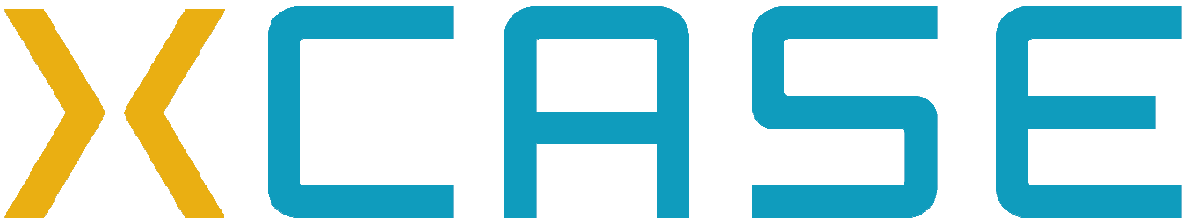


Programmer's Guide

v 1.0

Table of Contents

1 XCase overview - MVC

XCase is based on **Model-View-Controller** (MVC) architecture.

- **Model** stores all the data managed by XCase and raises events when the data is changed (which can be done only through Controllers or loading data from a file)
- **View** binds to Model's events and displays the current state in GUI. This binding is unilateral; View cannot directly access/change the model.
- **Controllers** provide methods for changing the model, which can be used by view and GUI (when processing user input), but doesn't change the View directly.

1.1 Example: Rename a Class

- User clicks on the *Rename* context menu item of a Class, types in a new name and presses Enter
- View checks whether the new name is different from the old one. If it is, it replaces the new name with the old one (because the new name will be set here in a different way and we want to see if everything works OK)
- View calls the Controller with the rename request
- Controller creates a Rename command, initializes it with the Model class and executes it
- The Rename command sets the *Name* property of the Model class and stores the old name for Undo
- The Model class detects the change to its *Name* property and invokes the *PropertyChanged* event
- The View representation of this class receives an update via Binding and updates the text box containing the class name.

2 Model

2.1 Introduction

This component is responsible for storing the semantics of the modeled data. It does not hold any information for a particular visualization of the model. But to support binding of the view, it defines a set of so-called *view helpers* that contain the basic data needed for any visualization (coordinates or dimensions). There will be a more detailed description of the view helpers later in the text.

The main purpose of the model is to provide an easy-to-use implementation of the UML and XSem models to the higher levels of the software (especially view and controller). The UML model constructs are restricted to those needed for UML class diagrams and even there we omitted some structures. The UML model is very general and XCase does not need more than a rather small specialized part of it. Therefore, we did not implement too general constructions as for example Classifier or RedefinableElement. The XSem model will be described in details later in this text but in this place let's say that it is entirely represented in the UML language by use of the UML classes and stereotypes.

As the base of our implementation of the UML model we took an existing open-source library called *nUML* [1] written by Rodolfo Campero. The nUML library is a nearly complete implementation of the UML 2.0 specification and supports import / export from / to XMI language which is a standard format for metadata exchange between different software tools. Despite this, it is still just a bunch of classes without any automatic ownership or objects relations management. Thus, we created a complete set of adapter classes that enrich the library objects by the mentioned relations management and several other capabilities that will be described in the following sections.

The whole Model component is divided into two parts: public interfaces and internal implementation classes. The interfaces use multiple inheritance, since UML also makes use of it in its definition. Internal classes are not visible to other layers. They implement the public interfaces and sometimes extend it by other methods and properties needed mostly for the automatic relations management and for the exposition of the adapted nUML element. The name of the implementation class is the same as the name of the interface but is preceded by an underscore (e.g. **Class** -> **_Class**).

To support binding with other layers all model constructs implements the **INotifyPropertyChanged** interface and so raise a *PropertyChanged* event whenever any property has changed. Also all the collections in the model are instances of the **ObservableCollection** class defined in the **System.Collections.ObjectModel** namespace that raises a *CollectionChanged* event on every interesting collection action (add / remove / move / replace item).

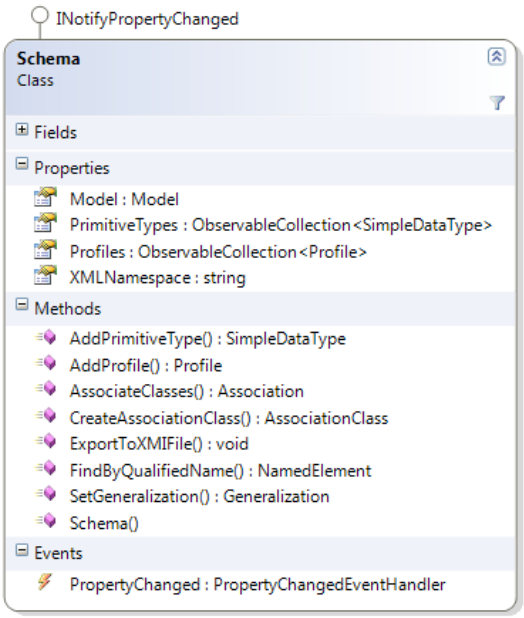
2.2 Two levels of abstraction

When dealing with the model, probably the most problematic part is to clearly distinguish between the metamodel and model levels. Every class defined in the Model library is a part of the metamodel and their instances participate to represent the user model. This distinction is very important especially when working with stereotypes and the type system.

A **Stereotype** is a metaclass describing any stereotype the user can create (or we have created for representing XSem). It defines collections for describing attributes of the stereotype and so on. An instance of this class is one concrete stereotype. The collections are filled with the **Property** instances defining the real attributes that the stereotype have, its name is set etc. So far, it was easy. When we want to apply a stereotype to a concrete model element a new metaclass comes to the scene: a **StereotypeInstance**. It is a distinct interface describing any applied stereotype. It has an attribute of type **Stereotype** referencing the instantiated stereotype (an instance of the **Stereotype** class) and defines a collection for storing the values of the attributes of the stereotype. Its instance describes one concrete instance of a given stereotype. The collection is filled with the instances of **InstantiatedProperty** class that specifies the concrete values of the stereotype attributes.

An analogue situation is in the type system. It is easy to yield to the temptation to use the types existing only in the metamodel with the model attributes et versa. The **DataType** interface is a common base class to all the data types used in the model. The **Property**.*Type* attribute has the **DataType** reference type. And the concrete type of a **Property** instance is an instance of the **DataType** class. Thus, for example, if you want to model an attribute that can reference any construct in the user model, you cannot give it the **Element** type, since it is a metamodel type. Instead you have to define a new class inherited from **DataType** called for example **Object** (or **Element** if you want) and set the **Type** reference of the modeled attribute to its instance. (*Note:* We have already created a primitive type called **object** intended for this purpose, but it was a good example).

2.3 The model interface



To use the model classes you need to create at least one instance of the **Schema** class. This object represents the whole modeled domain including the platform-specific models. New Schema is created with an empty **Model** instance called “User model” that is used for the user data and another prefilled **Model** instance with the name “UML” containing the UML metaclasses to be used for stereotypes (described in more details later in the text).
The UML model is protected but is inserted as a default metamodel reference to each new profile.
The Schema methods are the only way to create new associations, generalizations and association classes in the user model. Also it is the unique owner of all the profiles since they are not a part of the user model.
New schema is created without any profiles or primitive types but XCase uses a project template

file to initiate these collections by the XSem profile and the primitive types used in xml modeling.

2.4 The top-level hierarchy

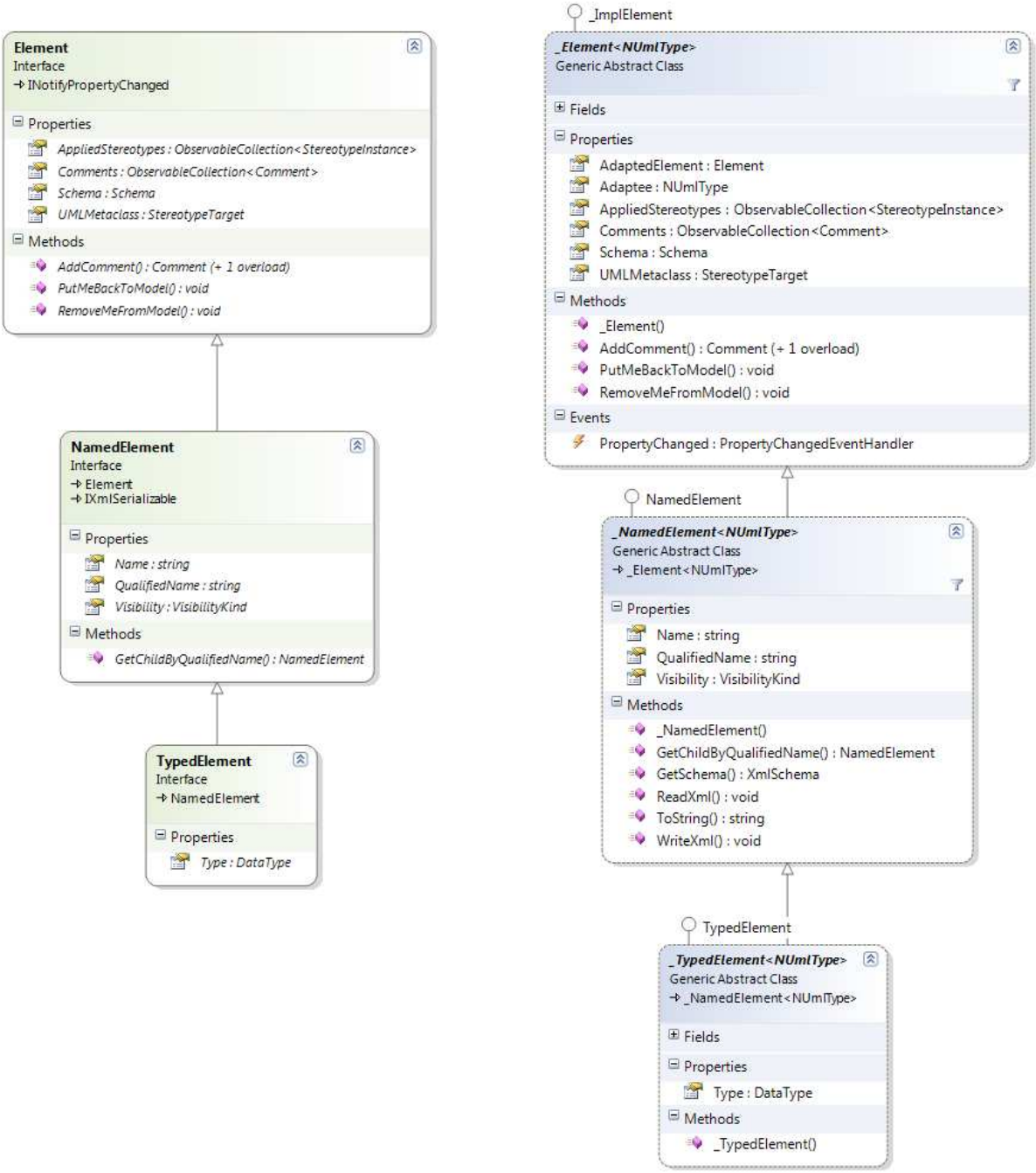


Figure 1 Top-level hierarchy : Interface & Implementation

2.4.1 Element

The topmost interface in the model object hierarchy is the **Element** and all the used UML and XSem constructs inherit directly or indirectly from this interface. It corresponds to the **Element** construct described in the UML specification but it is enriched by several new properties. One is the *AppliedStereotypes* collection that presents an effective way to find all the stereotypes applied to this particular element. This is not possible in the pure UML where the stereotypes have to be found by traversing the extensions instances present in the model.

The second one is the pair of *RemoveMeFromModel* / *PutMeBackToModel* methods that are responsible for correct removal of the element from the model and its return to the model when user undoes the operation that removed it. On the contrary the *OwnedElement* and *Owner* attributes defined in UML are not exposed since the ownership of the elements is solved type-safely between the classes that are concerned.

The implementation class is called **_Element** and extends the **Element** interface by the *Adaptee* property exposing the adapted nUML element. This class is generic and the type parameter is the adapted nUML element type. The type-safety of this property is ensured by the type constraints construction of the C# language.

2.4.2 NamedElement

A direct child of the **Element** interface is the **NamedElement** interface. It is a common base interface of all the constructs that have a name. It extends the **Element** by three properties: *Name* representing a name of the model element relative to its namespace (package). *QualifiedName* representing a name including all the names of the namespaces (packages) on the path from the owner package to the root of the model. The last property is *Visibility* defining the access rights to the element. The interface also provides a method to search for a subordinate element by its qualified name.
NamedElement inherits also from the **System.Xml.Serialization.IXmlSerializable** interface to support the serialization / deserialization of the model to an xml file.

2.4.3 TypedElement

The last interface described in this section is the **TypedElement**. It is a named element that has a type. The type is an instance of a **DataType** interface as described in the chapter 2.2.

2.5 Special interfaces

There is a set of 5 special interfaces defined in the model to ease the use of the model components:

- **IAssociationSource** – Identifies an element that can be a source of an association (i.e. an association can start in this element)
- **IAssociationTarget** – Idenitifies an element that can be a target of an association (i.e. an association can end in this element)
- **IHasAttributes** – Identifies an element that can contain attributes (**Property** instances)
- It defines an attributes collection and the methods for adding a new attribute
- **IHasOperations** – Identifies an element that can contain operations (*Operation* instances)
- It defines an operations collection and the methods for adding a new operation
- **IHasPSMAttributes** – Identifies an element that can contain PSM attributes (**PSMAttribute** instances)
- It defines a PSM attributes collection and the methods for adding a new attribute

2.6 PIM Constructs

The Platform Independent Model is realized by an UML class diagram model. Therefore, most of the PIM constructs have direct equivalents in the UML specification. Therefore, we

will describe only the most important ones. Some of them are extended by new properties mostly to support linking between the PIM and derived PSM constructs. There is a dedicated section describing them.

No component can be created on its own without an owner. When a new project is created, automatically two **Model** instances are created. One is empty and is intended for the user model and the second, invisible to the user, contains the definition of UML metaclasses that can be extended by the stereotypes. There is also one profile created containing the XSem stereotypes definitions. Any new user component can be created uniquely by calling an appropriate *Add* method on the existing model element that will contain it. For example, to add a new class into the model you have to use the *AddClass* method on an existing package that will contain the new class.

Each component has a reference to its owner that is set automatically when the component is inserted to the owner's collection. This reference is read-only for other layers of the software.

2.6.1 Class & PIM Class

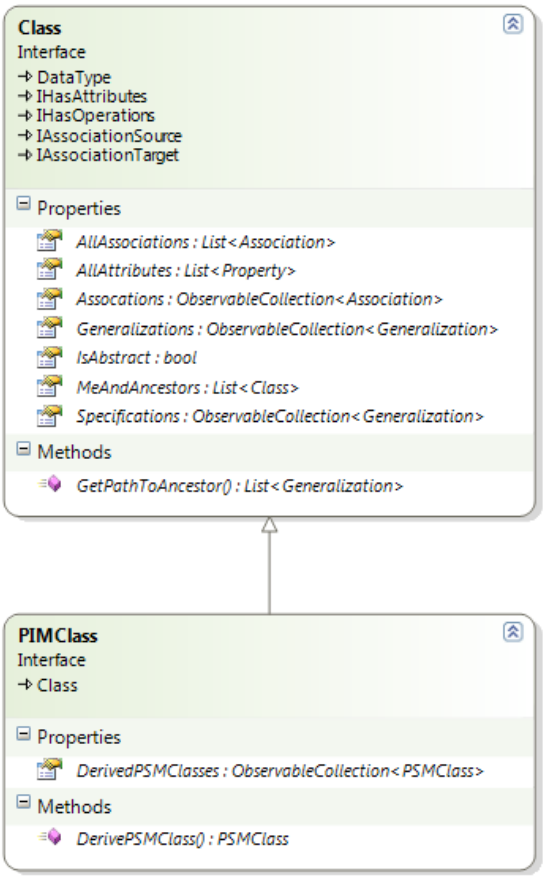


Figure 2 - Class & PIM Class

The **Class** interface corresponds to the UML Class structure. It is a data type that can have attributes, operations and can participate in the associations. The original construct is enriched by the following properties

- *MeAndAncestors* – Collection containing this class and all its ancestors (classes that this class inherits directly or indirectly from)
- *Associations* – Contains all the associations that include this class

- *AllAssociations* – Contains the content of the Associations collection and of the Associations collections of all the ancestors.
- *AllAttributes* – Contains the content of the attributes collection of this class and of all the ancestors of this class.
- *Generalizations* / *Specifications* – Present an effective way to identify all the generalizations that go to or from this class.

All the collections concerning the inheritance are virtual and their content is built on each access, so frequent reading of this property can result in a loss of performance.

The PIM class is a simple extension of the UML class allowing the user to create a new PSM class representing this one and track all the PSM classes that were derived from this class. The content of the *DerivedPSMClasses* is managed automatically. When a derived PSM class is correctly removed from the model (via *RemoveMeFromModel* call) it removes a reference to itself from this collection and puts it back if the *PutMeBackToModel* method is called.

2.6.2 Stereotypes – definition and usage

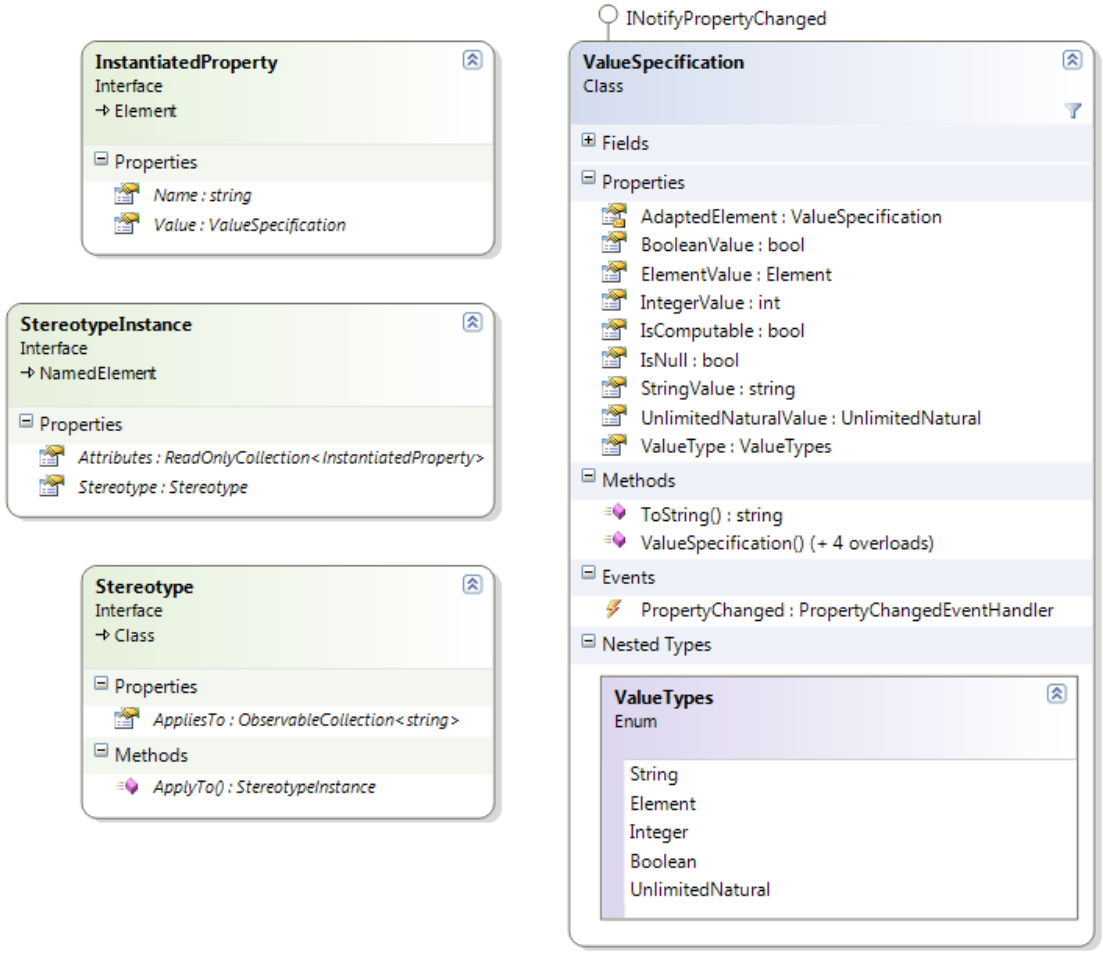


Figure 3 - Stereotypes - most important structures

Stereotypes are special kinds of classes defining how an existing metaclass can be extended. Therefore, an instance of a stereotype is always related to an instance of its related metaclass (or metaclasses since stereotypes can be sometimes applied to more than one element types). Since the stereotypes are special classes, they are also stored in a special type of package called a profile. However, the profiles are very similar to standard packages except that they

can reference one or more metamodels. A metamodel is a standard UML model that contains standard UML classes. But the names of the classes correspond to the metaclasses existing in the used model.

To make it more clear we take an example of the UML model. It contains many familiar metaclasses: Class, Association, Property, etc. The XSem profile in XCase that contains definition of the stereotypes used to represent XSem structures references an instance of the **Model** interface that has the name "UML" and contains instances of the **Class** interface having the *Name* attribute set to "Class", "Association", "Property", etc. When we apply a stereotype to some model element, we extend its definition by the attributes of the stereotype. Before continuing, let's look to an example of the stereotype usage:

In XCase PSM classes of the XSem model are represented as standard UML classes having the **XSem.PSMClass** stereotype. Note that both the PSM class and the UML class are metaclasses and user is working with their instances. The definition of the UML class is described in the previous section, the **PSMClass** stereotype contains attributes as *RepresentedClass* referencing the PIM class that is represented by a concrete PSM class or *Components* collection that contains all the PSM components subordinate to the PSM class. The **PSMClass** stereotype can be applied to the instances of the UML class which is (in UML) represented by an instance of the **Extension** metaclass that relates the stereotype and a class in the UML metamodel corresponding to the UML class. Thus, factually a new metaclass (let's call it a PSM Class) is created that integrates the definition of both the UML class and the **PSMClass** stereotype. When the stereotype is applied to an existing UML class instance, its definition is extended and corresponds to the PSM Class metaclass. Thus, the user gains access to all the attributes defined by the **PSMClass** stereotype and can see / set their values.

Now from the programmer's point of view. The stereotypes are described by the instances of the **Stereotype** interface. As you can see on the figure above, it inherits from the **Class** interface. Thus, its definition is the same as the definition of a standard UML class. It is extended by a collection called *AppliesTo* that contains the names of all the metaclasses that can be extended by this stereotype. The collection contains strings instead of the references to the concrete metaclasses to support the visualization and the serializator that does not serialize the UML metamodel. When a reference is needed a search by the metaclass name is performed in the metamodels referenced by the owner profile. The extensions mechanism mentioned in the previous paragraph is hidden from the other layers of the project and is represented only inside the nUML library.

The *Stereotype* interface also defines a method called *ApplyTo* that takes a reference to an existing element instance. When this method is called it first checks if the element metaclass can be extended by this stereotype and if yes a new instance of the **StereotypeInstance** is created and inserted to the element *AppliedStereotypes* collection. This is analogous to the object oriented programming. You have a class that describes generally the attributes of some entity and you instantiate it to create an object that represents one concrete instance of the entity. The *Stereotype* instance is a description of the attributes of the stereotype. For example in case of the **XSem.PSMStereotype** you know that this stereotype has an attribute called *RepresentedClass* and its type is a reference to a **PIMClass** instance. But to have a reference to a concrete PIM class you have to instantiate the stereotype and this is what the **StereotypeInstance** is for.

The **StereotypeInstance** instance contains instances of the **InstantiatedProperty** interface that defines the value of the attribute. A value is represented by an instance of the **ValueSpecification** interface.

If you want to create your own stereotype, proceed by the following scheme:

- Create a new profile or use an existing one. Profiles are accessible via the **Schema**.*Profiles* collection, to create a new one call **Schema**.*AddProfile.*
- Create a new Stereotype by calling the **Profile**.*AddStereotype* method.
- Set the name of the stereotype and define its attributes by calling **Stereotype**.*AddAttribute* method and setting the attributes of the created **Property** instances appropriately.
- Add the names of the metaclasses that can be extended by the new stereotype to its *AppliesTo* collection (all the extensions are created automatically)
- After that you can simply apply the stereotype using its *ApplyTo* method
-

2.7 PSM Constructs

In this section we list the structures defined in the XSem model used in XCase for the Platform-specific model and some important auxiliary constructs. In contrast to the PIM structures that can be a part of multiple diagrams but exist only once in the model, the existence of every PSM component is connected to the existence of the diagram that it is part of. And no PSM component can be drawn on more than one diagram.

2.7.1 PSMElement

PSMElement interface is the base interface of all the PSM components. It is inherited from the **NamedElement** interface and extends it by the *Diagram* property that references the PSM diagram that the component is part of.

2.7.2 PSMSuperordinateComponent

This is a common base interface of PSM components that can contain other PSM components. It defines a *Components* collection that is an ordered list of the subordinate PSM components and a method called *AddComponent* which adds a new component to the end (or to the specified position) in the components. This method accepts a reference to a factory that creates instances of the concrete **PSMSubordinateComponent** interface child.

2.7.3 PSMSubordinateComponent

This is a common base interface of PSM components that can make a part of some superordinate component content (components collection). Note that a PSM component can be both superordinate and subordinate at the same time. This is a case of for example PSM content choice structure that can exist only in the content of some other PSM component but contains other PSM components on its own.

2.7.4 PSMAssociationChild

This is a common base interface of PSM components that can appear at the child end of a PSM association.

2.7.5 IHasPSMAttributes

This is a common interface of PSM components that can contain PSM attributes. Beside the *PSMAttributes* collection and the methods to add new attributes, it defines a property called *RepresentedClass* that references the PIM class containing the attributes that are represented by the owned PSM attributes.

2.7.6 PSM Class & PSM Structural representative

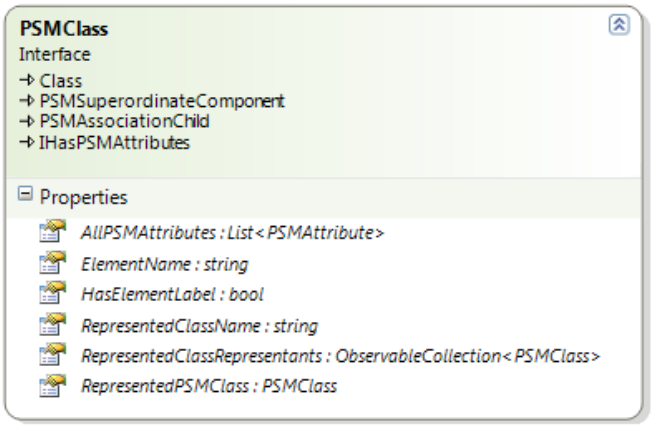


Figure 4 - PSM Class definition

The PSM class inherits from the **Class** interface. Thus, it has all the functionality of the standard UML class, which is especially useful for modeling the inheritance. It can appear at the end of the PSM association and can contain subordinate PSM components. It has a collection of PSM attributes that contains the same items as the *Attributes* collection inherited from the **Class**. The content of both collections is synchronized automatically and you cannot insert an attribute instance that does not implement the **PSMAttribute** interface to the attributes collection (an attempt to do this, results in an **ArgumentException**).

A PSM Class instance can be turned to a PSM Structural Representative instance by setting the *RepresentedPSMClass* attribute to a valid reference. In the background this results in a replacement of the **PSMClass** stereotype by a **PSMStructuralRepresentative** stereotype. Contrarily, a representative can be turned to a PSM class by setting the *RepresentedPSMClass* property to a null value.

The *AllPSMAttributes* collection includes the attributes from the *PSMAttributes* collection of this class and all the subordinate attribute containers (owned directly in the components or indirectly through another subordinate component). This collection is virtual and its content is built on each access to the property.

2.7.7 PSM Attribute

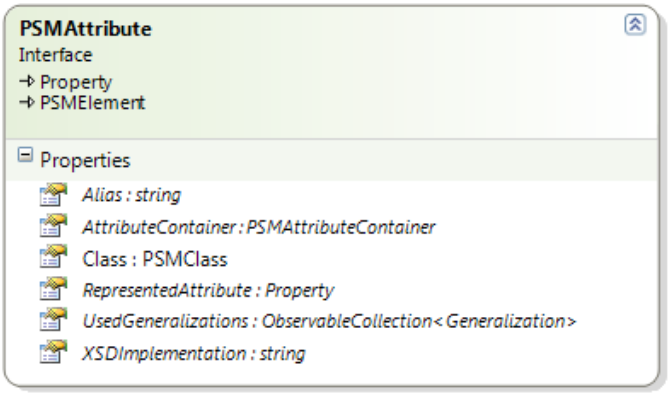


Figure 5 - PSM Attribute definition

A PSM attribute is a standard UML property extended by a reference to the represented PIM attribute, *Alias* attribute that defines the name of this attribute in the generated XML schema and *XSDImplementation* that contains the implementation of the type of this attribute in the generated schema.

2.7.8 PSM Attribute Container

An attribute container can contain some of the attributes of its superordinate PSM class. It is a subordinate component and has PSM attributes.

2.7.9 PSM Content Choice / Content Container

These two components are superordinate and subordinate at the same time. They can only exist in the *Components* collection of some PSM component but typically contains other PSM components.

2.7.10 PSM Association

A PSM association has exactly two ends, a parent and a child end. It is a subordinate component and so it is a part of the parent's content (content of the PSM component on the parent end). The semantics of the association is defined by one or more nesting joins. If the association contains more than one nesting join, the semantics is defined as their union.

2.7.11 PSM Class Union

A class union is a PSM association child and so can appear at the child end of a PSM association. It defines a *Components* collection that can contain any association child, so by now a PSM Class or another class union.

2.7.12 Nesting Join

Nesting join defines the semantics of a PSM association. It is formed mostly by instances of **PIMPath** interface which defines a path through associations in the platform-independent model. It is an ordered list of steps described by a **PIMClass** in which the step starts and a **PIMClass** in which the step ends and an *Association* used to get from the start to the end.

2.8 XSem Stereotypes

As we already mentioned earlier in this text, all the XSem structures are expressed in the standard UML language using the stereotypes. There is an "XSem" profile created upon the project creation that contains all the stereotypes needed for the representation of the XSem constructions.

We present their list in this section:

- **PSMClass** (applies to Class)
  RepresentedClass (object, 1..1) – Reference to the represented PIM class
  ElementName (string, 1..1) – Element label assigned to the PSM class
  Components (object, 0..*) – Ordered collection of references to subordinate PSM components
- **PSMAttribute** (applies to Property)
  RepresentedAttribute (object, 1..1) – Reference to the represented PIM attribute
  Alias (string, 1..1) - Alias of the PSM attribute
- **PSMAttributeContainer** (applies to Class)
  Parent (object, 1..1) – Reference to the component that contains this container
- **PSMClassUnion** (applies to Class)
  Components (object, 0..*) – Ordered collection of references to PSM components in the union
- **PSMContentContainer** (applies to Class)
  Parent (object, 1..1) – Reference to the PSM component that owns this container
  Components (object, 0..*) – Ordered collection of references to the subordinate PSM components
  ElementLabel (string, 1..1) - Name of the modeled XML element
- **PSMContentChoice** (applies to Class)
  Parent (object, 1..1) – Reference to the PSM component that owns this choice
  Components (object, 0..*) – Ordered collection of references to the subordinate PSM components
- **PSMAssociation** (applies to Association)
  NestingJoin (object, 1..*) – Collection of nesting joins defining the semantics of the association
- **PSMStructuralRepresentative** (applies to Class)
  RepresentedClass (object, 1..1) – Reference to the represented PIM class
  RepresentedPSMClass (object, 1..1) – Reference to the represented PSM class
  ElementName (string, 1..1) - element label assigned to the representative
  Components (object, 0..*) – Ordered collection of references to the subordinate PSM components

All the XSem stereotypes as well as the XSem profile are defined in the project template file.

2.9 Linking between the PIM and PSM levels

The model is responsible for keeping the platform-independent model and related platform-specific models consistent. Therefore, the constructs from the models are related from both sides by the references and collections. These references are in most cases handled automatically, but there are several cases in which the model does not have enough information to do this and the outer layers have to manage the binding themselves.

PIM Class has a collection containing all the PSM classes that were derived from it and every PSM class has a reference to the PIM class that it was derived from. This linking is handled

automatically. Changes of some PIM class properties are propagated automatically to the derived PSM classes (for now the Package property). Other can be easily added in the implementation of the **_PSMClass**.*OnRepresentedClassChanged* event handler.

PIM Attribute has a collection of all the PSM attributes that represent it and PSM attributes have reference to the represented PIM attribute. This linking is handled automatically. For a case when the PSM attribute in a class or container represents a PIM attribute that is not owned by the represented PIM class but by some of its ancestors, the PSM Attribute has a collection of all the generalizations (in PIM) that lead from the represented class to the class containing the represented attribute. Accordingly the generalization has a collection of all the PSM attributes that reference it. This linking is also handled automatically.

PIM Association has a collection of all the nesting joins that reference it. This linking is handled automatically.

For case when a PSM association contains a reference to a PIM association that is not directly in the *Associations* collection of the PIM class represented by the PSM class on the parent end, but in some of its ancestors, the PSM association defines the collection referencing all the generalizations (in PIM) that lead from the parent represented class to this ancestor. Likewise, a generalization has a collection of all the PSM associations that reference it. The filling of these two collections is left for to the outer layers since the model cannot simply get the necessary information. The rest of the management (when some of the PSM associations is removed from or put back to the model) is automatic.

2.10 How to add a new construct to the model

When creating a new structure for use in the model, the most problematic part is its representation in the UML. If you need to represent a structure outside the UML definition you have to create a stereotype that will add the requested functionality. The procedure to create a new stereotype is described in the section about the stereotypes.

The new structure should be divided into a public interface and an internal implementation class and should inherit directly or indirectly from the **Element** interface. Constraint the type parameter of the **Element** interface to the real nUML type.

In the constructor of the new element create the adapted nUML object using the appropriate **NUml.Uml2.Create** class method. Bind the properties of your structure to the corresponding properties of the adapted nUML element. For every writable property raise the *PropertyChanged* event when the property value changes, so that the visualization can reflect the change. For collections use the **ObservableCollection** type defined in the **System.Collections.ObjectModel**.
Override the *PutMeBackToModel* / *RemoveMeFromModel* methods.

2.11 Mapping of the model to the UML constructs

Most of the UML constructs in the model have direct mapping to the structures with the same name defined in the UML specification, so we will omit them and will list only the items that differ either by name or by their properties from their UML equivalents. We will certainly list here all the PSM constructions.

Sometimes we have chosen a different name than given in the UML specification, because the original name described a more general structure or because our name seemed clearer to us. Not all the properties have mapping to the UML model. As for example the **PIMClass**.*DerivedPSMClasses* collection. This collection is restored automatically when the project is loaded and presents only redundant information for the model added to find the derived classes effectively. Without this collection we are still able to identify all the derived classes of the specified one but the algorithm is more time complex. This is the case of many collections used to track dependencies between elements in the model (especially between PIM and PSM).

- Adapter name: DataType → Adapted construct Type
- **InstantiatedProperty → Slot**
- **PIMClass → Class**
- **PSMAssociation → Association** with the **XSem.PSMAssociation** stereotype
- **PSMAttribute → Property** with the **XSem.PSMAttribute** stereotype
- **PSMAttributeContainer → Class** with the **XSem.PSMAttributeContainer** stereotype
- **PSMClass → Class** with the **XSem.PSMClass** stereotype
- **PSMClassUnion → Class** with the **XSem.PSMClassUnion** stereotype
- **PSMContentContainer → Class** with the **XSem.PSMContentContainer** stereotype
- **PSMContentChoice → Class** with the **XSem.PSMContentChoice** stereotype
- **SimpleDataType → PrimitiveType**
- **StereotypeInstance → InstanceSpecification**

3 XCase drawing framework

We chose Windows Presentation Foundation for implementation of graphical user interface and drawing both the UML class diagrams and XSEM diagrams. This part covers some of the basic building blocks that we created for drawing diagrams and are part of View.dll assembly, the visual part of the View component of Model-View-Controller pattern (updates of View according to changes in Model are described in 1.1. Diagrams basically consist of *objects* (usually rectangular) and *lines* between these objects. The example diagram on Figure 6 shows both objects and lines.

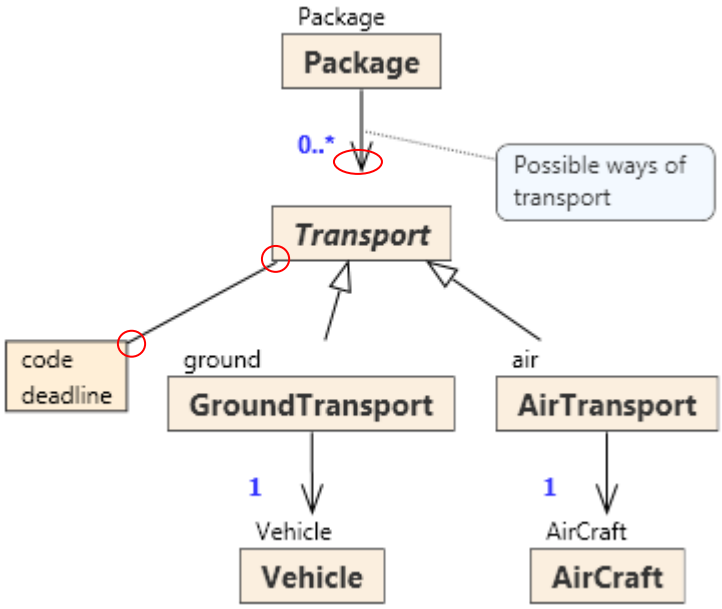


Figure 6 - Example diagram

Examples of *objects* are classes and attribute container and comment, all arrows (associations and generalizations), the component connector between class Transport and the attribute container and line attaching comment to association between classes Package and Transport are examples of *lines*. Endpoints of all lines (in red circles) are objects again (and can be dragged by mouse). The example diagram is a PSM diagram where all layouting is performed automatically, but in PIM diagrams, most of objects can be freely dragged on the canvas (including endpoints of lines that can be dragged around the borders of connected *object*) and lines can be broken to polylines.
All diagrams are created using class **XCaseCanvas**.

3.1 Canvas control – XCaseCanvas

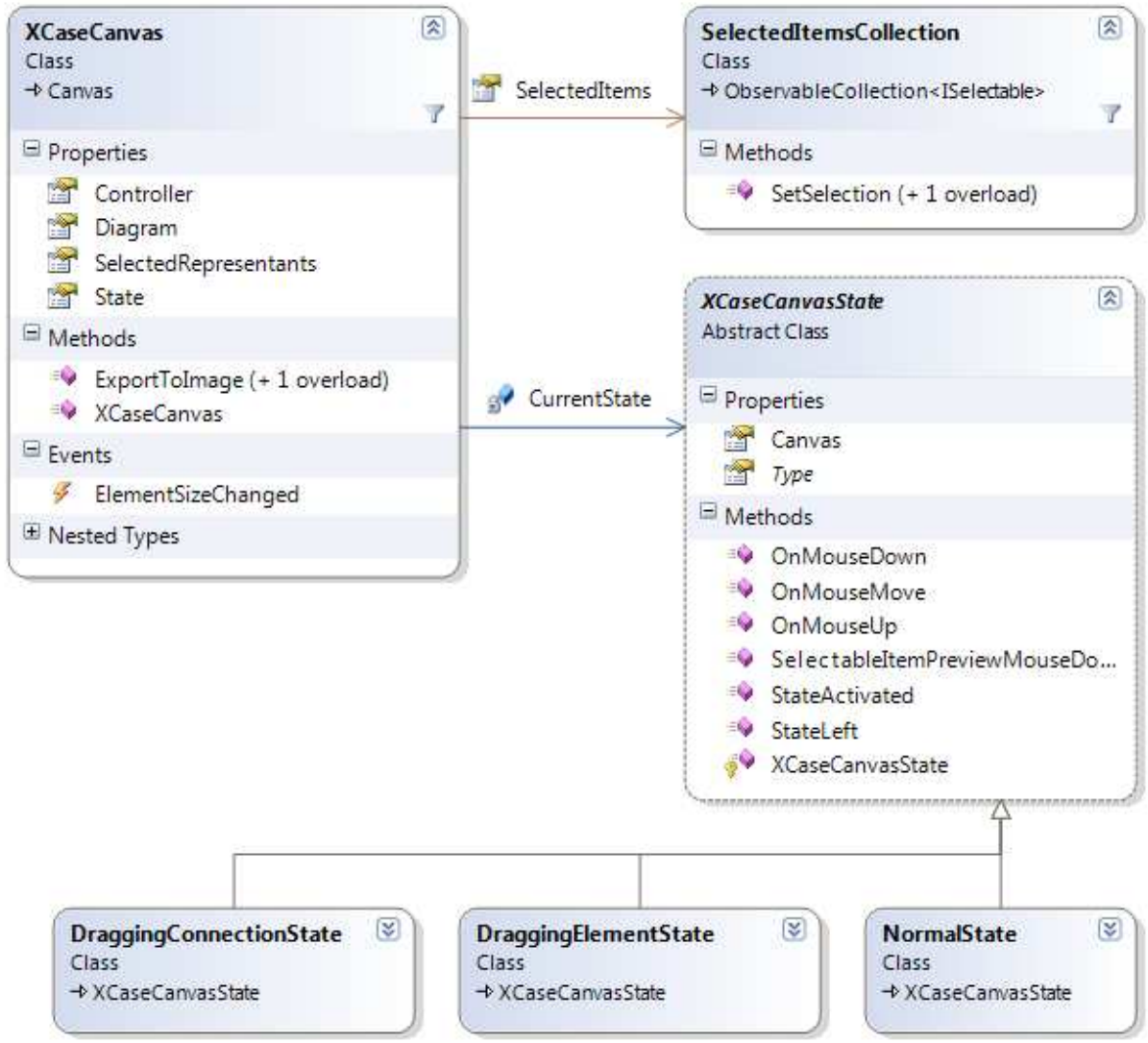


Figure 7

Canvas control is a counterpart of Diagram in View component of Model-View-Component pattern and is responsible for drawing diagrams. It also receives user input from mouse and keyboard. It is responsible for creating representations of elements added to the displayed diagram and removing the representations when the elements are removed from diagram. This part of **XCaseCanvas** functionality is covered in detail in the chapter 4.

Class SelectedItemsCollection stores items selected on the diagram, its method **SetSelection** deletes the current selection and adds items passed as arguments to new selection. This class is used as **SelectedItems** property. The property **SelectedRepresentants** acts as a filter above **SelectedItems** and returns only items of type **IModelElementRepresentant**. See the chapter 4.

XCaseCanvas uses State design pattern [2] to manage mouse input correctly. Canvas can be in three states and in each state mouse events are handled differently. Abstract class **XCaseCanvasState** declares empty operations that are overridden in derived classes. State is changed by assigning desired value to State property.

NormalState is the initial state of **XCaseCanvas**. In this state elements can be selected via mouse and selected elements of type DragThumb (this class is described in its own section).
DraggingElementState must be entered explicitly. In this state a new element is dragged onto canvas (in XCase this state is used when a new class or new comment is dragged from toolbar or navigator window). Canvas returns into **NormalState** immediately after the dragged element is dropped on canvas.
DraggingConnectionState must also be entered explicitly. In this state user can dragged connections between elements implementing **IConnectable**. The user can start dragging by clicking an element and continue by dragging the connection to another element. When the connection is dropped, event handler that processes the event is called and user can start dragging another element. This state is not left automatically (user can drag as many lines as he wants) and must be left explicitly.

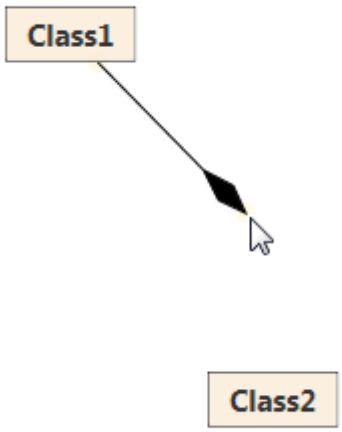
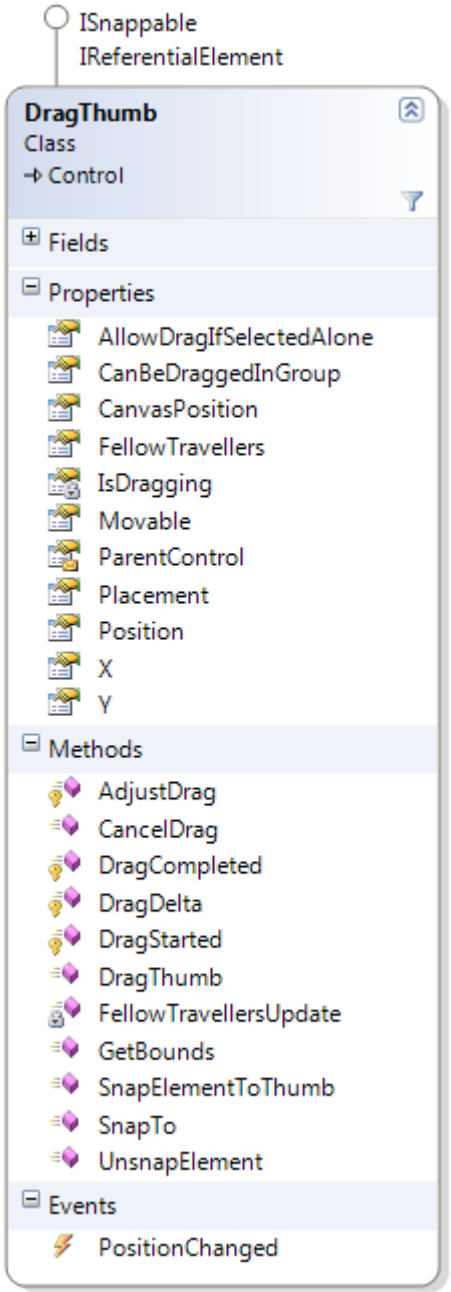


Figure 8 - XCaseCanvas in DraggingConnectionState

3.2 Objects

We created a set of several WPF classes to draw objects on diagram. Each of these objects encapsulates certain functionality. They don't have a visual appearance – this is where we rely on WPF styles and templates. To add visual appearance to a derived class, template is assigned to each class.

3.2.1 DragThumb



DragThumb is a class that can be dragged on the diagram via mouse. It is tightly related to **MoveElementCommand**, which wraps the dragging action. **DragThumb** is an ancestor of virtually all the elements in the diagram. It does not have visual representation itself – this is left to derived classes and their templates.

Properties *Placement*, *X*, *Y*, *Position*, and *CanvasPosition* and event *PositionChanged* are all related to position of the object on canvas.

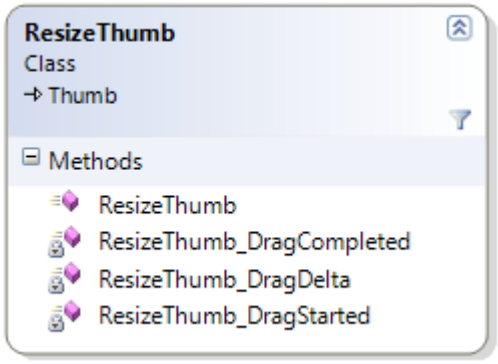
Position returns values of *X* and *Y* as **Point**.

CanvasPosition property is point with coordinates of the object on canvas. The value returned by the property depends on values of *X* and *Y* and also on value of Placement. Placement property describes the way how *CanvasPosition* is computed from X and Y. DragThumb can be placed absolutely on canvas or relatively to another element (snapped to another element – using methods *SnapTo* or *SnapElementToThumb*). When element is snapped to another element, it moves with that element when that element is moved. This way it is for example achieved that comments attached to another objects move along with those objects. Another option is to set Placement to **EPlacement.AbsoluteSubCanvas** - then Position is relative in coordinates of *ParentControl* and *CanvasPosition* is then equal to *ParentControl.CanvasPosition* + *this.Position*. **EPlacement.ParentAutoPos** – with this value the computation of *CanvasPosition* is the same as with **EPlacement.AbsoluteSubCanvas**, but besides that, *ParentControl* is responsible of moving the control. This setting is used for automatically adjusting positions of **EndPoints** of *lines* when the connected *object* is moved. Event *PositionChanged* is invoked every time **DragThumb** is moved.

By setting value of *DragThumb.Movable* to false, dragging of **DragThumb** is disabled (this is used for **DrugThumbs** on PSM diagrams, where *objects* and *lines* are positioned automatically. Property *AllowDragIfSelectedAlone* returns whether the *object* can be dragged

when it is the only object selected. Property *CanBeDraggedInGroup* returns true if the *object* can be dragged when more *objects* are selected.
Methods *DragStarted*, *DragDelta* and *DragCompleted* are handling the dragging itself. They are called depending on mouse input. *DragDelta* changes the values of *X* and *Y* and thus moves the element on the canvas. *DragCompleted* aggregates all the changes done in *DragDelta* and issues **MoveElementCommand**. This command actually does not move the object (because it only puts the object on the same position where it was before the command was executed), but the command is pushed into undo stack and thanks to that the dragging can be undone/redone (see chapters 6.3 and 6.4 for description of undo).
All the methods *DragStarted*, *DragDelta* and *DragCompleted* are protected virtual and can be overridden by derived classes.

3.2.2 ResizeThumb



ResizeThumb is a small class that handles resizing of another element (and is related to **ResizeElementCommand**). Again, it lacks any visual representation. **ResizeDecoratorTemplate** is a control template made of **ResizeThumbs** and it can be applied to virtually any **Control** and can be used to resize the control via drag and drop.

*At run time, **ResizeDecoratorTemplate** is shown on those elements that are selected at the time. The template allows resizing the selected element. ResizeDecoratorTemplate is defined in ControlTemplates.xaml. To allow resizing for an object, declare control with ResizeDecoratorTemplate. This is the declaration taken from **XCaseCommentTemplate**:*

```
<Grid x:Class="XCase.View.Controls.XCaseCommentTemplate">
      <Control Name="ResizeDecorator"
            Visibility="Collapsed"
            Template="{StaticResource ResizeDecoratorTemplate}"/>
</Grid>
```

*Normally, the **ResizeDecoratorTemplate** is collapsed. When the Visibility of ResizeDecorator is changed to **Visible** (usually when the control is selected), border-like control that allows resizing is shown:*

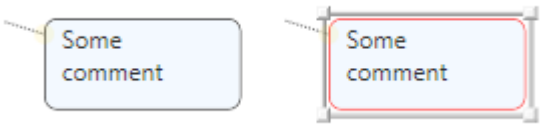


Figure 9 - Displayed ResizeDecoratorTemplate

3.2.3 IConnectable, ConnectableDragThumb

IConnectable is an interface required for *objects* that should be connected by *lines*, **ConnectableDragThumb** its basic implementation. **IConnectable** basically requires the element to be able to create end points for lines.
ConnectableDragThumb derives from **DragThumb**. Its two main abilities – dragging via mouse (derived from **DragThumb**) and connecting together make **ConnectableDragThumb** a suitable base class for most diagram elements. **ResizeDecoratorTemplate** is often used for resizing subclasses of **ConnectableDragThumb**.

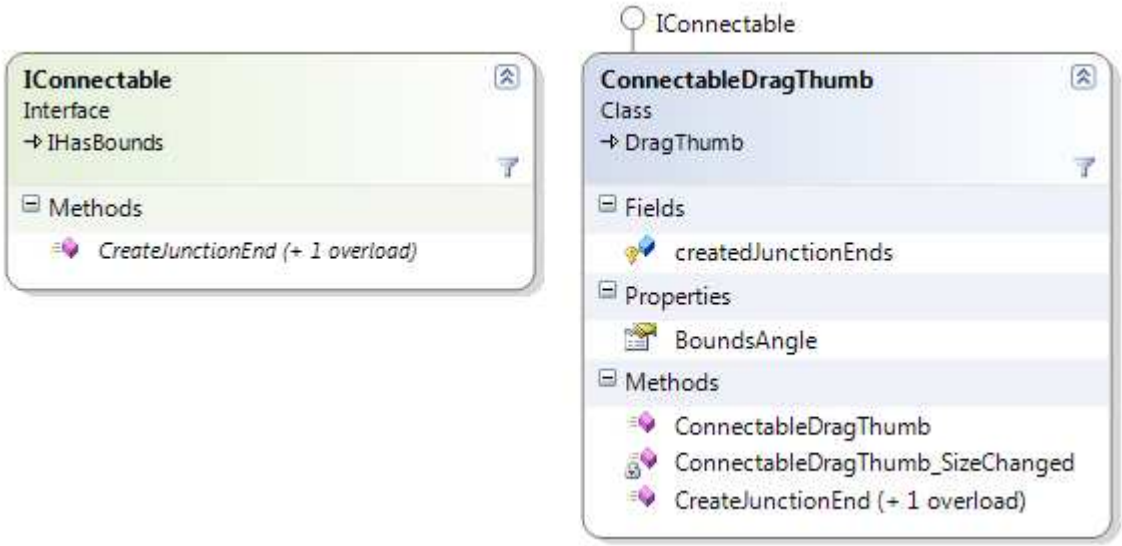


Figure 10

The key method of **IConnectable** is *CreateJunctionEnd* that creates new **JunctionPoint** and places it on the border of the control. Later, *line* can be attached to this point. References to created points are stored in *createdJunctionEnds*.

3.2.4 Templates of elements

As written earlier, most of the diagram elements derive from **ConnectableDrugThumb**. Usually each element has some template that contains its visual representation; the class itself contains only logic above the visual representation. The template is usually registered as a static resource and is loaded when the element is created (in the constructor).

*This example shows how **XCaseCommentaryTemplate** is assigned to **XCaseComment** – class that represent comments on diagrams:*

```
public XCaseComment(XCaseCanvas xCaseCanvas)
                : base(xCaseCanvas)
{
        #region Commentary Template Init
        Template = (ControlTemplate)Application.Current.Resources["XCaseCommentaryTemplate"];
        ApplyTemplate();
…
```

The template class can reference the elements in the template (using WPF *Template.FindName* call, that returns a control from a template by its name).

3.3 Lines

Two objects contained in View.dll assembly contain sufficient functionality for displaying all lines on diagrams. These objects are **XCaseJunction** and **XCasePrimitiveJunction**.

3.3.1 XCaseJunction

XCaseJunction is can draw a *line* between two *objects*. It can be a direct line or can be broken to a polyline. Elements connected by **XCaseJunctions** must implement **IConnectable** interface. Each point on a junction is a separate control (**JunctionPoint**) that can be dragged on the diagram. **XCaseJunction** is drawn as a polyline connecting the points.
On the example below, four junctions are used – first is the junction between Class1 and Class2, with two inner points created by *BreakLine* calls. Other three junctions connect classes 3-5 to association diamond.

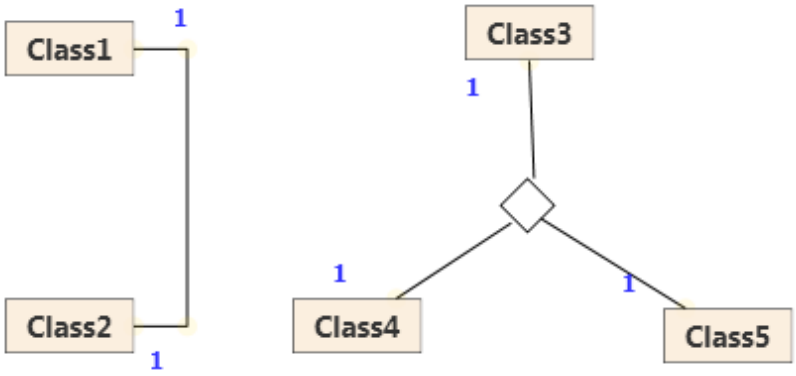


Figure 11 - Examples of XCaseJunction usaje

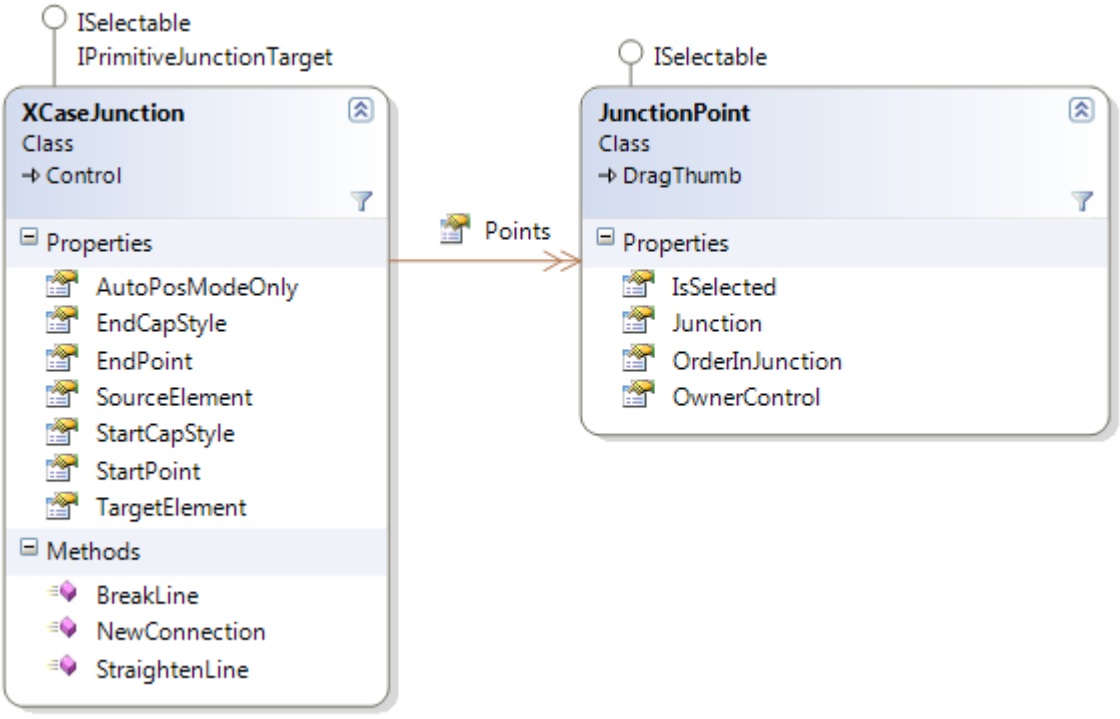


Figure 12

Points collection stores the **JunctionPoints** that the line consists of. More points can be added via *BreakLine* call and deleted via *StraightenLine* call. *NewConnection* call is an initializing method that connects two connectable elements.
Property *AutoPosModeOnly* is set to false by default, but when set to true, the junction will always have only two points – *StartPoint* and *EndPoint* and both will be positioned automatically. This setting is used on PSM Diagrams, where all positioning is done automatically.
Styles *EndCapStyle* and *StartCapStyle* control the figure that is drawn at the beginning and end of the line. Several styles are provided (diamonds, arrows, and triangles).
Property *OwnerControl* of JunctionPoint contains the reference to the control that created the point (see *IConnectable.CreateJunctionEnd*).

3.3.2 XCasePrimitiveJunction

XCasePrimitiveJunction is a much simpler control than **XCaseJunction** that is used to connect elements to other junctions (but can be used to connect an element to any object implementing **IPrimitiveJunctionTarget**)**.** It is always drawn as a straight line, not polyline. It is used for example to connect comments and associations.

4 Representing elements of diagrams

XCase uses Model-View-Controller design pattern. When diagram elements are created in the UML model, event mechanism notifies View about the changes. It is up to the View to reflect the changes in user interface and show the elements on the diagram.

4.1 XCaseCanvas

XCaseCanvas is the class that represents one diagram in user interface. When new diagram is opened, **XCaseCanvas** is empty. It listens to the events in **Diagram** class (*ElementAdded* and *ElementRemoved*). When an element is added **XCaseCanvas** creates its representation.

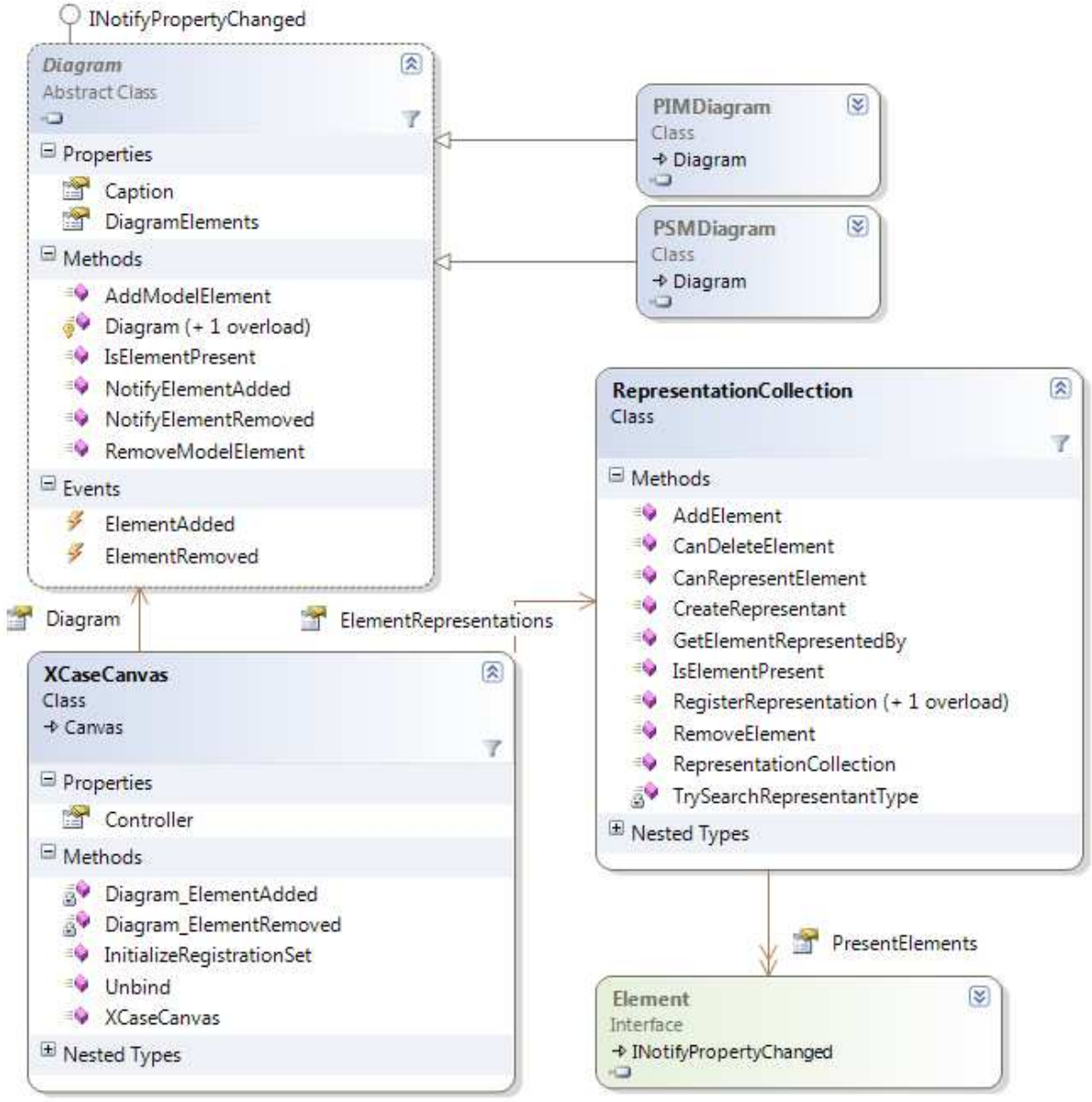


Figure 13

4.2 Representing elements

Part of **XCaseCanvas**' initialization is initialization of the *ElementRepresentations* collection.

ElementRepresentations is a collection of entries of type **RepresentantRegistration**. For each element that is part of the Model and should be represented in View a **RepresentantRegistration** entry must be added into the collection. The entry consist of

- *ModelElementType* – type of the registered diagram element in Model (subclass of **Element**)
- *RepresentantType* – type that should represent the element in View (usually WPF control) implementing **IModelElementRepresentant** interface
- *ControllerType* – type of the controller for the element
- *ViewHelperType* – type of the view helper used for the element

*This is an example of **RepresentantRegistration** entry for **PIM_Association**:*

```
new RepresentantRegistration(
      typeof(Model.PIMClass), // model element type
      typeof(View.Controls.PIM_Class), // representant type
      typeof(Controller.ClassController), // controller type
      typeof(Model.ClassViewHelper),..) // view helper type
```

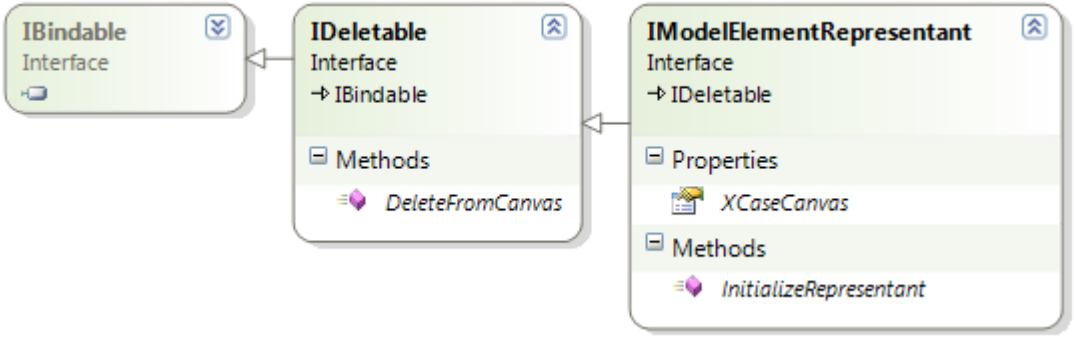


Figure 14

Visual representations of elements must implement **IModelElementRepresentant** interface. Since View uses WPF for drawing, visual representation of an element is usually composed of one or more WPF controls (for example: representation of an **Association** is made of points of the association, line that goes through the points and set of labels). The WPF controls that the representation composes of should be added to **XCaseCanvas** in the implementation of method *InitializeRepresentant*. The same controls that were added in *InitializeRepresentant* should be removed in *DeleteFromCanvas* method.

4.2.1 Sequence of actions performed when new element is added into a diagram

When new element is added into a diagram, *ElementAdded* event is invoked on Diagram class. This event fires the *Diagram_ElementAdded* event handler in **XCaseCanvas**. **XCaseCanvas** then uses its *ElementRepresentations* to instantiate the element.

RepresentationCollection looks up the entry for the type of the element and if it founds one, it can create new representant element, controller and view helper. Then *InitializeRepresentant* method is called on the representant and created controller and view helper are passed as arguments.

If the XCase model were to be extended by new elements, the representant type must be created (if any of the existing ones cannot be used). Probably new controller will also be created and maybe a new view helper. Then these four types need to be added as a **RepresentantRegistration** entry into the *ElementRepresentations* collection.

*Currently there are two sets of **RepresentantRegistrations** used in XCase – one for PSM Diagrams and one for PIM Diagrams; both are defined as static sets in **MainWindow** class and the entry for the new element would be probably added into one of these).*

4.3 Binding Model properties to View

Very often it is desired to update some properties of model element's representant when properties of represented element change (e.g. when Class' Name property is changed via **RenameElementCommand**, **PIM_Class'** *ElementName* property should be updated to contain the same value). Copying value of model element property into representant property can be easily achieved via set of metadata attributes XCase supports and thanks to property change notifications coming from the model elements (model elements implement **INotifyPropertyChange**, collections of elements implement **INotifyCollectionChange**).

Binding infrastructure expects that each model element representant will have two properties – one referring to represented model element and one to the element's ViewHelper. Declarative attribute markup is used to declare these two properties

This section deserves an example – comment example was chosen because it is quite simple, but can demonstrate most of the binding features.

*This is part of **XCaseCommentary** code declaring **ViewHelper** and model element references:*

```
public class XCaseCommentary: IModelElementRepresentant
{
      [ModelElement]
      public Comment ModelComment { get… }

      [ViewHelperElement]
      public CommentViewHelper ViewHelper { get… }
}
```

***ModelElement** attribute is used in **ModelComment** property declaration – this property will be used by the binding infrastructure as a source of model binding.* **ViewHelperElement** *property is used to declare source of view binding in a similar way.*

Another pair of attribute is used to declare binding between pair of properties itself:

```
[ModelPropertyMapping("Body")]
public string CommentText …
```

*The declaration above says that each update of **ModelComment.Body** will update property **CommentText**.*

***ViewPropertyMapping** attribute can be declared in the same way as **ModelPropertyMapping** attribute.*

There is also an equivalent way to define the mappings – use the attributes on classes instead of properties, following declaration is equivalent to the declaration above:

```
[ModelPropertyMapping("Body", "CommentText")]
public class XCaseCommentary
{
      …
      public string CommentText …
}
```

*This second way of declaring the mappings is useful when a property is declared in a base class but mapping is defined in derived class (thus there is no place to declare an attribute in the derived class without overriding the property). For example this is how X and Y properties of base class **DragThumb** (that is a base class to **ConnectableDragThumb**) are bound to **ViewHelper** properties; **X** and **Y** are not overridden in **XCaseCommentary**:*

```
[ViewHelperPropertyMapping("X", "X")]
[ViewHelperPropertyMapping("Y", "Y")]
public class XCaseCommentary : ConnectableDragThumb,
     IModelElementRepresentant
```

*This was the declarative part. The process of binding on a certain object must be explicitly started at runtime by calling method **StartBindings** (extension method of **IBindable** interface – it doesn't have to be implemented by model element representants, only **XCase.UMLController** interface must be imported).*

***CloseBindings** method suspends binding for the object.*

*Good time to call **StartBindings** is at the end of **InitializeRepresentant** method (from **IModelRepresentant** interface), good time to call **CloseBindings** is at the end of **DeleteFromCanvas** method (from **IDeletable**).*

*Note: C# compiler requires **this** qualifier when calling **StartBindings** and **CloseBindings**. These methods must be called using **this** qualifier: **this.StartBindings()** resp**. this.CloseBindings()**. Both methods also provide overrides that start/closes only model bindings or only view bindings.*

Internally, copying values from Model to View uses reflection to identify the properties with assigned mapping attributes. **TypeBindingData** class does this job.

5 TreeLayout

The purpose of this static class is to ensure layouting in case of PSM diagrams. In contrast to PIM diagrams, these have strictly tree structure in which also order of children of an element is important. For this reason, user's positioning is disabled and automatic layouting is performed.

5.1 Used Algorithm

5.1.1 Layouting of a Forest

When a PSM diagram contains several roots and so its elements form a forest instead of a single tree, these trees are layouted side by side from left to right, separated by a gap of fixed width.

5.1.2 Layouting of a Single Tree

The root of the tree gets the information of desired top and left coordinates of the entire tree. From height of the root and size of fixed gap between next levels of the tree, situation of the top of the root's children is computed. Then layouting of the first child is called recursively, returning real width of the child's subtree (if this subtree consists only of this node, width of this node is returned). This width is used to compute situation of the left of the second child (by adding size of fixed horizontal gap to the width), then this child can be layouted. Other children are layouted analogically. Now we know the width of the entire subtree and we can layout the root to the center.

5.2 TreeLayout Class

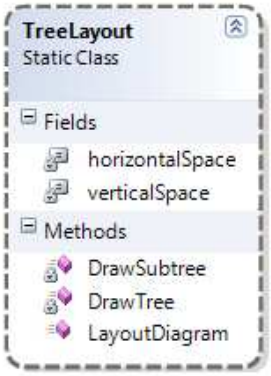


Figure 15 - TreeLayout class

- *active* – Indicates whether layouting is active now
- *horizontalSpace* – Size of fixed gap between neighboring nodes on the same level
- *verticalSpace* – Size of fixed gap between neighboring levels of a tree
- *LayoutDiagram()* – Performs complete layout of given diagram
- *DrawTree()* – Draws given element and all its children
- *DrawSubtree()* – Draws all children of given element
- *SwitchOff()* – Suppresses layouting
- *SwitchOn()* – Activates layouting

6 Controllers

Controller as in MVC in our case consists of two parts. Controllers and Commands. Controllers also consist of two parts, ElementControllers and CommandControllers.

6.1 Element Controllers

Classes providing methods for changing the model (or ViewHelpers) using Commands, each PIM and PSM element has its own controller providing means to rename, add/remove attributes, operations etc. Used mainly by view elements, so they do not create Commands on their own.

6.1.1 Example

PSM_ClassController Controls PSMClass Model element is used by PSM_Class View element. Provides methods such as DeriveNewRootPSMClass, AddClassSpecialization, GroupBy, AddChildren, ManageAttributes etc., which are usually wrappers for Commands creation.

6.2 CommandControllers

6.2.1 Diagram Controller

There is one for each diagram, stores diagram-specific settings like Diagram (model class representing a Diagram), and provides diagram-specific methods like *NewAssociation*, *NewGeneralization* etc., which are usually also wrappers for commands creation and initialization.

6.2.2 Model Controller

One per project (we support only one project so far), stores the undo stack and the redo stack, provides model-specific methods like *IsElementUsedInDiagrams*, *HasElementPSMDependencies*, *CreateSimpleType* etc.

6.2.3 View Controller

Provides methods for moving and resizing view elements, which are usually also wrappers for commands like **MoveElement**, **ResizeElement**, **BreakLine** etc.

6.3 Commands

XCase's commands are small objects that are typically created in response to some user action (typically toolbar click or menu selection) and somehow alter the UML model and diagrams built upon the model.

6.3.1 Commands overview

The key methods of each command are *Execute* and *UnExecute*. Execute performs some operations, *UnExecute* reverts the changes done by Execute. When creating a new command, one usually does not have to override *Execute* or *UnExecute*, because they are implemented as Template methods (design pattern) that rely on *CommandOperation* and *UndoOperation*.
CommandOperation should perform the task itself, *Execute* serves as a kind of wrapper of *CommandOperation* that integrates the task into the MVC infrastructure. Relation between *UnExecute* and *UndoOperation* is analogical. There is also a *RedoOperation* which you will need to override in case that *CommandOperation* cannot be used by Redo. Choose proper base class for the new command (typically it would be **DiagramCommandBase**,

ModelCommandBase or **MacroCommand**) to achieve desired integration into the program. Turn to overriding of *Execute* and *UnExecute* only when solving some uncommon scenario.
Base class of all commands is abstract **CommandBase** that contains some basic fields and abstract methods *CommandOperation*, *UndoOperation*, *RedoOperation* and *CanExecute* and also a default implementation of *Execute* method (that calls *CommandOperation*) and *UnExecute* (that calls *UndoOperation*). Inheriting classes have to implement these methods:

- *CommandOperation* – this method should perform the actual effective function of the command (e.g. add an element to a diagram, add attribute to a class)
- *UndoOperation* – should revert all changes done by *CommandOperation*
- *CanExecute* – should return true if command can be executed – when it is properly initialized and all relevant objects are in a state that permits execution of the command

Beside *CanExecute* there is another method that can be used to validate commands – marking properties of the command with *MandatoryArgument* and *CommandResult* attributes.

6.4 Command stacks

To support undo and redo operations, XCase works with stacks of commands. When a command is executed, it is pushed to the undo stack. When user wants to undo the last command, the command is popped from the undo stack, its operation is reverted and the command is pushed to the redo stack. When user wants to redo last undone command, the command is popped from the redo stack and executed and pushed again to the undo stack. Thanks to command stacks, XCase support undo/redo with unlimited depth.
Pair of stacks (undo and redo stack) exists only in one instance. All commands are pushed to these stacks. Both **DiagramController's** and **ModelController's** *getUndoStack* and *getRedoStack* methods return these stacks.
There are two actual commands that follow the two previous scenarious – **UndoCommand** and **RedoCommand**. They are executed when user wants to undo or redo his action.

6.5 More complex commands

StackedCommandBase is the next class in commands hierarchy. Again, it is an abstract class, but its Execute and UnExecute method work with command stacks. When created, reference to stacks is passed in the constructor (undo/redo stacks are part of **ModelController** object which is the constructor's parameter). When executed, command is pushed to command stack, when unexecuted, the command is pushed to the redo stack. Here the template methods *Execute* and *UnExecute* work with command stacks. **StackedCommandBase** implements **IStackedCommand** interface that can also be used to work with stacked commands.
DiagramCommandBase is again an abstract class – subclass of **StackedCommandBase** and parent of all commands that alter diagrams.
ModelCommandBase is another abstract subclass of **StackedCommandBase** and parent of all commands that alter the model.
When deciding whether to implement the new command as a diagram command or model command, look at the action the command does from the user point of view. Does the action alter only one diagram or the model under all the diagrams?
Diagram commands typically work with **Diagram** object and its *DiagramElements* collection (when adding an element to the diagram or removing an element from diagram). Another type of diagram command is **ViewCommand**. **ViewCommands** alter visualization of an element

on the particular diagram (its position, size etc.) – this is done by changing a certain **ViewHelper**.
Model commands typically alter properties of some model element (**Class**, **Association** etc.).
MacroCommand is a special kind of command that is composed of other commands. Using MacroCommand, more complex action can be executed as a single command.
When deciding whether to join commands into one MacroCommand, look again from the user point of view. When user selects "undo", should be the whole action reverted or should it be taken back step by step? The first scenario speaks for MacroCommand, the second for separate commands.

6.6 HOW-TO create a command

- Create a new class in the Controller.Commands folder or one of its subfolders (depends on what the command does)
- You may want to change the namespace to *XCase.Controller.Commands* if you do not want to include another namespace into the place where you use your new command
- Decide whether the command you are creating, involves only one diagram or the whole model and choose **DiagramCommandBase** or **ModelCommandBase** as the ancestor of your new command accordingly
- Decide the complexity of your command. If it is a command, which could utilize another commands, create new basic operations as separate commands and then group them into one **MacroCommand<ModelController>** or **MacroCommand<DiagramController>** along with the existing commands you would like to use using **MacroCommand.Commands** collection in a command preparation method of the newly created MacroCommand, like *Set()* or *InitializeCommand()* (it is up to you how you name this method, it is not standardized), which will be called by the user of your command (**ElementController**, **PropertiesWindow**, **MainMenuCommand** etc.) after command creation and before command execution.
- In the constructor of your command, set the *Description* property to a text description of what your command does. Store the description in the **CommandDescription.resx** file in the Commands folder.
- Create a Factory for your command. Look at another command of the same type (**DiagramCommandBase** or **ModelCommandBase**). The factories are all the same except names.
- If your command does not support Undo, you can set the *undoable* property to false in the constructor. This will cause that this command will not be placed on the **UndoStack** after execution.
- Override the *CanExecute*, *CommandOperation* and *UndoOperation* methods of the **CommandBase**.
- If you are returning **false** in *CanExecute* or ***OperationResult.Failed*** in *UndoOperation*, do not forget to fill the *ErrorDescription* property with an item from the **CommandError.resx** file.

- You may need to override *RedoOperation* also, if your command cannot use *CommandOperation* as *RedoOperation*. (When you are creating a new object, you don't want to create another one when Redoing, you want to return the already created one from **CommandOperation**)

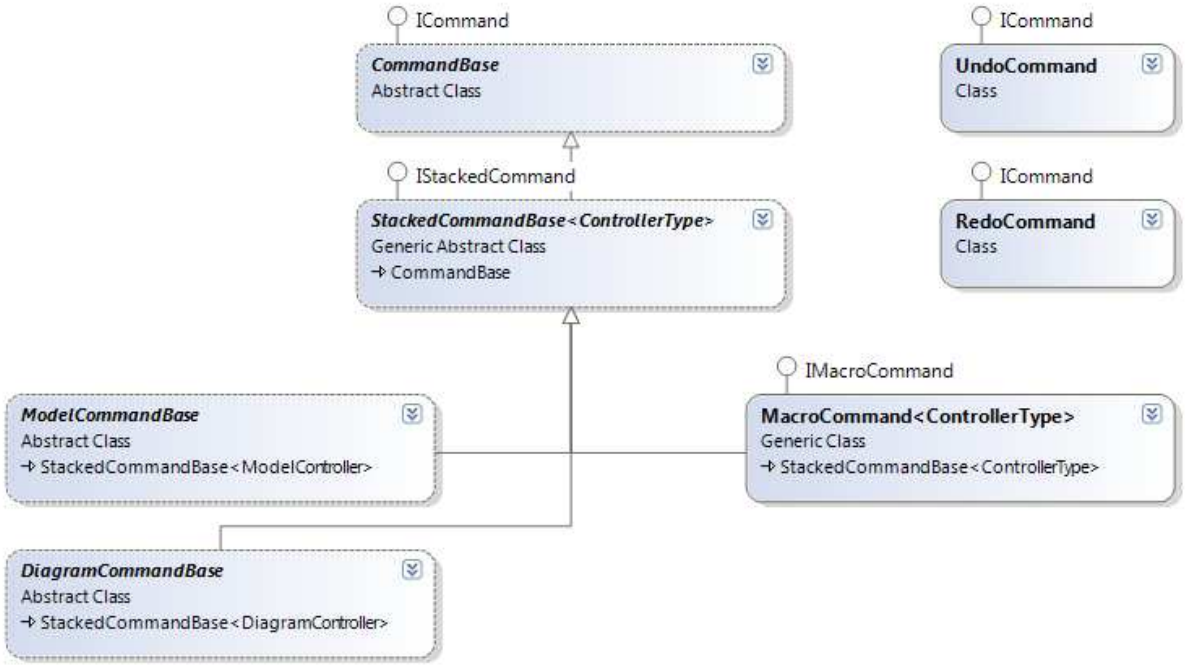


Figure 16

6.7 Command factories

Command factories are used as the only way of creating instances of standard Commands in Controller.
HOW-TO create a factory is included in HOW-TO create a command in Commands.

7 Setup

XCase Setup is realized as a standard Visual Studio 2008 Setup Project. It automatically detects dependencies; there are no manually added dialogs.
Prerequisites set:

- Windows Installer 3.1
- .NET Framework 3.5 SP1

There is a manually added icon, which is used as a desktop icon and a Start Menu icon. For some reason, the setup project doesn't allow to use the icon set to the XCase.exe file.
The Setup is set to create a Desktop Icon and the Start Menu Icons.
Also, the default banner is overridden with our own:

When run, the setup detects presence of the prerequisites and downloads them from the Microsoft's website as needed.

8 GUI - DockingLibrary

DockingLibrary is an external library [3] for managing dockable windows used in the XCase editor (Navigator, Project, Properties) and also for managing multiple diagrams at once in a **TabControl**. Its goal is to acquire a way of working with dockable panels very similar to Visual Studio.
The fundamental classes of this library are **DockManager**, **Pane** and **ManagedContent**.

8.1 DockManager

DockManager is responsible for managing the main window layout. It is a user control which can be easily embedded into a window using just several lines of code.

8.1.1 Pane

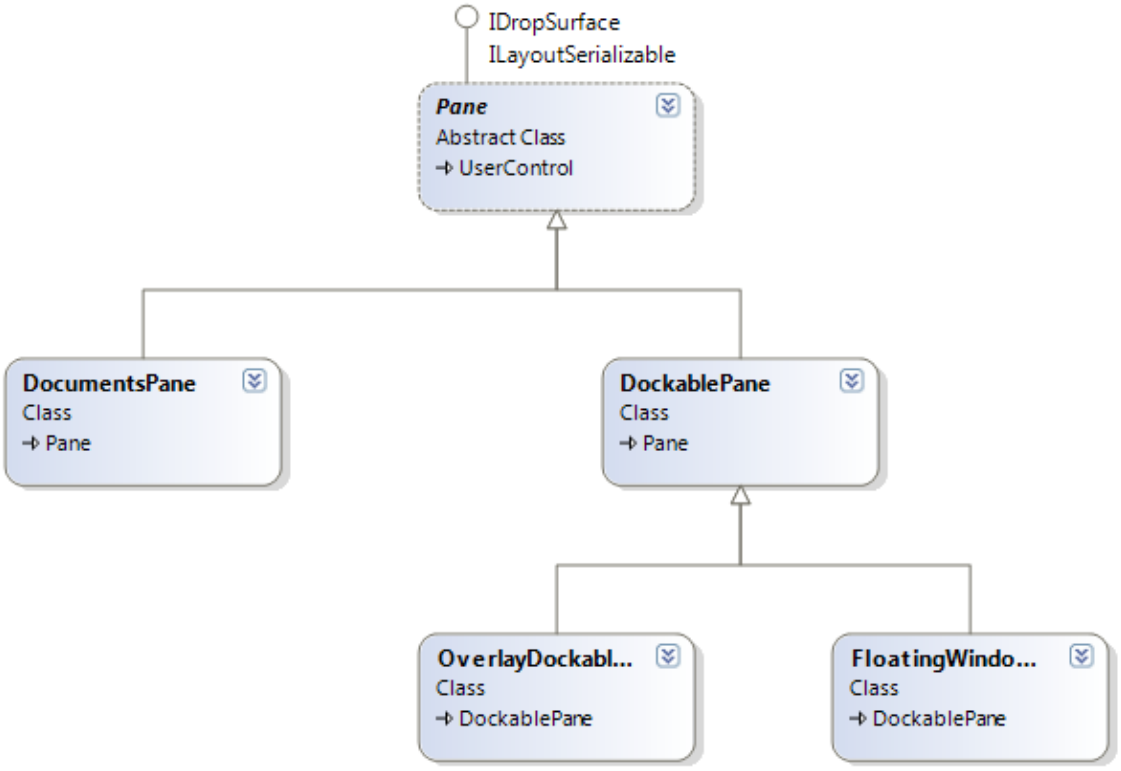


Figure 17 - Pane

Pane represents the window area which a) can be docked to a border (**DockablePane**), b) contains documents in the main part of the window (**Documents Pane**).

8.1.2 ManagedContent

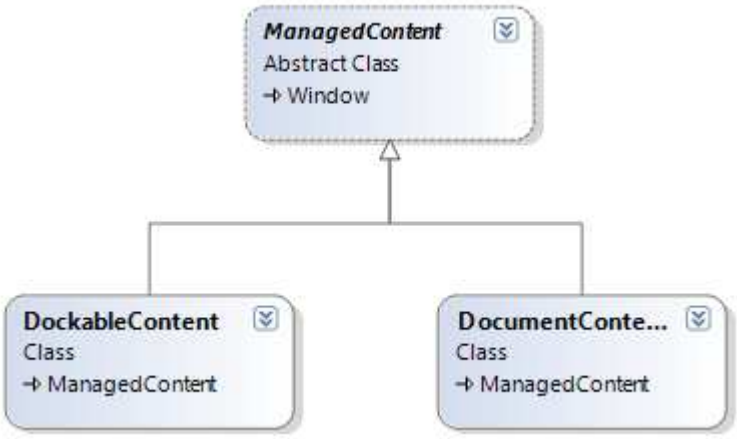


Figure 18 - ManagedContent

The content of Panes is compound of **ManagedContent**. **DockableContent** is contained in **DockablePane** and all dockable windows have to inherit from this class (in our case, these windows are **NavigatorWindow**, **ProjectsWindow** and **PropertiesWindow**). Analogically, **DocumentContent** is contained in **DocumentsPane** and all document windows have to inherit from it (**PanelWindow** in our case).

8.2 Changes to Original Library

We made several changes to this library. The majority of them are just subtle cosmetic changes, two changes which are of major importance are

- *DocumentsPane.ActivateTab(TabItem item*) method – allows a user to easily activate chosen tab
- *DockManager.ActiveTabChanged* event – invoked when an active tab in **DocumentsPane** is changed

9 GUI – Windows

9.1 Main window

Main window ensures proper displaying of dockable windows (Navigator, Project, Properties) and all diagram tabs as well as interaction among them.

9.1.1 Docking & Managing Diagrams

The management of dockable windows is achieved by using DockingLibrary – Main window incorporates **DockManager** and after loading registers Navigator, Project and Properties as dockable windows of **DockManager**. Main window also handles opening/closing/changing of diagrams and notifications to other windows about it.
When **DockManager** invokes event *ActiveTabChanged*, Main window reacts to it by finding a diagram associated with now active tab and invokes event *ActiveDiagramChanged*. This event is handled by method *OnActiveDiagramChanged(…)*. Handling of this event is required for proper synchronization amongst active diagram, Navigator, Properties and also the main toolbar.

9.1.2 Main toolbar

Main toolbar consists of several groups of buttons for controlling project, displayed dockable windows and, above all, editing diagrams. The selection of displayed groups depends on the type of active diagram, the possibility of clicking on particular buttons depends on the state of particular active diagram (e.g. logically, Undo is disabled on newly created diagram). That is the reason why main toolbar visualization has to be adjusted after each occurrence of *ActiveDiagramChanged* event.

9.2 Navigator window

Navigator window enables structuring of model classes into packages and their easy control. Using Navigator is also the only way to administer classes which are not present in any diagram at the moment.

9.2.1 Model Administration

When a project is loaded, Navigator is bound to it using the method *BindToProject(…)*. This ensures that model's collection *Classes* is used as items source of **TreeViewItem** *modelClasses* and collection *NestedPackages* is used as items source of **TreeViewItem** *nestedPackages*. Proper displaying of all classes, attributes, packages (and their recursively nested packages) is handled by **DataTemplates** *packageTemplate* and *classTemplate*.
Editing/adding/removing of these elements is possible via context menus of particular elements. These context menus are also defined in **DataTemplates** mentioned above.

9.2.2 Interaction with Other Windows

When a class is selected in Navigator, Navigator invokes event *NavigatorSelectedClass* (which includes class reference). Main window reacts to this event by selecting referred class on canvas, if the class is visualized in active diagram, and displaying this class in Properties.
On the other side, Navigator reacts to selecting a class on canvas (event *XCaseCanvas.SelectedItems.CollectionChanged*). As a reaction to the event, class selected on canvas is also selected in Navigator.

9.3 Project window

Project window enables projects' diagrams visualization and administration.

9.3.1 Overview

When a project is loaded, Project window is bound to it using the method *BindToProject(…)*. **DataTemplate** *projectTemplate* then ensures visualization of the project and *memberTemplate* handles visualization of particular diagrams. These templates also contain definitions of context menus of project and diagrams and assign event handler for handling double click on a diagram. Renaming of project and adding/renaming/removing diagrams is then administered through these context menus.

9.3.2 Interaction with Other Windows

Project window invokes *DiagramDClick* event after double click on a diagram. In reaction to this event, Main window opens a tab with selected diagram (if not open so far) and gives it focus.
Event *DiagramRemove* is invoked after removing a diagram through context menu. Main window reacts to it to ensure closing a tab containing this diagram visualization.
Event *DiagramRename* is invoked after renaming a diagram through context menu. Invoking this event enables Main window to change the header of appropriate tab accordingly to the new diagram caption.

9.4 Properties window

Properties window displays properties of a PIM or PSM element selected on the canvas or a PIM class selected in the Navigator window. If no or more elements are selected, Properties window does not display anything.

There are several independent **XCaseGridBase** components in the Properties window, each for one type of visual element. Only one is visible at time.

```
public abstract class XCaseGridBase : UserControl
{
    abstract public void UpdateContent();
    //…
}
```

In the following table there is a list of all elements that can be displayed in the Properties window together with the grid used for their displaying. All the listed grids can be found in *Gui\Windows\Properties*

PIM Element	XCaseGridBase grid used
PIM Class	PIMClassGrid
Association class	AssociationClassGrid
PIM Association	AssociationGrid
Comments	CommentGrid
PSM Element	XCaseGridBase grid used
PSM Class	PSMClassGrid
PSM Association	PSMAssociationGrid
PSM Attribute Container	AttributeContainerGrid
Content container	ContentContainerGrid
Comment	CommentGrid

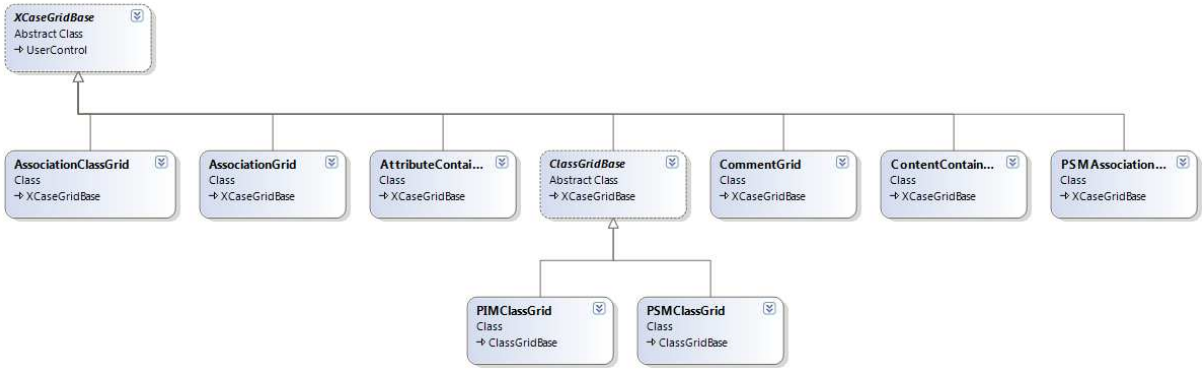


Figure 19

When a selected element on the canvas is changed (user selects something else), *SelectionChanged* is invoked. According to what has been selected, one of these methods is called and the appropriate specialized grid becomes visible.

```
private void DisplaySelectedPIMClass(XCaseViewBase c)
private void DisplaySelectedPSMClass(XCaseViewBase c)
private void DisplaySelectedComment(XCaseComment c)
private void DisplaySelectedAssociation(XCaseAssociation a)
private void DisplaySelectedAssociationClass(XCaseAssociationClass c)
private void DisplaySelectedPSMContentContainer(PSM_ContentContainer c)
private void DisplaySelectedPSMAssociation(PSM_Association p)
private void DisplayAttributeContainer(PSM_AttributeContainer c)
```

When user selects a PIM class in the Navigator window *DisplayModelClass(Class c)* method is called. In this case, just model properties of the class are displayed in the Properties window, not appearance properties.

When selection is being changed, the content of the previously displayed **XCaseGridBase** component is updated (*UpdateContent)* to ensure that all unsaved changes are saved. Then, new **XCaseGridBase** component is displayed in the Properties window (*Display* is called on the appropriate specialized grid).

10 Storing and loading of XCase projects

10.1 Serializator

XMLSerializator class provides an interface for serialization of the entire XCase project to a single XML file. We call this file with serialized XCase project 'XCase XML file' and use our own suffix **.XCase* for it. Class **XmlVoc** is used as a vocabulary of XML element and attribute names while serializing the XCase project.

Public interface:

```
public class XmlSerializator
{
      // project = XCase project to serialize
      public XmlSerializator(Project project);

      // filename = Name of output XCase XML file with serialized project
      // Returns true if serialization was successful, false otherwise
      public bool SerilizeTo(string filename);
}
```

XCase XML file hence contains full information about one XCase project and such project can be later completely restored from its XCase XML file by using **XMLDeserializator** class.

XCase XML file

The structure of XCase XML files is precisely described by *XCaseSchema.xsd* (Description provided in W3C XML Schema language). The overall structure of XCase XML file looks as the following:

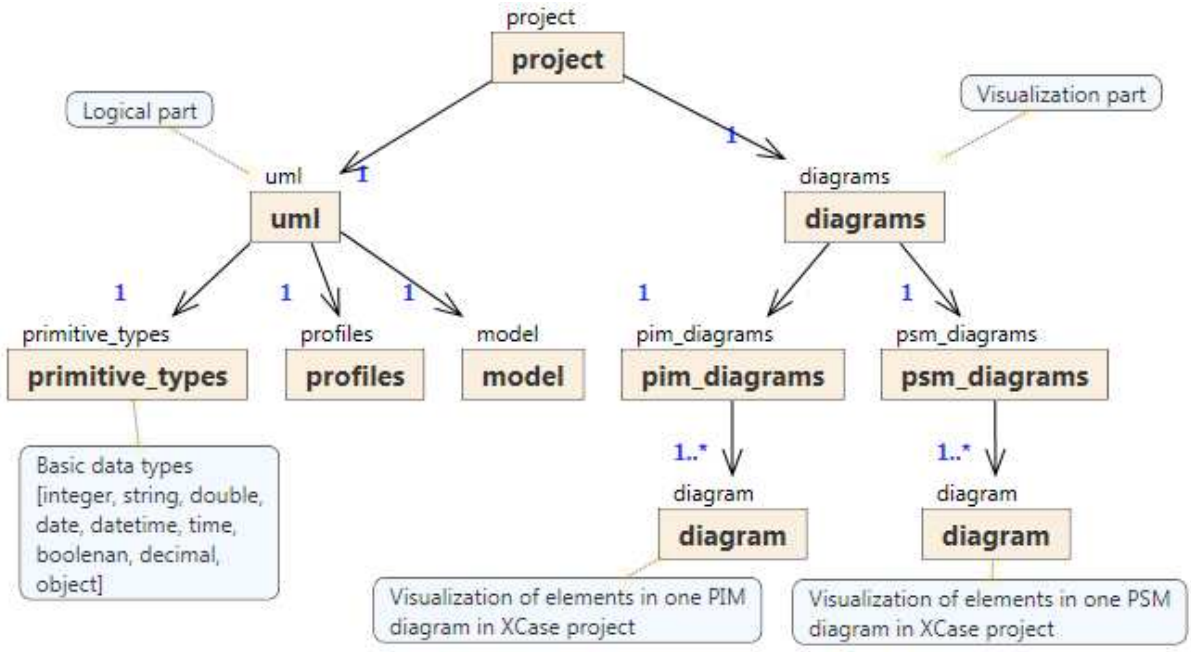


Figure 20

10.1.1 Serialization order

First, UML (metamodel+model) is serialized and then all the diagrams. This approach ensures that the logical part of the project (UML metamodel + model) and the visualization (diagrams) are completely separated in the XCase XML file.

Serialization order of the most important UML and diagrams parts is the following:

UML part

1] Data types (basic embedded data types: integer, string, date …)
2] Profiles
3] Model
    3.1] PIM classes
        - Their derived PSM classes
            - Their nested components (can be recursive):
                - PSM Association
                - PSM Attribute Container
                - PSM Content Choice
                - PSM Content Container
                - PSM Class Union
    3.2] PIM association classes
        - Their derived PSM classes
            - Their nested components (can be recursive) – the same as listed above

    3.3] PIM associations (all types: simple associations, aggregations, compositions)
    3.4] PIM generalizations

Diagrams part

1] PIM diagrams with their visual elements:

- Class (PIM class)
- Association class
- Association
- Comment
- Generalization

2] PSM diagrams with their visual elements

- Class (PSM class)
- PSM association
- Comment
- PSM attribute container
- PSM content container
- PSM content choice
- PSM class union

ID Table

During serialization, each serialized element gets its unique ID. This is then serialized as ID XML attribute and determines the serialized element. All elements with assigned ID attribute are being stored in HashTable *idTable* during serialization.

```
HashTable idTable; //[Key = Element; Value = ID]
```

References are then handled via ID/REF mechanism.

In the XCase XML file, elements referring to other elements use attribute with keyword containing *ref*. If there is an element with *@ref = n* somewhere in XCase XML file, it refers to an element with *@id = n*

10.1.2 Example

In the UML part there is a PSM class with @id = 43.

```
<xc:psm_class id="43" name="Class1">

    …

</xc:psm_class>
```

In diagrams part there is the visualization for this class, which is expressed by @ref = 43.

```
<xc:class ref="43"    methods_collapsed="False"
                      properties_collapsed="False"
                      element_label_collapsed="False"
                      element_label_aligned_right="False">
  <xc:appearance>       …          </xc:appearance>

…

</xc:class>
```

10.2 XML Deserializator

XMLDeserializator class provides an interface for restoration of then entire XCase project (UML model + visualization) from XCase XML file. Class **XmlVoc** is used as a vocabulary of XML element and attribute names while restoring the XCase project.

Public interface:

```
public class XmlDeserializator
{
  public XmlDeserializator();

  public static bool ValidateXML(System.IO.Stream input, ref String
  message);
  public static bool ValidateXML(string file, ref String message);

  // file = Valid XCase XML file
  // window = window where to restore the visualization diagrams
  public void RestoreProject(string file, MainWindow window);

  // input = Valid streamed XCase XML
  // window = window where to restore the visualization diagrams
  public void RestoreProject(System.IO.Stream input, MainWindow window);
}
```

Validation

Before starting the restoration itself it is recommended to check the validity of the input XCase XML file by calling *ValidateXML* method.
It returns *true* if the passed XML is a valid XCase XML file. *False* is returned otherwise.

ID Table

While reading XCase XML file, all restored elements are added to *idTable* along with their IDs. This ensures correct restoration of references between elements. References between elements are provides via @id/@ref attributes in the XCase XML file.

```
HashTable idTable; // [Key = ID; Value = Element]
```

10.2.1 Restoration Order

First, primitive types and profiles are restored. Then there are two phases: PIM and PSM. In PIM phase, all PIM diagrams with all their PIM elements are restored;, in PSM phase, all PSM diagrams with all their PSM elements. A PIM or a PSM element is always restored together with its visualization.

1] Primitive Types
2] Profiles
3] PIM elements
  3.1] Comments in PIM diagrams
  3.2] Datatypes
  3.3] Packages (recursive) – PIM elements
  3.4] PIM classes

3.5] Association classes
3.5] PIM Associations
3.6] PIM Generalizations
4] PSM elements
4.1] Derived PSM classes and their components (recursive)
4.2] PSM Generalizations
4.3] PSM Associations

10.3 XmlVocabulary

Class **XmlVoc** serves as a collection of static strings used as XML element and attribute names in XCase XML file. These static strings are used during serialization as well as during deserialization instead of writing the XML element names right into the source code.

Class **XmlVoc** offers all element and attribute names defined in *XCaseSchema.xsd* as well as some XPath queries constructed from these names.

11 Translation of PSM diagrams into XML schemas

XCase PSM diagram describe a given type of XML documents on the conceptual level. For practical reasons, we need to translate it to an XML schema that describes the type on the logical level. For this, we can apply an XML schema language like XML Schema [4] or RELAX NG [5].

11.1 Description of PSM diagram

PSM diagram contains mainly PSMClasses connected by PSMAssociations – these are main semantic elements and are representations of Classes and Associations in PIM diagrams. In addition it contains other elements that describe structure of the XML document and are not referencing any PIM elements. Those are **PSMClassUnion**, **PSMAttributeContainer**, **PSMContentContainer** and **PSMContentChoice**.
PSMContentContainer, **PSMContentChoice** and **PSMClass** all implement interface **PSMSuperordinateComponent** which means that they can have other classes implementing **PSMSubordinateComponent** interface among their components (*PSMSubordinateComponent.Components* collection). **PSMClassUnions** can have only **PSMClasses** among their components (*PSMClassUnion.Components* collection). **PSMAssociation** always starts in a **PSMSuperordinateComponent** and leads to **PSMAssociationChild**, which is either **PSMClass** or **PSMClassUnion** (see Figure 2).
PSMDiagram is basically a forest of trees composed of these elements where following elements act as nodes:

- **PSMClass**,
- **PSMContentChoice**,
- **PSMAttributeContainer**,
- **PSMContentContainer**
- **PSMClassUnion**

and following as edges:

- **PSMAssociation**
- lines connecting **PSMSuperordinateComponent** to its components
- lines connecting **PSMClassUnion** to its components
- (specializations also act as edges, they will be described later).

Roots of the forest are always **PSMClasses** and they are stored in the *Roots* collection of **PSMDiagram** (see Figure1).

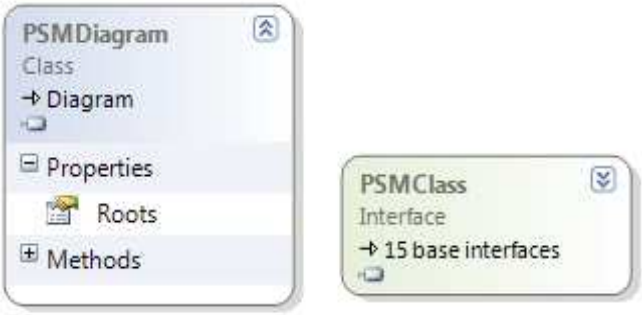


Figure 21 - PSMDiagram and its roots

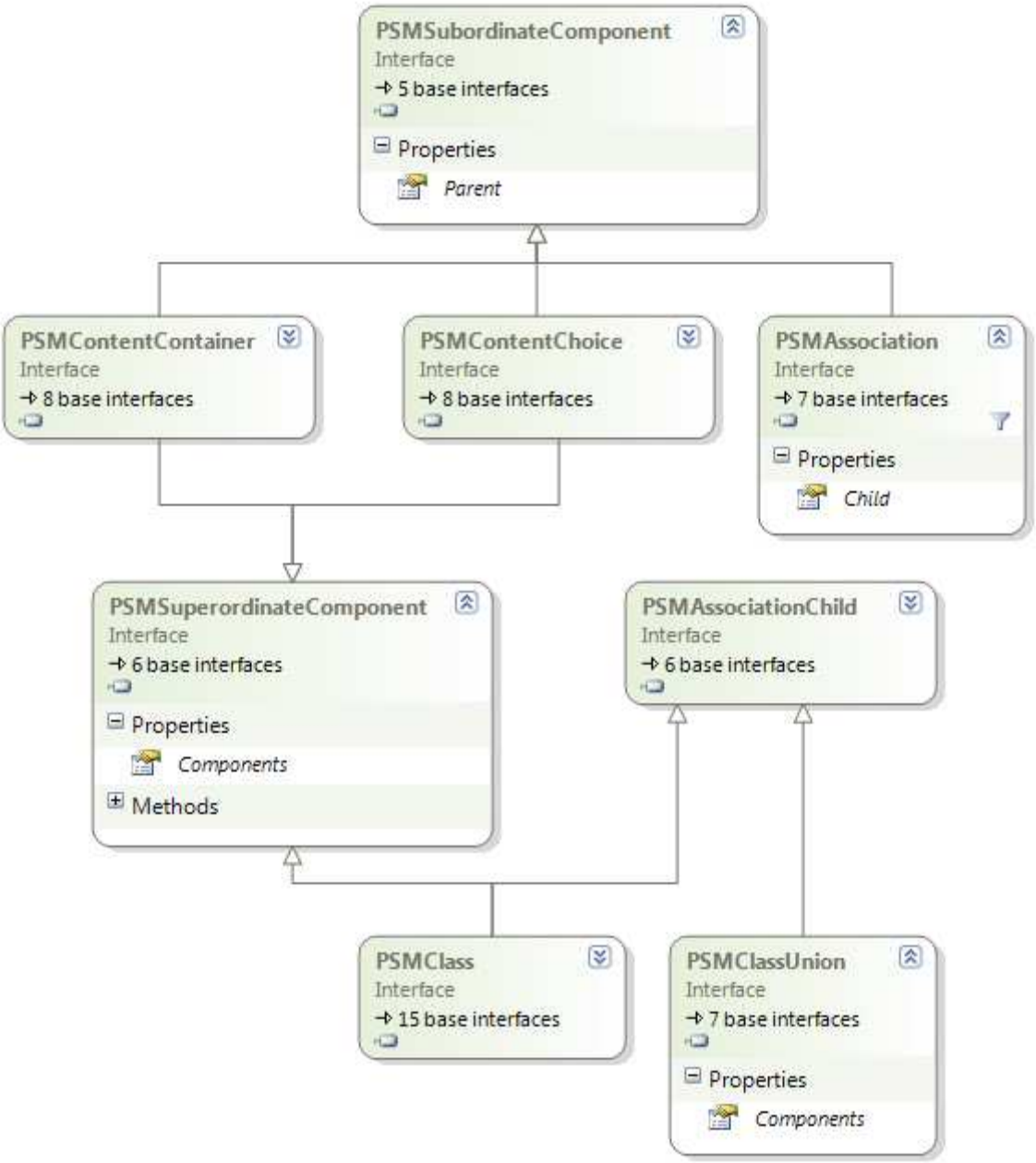


Figure 22 - Basic elements in PSM diagram

11.2 Translation infrastructure

Abstract class **DiagramTranslator** is meant to be parent class of all the classes that translate PSM diagram to a XML schema. One derived class – **XmlSchemaTranslator** is currently provided in XCase. **XmlSchemaTranslator** translates PSM Diagram to a XML schema in XML Schema language [4]. The translation is based on the algorithm described in [6].

There may be constructions that are valid in PSM diagrams (that describe a set of XML documents), but they cannot be expressed by the concrete translator (see Part 4 for a list of these constructions related to translation to XML Schema language). Errors and warnings caused by the constructions in PSM diagrams that cannot be expressed by the concrete translator should be kept in the **TranslationLog** class.
DiagramTranslator has a set of *Translate{..}* methods that all have empty default implementation (except for *TranslateSubordinateComponent* that continues the translation by calling more specific Translate{..} method and *TranslateSpecializations* that continues by calling *TranslateSpecialization* for each specialization. Other methods have empty bodies and it is up to the derived classes to override the body if needed. Also the order in which the methods are called is up to the derived classes. Each derived class must override abstract Translate method, which returns the result of the translation.

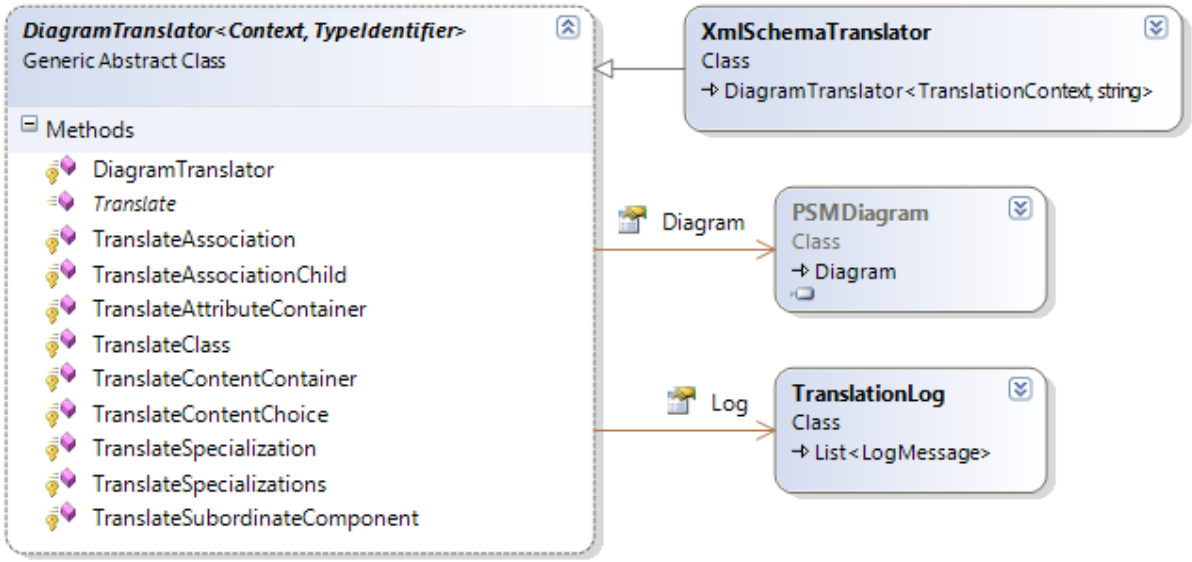


Figure 23 - Translators

11.3 Part 3 Translation to XML Schema language

XmlSchemaTranslator is a subclass of **DiagramTranslator** that translates PSM Diagram into a XML schema in XML Schema language. **XmlSchemaTranslator** and classes supporting the translation to XML Schema reside in namespace **XCase.Translation.XmlSchema**.
XmlSchemaWriter is a wrapper of standard .NET class **XmlWriter** that makes writing declarations of XML Schema language more convenient. XmlSchemaWriters are created by **WriterFactory** class. **SimpleTypesWriter** is a **XmlSchemaWriter** that can write definitions of simple types (definitions already written are kept in **DeclaredTypes** property).

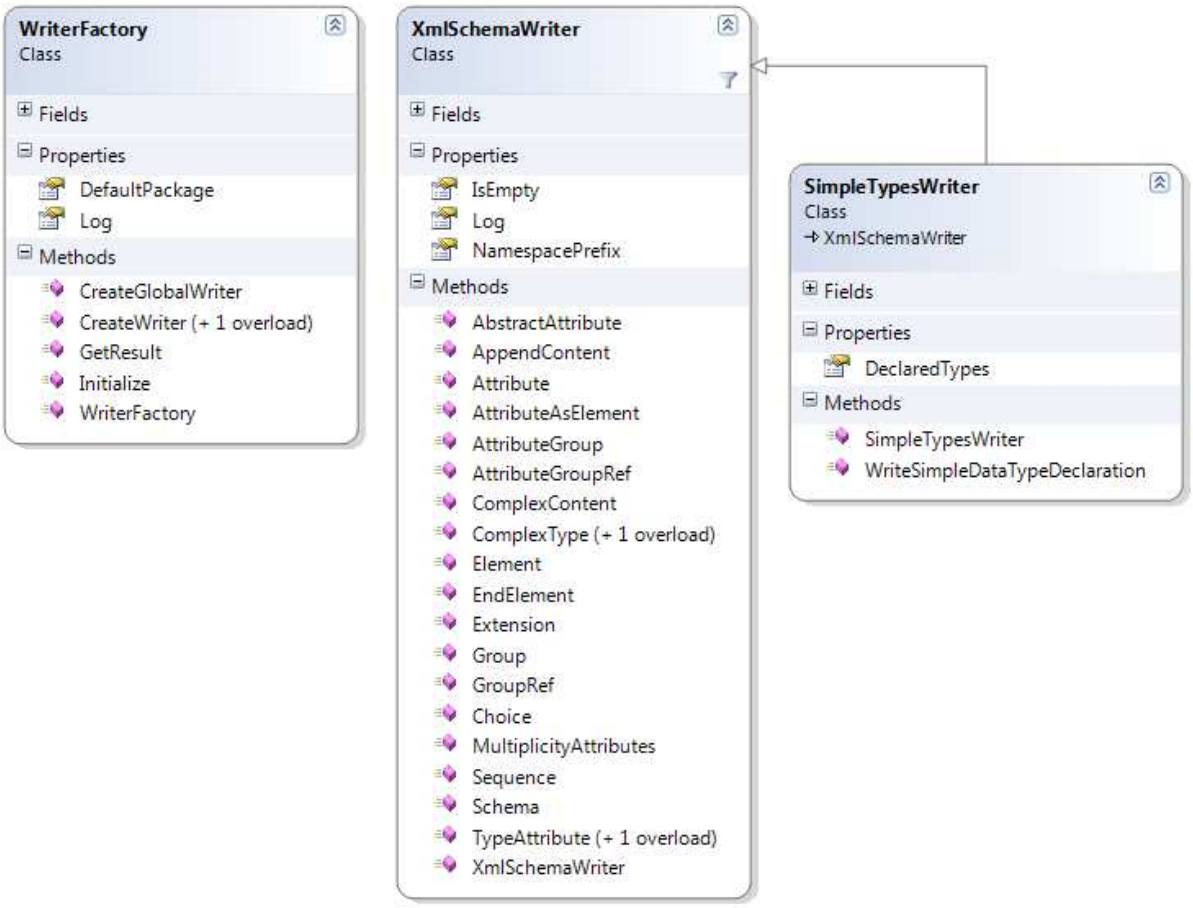


Figure 24 - Translation writers

Each time a new global declaration (complex type, group, attribute group) is needed, *WriterFactory.CreateGlobalWriter* is called and returned writer can be used to write the declaration. Contents of all writers created by these calls are concatenated when *WriterFactory.GetResult* is called (this method returns result of the whole translation). For temporary writers, *CreateWriter* method can be called (writers returned by this call are not considered in *GetResult*).

11.3.1 Basic translation principles

The algorithm [6] works as follows:
Classes with defined element labels are translated to global `xs:complexType` definitions. Classes from roots also add `global xs:element` definition. Classes without element labels are translated into model groups and attribute groups. Associations are translated into element declarations referencing complex types if child of the association is translated into a complex type. If it is translated into groups, references to the groups are propagated to the complex type above. (Note: xml document declaration and xml schema starting declaration will be omitted in the examples).
The algorithm creates a new global declaration or each root class and continues with translation of the contents of the root class and then proceeds recursively. All *XmlSchemaTranslator.Translate{..}* methods have a parameter of type **TranslationContext**. This is a class that contains references to three **XmlSchemaWriters**:

- *TreeDeclarations* – this is the "current writer" where the element is being translated
- *ComposedAttributes* – this writer is used when references to attribute groups are propagated to a complex type above (see section Translation of a class section (2))
- *ComposedContent* – this writer is used when references to groups are propagated to a complex type above (see section Translation of a class section (2))

Assume class U with Attributes $A_1, .., A_m$ and components $C_1, ... C_n$. Type of each Ai is usually a SimpleDataType (translated to built in XML Schema data type or to a restriction of another SimpleDataType), but other data types can be used in Class (see Part 3.5 – Translation of simple types). Each C_i is an association, attribute container, content container, or content choice. In PSM Diagram Class U models a sequence of XML elements and set of XML attributes. The XML elements are modeled by the content of U and XML attributes by the attributes of U. The components of U are translated to a `xs:sequence` declaration

```
<xs:sequence>
XSC1 ... XSCn
</xs:sequence>
```

where XSC_i denotes the translation of C_i (the translation is described later in this section). The `xs:sequence` is denoted XS^E_U.
The attributes of U are translated to a sequence

$$XSA_1 \ldots XSA_m$$

where XSA_i denotes the translation of A_i (the translation is described later in this section). The sequence is denoted XS^A_U.
U gets assigned an automatically generated unique name for the purposes of the translation. Names are generated by the class **NamingSupport** (this class ensures that created complex types gets assigned distinct names even when names of the PSM Classes behind them are identical, which is valid in PSM Diagrams). The assigned name is composed of the name of the type T represented by U and sequence number for the nodes representing T in the PSM Diagram. Name generated by NamingSupport for the class U will be denoted TN_U in this text.
If there is an association leading from U to V and V does not have an element label, the XML attributes modeled by V are propagated to U. Therefore, XS^A_U must be extended with declarations of such XML attributes. The only exceptions are the child nodes of U contained in a content container. In that case the XML attributes are propagated to the XML element

modeled by the content container (because content containers are also translated into `xs:complexType` declaration).
As we show in a moment, the declarations of the XML attributes modeled by classes without element labels are included in the resulting XML schema in a form of attribute groups (i.e. `xs:attributeGroup` construct) named TN_V-a. Therefore, for each child V of U without an element label and not contained in a content container, XS^A_U is extended with

```
<xs:attributeGroup ref="TN_V-a" />
```

11.3.2 Translation of attributes and content

In the following list we describe how U is translated to an XML schema representation. (1) describes the translation in case U has an element label. (2) describes the translation in case U does not have an element label. (3) covers specifics of abstract classes.

(1) Class with an element label

If U has an element label lU, the sequence of XML elements and set of XML attributes modeled by U is enclosed in an XML element named lU. To describe the content of the XML element lU we use a complex type (i.e. `xs:complexType` construct) composed of XS^E_U and XS^A_U . Therefore, XS^E_U and XS^A_U are included in the resulting XML schema in a form of a global complex type definition

```
<xs:complexType name="TN_U">
    XS^E_U
    XS^A_U
</xs:complexType>
```

Moreover, if U is a member of the Roots collection of the PSM Diagram then it models root XML elements named lU with the content described by the complex type TN_U. In that case a global element declaration

```
<xs:element name="lU" type="TN_U" />
```

is added to the resulting XML schema. If U is not among roots, it is an inner node in the tree. In that case either a) there exists an association going to the class U and then the declaration for lU XML elements is created in the scope of the translation of the edge going to U as we show later or b) U is among components of class union and then the element declaration will be created in the scope of the class union.

Method ***XmlSchemaTranslator.TranslateClassWithLabel*** *writes the complex type declaration described above.*

(2) Class without an element label

If U does not have an element label, the sequence of XML elements and set of XML attributes modeled by U is not enclosed in an XML element. The attributes of U model a set of XML attributes that is a subset of XML attributes modeled by the parent of U (if there is any). Similarly, the content of U models a sequence of XML elements that is a part of the sequence of XML elements modeled by the parent of U. Therefore, we cannot include XS^E_U and XS^A_U in the XML schema in the form of a complex type definition, as in the previous case, because `xs:complexType` construction can describe only the whole content of XML elements but not a part. Instead, we use model groups (i.e. `xs:group` construction) and attribute groups (i.e. `xs:attributeGroup` construction) to include XS^E_U and XS^A_U in the XML schema representation as follows

```
<xs:group name="TN_U-c ">
      XS^E_U
</xs:group>
<xs:attributeGroup name="TN_U-a">
      XS^A_U
</xs:attributeGroup>
```

with the name composed of TN_U followed by the string "-c" or "-a", respectively. Instead of `xs:element` declarations, that are created in the case (1), reference to group TN_U-*c* and attribute group TN_U-*a* is propagated to the first ancestor in PSM Diagram tree that is translated to a complex type (this can be either class with an element label or content container).
Note: there is a departure from the rule that says that class with an element label is translated without using groups. It refers to a situation concerning structural representatives and is described in the section devoted to structural representatives [see Part 11.3.3].
Note: translation of xs:attributeGroup is different when the class being translated is under a content choice or class union and the reference to the created group should be propagated to a complex type above the choice resp. class union. This rule is described in section (7) devoted to class unions.

*Method **XmlSchemaTranslator.TranslateClassContentToGroups** writes the group declarations described above.*

(3) Abstract classes

Class can be marked **abstract** in a PSM Diagram (by setting the property *IsAbstract* to true). This is taken into account in translation. Abstract classes are covered in more detail in Part 3.4 describing translations of specialization. In short – if the class is translated into a `xs:complexType` declaration, attribute `abstract="true"` is used for the complex type. `xs:element` declarations are not created for abstract class (with some exceptions described in Part 3.4). When the class is translated into groups, references to the groups are not propagated into containing complex type (again with the same exceptions).

*Translation of a class is executed in method **XmlSchemaTranslator.TranslateClass**, which calls either **XmlSchemaTranslator.TranslateClassWithLabel** when the class has an element label or method **XmlSchemaTranslator.TranslateClassContentToGroups** when the class has not an element label.*

Translation of class attributes and contents
In the following list, we describe how the attributes and content of *U* are translated. The content of *U* is composed of associations going to classes or class unions, content choices, attribute containers and content containers. The following list describes the translation of all these constructs. (4) is related to the attributes of *U*. The other items are related to the content of *U*.

(4) Attribute of a class

A simple attribute *A* with alias *NA* and type *domA* of *U* is translated to an attribute declaration

```
<xs:attribute name="NA" type="domA" />
```

Attributes can have their multiplicity properties Lower and Upper defined. But declaring multiplicity of attributes in XML documents is restricted to `use="optional"` and

use=”required” definitions. Thus, if Lower is set to 0 and Upper to 1 *A* is translated as follows:

```
<xs:attribute name="NA" type="domA" use="optional" />
```

If both Lower and Upper are set to 1 *A* is translated into:

```
<xs:attribute name="NA" type="domA" use="required" />
```

When either Lower or Upper values are greater than 1 a warning is put into translation log (because the same attributes cannot occur more than once in an element declaration in XML documents).
When both Lower and Upper are set to 0, the attribute is not translated.
Attributes can also have their Default property specified. If Default property of *A* is set to “value” then *A* is translated as follows:

```
<xs:attribute name="NA" type="domA" default="value" />
```

It is up to the user to set correct default values to generate a valid schema, XCase does not check the default value (it can be set to an arbitrary string) and it is always translated as is.
When both Default value and multiplicity are specified and multiplicity is set to "required", the use attribute is omitted and *A* is translated into:

```
<xs:attribute name="NA" type="domA" default="value" />
```

because use="required" and default="value" would not make sense (and is forbidden in XML Schema).
Note: there is a departure from the rules for multiplicity and that is when the attribute is under content choice or class union but should be propagated to a class or content container above the content choice resp. class union. This exception is described in Part 4.2 – Attributes under choice constructions.

XmlSchemaTranslator.TranslateAttributesIncludingRepresentative *method translates attributes of a class as described above. It also appends attribute group references for attributes propagated from classes beneath the translated class.*

(5) Attribute container

An attribute container that is composed of attributes $A_1 \dots A_k$ among the components of *U* is translated to a sequence

$XSA_1 \dots XSA_k$

where XSA_i is the following element declaration translated from the simple attribute A_i with alias NA_i and type $domA_i$:

```
<xs:element name="NAi" type=" domAi " />
```

Rules for translation of Default property are similar as for attributes in classes. If Ai has multiplicity properties Lower set to L_i and Upper to U_i, they are translated using minOccurs and maxOccurs declarations:

```
<xs:element name="NAi" type="domAi"
      minOccurs="Li" maxOccurs="Ui"/>
```

When Upper is set to value **UnlimitedNatural.Infinity**, it is translated into maxOccurs="unbounded" declaration.

XmlSchemaWriter.TranslateAttributeContainer *writes element declarations of all the attributes in the attribute container.*

(6) Association going to a class

Let E be an association going from U to class V where m and n are values assigned to cardinality properties Lower and Upper of the association. U can be another class, content choice, content container or class union.

- If V has an element label lV, E is translated to an element declaration

  ```
  <xs:element name="lV " type="TNV"
        minOccurs="m" maxOccurs="n" />
  ```

  TN_V is the name of the global complex type to which class V is translated.

- If V does not have not an element label, E is translated to a model group reference

  ```
  <xs:group ref="TNV -c"
        minOccurs="m" maxOccurs="n" />
  ```

  the model group reference is propagated to a nearest complex type above V using *TranslationContext.ComposedContent* writer. The group declaration is written when class V is translated.

XmlSchemaWriter.TranslateAssociationChild*(V) calls* ***TranslateClass(****V****)*** *and after V is translated, it writes the element declaration when V has an element label. When V does not have an element label, the group reference is written during the translation of components of V using* ***TranslationContext.ComposedContent*** *writer.*

(7) Association going to a class union

Let E be an association going from U to a class union with components $V_1, \ldots, V_n$. It is translated to a `xs:choice` content model

```
<xs:choice minOccurs="m" maxOccurs="n">
      XSE V1 ... XSE Vn
</xs:choice>
```

where m and n are values assigned to cardinality properties Lower and Upper of the association E and XS^E_{Vi} is an element declaration

```
<xs:element name="lVi" type="TNVi" />
```

if V_i has an element label lV_i, or a model group reference

```
<xs:group ref="TNVi-c" />
```

if V_i does not have an element label.

XmlSchemaTranslator.TranslateAssociationChild *writes translation of a class union and calls* ***TranslateAssociationChild*** *for each of the components* $V_1, \ldots, V_n$.

Both class union and content choice enter ***"choice context"****. Specifics of the translation when in choice context are described in Part 4.2 – Attributes under choice constructions.*

(8) Content container

Let *C* be a content container with a name *lC* among components of *U*. It is translated to an element declaration

```
<xs:element name="lC">
    <xs:complexType>
        XS^E_C
        XS^A_C
    </xs:complexType>
</xs:element>
```

XS^E_C is the translation of the content of *C*. The translation of the content of *C* is performed in the same way as a content of a class, i.e. it is a `xs:sequence` containing translations of the components from the content of *C* as described by (5) - (9). XS^A_C is a set of references to attribute groups translated from the child nodes of *U* that are contained in *C* and do not have an element label.

XmlSchemaWriter.TranslateContentContainer *writes translation of the content container. This method creates new* ***TranslationContext*** *which is passed to the translation of the Components. Using this new context, all propagated attribute group references and group references are included in the translation of the content container.*

(9) Content choice

A content choice with components $C_1 \ldots C_n$ is translated to a `xs:choice` content model

```
<xs:choice>
  XS_C1 ... XS_Cn
</xs:choice>
```

where XS_{Ci} is the translation of C_i as described by (5)-(9).

XmlSchemaTranslator.TranslateContentChoice *writes translation of the content choice. If there are any classes without element labels among the components of the content choice or beneath them (but not beneath en element that is translated to a complex type), references to groups and attribute groups are written in the* ***TranslationContext.ComposedContent*** *and* ***TranslationContext.ComposedAttributes*** *writers (so the references are passed to the first element translated to a complex type above the content container).*

Both class union and content choice enter ***"choice context"****. Specifics of the translation when in choice context are described in Part 11.4.2 – Attributes under choice constructions.*

11.3.3 Translation of structural representatives

PSM Diagrams allow structural representative construct. Let there be PSMClass *U* and its *PSMClass.RepresentedPSMClass* property is assigned a reference to another class *V*. Now *U* has all the attributes and content of *V* and also can have some content of its own. The whole structural representative concept is meant to allow several PSM Classes representing one PIM Class without repeating the whole definitions of their contents types. This concept is also very beneficial when defining recursive structures in XML documents.

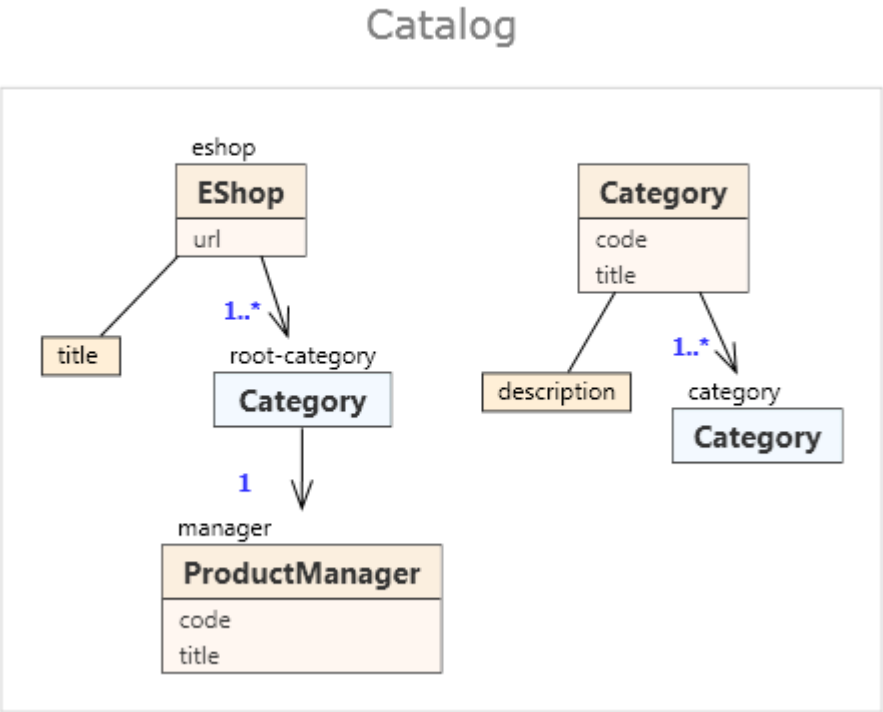


Figure 25 - Structural representatives used to defined recursive structure

Concept of structural representatives could be translated via several methods. XmlSchemaTranslator uses attribute groups and model groups which is more flexible than for example using `xs:extension` context (with attribute groups and model groups it is possible to propagate content and attributes to a complex type above, like it was described in Part 3.2). Let U be a structural representative of V. If V does not have an element label, it is translated according to Part 11.3.2 (2) to a model and attribute group. If it is does have an element label, it should be translated into a complex type according to section 11.3.2 (1). Because for the translation of a structural representative we need the represented class to be translated into model and attribute group, the pattern for translating V must be modified a little bit – so if V is a class with an element label and it is being referenced from a structural representative, it is not translated into this definition:

```
<xs:complexType name="TN_V">
    XS^E_V
    XS^A_V
</xs:complexType>
```

But rather to

```
<xs:complexType name="TN_V">
      <xs:sequence>
            <xs:group ref="TN_V-c" />
      </xs:sequence>
      <xs:attributeGroup ref="TN_V-a" />
</xs:complexType>
<xs:group name="TN_V-c">
      XS^E_V
</xs:group>
<xs:attributeGroup name="TN_V-a">
      XS^A_V
</xs:attributeGroup>
```

where XS^E_V and XS^A_V denote the translations of the content and attributes of V_i, respectively. With the algorithm altered like that it is ensured that V is translated into a model group and attribute group no matter whether V has an element label or not.

The altered translation of a PSMClass without label to groups is performed by ***XmlSchemaTranslator.TranslateClassWithLabelAsGroups*** *method.*

U itself is translated as follows. The content $C_1, .., C_n$ of U extends the content of V. It is therefore translated to a `xs:sequence` content model

```
<xs:sequence>
      <xs:group ref="TN_V-c">
      XS_C1 ... XS_Cn
</xs:sequence>
```

where XS_{Ci} denotes the translation of C_i. The `xs:sequence` is denoted XS^E_U. The attributes $A_1, ..., A_m$ of U extend the propagated attributes of $V_1, ..., V_k$ and are translated to the sequence

```
<xs:attributeGroup ref="TN_V-a">
XS_A1 ... XS_Am
```

where XS_{Ai} denotes the translation of A_i. The sequence is denoted XS^A_U. For each child V of U without an element label and not contained in a content container, XS^A_U is extended with the reference to the attribute group translated from V (if there is any).
The rest of the translation of U is the same as in the case of classes that are not structural representatives. This is described in Part 11.3.2 (1) and (2). It means that if U has an element label then XS^A_U and XS^E_U are included in the resulting XML Schema as a global complex type definition. Otherwise they are included as a model and attribute group.

Methods ***XmlSchemaTranslator.TranslateComponentsIncludingRepresentative*** *resp.* ***XmlSchemaTranslator.TranslateAttributesIncludingRepresentative*** *are both "structural representative aware" and thus if class being translated is a structural representative then if the*

represented class was not already translated, it is translated immediately, and then the contents resp. attributes of the translated class are translated into the sequences described above.

11.3.4 Translations of generalizations

PIM Diagrams allow defining generalizations and PSM Diagrams allow bringing these generalizations into the PSM level. XML Schema language has adequate constructions to allow translations of these relationships into XML Schema. There are two approaches used that follow the principles described in Part 11.3.2 (1) and (2).
Let U be a node specialized by nodes $V_1, \ldots, V_n$. $V_1, \ldots, V_n$ are translated as follows.

(10) U has an element label lU.

U is translated to a complex type definition TN_U as described in Part 11.3.2 (1). Moreover, if U is abstract, the complex type definition has set the parameter abstract to true. Each V_i is translated to a complex type definition

```
<xs:complexType name="TNVi">
    <xs:complexContent>
        <xs:extension base="TNU">
            XSEVi
            XSAVi
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

where XS^E_{Vi} denotes the translation of the content of V_i and XS^A_{Vi} denotes the translation of the attributes of V_i as described in Part 11.3.2. The complex type TN_{Vi} extends TN_U which corresponds to the semantics of the specialization of class U by class V_i.

(11) U does not have an element label.

U is not translated to a complex type definition but to model group TN_U-c and attribute group TN_U-a as described in Part 11.3.2 (2). Therefore, we cannot use the `xs:extension` construction (it can be applied only on complex types). Model and attribute groups are used again.
If V_i does not have an element label as well, it is translated to a model group and attribute group

```
<xs:group name="TNVi-c ">
      <xs:sequence>
            <xs:group ref="TNU-c" />
            XSEVi
      </xs:sequence>
</xs:group>
<xs:attributeGroup name="TNVi-a">
      <xs:attributeGroup ref="TNU-a" />
      XSAVi
</xs:attributeGroup>
```

If V_i has an element label lV_i, it is translated to a global complex type definition

```
<xs:complexType name="TNVi">
      <xs:sequence>
            <xs:group name="TNU-c" />
            XSEVi
      </xs:sequence>
<xs:attributeGroup name="TNU-a" />
      XSAVi
</xs:complexType>
```

To fully express the semantics of specializations, generalized classes must be substitutable by the specialized classes. This comes up in these occasions

- Global elements – in section Part 11.3.2 (1) it was stated that each class from PSM Diagram's Roots collection that is not abstract and that has an element label adds a global element declaration. For classes that specialize root this has to be extended.
- Element declarations that are a result of association being translated.
- References to groups that are propagated from a general class without an element label.

(12) Substitutions of roots

If U is a root with type name TN_U and element label lU. If U is not an abstract class, then

```
<xs:element name="lU" type=" TNU" />
```

global declaration is written. To satisfy substitutability, the same element declaration is created even when there is some non-abstract class C whose ancestor is U (not necessarily parent) and C *has not* an element label and there are no classes with element label between C and U in the inheritance hierarchy.

This condition can be checked using extension method ***PSMClass.CanBeDeclaredAsElement()*** *defined in namespace* ***XCase.Translation.XmlSchema****.*

Let U be a node specialized by nodes $V_1, \ldots, V_n$. If V_i is a specialization of U and V_i is not abstract and has an element label lV_i which is different from lUi a global element declaration:

```
<xs:element name="lVi" type="TNVi" />
```

is written. The complex type TN_{Vi} is declared because V_i has the element label. If U has an element label lU, a global element declaration

Because of (10), each V_i is translated to a complex type definition TN_{Vi} extending TN_U using `xs:extension` construct. Because of the semantics of `xs:extension`, the element lU with the type TN_U can also have a type inherited from TN_U. Therefore, the element declaration lU does not represent only the node U in the resulting XML schema but each V_i that does not have its own element label and therefore inherits lU from U.

Substitutions in associations and class unions

If U is not a root and there is an association E going from a class U_0 to U, E must be translated in a different way then we described in Part 11.3.2 (6) and (7). Instead the element declaration created by (6), the `xs:choice` content model

```
<xs:choice minOccurs="m" maxOccurs="n">
```

$XS_E\, XS_{E(V1)} \ldots XS_{E(Vk)}$

```
</xs:choice>
```

is created where $(m; n)$ is the cardinality of U_0 in E. If U does not have an element label and is abstract, XS_E is empty. Otherwise, XS_E is the result of the translation of E according to (5). If V_i has an element label lV_i, $XS_{E(Vi)}$ is an element declaration

```
<xs:element name="lVi" type="TNVi" />
```

Otherwise, $XS_{E(Vi)}$ is a model group reference

```
<xs:group ref="TNVi-c" />
```

These substitutions apply recursively (if V_i is specialized by W, substitutions for W is translated in a similar way as for V_i).

If V_i was translated to model group TN_{Vi}-c, the referenced is placed inside the choice declaration. But if there also is an attribute group TN_{Vi}-a, there is not a convenient place where to put this reference. Thus TN_{Vi}-a is not referenced during translation of U (reasons for this are given in Part 11.4.3 – Specialized classes without element labels).

Translation of a class union (described in Part 11.3.2 (7)) is modified in a similar way as translation of association.

*Method **XmlSchemaTranslator.TranslateSubstitutions**, which is called for each specialized class, tests conditions mentioned above and translates substitutions for each specialized class (recursively).*

11.3.5 Translation of simple types

XCase allows the user to define custom types on the PIM level. Type of an attribute on PIM level is propagated to PSM level. XML Schema language provides constructions to define custom attribute types and these have to be used for translation of a PSM Diagram into a XML schema. XCase supports deriving new simple types by restriction of built-in XML Schema data types and other already defined simple types. The rules for the restriction are entered by user into the property *SimpleDataType.DefaultXSDImplementation*. The translation algorithm considers string value of *SimpleDataType.DefaultXSDImplementation* to generate a simple type declaration.

Following properties of SimpleDataType are taken into account when generating the declaration:

- *Name* – for naming the type
- *Parent* – to use as the restriction base
- *DefaultXSDImplementation* – the actual restriction

Following declaration is generated for simple data type with values of Name, Parent and *DefaultXSDImplementation* equal to "*STName*", "*STParent*", "*STXSD*":

```
<xs:simpleType name="STName">
    <xs:restriction base="xs:STParent">
        STXSD
    </xs:restriction>
</xs:simpleType>
```

The value of *DefaultXSDImplementation* is pasted into declaration as is (only xml well-formedness is checked).

11.4 Limitations of XML Schema translation

PSM Diagram describes certain set of XML documents. Constructions of XML Schema language are not always sufficient enough to cover all possible constructions available in PSM Diagrams. On the other hand, there are some constructions provided by XML Schema language that cannot be achieved by PSM Diagrams. This section is devoted to these incompatibilities.

11.4.1 Mixed content

It is possible to define complex types with mixed content in XML Schema language. Mixed content is useful for "document oriented" XML documents (like XHTML pages) but less useful for documents describing data and since XCase is oriented to data modeling, mixed content is not supported

11.4.2 Attributes under choice constructions

XML Schema language does not allow any kind of "choice between attributes". Attribute declarations are not allowed in `xs:choice` declarations and the only instrument to control occurrence of attributes in XML Schema are `use="optional"` and `use="required"` attributes.

Consider following XML Diagram:

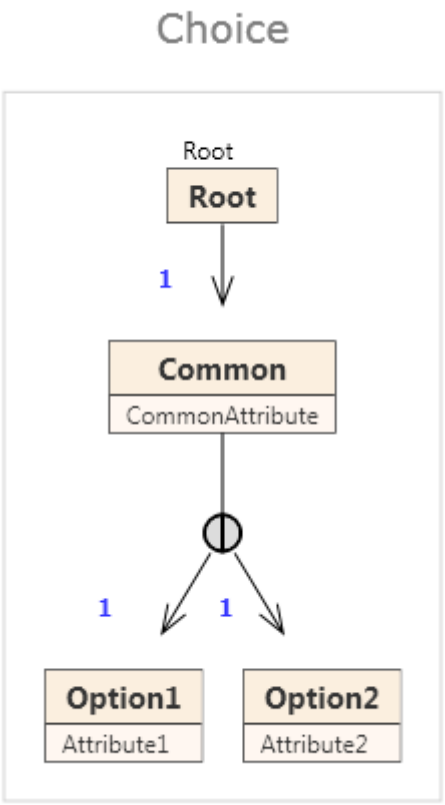


Figure 26 - Choice

XML documents modeled by this diagram will look like this:

```
<Root>
<Common CommonAttribute="..." Attribute1="... />
</Root>
```

or this:

```
<Root>
<Common CommonAttribute="..." Attribute2="... />
</Root>
```

However, such a construction cannot be expressed by XML Schema language, but could be expressed in other languages describing structure of XML documents (i.e. Schematron [7]). Basic translation algorithm described in Part 11.3.2 disturbs the semantic meaning of the diagram – according to Part 11.3.2 this diagram would be translated to following schema:

```
<xs:complexType name="Root">
     <xs:attributeGroup ref="Common-a" />
</xs:complexType>

<xs:attributeGroup name="Common-a">
     <xs:attribute name="CommonAttribute"
          type="xs:string" use="required" />
     <xs:attributeGroup ref="Option1 " />
     <xs:attributeGroup ref="Option2 " />
</xs:attributeGroup>

<xs:attributeGroup name="Option1 ">
     <xs:attribute name="Attribute1"
          type="xs:string" use="required" />
     </xs:attributeGroup>

     <xs:attributeGroup name="Option2 ">
     <xs:attribute name="Attribute2"
          type="xs:string" use="required" />
</xs:attributeGroup>
```

This is very different from the semantic meaning of the diagram, because all the three attributes in element Root are now required. The better solution is to declare only CommonAttribute as required and Attribute1 and Attribute2 as optional. This would not prevent the situation where all the three attributes are defined in Root element or only CommonAttribute is defined, but at least all documents valid according to the desired descriptions are valid according to the description in XML Schema. The additional check against using both attributes at once would have to be performed by other means [7].

The translation described in Part 11.3.2 is therefore changed to support the correct declarations. XML Schema does not support `use=″optional″` and `use=″required″` on attribute groups, only on attributes. This obstacle leads to introducing “opt-groups” that are created each time an attribute group needs to be referenced from a choice context. Translation is in choice context (see fields *inChoice* and *choiceCounter* in **XmlSchemaTranslator**) if it is translating elements between content choice or class union (that are translated to `xs:choice`) and nearest content container or class with element label (that are translated to a `xs:complexType`).

If the translation is in choice context, attributes in classes without element labels are translated into opt-groups. The diagram above will be translated into following schema:

```
<xs:complexType name="Root">
     <xs:attributeGroup ref="Common-a" />
</xs:complexType>

<xs:attributeGroup name="Common-a">
     <xs:attribute name="CommonAttribute"
          type="xs:string" use="required" />
     <xs:attributeGroup ref="Option1-a-opt" />
     <xs:attributeGroup ref="Option2-a-opt" />
</xs:attributeGroup>

<xs:attributeGroup name="Option1-a-opt">
     <xs:attribute name="Attribute1"
          type="xs:string" use="optional" />
     </xs:attributeGroup>

<xs:attributeGroup name="Option2-a-opt">
     <xs:attribute name="Attribute2"
          type="xs:string" use="optional" />
</xs:attributeGroup>
```

The drawback of opt-groups is that in some diagrams there have to be created both versions of an attribute group – standard version and opt-group differencing only in the `use` declarations.

When the attribute group is translated, only the version that is currently needed is written (opt or normal) by the method XmlSchemaTranslator.TranslateAttributesIncludingRepresentative. If the attribute group needs to be referenced as the other version, it is translated by calling XmlSchemaTranslator.TranslateAttributeGroupsAgain. The versions already created are tracked in ClassTranslationData.AttributeGroupUsage field (that can have values None, Optional, Normal and Both). Also a warning is written to Log when an opt-group is translated, because this is only a work-around that does not ensure perfect semantic correctness of the XML document being validated.

11.4.3 Specialized classes without element labels

PSM Diagrams allow declaring both general and specific classes without an element label. In that case both classes are translated into model and attribute groups (xs:complexType and xs:extension are not used here). Trouble comes when there is an edge going to the general class and attribute group of the specific class is not empty, because having an association going to a general class should allow the general class to be substituted by specialized class. The situation is a bit similar to the situation described in previous section, but is a bit trickier.

Consider following diagram:

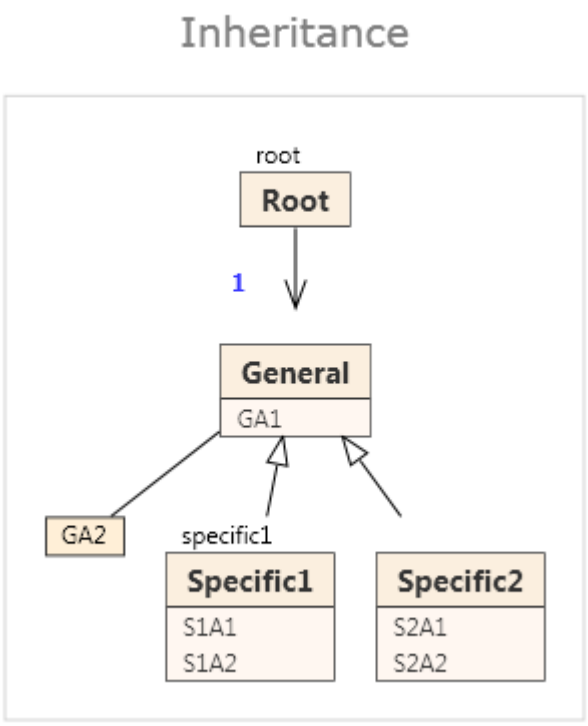


Figure 27 - Inheritance problem

The diagram contains classes General and Specific2 that have no element labels. They are translated into groups and groups belonging to Specific2 reference the groups belonging to General. General should be always substitutable by Specific2. The diagram describes XML documents that look like:

```
<root GA1="...">
      <GA2>...</GA2 >
</root>
```

or

```
<root GA1="...">
      <GA2>...</GA2 >
      <specific S1A1="..." S1A2="..."/>
</root>
```

or

```
<root GA1="..." S2A1="..." S2A2="...">
      <GA2>...</GA2>
</root>
```

But such a set of XML documents can hardly be expressed in XML Schema. One problem is that the first and third options are again choosing between two possible ways of declaration of attributes. This is similar to the problem described in Part 11.4.2 but trying to solve it again by creating opt-groups fails, because complex type root would result in following translation:

```
<xs:complexType name="Root">
      <xs:sequence>
      <xs:choice>
            <xs:group ref="General-c" />
            <xs:element name="specific1" type="Specific1" />
            <xs:group ref="Specific2-c" />
            </xs:choice>
      </xs:sequence>
      <xs:attributeGroup ref="General-a-opt" />
      <xs:attributeGroup ref="Specific2-a-opt" />
</xs:complexType>
...
<xs:attributeGroup name="General-a-opt">
      <xs:attribute name="GA1" type="xs:string" use="optional" />
</xs:attributeGroup>

<xs:attributeGroup name="Specific2-a-opt">
      <xs:attributeGroup ref="General-a-opt" />
      <xs:attribute name="S2A1" type="xs:string" use="optional" />
      <xs:attribute name="S2A2" type="xs:string" use="optional" />
</xs:attributeGroup>
```

and there are many problems with this translation:

- there is no check whether Specific2-c and Specific1-a-opt are used together in the document
- attributes in General-a-opt and Specific1-a-opt can be declared both ore none of them can be declared (this is similar to translation with choice context)
- attributes in General-a-opt are not only included in Root element but also in element specific1, because they are part of the content of Specific1 type
- attribute GA1 is in both groups – in General-a-opt, because it is declared there, and in Specifc2-a-opt because it references General-a-opt. The schema above is thus not only semantically inaccurate, but even invalid.

To overcome the last issue the groups created from specialized classes would have to not reference the groups from general classes (in schema above <xs:attributeGroup ref="General-a-opt" /> would be omitted from Specific2-a-opt. Then the schema above would be valid (although the first two problems would still remain) and would described a superset of XML Documents that were described by the PSM Diagram.
But there are new issues coming from this workaround. The Specific2-a-opt is now not a superset of General-a-opt, they are distinct set of attributes in fact. If Specific2-a-opt were referenced from some other class in the diagram via the structural representative construct described in Part 11.3.3, the attributes from General would not be included in the

representative's translation which is again semantically incorrect. The groups would have to be translated again using the original algorithm where the general groups are referenced from specific groups. And it mustn't be forgotten that the class can be referenced from a structural representative in two ways – from choice context or normally.
In the end there could be up to three translations of each attribute group (and since attribute groups can be nested and each of the nested groups would have to be translated in these three ways, the resulting schema would be extremely unclear).
That is why the attribute groups belonging to specialized classes are not referenced from translations of general classes, moreover situation can be simply solved by assigning an element label to the specialized class or moving the attributes of the specialized class into an attribute container (so that they are translated into `xs:elements`). If there is such a construction in the diagram, warning is written in Log.

11.4.4 Non-deterministic diagrams

It is possible to assign the same element labels to two associations going from one class like in the following diagram:

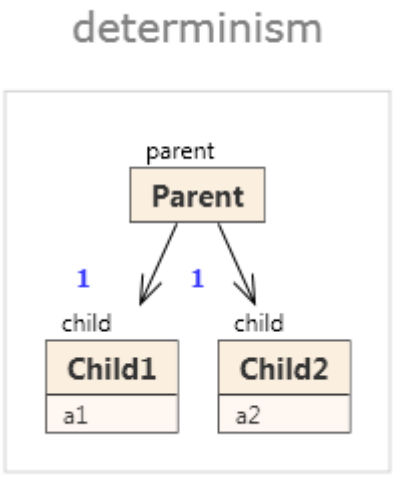


Figure 28 - Determinism

The diagram describes XML documents looking like:

```
<parent>
     <child a1="..." />
     <child a2="..." />
</parent>
```

But XML Schema language does not allow declaring two elements of the same name and different type in the same scope. Such a content is in general nondeterministic (the example above is deterministic, but when the multiplicities of associations are changed for example to 1..3 and aliases of both the attributes were the same, it would truly be nondeterministic).
XCase translation algorithm does not solve the problem and the solution is left up to the user – he is given an error when he tries to validate the generated XML Schema.
Nondeterministic schema can occur in more structures (for example when there is some content declared in two model groups and when the groups are both referenced in a complex type, the result is nondeterministic).
Another similar problem occurs when there are two attributes with the same name defined in different attribute groups or types and referenced together in a type or attribute group. This

problem is also ignored by the translation and the user must change his diagram to get a valid schema as a result.

11.4.5 Not package-aware

The translation algorithm does not consider packages and the whole schema is placed in one namespace common for the project. Creating XML namespaces according to package hierarchy cannot be used, because PSM Diagram can contain classes from several packages and the translated schema has to declare the derived types – and one schema cannot declare elements in multiple packages.

11.4.6 Multiplicity of attributes is discarded

As described in Part 11.3.2 (4), XML Documents cannot contain multiple instances of the same attribute in one elements, therefore only 0..0, 0..1 and 1..1 multiplicity specifications can be considered by the translation. Attributes can be moved to an attribute container, where their multiplicity can be fully expressed using `minOccurs` and `maxOccurs` attributes of element declaration.

References

[1] nUML project page on sourceforge.net
http://numl.sourceforge.net

[2] State design pattern.
E. Gamma, R. Helm, R. Johnson, J. M. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software.

[3] Original DockingLibrary webpage -
http://www.codeproject.com/KB/WPF/WPFdockinglib.aspx

[4] W3C, XML Schema Part 0: Primer Second Edition, October 2004,
http://www.w3.org/TR/xmlschema-0/

[5] Relax NG, a schema language for XML.
www.relaxng.org/

[6] M. Necasky: Conceptual Modeling for XML. Ph.D. thesis.
Faculty of Mathematics and Physics, Charles University, Prague. May 2008.
http://www.necasky.net/thesis.pdf

[7] Schematron, a rule-based validation language.
http://www.schematron.com/