

DevRep : Cours 3

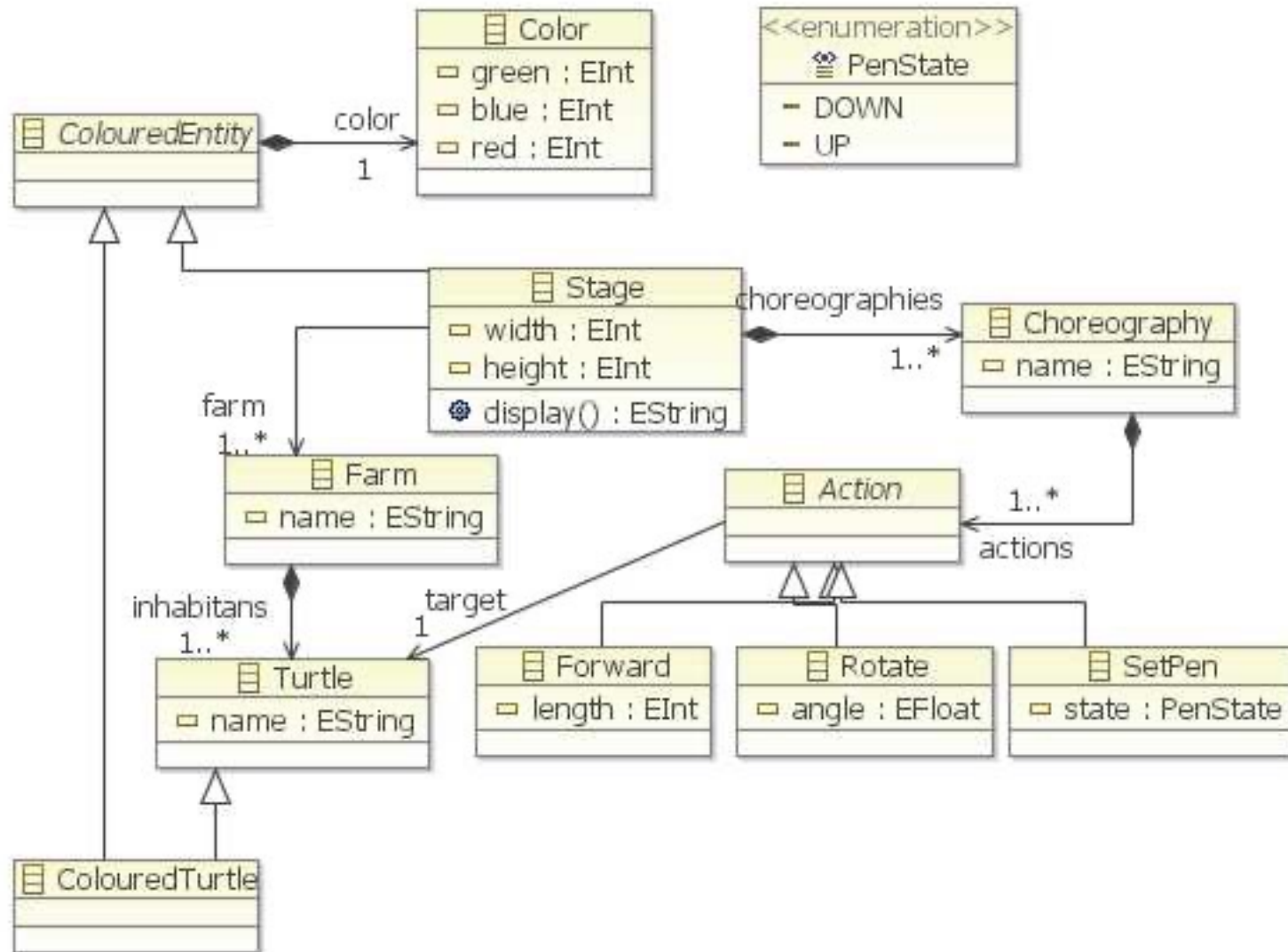
Dans ce cours

- DS(M)L
 - Syntaxe abstraite (Méta-modélisation - MOF/EMF)
 - Syntaxe concrète : textuelle. (Xtex)
 - Sémantique : génération de code
- Projet : DSML pour la robotique

TP1 & TP2

- Turtle
 - Méta-Modèle (.ecore)
 - Création de modèles instance

Méta-Modèle



EMF -Java

- Génération de l'éditeur arborescent
- Génération de l'API Java/EMF
- Manipulation de modèles en Java avec le code généré

```

import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;

import fr.lip6.turtle.TurtlePackage;
import fr.lip6.turtle.impl.TurtlePackageImpl;

/**
 * Utility class to save and load model instances of Turtle from XMI files.
 * @author lmh
 */
public final class ModelIO {

    private ModelIO() {}

    /**
     * Loads a model stored in a XMI file.
     * @param file the path to the incoming XMI file
     * @return the root object of the loaded model
     */
    public static final EObject loadModel(Path file) {
        Resource.Factory.Registry reg =
Resource.Factory.Registry.INSTANCE;
        Map<String, Object> m = reg.getExtensionToFactoryMap();
        m.put("", new XMIResourceFactoryImpl());

        // Obtain a new resource set
        ResourceSet resSet = new ResourceSetImpl();
        TurtlePackage tp = TurtlePackageImpl.init();
        EPackage.Registry ereg = resSet.getPackageRegistry();
        ereg.put(tp.getNsURI(), tp);

        Resource resource =
resSet.getResource(URI.createURI(file.toString()), true);
        return resource.getContents().get(0);
    }

    /**
     * Saves a model in a XMI file.
     * @param mo the model to save
     * @param file the path to the destination file.
     */
    public static final void saveModel(EObject mo, Path file) {
        Resource.Factory.Registry reg =
Resource.Factory.Registry.INSTANCE;
        Map<String, Object> m = reg.getExtensionToFactoryMap();
        m.put("", new XMIResourceFactoryImpl());

        // Obtain a new resource set
        ResourceSet resSet = new ResourceSetImpl();
        Resource resource =
resSet.createResource(URI.createURI(file.toString()));

        // Get the first model element and cast it to the right type,
        resource.getContents().add(mo);
    }
}

```

Syntaxe concrète avec XText

The image displays an IDE interface with several features highlighted by labels and arrows:

- Custom Editor:** Points to the editor window showing the code for `*ex2.imp`.
- Syntax Highlighting:** Points to the code in the `*ex2.imp` editor, where keywords like `import`, `data`, and `String` are highlighted in different colors.
- Code Folding:** Points to the foldable regions in the code editor, indicated by minus signs in the left margin.
- Importing other files:** Points to the `import ext="exp"` statement in the `*example.imp` editor.
- Customized Outline View:** Points to the `Outline` view on the right, which shows a tree structure of the code elements, including `records`, `record P001`, `record A001`, `record V001`, `record T001`, and `record S001`.
- Custom Constraints:** Points to the `Problems` view at the bottom, which shows two errors: "Couldn't resolve reference to 'firstNameXXX'" and "field not defined on type Patient".
- Code Completion:** Points to the yellow pop-up menu in the `*example.imp` editor, which suggests completions for the `p.firstNameXXX` expression, including `a: Address` and `p: Patient`.

```
import ext="exp" {

  data {

    data Patient {
      name: String
      firstName: String
      birthDate: String
      insuranceName: String
      insuranceNumber: String
      -> adr: Address
      -> tel: Telephone[]
      -> stays: Stay[]
    }

    data Address {
      street: String
      zip: String
      city: String
    }

    data Telephone {
      countryCode: String
      cityCode: String
      localNumber: String
    }
  }
}
```

```
import ext="exp" {

  ref "platform:/resource/exampleDa

  records {
    record P001 {

      p: Patient
      1 -> p.name
      2 -> p.firstNameXXX
      3 -> p.birthDate

    }

    record A001 {
      a: Address
      1 - a: Address
      2 - p: Patient
      3 - p.a
    }

    recor
    1 -
    2 -

  }

  reco
}
```

Description	Resource	Path	Locatic
Couldn't resolve reference to 'firstNameXXX'	example.imp	exampleData/src	line : 1
field not defined on type Patient	example.imp	exampleData/src	line : 1

15 Minutes Tutorial

https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html

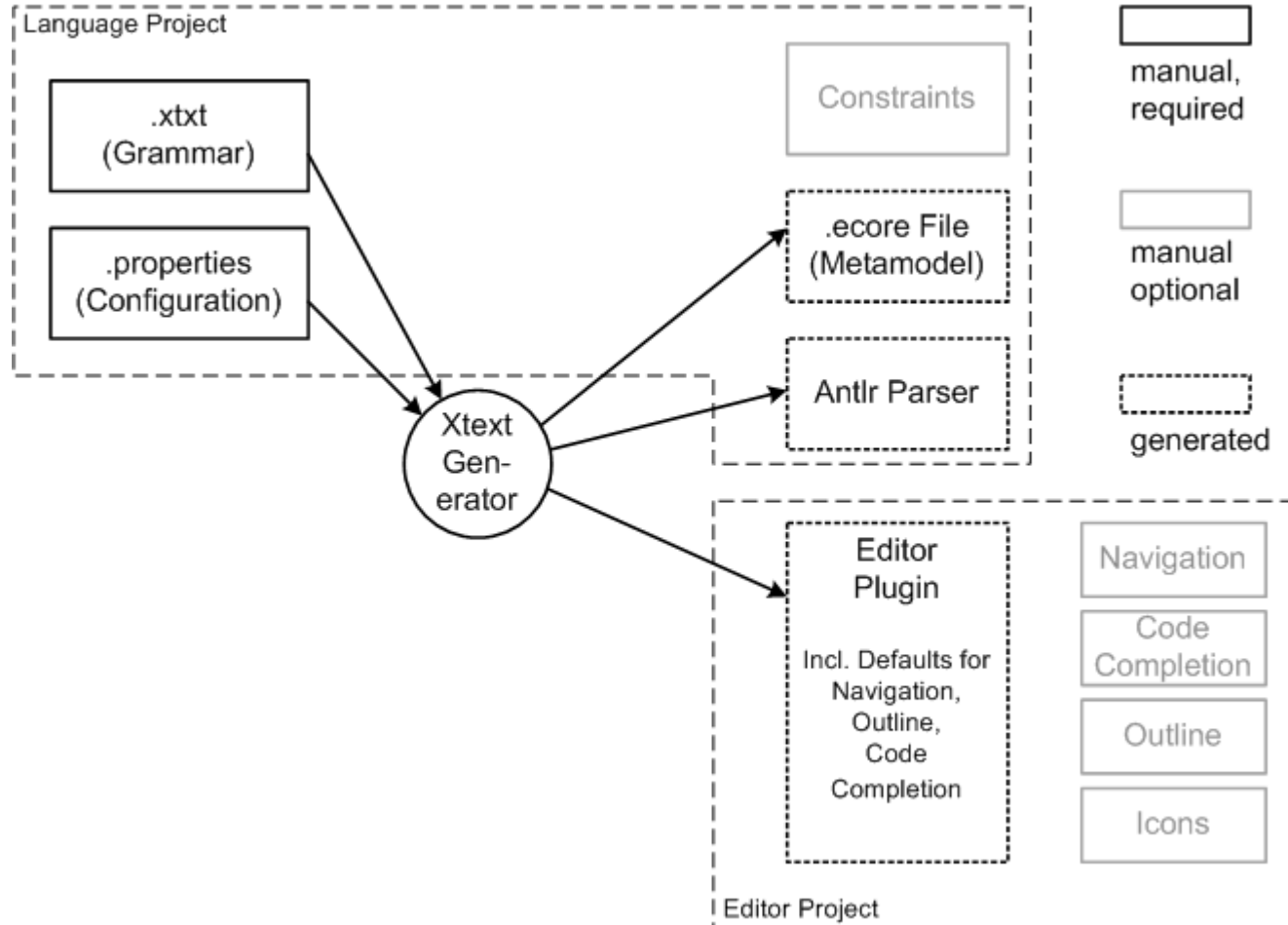
15 Minutes Tutorial

In this tutorial we will implement a small domain-specific language to model entities and properties similar to what you may know from Rails, Grails or Spring Roo. The syntax is very suggestive:

```
1. datatype String
2.
3. entity Blog {
4.     title: String
5.     many posts: Post
6. }
7.
8. entity HasAuthor {
9.     author: String
10. }
11.
12. entity Post extends HasAuthor {
13.     title: String
14.     content: String
15.     many comments: Comment
16. }
17.
18. entity Comment extends HasAuthor {
19.     content: String
20. }
```

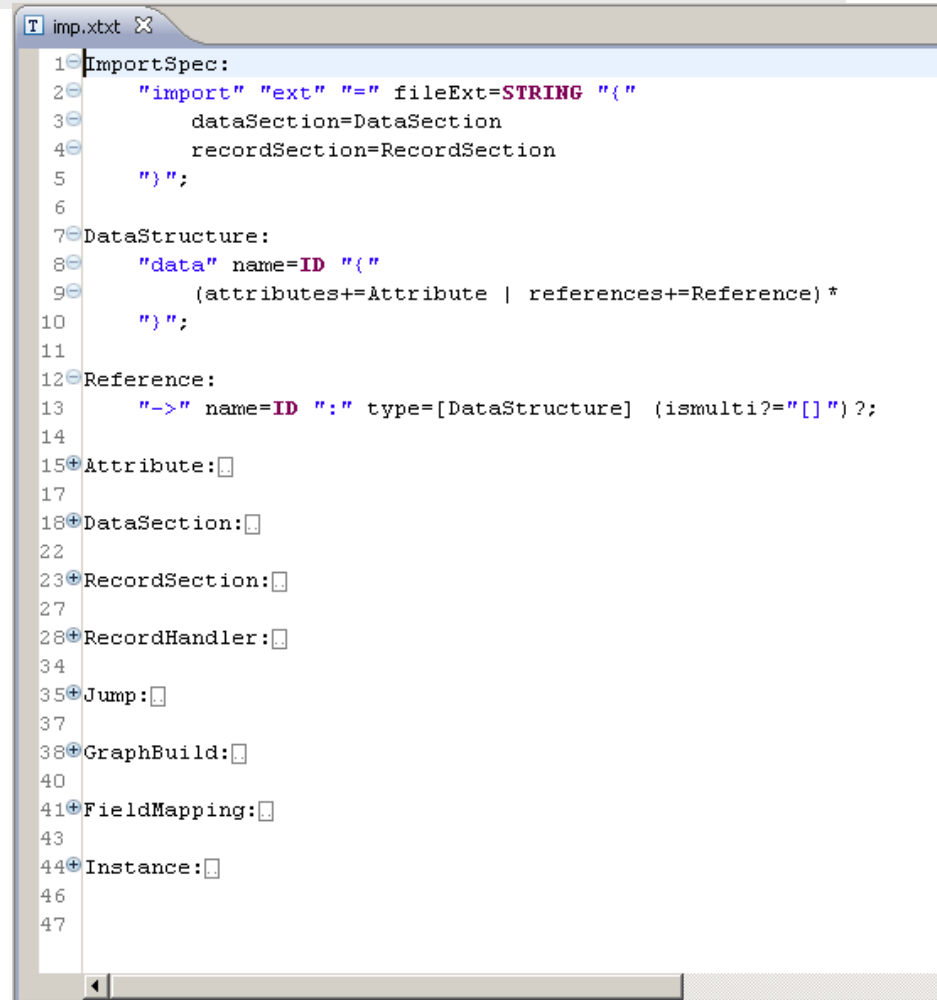
After you have installed Xtext on your machine, start Eclipse and set up a fresh workspace.

XText



Xtext : Grammaire

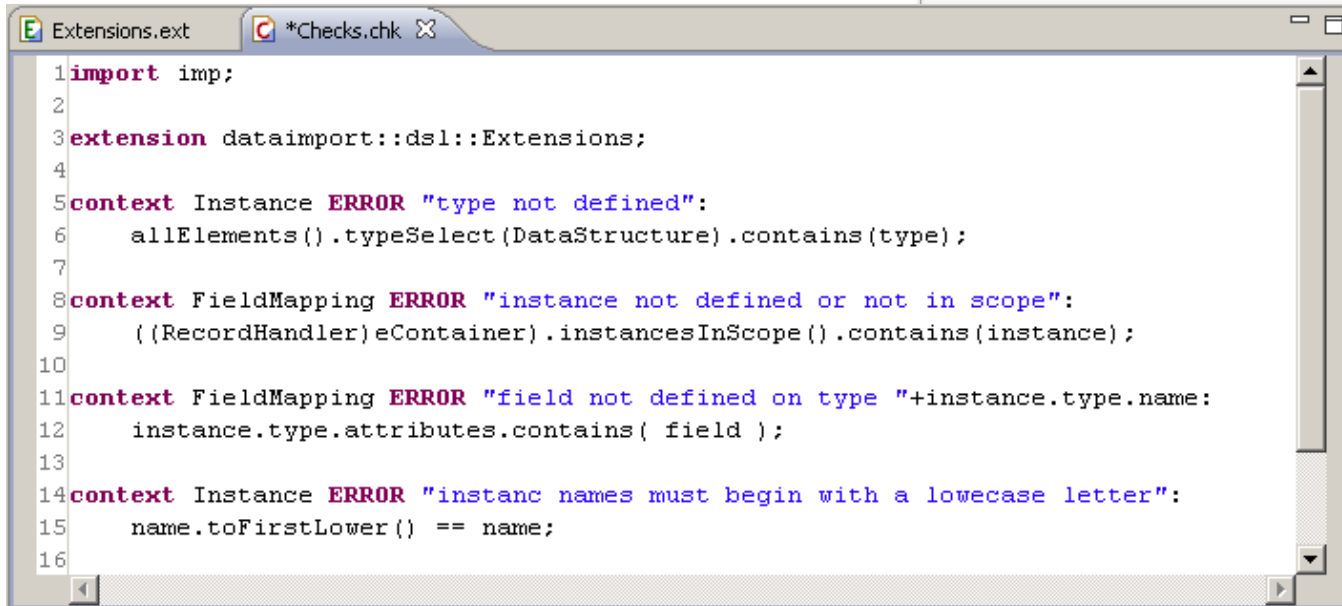
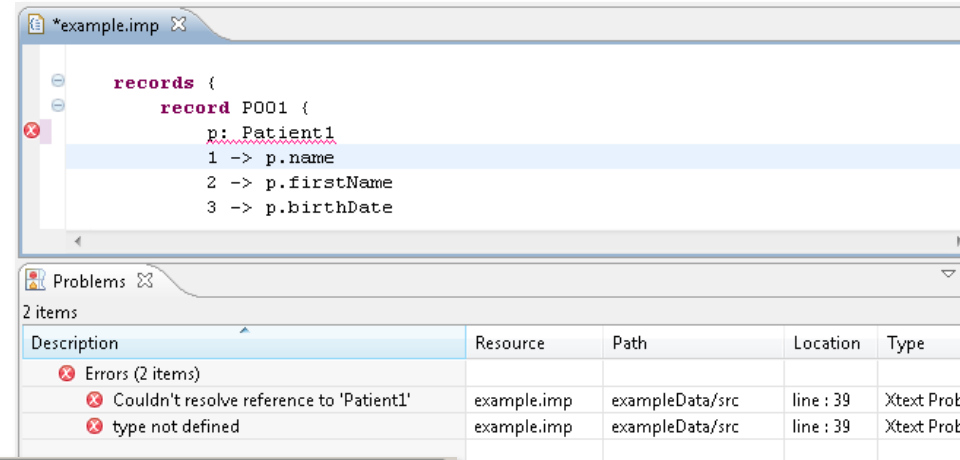
- Format EBNF-like dans Xtext editor.
- Xtext generator construit un EMF meta model à partir de la grammaire.



```
1 ImportSpec:
2   "import" "ext" "=" fileExt=STRING "{" "
3     dataSection=DataSection
4     recordSection=RecordSection
5   "}";
6
7 DataStructure:
8   "data" name=ID "{" "
9     (attributes+=Attribute | references+=Reference) *
10   "}";
11
12 Reference:
13   "->" name=ID ":" type=[DataStructure] (ismulti?="[]" )?;
14
15 Attribute:[]
16
17
18 DataSection:[]
19
20
21
22
23 RecordSection:[]
24
25
26
27
28 RecordHandler:[]
29
30
31
32
33
34
35 Jump:[]
36
37
38 GraphBuild:[]
39
40
41 FieldMapping:[]
42
43
44 Instance:[]
45
46
47
```

Xtext : Validation

- oAW Check language, basé sur Xtend
 - Warnings
 - Errors



M2T: Model To Text (Code génération)

Principe de base

- La génération de code consiste à parcourir un modèle afin de générer du texte
 - D' un graphe d' objets vers une séquence de caractères
- Un générateur est un programme objet qui visite les objets représentant le modèle et écrit du texte
- Un standard OMG: MOF to Text

Exemple UseCase vers Texte

```
Systeme s = loadXMI(« monModel.xmi »);  
print(« le système a pour nom »+s.name);  
for (int i=0 ; i < s.cas.size() ; i++) {  
    print(« il possède un cas intitulé »+cas[i].intitule);  
}
```

Le système a pour nom PetStore
Il possède un cas intitulé Commander Panier
Il possède un cas intitulé Valider Article

L' approche Template

- Les programmes « générateur de code » ont pour inconvénient de rendre peu lisible le texte généré
- Cet argument est important pour les phases de développement et de maintenance
- L' approche par **Template** permet au développeur de préciser le texte généré en définissant des **Variables**
- Ces variables seront affectées par le modèle

Exemple avec JET

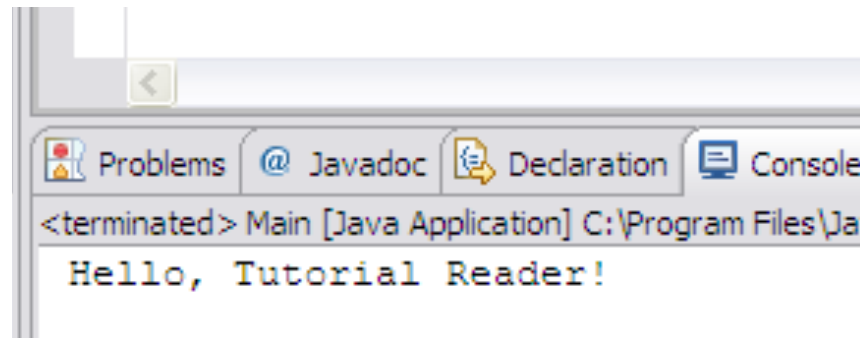
Fichier contenant le template (greetings.txtjet)

```
<%@ jet package="hello" class="GreetingTemplate" %>
  Hello, <%=argument%>!
```

Client Java (main):

```
GreetingTemplate sayHello = new GreetingTemplate();
String result2 = sayHello.generate("Tutorial Reader");
System.out.println(result2);
```

Résultat sur la console:



Maintenance du code

- L'approche par template semble beaucoup plus intéressante pour la maintenance du code
- Pour autant, les langages template restent propriétaires
 - Pour 1 développeur template combien de développeurs Java ?
- Plus une génération est complexe, plus il faut gérer un ensemble de templates ou un ensemble de classes
- Les tests et le debugage sont plus outillés dans les langage de programmation classiques (Java)
 - Dans les langages à Templates, on retrouve l'erreur en regardant le code généré et non pas celui du template

L'approche par marqueur

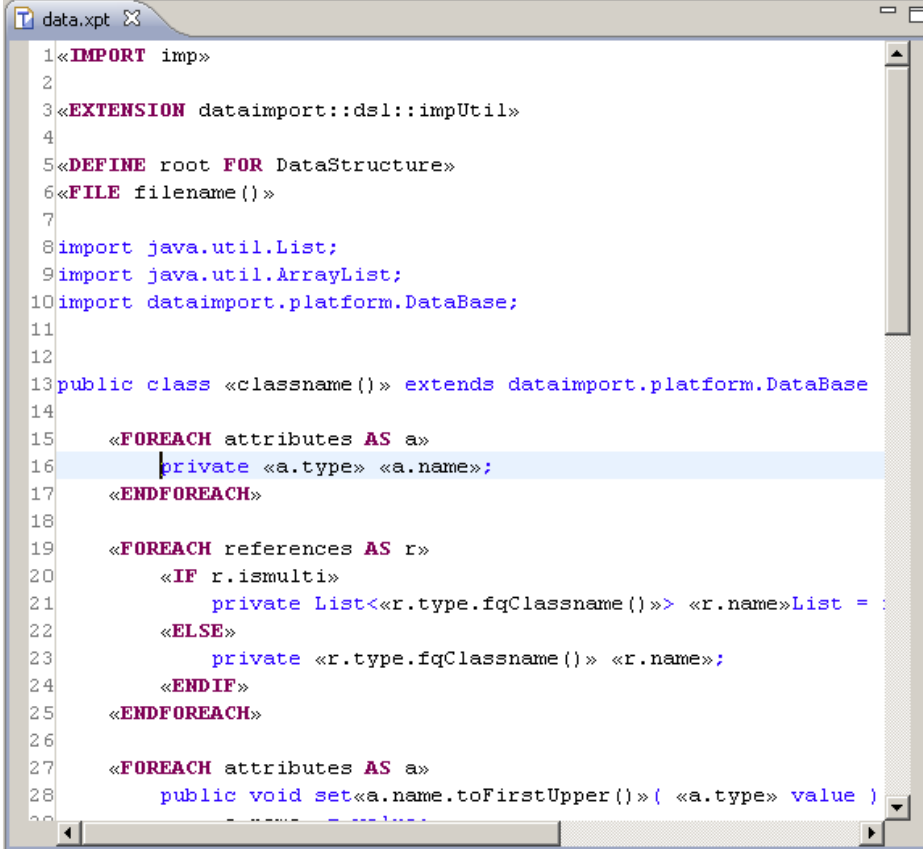
- Le générateur de code écrit des marqueurs dans le texte
- Ces marqueurs délimitent des zones dans lesquels le texte peut être changé
- Ce texte ne sera pas modifié lors de la re-génération
- Cette approche assure une synchronisation du modèle vers le texte
 - "**generated**" dans EMF
 - Identifiants dans Objectteering (projet round-trip)

Les générateurs du moment

- JET (Java Emitter Templates)
 - Pionnier dans les projets Eclipse
 - Utilisé pour la génération de code dans EMF
 - Facile à utiliser, JSP-like
 - JET 2 en cours de développement
- Xpand/Xtend/Workflow
 - Nouveau projet Eclipse
 - Très prometteur
 - Un peu plus complexe mais permet d'exprimer des générations très compliquées
- MTL
 - Implémentation du standard OMG: MOF to Text
 - Outilleur: Acceleo

Xtext : Génération de code

- **Xpand** template language is a powerful and well established code generation facility with nice tooling.
- You can easily **traverse the model/meta model** using the **Xtend** language (à la OCL)



```
1«IMPORT imp»
2
3«EXTENSION dataimport::dsl::impUtil»
4
5«DEFINE root FOR DataStructure»
6«FILE filename()»
7
8import java.util.List;
9import java.util.ArrayList;
10import dataimport.platform.DataBase;
11
12
13public class «classname()» extends dataimport.platform.DataBase
14
15    «FOREACH attributes AS a»
16        private «a.type» «a.name»;
17    «ENDFOREACH»
18
19    «FOREACH references AS r»
20        «IF r.ismulti»
21            private List<«r.type.fqClassname()»> «r.name»List = ;
22        «ELSE»
23            private «r.type.fqClassname()» «r.name»;
24        «ENDIF»
25    «ENDFOREACH»
26
27    «FOREACH attributes AS a»
28        public void set«a.name.toFirstUpper()»( «a.type» value )
29            «a.name» = value;
```

M2M: Model To Model

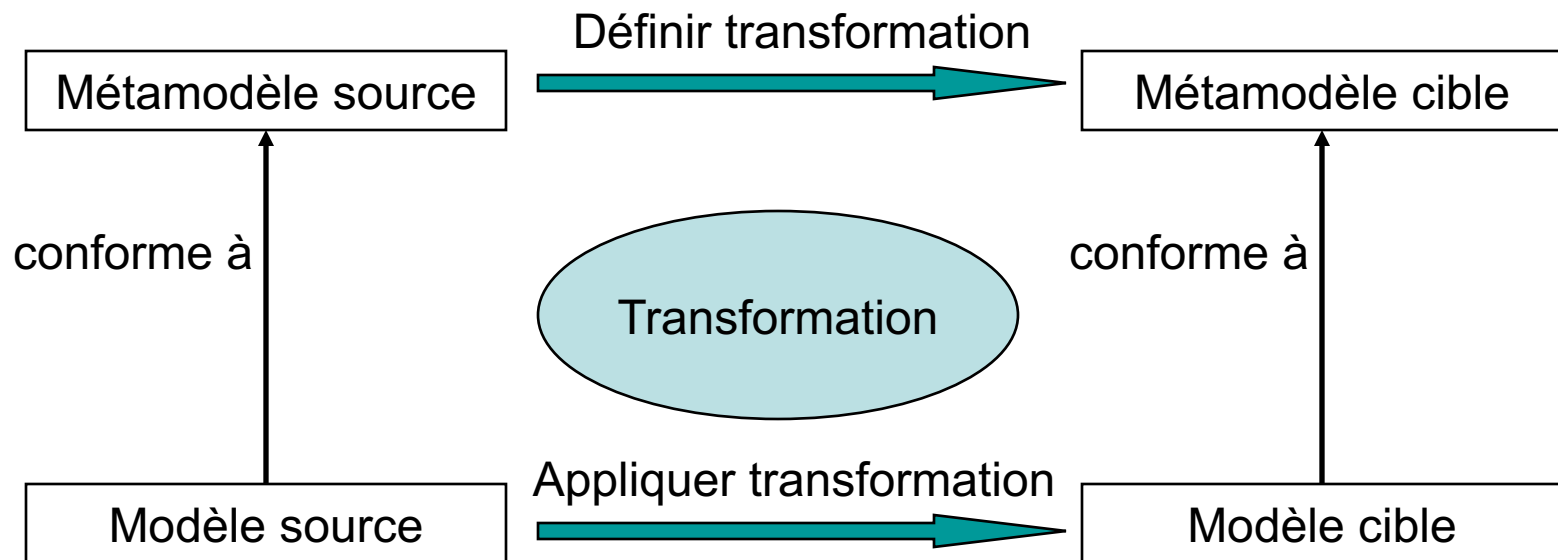
Transformations de Modèles

- Une transformation est une opération qui
 - Prend un (ou plusieurs) modèle source en entrée
 - Fournit un (ou plusieurs) modèle cible en sortie
- Transformation endogène
 - Les modèles source et cible sont conformes au même méta-modèle
 - Exemple : Transformation d'un modèle UML en un autre modèle UML
- Transformation exogène
 - Les modèles source et cible sont conformes à des méta-modèles différents
 - Exemples : Transformation d'un modèle UML en schéma de BDD

Principe de base

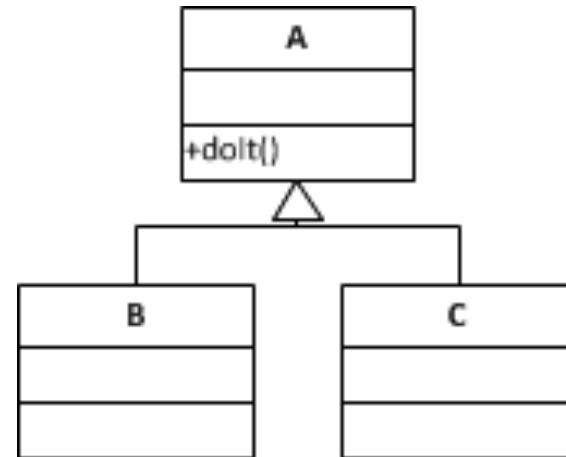
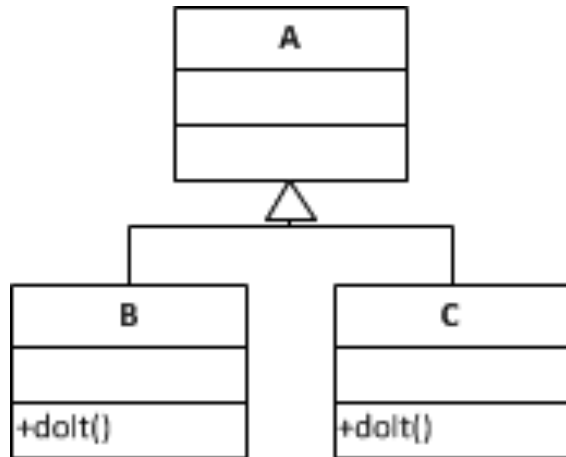
- La transformation de modèle consiste à parcourir un modèle pour le modifier ou pour générer un autre modèle
 - D' un graphe d' objets vers un autre graphe d' objets
- Un transformateur est un programme objet qui visite les objets représentant le modèle et construit de nouveaux objets représentant un autre modèles
- Les modèles source et cible doivent correspondre à leur Méta-modèles respectifs (peut être le même)

Architecture des transformations



Transfo. Endogène: Exemple

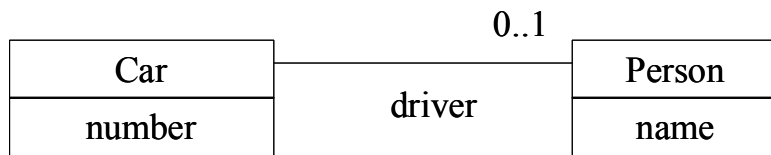
- Si deux classes héritent d'une même classe et qu'elles ont une opération qui a le même nom, il faut remonter l'opération dans la classe mère.



Transfo. Endogène: Exemple

- Transformation d'un diagramme de classes UML vers un schéma de base de données relationnelle

UML



RDBMS

Table PERSON	
PERSONNE_ID(pk)	NAME
1	Dupond
2	Tintin

Table Car	
NUMBER (pk)	PERSONNE_ID (fk)
234 GH 62	2
1526 ADF 77	1

Transformations : types de modèles

- 3 grandes familles de modèles et outils associés
 - Données sous forme de séquence
 - Ex : fichiers textes (AWK)
 - Données sous forme d'arbre
 - Ex : XML (XSLT)
 - Données sous forme de graphe
 - Ex : diagrammes UML
 - Outils
 - Transformateurs de graphes déjà existants ex. AGG
 - Nouveaux outils du MDE et des AGL (QVT, J, ATL, Kermeta ...)

Techniques de transformation

- **Principe : découpage des transformations en règles**
 - Une règle définit la manière dont un ensemble de concepts du méta-modèle source est transformé (**mappé**) en un ensemble de concepts du méta-modèle destination.
- **Approche déclarative : focalisation sur ce qui est créé**
 - Fonctionnement :
 - Recherche de certains patrons (d'éléments et de leurs relations) dans le modèle source
 - Chaque patron trouvé est remplacé dans le modèle cible par une nouvelle structure d'élément
 - Le moteur de règle choisi la façon dont sont exécutées les règles (ordre)
 - Propriété :
 - + Ecriture de la transformation « assez » simple
 - mais ne permet pas toujours d'exprimer toutes les transformations facilement
 - **Pas de maîtrise sur la navigation (le moteur)**

Techniques de transformation

- Approche impérative : focalisation sur comment la règle est exécutée
 - Proche des langages de programmation
 - On parcourt le modèle source dans un certain ordre (explicite) et on génère le modèle cible lors de ce parcours
 - + Ecriture de la transformation plus complexe mais permet de toutes les définir
 - Savoir naviguer dans le modèle!

Techniques de transformation

- **Approche hybride : à la fois déclarative et impérative**
 - Celle qui est utilisée en pratique dans la plupart des outils
 - Approche principalement à base de règles
 - Des « helper » sont programmés afin de faciliter la navigation ou la construction de certaines parties du modèle
 - Ces « helper » peuvent être appelés à partir des règles (en navigation ou en construction)

Exemple en impératif (1/2)

Naviguer dans le modèle (exemple transfo Endogène)

1. Héritage puis même opération

1. Parcourir toutes les classes du package
2. Isoler celles qui ont deux classes filles
3. Vérifier que les deux classes filles ont une opération de même nom

• Même opération puis héritage

1. Parcourir 2 à 2 toutes les classes du package
2. Vérifier qu'elles ont une opération de même nom
3. Vérifier qu'elles ont une même classe mère

Même complexité ?

Modèle en Java d'un diag. d'états : Pattern Visiteur

```
public class StateMachine {
    protected Vector states = new Vector();
    protected Vector transitions = new Vector();
    public void addState(State s) {
        states.add(s); }
    public void addTransition(Transition t) {
        transitions.add(t); }
    ... }

public class Transition {
    protected State from, to;
    protected String event;
    public Transition(State from, State to, String evt) {
        this.from = from;
        this.to = to;
        event = evt; }
}

public class State {
    protected String name;
    public State(String name) {
        this.name = name; }
}
```


Modèle en Java d' un diag. d' états

Définition d'un modèle : instances et liaisons de ces 3 classes

```
...
StateMachine sm;
State s1,s2;
...
sm = new StateMachine();
s1 = new State("Ouvert");
s2 = new State("Ferme");
sm.addState(s1);
sm.addState(s2);
sm.addTransition(new Transition(State.Initial, s1, ""));
sm.addTransition(new Transition(s1, s2, "fermer"));
sm.addTransition(new Transition(s2, s1, "ouvrir"));
```

Modèle en Java d'un diag. d'états : Pattern Visiteur

- Ajout de méthodes pour lire les éléments du modèle
 - Ex. pour **StateMachine** : **getStates()**, **getTransitions()**
- Parcours du modèle via ces méthodes
 - Approche impérative
- Utilisation possible du patron Visiteur
- Génération d'un nouveau modèle à partir de ce parcours
 - En utilisant éventuellement des méthodes des différentes classes pour gérer la transformation des éléments un par un

```
public class State {  
    ...  
    public String toXML() {  
        return new String("<state>\n\t<name>" + name  
            + "</name>\n</state>\n"); } ...
```

Modèle en Java d'un diag. d'états : Pattern Visiteur

- Parcours de type « impératif » du graphe d'objet pour sérialisation en XML

```
String xml = "<statemachine>";
Iterator it = sm.getStates().iterator();
while(it.hasNext())
    xml += ((State)it.next()).toXML();
it = sm.getTransitions().iterator();
while(it.hasNext())
    xml += ((Transition)it.next()).toXML();
xml += "</statemachine>";
System.out.println(xml);
```

Modèle en Java d'un diag. d'états : Pattern Visiteur

- Résultat sérialisation XML

```
<statemachine>
```

```
  <state>
```

```
    <name>Ouvert</name>
```

```
  </state>
```

```
  <state>
```

```
    <name>Ferme</name>
```

```
  </state>
```

```
  <transition>
```

```
    <from>initial</from>
```

```
    <to>Ouvert</to>
```

```
    <event></event>
```

```
  </transition>
```

```
  <transition>
```

```
    <from>Ouvert</from>
```

```
    <to>Ferme</to>
```

```
    <event>fermer</event>
```

```
  </transition>
```

```
  <transition>
```

```
    <from>Ferme</from>
```

```
    <to>Ouvert</to>
```

```
    <event>ouvrir</event>
```

```
  </transition>
```

```
</statemachine>
```