Generics

Generics permit classes, structs, interfaces, delegates, and methods to be parameterized by the data types they store and manipulate. Generic class declaration should be mapped to the UML classifier (class or interface) with a Template parameter. Additionally, Generics still affect other parts of the program structures such as Attribute, Operation, Parameter, Parent Class, and Overloading Operators. In this chapter you will find how C# code structures are mapped to the UML model.

Generic Class

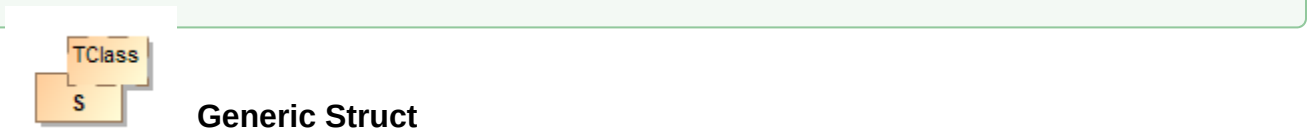
Class S has one template parameter named T. The default type of T is Class.

Example

Code:

```
public class S<T>
{
}
```

Reversed UML model:



Generic Struct

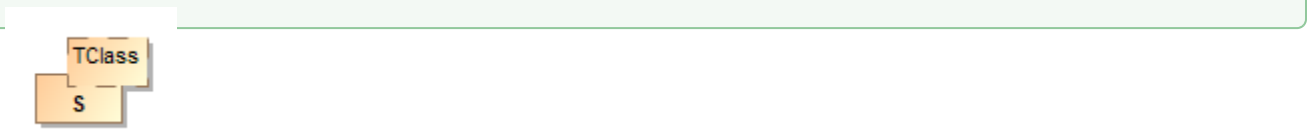
The type parameter of generic struct is created the same as generic class, but we apply «C#Struct» stereotype to the model.

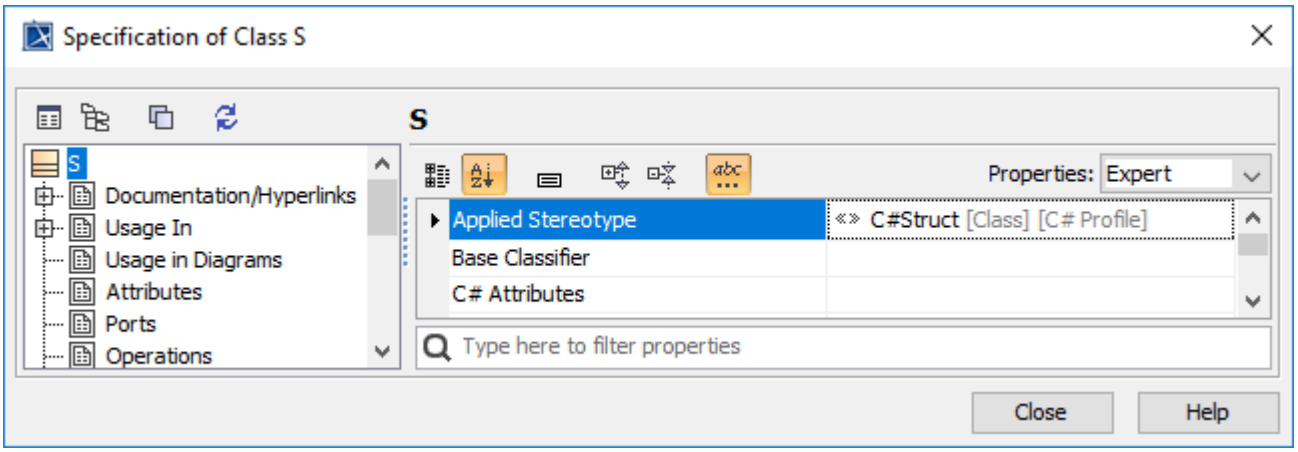
Example

Code:

```
public class S<T>
{
}
```

Reversed UML model:





Sample: Class S with applied stereotype C#Struct

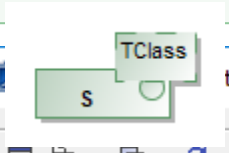
Generic Interface

Example

Code:

```
interface S<T>
{
}
```

Reversed UML model:



Sample: Interface S template parameters

Generic Delegate

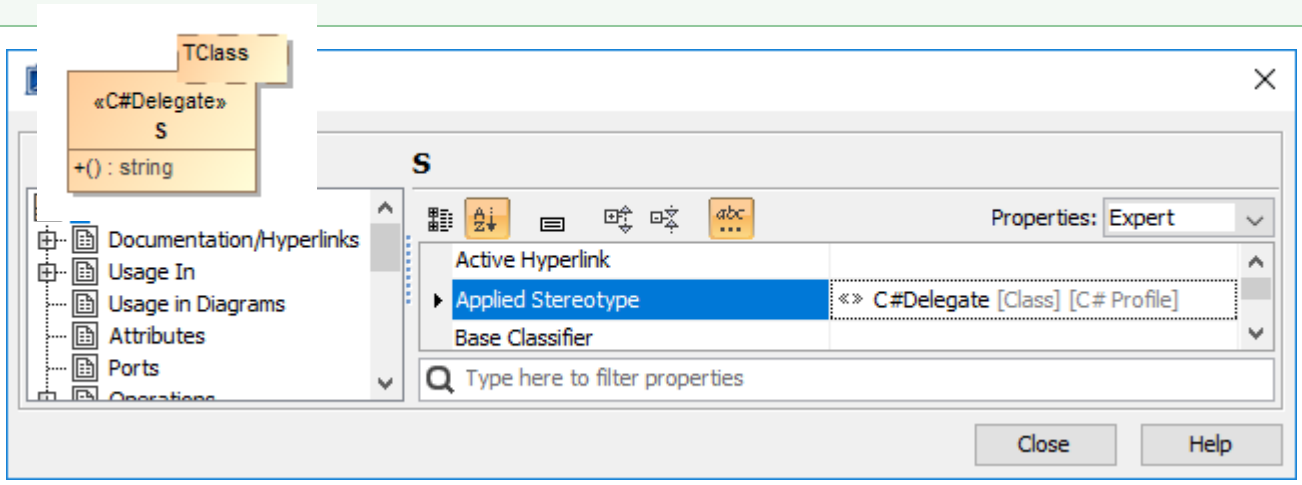
To create a generic delegate, we create a class model and apply the «C#Delegate» to the model. We, then, create an empty named method with delegate return type, and add template parameter like a normal generic class.

Example

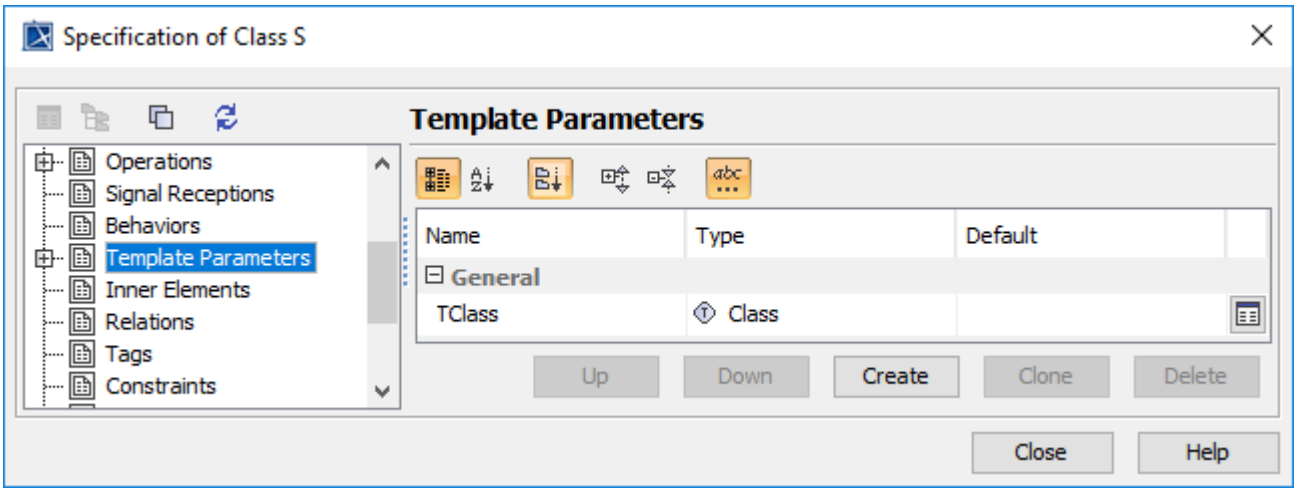
Code:

```
delegate string D<T>();
```

Reversed UML model:



Sample: Applied Stereotype is C#Delegate



Sample: Template parameters

Generic Attribute

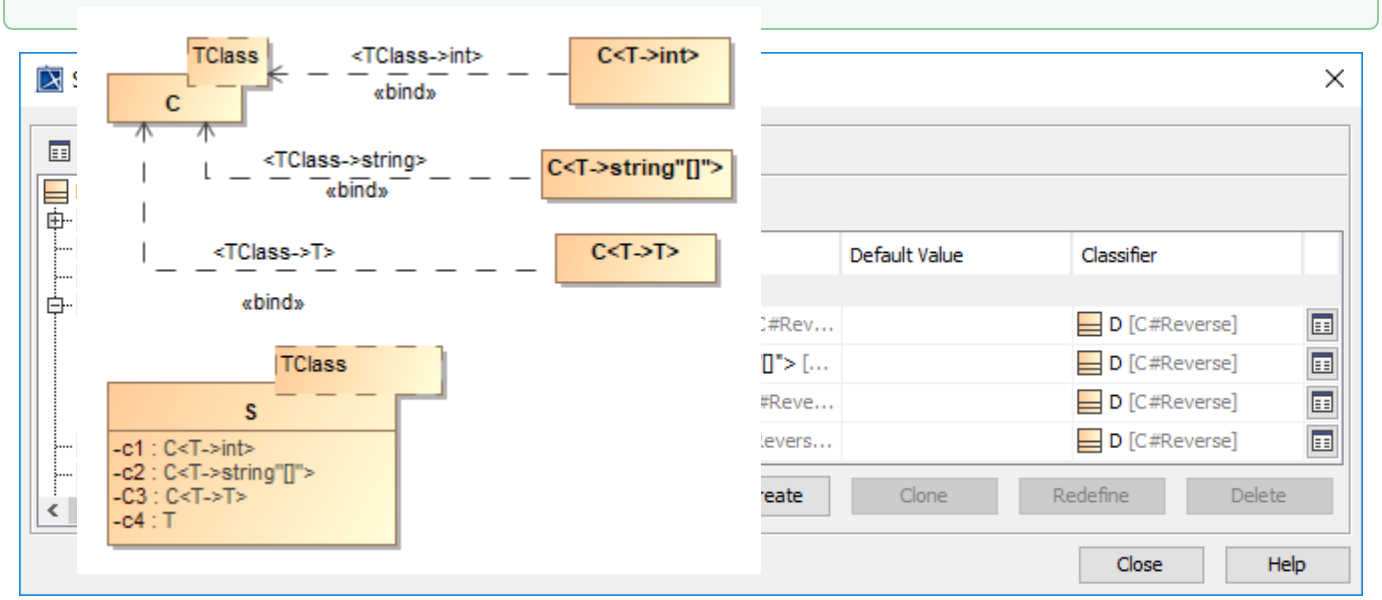
The type of attributes in the class can be generic binding (template binding) or template parameter of owner class. The example code shows attributes that use template binding as its type

Example

Code:

```
class C<T>
{
}
class D<T>
{
        private C<int> c1;
        private C<string[]> c2;
        private C<T> c3;
        private T c4;
}
```

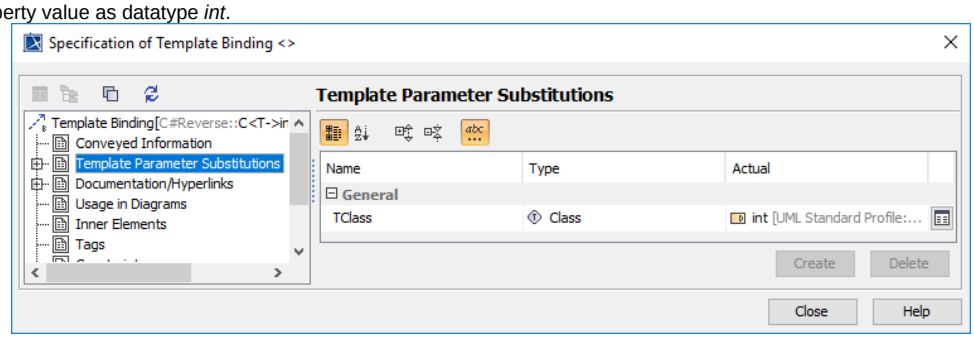
Reversed UML model:



Class *D* Attributes with generic type

The following shows how Template binding for *C<int>* is created:

1. From the Template binding link shortcut menu, select **Specification**.
2. From Specification window property group list, select **Template parameter Substitutions** property group.
3. Enter **Actual** property value as datatype *int*.

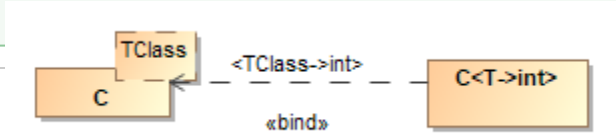


Template binding for C<int>

Code:

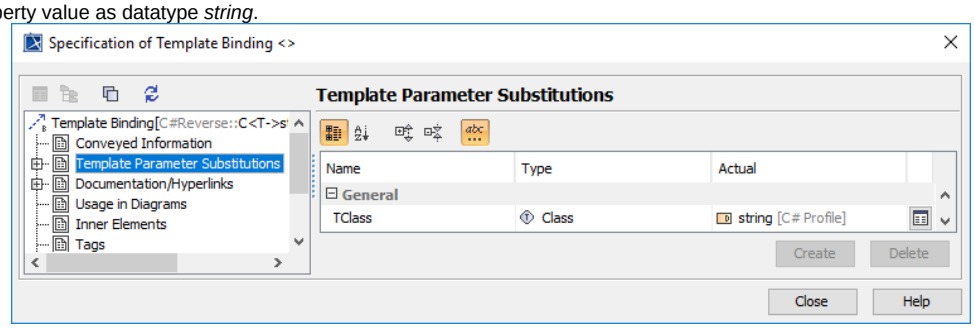
```
..
Private C<int> c1;
```

Reversed UML model:



Creating Template binding for *C<string[]>*:

.. From the Template binding link shortcut menu, select **Specification**.
2. From Specification window property group list, select **Template parameter Substitutions** property group.
3. Enter **Actual** property value as datatype *string*.

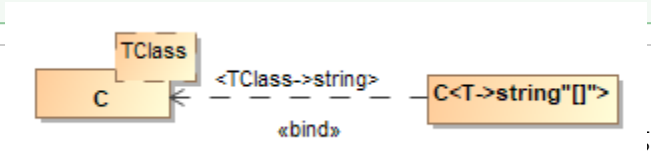


Template binding for C<string[]>

Code:

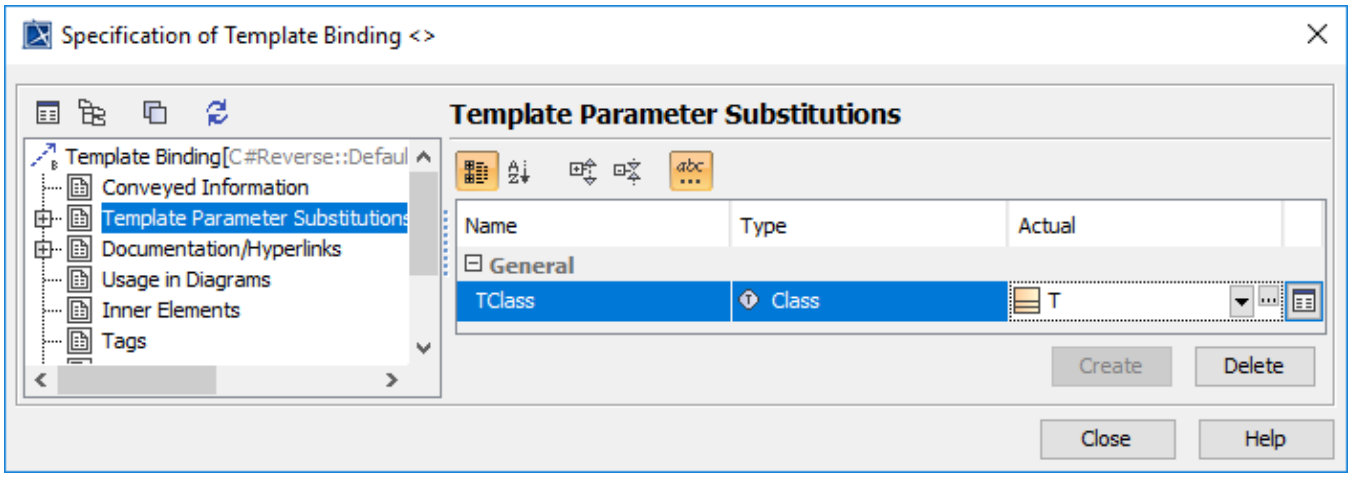
```
..
Private C<string[]> c2;
```

Reversed UML model:



Creating Template binding for *C<T>*:

.. From the Template binding link shortcut menu, select **Specification**.
2. From Specification window property group list, select **Template parameter Substitutions** property group.
3. Enter **Actual** property value as Binding type *T*. *T* is the template parameter of its Owner Class *S<T>*.

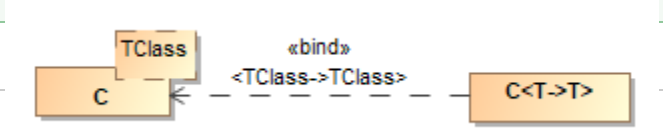


Template binding for C<T>

Code:

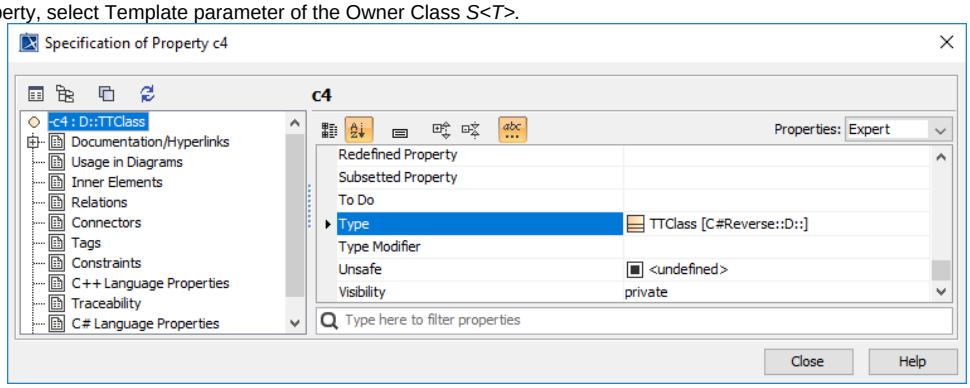
```
..
Private C<T> c3;
```

Reversed UML model:

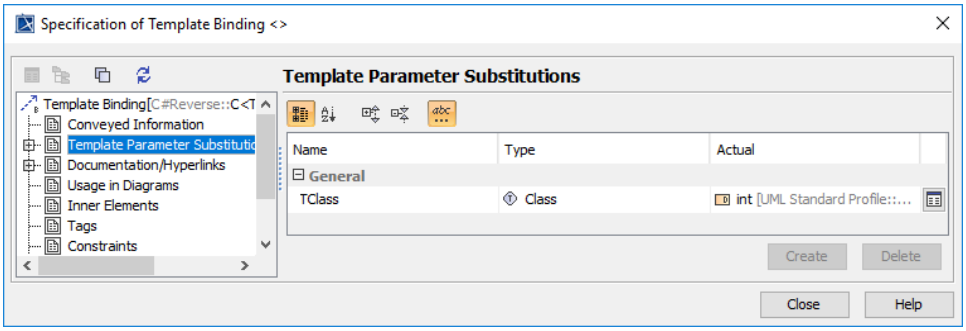


Creating attribute with the type template parameter of the owner class S<T>:

1. Create Attribute *c4* (c4 is not a template binding class).
2. From the Attribute link shortcut menu, select **Specification**.
3. Specify **Type** property, select Template parameter of the Owner Class *S<T>*.



4. From Specification window property group list, select **Template parameter Substitutions** property group.
5. Enter **Actual** property value as datatype *int.*



Template binding for Attribute c4

Code:

```
..
Private T c4;
```

Generic Operation

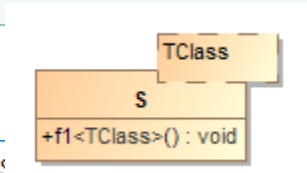
Language extention Generic can be applied to **Operation**.

Example

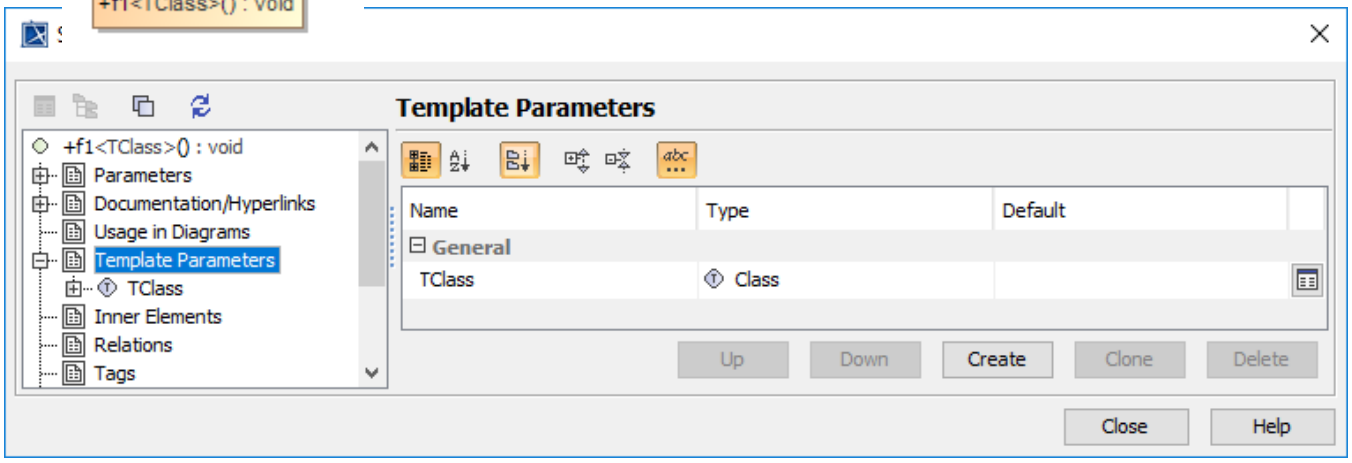
Code:

```
public class S<T>
{
	public void f1<T>() {}
}
```

Reversed UML model:



To create **Generic Operation**, open the operation Specification window, and from **Property group** createTemplate Parameters.



To create Operations with parameters and return type, create them like a **Generic Attribute**.

Binding Types

We have to create or select the correct binding types.

In the example code, the type of parameter *t* is public void *f1<T> (T t)*, it must be the **Template Parameter** of the owner method, but type *f1<T>* is not the template parameter of the owner Class *S<T>*.

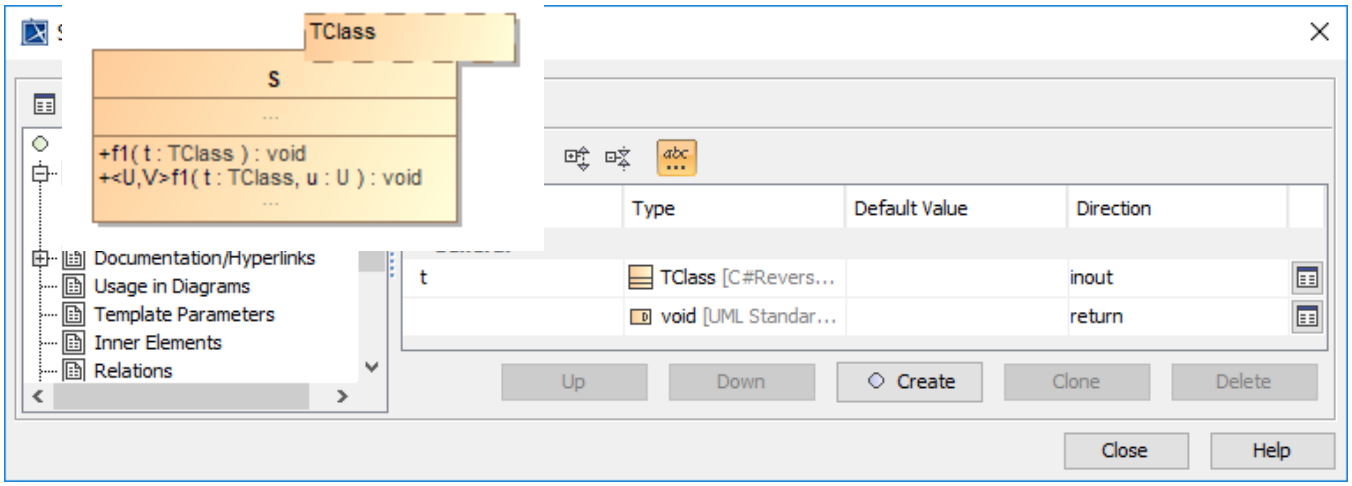
In the second method, public void *f1<U, V> (T t, U u)*, parameter *t* type must be the **Template Parameter** of the owner Class, *S<T>*.

Example

Code:

```
public class S<T>
{
public void f1<T> (T t)
{
}
public void f1<U, V> (T t, U u)
{
}
}
```

Reversed UML model:

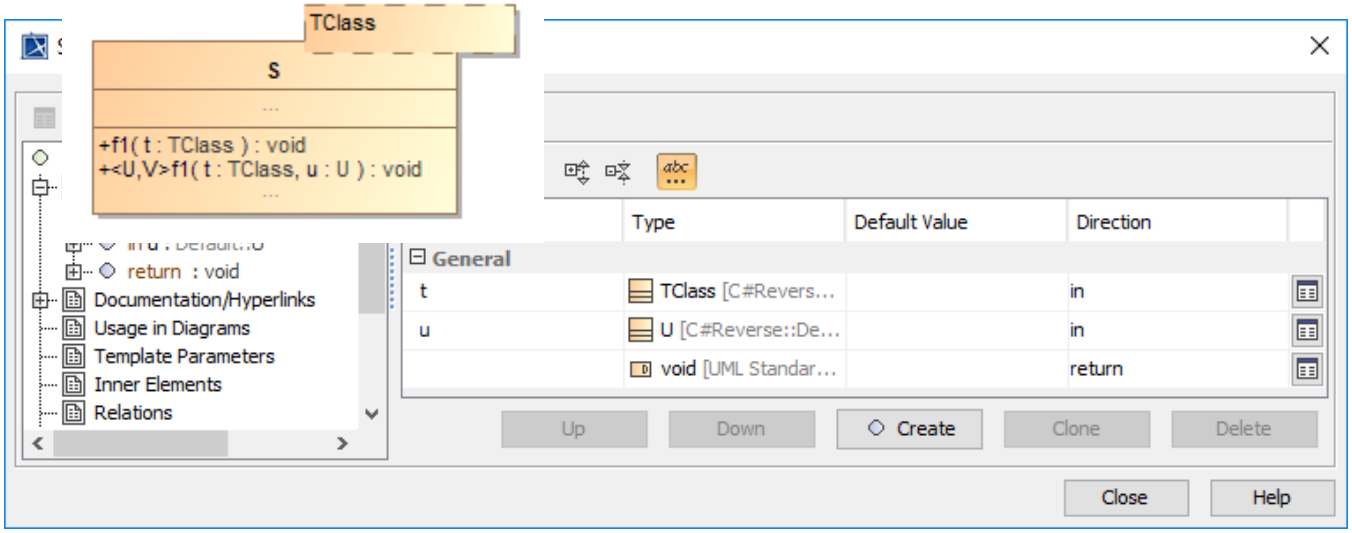


Example

Code:

```
class b
{
 public T f1<T, U>(T t, U u)
       whereU:T{returnt; }
}
```

Reversed UML model:



Generic overloading

Methods, constructors, indexers, and operators within a generic class declaration can be overloaded. While signatures as declared must be unique, it is possible that substitution of type arguments results in identical signatures.

Example

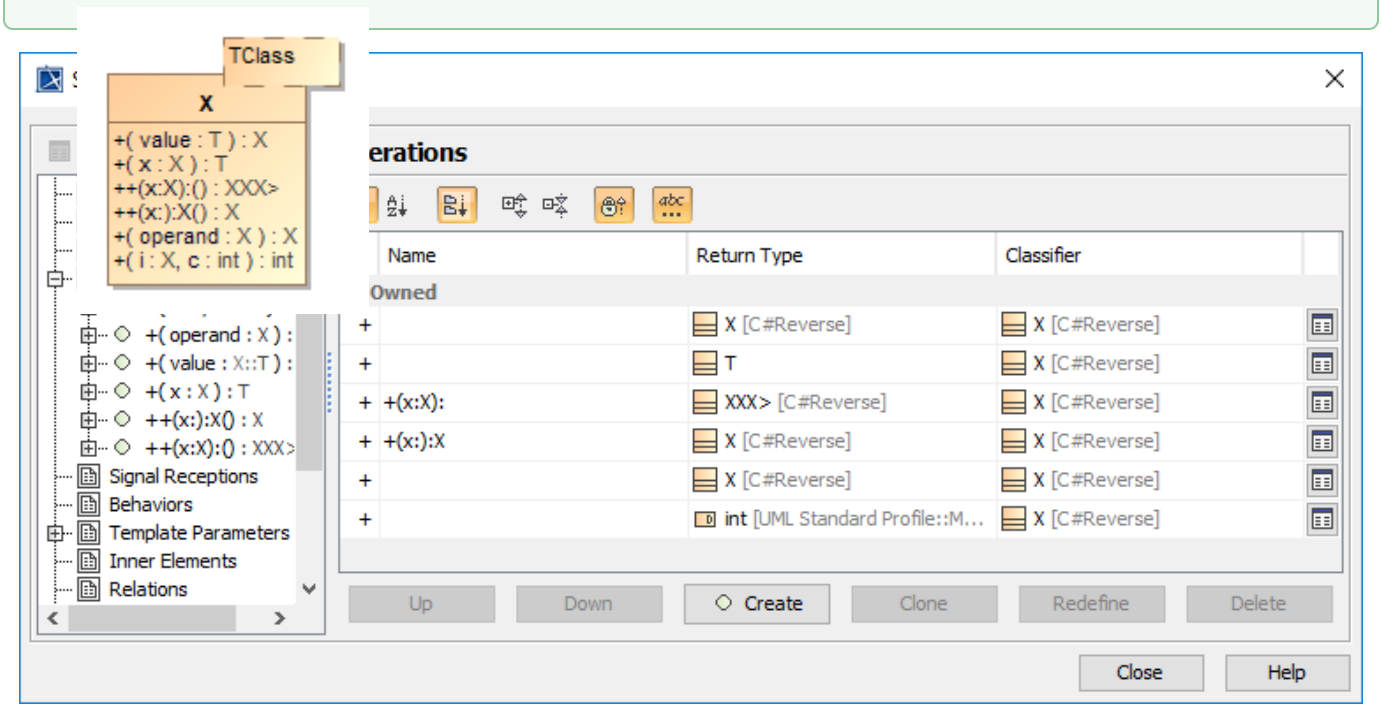
Code:

```
class X<T>
{
                public static explicit operator X<T>(T value)
                { return null; }
                public static implicit operator T(X<T> x)
                { return x.item; }
                public static explicit operator
XXX<int>(X<T> x)
                { return null; }
                public static explicit operator
X<T>(XXX<int> x) }
                { return null; }
                public static X<T> operator ++(X<T> operand)

                { return null; }
                public static int operator >>(X<T> i, int c)

                { return c; }
}
```

Reversed UML model:



Generic parent class

Example

Code:

```
class b<T, U>
{}
class b1<X> : b<X[], X[,]>
{}
```

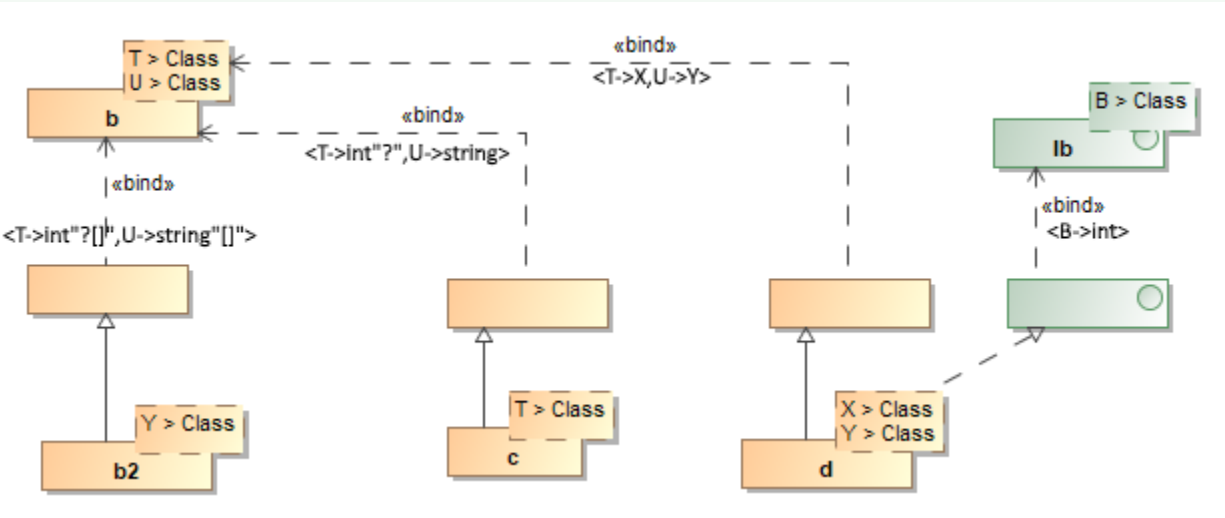
Reversed UML model:

Example

Code:

```
class b<T, U>
{}
interface Ib<B>
{}
class b2<Y> : b<int?[], string[]>
{}
class c<T> : b<int?, string>
{}
class d<X, Y> : b<X, Y>, Ib<int>
{}
```

Reversed UML model:



Generic using alias

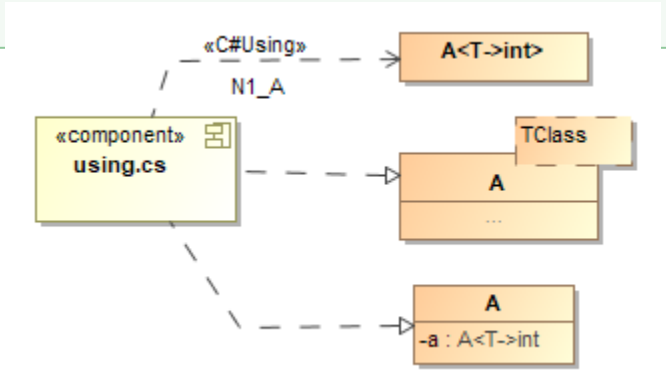
For example, using N1_A = N1.A<int>, we create Template binding for *A<int>* in Namespace *N1*, and then we create the dependency *Usage* from the parent component (in this case it is file component) to the class with template binding.

Example

Code:

```
using N1_A = N1.A<int>; namespace N1
{
public class A<T> {}
}
class A
{
N1_A a;
}
```

Reversed UML model:



Generic constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type-parameter-constraints-clauses:*

type-parameter-constraints-clause
type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause:

where *type-parameter : type-parameter-constraints*

The list of constraints given in a where clause can include any of the following components, in this order: a single primary constraint, one or more secondary constraints, and the constructor constraint, new().

type-parameter-constraints:

primary-constraint
secondary-constraints
constructor-constraint
primary-constraint , secondary-constraints
primary-constraint , constructor-constraint
secondary-constraints , constructor-constraint
primary-constraint , secondary-constraints , constructor-constraint

A *primary constraint* can be a class type or the reference type constraint class or the value type constraint struct.

class-type

`class`

`struct`

A *secondary constraint* can be a type-parameter or interface-type:

interface-type

type-parameter

secondary-constraints , interface-type

secondary-constraints , type-parameter

constructor-constraint:

`new ( )`

Each *type-parameter-constraints-clause* consists of the token *where*, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be at most one where clause for each type parameter, and the where clauses can be listed in any order.

The given list of constraints in a *where* clause can include any of the following components, in this order: a single primary constraint, one or more secondary constraints, and the constructor constraint, new().