une expression régulière) et un type primitif, comme par exemple une chaîne de caractères, un booléen ou un entier. Par ailleurs, le terminal `STRING` est défini dans la grammaire héritée par notre DSL. Celle-ci propose un certain nombre de terminaux couramment utilisés. La définition du terminal `STRING` est un peu complexe, il suffit de retenir pour le moment que le terminal `STRING` correspond à une chaîne de caractères, mais les plus curieux pourront aller voir la définition du terminal en effectuant un `CTRL-Clic` sur le nom du terminal. L'attribut `importURI` de la classe `Import` aura donc comme type une chaîne de caractères. Par ailleurs, l'attribut `importURI` est reconnu par Xtext comme spécial et permet de gérer l'import de fichier. Cela signifie qu'il sera possible de référencer depuis le fichier en cours, des éléments définis dans un autre fichier, tout comme le mot clé `import` en Java permet de référencer des classes et méthodes définies dans une autre classe.

```
Type:
    SimpleType | Entity;
```

La règle Type indique ici qu'un Type peut être soit un SimpleType, soit une Entity. Sur le plan structurel, cela se traduit par une règle d'héritage, les classes SimpleType et Entity étendent la classe Type. Sur le plan syntaxique, cela signifie que lorsque l'on rencontre la règle Type, on peut saisir soit un SimpleType soit une Entity.

```
SimpleType:
    'type' name=ID;
```

La règle SimpleType est similaire à la règle Import. Sur le plan structurel elle définit la classe SimpleType et son attribut name. Cette règle permet de saisir le texte suivant, d'abord le mot clé `type` puis la valeur de l'attribut `name` :

```
type mytype
```

L'attribut name est cette fois-ci défini par le terminal ID. Tout comme le terminal `STRING`, `ID` fait partie des terminaux prédéfinis par Xtext. Voyons cette fois-ci plus en détail la définition de ce terminal :

```
terminal ID returns ecore::EString :
    ('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
    ;
```

Un `ID` est une chaîne de caractères qui ne peut contenir que des lettres minuscules ou majuscules, des underscores et des chiffres, mais ne peut commencer par un chiffre. La classe retournée par cette règle est `ecore::EString`, c'est une classe de base d'EMF représentant une chaîne de caractères. Il est à noter que « `returns ecore::EString` » est optionnel car, par défaut, un terminal renvoie une `EString`. Retournons à notre grammaire pour terminer avec les dernières règles `Entity` et `Property`, qui constituent le cœur de notre grammaire :

```
Entity :
    'entity' name=ID ('extends' extends=[Entity])? '{'
        properties+=Property*
    '}';

Property:
    'property' name=ID ':' type=[Type] (many?='[]')?;
```

Avec la règle Type, les règles Entity et Property permettent de saisir le code illustré en aperçu plus haut :

```
type string

entity Node {
    property name : string
}

entity Folder extends Node {
    property items : Node []
}
```

Sur le plan structurel, la règle `Entity` définit la classe `Entity` et ses attributs `name`, `extends` et `properties`. L'attribut `name` est une chaîne de caractères de type ID. L'attribut `properties` est une liste de `Property`.

```
('extends' extends=[Entity])?
```

L'extrait ci-dessus indique deux choses, tout d'abord, le point d'interrogation indique que l'attribut `extends` est optionnel. Sur le plan structurel, cela signifie que l'attribut `extends` pourra ne pas être renseigné et sur le plan syntaxique, cela signifie que l'utilisateur pourra ne pas taper `'extends'`. Dans cet extrait, la syntaxe `[Entity]` définit également que l'attribut `extends` référence une autre `Entity` (au sens de classe `Entity`, et non de la règle `Entity`), définie par ailleurs dans le modèle. Il s'agit ici de liaison croisée, ou « cross linking » en anglais. Le cross linking est très riche dans Xtext. Avec la seule information `[Entity]`, Xtext va gérer plusieurs fonctionnalités très intéressantes pour l'utilisateur final :

1. la complétion en proposant à l'utilisateur lors d'un `CTRL ESPACE` une liste des autres `Entity` définies dans le fichier : [Fig.2]
2. valider que l'`Entity` référencée existe
3. et l'utilisateur pourra également naviguer vers la définition de l'élément référencé en faisant un `CTRL clic` sur la référence : [Fig.3]

L'INFRASTRUCTURE GÉNÉRÉE PAR XTEXT

Après avoir étudié la grammaire, regardons maintenant les différentes composantes de l'infrastructure générée. Pour lancer la génération, faire un clic droit sur le fichier GenerateIntroDsl.mwe, puis sélectionner Run As > MWE Workflow. Xtext. Cela va générer :

- **Un méta-modèle** : Ce méta-modèle définit les concepts de notre domaine exprimés par la grammaire sous la forme de classes et attributs. Souvenez-vous, le nom unique de ce méta-modèle était défini dans l'en-tête de la grammaire et vous devez retrouver chacune des classes définies par la grammaire : [Fig.4]
- **Un analyseur lexical** (ou lexer) et un **parseur arborescent** instanciant directement le méta-modèle
- **Un cadre de formatage automatique du code** : Intégré au `CTRL-SHIFT-F`. La configuration de formatage doit, en revanche, être configurée à la main, c'est l'une des très rares fonctionnalités pour lesquelles Xtext ne peut fournir de comportement correct par défaut. Non configuré, le formatage fonctionne mais place tous les éléments sur une même ligne. C'est rarement le comportement souhaité.
- **Un cadre de Validation** permettant de définir des règles de validation sur le modèle. Les règles de validation sont définies selon trois coûts d'exécution, pour optimiser l'expérience utilisateur : coût faible, modéré et important. Les règles les moins coûteuses sont exécutées immédiatement à la saisie donnant à l'utilisateur un retour instantané sur l'état de ce qu'il tape. Les règles au coût modéré sont exécutées à l'enregistrement du fichier. Enfin, les plus

coûteuses doivent être exécutées explicitement par l'utilisateur.

- **Un éditeur pour Eclipse** supportant la complétion, la coloration syntaxique, une outline et intégrant les règles de validation dans l'éditeur. L'outline fournit à l'utilisateur une vision du modèle construit en mémoire tandis qu'il édite/modifie son code. :
- **Un cadre de gestion de la portée de visibilité des éléments**, « scoping » en anglais. Dans la grammaire présentée ici, les éléments sont définis de manière globale, mais Xtext permet de contrôler de manière très fine la portée des éléments.

EXÉCUTION DE L'ÉDITEUR

Pour tester notre éditeur, le plus simple consiste à sélectionner le projet principal, à faire un clic droit et à sélectionner `Run As > Eclipse Application`. Dans l'environnement nouvellement ouvert, créer un nouveau projet simple et créer un fichier dont l'extension est celle de notre langage. Pour vérifier que l'éditeur fonctionne, il suffit de commencer à taper ou plus simplement d'invoquer la complétion de code en tapant CTRL-ESPACE : [Fig.5]

MAIS QUE FAIRE DE NOTRE DSL MAINTENANT ?

Nous venons de voir à quel point il est facile d'écrire une DSL avec Xtext, mais à ce stade, notre DSL ne sert qu'à taper du code ou modéliser, c'est une question de point de vue qui dépend du niveau d'abstraction du domaine exprimé par notre DSL. Pour certains, cette possibilité en soi peut avoir beaucoup de valeur. Par exemple, des analystes financiers qui conçoivent des produits financiers peuvent souhaiter spécifier de manière formelle leurs produits et cette possibilité à elle seule peut justifier d'investir dans la création d'une DSL. Mais souvent, notre DSL est un point de départ. Quels sont alors les usages possibles que l'on peut associer à notre DSL ?

1. **Génération de code** : cette approche est l'usage le plus courant d'une DSL. Une DSL est souvent utilisée dans le cadre de projets informatiques pour générer du code et gagner en productivité, en qualité et en homogénéité. D'ailleurs, si dans l'assistant de création du projet, vous avez sélectionné la case à cocher « Create generator project », celui-ci aura matérialisé un autre projet de génération de code préconfiguré avec un exemple de template de génération de code et un exemple de modèle : [Fig.6]. Pour lancer la génération de code, faire un clic droit sur le fichier `IntroDslGenerator.mwe` et lancer `Run As > MWE Workflow`. Si tout se passe bien, un fichier `output.txt` est généré dans le dossier `src-gen`.
2. **Interpréteur** : Un autre usage possible d'une DSL est de l'interpréter. À titre d'exemple, un tableur interprète le langage des formules des cellules d'un tableau de calcul.

Mais nous pouvons aussi utiliser Xtext comme un framework de création d'éditeur pour un langage préexistant, non doté d'un éditeur pour Eclipse. Ce langage a donc déjà son utilité par ailleurs.

CONCLUSION

Xtext permet également de créer une syntaxe textuelle pour un modèle préexistant, d'intégrer votre langage à des langages de programmation existants comme Java avec le JDT, ou bien à des sources de données externes (schéma de base de données, annuaire JCR, annuaire de web services, annuaire d'artifacts Maven, etc.). Les possibilités sont vastes : tous les aspects de Xtext sont configurables, grâce à son utilisation de Guice, un moteur IoC, ou Inversion de Contrôle. Un tel moteur permet de remplacer de manière fine et ciblée chacun des composants de Xtext par une implémentation différente. Xtext est conçu pour créer rapidement de petits langages, mais sa versatilité et son extensibilité permettent d'envisager des langages plus importants et pourquoi pas des langages de programmation complets. C'est l'orientation que souhaitent prendre ses créateurs en ajoutant toujours plus de fonctionnalités rendant toujours plus simple la création de nouveaux langages. Dans les articles suivants, nous présenterons des exemples de DSL plus complexes illustrant des intégrations à d'autres frameworks de l'écosystème Eclipse.

■ Cédric **Vidal**
Architecte JEE et MDSD, Responsable Technique de ProxiAD Ile de France.
c.vidal@proxiad.com

Fig.1

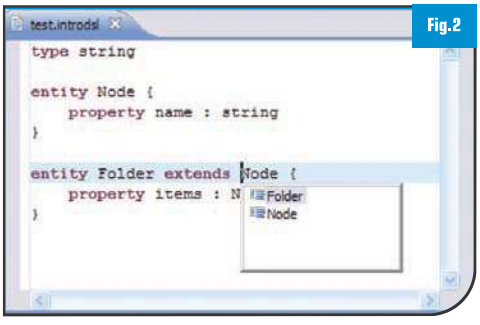


Fig.2

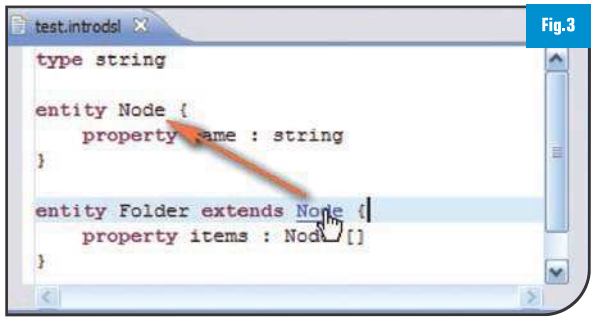


Fig.3

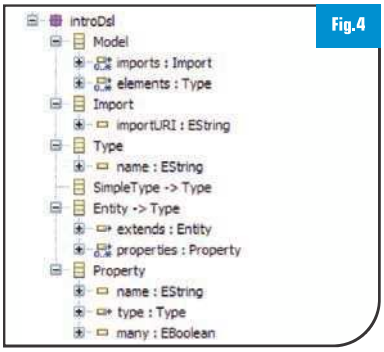


Fig.4

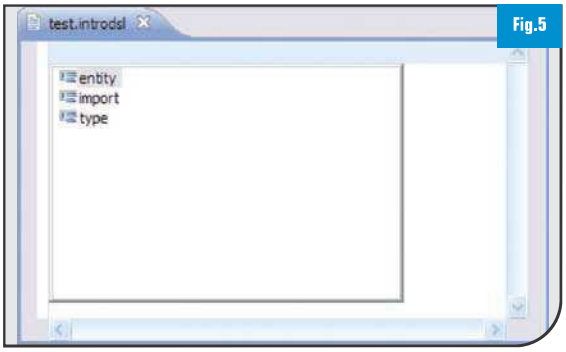


Fig.5

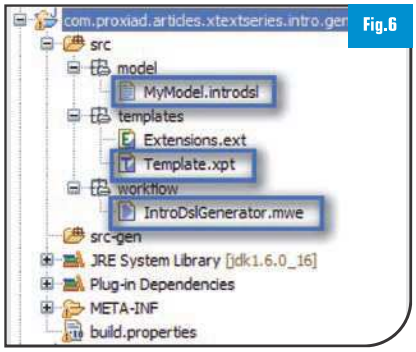


Fig.6