X-component 4.2

User Manual

Author: **Jindřich Kocman, Václav Trojan**
Version: **4.2.2.26**
Date: **2024-03-31**

Contents

Inventory of changes

Default version X-definition 4.2, build 4.2.0.5

1 Notice

Questions, remarks, and bug reports, please send to: xdef@syntea.cz.
The actual version of X-definition can be downloaded from https://github.com/Syntea/xdef
or https://www.xdefinice.cz/en/download/.
You can also download jar files, documentation, and sources from the Nexus repository manager at:
https://oss.sonatype.org/#nexus-search;gav~org.xdef.

2 Introduction

This document is a user's guide that describes the basic use of the "X-component" technology. It is designed for programmers who need to convert data from XML into Java objects and back. The X-component is built on top of the X-definition and extends its capability of working with XML data. For using the X-component, a basic knowledge of the X-definition is required.

2.1 Terms and abbreviations

X-definition	1. The language used for the description of the structure, content, processing, and construction of XML objects. 2. XML element in the X-definition language.
X-component	Technology for creating Java classes from the X-definitions
X-component	Java class created by the X-component technology.
X-component	Java interface org.xdef.component.X-component implemented by X-components.
Unmarshalling	Populate an object with data from an XML source.
Marshaling	Data conversion of an object into XML. The opposite of unmarshalling.
xd:component	An XML element containing a list of descriptions of X-components in the X-definition
X-position	A description of the location of a model in the set of X-definitions (compiled into XDPool)

3 X-component

The X-component is a Java source code of a class generated according to a particular X-definition model. Each model of data in the X-definition corresponds to one generated X-component. The values of JSON objects. XML attributes, XML elements, and text values in the X-component are accessible by “getters” - methods get***NAME*** and set***NAME***, where “***NAME***" is the name of the item in the X-definition data model. The conversion of the value of an item described in the X-definition and the type of object in Java is given by the preset relations shown in the following table:

X-definition datatype	**X-component**
`anyURI`	`java.net.URI`
`Boolean`	`java.lang.Boolean`
`byte, short, int, long, integer, float, double, and derived types`	`java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.math.BigInteger`
`Char`	`Java.lang.Character`
`float, double`	`java.lang.Float, java.lang.Double`
`decimal, dec`	`java.math.BigDecimal`
`datetime, xdatetime, gDate, gTime, gYear, gMoth, gYearMonth, gMonthDay`	`org.xdef.sys.SDatetime` (also available as `java.util.Calendar, java.util.Date, java.sql.Timestamp`)
`Duration`	`org.xdef.sys.SDuration`
`base64Binary, hexBinary, base64, hex`	`byte[]`
`Enum`	`Java.lang.String` or `Enum` in Java

Currency	java.util.Currency
emailAddr	org.xdef.XDEmailAddr
Gps	org.xdef.sys.GPSPosition
ipAddr	java.net.InetAddress
Price	org.xdef.sys.Price
Telephone	org.xdef.XDTelephone
-------	
other datatypes	java.lang.String

The values of child items are represented by X-component objects. If the maximum number of the quantifier is higher than one, then it is represented by java.util.List< correspondent X-component>. Also, if more than one text value occurs at a position, it will be represented as java.util.List<correspondent datatype>.

All types are objects (not primitive values), so it is possible to test in a program if a value exists in the X-component object (e.g., optional values may not exist). If an element or a value has more occurrences, it creates an array of values (using java.util.List), and instead of a setter, the method add**NAME** is generated. Because there are more possibilities to work with datatypes of DateTime, the getters are generated in more variants: timestampOf***NAME***, calendarOf***NAME***, and dateOf***NAME***.

Note that some of the commands in the X-definition affect the creation of the X-component.

- *Ignore – the X-definition object is ignored and its code is not generated in the X-component.*
- *Forget – the elements are deleted from the memory after processing the document. However, the corresponding X-component is created.*

Note that the X-components implement the interface java.io.Serializable, so it is possible to write them to java.io.ObjectOutput stream and to read them from java.io.ObjectInputStream.

- *Ignore – the X-definition object is ignored and its code is not generated in the X-component.*
- *Forget – the elements are deleted from the memory after processing the document. However, the corresponding X-component is created*

Example

Let's have a master X-definition describing an insurance contract:

```
<Contract
    VIN   = "required int()"
    Date  = "required date()">
  <Owner
    Name        = "required string()"
    CompanyID   = "required num(8)" />
  <Keeper  xd:script = "occurs 1.."
    PersonalID = "required num(10)" />
</Contract>
```

The generated X-component will look as follows (the constructors and the methods of the interface **org.xdef.component.X-component** are not displayed):

```
package user.kocman.manual;

public class Contract implements org.xdef.component.X-component{
  public org.xdef.sys.SDatetime getDate() {return _Date;}
  public java.util.Date dateOfDate() {
    return org.xdef.sys.SDatetime.getDate(_Date);
  }
  public java.sql.Timestamp timestampOfDate() {
    return org.xdef.sys.SDatetime.getTimestamp(_Date);
  }
  public java.util.Calendar calendarOfDate() {
    return org.xdef.sys.SDatetime.getCalendar(_Date);
  }
  public Long getVIN() {return _VIN;}
  public Contract.Owner getOwner() {return _Owner;}
  public java.util.List<Contract.Keeper> listOfKeeper() {return _Keeper;}
  public void setDate(org.xdef.sys.SDatetime x) {_Date = x;}
  public void setDate(java.util.Date x) {
    _Date = x==null ? null : new org.xdef.sys.SDatetime(x);
  }
  public void setDate(java.sql.Timestamp x) {
```

```
    _Date = x==null ? null : new org.xdef.sys.SDatetime(x);
  }
  public void setDate(java.util.Calendar x) {
    _Date = x==null ? null : new org.xdef.sys.SDatetime(x);
  }
  public void setVIN(Long x) {_VIN = x;}
  public void setOwner(Contract.Owner x) {_Owner = x;}
  public void addKeeper(Contract.Keeper x) {
    if (x!=null) _Keeper.add(x);
  }
  private org.xdef.sys.SDatetime _Date;
  private Long _VIN;
  private Contract.Owner _Owner;
  private final java.util.List<Contract.Keeper> _Keeper =
    new java.util.ArrayList<Contract.Keeper>();
  // Constructors and implementation of the interface X-component ...

public static class Owner implements org.xdef.component.X-component{
  public String getICO() {return _ICO;}
  public String getName() {return _Name;}
  public void setICO(String x) {_ICO = x;}
  public void setName(String x) {_Name = x;}
  public String xposOfICO(){return XD_XPos + "/@ICO";}
  public String xposOfName(){return XD_XPos + "/@Name";}
  // Constructors and implementation of the interface X-component ...
}
public static class Keeper implements org.xdef.component.X-component{
  public String getPersonalID() {return _PersonalID;}
  public void setPersonalID (String x) {_PersonalID = x;}
  public String xposOfRC(){return XD_XPos + "/@PersonalID ";}
  private String _PersonalID;
  // Constructors and implementation of the interface X-component ...
}
}
```

3.1 An instance of an X-component

Each X-component implements the Java interface `org.xdef.component.XComponent` that allows conversion from/to the XML data format (note that this interface extends java.io.Serializable, so you can read/write the instances of X-component data to java. Io.ObjectInputStream or java.io.ObjectOutputStream). When it comes to populating the appropriate data from the XML (operation unmarshalling), AN X-component instance can be created when parsing the XML according to the model in the X-definition. X-component can also be created using the constructor and populated with the data by the Java program. From an instance of an X-component, you can create the XML element by calling the method `org.xdef.XComponent.toXml()` (marshaling operation).

3.2 X-position

X-position is a description of the location of a model in the set of X-definitions (compiled to XDPool object). It consists of the name of X-definition, followed by the #, and the name of the model. A model can be followed by the "/" character, and either the name of the nested element "$text" for a text value, or "@", and the name of the attribute. If the name contains a prefix of a namespace, it is written with the prefix, as listed in the X-definition. If a set of descendants of an element contains more nodes with the same name, it is described by the serial number of this item in square brackets (the numbers start from one: "[1]"). The first element is taken if the number is not specified (i.e., "[1]" doesn't have to be specified to be written).

In the following example are various X-positions of individual items:

```
<xd:def name = "Model">
  <A>                           Model#A
    <B                          Model#A/B
      b = "string()" />         Model#A/B/@b
    <C />                       Model#A/C
    <B>                         Model#A/B[2]
      required string();        Model#A/B/$text
    </B>
  </A>
</xd:def>
```

3.3 Commands to generate X-components

The list of commands used for creating an X-component is written in the X-definition as the text of the element <xd:component>. The <xd:component> element may be written in a separate X-definition or it can be part of any other X-definition. Each command in the list is terminated with ";".

3.3.1 Command %class

The command `%class` defines a Java class generated according to a model of an element in the X-definition. The %class keyword is followed by a fully qualified class name and by the keyword `%link,` which specifies the X-position of the model in a set of X-definitions, from which the X-component is generated. If the X-component extends a Java class or it implements a Java interface, then you can specify "`extends SuperClassName implements InterfaceName`" after the name of the Java class, where SuperClassName and InterfaceName must be fully-qualified names. The syntax is the same as in the declaration of a class in the Java language.

Example 1

Source X-definition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
   xd:name="Vehicle"
   xd:root="Vehicle">

    <Vehicle
          VIN  = "required string()"
    />

    <xd:component>
        %class cz.syntea.tutorial.Vehicle
               extends cz.syntea.tutorial.PreVehicle
               implements cz.syntea.tutorial.IVehicle
            %link Vehicle#Vehicle;
    </xd:component>
</xd:def>
```

Generated X-component:

```
package cz.syntea.tutorial;

public class Vehicle extends cz.syntea.tutorial.PreVehicle
  implements cz.syntea.tutorial.IVehicle, org.xdef.component.X-component{
  public String getVIN() {return _VIN;}
  public void setVIN(String x) {_VIN = x;}
  // Constructors and implementation of the interface X-component ...
}
```

3.3.2 Command %bind

Command `%bind` can set a new name of an item in an X-definition (the attribute model, element model, or text node model) and the corresponding names of the getters and setters. The keyword %bind is followed by the name that will be applied instead of the automatically generated one. After the specification of the name, the command continues with the keyword `"%with"` which is followed by the list of X-positions (separated by a comma) to which the statement relates. The same name can be used in many models. The getters and setters will be automatically adjusted to match the newly assigned name (see Example 2). If the generated Java class has an ancestor, you can use the %bind command to bind the getter and setter defined in the ancestor. In this case, a given variable, including getters and setters, will not be generated, and it will use the implementation of these methods in the ancestor.

Example 2

Let's have a truck that extends the vehicle defined in the previous example. The generated X-component does not contain getters and setters of the items that are inherited:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
    xd:name="Truck"
    xd:root="Truck">

    <Truck xd:script = "ref Vehicle#Vehicle"
          MaxWeight = "required int()"
    />
```

```
    <xd:component>
        %bind VIN %with cz.syntea.tutorial.Vehicle %link Truck#Truck/@VIN;
        %class cz.syntea.tutorial.Truck
                extends cz.syntea.tutorial.Vehicle
            %link Truck#Truck;
    </xd:component>
</xd:def>
```

Generated X-component:

```
package cz.syntea.tutorial;
public class Truck extends cz.syntea.tutorial.Vehicle
  implements org.xdef.component.X-component{
  public Long getMaxWeight() {return _MaxWeight;}
  public void setMaxWeight(Long x) {_MaxWeight = x;}
  private Long _MaxWeight; getVIN()
  // The methods getVIN and setVIN are already implemented in the superclass Vehicle!

  // Constructors and implementation of the interface X-component ...
}
```

3.3.3 Command %interface

The command `%interface` is used when the (final) model takes the structure of another (referenced) model, and, where appropriate, it also adds additional attributes, text values, or elements. To make the X-components generated from the models behave like the X-component created from the referenced ones, you can create an interface from the given model. This interface can be added for generating final models (see 3.3.1). The interface command starts with the keyword %interface, which is followed by the fully qualified name of the interface and by the keyword `%link,` followed by an X-position of the model in the project.

Example 3

The personal car shares part of the structure of the Vehicle. This structure was pulled out, and the element Personal links to it.

```
<xd:def  xmlns:xd="http://www.xdef.org/xdef/4.2"
    xd:name="Personal"
    xd:root="Personal">

    <Vehicle
        VIN  = "required string()" />

    <Personal xd:script = "ref Vehicle"
        MaxPersons = "required int()" />

    <xd:component>
        %interface cz.syntea.tutorial.IVehicle %link Personal#Vehicle;
        %class cz.syntea.tutorial.Personal
                implements cz.syntea.tutorial.IVehicle
            %link Personal#Personal;
    </xd:component>

</xd:def>
```

The generated interface declares all the getters and setters of the Vehicle class:

```
package cz.syntea.tutorial;

public interface IVehicle extends org.xdef.component.X-component {
  public String getVIN();
  public void setVIN(String x);
  public String xposOfVIN();
}
```

The X-component Personal implements the interface IVehicle:

```
package cz.syntea.tutorial;

public class Personal implements cz.syntea.tutorial.IVehicle, org.xdef.component.X-component{
  public Long getMaxPersons() {return _MaxPersons;}
  public String getVIN() {return _VIN;}
  public void setMaxPersons(Long x) {_MaxPersons = x;}
  public void setVIN(String x) {_VIN = x;}
  // Constructors and implementation of the interface X-component ...
```

```
}
```

3.3.4 Command %ref

It often happens that the project (XDPool) is generated from many X-definitions. In this case, the X-component is generated from a given X-definition, but XDPool is different (for example, some X-definitions are missing, etc.). It can be used in the already created X-component, and it's necessary to avoid its new generation (for example, if the X-component is located in another Jar file). The reference to the already generated X-component is provided by the command `%ref` with the fully qualified name of the already generated X-component and with the keyword `%link` with an X-position of the model in XDPool.

Example

In the Jar file, which is in the current classpath, there is already the generated X-component **`cz.syntea.test.prexd.Vehicle`** from Example 1. So, when you create a set of new X-components, the already generated X-component is used and it is not generated again.

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
    xd:name="Fleet"
    xd:root="Fleet">

    <Fleet>
        <Personal  xd:script = "ref Vehicle#Vehicle; occurs 0.."/>
        <Van xd:script = "ref Vehicle#Vehicle; occurs 0.."/>
    </Fleet>

    <xd:component>
        %ref cz.syntea.tutorial.Vehicle   %link Vehicle#Vehicle;
        %class cz.syntea.tutorial.Fleet %link Fleet#Fleet;
    </xd:component>
</xd:def>
```

```
package cz.syntea.tutorial;

public class Fleet implements org.xdef.component.X-component{
  public java.util.List<Vehicle> listOfPersonal () {return _Personal;}
  public java.util.List<Vehicle> listOfVan() {return _Van;}
  public void addPersonal (Vehicle x) {
    if (x != null) {
        if (x.xGetXPos() == null)
          x.xInit(this, "Personal", null, "Fleet#Fleet/Personal");
      _Personal.add(x);
    }
  }
  public void addVan(Vehicle x) {
    if (x != null) {
        if (x.xGetXPos() == null)
          x.xInit(this, "Van", null, "Fleet#Fleet/Van");
      _Van.add(x);
    }
  }
 // Constructors and implementation of the interface X-component ...
}
```

3.3.5 Command %enum

If the X-definition data type enum is specified, its value in the X-component is represented by default as a String. However, in case we want to have a choice in the code only from a set of allowed values, it is possible to generate the data value of the enum as a Java enum type. The data type must be defined in the X-script section <xd:declaration>. The enum will be generated by using the command %enum, followed by the fully qualified name of the enum class and the name of the data type.

Example

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" xd:name="Vehicle2" xd:root="Vehicle2">

    <Vehicle2
          VIN   = "required string()"
          Color = "required color()"
    />
```

```
    <xd:declaration>
        type color enum('white', 'blue', 'green');
    </xd:declaration>

    <xd:component>
        %enum cz.syntea.tutorial.Color color;
        %class cz.syntea.tutorial.Vehicle2 %link Vehicle2#Vehicle2;
    </xd:component>
</xd:def>
```

```
package cz.syntea.tutorial;
public enum Color implements org.xdef.component.XCEnumeration {
        white,
        blue,
        green;
        @Override
        public final Object itemValue() {return name();}
        @Override
        public final String toString() {return name();}
        public static final Color toEnum(final Object x) {
                if (x != null)
                        for(Color y: values())
                                if (y.itemValue().toString().equals(x.toString())) return y;
                return null;
        }
}
```

```
package cz.syntea.tutorial;

public class Vehicle2 implements org.xdef.component.X-component{
  public cz.syntea.tutorial.Color getColor() {return _Color;}
  public String getVIN() {return _VIN;}
  public void setColor(cz.syntea.tutorial.Color x) {_Color = x;}
  public void setVIN(String x) {_VIN = x;}
  // Constructors and implementation of the interface X-component ...
}
```

4 X-component created from JSON and other data models

X-components can be generated not only from XLM element models, but also from JSON/ XON, INI, or CSV models. From an instance of these models, the corresponding XON object can be retrieved using the toXon() method.

4.1 X-component created from JSON map

Example of the model of JSON map with named items „a“ and „b“:

```
<xd:def xmlns:xd='http://www.xdef.org/xdef/4.2' root="A">
<xd:json name="A">
  { a: "int();", b: "date();" }
</xd:json>
<xd:component> %class mytests.MyTestXonA %link #A; </xd:component>
</xd:def>
```

Getters and setters of named values in JSON map have the "$" character before the name.

The resulting object is a map, containing the values "a" and "b". In the X-component of MyTestXonA, this object is accessed by the getMap$() method. The getMap$ method will be generated. The values "a" and "g" are accessed by the corresponding getters and setters:

```
package mytests;
public class MyTestXonA implements org.xdef.component.X-component {
...
public java.util.Map<String, Object> getMap$() ...

public Integer get$a() ...
public org.xdef.sys.SDatetime get$b() ...
```

```
public void set$a(Integer x) ...
public void set$b(org.xdef.sys.SDatetime x) ...
...
```

4.2 X-component created from JSON array

The names of getters and setters of items of the JSON array start with „item$“. If more items are described in the array model, then the "_" character and the item serial number are added after "Item$". E.g. getItem$_1, setItem$_2, etc.
Example of the model of a JSON array with 3 items:

```
<xd:def xmlns:xd='http://www.xdef.org/xdef/4.2' root="B">
<xd:json name="B">
  [ "int();", "date();", "* boolean();" ]
</xd:json>
<xd:component> %class mytests.MyTestXonB %link #B; </xd:component>
</xd:def>
```

The object in the model „B“ in the X-definition is now an array of values:

```
<xd:def xmlns:xd='http://www.xdef.org/xdef/4.2' root="B">
<xd:json name="B">
  [ "int();", "date();", "* boolean();" ]
</xd:json>
<xd:component> %class mytests.MyTestXonB %link #B; </xd:component>
</xd:def>
```

The JSON model of the array can be retrieved using the getArray$() method. The individual elements described in the model are accessed using the get$item$, get$item$_1, get$item$_2 getters and the set$item$, set$item$_1, set$item$_2 setters:

```
package mytests;
public class MyTestXonB implements org.xdef.component.X-component {
...
public java.util.List<Object> getArray$() ...

public Integer get$item$() ...
public org.xdef.sys.SDatetime get$item$_1() ...
public Boolean get$item$_2() ...

public void set$item$(Integer x) ...
public void set$item$_1(org.xdef.sys.SDatetime x) ...
public void set$item$_2(Boolean x) ...
...
```

5 How to work with the X-components

5.1 Generation of X-component

To generate X-components from X-definitions, you need the object org.xdef.XDPool (you can create it, for example, by the method org.xdef.XDFactory.compileXD, or you can load the compiled XDPool from a file by the method **org.xdef.XDFactory.readXDPool**, etc.). The X-components are generated using the method **org.xdef.component.GenX-component.genX-component**():

```
XDPool xdPool = XDFactory.compileXD(null, "resources/manual/Vehicle.xdef");
GenX-component.genX-component(
                xdPool,                                          // XDPool
                new File("src/main/java").getAbsolutePath(),  // directory where to generate
                null                                             // character set (default UTF-8)
        );
```

The code above generates the source code of X-components defined in the passed XDPool. After the compilation of the generated Java classes, they are ready for use.

5.2 Create an instance of X-component from XML data (unmarshalling)

XDPool is a fully reentrant object; it can be saved as a static final variable that is statically initiated, and therefore it is accessible to all programs that work with it.

Example of creating an X-component from XML:

```
public static final XDPool XD_POOL = XDFactory.compileXD(null, "/manual/Vehicle.xdef");
...

ArrayReporter reporter = new ArrayReporter();
File source = new File("src/main/resources/manual/Vehicle.xml");
...

// 1. Create XDDocument
XDDocument doc = XD_POOL.createXDDocument("Vehicle");
// 2. Create an X-component with the method parseX-component().
X-component xc = doc.parseX-component(source, Vehicle.class, reporter);
// 3. The method parseX-component returns the X-component created according to the command %class.
//    We can cast it, therefore, to the class defined by the %class command and use
//    the getters and setters: getMyObjekt(), ... setMyObjekt(...) ...
Vehicle vehicle = (Vehicle) xc;
```

5.3 Create XML from X-component (marshaling)

From an X-component, it is possible to create an XML element that corresponds to the values in the X-component. To do this, use the method toXml():

```
Element el = xc.toXml();
```

5.4 Transformation to another X-component

X-components allow you to create a new X-component with a different structure (a variant of the construction mode) from a particular X-component.

Example

```
X-component oldxc;                                        // original X-component
...
XDPool xp;                                               // given XDPool
X-component newxc = X-componentUtil.toX-component(oldc, // new X-component
    xp,                                                  // XDPool
 "def#model");                                          // X-position of the model in the new X-component
```

6 Creating and working with X components

Let us have X-definition, which describes a town in which there are streets and houses with tenants (the file **src/data/Town.xdef**):

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
    xd:name="Town"
    xd:root="Town " >

    <Town Name = "required string()">
        <Street xd:script="occurs 0..;"
                Name = "required string()" >
            <House xd:script="occurs 0..; ref House" />
        </Street>
    </Town>

    <House Num     = "required int()"
           Address = "optional string()">
        <Person xd:script="occurs 0..; ref Person" />
    </House>

    <Person FirstName = "required string()"
            LastName  = "required string()" />

</xd:def>
```

XML data is in the file "**src/data/Town.xml**" (processed by X-definition "Town"):

```
<Town Name="Nonehill">
  <Street Name="Long">
    <House Num="1">
      <Person FirstName="John" LastName="Smith"></Person>
      <Person FirstName="Jane" LastName="Smith"></Person>
    </House>
    <House Num="2"/>
    <House Num="3">
      <Person FirstName="James" LastName="Smith"></Person>
    </House>
  </Street>
  <Street Name="Short">
    <House Num="1">
      <Person FirstName="Jeremy" LastName="Smith"></Person>
    </House>
  </Street>
</Town>
```

The X-definition in the XML file "**src/Town_XC.xdef**" describes the generation of X-components and the interface "Citizen":

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
   xd:name="Town-XC">

    <xd:component>
        /*******************************************************
        *   X-components generated from the X-definition Town  *
        *******************************************************/
        %class town.Town %link Town#Town;
        %class town.House %link Town#House;

        /* Create the interface according to the X-component from Town#Person */
        %interface town.Citizen %link Town#Person;
        /* The X-component has the specified interface */
        %class town.Tenant implements town.Citizen %link Town#Person;
    </xd:component>
</xd:def>
```

Let's show how to create a file in which the XDPool with the compiled X-definitions will be saved. Then we'll generate the X-components from it:

```
public class Example_XC {
    public static void main(String[] args) throws Exception {
       // 1. Compile X-definitions
```

```
        XDPool xdPool = XDFactory.compileXD(null,       // null -> use System properties as default
            "src/data/*.xdef");                         // X-definitions

        // 2. Save XDPool to the file
        xdPool.writeXDPool(new File("resources/XDPool.dat"));

        // 3. Generate X-components
        GenX-component.genX-component(xdPool,   // XDPool
            new File("src").getAbsolutePath(),  // directory where to generate
            null);                              // character set (default UTF-8)
        }
}
```

Now we can write the program that will use the X-component. We first set the static variable XP (using the method getXDPool). From the input file "resources/manual/Town.xml", we create the instance of X-component Town, and then we print its contents. Finally, we add the addresses of houses in the X-component and write them to the file resources/Town_processed. xml:

```
public class Example_XC1 {
    // read compiled XDPool from the file to XP
    public static final XDPool XP;
    static {
        try {
            XD = XDFactory.readXDPool("resources/XDPool.dat");
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws Exception {
        // 1. create XDDocument
        XDDocument xd = XD.createXDDocument("Town");

        // 2. Create an instance of X-component Town (unmarshal) from XML data
        File townFile = new File("src/Example_XC1.xml");
        Town town = (Town) xd.parseX-component(townFile, town.Town.class, null);

        // 3. Print the contents of the X-component Town
        System.out.println("Town " + town.getName());
        for (Town.Street street: town.listOfStreet()) {
            System.out.println("Street " + street.getName() + ":");
            for (House house: street.listOfHouse()) {
                System.out.print("House " + house.getNum() + ". ");
                if (dum.listOfPerson().size() > 0) {
                    System.out.println("Tenants :");
                    for (Person person: house.listOfPerson()){
                        System.out.println(person.getFirstName() + " " + person.getLastName());
                    }
                } else {
                    System.out.println("No tenants in this house");
                }
            }
        }

        // 4. Add to each house address.
        for (Town.Street street: town.listOfStreet()) {
            for (House house: street.listOfHouse()) {
                house.setAddress(town.getName() + ", " + street.getName() + " " + house.getNum());
            }
        }

        // 5. Save the XML document with addresses to the file Town_processed.xml (marshall)
        Element el = town.toXml();
        KXmlUtils.writeXml("resources/Town_processed.xml", el, true, false);
    }

}
```

6.1 X-component utilities

In the org.xdef.componenet.XComponentUtil class, with static methods for working with X-component objects, is implemented.

Table 1 – Types of parameters in parameters are expressed by the following letters:

Type of parameter	Java type
B	boolean
D	org.cdef.sys.SDatetime
I	int
L	Java.util.List
LX	Java.util.list<org.xdef.XDDocument>
O	Java.lang.Object
S	Java.lang.String
XC	org.xfef.component.XComponent
XD	org.xdef.XDDocument
XDC	org.xdef.XDContainer
XM	org.xdef.model.XMElement
XP	org.xdef.XDPool
XPR	org.xdef.XDParseResult
XV	org.xdef.XDValue

Table 2 – Public static methods in the org.xdef.component.XComponentUtil class:

Method name	Description	Result type
add(XC, S, O)	Add value O to the list named S from the X-component XC.	
addText(XC, S, LX, O, I)	Create XComponent from text S and add it to the child list LX.	
addXC(LX, LX)	Add XComponents from the list LX to the list LX.	
addXC(LX, XC)	Add XComponent XC to the list LX.	
containerToJlist(XDC)	Converts XDContainer XDC to a string in JSON array form.	String
canonizeXC(LX)	Update the sequential numbers to the items from the list LX.	
dateToJstring((D)	Convert argument D with date to the ISO8601 format.	String
jlistToList(L)	Create java.util.List with Binary objects from the list with XCVapues	List
listToString(L, B)	Create a source list of items with separators (value of parsed list). If parameter B is true generates a JSON array form (i.e. „[…]“).	String
get(XC, S)	Return value returned by getter S from XComponent XC.	Object
getMap(XC)	Get the value of XComponent as the java.util.Map	Java.util.Map
getVariable(XC, S)	Return value from variable S in XComponent XC.	Object
getx(XC, S)	Invoke the method with the name S from XComponent XC and return the value from the method.	Object
listToJlist(L)	Create a list of JSON/XON objects from the list of binary objects L.	List
listToString()	Create the string with source items with separators (value of parsed list).	String
parseResultToList(XPR)	Create Java.util.List and add to it items from the parsed result XPR.	List
parseResultToList(L, XV)	Add value XV to the list L. If XV is XDContainer, add its items to the list.	
set(XC, S, O)	Invoke setter with name S from XComponent XC with value from argument O1.	
setx(XC, S, O)	Invoke method with name S and parameter O from XComponent XC (typically a setter).	
setVariable(XC, S, O)	Set value O to variable S in XComponent XC.	
toXComponent(XC, XM)	Create XComponent from XComponent XC according to the given model XM.	
toXComponent(XC, XD, S)	Create XComponent from XComponent XC according to the model S in XDocument XD.	XComponent

toXComponent(XC, XP, S)	Create XComponent from XComponent XC according to the model in XDPool described by XPosition S.	XComponent
toXml(XC, XM)	Create an Xml element from XComponent XC according to the model XM.	Element
toXml(XC, XD, S)	Create an Xml element from XComponent XC according to model name S from XDocument XD.	Element
toXml(XC, XP, S)	Create an Xml element from XComponent XC according to XPosition S from XDPool XP.	Element
toXon(XC)	Create XON object from X-component	Object
toXonArra(XC)	Create XON array from XComponent XC.	List<Object>
toXonMap(XC)	Create XON map from XComponent XC.	Map<String, Object>
xmlToJavaName(S)	Convert the XML name S to Java form name.	String
updateXPos(XC)	Update XPositions in the child list of XComponents of the given XComponent XC, starting from 0.	
updateXPos(XC, S, I)	Update XPostitions S in the XComponent XC starting from index I.	
valueToList(XDC, I)	Convert the value of XDContainer to java.util.List as values with the typeid I.	List
valueToList(XPR, I)	Convert the value of XDContainer from ParseResult XPR with XDContainer to java.util.List as values with the typeId I..	List

6.2 Example of transformation

Let's try to define an X-definition describing another XML data containing a list of tenants in the city, and let us use the clause `create` to describe how to create a new transformed file (according to the X-definition "Tenants ") from the input data:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
        xd:name = "Tenants"
        xd:root = "Tenants">

    <Tenants>
        <Tenant xd:script="occurs 0..; create from('//Person');"
              GivenName  = "required string(); create from('@FirstName')"
              FamilyName = "required string(); create from('@LastName')"
              Address    = "required string(); create from('../@Address')" />
    </Tenants>

    <xd:component>
        /*****************************************
        *  X-components of the X-definition Tenants *
        *****************************************/
        %class cz.syntea.tutorial.Tenants %link Tenants#Tenants;

        %bind FirstName %link Tenants#Tenants/Tenant/@GivenName;
        %bind LastName  %link Tenants#Tenants/Tenant/@FamilyName;
    </xd:component>

</xd:def>
```

Transformation of the X-component Town according to the model Tenants by the method **`org.xdef.component.CompomentUtil.toX-component`**(...):

```
public class Example_XC2 {

    public static void main(String[] args) throws Exception {
        // 1. Create XDPool a XDDocument
        XDDocument xd = Example_XC1.XD.createXDDocument("Town");

        // 2. Create the instance of the X-component Town (unmarshal)
        File townFile = new File("resources/Town_processed.xml");
        Town town = (Town) xd.parseX-component(townFile, Town.class, null);

        // 3. Create a transformation to the X-component Tenants
        Tenants tenants =
```

```
            (Tenants) X-componentUtil.toX-component(town, Example1.XD, "Residents#Residents");

        // 4. Save data to the file Residents.xml
        Element el = tenants.toXml();
        KXmlUtils.writeXml("resources/Residents.xml", el, true, false);

        // 5. print the list of tenants
        for (Tenants.Resident x: tenants.listOfResident()) {
            System.out.println(x.getFirstName() + " " + x.getLastName() + "; " + x.getAddress());
        }
    }
}
```

The source code of the examples above is distributed together with the X-definition files and can be downloaded at: https://www.xdefinice.cz/en/download/ in the folder “examples”.