Yann Thierry-Mieg
Laboratoire d'Informatique Paris 6
M2 – SAR ue Ingénierie des Modèle IDM

Xtext pour créer des DSL textuels

EMF

- EMF : Eclipse modeling framework
  - Un cadre pour modéliser et méta-modéliser
  - Services autour du modèle
    - I/O : stockage au format standard XMI
    - Manipulation : genmodel et API Java
    - Validation : OCL
- Limites
  - Dépendance du format de stockage au MM (versions ?)
  - Synchronisation des artefacts et MM avec des formats textuels (M2T + T2M symétrique…)

GMF, Graphiti…

- Début 2000, propension au DSL semi-graphique
  - Poussé par UML
  - Frameworks type GMF, ou (plus moderne) Graphiti
- Limites rapidement atteintes
  - Passage à l'échelle du graphique
  - Customisation et mise en place trop lourdes (cout d'entrée), malgré des progrès
  - Au final des annotations munies d'une grammaire propre sont nécessaires pour échanger avec les humains (e.g. notes de code en UML)

Les DSL textuels

- Si on a des outils qui passent de grammaire à instance de modèle, pourquoi s'encombrer des graphismes ?
  - Au pire on pourra toujours faire des visualisations de parties du modèle (esprit diagramme UML)
- Des DSL textuels bien connus à succès depuis 30 ans : Make (compilation), GraphViz dot (graphes), SQL (BD relationnelles)
- Objectif : synchroniser méta-modèle et grammaire
  - Un parser classique construit un AST, qu'il faut ensuite monter vers les structures de données de stockage et de manipulation de l'application
  - On souhaite un parser qui construise un vrai graphe, en reconstruisant les refs croisées
  - Le type des nœuds du graphe = classes du MM

Xtext : Architecture globale

- Xtext une solution récente mais qui permet de profiter de toute l'artillerie EMF à faible coût
- A la base : une grammaire MonDSL.xtext permet de générer
  - Un métamodèle (=> via EMF les bénéfices habituels)
  - Un parser(T2M)/sérialiseur(M2T)
  - Un éditeur riche intégré dans eclipse
  - Dés facilités pour customiser les feature les plus agréables de l'éditeur de code d'eclipse, auto-completion, corrections au cours de la frappe, template de code, quickfix…
  - Des facilités pour construire un interprète pour le DSL

Xtext : la Grammaire

- Comme tout langage de grammaire, on distingue lexème et règle grammaticale
- Règle lexicale : reconnait une chaine de caractères de longueur arbitraire, spécifiée avec des regex standards
  - On dispose par défaut de terminaux usuels : ID (identifiant C), INT (un entier), STRING (avec des quotes, un texte arbitraire), SL_COMMENT, ML_COMMENT (commentaires // et /* */)…
- Règle grammaticale : définit une méta-classe
  - Les éléments de la règle donnent des attributs dans la classe

Un exemple

- Grammaire : règle Person

Person : ‘person’ name=ID ‘age’ age=INT ‘address’ address=STRING ;

- Syntaxe concrète

person Bob age 32 address « Jamaica Bay »

- Métamodèle

Classe Person, attributs name:EString, age:EInt, address:String

- Le type des attributs peut être le même pour des règles lexicales différentes (ici STRING et ID)

Attributs multiplicité *

- Un élément important de la méta-modélisation

Agenda : ‘agenda’ name=ID ‘{‘ (persons+=Person ‘;’)* ‘}’

- Exemple

agenda Friends {

person Bob age 32 address « Jamaica Bay » ;

person Boule age 8 address « Verneuil » ;

}

- Méta-modèle

Classe Agenda, attribut multiplicité *, contenance par composition persons:Person[*]

- On mixe BNF standard et règles de définition du MM

Références

- Deuxième élément fondamental des méta-modèles : les modèles sont des graphes !
- Règles EMF nécessitent d'appréhender composition vs. aggregation
- Pour avoir un format de stockage qui ait du sens, tout élément est navigable depuis la racine via un **un et un seul** lien de composition
- Au graphe du modèle (arbitraire) on superpose un arbre couvrant qui explique la position physique des objets
- D'où les méthodes EMF (e.g. EcoreUtil.copy) et leur sémantique : sous arbre copié, références internes au sous arbres dupliquées (tout le sous graphe couvert par l'arbre est copié), références vers l'extérieur du sous arbre copiées par aliasing
- Les relations parents/enfants sont stockées (getParent() et getChildren()) et sont mise à jour quand on affecte (setXXX) un attribut noté « composition »

Références Xtext

- Comme une règle habituelle mais le type (règle grammaticale à invoquer) est noté entre crochets

Dog : 'dog' name=ID '(' master=[Person] ')' ;

Elephant : 'elephant' name=ID '('

memory+=[Agenda] (',' memory+=[Agenda])* ')' ;

- Exemple

dog Bill(Boule)

elephant Babar(Friends,Work)

- Métamodèle

Classe Dog, attributs name:String et master:Person (aggregation).

- Le fait d'avoir un attribut « name » est significatif vis-à-vis des règles permettant de reconstruire les références : on peut référer à une Person par son nom par défaut.
- +=[Cible] permet de créer des attributs aggregation multiplicité *
- Par défaut la règle de parse est ID, on peut préciser autre chose, e.g. [Person|STRING] si le nom est sous double quote dans ce contexte, où FQN si on utilise des scope type QualifiedName

Héritage

- Un méta-modèle utilise le plus souvent une bonne dose d'héritage
  - toute structure arborescente donne un DP Composite dans le MM
  - Pensez compilation, un statement est une déclaration ou une affectation ou un if, muni de statement iftrue/iffalse, ou une séquence de statement (bloc) ou…
- Une règle Xtext = une méta-classe
- La notation | permet d'exprimer l'héritage

Animal : Dog | Elephant ;

Zoo : 'zoo' '{' (animals+=Animal ';')* '}'

- Métamodèle : classe Animal, attribut name:String, Dog et Elephant extends à présent Animal et perdent leur name
- On n'est pas obligé d'invoquer la règle pour affecter le MM !

Héritage (suite)

- On peut avoir différentes règles grammaticales qui produisent des occurrences de la même métaclasse, i.e. d'annoncer qu'une règle rend une métaclasse différente du nom de la règle
- On peut avoir besoin de forcer un typage particulier

Chenil : 'dogs' '{' (dog+=FastDog) '}';

FastDog returns Dog : name=ID'-'master=[Person];

- Pas de métaclasse produite ici
- Les propriétés non affectées prennent leur valeur par défaut

Priorités et parse complexe

- Un problème des grammaires : gérer la priorité des opérateurs : 3+2*5 = 3+(2*5) et pas (3+2)*5
- Solution classique : ordonner par priorités croissante les règles, mais attention au typage

```
Addition returns Expression:
        Multiplication ({BinaryIntExpression.left=current} op=('+'
| '-') right=Multiplication)*;

Multiplication returns Expression:
        UnaryExpr ({BinaryIntExpression.left=current} op=('/' | '*'
) right=UnaryExpr)*;
UnaryExpr returns Expression : UnaryMinus | Primary;
UnaryMinus returns Expression:
   {UnaryMinus} '-' expr=UnaryExpr;
```

Priorités et parse complexe

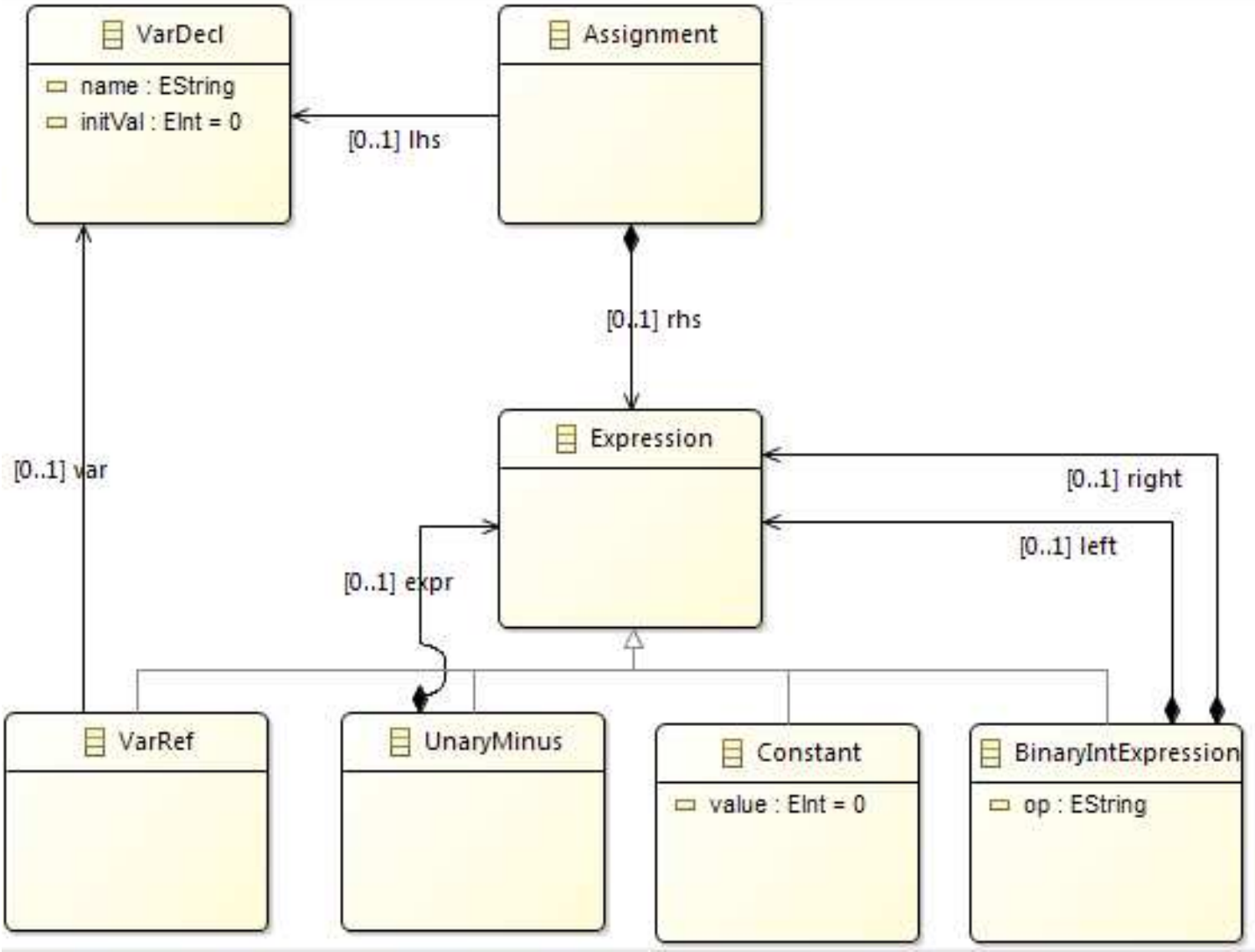
- Les cas terminaux

```
Primary returns Expression:
        VarRef |
        Constant |
        (=> ('(' Addition ')' ));
Constant : value=INT;
VarRef : var=[VarDecl];
```

- Un contexte

```
VarDecl : 'int' name=ID '=' initVal=INT ';';
Assignment : lhs=[VarDecl] '=' rhs=Addition ';' ;
Model : (vars+=VarDecl  |  actions+=Assignment)*;
```

Meta-Modele Calc



WorkFlow

- Muni de la grammaire, on exécute le workflow MWE2
- MWE2 : framework de model weaving, automatise une choréographie de services au modèle
- Xtext génère :
  - Un MM au format .ecore
  - Une grammaire ANTLR 3.2 .g , qui s'appuie sur l'API d'édition et de factory d'EMF pour passer de l'AST au modèle
  - Un sérialiseur, pour aller du modèle au texte
  - Un linker, pour résoudre les références croisées
  - Un éditeur incluant
    - Validations
    - Completions de code
    - Outline…

Configuration via DI

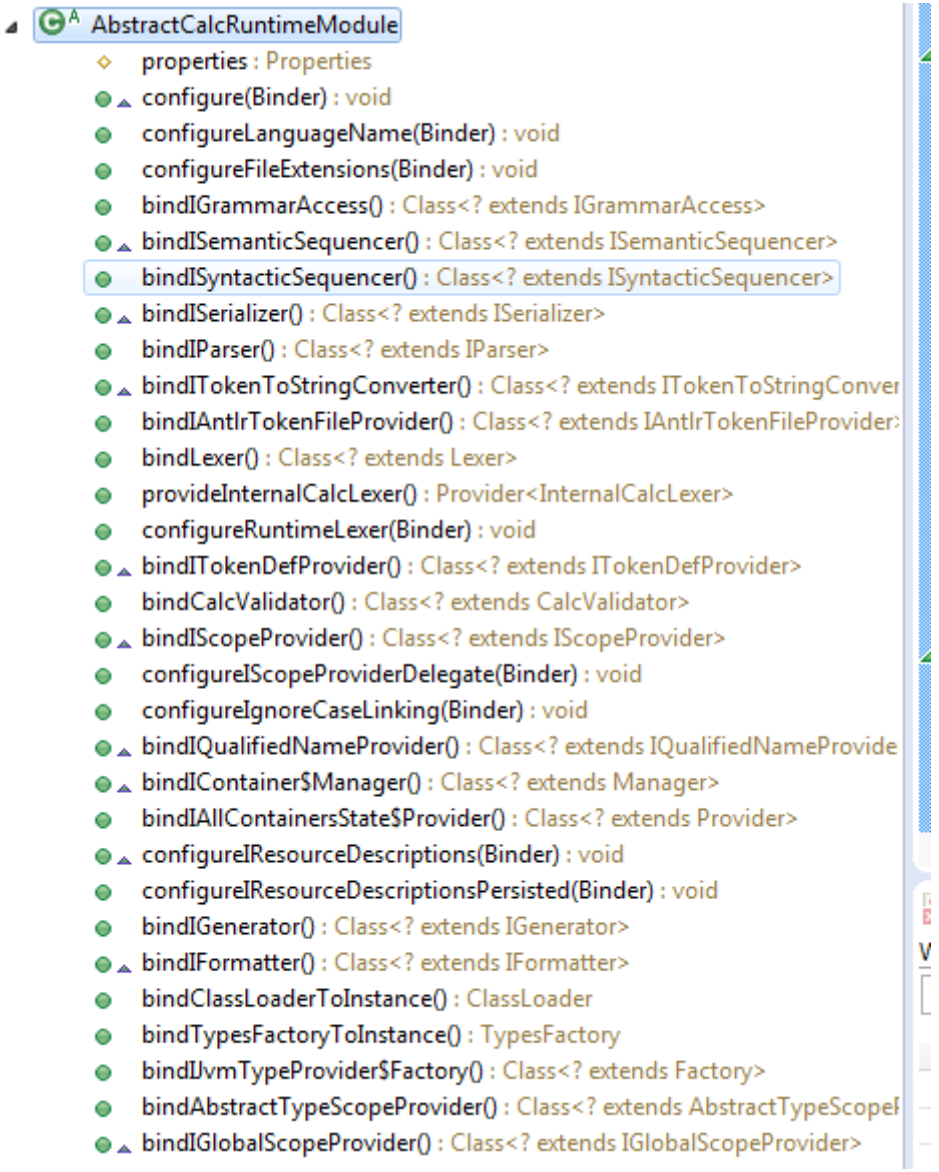
- L'injection de dépendances est une technique puissante
  - Au lieu de : IRequis requis = new ConcreteRequis(), la classe veut que la réalisation de IRequis soit positionnable de l'extérieur (cf. aussi pattern Strategy)
  - Permet d'aller vers des codes abstraits où l'on a toute la souplesse des dépendances fonctionnelles et pas structurelles
  - Evite d'avoir à développer et configurer des Factory soi même
- Lien sur AOP (Aspect Oriented)
  - En AOP, on veut séparer les problèmes pour s'en préoccuper séparément (e.g. couche sécurité…)
  - Ici, la configuration de l'utilisateur (dont son métamodèle, et les éléments produits à partir de la grammaire par Xtext) est injectée dans la config « de base » d'Xtext.

Google Guice

- Framework pour l'injection de dépendances
- Les classes déclarent leurs dépendances via des annotations
  - @Inject IRequis requis ;
- Les **Modules** Guice definissent la façon de résoudre les dépendances
- Le module définit un ensemble de bindings entre des types (abstraits, typiquement des interfaces) et des réalisations concrètes
- Le code client ne voit plus que les interfaces dont il dépend !

Guice dans Xtext

- XText utilise énormément Guice
  - Rend la prise en main moins aisée
  - Niveau de configurabilité excellent
- Pour « faciliter » la prise en main, depuis 2.5 Xtext promeut aggressivement Xtend, YADSL !
- Heureusement le sentier est bien balisé pour les use case habituels



Configuration du scope

- Le Scope définit quels objets sont des cibles raisonnables pour une référence dans un certain contexte
- Deux Scope par défaut disponibles

Source : XtextCon 2014 : Scoping, Linking and Indexing,
Moritz Eysholdt, itemis AG
Dispo en ligne :
http://xtextcon.org/slides/Scoping,%20Linking%20and%20Indexing%20-%20Moritz%20Eysholdt.pdf

```
fragment = scoping.ImportNamespacesScopingFragment {}
fragment = exporting.QualifiedNamesFragment {}
```

- references use (semi) qualified names
- import statements for qualified names
- uses index
- honors project dependencies

```
fragment = scoping.ImportURIScopingFragment {}
fragment = exporting.SimpleNamesFragment {}
```

- references use simple names
- import statements for file names (URIs)
- does not need index
- project dependencies are ignored

Configuration fine du Scope

- On peut définir un Scope à soi en se bindant au point d'extension ScopeProvider il faut fournir une méthode : IScope scope (EObject context, EReference reference)
- Le contexte, c'est l'objet à qui appartient la référence, e.g. une occurrence de Dog comme Bill
- La réference, c'est ce qu'on cherche à satisfaire, ici l'attribut maitre typé Person de Dog
- On peut naviguer depuis le contexte (getParent()…) pour accéder aux éléments cible potentiels (ici l'ensemble des Personnes de tous les agendas)
- Via Xtend on édite MyDSLScopeProvider.xtend

Validations

- XText respecte (assez bien) le point d'extension EMF Validation : les validations fournies par des tiers lèvent bien des marqueurs d'erreur dans l'éditeur
- Les validations écrites dans le framework sont mieux intégrées. Typiquement, on écrit en Java sur l'API EMF (ou plutôt en XTend sur l'API de votre DSL produite par EMF).
- En pratique, éditer MyDSLValidator.xtend
- Chaque erreur, un peu comme dans EMF Validation, doit correctement incriminer un élément du modèle

Validation : exemples

- Niveau mwe2:

```
 fragment = validation.ValidatorFragment auto-inject {
composedCheck = "org.eclipse.xtext.validation.ImportUriValidator"
composedCheck = "org.eclipse.xtext.validation.NamesAreUniqueValidator" }
```

- Niveau Xtend

```
@Check
def checkGreetingStartsWithCapital(Greeting greeting) {
if (!Character.isUpperCase(greeting.name.charAt(0))) {
warning('Name should start with a capital',
MyDslPackage.Literals.GREETING__NAME,
INVALID_NAME)
}}
```

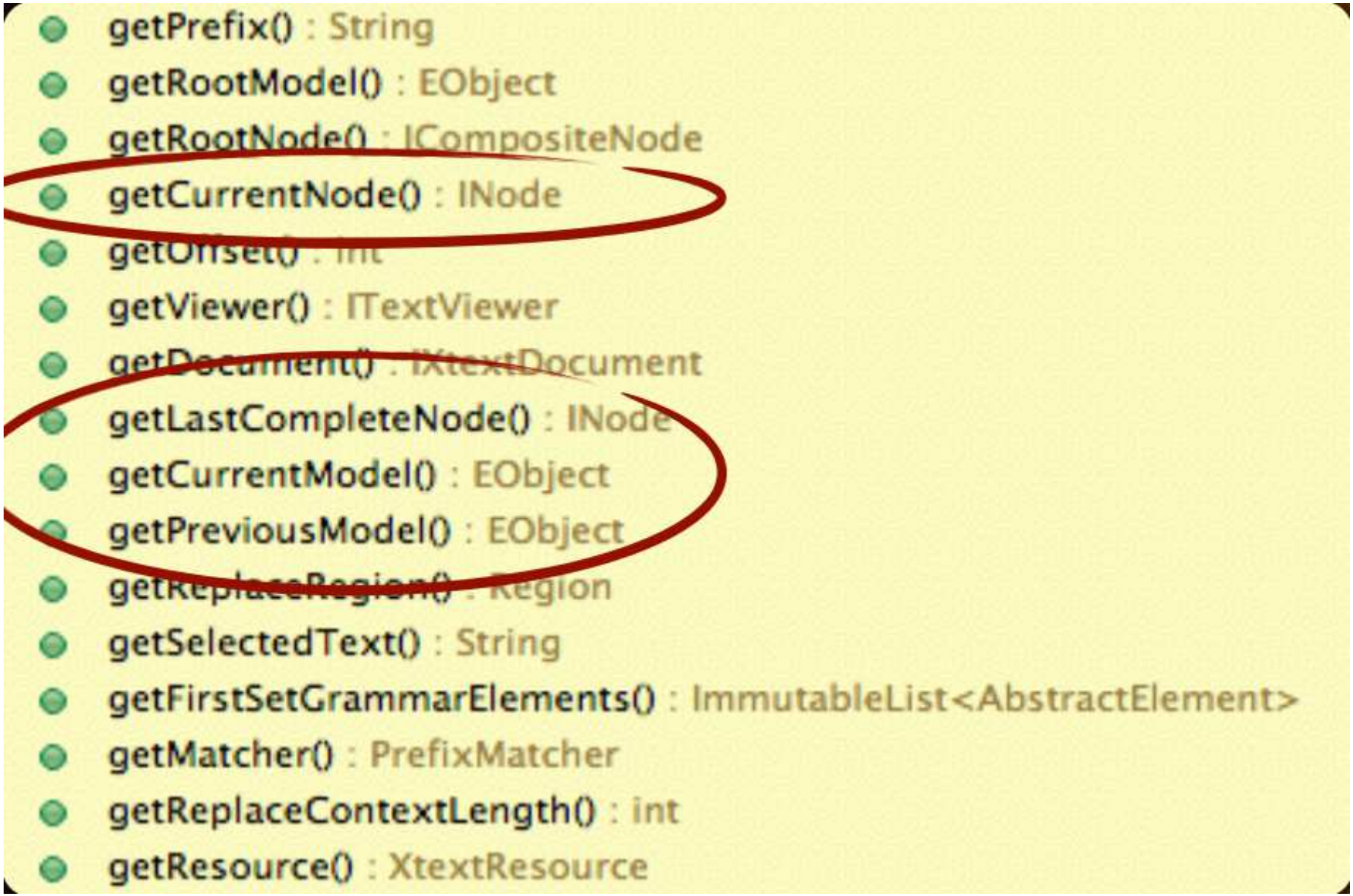
- L'objet incriminé est greeting, sa propriété en faute est GREETING__NAME
- INVALID_NAME donne un code d'erreur, qui sera utilisé pour définir des quickfix

QuickFix

- A spécialiser dans le projet .ui
- Propose des solutions (refactorings)
- Exemple

```
@Fix(MyDslValidator::INVALID_NAME)
def capitalizeName(Issue issue, IssueResolutionAcceptor acceptor) {
acceptor.accept(issue, 'Capitalize name', 'Capitalize the name.', 'upcase.png') [
context |
val xtextDocument = context.xtextDocument
val firstLetter = xtextDocument.get(issue.offset, 1)
xtextDocument.replace(issue.offset, 1, firstLetter.toUpperCase)
]
}
```

Quickfix : Context



Autres éléments customisables (facilement)

- Outline view
- Template proposals
- Formatter

XBase

- Une grammaire pour Xtext qui est compatible Java
- Permet rapidement d'ecrire des grammaires qui reconnaissent des expressions, des fonctions etc…
- Donne accès au sein du DSL aux fonctions de Java
- Support pour écrire un générateur/interprète du DSL, un debugger … Pratique pour rapidement avoir un genre de langage de script EXECUTABLE + IDE eclipse !
- MAIS :
  - Perte de contrôle sur le MM et partiellement sur la syntaxe du DSL
  - Dépendance énorme à XBase (encore instable)
  - Grosse usine à gaz ? On restera dans l'écosystème Xtext avec ces modèles.

Bilan XText

- Qualités :
  - La grammaire + MM « deux en un » résoud un grand nombre de problèmes de maintenabilité des MM
  - Résultat visuellement très convaincant
  - Prise en main aisée, on est pas sur GMF… Facilite l'entrée sur EMF (édition ecore à la main?)
- Défauts :
  - Documentation lacunaire, voire inexistante (javadoc les mecs, au moins les API publiques !). Particulièrement crucial pour les customisations fines. Présentations XTextCon = meilleure source de doc ?
  - Nécessite une bonne compréhension du sous jacent pour debugger : il n'est pas rare de devoir debugger la grammaire Xtext en inspectant la grammaire ANTLR produite ou de devoir comprendre les bindings Guice
  - XTend, YADSL ! Mais on peut se passer d'Xtend, écrire en java (remplacer le fichier xtend par un fichier Java) ça marchera OK.
  - Versions en évolution rapide, API un peu instable, mais bugs et ressources consommées évoluent dans la bonne direction