GENERATION DE CODE

2018 - 2019

Robin LOUARN
Robin.louarn@icloud.com

Table des matières

Table des figures et annexes

1 Introduction

Minispec est un langage de modélisation de données. Il permet la spécification d'entités, comprenant un ensemble d'attributs typés.

```
1  entity Satellite;
2     nom: String ;
3     parent:Flotte ;
4  end_entity ;
5
6  entity Flotte;
7     satellites : List of Satellite;
8  end_entity;
```

Figure 1 - Exemple d'une entité Minispec

Le projet consiste à mettre en œuvre un noyau logiciel utilisant minispec comme langage pivot pour de la génération de code.

J'ai réalisé ce projet de manière itérative. Chaque itération ajoute un besoin supplémentaire. L'idée est de faire évoluer progressivement la mise en œuvre pour répondre aux nouveaux besoins.

Je vais maintenant présenter ma version finale qui se compose de deux parties :

- Présentation de la génération d'un projet Java complet
- Présentation de la gestion d'un lot d'instances

2 Présentation de la génération d'un projet Java complet

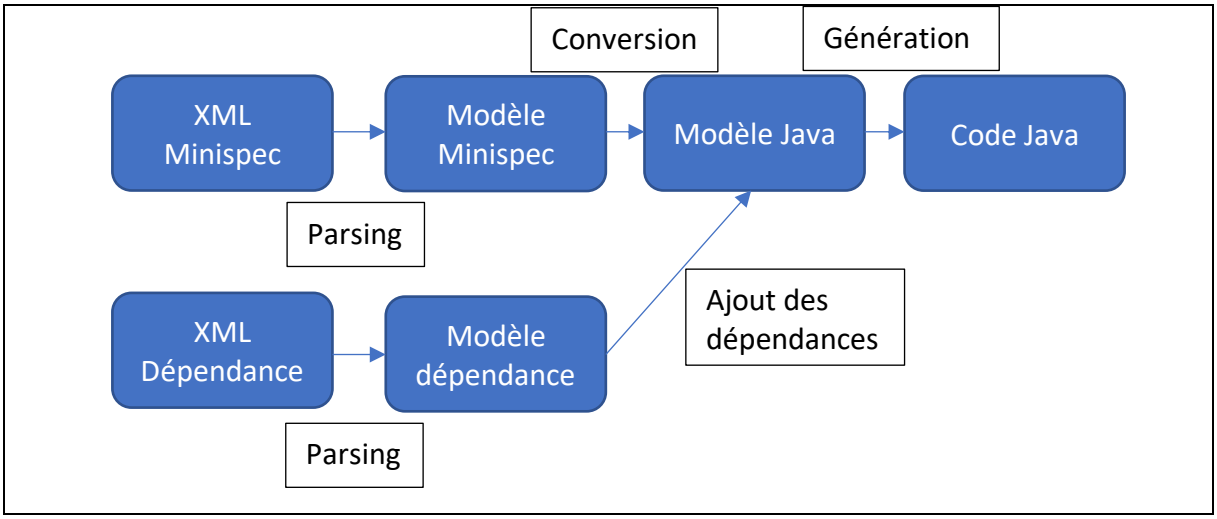

Figure 2 - Schéma de la chaine d'outils

Pour générer le code Java, je réalise plusieurs étapes :

- générer le modèle minispec à partir du fichier XML qui contient les entités
- convertir le modèle minispec en modèle Java
- générer le modèle de dépendance à partir du fichier XML qui contient les dépendances
- associer les dépendances aux classes Java
- générer le code Java

Je vais ci-dessous expliquer les différents blocs présents dans la *figure 2*.

2.1 Présentation du modèle Minispec

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 la *figure 1*. Une seconde solution plus simple consiste à utiliser une syntaxe XML pour représenter le code Minispec.

```
1  <?fr.ubo.m2tiil.louarn.xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE modeleJava SYSTEM "minispec.dtd">
3  <modeleMinispec name="modele">
4     <entity name="Satellite">
5        <attributeMinispec name="nom">
6           <typeElement type="String"/>
7        </attributeMinispec>
8        <attributeMinispec name="parent">
9           <typeElement type="Flotte"/>
10       </attributeMinispec>
11    </entity>
12    <entity name="Flotte">
13       <attributeMinispec name="satellites">
14          <collection typeCollection="List">
15             <typeElement type="Satellite"/>
16          </collection>
17       </attributeMinispec>
18    </entity>
19 </modeleMinispec>
```

Figure 3 - Spécification du modèle Minispec en XML

La première étape consiste à créer un fichier XML qui représente le fichier Minispec.
Par exemple, le code XML de la *figure 3* est l'équivalent des entités Balise et Satellite de la *figure 1*. La DTD de la *figure 3* est représentée dans l'*annexe 1*.

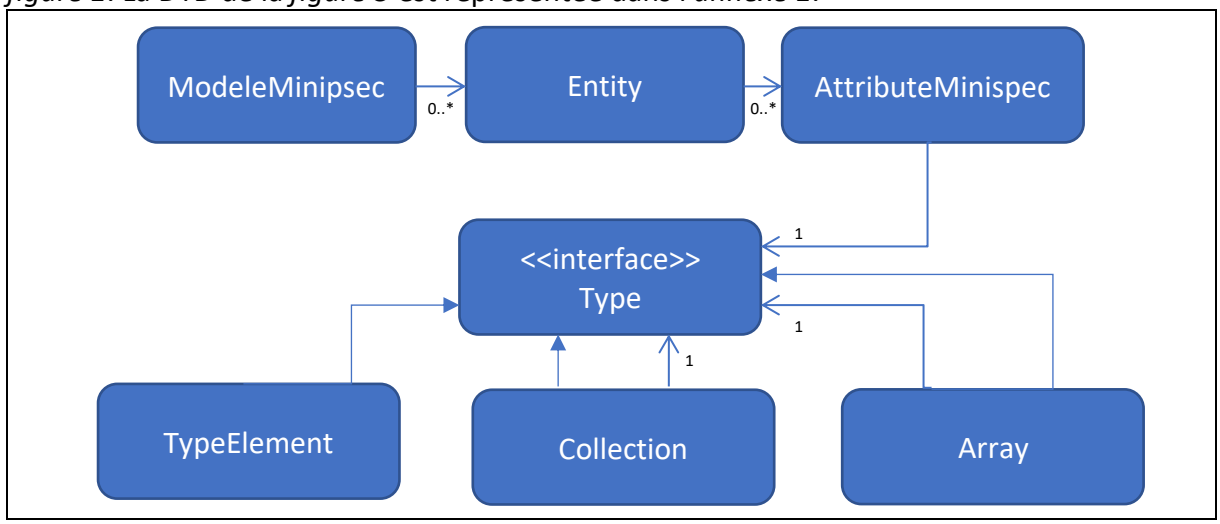

Figure 4 - Méta modèle de Minispec en UML

Puis un analyseur XML produit un document DOM ce qui permet à une *factory* d'exploiter ce document pour produire le graphe d'objets représenté par la *figure 4*.

2.1.1 Association simple et multiple

Pour gérer les associations simples et multiples, j'ai créé l'interface « *Type* » décrite dans la *figure 4*. Cette interface est implémentée par 3 classes :

- *TypeElement* : elle permet de représenter un type primitif ou une entité,
- *Collection* : elle permet de représenter une collection,
- *Array* : elle permet de représenter un tableau.

A titre, la *figure 3* montre :

- une association simple entre l'entité *Satellite* et l'entité *Flotte* avec l'attribut *parent*
- une association multiple avec l'attribut *Satellite* de l'entité *Flotte* qui spécifie une collection d'instances de *Satellite*

2.2 Conversion du document DOM en graphe d'objet Minispec

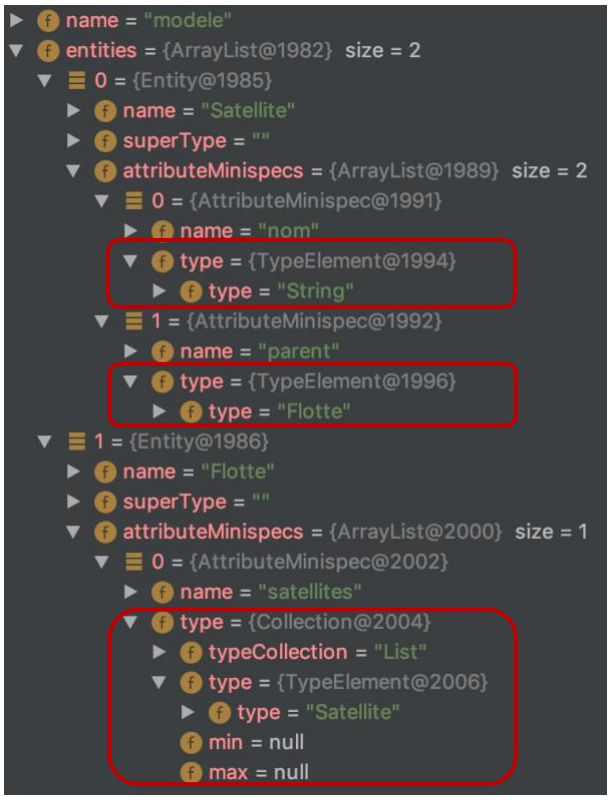

Figure 5 - Graphe d'instances du modèle Minispec

Le but de cette phase est de créer le lot d'instances présenté dans la *figure 5,* décrit par le fichier XML de la *figure 3*. J'ai, donc, créé une *factory* qui prend en entrée un document DOM et qui génère ce lot d'instances. La classe qui gère ce traitement est « *ParserXmlMinispec* » (pour plus de détail Cf. la classe *JUnit* qui permet de tester la *factory)*.

La partie la plus complexe de la conversion fut la matérialisation de la balise « *<attributeMinispec>* » et plus particulièrement le type de l'attribut.

```
private Type getType(Element element) {
    switch (element.getNodeName()) {
        case "array":
            Array array = new Array();
            array.setSize(new Integer(element.getAttribute("size")));
            array.setType(this.getTypeElement(element));
            return array;
        case "collection":
            Collection collection = new Collection();
            collection.setTypeCollection(element.getAttribute("typeCollection"));
            // ce champ n'est pas obligatoire
            // si une exeption survient il ne faut pas metre a jour la valeur
            try {
                collection.setMin(new Integer(element.getAttribute("min")));
            } catch (NumberFormatException e) {}
            // ce champ n'est pas obligatoire
            // si une exeption survient il ne faut pas metre a jour la valeur
            try {
                collection.setMax(new Integer(element.getAttribute("max")));
            } catch (NumberFormatException e) {}
            collection.setType(this.getTypeElement(element));
            return collection;
        case "typeElement":
            TypeElement typeElement = new TypeElement();
            typeElement.setType(element.getAttribute("type"));
            return typeElement;
}
```

Figure 6 - Méthode Java pour créer le lot d'instances d'un type d'un attribut Minispec

Pour atteindre mon objectif, j'ai élaboré la méthode récursive affichée dans la *figure 6* pour créer un lot d'instances de « *Type* ». Un exemple de ce lot d'instances est présenté dans la *figure 5 (les parties encadrées en rouge)*.

Dans la méthode « *getType »,* quand on rencontre une balise de type *collection* ou *array* alors la méthode va s'appeler récursivement pour obtenir le type de la collection ou du tableau. La récursivité s'arrête quand on tombe sur une balise de type *TypeElement*.

2.3 Conversion du modèle minispec en modèle java

Il n'est pas nécessaire de convertir le modèle Minispec en modèle Java pour générer le code Java.

Mais grâce à cette conversion, si je voulais ultérieurement ajouter la gestion d'un autre langage de modélisation, il me faudrait juste créer une nouvelle *factory* et un nouveau *converter* vers le modèle Java. Toute la chaine d'outils ne sera pas à refaire car celle-ci ne se base pas sur le modèle de modélisation (dans notre cas Minispec).

Si j'utilisais le modèle de modélisation pour la suite de la chaine d'outils et que j'ajoutais un autre système de modélisation, je devrai alors modifier les outils qui permettent :

- la vérification de l'héritage,
- l'intégration des dépendances Java,
- le générateur de code.

Ce système de conversion permet l'ajout de nouvelles classes pour créer de nouvelles fonctionnalités et non la modification des classes existantes ce qui serait susceptible de provoquer de grosses régressions.

2.4 Intégration des dépendances au modèle Java

```
<!ELEMENT java-code (depandance)*>
    <!ELEMENT depandance (#PCDATA)>
    <!ATTLIST depandance
        name CDATA #REQUIRED
        type CDATA #REQUIRED
        package CDATA #IMPLIED>

<?fr.ubo.m2tiil.louarn.xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-code SYSTEM "java-code.dtd">
<java-code>
  <depandance name="List" type="List" package="java.util.List"/>
</java-code>
```

Figure 7 - Spécification du modèle de dépendance en XML et sa DTD

Pour générer du code Java sans erreurs de compilation, il faut ajouter les dépendances aux classes du modèle Java.

```
public interface VisitorCommun {
        void visite(Array array);
        void visite(Collection collection);
        void visite(TypeElement typeElement);
}
```

Figure 8 - Interface de la classe VisitorDependance

```
// ajout des depandance dans les class du modele
for (Clazz clazz : modeleJava.getClazzes()) {
  clazz.setDependances(visitorDependance.getClazzDependances(clazz));
}
```

Figure 9 - Ajout des dépendances aux classes du modèle

Ce traitement s'effectue une fois le modèle Minispec converti en modèle Java. Il ne reste plus qu'à parcourir toutes les classes du modèle et à appliquer le visiteur « *VisitorDependance* » Cf. *figure 9*.

Le visiteur connait la liste des dépendances, dès qu'il tombera sur un objet du type « *Type* », il visitera Cf. *figure 8* et créera une liste avec seulement les dépendances utilisées par la classe.

2.5 Génération du code Java

```
public interface VisitorJava {
        void visite(Argument argument);
        void visite(AttributeJava attributeJava);
        void visite(Clazz Clazz);
        void visite(Constructeur constructeur);
        void visite(Methode methode);
```

```
7           void visite(ModeleJava modeleJava);
8           void visite(Bloc bloc);
9           void visite(MotsCles motsCles);
10          void visite(List<MotsCles> motsCles);
11          void visite(Dependance dependance);
12  }
```

Figure 10 - Interface VisitorJava

```
1   String PATH_TARGET_GENERATE_SOURCE = "/Users/rlouarn/Desktop/ProjetGenere/src";
```

Figure 11 - Constante de la classe Constants

La classe qui génère le code Java est un visiteur qui implémente les interfaces des *figures 8 et 10* ce qui lui permet de parcourir l'ensemble du modèle Java. Ce visiteur va créer un fichier pour chaque classe avec le contenu de celle-ci.

Pour paramétrer l'emplacement du projet, il faut modifier la constante de la *figure 11*.

Si on prend l'exemple de la *figure 3* comme entrée du programme alors les classes Satellite et Flotte seront générées.

```
1   package modele;
2   import modele.Flotte;
3   public class  Satellite {
4           public String nom;
5           public Flotte parent;
6           public  Satellite() {
7           }
8           public String getNom(){
9                   return this.nom;
10          }
11          public void setNom(String nom){
12                  this.nom = nom;
13          }
14          public Flotte getParent(){
15                  return this.parent;
16
17          }
18          public void setParent(Flotte parent){
19                  this.parent = parent;
20          }
21  }
```

Figure 12 - Classe générée

2.6 Présentation de la gestion d'un lot d'instances

Je n'ai malheureusement pas eu le temps de générer automatiquement le lot d'instances. Mais je l'ai directement codé. En perspective de sa génération automatique j'ai créé un parser pour créer un modèle Java à partir d'un fichier XML. Ce parser aurait servi à créer les classes qui ne changent pas en fonction du code généré. Ces classes sont :

- AbstractInstance
- AbstractRepository
- VisitableInstance
- VisiteurInstance
- GlobalRepository

Si on prend l'exemple de la figure 3, les classes générées seraient :

- Balise
- Satelite
- VisiteurInstanceEcrire
- FlotteRepository
- BaliseRepository
- Instance.dtd

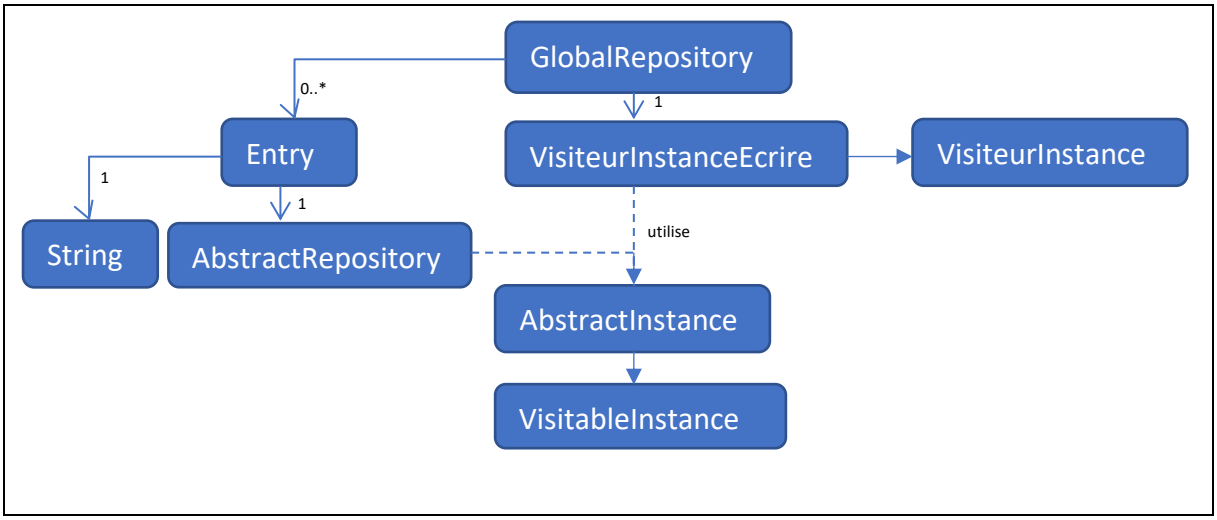

Figure 13 - Modèle de classe du projet généré en UML

2.6.1 Rôle des classes

2.6.1.1 L'interface GlobalRepository

```
1  public interface GlobalRepository {
2      private Map<String, AbstractRepository> repositoryMap;
3      private Set<AbstractInstance> instances;
4      public Document ecrire();
5      public List<AbstractInstance> lire(Document document)
```

```
6  }
```

Figure 14 - Interface de la classe GlobalRepository

Cette classe permet la lecture d'un document XML et de le transformer en instance. Elle permet aussi de faire l'opération inverse.

Pour lire un lot d'instances, j'utilise le nom de la balise XML qui permet la récupération d'un *repository*. Grâce à ce *repository,* je peux créer une « *AbstactInstance »* à partir d'un élément du document XML. Ce *repository* est lié au nom d'une balise XML grâce à la map *ligne 2* de la *figure 14*.

Pour écrire un document XML, j'utilise le visiteur « *VisiteurInstance* » ce qui me permet de visiter les instances stockées dans la collection *ligne 3*.

2.6.1.2 La classe AbstractRepository

```
1  public abstract class AbstractRepository {
2      private static Map<String, AbstractInstance> INSTANCES;
3      public abstract AbstractInstance lire(Element element, Document document);
4      public abstract Element ecrire(Instance instance, Document document);
5  }
```

Figure 15 - Interface de la classe AbstractRepository

Si on prend l'exemple de la figure 3, les classes « *SatelliteRepository* » et « *FlotteRepository* » seront générées. Ces deux classes hériteront de la classe « *AbstractRepository* » ce qui permettra soit de lire soit d'écrire une instance du modèle généré.

La map « *INSTANCES* » à la *ligne 2* de la *figure 15* permet d'empêcher la création d'une instance si celle-ci existe déjà. Ce contrôle est indispensable si on se retrouve dans le cas suivant :

1. la classe A a un attribut de type B
2. B à un attribut de type A
3. lors de la création de A
4. on va créer B
5. puis on va créer A
6. et on crée B…

Sinon on se retrouve dans une boucle infinie.

2.6.1.3 La classe AbstractInstance

```
1  public abstract class AbstractInstance implements VisitableInstance {
2      private String id;
3  }
```

Figure 16 - Interface de la classe AbstractInstance

```
1  <!ELEMENT Instances (SateliteInstance|FlotteInstance)*>
2      <!ELEMENT SateliteInstance EMPTY>
3      <!ATTLIST SateliteInstance
```

```
4               id ID #REQUIRED
5               nom CDATA #REQUIRED
6               parent IDREF #REQUIRED>
7       <!ELEMENT FlotteInstance EMPTY>
8       <!ATTLIST FlotteInstance
9               id ID #REQUIRED
10              satelites IDREFS #IMPLIED>
11  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
12  <!DOCTYPE Instances SYSTEM "instance.dtd">
13  <Instances>
14      <SateliteInstance id="id1" nom="satelite1" parent="id0"/>
15      <FlotteInstance id="id0" satelites="id1 "/>
18  </Instances>
```

Figure 17 – Représentation d'une instance en XML

Une instance possède obligatoirement un id ligne 2 de la figure 16 car c'est cet id qui me permet de le référencer dans le document XML.
Dans le ficher XML, les associations simples sont représentées par un id exemple avec l'attribut parent de *ligne 14* de la *figure 17*.
Les associations multiples sont représentées dans le document XML par une liste d'id exemple avec l'attribut *satellites* à la *ligne 15*.

3 Annexes

Annexe 1 - DTD complète de Minispec

```
<!-- modeleJava -->
<!ELEMENT modeleJava (entity)*>
<!ATTLIST modeleJava
      name CDATA #REQUIRED
      >

<!-- entity -->
<!ELEMENT entity (attributeMinispec)*>
<!ATTLIST entity
      name CDATA #REQUIRED
      supertype CDATA #IMPLIED
      >

<!-- attributeMinispec -->
<!ELEMENT attributeMinispec (
      array
      |collection
      |typeElement)>
<!ATTLIST attributeMinispec
      name CDATA #REQUIRED
      >

<!-- typeElement -->
<!ELEMENT typeElement EMPTY>
<!ATTLIST typeElement
      type CDATA #REQUIRED
      >

<!-- array -->
<!ELEMENT array (
      array
      |collection
      |typeElement)>
<!ATTLIST array
      size CDATA #REQUIRED
      >

<!-- collection -->
<!ELEMENT collection (
```

```
        array
        |collection
        |typeElement)>
<!ATTLIST collection
        typeCollection CDATA #REQUIRED
        min CDATA #IMPLIED
        max CDATA #IMPLIED
        >
```