UML Mapping to IDL

1 Introduction

This document describes the main elements of the UML to IDL mapping implemented by the Enterprise Architect IDL4 Add-in.

2 Mapping rules

Please refer to Figure 1 below for an example of how the Basic rules are applied.

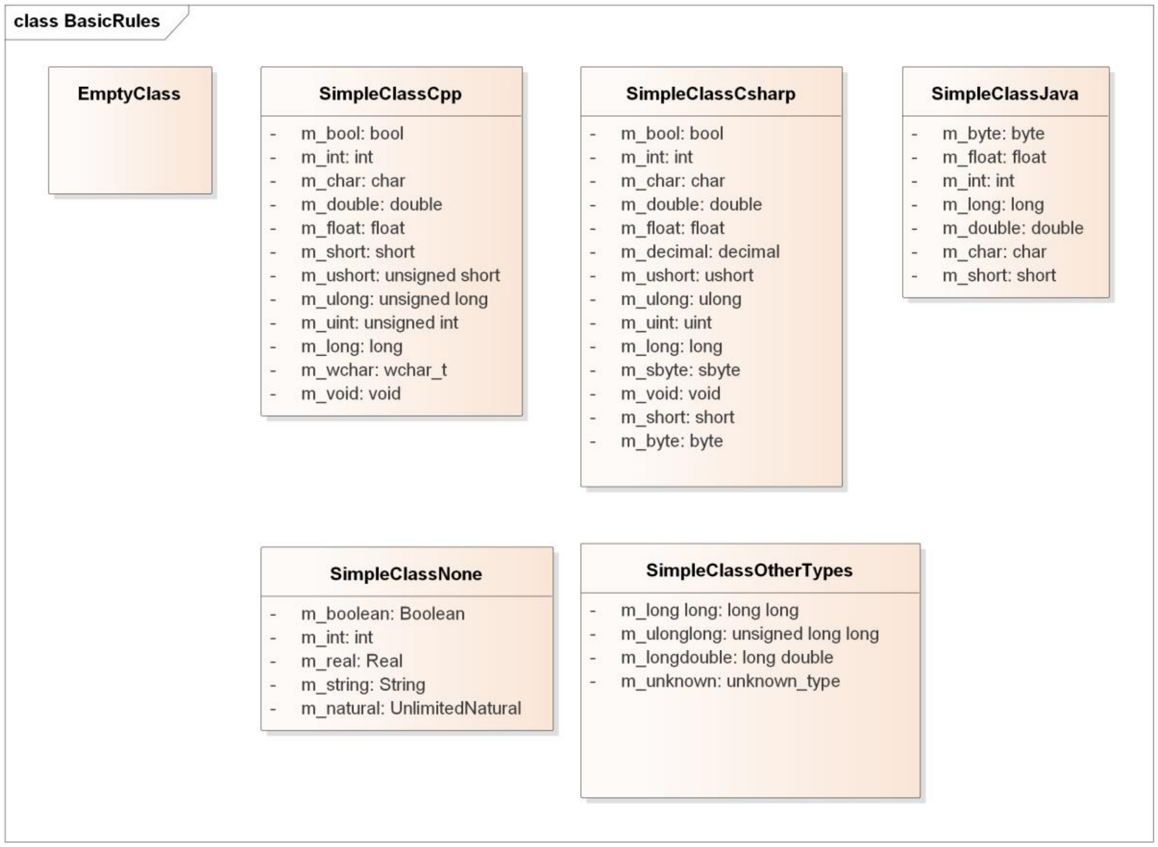


Figure 1. *Example Model for basic rules.*

2.1 Classes

Classes are mapped to IDL4 structures. Note that IDL4 structures support single inheritance.

2.2 Empty classes

IDL does not support empty classes. If a UML class is empty the generated structure gets a dummy member of type octet. For example the class named `EmptyClass` in Figure 1 is mapped to the following IDL:

```
struct EmptyClass {
    octet __dummy_prevent_empty_class_EmptyClass;
};
```

The class name (`EmptyClass` in the example) is appended to the member name to ensure member names are unique even in the case of inheritance from empty base classes.

2.3 Class attributes

UML Class attributes appear as IDL structure members of the same name.

2.3.1 Primitive class attributes

The primitive types offered by Sparx Enterprise Architect depend on the "Programming Language" that has been defined for the Class. To ease the mapping to IDL we recommend you select "C++" or "C#" since the types that Sparx Enterprise Architect will offer for those languages are the ones that most closely resemble the IDL4 types.

For example the class `SimpleClassCpp` shows the primitive types in the UML model when the programming language for the class is "C++". This class is mapped to the following IDL4 structure:

```
struct SimpleClassCpp {
    bool m_bool;
    long m_int;
    char m_char;
    double m_double;
    float m_float;
    short m_short;
    unsigned short m_ushort;
    unsigned long m_ulong;
    unsigned long m_uint;
    long m_long;
    wchar_t m_wchar;
    void m_void;
};
```

UML primitive types that appear in the model will fit various programming languages as shown in Table 1.

Table 1. *Mapping of UML primitive types*

Primitive Member Types in UML model	*Mapped IDL4 type*
"boolean", "bool"	boolean
"char"	char
"wchar", "wchar_t"	wchar
"octet", "byte", "sbyte", "int8", "int8_t", "uint8", "uint8_t"	octet
"short", "int16", "int16_t"	short
"long", "int", "int32", "int32_t", "integer", "decimal"",	long

`"unlimited natural".`	
`"long long", "int64", "int64_t"`	`long long`
`"unsigned short", "ushort", "uint16", "uint16_t"`	`unsigned short`
`"unsigned long", "unsigned int", "ulong", "uint", "uint32", "uint32_t"`	`unsigned long`
`"unsigned long long", "uint64", "uint64_t"`	`unsigned long long`
`"float", "float32", "number", "real"`	`float`
`"double", "float64"`	`double`
`"string"`	`string`
`"wstring"`	`wstring`

2.3.2 Arrays and sequences as members

Given a UML a class with a member `m_t` of type `T`, the corresponding IDL4 Structure may map it to a plain member of a type mapped from `T`—as described above—or to an array or sequence of the aforementioned type.

The decision on whether the type of the mapped IDL4 member is a simple type, an array, or a sequence is based on the UML class attribute multiplicity; specifically on the values of the `LowerBound` and `UpperBound`.

- If `LowerBound == UpperBound == 1` then the attribute "a" maps to a plain member: "T m;"
- If `LowerBound == UpperBound` and UpperBound is different to 1 then it maps to an array as in: `T m[UpperBound];`
- If `LowerBound == 0` AND `UpperBound == 1`, then it maps to a plain member annotated as `Optional` as in: `@Optional T m;`
- If `LowerBound < UpperBound`, then it maps to a sequence as on: `sequence<UpperBound, T> m`. Note that in this case the precise value of `LowerBound` has no consequence to the IDL4 mapping.
- If `LowerBound > UpperBound` the value of `LowerBound` is ignored. As if it had been zero. In this case the precise value of `LowerBound` has no consequence in the IDL
- If `UpperBound == 0` the value of `LowerBound` is ignored and the member maps to an unbounded Sequence: `sequence<T> m`.

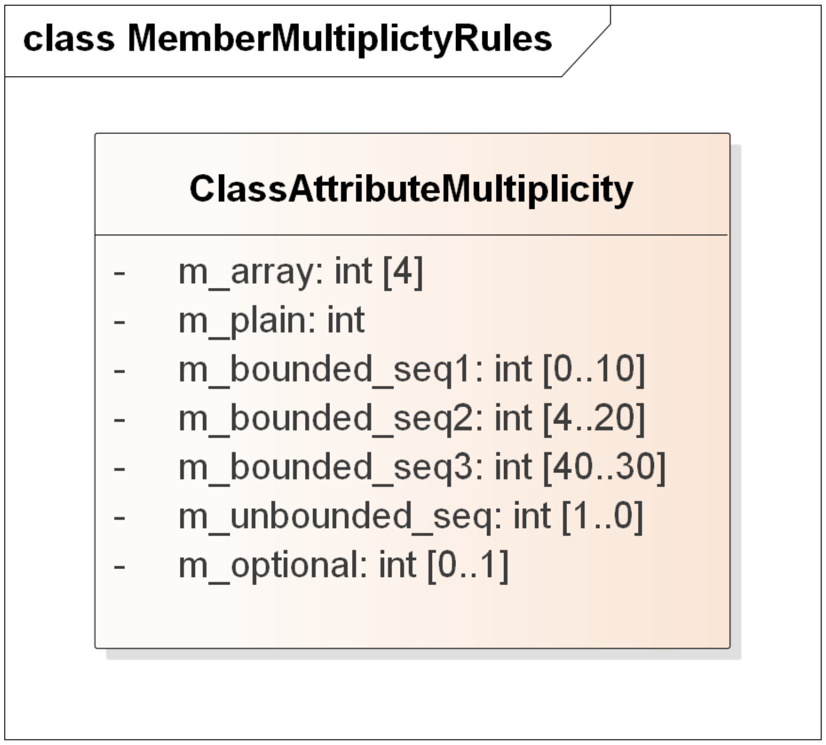

class MemberMultiplictyRules
ClassAttributeMultiplicity
- m_array: int [4]
- m_plain: int
- m_bounded_seq1: int [0..10]
- m_bounded_seq2: int [4..20]
- m_bounded_seq3: int [40..30]
- m_unbounded_seq: int [1..0]
- m_optional: int [0..1]

Figure 2. *Example Model with attributes having different multiplicity bounds.*

For example, the UML class `ClassMemberMultiplicity` shown in Figure 2 is mapped to the following IDL4 structure:

```
struct ClassAttributeMultiplicity {
    long m_array[4];
    long m_plain;
    sequence<long,10> m_bounded_seq1;
    sequence<long,20> m_bounded_seq2;
    sequence<long,30> m_bounded_seq3;
    sequence<long> m_unbounded_seq;
    long m_optional;  //@Optional
};
```

2.3.3 UML attribute tags values and IDL4 annotations

UML class attributes may have tagged values. The ones that are recognized are mapped to corresponding IDL4 annotations for the mapped member. Table 2 indicates the recognized UML tagged values and corresponding IDL4 annotation.

Table 2. *Mapping of UML attribute tagged values*

Attribute Tagged Values in UML model	*Mapped IDL4 annotation*	*Notes*
"key"	@key	
"optional"	@optional	
"must_understand"	@must_understand	
"id”	@id	

"value"	@value	Applies only to members of an enumeration
"nested"	long	
"external"	long long	
"unsigned short", "ushort", "uint16", "uint16_t"	unsigned short	
"unsigned long", "unsigned int", "ulong", "uint", "uint32", "uint32_t"	unsigned long	
"unsigned long long", "uint64", "uint64_t"	unsigned long long	

Despite being valid IDL4 annotations, the following tagged values are currently not recognized: `default`, `range`, `min`, `max`, `unit`, and `verbatim`.

2.3.4 UML Attribute properties and IDL @key

Some attribute properties affect the IDL4 mapping to the corresponding structure member. In section 2.3.2 we saw the impact of the UML attribute "multiplicity" property. In addition the attribute property "Is ID" also affects the generated member. If the "Is ID" is set to `true` then the generated IDL4 member will have the `@key` annotation. Note that the `@key` annotation may also be obtained using a UML Tag Value as described in section 2.3.3.

2.3.5 UML class operations

UML class operations are currently ignored in the IDL4 mapping.

3 UML Class tagged values and IDL Annotations

UML classes may have associated tagged values. The ones that are recognized are mapped to corresponding IDL4 annotations for the whole structure. Table 3 indicates the recognized UML tagged values and corresponding IDL4 annotation.

Table 3. *Mapping of UML class tagged values*

Class Tagged Values in UML model	*Mapped IDL4 annotation*	*Notes*
"extensibility"	@extensibility	
"mutable"	@mutable	
"extensible"	@extensible	
"final"	@final	
"autoid"	@autoid	
"nested"	@nested	

Despite being valid IDL4 annotations, the following tagged values are currently not recognized: `verbatim`, `service`.

4 Class inheritance

Class inheritance maps to IDL4 structure inheritance. Multiple structure inheritance is unsupported in IDL4 so only the “first” base class will appear as the IDL base structure.

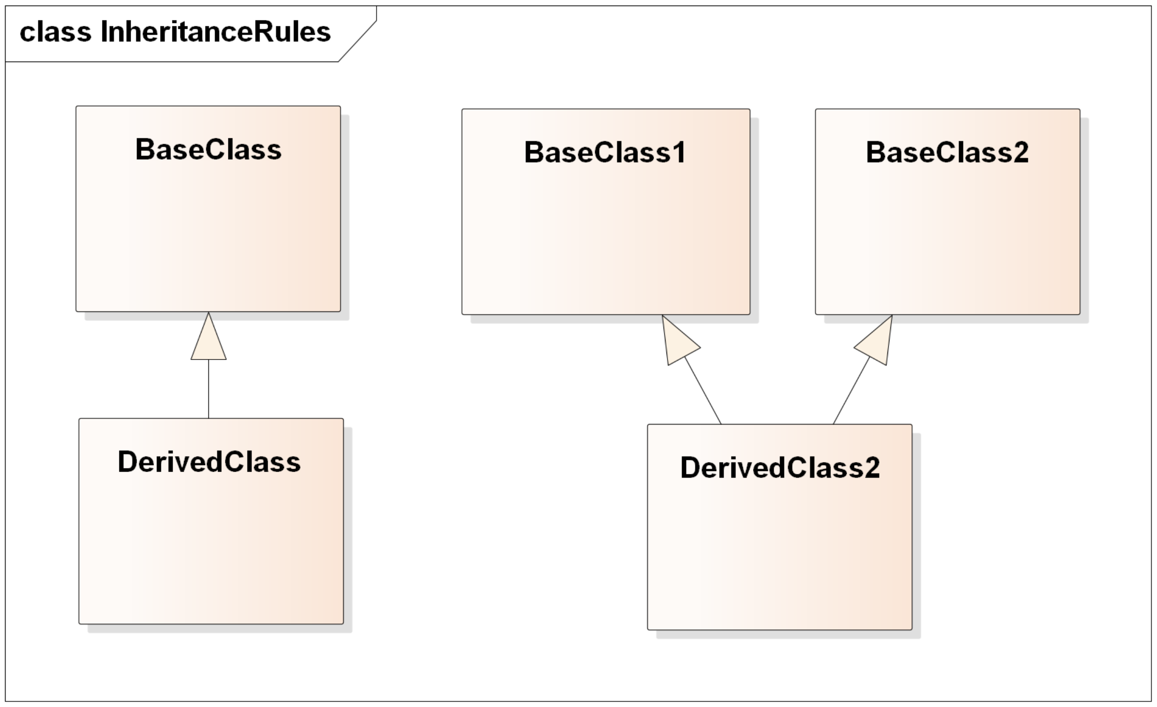


Figure 3. *Example Model with class inheritance.*

For example, the UML classes `DerivedClass` and `DerivedClass2` shown in Figure 2 above are mapped to the following IDL4 structures:

```
struct DerivedClass : BaseClass {
    octet __dummy_prevent_empty_class_DerivedClass;
};

struct DerivedClass2 : BaseClass1 {
    octet __dummy_prevent_empty_class_DerivedClass;
};
```

5 Order of the generated classes

In IDL and many programming languages like C and C++, a type must be declared before it is referenced by another type.

Given that the iterators on the UML model can return classes in any order, the IDL generator may have to iterate through the model multiple times, each time generating types that only depend on types that have already been generated.

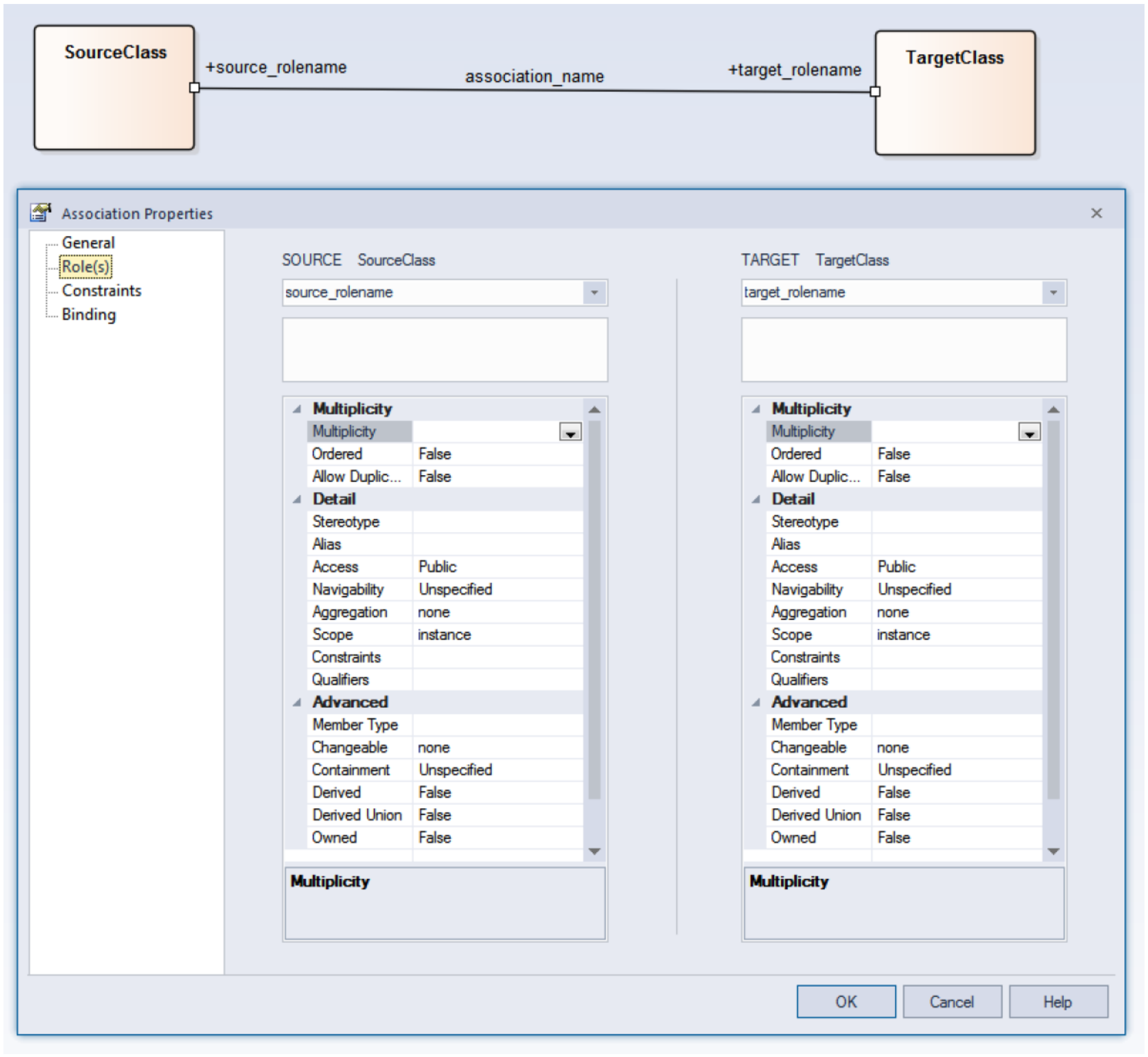
6 UML relationships between classes

Certain class relationships result in the generation of IDL4 structure members on the source or target IDL4 structure.

The decision of generating an IDL4 structure member as a result of the relationship, the name of the generated member, its characteristic (plain member, array, or sequence), and annotations depends on the relationship role properties.

6.1 Rules for generation of a structure member from a relationship

Please refer to Figure 4 below for an example of how the generation rules are applied. Assume class `SourceClass` has an association with class `TargetClass`. The association name is `association_name`. The Source role name is `source_rolename` and the target role name is `target_rolename`.



SOURCE SourceClass

source_rolename

TARGET TargetClass

target_rolename

	SOURCE	TARGET
Multiplicity		
Multiplicity		
Ordered	False	False
Allow Duplic...	False	False
Detail		
Stereotype		
Alias		
Access	Public	Public
Navigability	Unspecified	Unspecified
Aggregation	none	none
Scope	instance	instance
Constraints		
Qualifiers		
Advanced		
Member Type		
Changeable	none	none
Containment	Unspecified	Unspecified
Derived	False	False
Derived Union	False	False
Owned	False	False

Multiplicity

Multiplicity

Figure 4. *Example association*

All the following conditions must be met in for the association `source_rolename` to result on the generation of a member in the IDL4 structure `SourceClass`.

- The relationship must be of kind “Aggregation”, “Composition”, or “Nesting” (see Figure 5). In the case of an association it means the **Aggregation** must be either “Shared” or “Composite”.
- The association must be navigable from `SourceClass` to `TargetClass`. This can be seen in the value of the **Navigability** property set to “navigable” on the TARGET role (i.e., not “none” or “non-navigable”).
- The SOURCE role **Aggregation** property must be set to “shared” or “composite” (i.e., not “none”).
- The TARGET role **Containment** (see Advanced properties) must be “Value” as opposed to “Unspecified” or “Reference”.

Figure 5 shows how the different kinds of class relationships appear visually in Enterprise Architect:

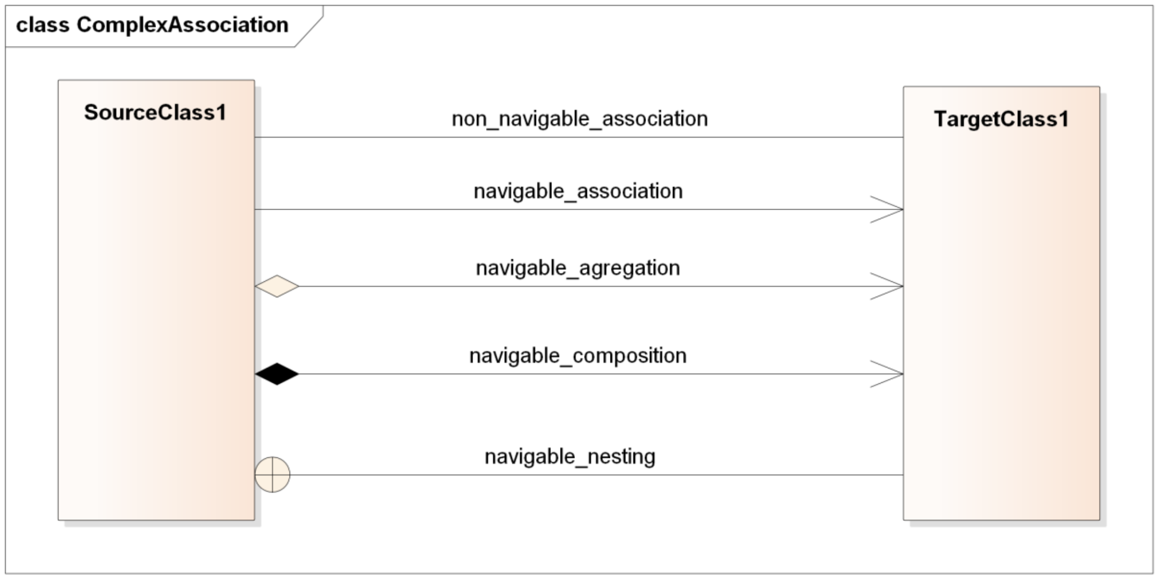


Figure 5. *Example associations: non-navigable association (Aggregation = “none”), navigable association, navigable aggregation (Aggregation=”Shared”), navigable composition (“Aggregation=”Shared”), and navigable nesting.*

Applying the rules to the associations that appear in Figure 5 results on `SourceClass1` having members for `navigable_aggregation`, `navigable_composition`, and `navigable_nesting`. It does not have members for the `non_navigable_association` and `navigable_association` because they are either non-navigable or the **Aggregation** property is “none”.

6.2 Name for members generated from associations

If the presence of an association between a SOURCE class and a TARGET class results in the generation of a member in the IDL4 mapping of the SOURCE class the generator must choose a name for the generated member. The name is chosen according to the following rules:

- If the TARGET role has been given a name, the use the TARGET rolename.
- Otherwise if the overall association has been given a name, use the association name.
- Otherwise if the typename of the TARGET class starts with upper case, use the TARGET typename but changing to starting with lowercase.
- Otherwise use the TARGET typename preceded by `m_`.

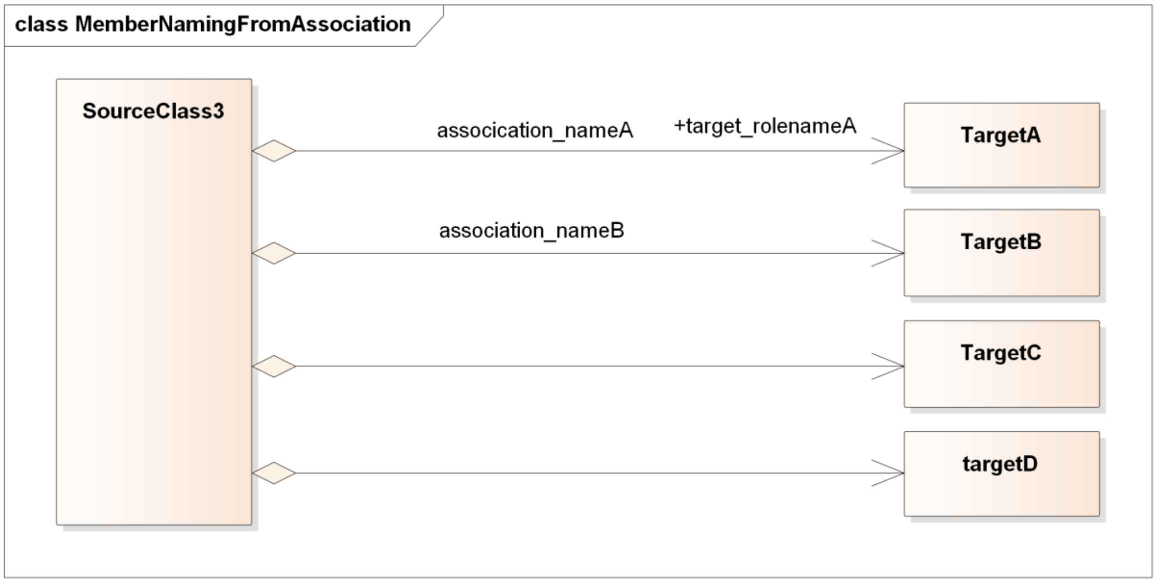


Figure 6. *The member name is determined by the association target rolename, the association name, or the name of the target class.*

For example, the relationship between the SOURCE class `SourceClass2` and the classes shown in Figure 6 are mapped to the following IDL4 structure:

```
struct SourceClass3 {
    TargetA*  target_rolenameA;
    TargetB*  association_nameB;
    TargetC*  targetC;
    targetD*  m_targetD;
};
```

6.2.1 Associations resulting on arrays and sequences as members

Similar to IDL structure members created from UML class attributes, the structure members created from class relationships can also appear as IDL4 `@Shared` members (pointers in RTI Connext DDS 5.2) or sequences.

The decision on whether the type of on the mapped IDL4 member `@Shared` or a sequence and the bounds of the sequence are based on the value of the **Multiplicity** Property of the TARGET Role of the association.

- If `Multiplicity == 1 or Multiplicity == 0`, or it is unspecified, then the association maps to a `@Shared` member.
- If `Multiplicity  == *` or it ends in `*` as in `1..*` or `..*`, then the association maps to an unbounded sequence.
- In other cases it maps to a bounded sequence with max size the upper limit of the multiplicity range.
    - In this case the lower limit is ignored
    - If the upper limit is set to `0` then it is treated as if it was `1`

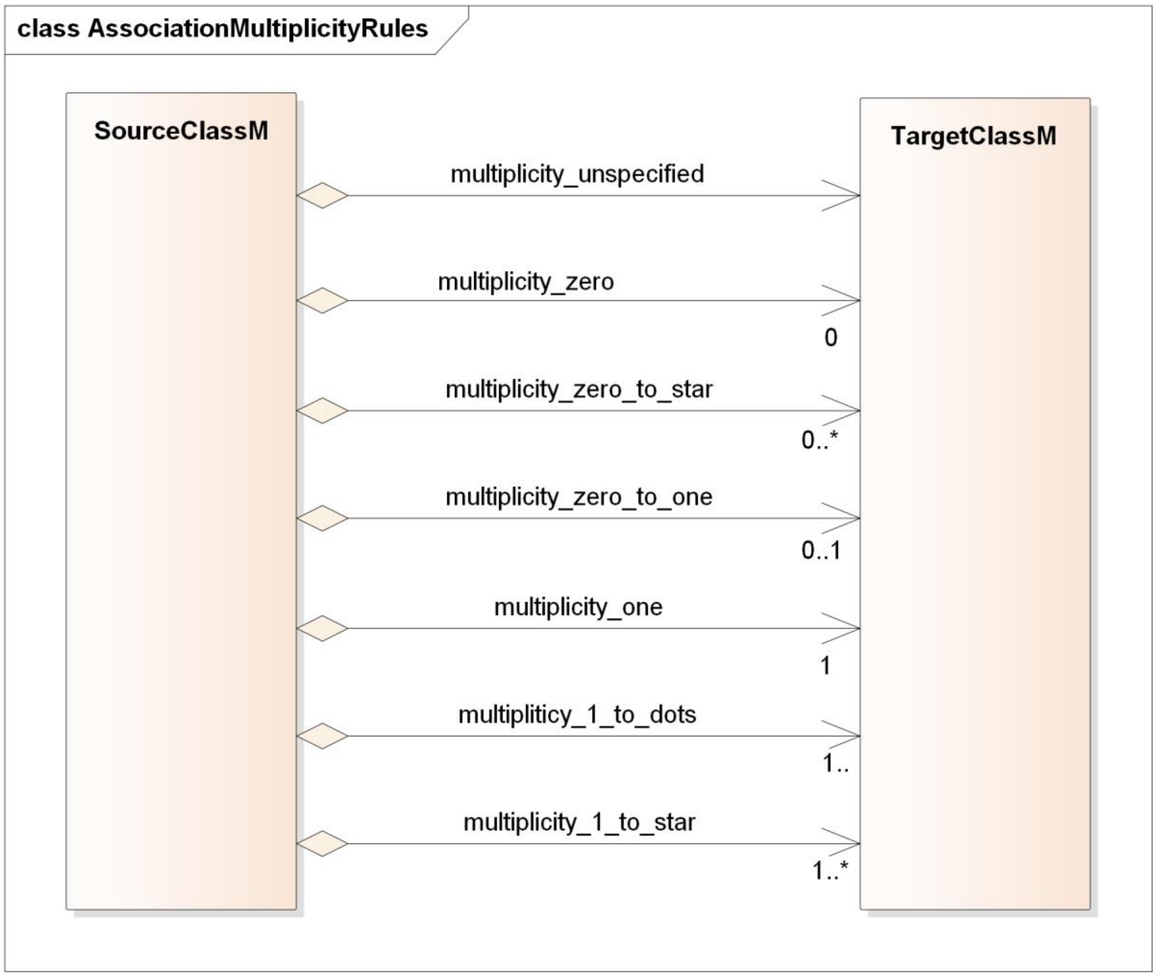


Figure 7. *The member name qualifiers (`@Shared` or sequence) is determined by the association target multiplicity.*

For example, the relationships between the SOURCE class `SourceClassM` and the TARGET classes shown in Figure 7 are mapped to the following IDL4 structure:

```
struct SourceClassM {
    @Shared  TargetClassM    multiplicity_unspecified;
    sequence<TargetClassM,1> multiplicity_zero;
    sequence<TargetClassM>   multiplicity_zero_to_star;
    sequence<TargetClassM,1> multiplicity_zero_to_one;
    @Shared  TargetClassM    multiplicity_one;
    sequence<TargetClassM>   multiplitiicy_1_to_dots;
    sequence<TargetClassM>   multiplicity_1_to_star;
};
```

7 Debugging the IDL4 generated from a model

One of the most useful ways to debug a model and modify it such that the generated IDL is what is expected is using the "Mapping Details" option menu.

At the bottom of the window that the IDL4 Extension opens there is an option pull down that allows the selection between: "Mapping Details – Suppressed", "Mapping Details – Basic", and "Mapping Details – Full".

By default, the menu is selecting "Mapping Details – Suppressed", which omits any details on the rules applied from the output. If "Mapping Details – Basic" or "Mapping Details – Full" is selected, the output IDL will contain in-line comments explaining the reasons why certain members were produced or not, and why they were mapped to optional members, pointers, sequences, etc. By looking at the comments that appear and iteratively modifying the model according to those comments, it is often possible to obtain the desired IDL.