Chapitre XII

De la modélisation aux traitements

Ce chapitre très court a pour objet de restituer ce qui a été étudié précédemment en formalisation des données et d'introduire la suite, c'"est à dire les traitements sur les données par un langage appelant (Java, C#, JS, ...).

1 Statique/Dynamique

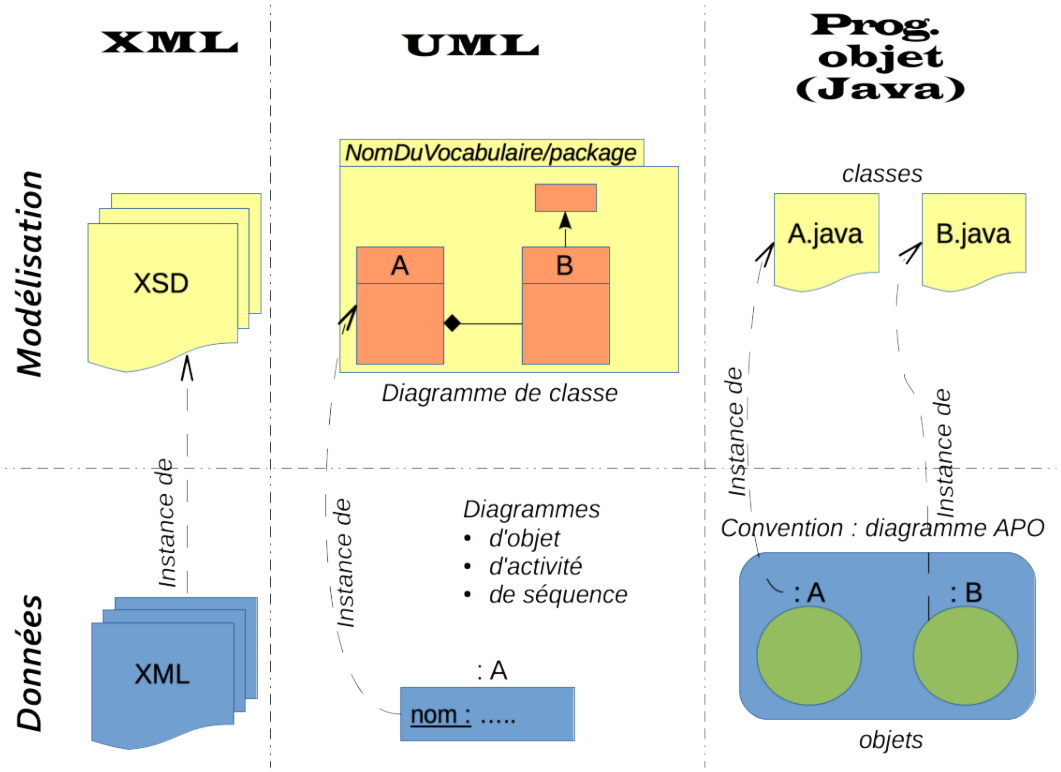

Figure XII.1: Analogie entre les modèles et les instances, en XML, UML et Java.. Evidemment, ce schéma est valable avec n'importe quel langage objet, par exemple C#.

Dans les chapitres précédents, nous avons vu l'instanciation et la modélisation de données *statiques*, respectivement en XML (instanciation) et en XMLSchema (modélisation). Pour spécifier des transfor-

mations sur ces données statiques, nous avons utilisé XSLT. Nous allons à présent voir des traitements dynamiques sur les données en utilisant Java et C#. Vous trouverez ainsi la déclinaison de ce même cours à la fois pour Java et pour C# ; le langage utilisé sera systématiquement précisé ; lorsque les exemples sont semblables dans les deux langages, on écrira `Java/C#`. Encore une fois, tout ce qui sera présenté ici et dans les chapitres à venir sur le traitement des données, est valable avec de nombreux autres langages de programmation incluant le paradigme objet.

Nous verrons au cours des chapitres qui suivent les techniques de sérialisation/désérialisation et l'usage des analyseurs syntaxiques (parsers).
Mais rappelons d'abord l'analogie forte qu'il y a entre les **schéma XSD** et les **classes d'un langage objet** ainsi qu'entre les **instances XML** qui contiennent les données et les **instances d'objets**. La figure XII.1 illustre ceci.

2 Équivalence XML Schema ↔ Java / C#

En Java/C# comme en XML, un concept égale une classe/un type. On aura donc autant de classes Java/C# que de concepts à modéliser dans un programme, plus une classe additionnelle pour héberger la méthode `main`. On pourra avoir tout autant de types XMLSchema ! Nous verrons dans le chapitre "Data Binding" que l'on peut se contenter de modéliser les données en XMLSchema, puis de réaliser une compilation de Schema XML pour générer les classes Java correspondantes. En C# ceci aussi est possible mais dépendant d'outils Microscoft.
Un type XMLSchema comme une Classe *encapsule* ses membres (attributs et méthodes en langage objet, attributs seulement en XMLSchema). Les membres d'une classe / d'un type XMLSchema la/le décrivent et lui appartiennent.
Les attributs des Classes comme des types XMLSchema peuvent la plupart du temps être assimilés.

En XML	En Java/C#
Types complexes & Types simples	Une classe
Types prédéfinis • xsd:int • xsd:double • xsd:string	Types primitifs • int • double • String (Attention, ici, String est une classe et non un type primitif, valable en Java et C#) ou string (un type primitif en C#)

Tous les types prédéfinis en XMLSchema n'ont pas toujours/exactement un type primitif équivalent en langage objet.
Pour les types de même nom, on n'a pas toujours les même valeurs possibles.

Une Classe est la description en langage objet d'un type complexe, au même titre qu'un `xs:complexType` est la description en XMLSchema d'un type de données complexe. Elle décrit la composition et les valeurs possibles pour les éléments, ses données membres nommées *attributs*. De plus, une Classe peut contenir :

- la liste des opérations/traitements/actions (nommés *méthodes*) possibles sur le type complexe
- l'implémentation (i.e. le code) de ces opérations.

La comparaison Langage Objet – XMLSchema s'arrête donc là ! En effet, comme nous le verrons dans le cours de sérialisation, XML ne peut contenir que des données et non des méthodes. Nous serons donc amenés, en Java/C#, pour plus d'efficacité, de modaluratité, de maintenabilité, surtout lors de la sérialisation, à séparer les données et les méthodes dans des classes différents. Si on reprend l'exemple du rectangle donné plus haut, en Java/C#, nous aurions alors deux classes (voir figure XII.2) : `RectangleData` et `Rectangle`, la dernière possédant un attribut du type de la première.
En XMLSchema, seul le type `RectangleData` serait modélisé (figure XII.3) et instanciable en XML (figure XII.4).

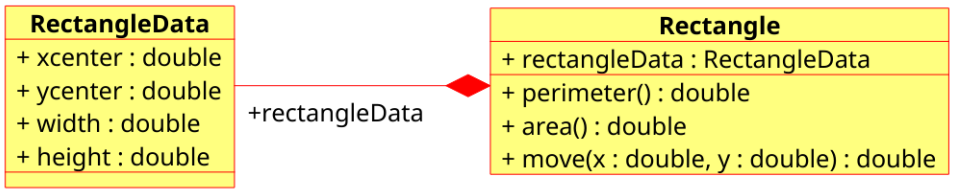

Figure XII.2: Séparation des données et des méthodes en Langage Objet (Java, C# ...).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.univ-grenoble-alpes.fr/rectangle"
        xmlns:geom="http://www.univ-grenoble-alpes.fr/rectangle"
        elementFormDefault="qualified">

    <element name="rectangleData" type="geom:RectangleData"/>

    <complexType name="RectangleData">
        <sequence>
            <element name="x" type="double" minOccurs="1" maxOccurs="1"/>
            <element name="y" type="double" minOccurs="1" maxOccurs="1"/>
            <element name="width" type="geom:DoublePositif" minOccurs="1"
                maxOccurs="1"/>
            <element name="height" type="geom:DoublePositif" minOccurs="1"
                maxOccurs="1"/>
        </sequence>
    </complexType>

    <simpleType name="DoublePositif">
        <restriction base="double">
            <minInclusive value="0"/>
        </restriction>
    </simpleType>

</schema>
```

Figure XII.3: Modélisation des données d'un rectangle.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rectangleData
        xmlns="http://www.univ-grenoble-alpes.fr/rectangle"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.univ-grenoble-alpes.fr/rectangle
            Rectangle.xsd">
    <x>-6</x>
    <y>40</y>
    <width>10</width>
    <height>20</height>
</rectangleData>
```

Figure XII.4: Instance XML contenant les données d'un rectangle.

et voici typiquement une classe C# (ici un `record struct` en fait) dont le type correspondrait à celui du schéma. Cette classe est sérialisable : ses propriétés `X`, `Y`, `Width` et `Height` le sont directement. On voit qu'elle dispose en plus d'une propriété `Area` qui calcule l'aire du rectangle. Notez les spécificateurs entre crochets pour la sérialisation. Nous verrons tout cela dans le chapitre dédié.

```
namespace Serialization;
```

```
2
3  [XmlRoot("rectangleData", Namespace = "http://www.univ-grenoble-alpes.fr/
       rectangle")]
4  public record struct Rectangle {
5      public Rectangle(double x, double y, double width, double height) {
6          X = x;
7          Y = y;
8          Width = width;
9          Height = height;
10     }
11
12     public Rectangle() {
13         X = 0;
14         Y = 0;
15         _width = null;
16         _height = null;
17     }
18
19     [XmlElement("x"/*, Namespace="geom"*/)] public double X { get; set; }
20     [XmlElement("y")] public double Y { get; set; }
21     [XmlElement("width")] public double Width {
22         get => _width??0;
23         set {
24             if (value <= 0) throw new System.Exception("Valeur invalide");
25             _width = value;
26         }
27     }
28     private double? _width;
29
30     [XmlElement("height")] public double Height {
31         get => _height??0;
32         set {
33             if (value <= 0) throw new System.Exception("Valeur invalide");
34             _height = value;
35         }
36     }
37     private double? _height;
38
39     [XmlIgnore] public double Area {
40         get => (_width * _height)??0;
41     }
42 }
```