Génération de code et interopérabilité
Première partie

Alain Plantec

7 mars 2016

Table des matières

1 Une version minimale pour démarrer **2**

2 Associations unidirectionnelles entre entités **5**
2.1 Association simple 5
2.2 Association multiple 6
2.3 Association multiple à deux dimensions ou plus 7

3 Héritage et Modèle **8**
3.1 Introduction de l'héritage 8
3.2 Modèle et résolution des dépendances 8

4 Valeurs initiales des attributs **10**

5 Lot d'instances **11**

6 Rebouclage **12**

```
entity Satellite ;
   nom: String ;
   id:Integer ;
end_entity ;
```

FIGURE 1 – Un exemple de specification avec *minispec*

minispec est un langage de modélisation de données. Il permet la spécification d'entités, comprenant un ensemble d'attributs typés. Le projet consiste à mettre en œuvre un noyau logiciel utilisant *minispec* comme langage pivot pour de la génération de code et pour la mise en œuvre de langages spécialisés.

Le projet se constitue d'une suite d'exercices. Chaque exercice ajoute un besoin supplémentaire. L'idée est de faire évoluer progressivement votre mise en œuvre pour répondre aux nouveaux besoins. La résolution d'un exercice peut donc nécessiter de revoir la façon dont vous avez traité les exercices précédents.

1 Une version minimale pour démarrer

Vous allez développer un premier méta-modèle pour une version minimale du langage *minispec* et un outil de génération de code vers Java.

Le travail à éffectuer consiste à développer une première version simple d'un générateur de code vers Java.

La première version de *minispec* intègre uniquement la déclaration d'entités avec des attributs de type primitif (String, Integer...). La figure 1 montre un exemple simple de specification avec *minispec*. Une telle spécification peut servir de source à un générateur de code vers un langage de programmation. Par exemple, il est possible de générer le code *Java* de la figure 2

Comme le montre la figure 3, pour générer ce code, les deux étapes suivantes sont mises en oeuvre :

1. il faut analyser le code *minispec* et construire une représentation mémoire de ce code sous la forme d'un graphe d'objets ;
2. il faut parcourir le graphe d'objet pour produire une chaines de caractères contenant le code *Java*.

```java
public class Satellite {
   String nom ;
   Integer id ;
   public Satellite() { }
   public String getNom() { return nom ; }
   public void setNom(String nom) { this.nom = nom ; }
}
```

FIGURE 2 – Le code Java pouvant être généré à partir de notre exemple

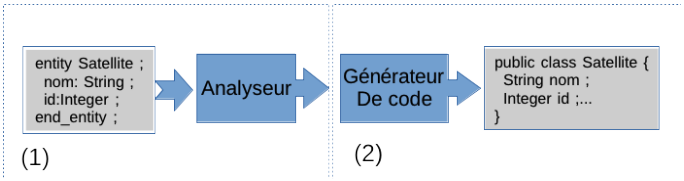

FIGURE 3 – Mise en oeuvre classique d'une génération de code

Pour l'analyse du code *minispec*, il faut disposer de la grammaire du langage utilisé en entrée et développer l'analyseur pour le langage. Pour une syntaxe textuelle telle que celle utilisée dans l'exemple, on peut utiliser un générateur de compilateur tel que Lex/Yacc. La création de la représentation interne (graphe d'objets) est alors programmée dans les actions sémantiques.

Une seconde solution plus simple consiste à utiliser une syntaxe XML pour représenter le code *minispec*. Cette solution permet de tester la génération de code sans avoir à programmer un analyseur pour la syntaxe textuelle

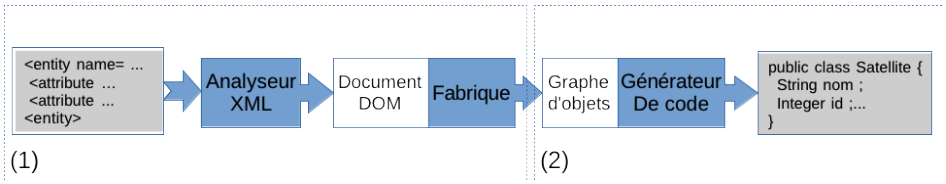

FIGURE 4 – Mise en oeuvre d'une génération de code en utilisant XML pour la représentation source

```
<entity name="Satellite">
  <attribute name="nom" type="String"/>
  <attribute name="id" type="Integer"/>
</entity>
```

FIGURE 5 – Specification de notre exemple de modèle en XML

FIGURE 6 – Un métamodèle de minispec présenté en UML

spécifique. Cette solution est montrée par la figure 4. En entrée, la première étape lit un fichier XML qui représente une spécification en minispec. Par exemple, le code XML de la figure 5 peut être utilisé comme l'équivalent de notre exemple de modèle avec l'entité Satellite :

L'analyseur XML produit un document DOM. Pour l'étape 2, une fabrique exploite le document DOM pour produire le graphe d'objets de la représentation interne. La représentation interne est parcourue à l'aide d'un visiteur qui produit le code Java.

Pour générer le code *Java*, on utilise une représentation interne issue du modèle XML en entrée. Cette représentation interne est un ensemble d'instances constituant un graphe d'objets. Les classes de ces objets sont décrits par un modèle particulier : le méta-modèle de *minispec*. Ces classes peuvent être par ailleurs décrite à l'aide d'un modèle UML constituant ainsi une autre représentation du méta-modèle de *minispec*. La représentation interne est dite conforme aux méta-modèles. Le modèle UML d'un exemple de méta-modèle pour *minispec* est montré dans la figure 6.

La figure 8 montre le squelette des classes Java correspondant au modèle UML 6.

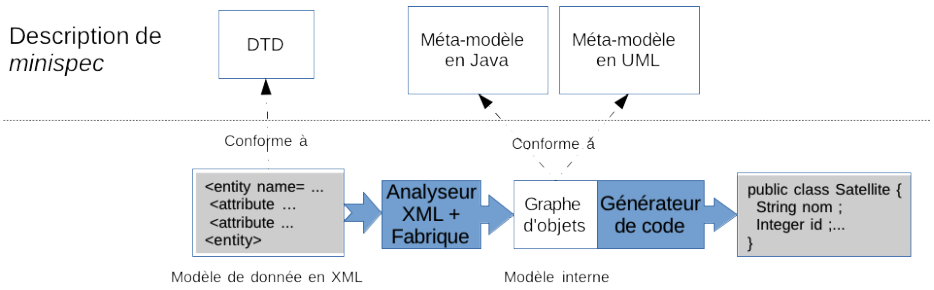

FIGURE 7 – Schéma de la chaine d'outils avec les méta-modèles

```
1  class Model {
2     String nom ;
3     List<Entity> entities;
4  }
5  class Entity {
6     String nom ;
7     List<Attribute> attributes;
8  }
9  class Attribute {
10    String nom ;
11    String type;
12 }
```

FIGURE 8 – Un métamodèle de minispec présenté en Java

2 Associations unidirectionnelles entre entités

2.1 Association simple

Avec *minispec*, une association unaire d'une entité A avec une entité B est modélisée à l'aide d'un attribut de l'éntité A dont le type est B. Cette association spécifie que toute instance de l'entité A est associé avec une instance de l'entité B. La figure 9 montre une association simple entre l'entité *Satellite* et l'éntité *Flotte*. Cette association est spécifiée par l'attribut *parent*. Le nom *Flotte* est utilisé comme type de l'attribut parent. La figure 10 montre comment représenter cette association en XML. On remarque que le nom *Flotte* est directement utilisé comme valeur de l'attribut *type* pour indiquer le type

```
entity Satellite ;
   nom: String ;
   id:Integer ;
   parent : Flotte;
end_entity ;
entity Flotte;
end_entity;
```

FIGURE 9 – Un exemple de specification avec une association simple entre deux entités

```
<entity name="Satellite">
   <attribute name="nom" type="String"/>
   <attribute name="id" type="Integer"/>
   <attribute name="parent" type="Flotte"/>
</entity>
<entity name="Flotte">
</entity>
```

FIGURE 10 – Specification d'une association simple en XML

de l'attribut *parent*.

2.2 Association multiple

Une association multiple est utilisée pour indiquer une relation d'une instance d'entités avec potentiellement plusieurs autres instances d'entités. Dans la figure 11, l'entité *Flotte* est complétée avec l'attribut *satellites* qui spécifie une collection d'instances de *Satellite*.

```
entity Flotte;
   satellites : List of Satellite;
end_entity;
```

FIGURE 11 – Une association multiple entre deux entités

```
entity Flotte;
  satellites : List [1:10] of Satellite;
end_entity;
entity Satellite;
  panneaux : Array [2] of PanneauSolaire;
end_entity;
```

FIGURE 12 – Exemples d'associations multiples avec des cardinalités (List) ou une taille fixe (Array)

Il peut être interessant d'indiquer des cardinalités min et max. Par exemple, dans la figure 12, les cardinalités indiquent explicitement qu'une flotte comporte au minimum 1 satellite et au maximum 10. Ces informations peuvent être utilisées pour implanter des controles dans les méthodes permettant de modifier les collections ou générer des tests.

Le type de collection utilisé peut etre *List*, *Set*, *Bag* ou *Array*. Attention donc aux points suivants :

- Le métamodèle doit être adapté pour supporter les collections à plusieurs dimensions et les collections de natures différentes (*List*, *Array*, *Set*, *Bag*) ;
- Un tableau (Array) est de taille fixe, les autres collections (List, Set, Bag) peuvent avoir une cardinalité minimale et maximale ;

La première version du méta-modèle montrée dans la figure 8 n'est plus suffisante. Il vous faut l'adapter pour pouvoir stocker les informations concernant les associations multiples avec les cardinalités ou la taille.

De même, il vous faut déterminer une syntaxe XML adéquate pour spécifier une association multiple.

2.3 Association multiple à deux dimensions ou plus

Il est courant de vouloir spécifier des associations multiples à plusieurs dimensions. Par exemple, considérons l'exemple de modélisation d'un jeu de bataille navale. Une telle modélisation peut contenir une entité *Grille* comprenant un attribut *cases* correspondant à la matrice de cases du jeux. L'entité Grille est montrée dans la figure 13.

Attention, pour la spécification XML, utiliser directement le nom d'un type pour le référencer n'est plus possible à cause de la seconde dimension

```
entity Grille;
   cases : Array [25] of Array [25] of Boolean;
end_entity;
```

FIGURE 13 – Une association multiple à plusieurs dimensions

```
entity Chaise subtype of (Meuble);
end_entity;
```

FIGURE 14 – Utilisation de l'héritage

du tableau (type anonyme).

3 Héritage et Modèle

3.1 Introduction de l'héritage

Une entité peut hériter d'une autre entité. L'héritage est simple, une entité ne peut hériter que d'une seulle autre entité. Un exemple d'héritage est montré par la figure 14. Dans cet exemple, l'entité *Chaise* hérite de *Meuble*.

Du fait de l'héritage, certaines erreurs d'utilisation de minispec peuvent etre introduites. Les deux erreurs courante sont la circularité (B hérite de A qui hérite de B) et la définition multiple d'attributs (l'entité A hérite de B, A et B contiennent toutes les deux un attribut de nom i).

Il vous faut donc adapter le méta-modèle et la génération de code pour intégrer la relation d'héritage. Il vous aussi vérifier la spécification avant génération de code pour éviter les dépendances circulaires et les définitions multiples d'attributs.

3.2 Modèle et résolution des dépendances

Les entités appartiennent à un modèle. Dans minispec, un modèle correspond au package en Java. Un modèle porte donc un nom et permet de regrouper des entités.

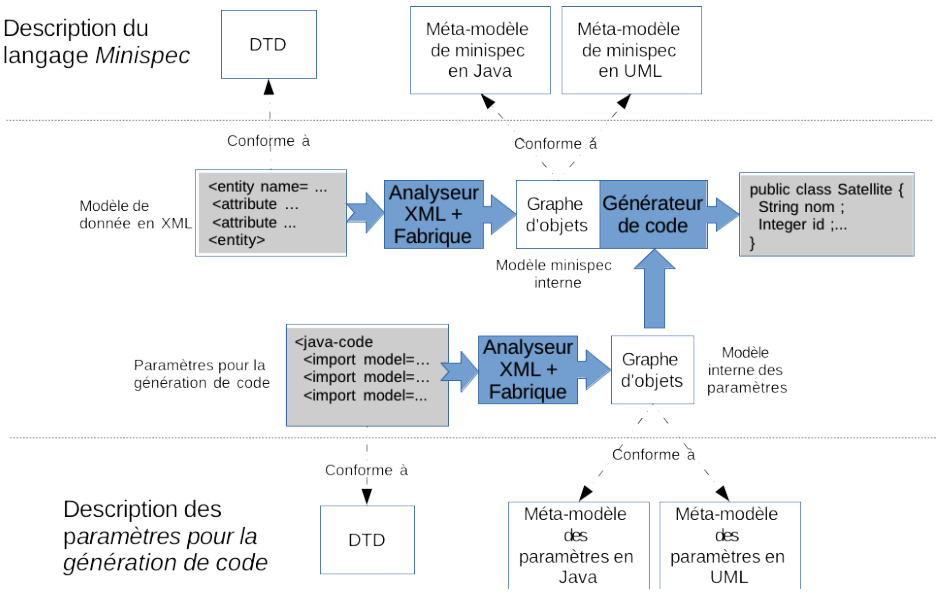

FIGURE 15 – Schéma de la chaine d'outils avec les méta-modèles pour minispec et les paramètres de génération de code

Jusqu'à présent le concept de modèle a été éludé. Or, il est important de pouvoir produire du code valide qu'il n'est pas nécessaire de corriger manuellement après génération de code. En java, les dépendances sont explicitement indiquées via les *imports*. Les imports doivent donc être ajoutés automatiquement par le générateur de code. Pour générer les imports, il faut donc se baser sur les modèles de *minispec*. Cependant, il peut être utile d'introduire un second méta-modèle pour permettre de paramétrer correctement la production des imports. Le schéma de la figure 15 montre la chaine d'outils prenant en compte ce nouveau méta-modèle introduit pour le paramétrage de la génération de code.

Le méta-modèle et la syntaxe XML pour le paramétrage doivent donc être ajoutés. Dans cette version et pour cet exercice, le contenu du paramétrage permet essentiellement d'établir le lien entre un modèle minispec et un package Java et d'indiquer le lien entre des types primitifs et leur équivalent en Java. La figure16 montre un exemple de paramétrage.

Un outil supplémentaire de gestion des dépendances entre modèles doit aussi être mise en oeuvre. Cet outil doit permettre de récupérer et d'ajouter

```
<java-code>
  <model name="Meuble" package="m2tiil.meuble"/>
  <model name="Bataille" package="m2tiil.batailleNavale"/>
  <primitive name="String" type="String"/>
  <primitive name="Integer" type="Integer"/>
  <primitive name="List" type="ArrayList" package="Java.util.ArrayList"/>
  <primitive name="Array" type="Array" package="m2tiil.Array"/>
  <primitive name="Set" type="HashSet" package="Java.util.HashSet"/>
  <primitive name="Bag" type="Bag" package="m2tiil.Bag"/>
</java-code>
```

FIGURE 16 – Exemple de paramétrage en XML

à un package java généré tous les imports nécessaires à sa compilation. Cet outil s'appuie sur un arbre de dépendances entre modèles construit avant génération de code et sur les paramètres de génération de code qui permettent d'ajouter les imports necessaires.

4 Valeurs initiales des attributs

Jusqu'à présent, le générateur de code que vous avez programmé n'a pas tenu compte des valeurs initiales des attributs des entités et en conséquence, celle des variables d'instances des classes crées. Les valeurs initiales sont donc celle par défaut en java.

Cependant, il peut être très utile de pouvoir indiquer des valeurs initiales. La figure 17 montre l'exemple de l'entité *Bateau* avec des valeurs initiales. Une valeur initiale est une expression qui peut être une valeur primitive (comme pour l'attribut *valide*) ou bien l'appel d'une fonction ou d'un constructeur (comme pour les attributs *position* et *id*).

Pour tenir compte des valeurs initiales, il vous faut modifier le méta-modèle de minispec, le format XML utilisé pour spécifier des modèles et enfin le générateur de code.

```
entity Bateau;
   position : Point := Point(0,0);
   nom : String;
   id : Integer := nextId();
   valide : Boolean := false;
end_entity;
```

FIGURE 17 – Spécification de valeurs initiales avec minispec

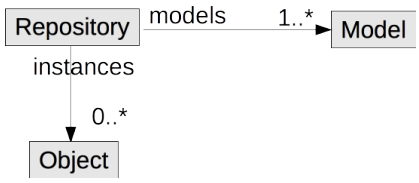

FIGURE 18 – Modélisation d'un lot d'instances en UML

5 Lot d'instances

Un lot d'instances est un objet qui gère un ensemble d'instances d'un ou plusieurs modèles. Physiquement, il est représenté par un fichier XLM. En mémoire, un lot comprend les instances et les modèles comprenant les entités dont les instances sont conformes. La figure 18 montre le modèle d'un lot d'instances spécifié comme un *Repository*.

Les trois fonctions principales d'un lot d'instances sont l'enregistrement, la sérialisation et la matérialisation des instances :

- L'enregistrement d'une instance consiste à l'intégrer au lot d'instance.
- La sérialisation consiste à produire un fichier (par exemple XML) à partir de toutes les instances enregistrées dans un lot d'instances.
- La matérialisation consiste à lire un fichier et à reconstituer les instances et les enregistrer dans le lot d'instances.

Par exemple, considérons le modèle de la flotte de satellite de la figure 19. Un lot d'instances pour ce modèle contient des instances de l'entité *Flotte* et de l'entité *Satellite*. Un résultat possible de la sérialisation d'un tel lot d'instances est montré par la figure 20. Dans cet exemple, on met en oeuvre les références à l'aide d'identifiants (attribut *id*).

```
entity Flotte;
   satellites : List [1:10] of Satellite;
end_entity;
entity Satellite;
end_entity;
```

FIGURE 19 – Exemple de modèle simple

```
<Satellite id="#1">
</Satellite>
<Satellite id="#2">
</Satellite>
<Satellite id="#3">
</Satellite>
<Flotte satellites="(#1 #2 #3)">
</Flotte>
```

FIGURE 20 – Un exemple de sérialisation

Vous devez donc modifier votre générateur de code Java pour générer automatiquement un *Repository* spécifique à un ou plusieurs modèles. Le nom du repository peut etre donné par le paramétrage du générateur de code mis en oeuvre précédemment. Les fonctions de sérialisation et de matérialisation du *Repository* sont aussi générées automatiquement.

La mise en oeuvre du Repository généré peut hériter d'une classe abstraite, par exemple AbstractRepository qui met en oeuvre une partie des fonctions de façon générique. D'autre part, les fonctions de sérialisation et de matérialisation sont basées sur la description des entités présentes dans les modèles associés au lot d'instances.

6 Rebouclage

A ce stade de la mise en oeuvre de minispec, on peut remarquer qu'une partie ce qu'on a mis en oeuvre consiste en du code que l'on sait générer à partir d'un modèle minispec :

— la définition d'une entité permet de générer la classe avec les accesseurs

```
model Minispec;
  entity NamedElement;
    name : String;
  end_entity;
  entity ModelDesc subtype of (NamedElement);
    entities : List of EntityDesc;
  end_entity;
  entity EntityDesc subtype of (NamedElement);
    attributes : List of AttributeDesc;
  end_entity;
  entity AttributeDesc subtype of (NamedElement);
    domain : TypeDefinition;
    defaultValue : String;
  end_entity;
  entity TypeDefinition;
  end_entity;
  entity EntityType subtype of (TypeDefinition);
    referenced : String;
  end_entity;
  entity CollectionType subtype of (TypeDefinition);
    ...
  end_entity;
  ...
end_model;
```

FIGURE 21 – Ebauche de métamodèle de minispec en minispec

en lecture et écriture,

— le lot d'instance généré permet la sérialisation et la matérialisation à partir de XML.

Nous pouvons donc utiliser notre chaine d'outils pour *minispec* lui même en spécifiant le métamodèle de *minispec* avec *minispec*. La figure 21 montre une ébauche de méta-modèle simplifié de *minispec* exprimé avec *minispec*. Ce métamodele peut être exprimé à l'aide de notre format XML et donc être utilisé par notre chaine d'outils pour générer du code java. Or il se trouve que le code java généré peut constituer une partie de la mise en oeuvre de nos outils pour *minispec*...

A vous donc de définir votre métamodèle en vous étant fidèle à votre mise en oeuvre de *minispec* et de générer et utiliser le code java généré pour mettre en oeuvre votre propre chaine d'outils.