Chapitre XIV

Data Binding - Sérialisation en C#

Nous allons voir comment lier directement des objets en mémoire (typés par des classes C# dans ce chapitre, Java dans le suivant) et des instances XML, sans passer explicitement par l'usage de parsers. Les 2 chapitres se ressemblent beaucoup. Si vous souhaitez lire la partie Java, il est toutefois recommandé de lire la partie commune dans le chapitre dédié à C#.

1 Classes et schémas XML - Equivalence

On retrouve beaucoup de principes communs entre XML Schema et la programmation orientée objet. Notamment:

- Typage
- Encapsulation
- Espaces de nom
- Héritage
- Composition
- ...

La figure XIV .1 illustre cette équivalence entre XML (schémas) et UML (diagrammes de classes) avec des modèles et instances, en XML d'une part (modèle XML schema et instances de documents), d'autre part en UML (classe et instances de classe). On remarque ainsi qu'il y a deux niveaux d'équivalence : au niveau des modèles et au niveau des données.
Au niveau des modèles, un XML Schema :(i) contient des types et (ii) permet les relations de composition (un type est contenu dans un autre), d'utilisation (un type se réfère à un autre), d'héritage (un type hérite d'un autre), tandis qu'un diagramme de classes UML (i) contient des classes (= types) et (ii) permet aussi les relations de composition, utilisation et héritage. Il y a toutefois une différence : les méthodes de classe n'ont pas de correspondance en XML Schema : XML ne contient que des données, pas des méthodes.
Au niveau des données, Un document XML (i) contient des données, et (ii) est conforme à un type (instances valides, conformes à un schéma), tandis que le diagramme d'objets UML (i) contient des données et (ii) est conforme à une classe (ce sont des instances de classes).

Il existe donc une équivalence entre {XSD/XML} d'une part et les {diagrammes de classes/diagrammes d'objets UML} d'autre part. De même, il existe une équivalence entre les {diagrammes de classes/diagrammes d'objets UML} d'une part et les {Classes/instances} des langages orientés objet (java, C++, C# ...) d'autre part. Par transitivité, il existe donc une équivalence entre {XSD/XML} d'une part et les {Classes/instances} des langages orientés objet d'autre part.

Il ne nous manque alors plus qu'un moyen (logiciel) de passer d'assurer l'équivalence des modèles XSD <–> Classes et l'équivalence des données XML<–>instances de classes. Il faut que ce moyen respecte des conventions de programmation orientée objet, en particulier les concepts d'encapsulation, mais aussi l'utilisation d'interfaces et l'instanciation cachée. Ce concept s'appelle le `Data Binding` et implémente des méthodes de `Sérialisation` et `Désérialisation`. Le data binding est un concept plus large que sa restriction au `Data Binding XML`. Dans ce cours, nous focaliserons sur ce dernier.

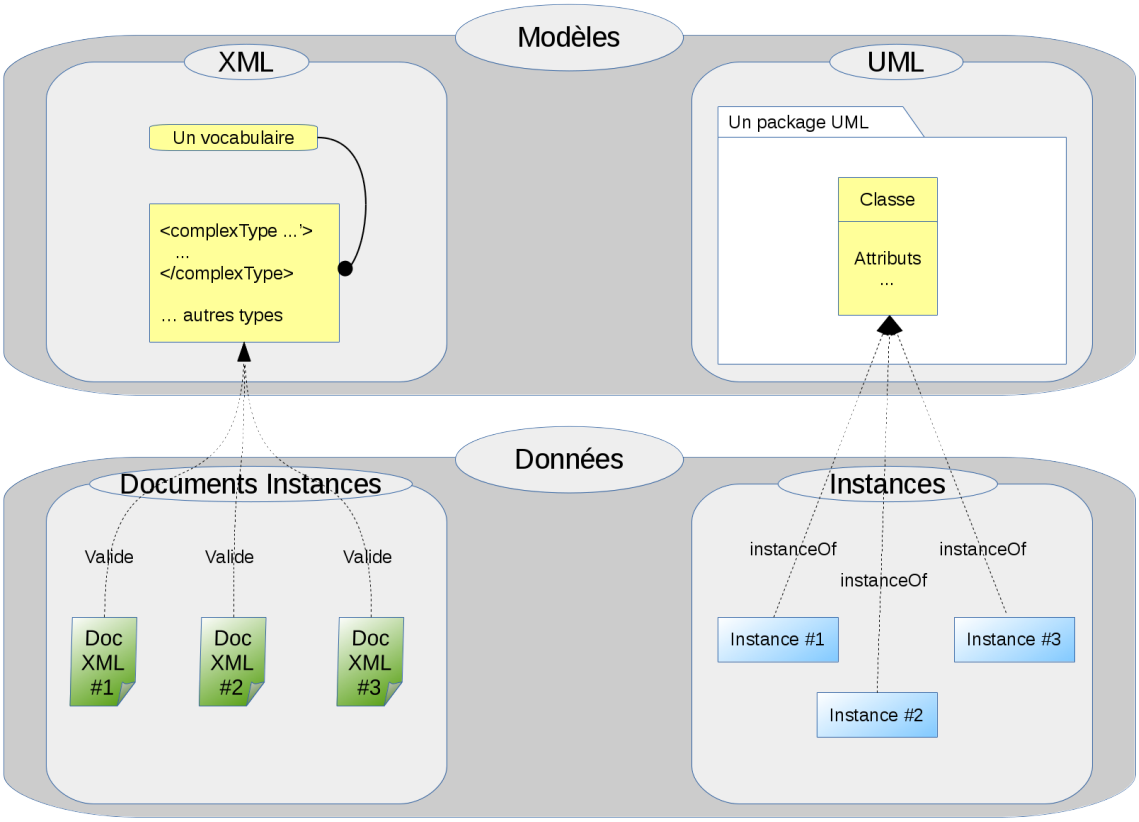

Figure XIV .1: **Relation XML - UML**.

2 Principes du Data Binding, de la sérialisation et désérialisation

Le Data Binding avec XML est un processus en 2 étapes. La première assure une correspondance des types (type XSD <-> classe) en les liant. La deuxième concerne le niveau des données (XML <-> instance objet).

La première étape consiste en à faire en sorte de développer, dans le langage objet, un modèle de données (classe, struct ...) représentant fidèlement le schéma XML, ou du moins faire en sorte qu'à chaque champ modélisé dans le schéma, on propose une équivalence qui puisse être mise en mémoire (dans une/des instance(s) de classe(s)). Vous remarquerez que c'est là que se situe la différence fondamentale avec la représentation des données XML dans un arbre DOM : l'arbre DOM lui-même, s'il contient également toutes les informations du document XML, n'a absolument pas la même structure que ce que modélise le schéma XML. Cette étape peut être faire à la main (on écrit une classe qui modélise la même chose que le schéma et on fait en sorte qu'il soit possible de faire transiter toutes les valeurs d'un objet de cette classe vers une instance XML obéissant à ce schéma (sérialisation) ou l'inverse (désérialisation). Cette étape peut aussi être réalisée par une *compilation du schéma XML*. Plusieurs API dans divers langages proposent de tels compilateurs de Schémas, par exemple JAXB en Java.

La deuxième étape est réalisée par les opérations de *marshalling* (*sérialisation*) et *unmarshalling*

(*désérialisation*) qui assurent la communication (entrées/sorties) entre les données stockées dans un document XML et les valeurs pouvant être stockées dans les attributs des objets (classes). Ces deux étapes sont représentées dans la figure XIV .2 (c'est tout aussi valable en C# évidemment).

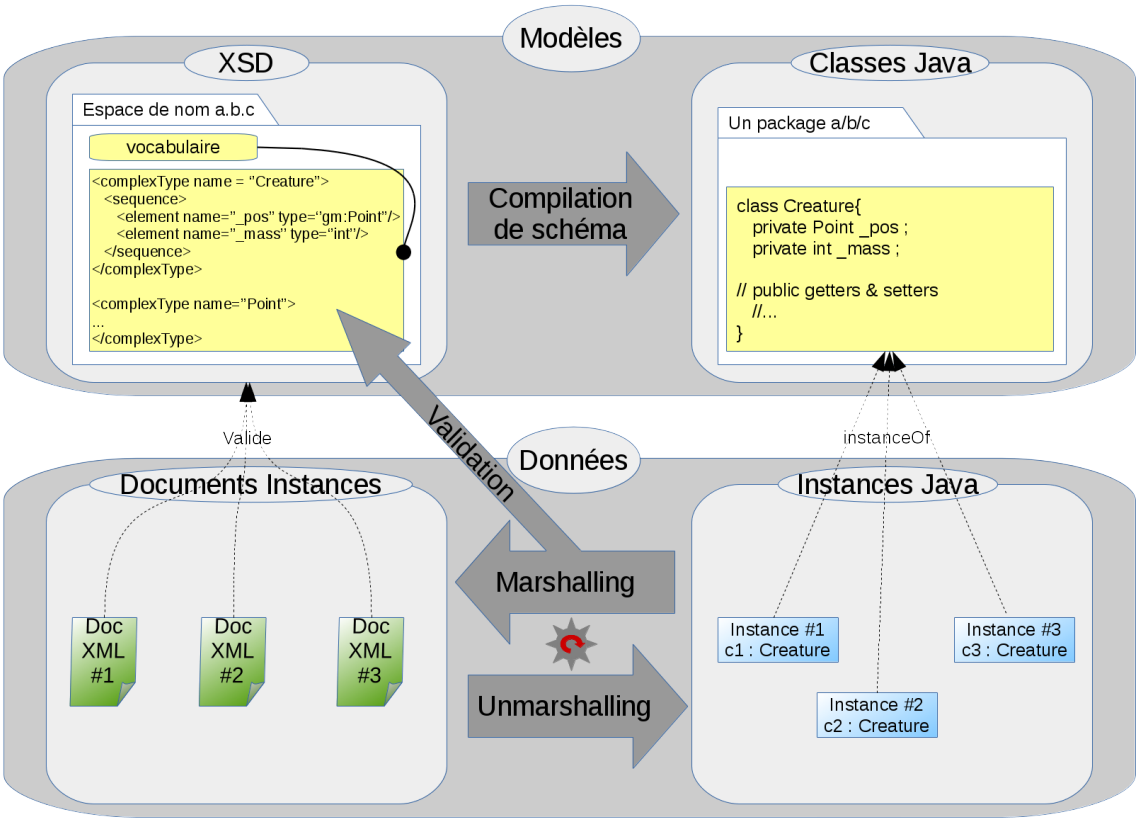

Figure XIV .2: **Relation XML - Java**.

3 Data Binding : *Design-time* ou *Run-time*

La génération des classes en Java, C, C++, C#, Python ... peut être faite durant le temps de design du programme (design-time) (l'écriture avant compilation), mais elle peut aussi être faite dans certaines situations, lorsque le langage permet d'exploiter des modules compilés à la volée, durant l'execution du programme (run-time).

Par exemple, en Python, il existe une bibliothèque, `xmlobjects` qui permet de générer des classes Python à la volée (runtime).

La mise en place du data binding se fait néanmoins principalement durant la phase de design du code (design time). Ce sera le cas dans notre pratique du Data Binding en C#.

4 Compilateurs XML Schema / sérialisation / persistance

Je limiterai ce paragraphe aux outils disponibles pour les langages Java, C++ et Python.

Parmi les principaux compilateur de schéma XML en Java, il faut compter *JAXB* et *Apache xmlbeans*. Nous recommandons l'usage de JAXB qui est simple d'utilisation, bien intégré à netbeans, prend en charge tous les types de schémas XML, produit une architecture propre avec respect des design patterns, du nom du package, etc.

En C++, on trouve aussi de nombreuses resources dédiées à la manipulation du XML (parseurs, outils de sérialisation, compilateurs de schema ...), comme *Xerces-C++* d'*Apache* (`http://xerces.apache.org/xerces-c/`), *gSoap* de la société *Genivia* (`https://genivia.com/index.html`), *LXM* de la société *CodeLogic* (`https://www.codalogic.com/lmx/c++-xml-data-binding-explained.php`), la bib-

liothèque de sérialisation de *Boost* (http://www.boost.org/doc/libs/1_65_1/libs/serialization/doc/index.html)... L'API *.Net* de *Microsoft* permet le Marshalling. Leur utilisation est cependant bien plus complexe qu'en Java.

En Python, une bibliothèque plusieurs modules simples d'utilisation sont disponibles (voir https://docs.python.org/2/library/persistence.html).

5 Avantages et limitations

Les **avantages** du Data Binding sont indéniables ! on charge directement les données dans un objet d'un langage et ses API qu'on a bien plus l'habitude de manipuler qu'un parser que l'on programme avec SAX, StAX ou XmlReader, ou même que DOM. En fait la philosophie est différente. On programme un parser particulier quand on souhaite, à la volée, obtenir certaines informations d'un document XML. On utilise DOM lorsqu'on souhaite avoir une répresentation de l'ensemble des données sous la forme d'un arbre d'instance et naviguer dedans particulièrement en utilisant XPath. On utilise le Data Binding lorsqu'on souhaite une représentation objet des données. Les objets qui contiennent les données exploitent idéalement les API, en particulier implémentent des listes (`List<...>`). Ils permettent aussi d'"augmenter" ces données d'une part par des restrictions d'accès ou de modification (ex: immuabilité), d'autre part en ajoutant des méthodes apportant d' l'"intelligence" à ces données.

Les **inconvénients** sont aussi nombreux ! Dans les classes ...

- l'encodage disparaît
- les commentaires disparaissent
- les restrictions disparaissent pour la plupart, les restrictions complexes en tout cas (ex: les patterns ou des valeurs comprises entre un max et un min)
- les contraintes d'existence et d'unicité disparaissent
- les sections CDATA ne sont pas préservées
- les contenus mixtes correspondent mal
- ...

6 Exemples qui seront traités dans les section suivantes

Considérons ces 2 exemples de documents XML et leurs classes correspondantes en C# :

Un vecteur et un rectangle Nous avons vu que certains objets ne contiennent que des attributs (par exemple un objet `Vector2` qui contient 2 valeurs `X` et `Y` (voir Figs. XIV .3 et XIV .4)).

```xml
<?xml version="1.0"?>

<schema version="1.0"
        targetNamespace="http://myGame/models"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:gm="http://myGame/models"
        elementFormDefault="qualified">

    <complexType name="Vector2">
        <sequence>
            <element name="X" type="double"/>
            <element name="Y" type="double"/>
        </sequence>
    </complexType>

</schema>
```

Figure XIV .3: Le même vecteur que dans le listing XIV .4 modélisé sous la forme d'un schéma XML.

et une implémentation possible en Java ou C# :

```java
public class Vector2 {

    public double X;
    public double Y;

    public Vector2() {
        this.X = 0;
        this.Y = 0;
    }
}
```

Figure XIV .4: Un exemple d'objet "donnée" : une classe Java/C# définissant un vecteur 2D et son constructeur par défaut.

Rmq: le type `Vector2` est déjà implémenté dans l'API C#.

D'autres sont plus complexe et peuvent impliquer des calculs, par exemple ceux de l'aire d'un rectangle comme le montrent les listings suivants.

Le schéma :

```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>
```

l'instance :

```
<?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>
```

et une implémentation possible en C# :

```
public record struct Rectangle {

    public Rectangle(double x, double y, double width, double height) {
        X = x;
        Y = y;
        Width = width;
        Height = height;
    }

    public Rectangle() {
        X = 0;
        Y = 0;
        _width = null;
        _height = null;
    }

    public double X { get; set; }

    public double Y { get; set; }

    public double Width {
        get => _width??0;
        set {
            if (value <= 0) throw new System.Exception("Valeur invalide");
            _width = value;
        }
    }

    private double? _width;

    public double Height {
        get => _height??0;
        set {
            if (value <= 0) throw new System.Exception("Valeur invalide");
            _height = value;
        }
    }

```

```
39      private double? _height;
40
41      public double Area {
42          get => (_width * _height)??0;
43      }
44
45  }
```

Une creature (d'un jeu) Cette notion d'objet "de données" peut être étendue à la notion d'objet plus complexe contenant à la fois des attributs (par exemple la masse, et la position d'une créature) et des méthodes donnant un comportement à ces objets (par exemple, une méthode pour manger, qui fait accroitre la masse, ainsi qu'une méthode pour se déplacer qui modifie sa position). C'est ce qui est montré dans le listing **??**.

Ainsi, dans une classe Java, on va pouvoir stocker un ensemble de données. Comme ces données sont dynamiques (i.e. modifiables, calculables), on va pouvoir ajouter de *l'intelligence* aux instances, c'est-à-dire qu'elles vont pouvoir gérer/modifier leurs données. En programmation orientée objet, on regroupe habituellement les données et les actions possibles dans une classe. c'est en tout cas le paradigme objet enseigné dans les chapitres précédents.

Si l'on considère le schéma d'une créature évoluant dans un environnement :

```
1   <?xml version="1.0"?>
2
3   <schema version="1.0"
4           targetNamespace="http://myGame/models"
5           xmlns="http://www.w3.org/2001/XMLSchema"
6           xmlns:gm="http://myGame/models"
7           elementFormDefault="qualified">
8
9       <include schemaLocation="Vector2.xsd"/>
10
11      <element name="creature" type="gm:CreatureData"/>
12
13      <complexType name="CreatureData">
14          <sequence>
15              <element name="pos" type="gm:Vector2"/>
16              <element name="boundingBox" type="gm:Vector2" minOccurs="4"
                    maxOccurs="4"/>
17              <element name="mass" type="double"/>
18          </sequence>
19      </complexType>
20
21  </schema>
```

Figure XIV .5: L'objet `CreatureData` de le listing XIV .6 modélisé sous la forme d'un schéma XML utilisant le schéma de `Vector2` montré listing XIV .3.

On peut imaginer une représentation possible en C# :

```
1   public class Creature {
2       public Vector2 Pos {
3           get => Pos;
4           set {  if (value.X>=0 && value.X<Environment._Instance.getWidth()
5                                 && value.Y>=0
6                                 && value.Y<Environment._Instance.getHeight())
7               Pos = value;}
8       }
9
10      private Vector2[] _boundingBox;
11
```

```
    public int Mass {
        get => Mass;
        set { if (value<9) Mass = 9;}
    }

    public int SmallerMass { get; init; }

    Creature() : this(9) {}

    Creature(int mass) {
        Mass = mass;
        SmallerMass = Mass;
        _boundingBox = new Vector2[4];
    }

    //- Getters and setters -
    public Vector2[] getBoundingBoxes() {
        int c = (int) Math.Sqrt(Mass);
        _boundingBox[0].X = Pos.X - c;
        _boundingBox[0].Y = Pos.Y - c;
        _boundingBox[1].X = Pos.X - c;
        _boundingBox[1].Y = Pos.Y + c;
        _boundingBox[2].X = Pos.X + c;
        _boundingBox[2].Y = Pos.Y + c;
        _boundingBox[3].X = Pos.X + c;
        _boundingBox[3].Y = Pos.Y - c;
        return _boundingBox;
    }

    // - Behaviours -
    public void eat() {
        Mass += 1; // add one unit of mass each time the creature eats
    }

    public void walkLeft() {
        if (Pos.Y == 0) // walk on the floor only
            Pos = Pos with { X = Pos.X - 1 }; // move by one unit to the left
    }

    // all other behaviours such as walkRight, fly, ...
}
```

avec l'environnement correspondant :

```
public class Environment {
    // To make a singleton
    private static Environment _instance;

    // Proterty - get access to "this"
    public static Environment _Instance {
        get {
            if (_instance == null) {
                XMLManager<Environment> _xmlEnvironment = new XMLManager<
                    Environment>();
                _instance = _xmlEnvironment.Load("XML/Environment.xml");
            }
            return _instance;
        }
    }

    [XmlElement("dimensions")] public Vector2 Dimensions { get; init; }

    public Environment() {
        Dimensions = new Vector2(800,600);
```

```
    }

    public Environment(int width, int height) {
        if (width < 1) width = 1;
        if (height < 1) height = 1;
        Dimensions = new Vector2(width, height);
    }

    public int getWidth() { return (int) Dimensions.X; }
    public int getHeight() { return (int) Dimensions.Y; }
}
```

L'environnement, dont le schema XML est donné ci-dessous, est un singleton : il est instancié une seule fois comme une instance statique. Il est sérialisable. Sa sérialisation utilise une classe `XMLManager` que nous verrons plus tard.

```
<?xml version="1.0"?>
<schema version="1.0"
        targetNamespace="http://myGame/models"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:gm="http://myGame/models"
        elementFormDefault="qualified">

    <include schemaLocation="Vector2.xsd"/>

    <element name="environment" type="gm:Environment"/>

    <complexType name="Environment">
        <sequence>
            <element name="dimensions" type="gm:Vector2"/>
        </sequence>
    </complexType>

</schema>
```

et en voici une instance possible :

```
<?xml version="1.0" encoding="utf-8"?>
<environment
        xmlns="http://myGame/models"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://myGame/models Environment.xsd">
    <dimensions>
        <X>800</X>
        <Y>480</Y>
    </dimensions>
</environment>
```

Obtenir l'équivalent exact de la classe `Creature` sous la forme d'un schéma XML n'est a priori pas évident. Nous n'avons "aucun moyen simple" de signifier en XML la présence de méthodes, notamment celles qui donnent un comportement à la créature, dans notre schéma XML. Le XML est fait pour ne contenir que des données, pas des comportements !

Une façon de résoudre cela du point de vue POO, c'est de considérer cette fois un ensemble formé par deux classes, l'une contenant les données uniquement, `CreatureData`, l'autre gérant en plus les comportements `CreatureBehaviours`. Dans cet ensemble, on peut utiliser `CreatureData` comme classe mère de `CreatureBehaviours` (héritage), ou bien faire une composition de `CreatureData` dans `CreatureBehaviours` (composition) ; Voir les listings ci-dessous montrant la composition.

```cs
public class CreatureData {
    public Vector2 Pos {
        get => Pos;
        set {  if (value.X>=0 &&
            value.X<Environment.
            _Instance.getWidth()
                                &&
                                    value
                                    .Y
                                    >=0

                                &&
                                    value
                                    .Y
                                    <
                                    Environment
                                    .
                                    _Instance
                                    .
                                    getHeight
                                    ()
                                    )
            Pos = value;}
    }


    public int SmallerMass { get;
        init; }

    CreatureData() : this(9) {}

    CreatureData(int mass) {
        Mass = mass;
        SmallerMass = Mass;
    }
}
```

```cs
public class CreatureBehaviours {

    private CreatureData _c;
    private Vector2[] _boundingBox;

    CreatureBehaviours(CreatureData
        c) {
        this._c = c;
    }

    public void ChangeCreature(
        CreatureData c) {
        this._c = c;
    }

    // - Behaviours -
    public void Eat() {
        _c.Mass += 1;
    }

    public void WalkLeft() {
        if (_c.Pos.Y == 0) // walk
            on the floor only
            _c.Pos = _c.Pos with {
                X = _c.Pos.X - 1 };
                // move by one unit
                to the left
    }

    //- Getters and setters -
    public Vector2[]
        GetBoundingBoxes() {
        // la taille de la créature
            croit comme la
        // racine carrée de sa
            masse
        int c = (int) Math.Sqrt(_c.
            Mass);
        _boundingBox[0].X = _c.Pos.
            X - c;
        _boundingBox[0].Y = _c.Pos.
            Y - c;
        _boundingBox[1].X = _c.Pos.
            X - c;
        _boundingBox[1].Y = _c.Pos.
            Y + c;
        _boundingBox[2].X = _c.Pos.
            X + c;
        _boundingBox[2].Y = _c.Pos.
            Y + c;
        _boundingBox[3].X = _c.Pos.
            X + c;
        _boundingBox[3].Y = _c.Pos.
            Y - c;
        return _boundingBox;
    }

    // all other behaviours such as
        walkRight, fly, ...
}
```

Figure XIV .6: Cette fois, la créature est modélisée sous la forme de 2 classes : une pour contenir ses données `CreatureData`, l'autre pour gérer ses comportements `CreatureBehaviours`.

Une collection de valeurs - exemple du polygone Les objets contiennent souvent des collections de valeurs, soit des valeurs de type primitives (tableaux d'entiers par exemple), soit des valeurs de type complexes (listes de créatures ...).

La définition d'un polygone contenant une collection de coordonnées (sommets) `(x,y)` correspond à ce cas :

```
<?xml version="1.0"?>
<xs:schema version="1.0"
        targetNamespace="http://myGame/models"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:pl="http://myGame/models"
        elementFormDefault="qualified">

    <xs:include schemaLocation="Vector2.xsd"/>

    <xs:element name="polygone" type="pl:Polygone">
        <xs:unique name="idUnique">
            <xs:selector xpath="pl:sommet"/>
            <xs:field xpath="@id"/>
        </xs:unique>
    </xs:element>

    <xs:complexType name="Polygone">
        <xs:sequence>
            <xs:element name="sommet" type="pl:Vector2" minOccurs="3" maxOccurs
                ="unbounded"/>
        </xs:sequence>
        <xs:attribute name="nom" type="xs:string"/>
    </xs:complexType>

</xs:schema>
```

et une instance :

```
<?xml version="1.0" encoding="UTF-8"?>
<pl:polygone
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xmlns:pl='http://myGame/models'
        xsi:schemaLocation='http://myGame/models polygone.xsd'
        nom="star">
    <pl:sommet>
        <pl:X>100</pl:X>
        <pl:Y>10</pl:Y>
    </pl:sommet>
    <pl:sommet>
        <pl:X>40</pl:X>
        <pl:Y>198</pl:Y>
    </pl:sommet>
    <pl:sommet>
        <pl:X>190</pl:X>
        <pl:Y>78</pl:Y>
    </pl:sommet>
    <pl:sommet>
        <pl:X>10</pl:X>
        <pl:Y>78</pl:Y>
    </pl:sommet>
    <pl:sommet>
        <pl:X>160</pl:X>
        <pl:Y>198</pl:Y>
    </pl:sommet>
</pl:polygone>
```

Une classe correspondante pourrait être :

```
public class Polygone {

    public string _Nom { init; get; }
    public List<Vector2> _sommets;

    public override string ToString() {
        string s = "";
        foreach (var sommet in _sommets) {
            s += "(" + sommet.X + "," + sommet.Y + ")\n";
        }
        return s;
    }

    public Polygone(string nom) {
        _Nom = nom;
        _sommets = new List<Vector2>();
    }
}
```

7 Data Binding et Serialisation en C#

7.1 Compilation du XML Schema

En C#, concernant la génération automatique de classes, le choix est assez limité.

- Sous windows c'est simple, vous pouvez utiliser l'executable `XSD.exe`. Il existe aussi Dingo et XXsd2Code comme alternatives.
- Sous les autres plates-formes, ... il n'y a rien ! rien du tout ! Vous savez ce qu'il vous reste à faire si vous voulez générer des classes C# sur la base de schémas XML : réviser votre cours sur les parsers et/ou votre cours sur XSLT et passer du temps à programmer une lib !

Donc comment on fait sinon ? Eh bien, à la main ! Mais la bonne nouvelle c'est que ce n'est pas très compliqué en C#.

7.2 Design des classes pour la sérialisation

En C#, pour qu'une classe soit sérialisable, il est nécessaire de préciser [`Serializable`] devant sa déclaration. De plus, si cette classe constitue le type qui correspond à la racine du document qu'on souhaite sérialiser, on précisera avec un tag [`XmlRoot("nomRacineXML", Namespace="URI_NS")`], par exemple :

```
[XmlRoot("polygone", Namespace = "http://myGame/models")]
[Serializable]
public class Polygone { /* ... */ }
```

Seuls les champs (variables membres et propriétés) publics sont sérialisables.
Il est de plus possible de spécifier devant chaque champ s'il sera sérialisé ou ignoré (avec [`XmlIgnore`]), et si oui son nom (en tant qu'élément ou attribut) dans le document XML, à l'aide des tags [`XmlElement("nomElementXML")`] et [`XmlAttribute("nomAttributXML")`], par exemple :

```
[XmlRoot("polygone", Namespace = "http://myGame/models")]
[Serializable]
public class Polygone {
    [XmlAttribute("nom")] public string _Nom { init; get; }
    [XmlElement("sommet")] public List<Vector2> _sommets;

    public override string ToString() {
        string s = "";
        foreach (var sommet in _sommets) {
            s += "(" + sommet.X + "," + sommet.Y + ")\n";
        }
```

```
        return s;
    }

    public Polygone(string nom) {
        _Nom = nom;
        _sommets = new List<Vector2>();
    }

    public Polygone() {
        _Nom = String.Empty;
        _sommets = new List<Vector2>();
    }
}
```

Cette classe sérialisable correspond au Schema XML présenté plus haut. On remarquera comment est prise en charge la séquence de sommets. On déclare une liste de Vector2 (`List<Vector2>`) mais on précise que les items de cette séquence dans le fichier XML se nomment `sommet` à l'aide de [`XmlElement("sommet")`]. Notez qu'on aurait pu tout aussi bien remplacer la liste par un tableau : `Vector2[]`
On remarque également dans le Schema XML que l'attribut `nom` est optionnel (il n'est pas `required`). Pourtant la classe comporte une propriété membre `_Nom`. On peut se poser la question de si l'absence de l'attribut `nom` dans certaines instances de documents XML pourrait poser problème. En réalité non ! La désérialisation des membres d'une classe n'est pas obligatoire. Il faut juste penser à bien initialiser le membre dans le constructeur de la classe, au cas où celui-ci n'était pas instancié et initialisé lors de la désérialisation.

Exemple de la créature et de l'environnement Voici ce que deviendrait la classe `Creature` pour qu'elle devienne sérialisable :

```
using System.Numerics;
using System.Xml.Serialization;

namespace SerializationExemple;

[XmlRoot("creature", Namespace = "http://myGame/models")]
[Serializable]
public class Creature {
    [XmlElement("pos")] public Vector2 _Pos {
        get => _Pos;
        set {  if (value.X>=0 && value.X<Environment._Instance.getWidth()
                              && value.Y>=0
                              && value.Y<Environment._Instance.getHeight())
            _Pos = value;}
    }

    private Vector2[] _boundingBox;

    [XmlElement("mass")] public int _Mass {
        get => _Mass;
        set { if (value<9) _Mass = 9;}
    }

    [XmlIgnore] public int _SmallerMass { get; init; }

    Creature() : this(9) {}

    Creature(int mass) {
        Mass = mass;
        SmallerMass = Mass;
        _boundingBox = new Vector2[4];
    }

    //- Getters and setters -
```

```
35      public Vector2[] getBoundingBoxes() {
36          int c = (int) Math.Sqrt(Mass);
37          _boundingBox[0].X = Pos.X - c;
38          _boundingBox[0].Y = Pos.Y - c;
39          _boundingBox[1].X = Pos.X - c;
40          _boundingBox[1].Y = Pos.Y + c;
41          _boundingBox[2].X = Pos.X + c;
42          _boundingBox[2].Y = Pos.Y + c;
43          _boundingBox[3].X = Pos.X + c;
44          _boundingBox[3].Y = Pos.Y - c;
45          return _boundingBox;
46      }
47
48      // - Behaviours -
49      public void eat() {
50          Mass += 1; // add one unit of mass each time the creature eats
51      }
52
53      public void walkLeft() {
54          if (Pos.Y == 0) // walk on the floor only
55              Pos = Pos with { X = Pos.X - 1 }; // move by one unit to the left
56      }
57
58      // all other behaviours such as walkRight, fly, ...
59  }
```

On voit dans cet exemple que

- le champ `_boundingBox : Vector2[]` ne sera pas sérialisé car privé.
- les propriétés comme les variables membres peuvent être indifféremment sérialisées : ici, `_Pos : Vector2` et `_Mass : int` sont des propriétés.
- le champ `_SmallerMass : int` (une propriété) ne sera pas sérialisé car taggé `[XmlIgnore]`

Remarquez que la classe singleton `Environnement` présentée plus haut était déjà serialisable.

On remarquera également qu'on peut indifféremment sérialiser les classes `Creature` ou `CreatureData` ; autrement dit, et fort heureusement, une classe n'a pas besoin d'être un objet de données pur pour pouvoir être sérialisé ; il peut ainsi contenir des méthodes. Il faut noter cependant que si l'on utilise un générateur de classes à partir d'un Schema XML (en C# sous Windows avec l'outil `XSD.exe`, sous Java avec le compilateurde Schema `JaxB`), dans ce cas les classes générées ne comportent pas de méthodes. On utilise alors la représentation séparée des objets de données et objets avec méthodes comme présenté avec `CreatureData` et `CreatureBehaviours` (en les combinant soit par héritage, soit par composition).

7.3 Transit des données : *marshalling/sérialisation* et *unmarshalling/désérialisation* en C#

Une fois les classes générées, pour que le DataBinding soit opérationnel, on doit disposer de méthodes de transit des données entre instances de classes et instances de document XML valides. Les opérations de (i) *unmarshalling/désérialisation* et de (ii) *marshalling/sérialisation*, permettent de manipuler ces instances. Elles permettent respectivement (i) d'instancier et d'initialiser un Objet (C#) conforme au schéma XML, et (ii) de sauvegarder les données contenues dans un Objet "donnée" (C#) dans une nouvelle instance de document XML (une instance de document valide, que l'on doit générer sur la base du même XML Schema). Le listing XIV .7 montre comment, en C#, réaliser la (i) *désérialisation (unmarshalling)* et la *sérialisation (marshalling)* entre instances de classe Creature et instances de document XML conformes au schéma de `creature` décrit dans les fichiers `Creature.xsd` et `Vector2.xsd`.

```
public static class CreatureXMLManager {

    // --- unmarshalling - désérialisation ---
    public static Creature Load(string path) {
        Creature _instance; // déclare une instance de créature
        // reads the text (xml) file and store in reader
        using (TextReader reader = new StreamReader(path)) {
            // create an instance of the XmlSerializer
            var xml = new XmlSerializer(typeof(Creature));
            // deserialize and casts the text into the type Creature
            _instance = (Creature)xml.Deserialize(reader);
        }
        // returns the document as a serialized object of type Creature
        return _instance;
    }

    // --- marshalling - sérialisation ---
    public static void Save(string path, object obj) {
        using (TextWriter writer = new StreamWriter(path)) {
            var xml = new XmlSerializer(typeof(Creature));
            xml.Serialize(writer, obj);
        }
    }

    // --- marshalling - sérialisation ---
    public static void Save(string path, object obj, XmlSerializerNamespaces ns
        ) {
        using (TextWriter writer = new StreamWriter(path)) {
            var xml = new XmlSerializer(typeof(Creature));
            xml.Serialize(writer, obj, ns);
        }
    }
}
```

Figure XIV .7: Cette classe contient des méthodes qui assurent la *désérialisation* d'un fichier XML (une instance de document XML) vers un Objet de type Creature, et inversement (sérialisation)

Concernant la **désérialisation** (XML -> Objet en mémoire (côté C#)), le document est lu par un lecteur de texte (`TextReader` instancié comme un `StreamReader`). On crée une instance de sérialiseur XML (une instance de `XmlSerializer`). Le constructeur du `XmlSerializer` requiert le type de l'objet à sérialiser, ici une `Creature`. Ce type d'objet est capable de créer une instance d'objet générique (`Object`) dont l'instance (en mémoire) comprend les champs du type qu'on a passé au constructeur et les a initialisé avec les valeurs lues dans le reader. On caste cet instance d'`Object` selon le type que l'on souhaite désérialiser (ici une `Creature`. La méthode retourne ensuite cette instance.

Concernant la **sérialisation** (Objet en mémoire (côté C#) -> XML): On crée un `TextWriter` instancié comme un `StreamWriter` ici. Comme précédemment, on instancie un `XmlSerializer` dont la construction requiert le type de l'objet à sérialiser (la `Crreature`), puis on couple le writer à l'objet à sérialiser via la méthode `Serialize`. Remarquez que 2 versions sont implémentées ici : une qui ne fournit pas de namespace, et une dans laquelle on peut spécifier un namespace.

Généralisation d'une classe de sérialisation / désérialisation Il est très simple de généraliser l'usage de cette classe et méthodes à n'importe quel type d'objet `T` comme suit :

```
using System.Xml.Serialization;

/* An XMLManager is used to Serialize (save) /Deserialize (load) XML documents
    and Objects.
It's a generic class that must be specialized for a particular object that
    corresponds to the XML document.
```

```
documentation : https://learn.microsoft.com/fr-fr/dotnet/api/system.xml.
    serialization.xmlserializer.deserialize?view=net-8.0
*/
public class XMLManager<T> {
    public T Load(string path) {
        // Declare a generic T variable of the type to be deserialized to store
             the deserialized xml file
        T _instance;
        // reads the text (xml) file and store in reader
        using (TextReader reader = new StreamReader(path)) {
            // create an instance of the XmlSerializer
            var xml = new XmlSerializer(typeof(T));
            // deserialize and casts the text into the type T
            _instance = (T)xml.Deserialize(reader);
        }

        // returns the document as a serialized object of type T
        return _instance;
    }

    public void Save(string path, object obj) {
        using (TextWriter writer = new StreamWriter(path)) {
            var xml = new XmlSerializer(typeof(T));
            xml.Serialize(writer, obj);
        }
    }
    public void Save(string path, object obj, XmlSerializerNamespaces ns) {
        using (TextWriter writer = new StreamWriter(path)) {
            var xml = new XmlSerializer(typeof(T));
            xml.Serialize(writer, obj, ns);
        }
    }

}
```