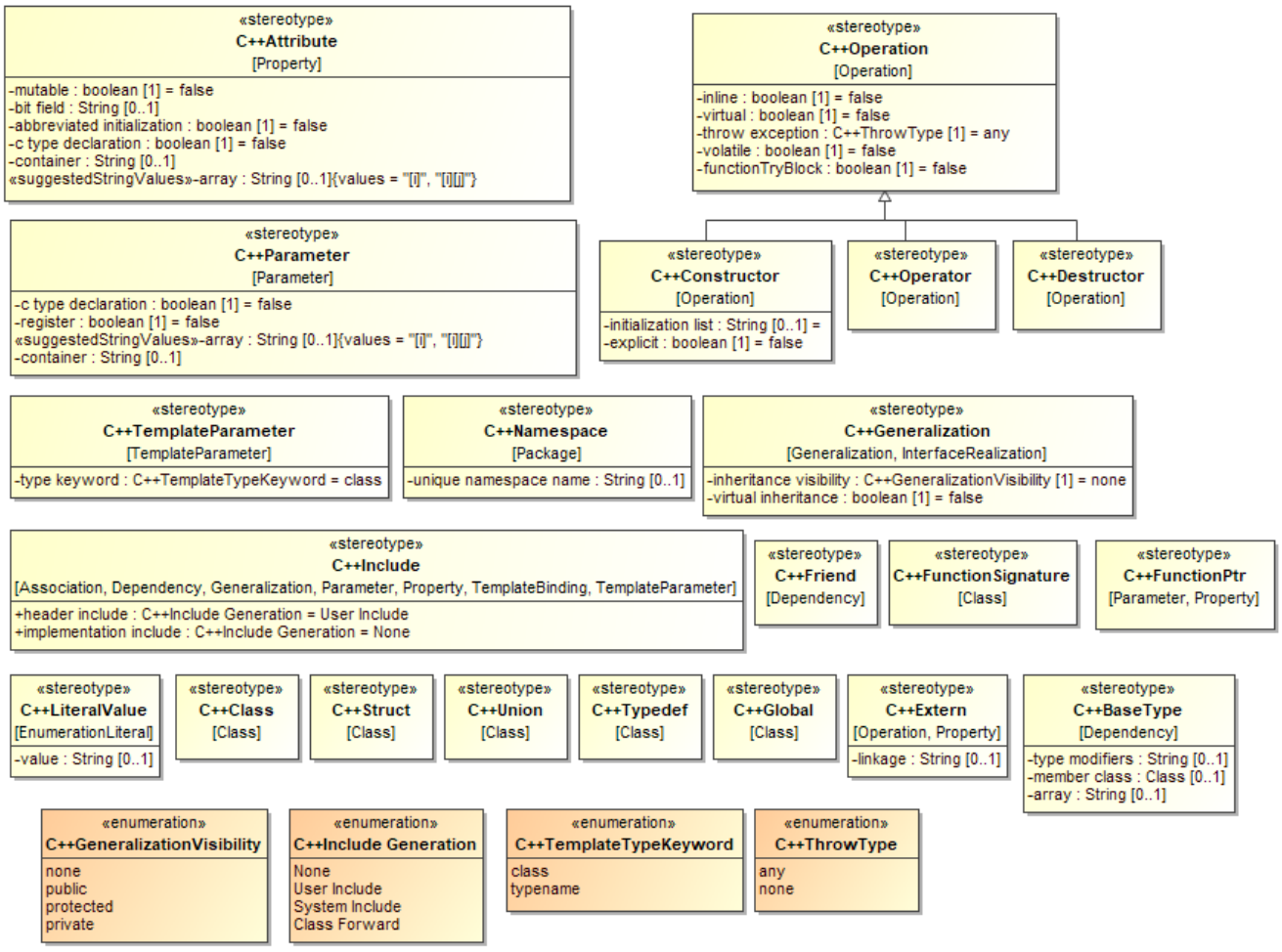
C++ Mapping to UML

This chapter describes the mapping between C++ and UML.



Stereotypes with Tag values

- Tag Abbreviated Initialization set to true.

Const Volatile Qualified Type
- Tag bit field with value 2

Function
- Return type of function

Function Variable-Length Parameter List
Void Parameter
- Operation with type void

Register Parameter
- Tag register

Function Modifiers
- Tag explicit

Function Pointer
- Tag explicit

Function Operator
Exception
- Property throw exception

Visibility
Static members
- Property Is Static

Pure virtual function and abstract class
- Operation property Is Abstract

Friend Declaration
Struct
Union
Enumeration
Typedef
Namespace
Global Functions And Variables
Class Definition
Class Template Definition
- Template Parameters
- Template Parameter Tag type keyword.

Function Template Definition
- Template Parameters

Default Template Parameter
Template Instantiation
Partial Template Instantiation
Template Specialization
Forward Class Declaration
- Template Parameters
- header include value Class forward

Include Declaration
- User Include
- System include
- User Include
- System include

C++Class

C++ class map to a UML class

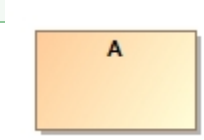
Stereotype «*C++Class*» is an invisible stereotype used to include language properties for any C++ variable and belongs to Meta class *Class*.

Example

C++ source code

```
class A {
};
```

UML model:



Base Class Definition

A *base class* is a Class from which other Classes are derived. It facilitates the creation of other Classes that can reuse the code implicitly inherited from the *base class* (except constructors and destructors).

Base class definition is mapped to UML generalization, a generalization is created between the *base class* and the *supe r class*.

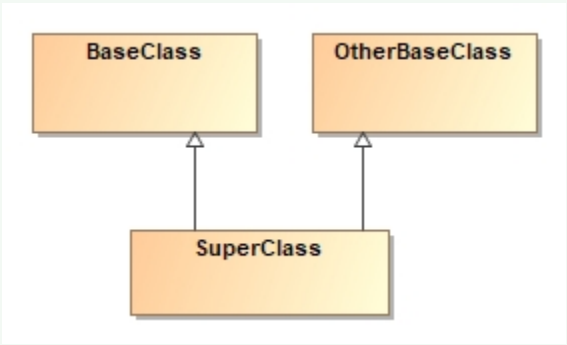
Access visibility (public, protected and private) and virtual properties of the *base class* are mapped to C++ language properties of the UML generalization.

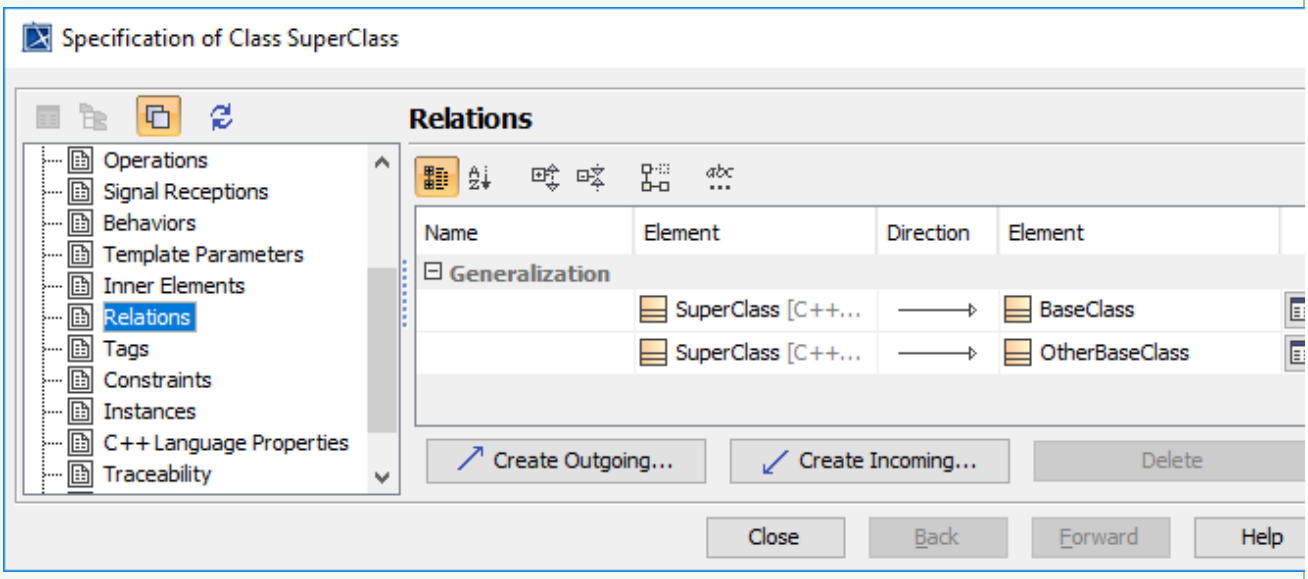
Example

C++ source code

```
class BaseClass {};
class OtherBaseClass {};
class SuperClass :
 public BaseClass,
 protected virtual OtherBaseClass {
};
```

UML model:





Class relations

Class Member Variable

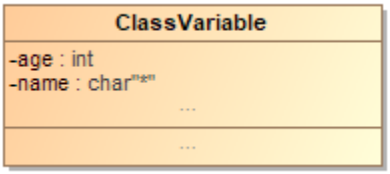
Class member variables are mapped to UML attributes. See **Variable** for more info.

Example

C++ source code

```
class ClassVariable {
 int age;
 char* name;
};
```

UML model:



Class Member Function

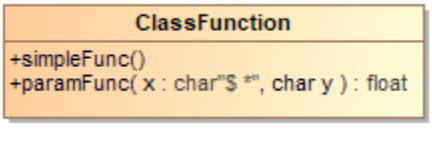
Class member functions are mapped to UML operations.

Example

C++ source code

```
class ClassFunction {
public:
 void simpleFunc();
 float paramFunc(int x,char y);
};
```

UML model:



Class Constructor and Destructor

C++Constructor

Stereotype *«C++Constructor»* is used to define C++ Constructor. This stereotype extends stereotype *«C++Operation»*.

Name	Meta class	Constraints
C++Constructor	Operation	name = owner.name
Tag definition	**Type and default value**	**Description**
explicit	boolean[1]=false	`Constructor` *`Explicit`* `explicit a();`
initialization list	String[0..1]	Constructor initialization: `a() : x(1) {}`

C++Destructor

Stereotype *«C++Destructor»* is used to define C++ destructor. This stereotype extends stereotype *«C++Operation»*.

Name	Meta Class	Constraints
C++Destructor	Operation	`name = "~"+owner.name`

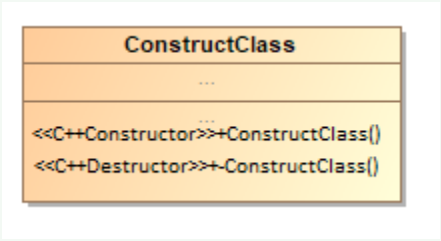
C++ class constructor and destructor are mapped to UML operation with stereotypes *«C++Constructor»* and «C++Destructor»:

Example

C++ source code

```
class ConstructClass {
public:
 ConstructClass();
 ~ConstructClass();
}
```

UML model:



Variable

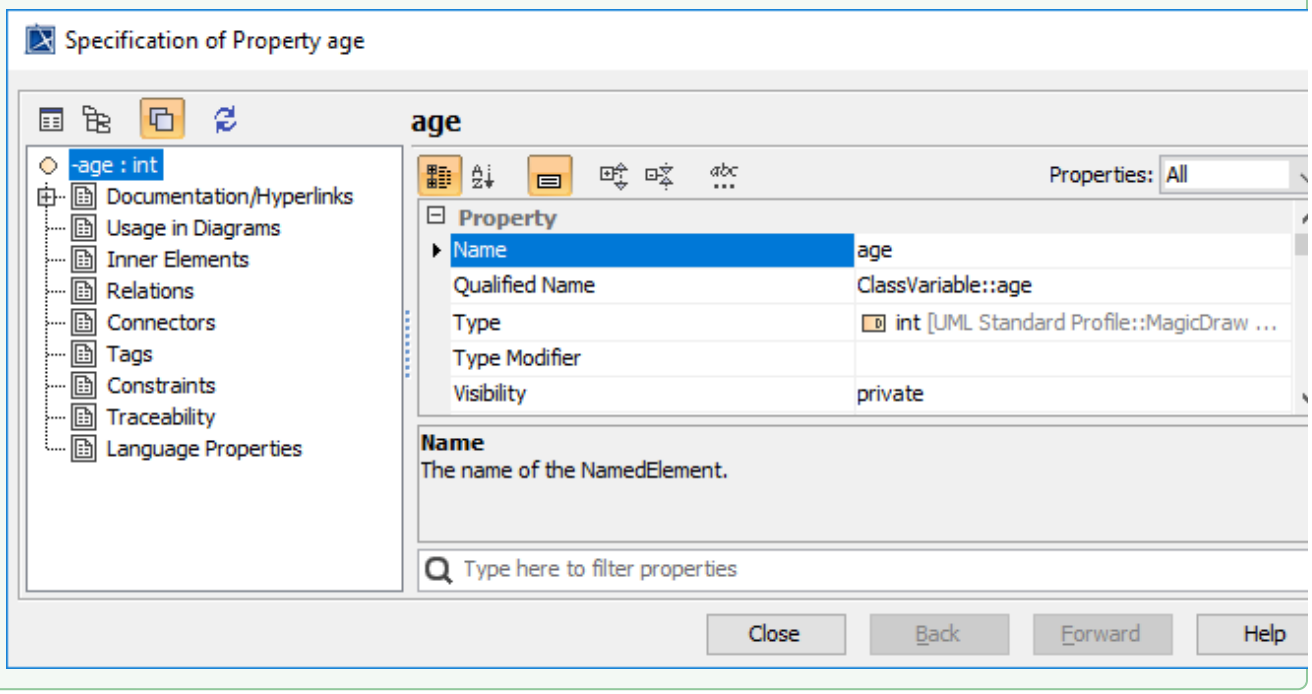
C++ variable is mapped to UML *attribute*, the variable type is mapped to the attribute type.

Example

C++ source code

```
int age;
```

UML model specification:



Pointer and reference

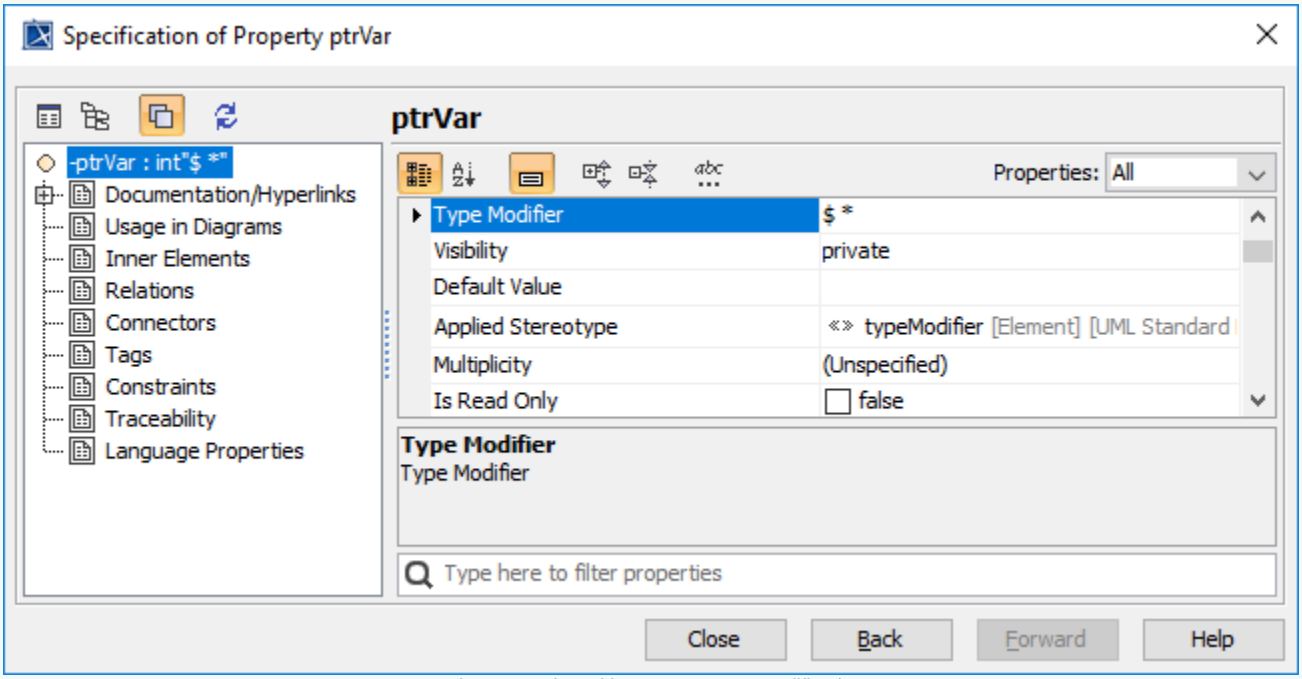
C++ type *pointer* and *reference* is mapped to property *Type Modifier* of the attribute. Character *$* is replaced by the type name.

Example

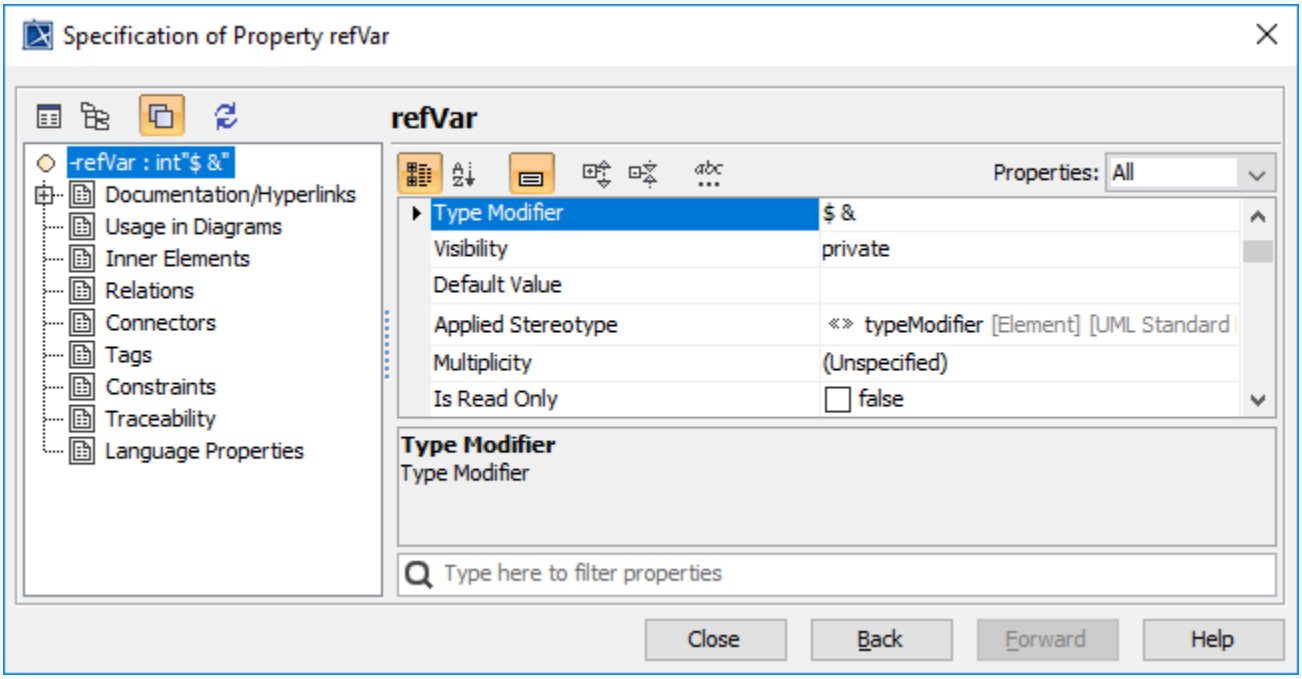
C++ source code

```
int* ptrVar;
int& refVar
```

UML model specification:



Pointer mapping with property *Type Modifier $**



Reference mapping with property *Type Modifier $&*

Array

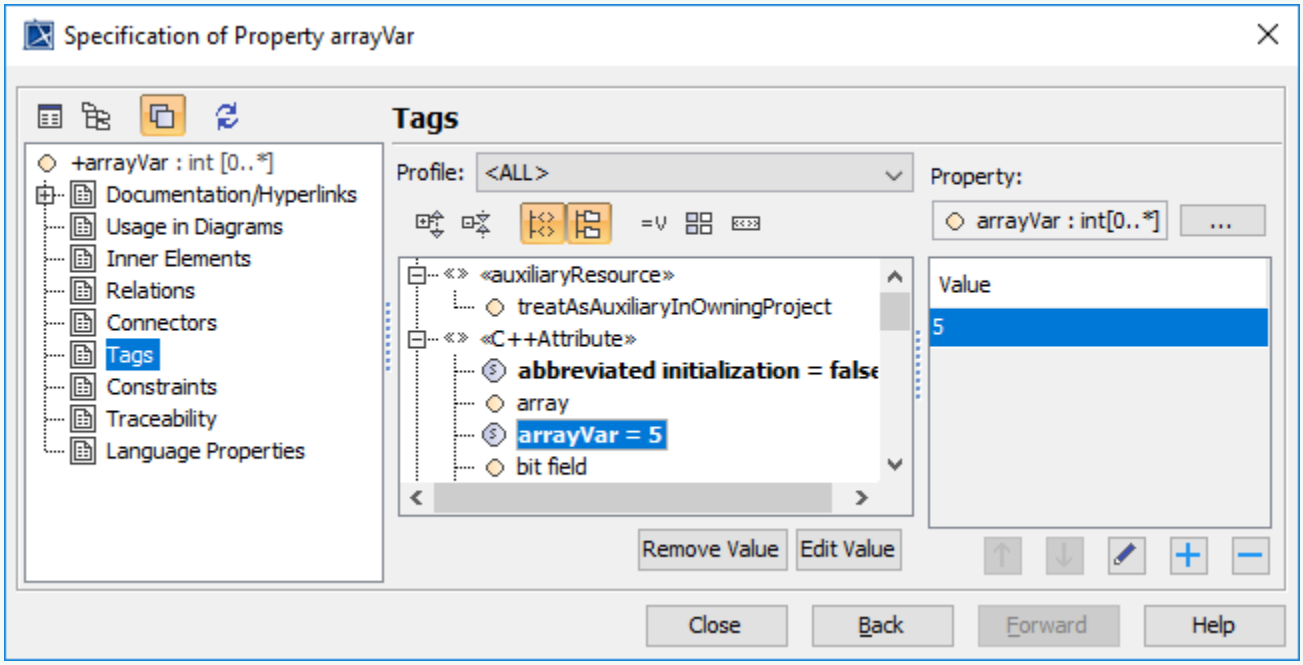
C++ array type is mapped to array tag value of the attribute. If array is set, then multiplicity property of UML attribute is set to “[0..*]”

Example

C++ source code

```
int arrayVar[5];
```

UML model Tag specification:



<<C++Attribute>> Tag *arrayVar* value is 5

Mutable variable modifiers

Mutable variable modifiers are mapped to UML attribute's language property *Mutable*.

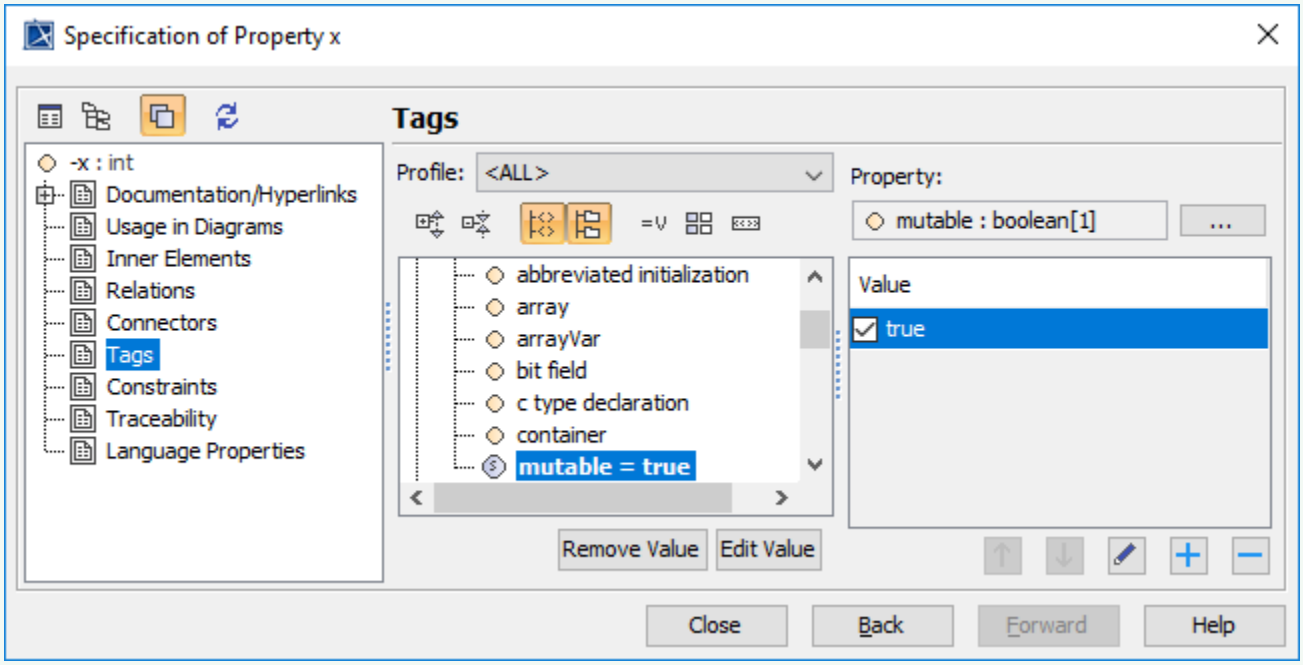
Constraint: only member variable can be *Mutable* (*Global variable* cannot be *Mutable).*

Example

C++ source code

```
mutable int x;
```

UML model Tag specification:



When variable is mutable, its Tag *mutable* value is true

Bit field

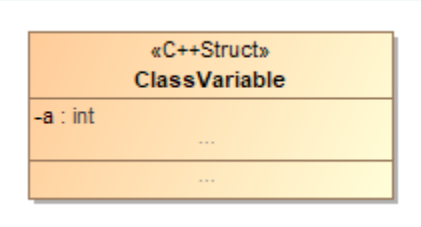
Bit field is mapped to tag value *Bit field*.

Example

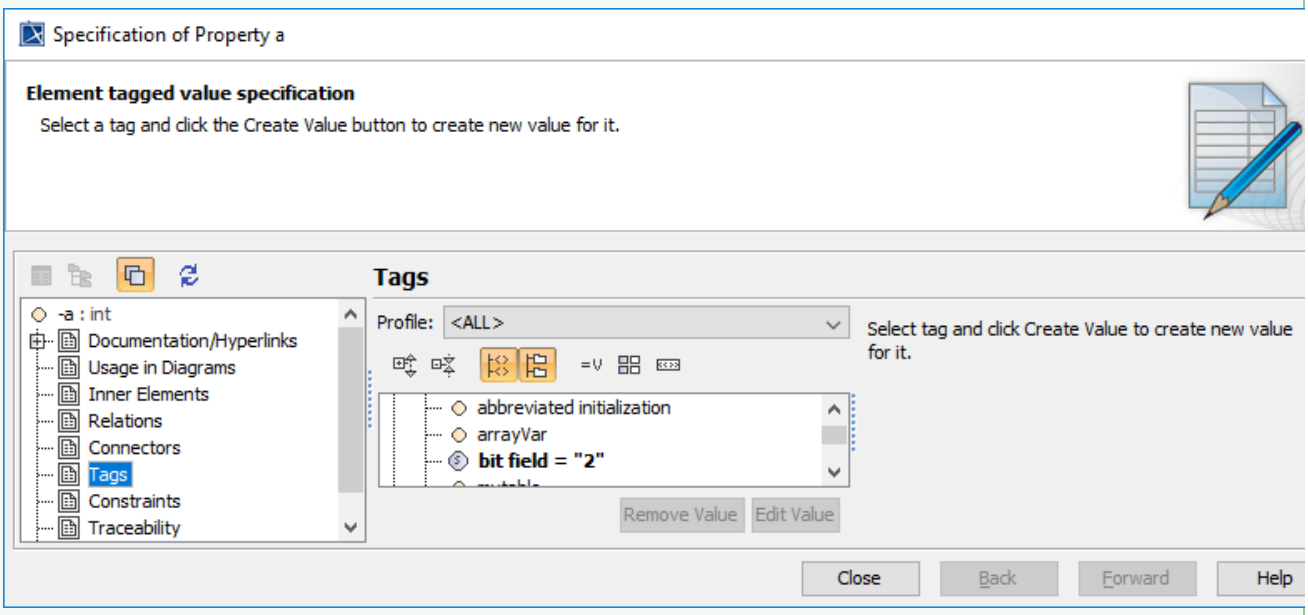
C++ source code

```
struct BitStruct {
 int a:2;
};
```

UML model:



UML model Tag specification:



Tag *bit field* with value 2

Variable Extern

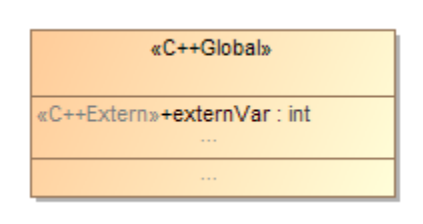
C++ extern variable is mapped to stereotype *«C++Extern»*. Tag *Linkage* value is used to specify the kind of linkage *C* or *C++*, if *linkage* is not specified (or without value), extern variable without *linkage* is generated.

Example

C++ source code

```
extern int externVar;
```

UML model:



Variable Default Value

Variable initial value is mapped to UML attribute's default value. Variable initial value set using function style method is mapped to UML attribute's default value and attribute's language property Tag *Abbreviated Initialization* set to true.

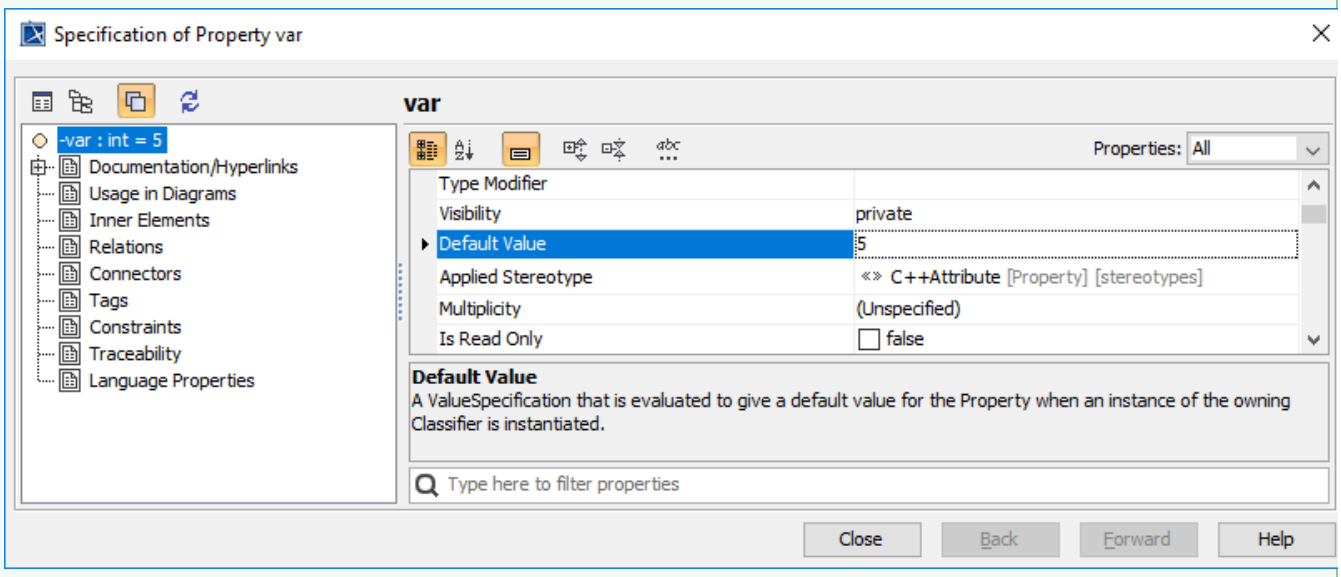
Constraint: only *static const* member variables can be initialized, and they cannot be initialized using function style method.

Example

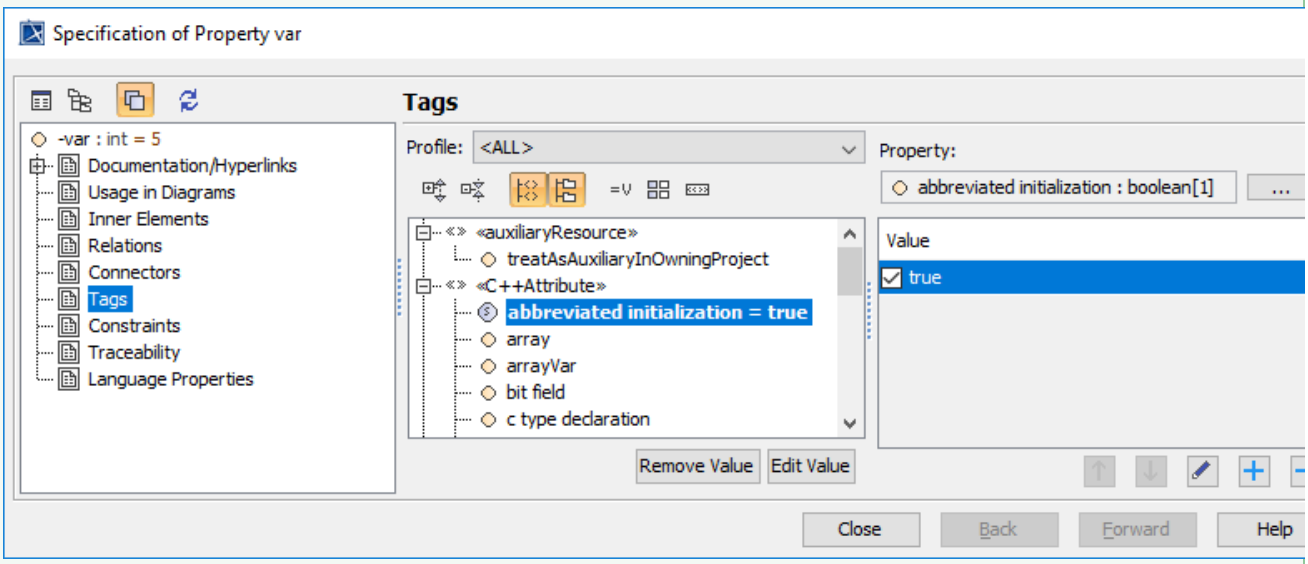
C++ source code

```
int var = 5;
int var2(10);
```

UML model Tag specification:



Variable initial value is mapped to UML attribute's default value



Tag *Abbreviated Initialization* set to true.

Const Volatile Qualified Type

C++ const and volatile modifiers for attribute/function parameter are mapped to Type Modifiers properties.

For attribute *const*, the property **Is Read Only** is set to *true* during reverse.

The character *$* in **Type Modifier** value is replaced by the type name.

Constraint : If the property **Is Read Only** is set and **Type Modifiers** is not set to *const* or *const volatile* - set to const, or an error message will display during syntax check.

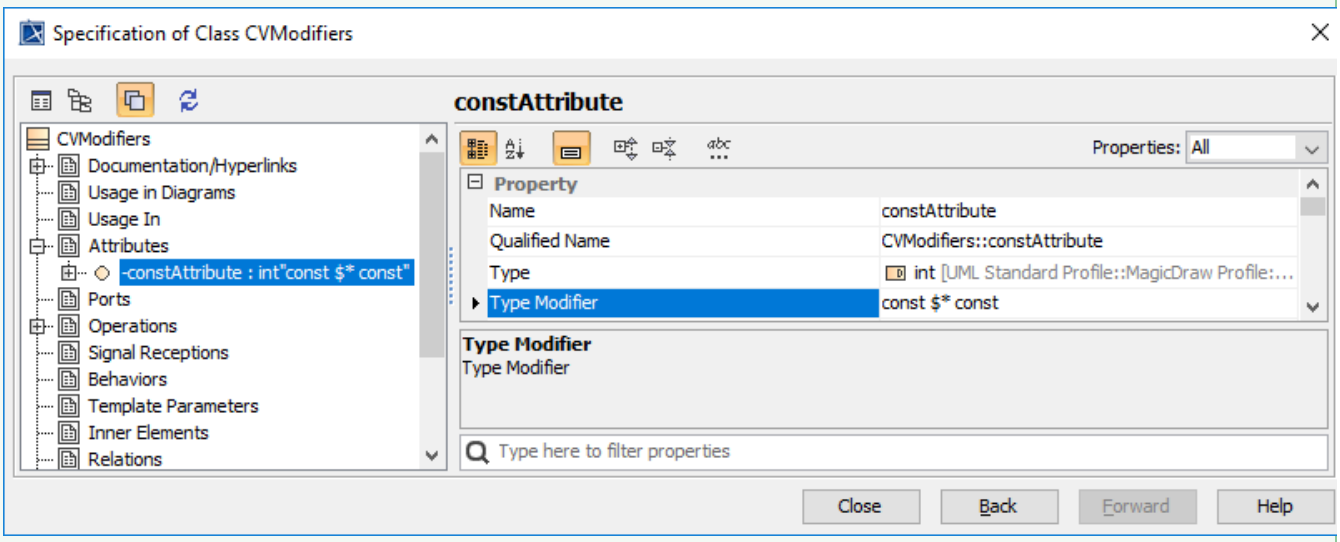
Example

C++ source code

```
class CVModifiers {
const int* const constAttribute;
}
```

UML model:

UML model Tag specification:



Tag *bit field* with value 2

Function

C++ function is mapped to UML operation, parameter of function is mapped to UML parameter with property direction set to *inout*, return type of function is mapped to UML parameter with property direction set to *return*. Type of parameter is mapped to type of UML parameter.

C++ default parameter value is mapped to *defaultValue* property of UML parameter.

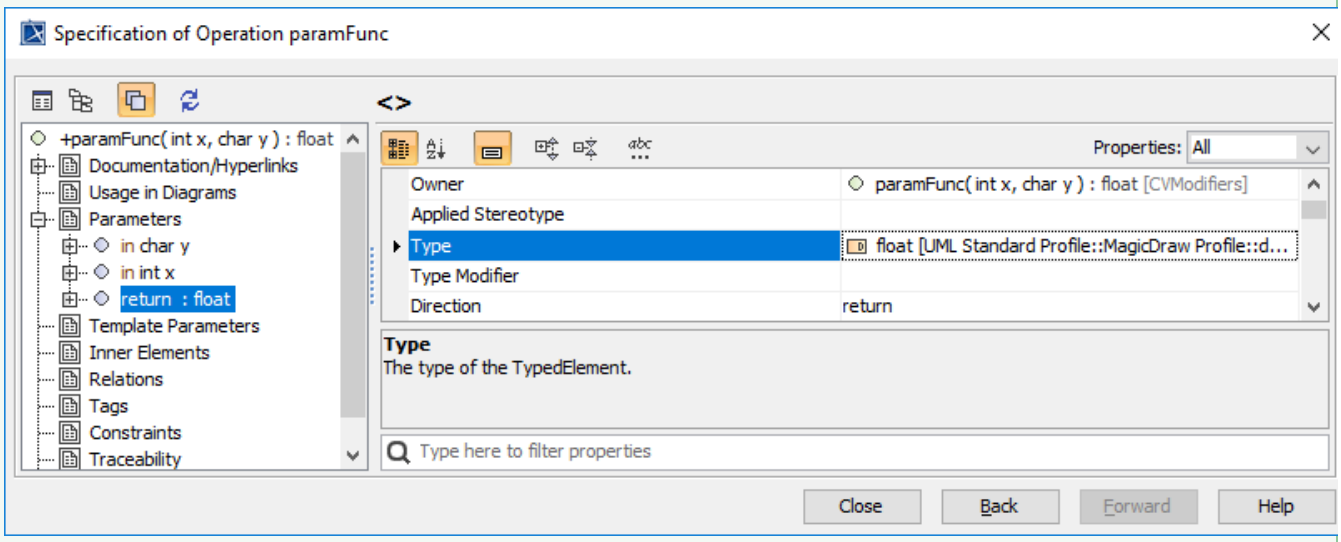
Pointer, reference and array type of parameter are mapped to property **Type Modifier** of parameter.

Example

C++ source code

```
float paramFunc(int x, char x);
```

UML model Tag specification:



Return type of function

Function Variable-Length Parameter List

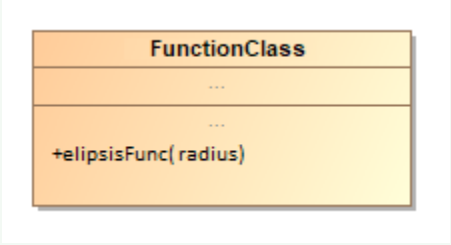
C++ function variable-length parameter list is mapped to a UML parameter with name "..." and without type.

Example

C++ source code

```
Class FunctionClass {
public:
 EllipsisFunc(radius);
};
```

UML model:



Void Parameter

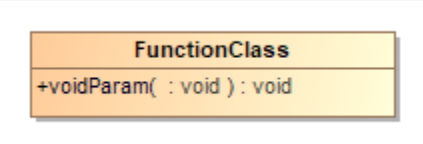
C++ void function parameter is mapped to a UML parameter without name and with type *void*.

Example

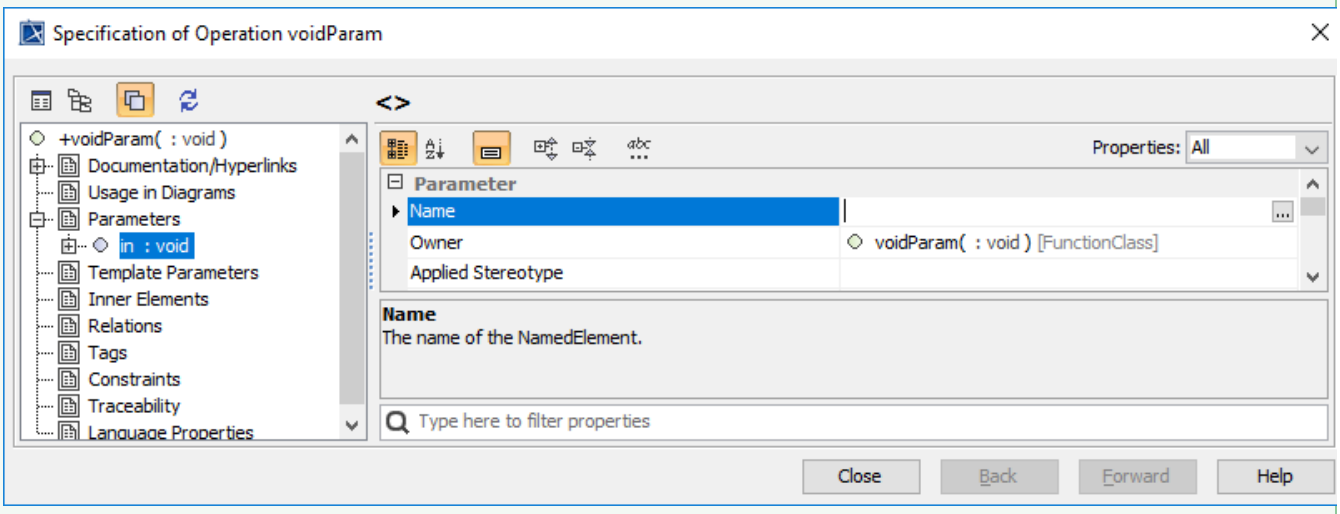
C++ source code

```
Class FunctionClass {
public:
 void voidParam(void);
};
```

UML model:



UML model Tag specification:



Operation with type void

Register Parameter

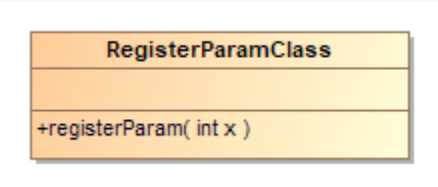
C++ register parameter is mapped to UML parameter language property **Register Depending** on the compiler, register can be limited on some types (int, char).

Example

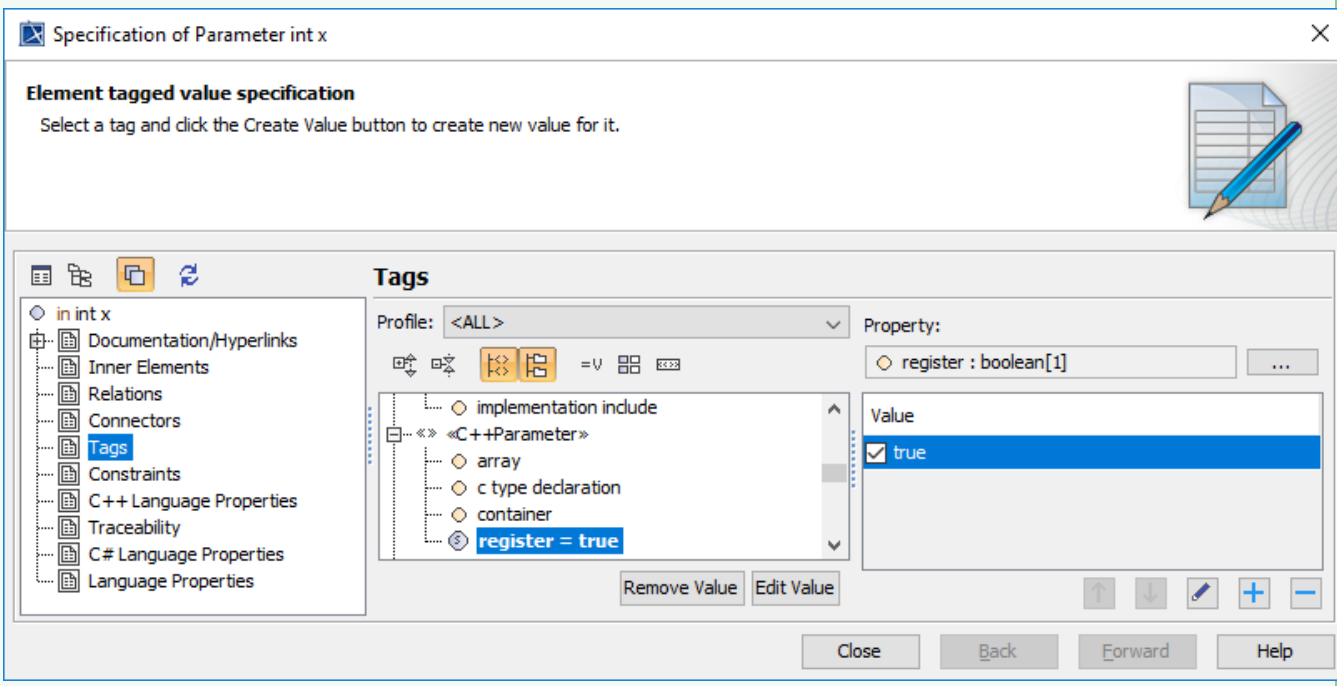
C++ source code

```
class RegisterParamClass {
void registerParam(register int x);
};
```

UML model:



UML model Tag specification:



Tag *register*

Function Modifiers

C++ function modifiers are mapped to Language properties of Operation.

Virtual function is mapped to *Virtual modifier* property.

Inline function is mapped to *Inline modifier* property.

Explicit function is mapped to *Explicit modifier* property. Constraint: explicit is only valid for constructor.

Const function is mapped to UML operation *Is Query* property.

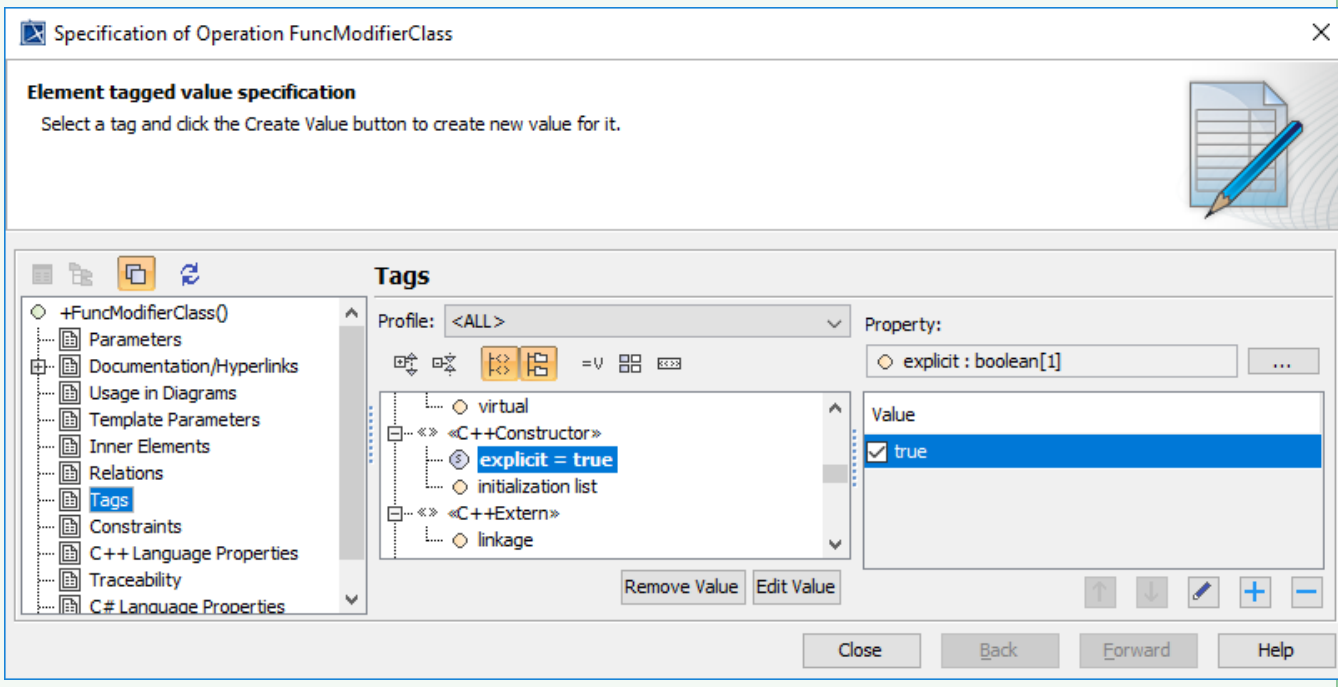
Volatile function is mapped to Tag value *volatile*.

Example

C++ source code

```
class FuncModifierClass {
      explicit FuncModifierClass();
};
```

UML model Tag specification:



Tag *explicit*

Function Pointer

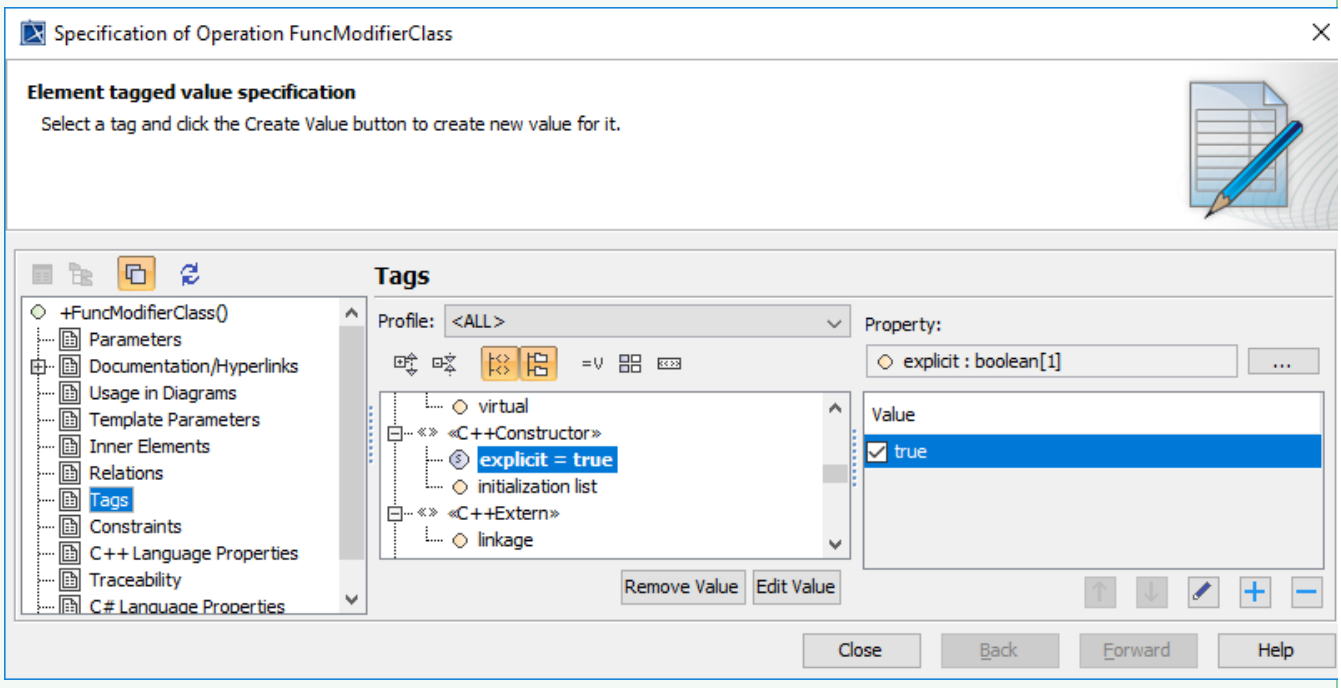
C++ function pointer type is mapped to attribute/parameter with «C++FunctionPtr» stereotype, a dependency with «C++BaseType» stereotype link from the attribute/parameter to the operation in a «C++FunctionSignature» class, and type modifiers of the dependency is set to *$. Member function pointer use the same mapping, and member class tag of «C++BaseType» stereotype point to a class.

Example

C++ source code

```
float (A*funcPtr)(int);
```

UML model Tag specification:



Tag *explicit*

Function Operator

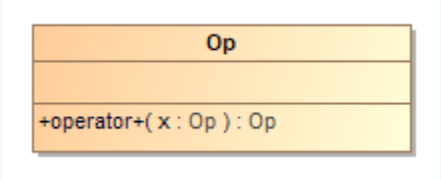
C++ function operator is mapped to normal function with the C++ operator name mapped to UML operation name. See **C++Operator** for more info.

Example

C++ source code

```
Class Op {
Public:
 Op operator+(Op x);
};
```

UML model:



Exception

C++ exception is mapped to UML operation's raised *exception* properties.

If *raisedExpression* is empty, and *throw exception* tag is set to *none* a throw without parameter is generated.

If *raisedExpression* is empty, and throw exception tag is set to *any* throw keyword is not generated.

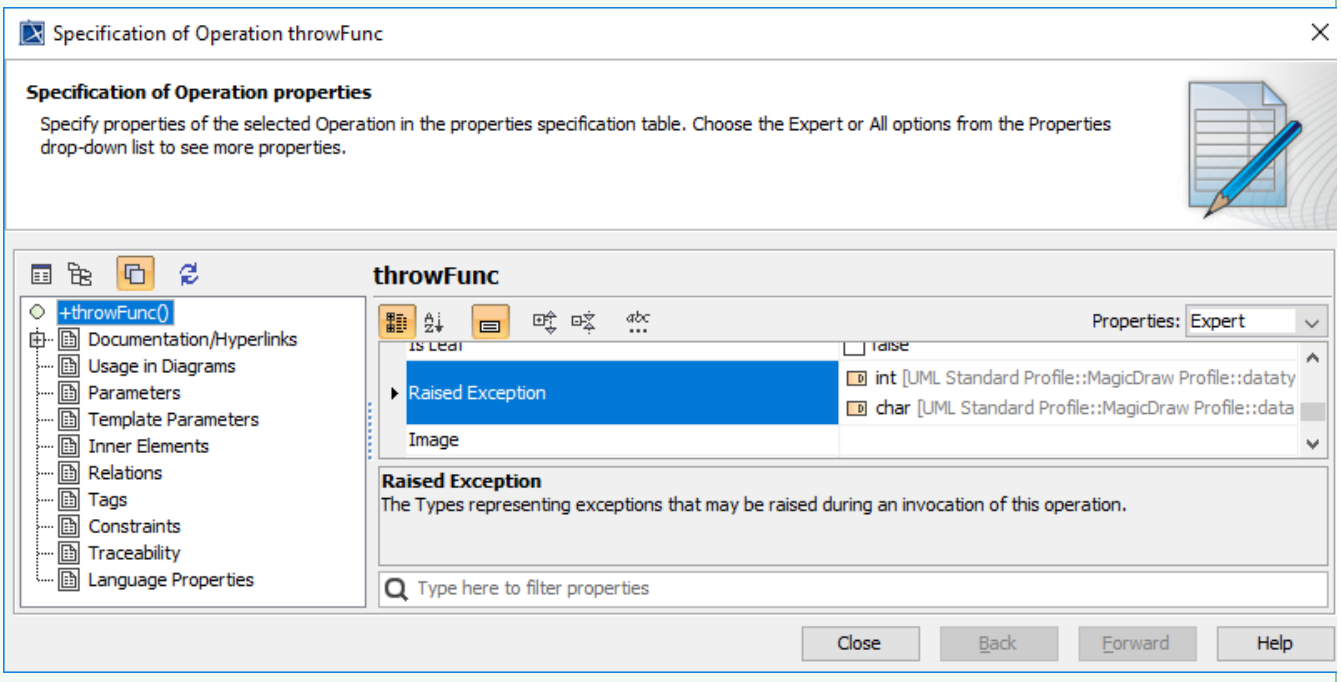
If the tag *throw exception* is not set, then generate specific *raisedExpression*, or do not generate throw if *raisedExpression* is empty.

Example

C++ source code

```
void throwFunc() throw (int,char);
```

UML model Tag specification:



Property *throw exception*

Visibility

Variables and function visibility are mapped using the UML visibility property.

Members of C++ class without access visibility specified are private.

Members of C++ struct or union without access visibility specified are public.

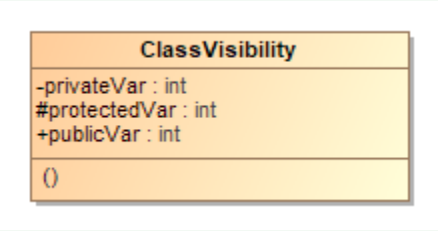
Variables and functions outside a class/struct/union are public.

Example

C++ source code

```
class ClassVisibility {
 int privateVar;
protected:
 int protectedVar;
public:
 int publicVar;
};
```

UML model:



Static members

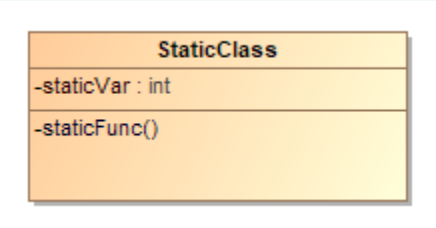
Static variables and functions are mapped to UML *Is Static* property.

Example

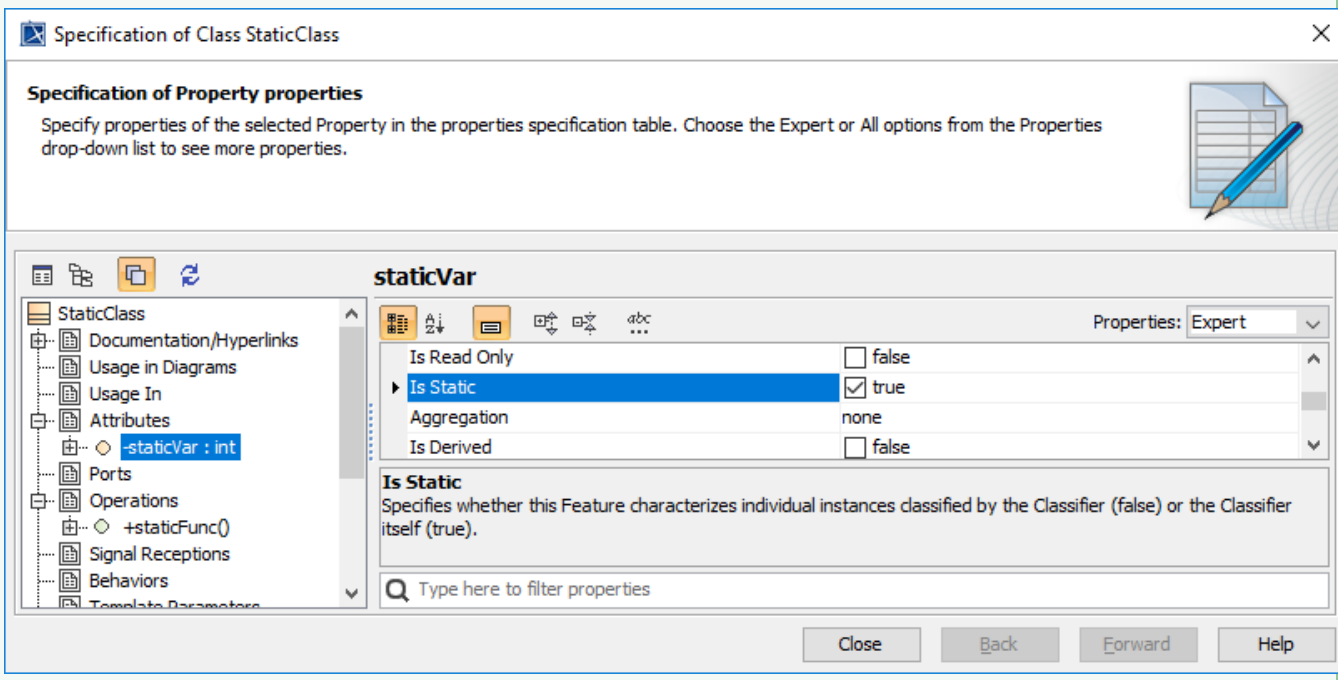
C++ source code

```
class StaticClass {
 static int staticVar;
 static void staticFunc();
};
```

UML model:



UML model Tag specification:



Property *Is Static*

Pure virtual function and abstract class

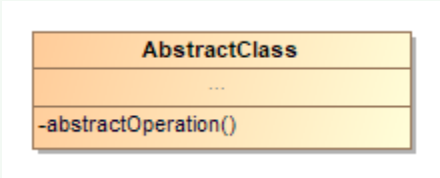
Pure virtual C++ function is mapped to UML operation with property *Is Abstract* set to true. If one or more functions are abstract in a class, the property *Is Abstract* of the UML class is set to true. Constraint: if no operation is abstract, the class cannot be abstract.

Example

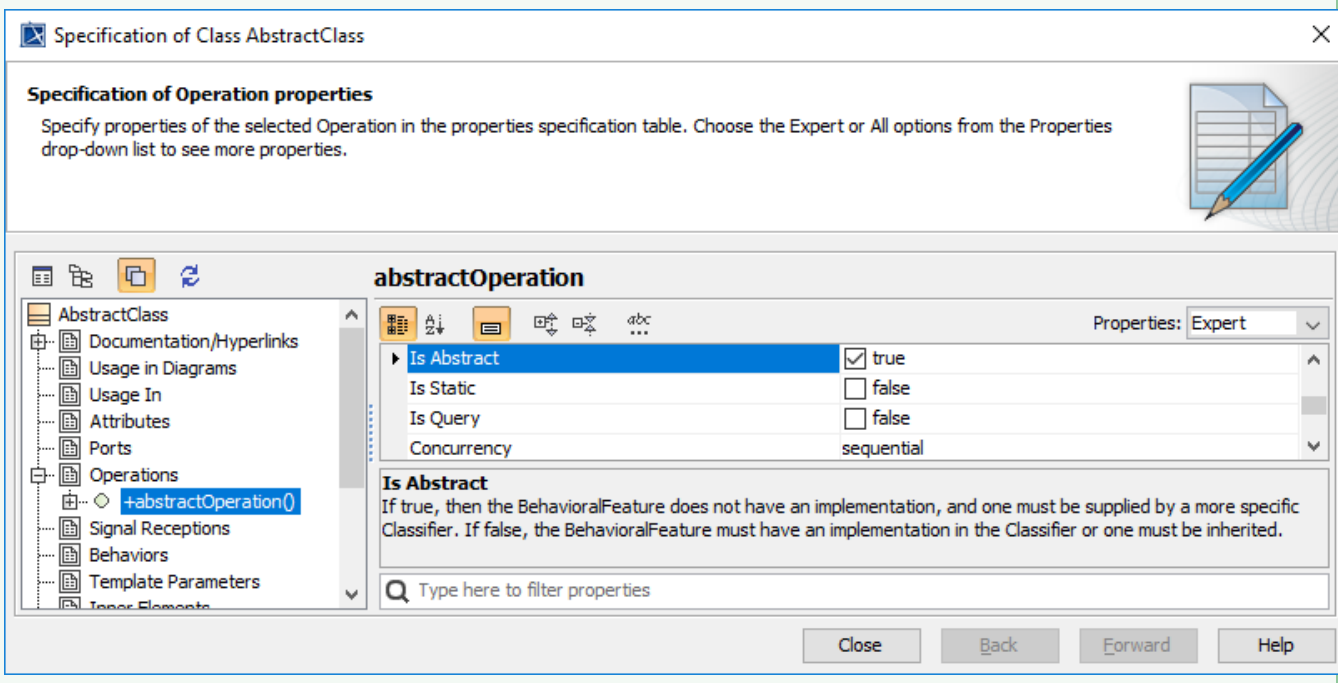
C++ source code

```
class AbstractClass {
 virtual abstractOperation()=0;
};
```

UML model:



UML model Tag specification:



Operation property Is Abstract

Friend Declaration

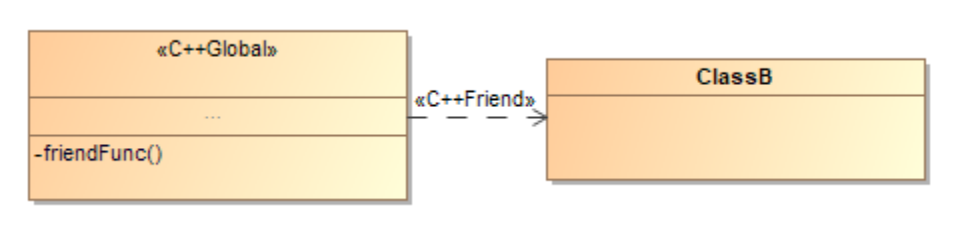
C++ friend function is mapped with a «C++Friend» stereotyped dependency relationship between the function (an UML operation) and the friendClass. This relationship grants the friendship to the friendClass.

Example

C++ source code

```
class ClassB {
public:
 friend void friendFunc();
};
void friendFunc();
```

UML model:



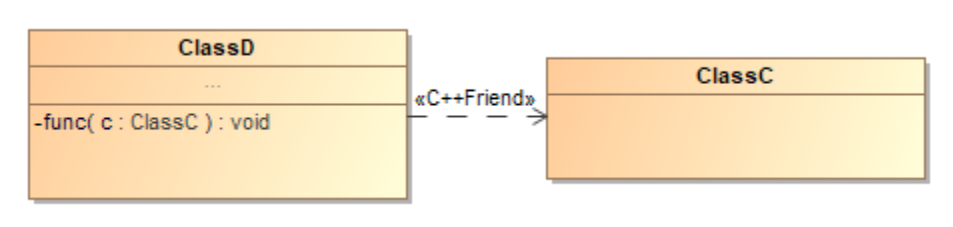
C++ friend member function is mapped with a «C++Friend» stereotyped dependency relationship between the member function and the friend class. This relationship grants the friendship to the friend class.

Example

C++ source code

```
class ClassD {
void func(ClassC c);
};
class ClassC {
    friend void ClassD::func(ClassC
c);
};
```

UML model:



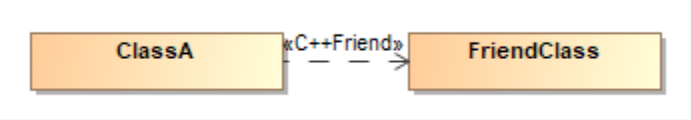
C++ friend class are mapped with a «C++Friend» stereotyped dependency relationship between the class and the friend class. This relationship grants the friendship to the friend class.

Example

C++ source code

```
class FriendClass {
public:
 friend class ClassA;
};
class ClassA {
};
```

UML model:



Struct

C++ struct are mapped to a UML class with stereotype «C++Struct». See **C++Struct** for more info.

Current Modeling tool

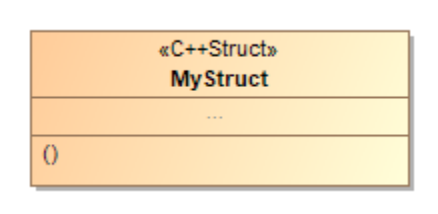
The current version of MD use class's language property "Class Key"

Example

C++ source code

```
struct MyStruc {
};
```

UML model:



Union

C++ union is mapped to a UML class with stereotype «C++Union». See C++Union for more info.

Current Modeling tool

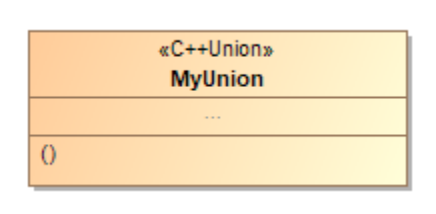
The current version of MD use class's language property "Class Key"

Example

C++ source code

```
union MyUnion {
};
```

UML model:



Enumeration

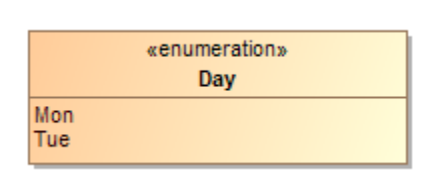
C++ enum is mapped to UML enumeration. C++ enum fields are mapped to UML enumeration literals. C++ enum field with a specified value is mapped to tag value of «C++LiteralValue» stereotype.

Example

C++ source code

```
enum Day {
     Mon,
     Tue=2
};
```

UML model:



Typedef

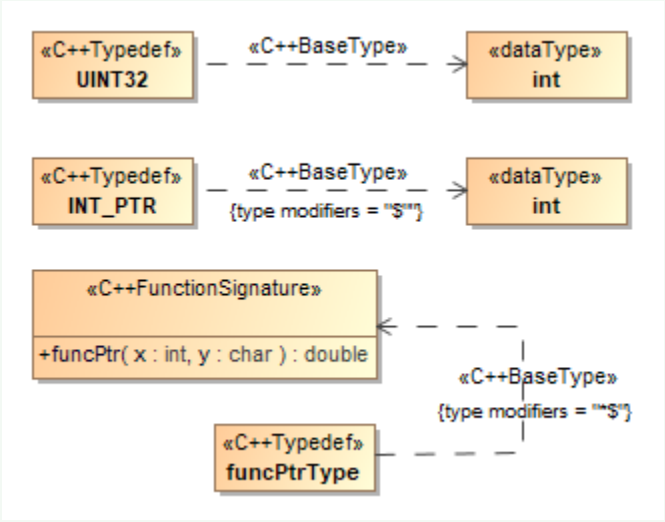
C++ typedef is mapped to a class with «C++Typedef» stereotype. A «C++BaseType» dependency links to the original type. Type modifiers tag of «C++BaseType» dependency is used to define type modifiers. *$* character is replaced by the type name. A typedef on a function pointer is mapped by linking a «C++BaseType» dependency to an operation and type modifiers tag of «C++BaseType» dependency is set to **$*. Operation signature can be stored in a «C++FunctionSignature» class.

Example

C++ source code

```
typedef int UINT32;
typedef int* INT_PTR;
typedef double (*funcPtrType)(int, char);
;
```

UML model:



Namespace

C++ namespace is mapped to a UML package with the stereotype «C++Namespace». Unnamed namespace is named unnamed+index number of unnamed namespace (start at 1), and unique namespace name tag is set to the source file path+:+index number of unnamed namespace (start at 0).

```
namespace n {
     namespace m {
     }
}
```

Global Functions And Variables

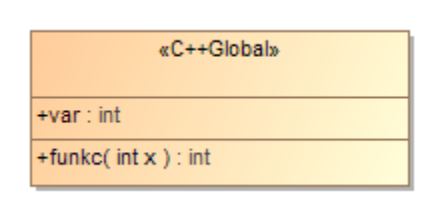
Global functions and variables are mapped to operations and attributes into an unnamed class with stereotype «C++Global». «C++Global» class resides in its respective namespace, or in a top package.

Example

C++ source code

```
int var;
int func(int x);
```

UML model:

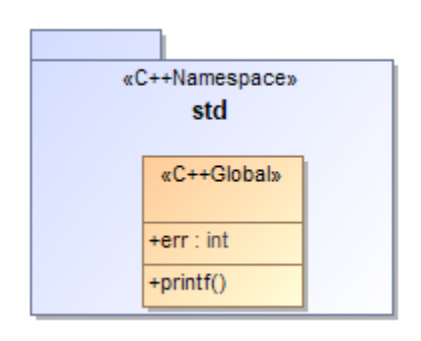


Example

C++ source code

```
namespace std {
    int err;
    void printf();
}
```

UML model:



Class Definition

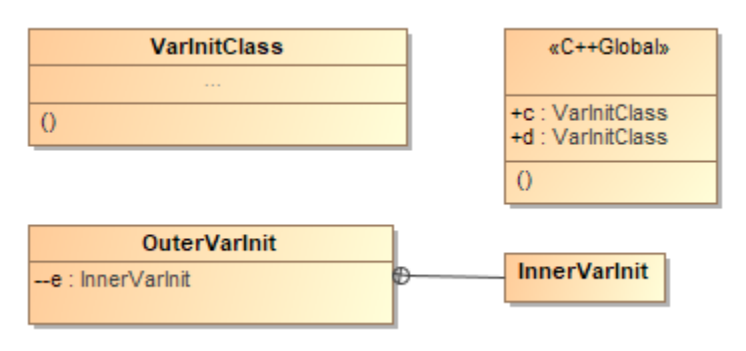
Variables can be created after a class/struct/union declaration. These variables are mapped to UML attribute, and placed in their respective namespace /global/class container.

Example

C++ source code

```
class VarInitClass {
} c, d;
class OuterVarInit {
     class InnerVarInit {
     } e;
};
```

UML model:



Class Template Definition

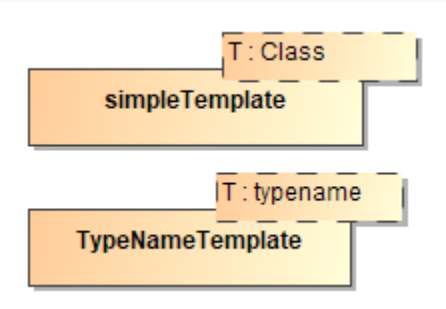
C++ template class is mapped to UML class with template parameters properties added. Type of template parameter is always set to UML Class. To generate/reverse typename keyword, type keyword tag is set to typename.

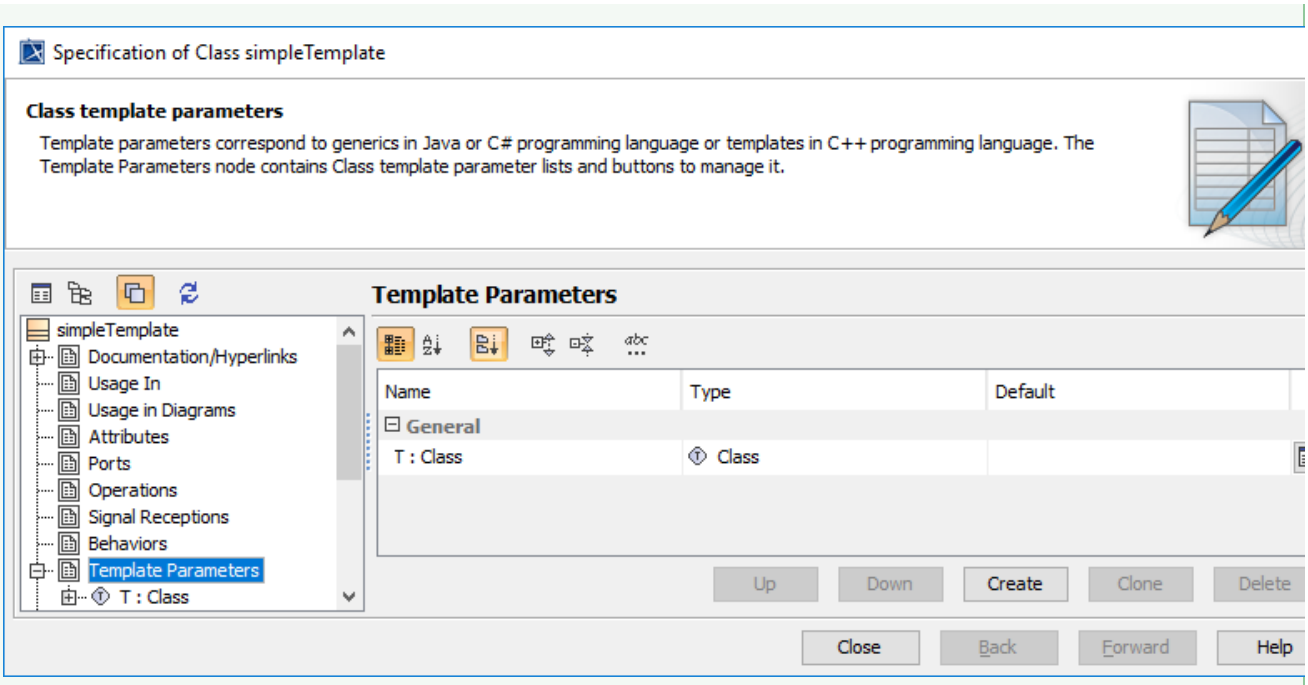
Example

C++ source code

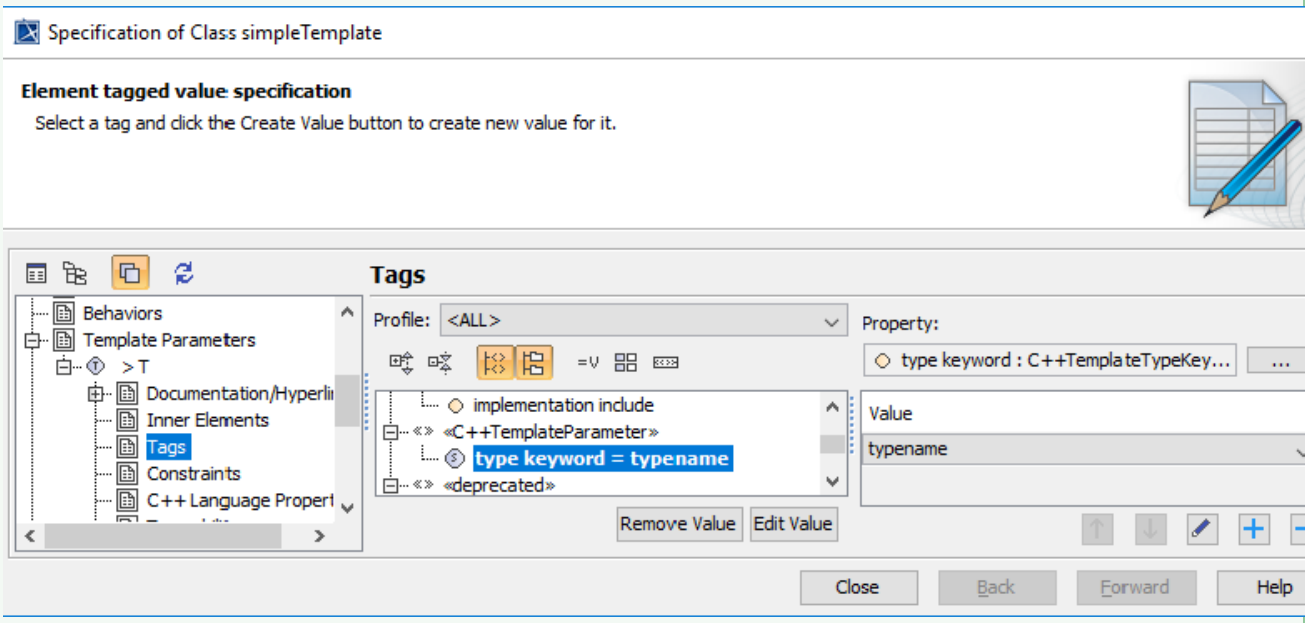
```
template <class T>
class simpleTemplate {
};
template <typename T>
class TypeNameTemplate {
};
```

UML model:





Template Parameters



Template Parameter Tag type keyword.

Function Template Definition

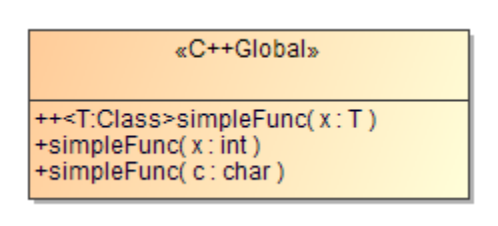
C++ template function is mapped to UML operation with template parameters properties added. C++ template function overload is mapped to a normal function. (the same name with the same number of parameter, but different type of parameter) New style of template function overloading is mapped to a normal function. (the same name with the same number of parameter, but different type of parameter) and a template binding relationship is created between the overload operation and the template operation, with specific template parameter substitutions.

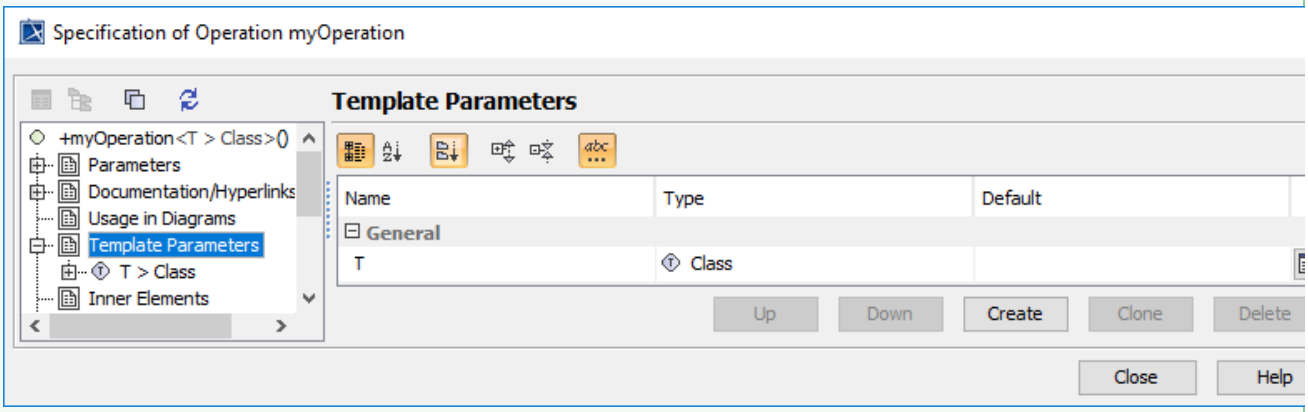
Example

C++ source code

```
template <class T>
void simpleFunc(T x);
// overload old style
void simpleFunc(int x);
// overload new style
template<>
void simpleFunc<char>(char x);
```

UML model:





Template Parameters

Default Template Parameter

C++ default template parameter is mapped to UML default template parameters. Instantiation using the default template parameter is mapped using a template binding relationship with an empty actual property for the template parameter substitution.

```
template <class T=int>
class defaultTemplate {
};
```

Template Instantiation

Template instantiation are mapped to template binding relationship between the template class and the instantiate class, the template parameter substitution of the binding relationship is set using the template argument.

```
template <class T>
class simpleTemplate {
};
simpleTemplate<int>
simpleTemplateInstance;
```

For template argument using template instantiation as argument, an intermediate class is created with the specific binding.

```
template <class T>
class T1Class {
};
template <class T>
class T2Class {
};
T1Class<T2Class<int>> ...
```

For template argument using multiple template instantiations in an inner class (b<int>::c<char>), the intermediate class instance is created in the outer class instance.

```
template <class T>
class b {
 template <class T>
 class c {
 };
};
b<int>::c<char> ...
```

Example of complex template instantiation. Containment relationship are placed on diagram for information only, these relationships are not created during a reverse process. Containment relationship is modeled by placing a class into a specific class/package. See Containment tree below the diagram.

Partial Template Instantiation

C++ partial template instantiation use the same mapping as Template Instantiation and the unbinded parameter is binded to the specific template parameter class.

```
template <class T,class U,class V>
class PT {};
template <class A,class B>
class PT<B, int, A> {};
```

Template Specialization

C++ Template specialization uses the same mapping as Template Instantiation.

```
template <class T>
class TS {};
template <>
class TS<int> {};
```

Forward Class Declaration

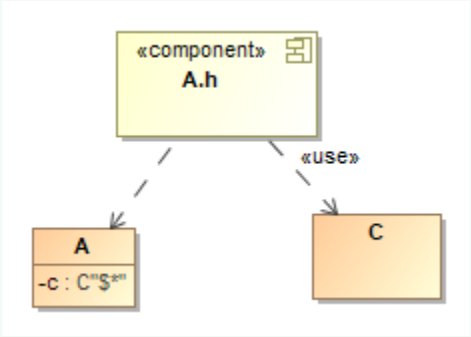
The example code is declared in A.h file. The file component A.h has the «use» association applied by «C++Include» stereotype with *Class Forward* tag value.

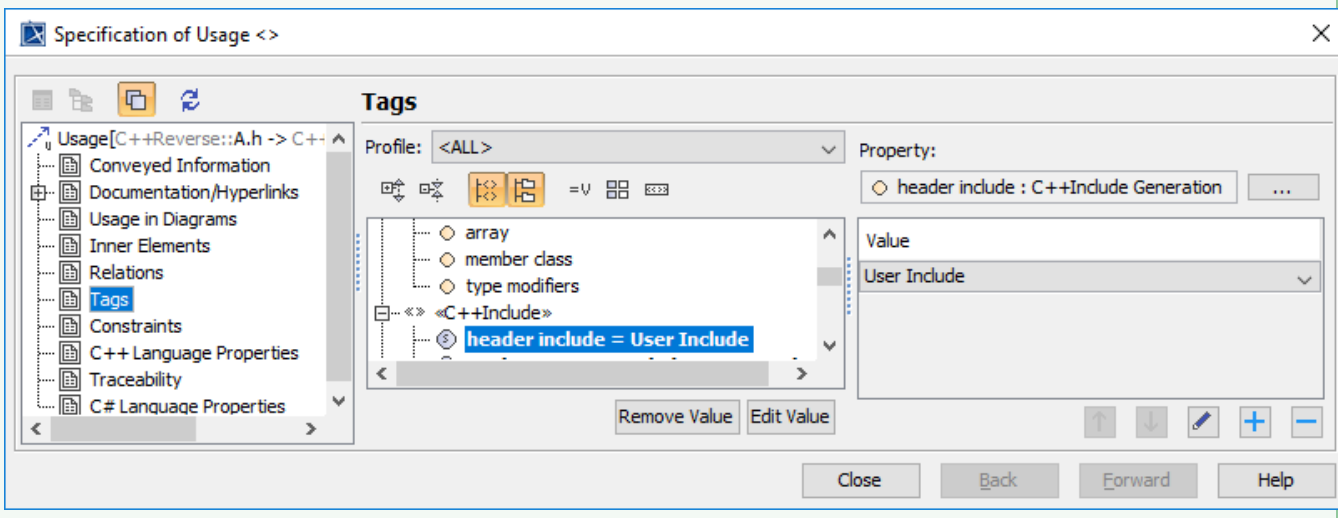
Example

C++ source code

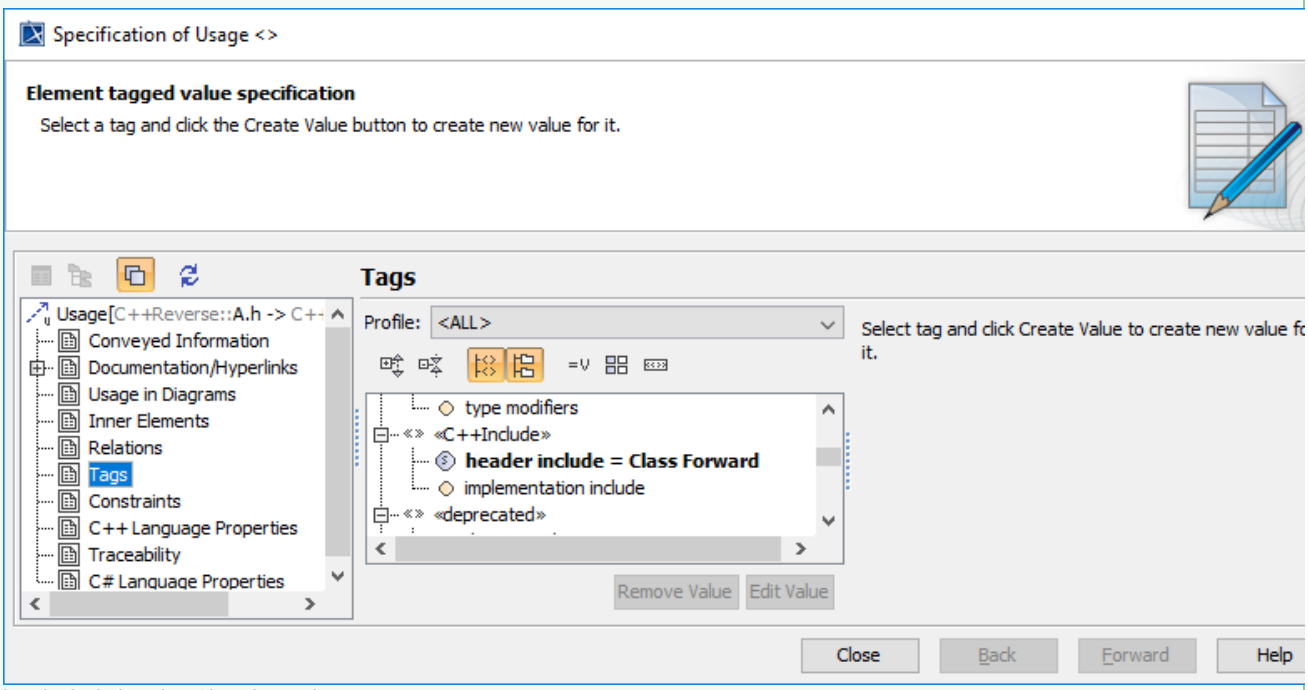
```
class C;
class A {
 private:
 C* c;
};
```

UML model:





Template Parameters



header include value *Class forward*

Include Declaration

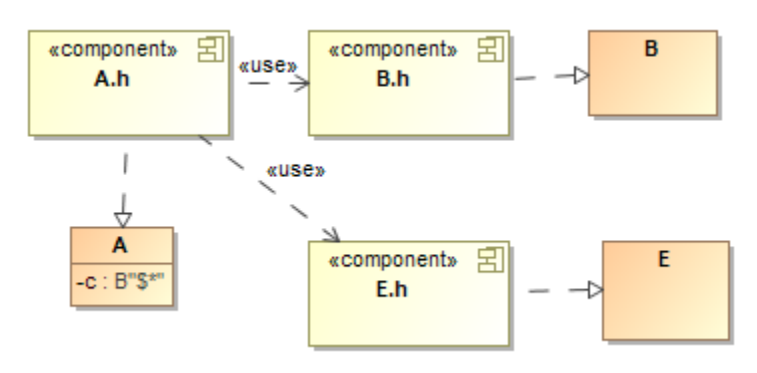
The example code is declared in A.h file. The «use» association is also applied to C++Include stereotype shown in **Forward class declaration**.

Example

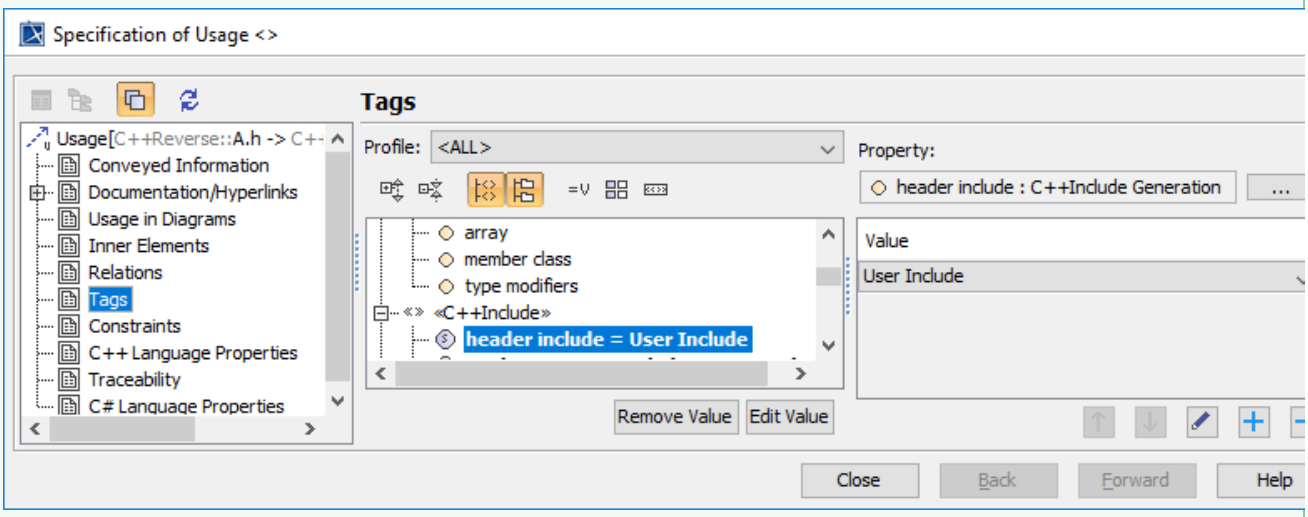
C++ source code

```
#include "B.h"
#include <E.h>
class A {
 private:
 B* b;
 E* e;
};
```

UML model:

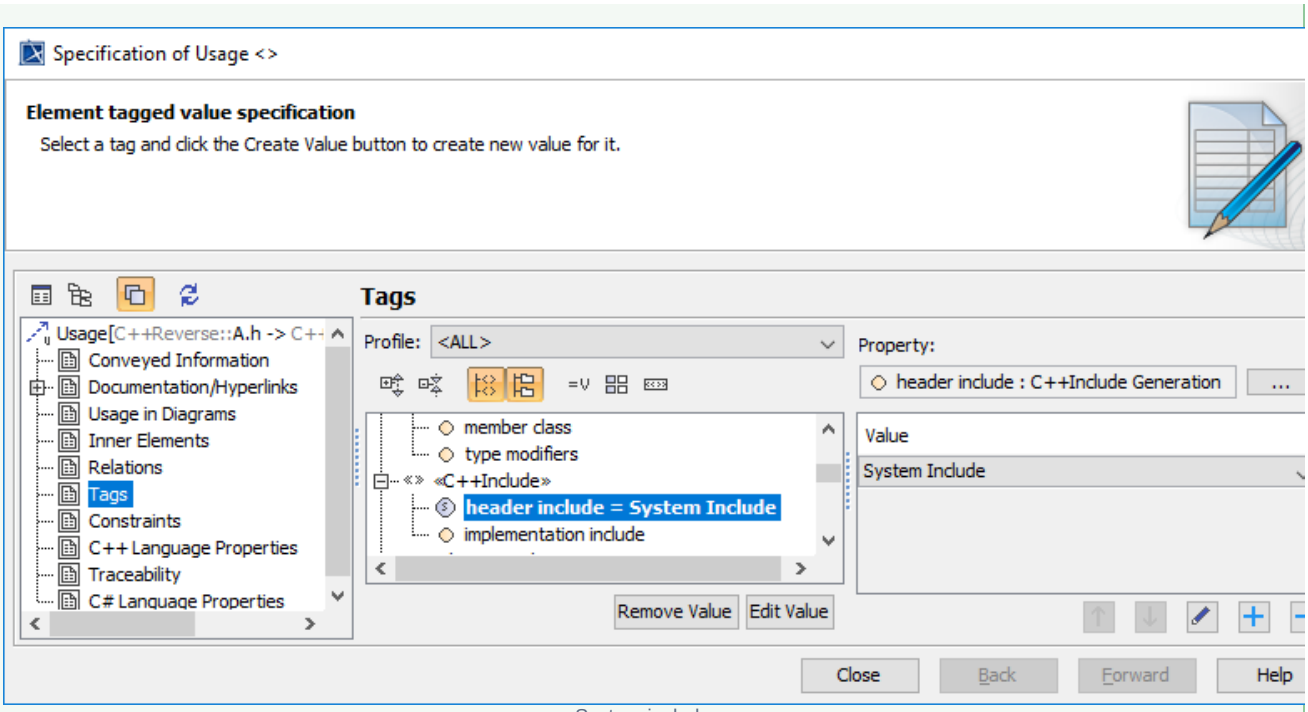


Specification for #include "B.h":



User Include

Specification for #include <E.h>:



System include

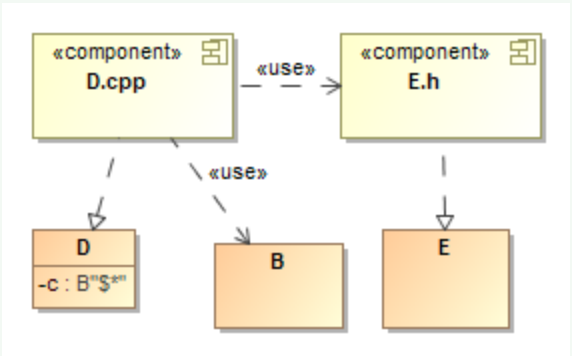
The example code is declared in D.cpp file.

Example

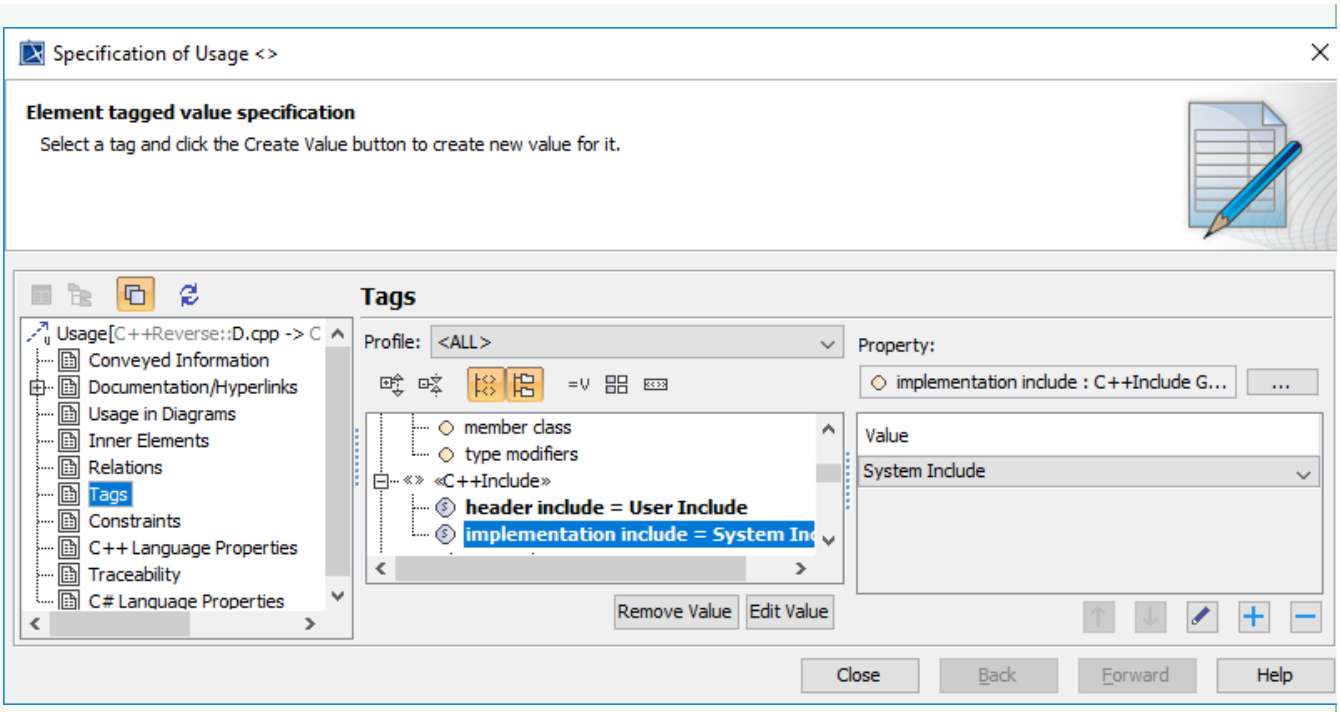
C++ source code

```
#include <E.h>
class B;
class D {
private:B* b;
E* e;
}
```

UML model:

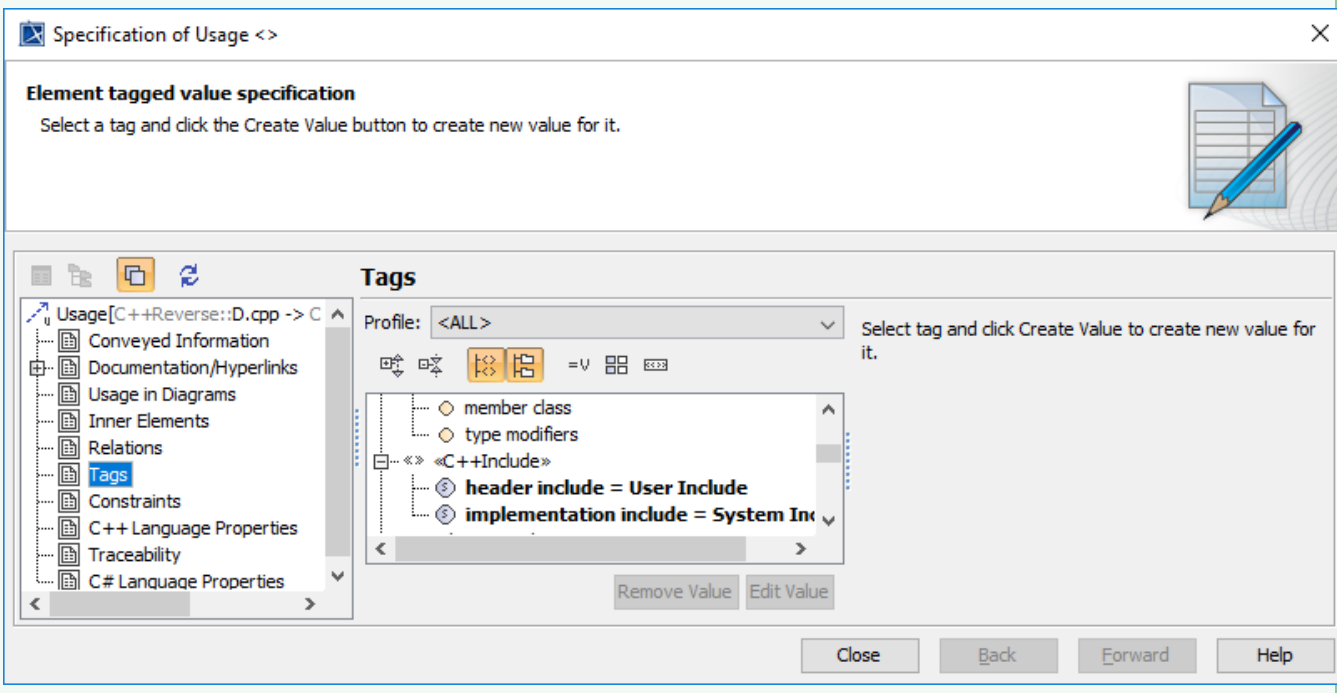


Specification for #include "B.h":



User Include

Specification for #include <E.h>:



System include