

UML nach IDL/OCL Konverter

Robert Lechner

23. Februar 2004

Inhaltsverzeichnis

1	Einleitung	2
1.1	Anwendung	2
1.2	UML	2
1.3	Besonderheiten	2
1.4	zyklische Abhängigkeiten	3
2	IDL-Grammatik und UML-Darstellung	4
2.1	BNF	4
2.2	module	5
2.3	interface	6
2.4	valuetype	8
2.5	typedef	11
2.6	const	12
2.7	struct	13
2.8	union	14
2.9	enum	16
2.10	exception	17
2.11	sequence	18
2.12	array	20
2.13	operation	22
2.14	attribute	23
2.15	component	25
2.16	event	28
2.17	home	29
3	OCL	31

1 Einleitung

1.1 Anwendung

Das Paket `uml2idl` beinhaltet einen UML→IDL/OCL Konverter. Der Zweck besteht darin, Komponenten und Interfaces (IDL ¹) zusammen mit der formalen Definition der Schnittstellen (OCL ²) in einem UML-Editor zu modellieren. Der Konverter generiert daraus die für die `ccmtools` benötigten IDL- und OCL-Dateien. Als UML-Parser wird das vom Tool `dtd2java` ³ erzeugte Package `uml_parser` verwendet.

Gestartet wird der Konverter mit dem Aufruf:

```
java uml2idl.Main UML-file IDL/OCL-prefix
```

UML-file ...Name der XMI-Datei ⁴

IDL/OCL-prefix ...Prefix für die Namen der IDL- und OCL-Dateien

1.2 UML

Die Komponenten und Schnittstellen werden mit Hilfe des UML profile for CORBA modelliert. Als Grundlage dient die entsprechende OMG-Spezifikation ⁵.

In Abschnitt 2 wird die Anwendung genauer erklärt.

1.3 Besonderheiten

- Anonyme Sequenzen und Felder bekommen einen Namen mit dem Prefix `Anonymous` und es wird (wie bei normalen Sequenzen und Feldern) eine `typedef`-Anweisung erzeugt.
- Die Vorgabe der Reihenfolge mit dem *tagged value* `IDLOrder` ist unnötig (und wird auch nicht unterstützt), da die richtige Reihenfolge automatisch berechnet wird. Es kann aber nötig sein, implizite Abhängigkeiten über eine *dependency* explizit zu deklarieren.
- Zyklische Abhängigkeiten können nicht aufgelöst werden!
Das ist aber mit dem *tagged value* `IDLOrder` ebenfalls nicht möglich.
Siehe Abschnitt 1.4.
- *fixed types* werden derzeit nicht unterstützt.

¹Interface Definition Language; Teil der CORBA-Spezifikation:

http://www.omg.org/technology/documents/formal/corba_iiop.htm

<http://www.omg.org/technology/documents/formal/components.htm>

²Object Constraint Language; Teil der UML-Spezifikation

³Paket `dtd2java`: erzeugt Java-Modelle aus DTD-Dateien

⁴XML Metadata Interchange: XML-Dateien zum Speichern von UML-Modellen

⁵UML Profile for CORBA Specification, Version 1.0, April 2002

UML Profile for CCM, RFP Version 2, May 2003

1.4 zyklische Abhängigkeiten

In Abbildung 1 ist eine Abhängigkeit zu sehen, die nicht aufgelöst werden kann. Um diese Struktur in IDL zu erhalten, muss entweder C3 oder C2 als „forward declaration“ definiert werden. Die Berechnung solcher spezieller Deklarationen ist in `uml2idl` nicht implementiert. Es stellt sich aber die Frage, ob solche Abhängigkeiten nicht auf einen Designfehler hinweisen.

Keine Schwierigkeiten stellen die Abhängigkeiten in Abbildung 2 dar.

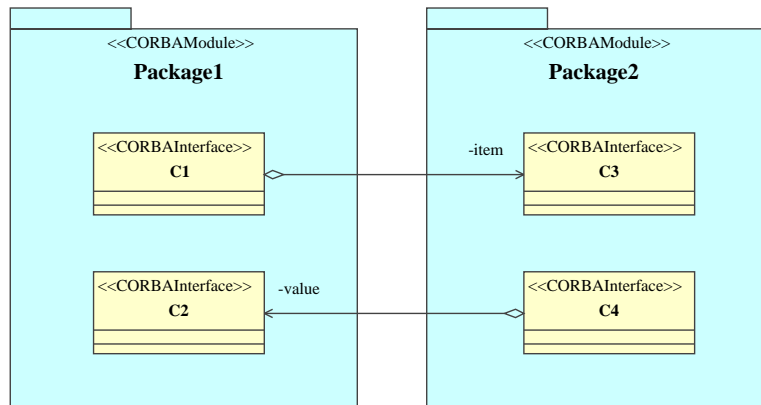


Abbildung 1: zyklische Abhängigkeit

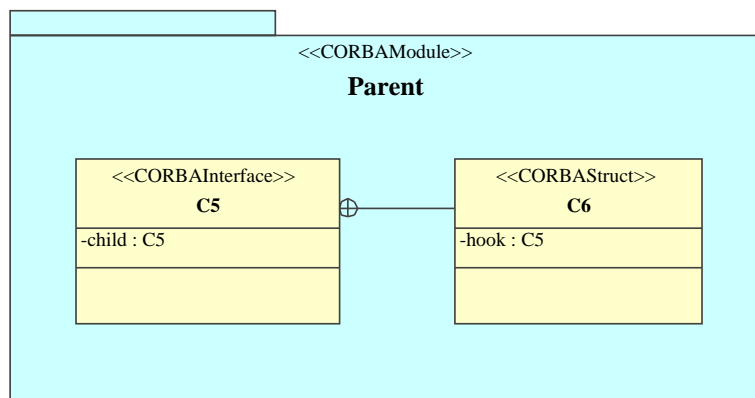


Abbildung 2: Abhängigkeiten ohne Probleme

2 IDL-Grammatik und UML-Darstellung

2.1 BNF

`<specification> ::= <import>* <definition>+`

`<import> ::= "import" (<scoped_name> | string_literal) ";"`

`<definition> ::= <type_dcl>
 | <const_dcl>
 | <except_dcl>
 | <interface>
 | <module>
 | <value>
 | <type_id_dcl>
 | <type_prefix_dcl>
 | <event>
 | <component>
 | <home_decl>`

`<export> ::= <type_dcl>
 | <const_dcl>
 | <except_dcl>
 | <attr_dcl>
 | <op_dcl>
 | <type_id_dcl>
 | <type_prefix_dcl>`

2.2 module

IDL3

`<module> ::= "module" <identifier> "{" <definition>+ "}";`

UML

package mit *stereotype* CORBAModule

Beispiel

```
module docModule {  
    module Parent {  
        module Child {  
        };  
    };  
};
```

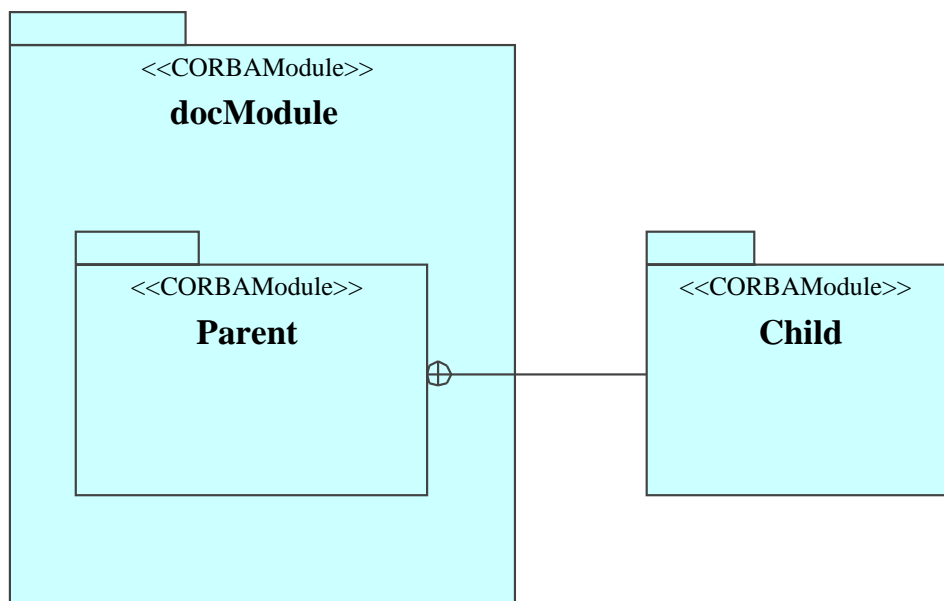


Abbildung 3: Zwei Möglichkeiten um Module zu verschachteln.

2.3 interface

IDL3

```
<interface> ::= <interface_dcl>
               | ["abstract" | "local"] "interface" <identifier> ";"

<interface_dcl> ::= <interface_header> "{" <export>* "};"

<interface_header> ::= ["abstract" | "local"] "interface" <identifier>
                       [":" <interface_name> {"," <interface_name>*}]
```

UML

class mit *stereotype* CORBAInterface
“abstract”: abstrakte Klasse
“local”: *tagged value* isLocal = TRUE
Vererbung: normale Klassen-Vererbung

Beispiel

```
module docInterface {
    local interface I1 {
    };

    abstract interface I2 {
    };

    interface I4 {
    };

    interface I3 : I2, I4 {
        struct S1 {
            long value;
        };

        attribute string myAttr;
        attribute S1 myStruct;
        long myOp(in double param);
    };
};
```

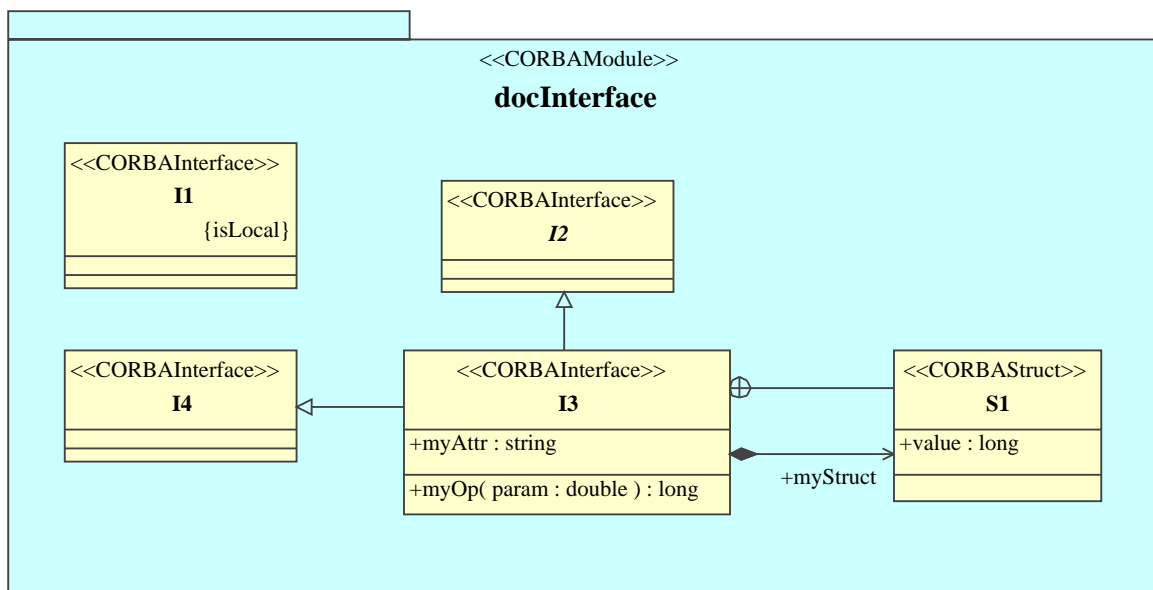


Abbildung 4: UML-Klassendiagramm zu Abschnitt 2.3

2.4 valuetype

IDL3

```
<value> ::= (<value_dcl> | <value_abs_dcl> | <value_box_dcl>)
          | ["abstract"] "valuetype" <identifier> ";"

<value_box_dcl> ::= "valuetype" <identifier> <type_spec> ";"

<value_abs_dcl> ::= "abstract" "valuetype" <identifier>
                  [<value_inheritance_spec>] "{" <export>* "};"

<value_dcl> ::= <value_header> "{" <value_element>* "};"

<value_header> ::= ["custom"] "valuetype" <identifier>
                  [<value_inheritance_spec>]

<value_inheritance_spec> ::= [":" ["truncatable"] <value_name> {"", <value_name>}*]
                           ["supports" <interface_name> {"", <interface_name>}*]

<value_element> ::= <export> | <state_member> | <init_decl>

<state_member> ::= ("public"|"private") <type_spec> <declarators> ";"

<init_decl> ::= "factory" <identifier> "(" [<init_param_decls>] ")"
              [<raises_expr>] ";"

<init_param_decls> ::= <init_param_decl> {"", <init_param_decl>}*

<init_param_decl> ::= "in" <param_type_spec> <simple_declarator>
```

UML

- *class* mit *stereotype* CORBAValue (Ausnahmen: “custom“ und *boxed value type*)
 - “custom“: *stereotype* CORBACustomValue
 - *boxed value type*: *stereotype* CORBACustomValue und Vererbung
- “abstract“: abstrakte Klasse
- Vererbung: normale Klassen-Vererbung
- “truncatable“: Vererbung mit *stereotype* CORBATruncatable
- “supports“: Vererbung mit *stereotype* CORBAValueSupports
- “factory“: *operation* mit *stereotype* CORBAValueFactory

Beispiel

```
module docValuetype {
    struct Date {
        string value;
    };

    interface PrettyPrint {
        string print();
    };

    valuetype Time {
        public short hour;
        public short minute;
    };

    valuetype DateAndTime : Time supports PrettyPrint {
        private Date the_date;

        factory init(in short hr, in short min);
        Date getDate();
    };

    valuetype BoxedDate Date;
};
```

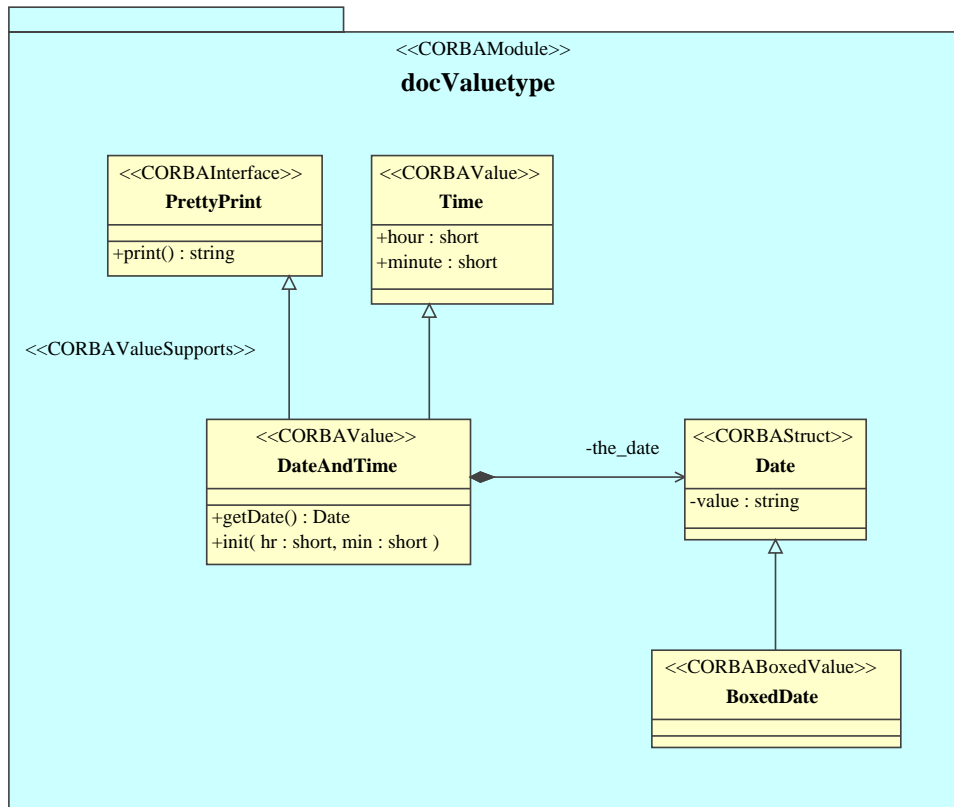


Abbildung 5: value type

2.5 typedef

IDL3

```
<type_dcl> ::= "typedef" <type_spec> <declarators> ";"  
            | (<struct_type> | <union_type> | <enum_type>)  
            | "native" <identifier> ";"  
            | ("struct" | "union") <identifier> ";"
```

```
<declarators> ::= <declarator> {"," <declarators>}*
```

```
<declarator> ::= <identifier> | <array_declarator>
```

UML

class oder *datatype* mit *stereotype* CORBATypedef
und Vererbung mit dem gewünschten Typ.

Beispiel

```
module docTypedef {  
    interface Interface1 {  
    };  
  
    typedef Interface1 MyInterface1;  
  
    typedef string MyString;  
};
```

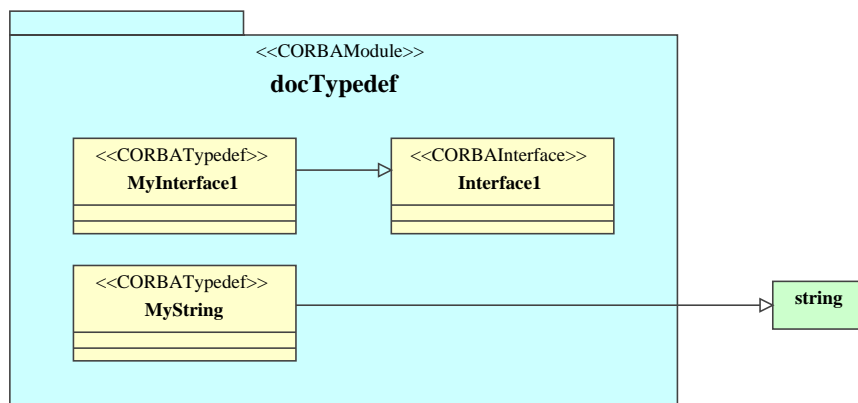


Abbildung 6: type definition

2.6 const

IDL3

`<const_dcl> ::= "const" <const_type> <identifier> "=" <const_exp> ";"`

UML

attribute mit Stereotype **CORBAConstant**

`<const_exp>`: *initial value* des Attributs

Globale Konstanten oder Konstanten eines Moduls:

class mit *stereotype* **CORBAConstants** und dem Namen **Constants**; die Konstanten werden wie oben als Attribute mit Stereotype **CORBAConstant** definiert.

Abhängigkeiten werden als einfache *dependencies* dargestellt.

Beispiel

```
module docConst {  
    const short S = 3;  
  
    interface A {  
        const long L = S+20;  
    };  
};
```

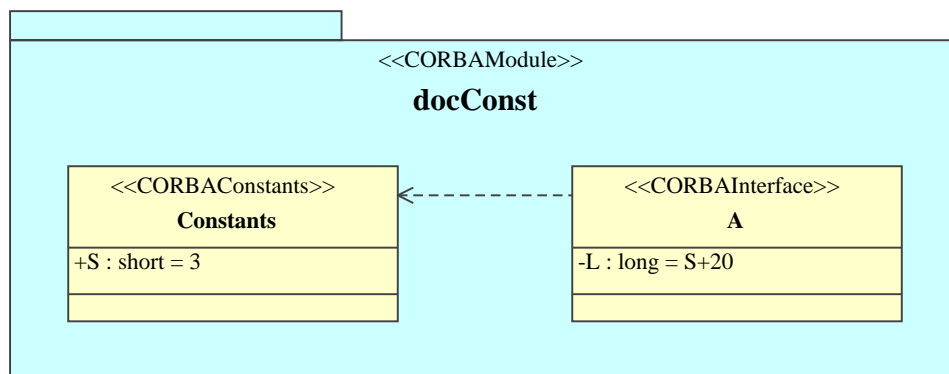


Abbildung 7: Konstanten mit Abhängigkeit

2.7 struct

IDL3

`<struct_type> ::= "struct" <identifier> "{" <member>+ "}";`

`<member> ::= <type_spec> <declarators> ";"`

UML

class mit *stereotype* CORBAStruct

Beispiel

```
module docStruct {  
    struct A {  
        struct B {  
            short k;  
            long j;  
        };  
  
        string q;  
        B p;  
    };  
};
```

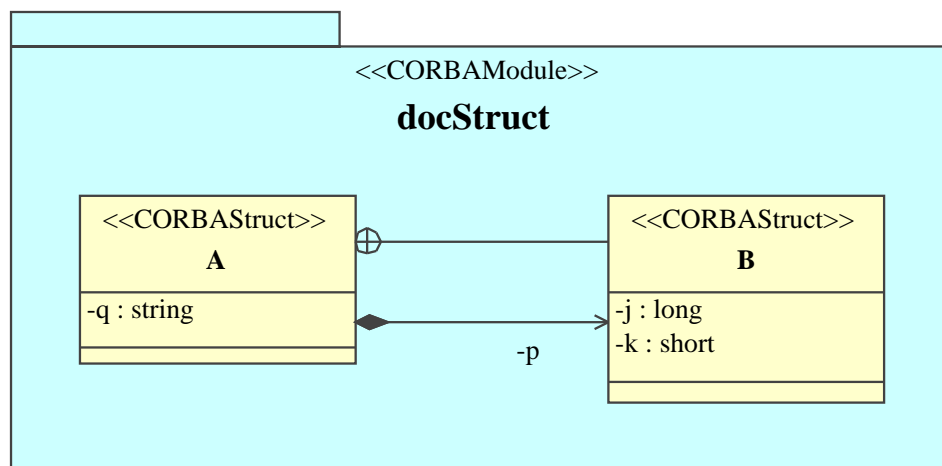


Abbildung 8: zwei verschachtelte Strukturen

2.8 union

IDL3

```
<union_type> ::= "union" <identifier> "switch"
               "(" <switch_type_spec> ")"
               "{" <case>+ "}";

<case>        ::= <case_label>+ <type_spec> <declarator> ";

<case_label> ::= "case" <const_exp> ":"
               |  "default:"
```

UML

class mit *stereotype* CORBAUnion

<switch_type_spec>: *association* zum Typ mit *stereotype* switchEnd oder Attribut mit *stereotype* switch

“case“: Attribut oder *association* mit *tagged value* Case=LabelName

“default“: Attribut oder *association* mit *tagged value* Case=default

Beispiel

```
module docUnion {
    enum Contents {
        INTEGER_CL,
        FLOAT_CL,
        DOUBLE_CL,
        COMPLEX_CL,
        STRUCTURED_CL
    };

    union Reading switch(Contents) {
        case INTEGER_CL: long a_long;
        case FLOAT_CL: case DOUBLE_CL: double a_double;
        default: any an_any;
    };

    struct PropertyValue {
        string value;
    };

    union ValOpt switch(boolean) {
        case TRUE: PropertyValue pv;
    };
};
```

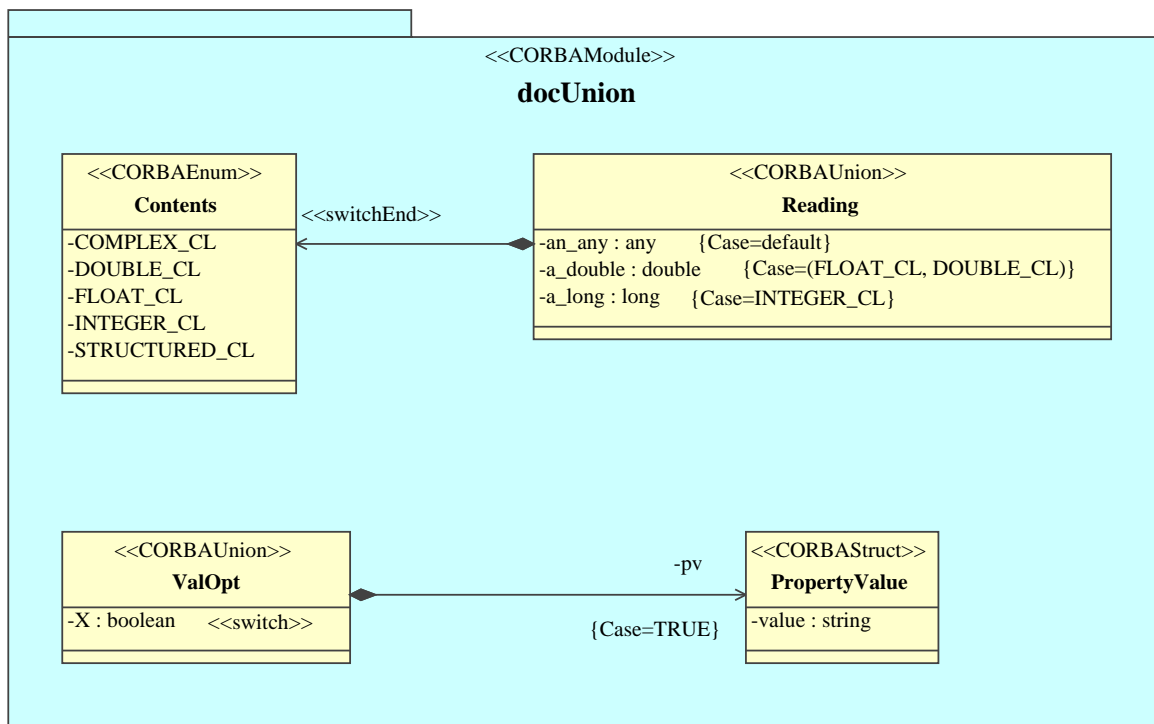


Abbildung 9: union

2.9 enum

IDL3

`<enum_type> ::= "enum" <identifier> "{" <identifier> {"," <identifier>}* "};"`

UML

class mit *stereotype* CORBAEnum

Die Elemente sind Attribute; es wird nur der Name verwendet.

Beispiel

```
module docEnum {  
    enum Types {  
        TYPE_INTEGER,  
        TYPE_FLOAT,  
        TYPE_COMPLEX  
    };  
  
    struct Value {  
        string name;  
        Types type;  
    };  
};
```

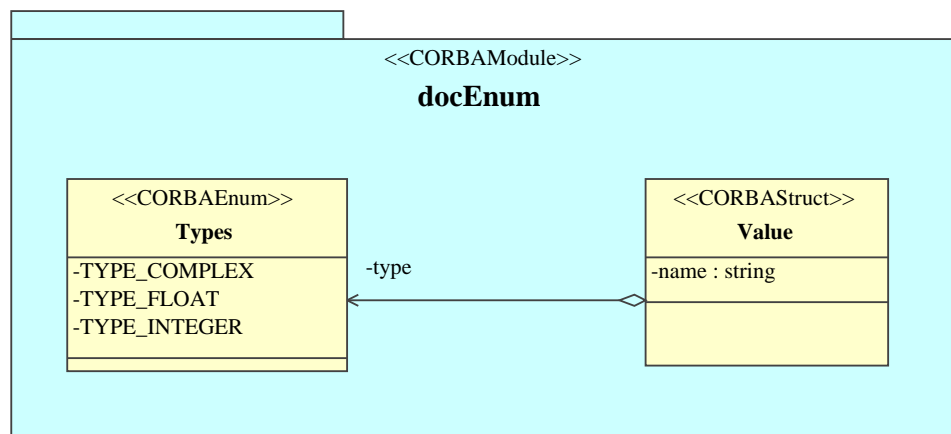


Abbildung 10: enumeration

2.10 exception

IDL3

`<except_dcl> ::= "exception" <identifier> "{" <member>* "}";`

`<member> ::= <type_spec> <declarators> ";"`

UML

class mit *stereotype* CORBAException

Beispiel

```
module docException {  
    interface Tex {  
        exception Badness2000 {  
            string err_msg;  
        };  
  
        void process_token(in string tok) raises(Badness2000);  
    };  
};
```

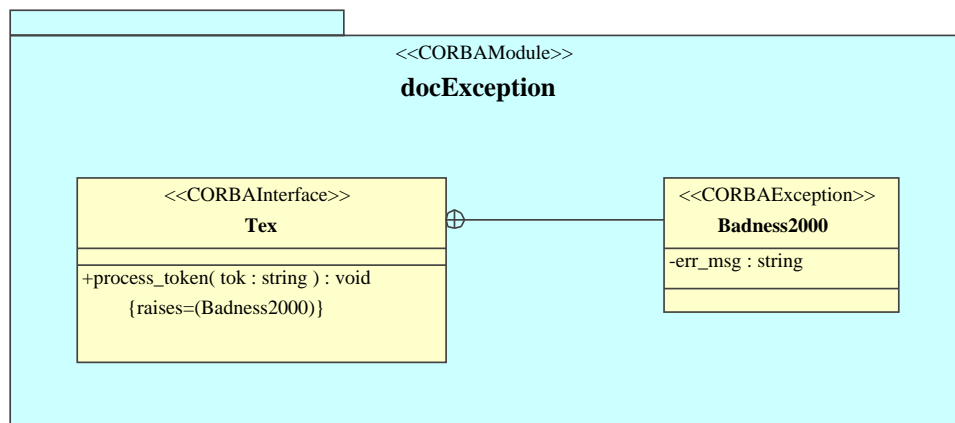


Abbildung 11: exception

2.11 sequence

IDL3

```
<sequence_type> ::= "sequence<" <simple_type_spec> ">"  
                  |  "sequence<" <simple_type_spec> "," <positive_int_const> ">"
```

UML

class mit *stereotype* CORBASequencce oder CORBAAnonymousSequence

association zum Typ mit *qualifier* vom Typ **long** und der gewünschten Vielfachheit

Beispiel

```
module docSequence {  
    typedef sequence< short > ShortSequence;  
  
    typedef sequence< float > Anonymous1;  
  
    typedef sequence< Anonymous1 > FloatMatrix;  
  
    struct Struct1 {  
        double value;  
    };  
  
    typedef sequence< Struct1 , 5 > Struct1Seq;  /* range: 0..4 */  
};
```

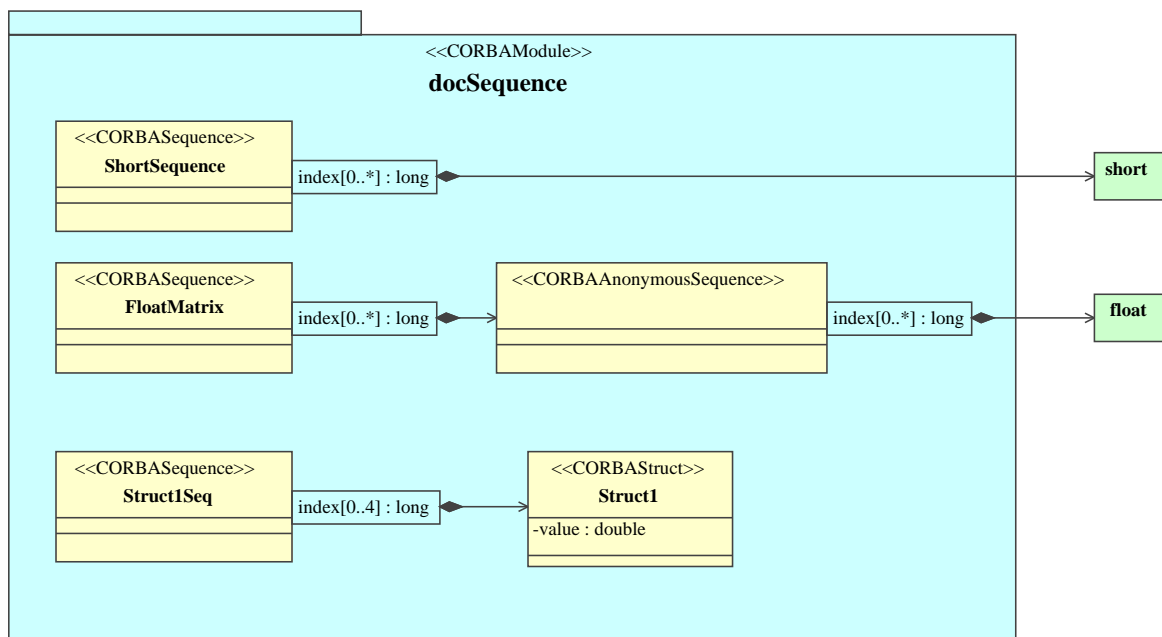


Abbildung 12: sequence

2.12 array

IDL3

`<array_declarator> ::= <identifier> <fixed_array_size>+`

`<fixed_array_size> ::= "[" <positive_int_const> "]"`

UML

class mit *stereotype* CORBAArray oder CORBAAnonymousArray

association zum Typ mit *qualifieres* vom Typ **long** und den gewünschten Vielfachheiten

Beispiel

```
module docArray {
    struct MyStruct {
        long value;
    };

    typedef MyStruct MyArray[5][10];

    typedef MyStruct Anonymous2[4];

    struct Data1 {
        Anonymous2 field;
    };
};
```

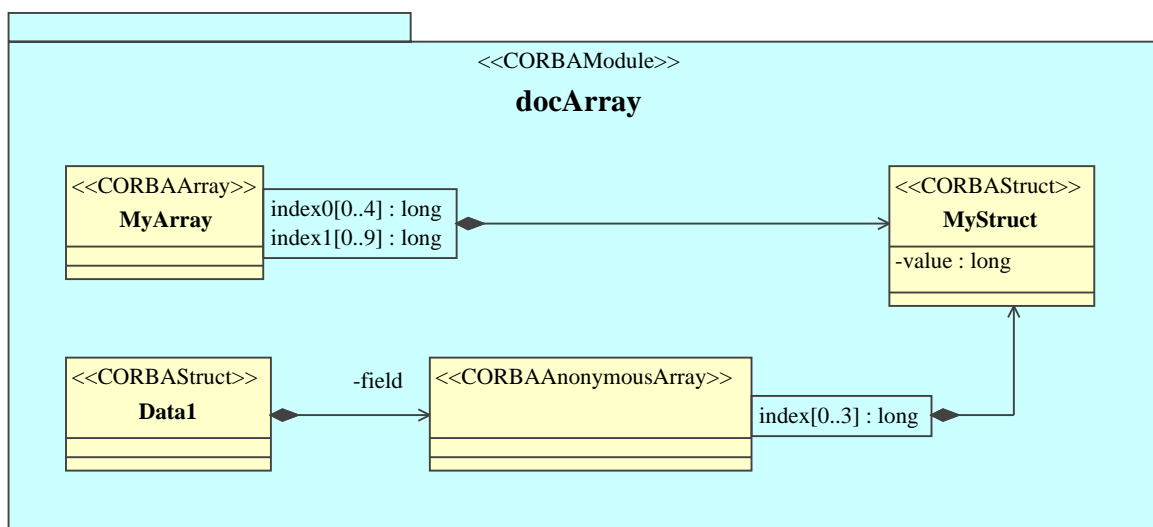


Abbildung 13: ein- und mehrdimensionale Felder

2.13 operation

IDL3

```
<op_dcl> ::= ["oneway"] <op_type_spec> <identifier> <parameter_dcls>  
           [<raises_expr>] [<context_expr>] ";"  
  
<op_type_spec> ::= <param_type_spec> | "void"  
  
<parameter_dcls> ::= "(" <param_dcl> {"," <param_dcl>}* ")"  
                  | "()"   
  
<param_dcl> ::= ("in" | "out" | "inout")  
               <param_type_spec> <simple_declarator>  
  
<raises_expr> ::= "raises(" <scoped_name> {"," <scoped_name>}* ")"  
  
<context_expr> ::= "context(" <string_literal> {"," <string_literal>}* ")"
```

UML

UML-*operation*

“oneway”: *stereotype oneway*

“raises”: *tagged value raises=(exception1, exception2, ...)*

“context”: *tagged value context=(ctx1, ctx2, ...)*

2.14 attribute

IDL3

```
<attr_dcl> ::= "attribute" <param_type_spec> <attr_declarator> ";"
           |  "readonly attribute" <param_type_spec> <readonly_attr_declarator> ";"

<attr_declarator> ::= <simple_declarator> <attr_raises_expr>
                   |  <simple_declarator> {"," <simple_declarator>}*

<attr_raises_expr> ::= <get_excep_expr> [<set_excep_expr>]
                   |  <set_excep_expr>

<get_excep_expr>  ::= "getraises" <exception_list>

<set_excep_expr>  ::= "setraises" <exception_list>

<exception_list>  ::= "(" <scoped_name> {"," <scoped_name>}* ")"

<readonly_attr_declarator> ::= <simple_declarator> <raises_expr>
                             |  <simple_declarator> {"," <simple_declarator>}*
```

UML

UML-*attribute* oder UML-*association*

“readonly“ (*attribute*): *stereotype* readonly

“readonly“ (*association*): *stereotype* readonlyEnd beim Typ

“raises“: *tagged value* raises=(*exception1*, *exception2*, ...)

“getraises“: *tagged value* getRaises=(*exception1*, *exception2*, ...)

“setraises“: *tagged value* setRaises=(*exception1*, *exception2*, ...)

Beispiel

```
module docAttribute {
    struct MyStruct {
        string value;
    };

    interface MyInterface {
        attribute long number;
        readonly attribute float myReadonlyValue;
        attribute MyStruct value1;
        readonly attribute MyStruct value2;
    };
}
```

```

interface UglyInterface {
    readonly attribute long readonlyValue raises(MyException);
    attribute long value1 getraises(MyGetException) setraises(MySetException);
};
};

```

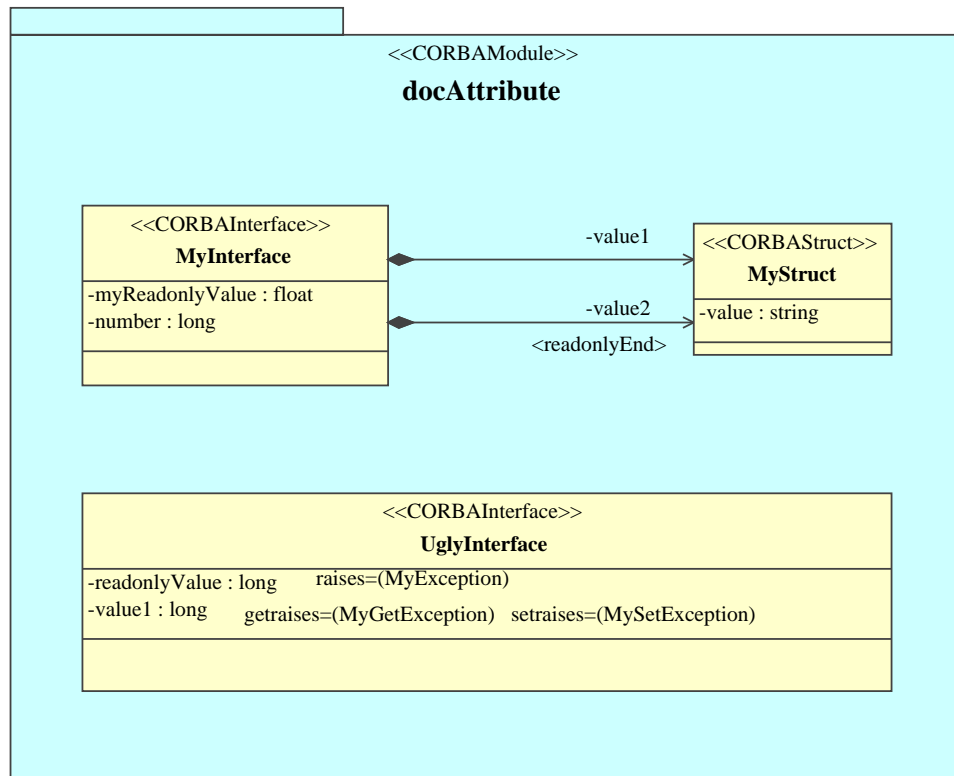


Abbildung 14: Attribute

2.15 component

IDL3

```
<component> ::= <component_header> "{" <component_export>* ";"  
              | "component" <identifier> ";"  
  
<component_header> ::= "component" <identifier> [":" <component_name>]  
                      ["supports" <interface_name> {"," <interface_name>}*]  
  
<component_export> ::= "provides" <interface_type> <identifier> ";"  
                      | "uses" ["multiple"] <interface_type> <identifier> ";"  
                      | "emits" <event_name> <identifier> ";"  
                      | "publishes" <event_name> <identifier> ";"  
                      | "consumes" <event_name> <identifier> ";"  
                      | <attr_dcl>  
  
<interface_type>  ::= <scoped_name> | "Object"
```

UML

class mit *stereotype* CCMComponent

Vererbung: normale Klassen-Vererbung

“supports”: Vererbung mit *stereotype* CCMSupports

“provides” (facet): *association* mit *stereotype* CCMProvides

“uses” (receptacle): *association* mit *stereotype* CCMUses

“multiple”: Vielfachheit 1..*

“emits”: *association* mit *stereotype* CCMEmits

“publishes”: *association* mit *stereotype* CCMPublishes

“consumes”: *association* mit *stereotype* CCMConsumes

Anstelle der *association* kann auch ein normales Attribut mit dem entsprechenden *stereotype* verwendet werden.

Beispiel

```
module docComponent {
  interface I1 {
  };

  interface I2 {
  };

  interface Facet1 {
  };

  component Comp1 supports I1, I2 {
    attribute long value1;
    uses Facet1 hook;
  };

  interface I3 {
  };

  interface Facet2 {
  };

  interface Facet3 {
  };

  eventtype MyEvent {
  };

  component Comp2 : Comp1 supports I3 {
    provides Facet1 facet1;
    uses Facet2 receptacle1;
    uses multiple Facet3 receptacle2;
    emits MyEvent event1;
    publishes MyEvent event2;
    consumes MyEvent event3;
  };
};
```

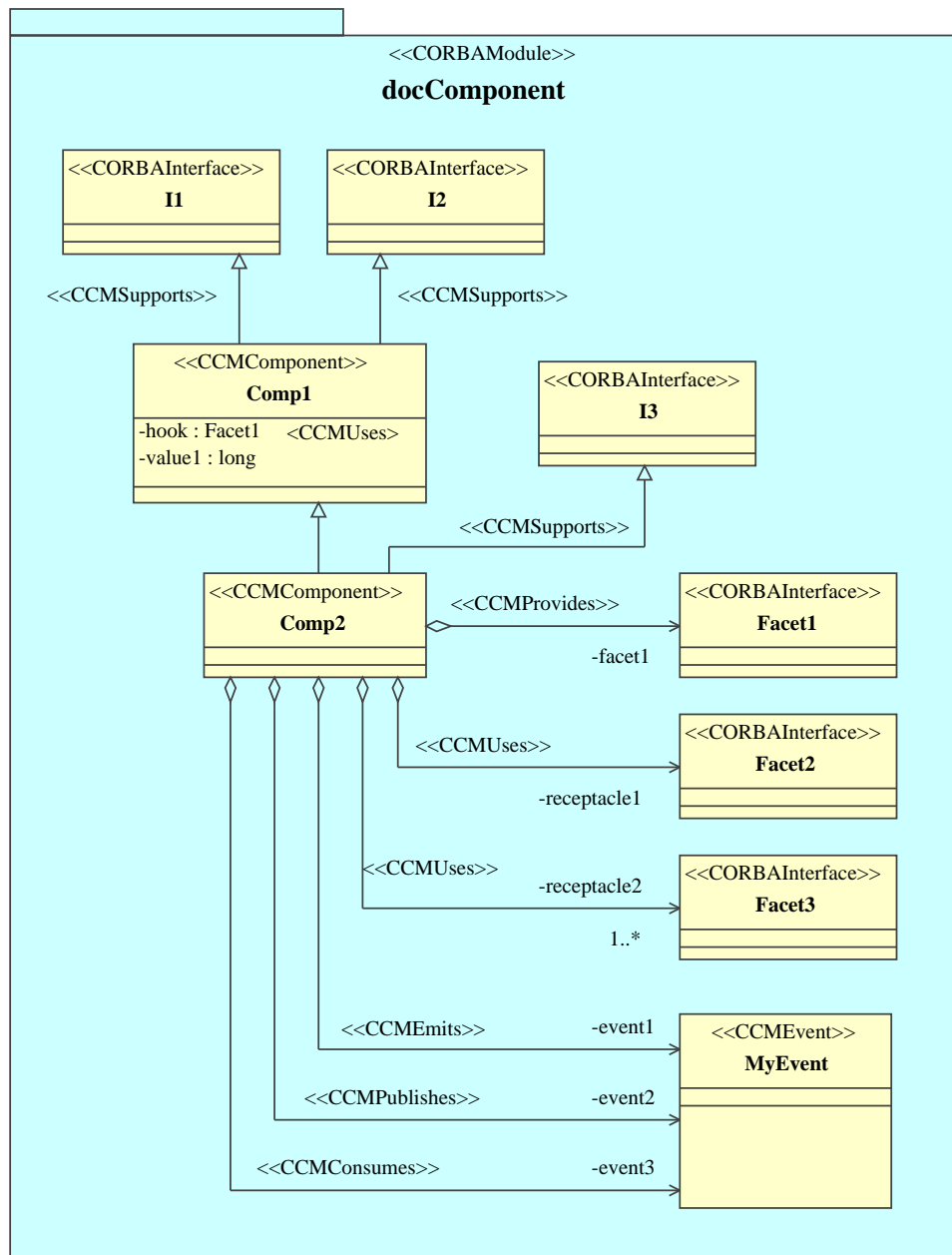


Abbildung 15: Komponenten

2.16 event

IDL3

```
<event> ::= (<event_dcl> | <event_abs_dcl>)  
          | ["abstract"] "eventtype" <identifier> ","
```

```
<event_abs_dcl> ::= "abstract eventtype" <identifier> [<value_inheritance_spec>]  
                  "{" <export>* "}";
```

```
<event_dcl>      ::= <event_header> "{" <value_element>* "}";
```

```
<event_header> ::= ["custom"] "eventtype" <identifier> [<value_inheritance_spec>]
```

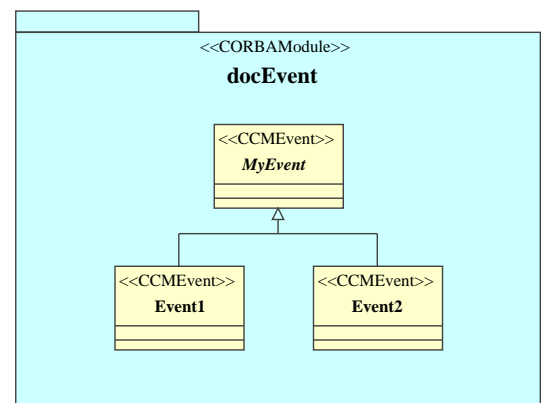
UML

class mit *stereotype* CCMEvent

siehe **valuetype** in Abschnitt 2.4 auf Seite 8

Beispiel

```
module docEvent {  
    abstract eventtype MyEvent {  
    };  
  
    eventtype Event1 : MyEvent {  
    };  
  
    eventtype Event2 : MyEvent {  
    };  
};
```



2.17 home

IDL3

```
<home_decl> ::= <home_header> "{" <home_export>* "};"

<home_header> ::= "home" <identifier> [":" <home_name>]
                  ["supports" <interface_name> {"," <interface_name>*}]
                  "manages" <component_name>
                  ["primarykey" <scoped_name>]

<home_export> ::= <export>
                  | <factory_dcl>
                  | <finder_dcl>

<factory_dcl> ::= "factory" <identifier> "(" [<init_param_decls>] ")"
                  [<raises_expr>] ";";

<finder_dcl>  ::= "finder" <identifier> "(" [<init_param_decls>] ")"
                  [<raises_expr>] ";";
```

UML

class mit *stereotype* CCMHome

Vererbung: normale Klassen-Vererbung

“supports“: Vererbung mit *stereotype* CCMSupports

“manages“: *association* mit *stereotype* CCManages

“factory“: *operation* mit *stereotype* CCMFactory

“finder“: *operation* mit *stereotype* CCMFinder

“primarykey“: bei der *association* (CCManages) mit der Komponente entweder eine *association class* mit *stereotype* CCMPrimaryKey oder ein *tagged value* mit dem Namen `primarykey`

Beispiel

```
module docHome {
    interface I1 {
    };

    component Comp1 {
        uses I1 hook;
    };
}
```

```

home Home1 manages Comp1 primaryKey Key {
    attribute long value;
    string print();
    finder findByName(in string name);
    factory create(in string name);
};

interface I2 {
};

home Home2 : Home1 supports I2 {
};

```

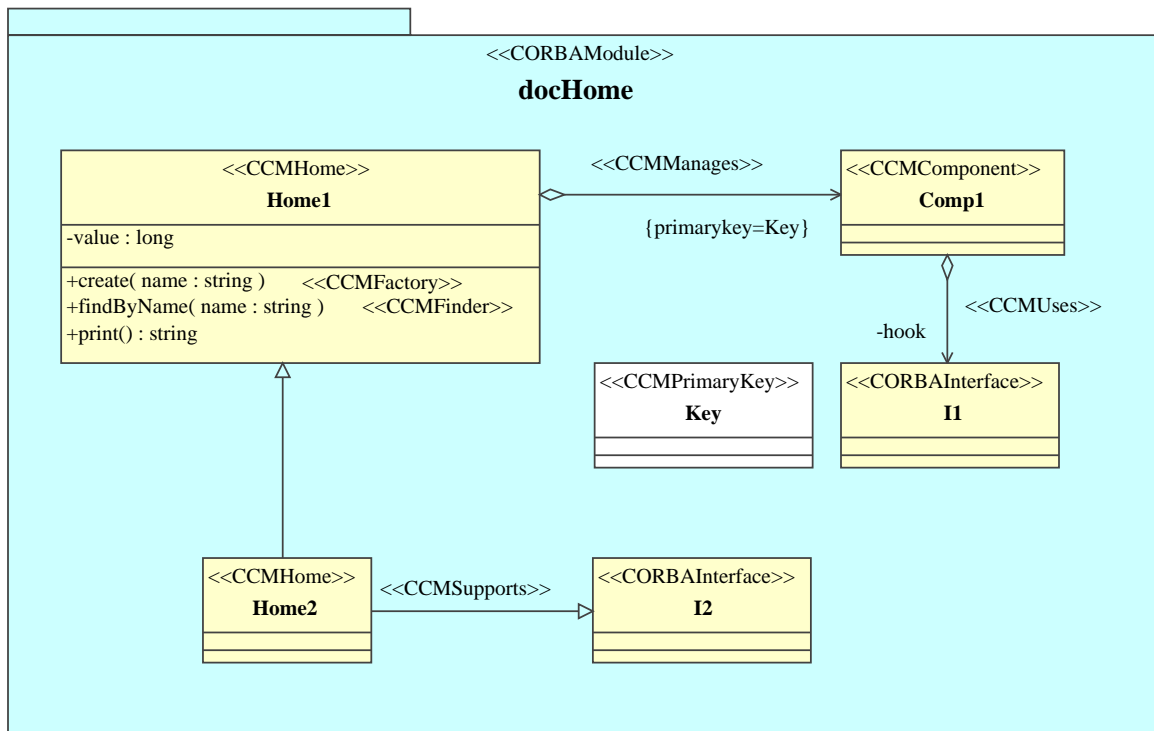


Abbildung 16: home

3 OCL

Die *constraints* haben normalerweise folgende Form:

```
package Parent::Child
  context MyClass
    inv: company='Salomon'
    inv Name: value>0

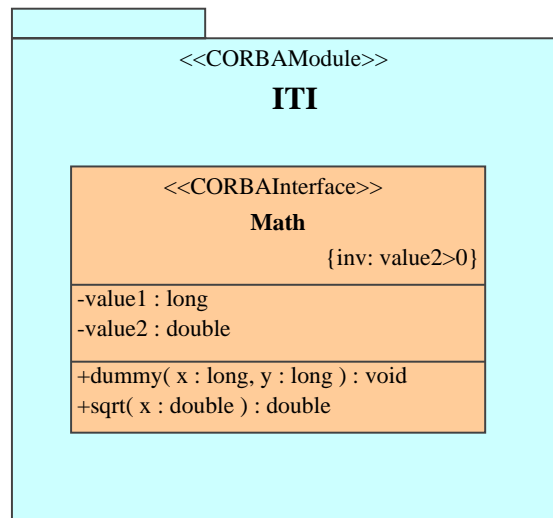
  context MyClass::myOperation(x:Real)
    pre: x>=0
    post: result<0
endpackage
```

Solche Ausdrücke können an beliebiger Stelle definiert werden; der Konverter schreibt sie ohne Änderung in die OCL-Datei. Um die Eingabe zu vereinfachen, sind folgende Abkürzungen möglich:

- Pre- und Postconditions können bei der Operation oder bei einem Parameter ohne **package** und **context** angegeben werden:
pre: x>=0
post check_result: result>0
- Invarianten können bei der Klasse oder bei einem Attribut ebenfalls ohne **package** und **context** angegeben werden:
inv: value>=0

Die fehlenden **package**- und **context**-Anweisungen werden automatisch erzeugt.

Beispiel:



IDL:

```
module ITI {
    interface Math {
        attribute long value1;
        attribute double value2;
        double sqrt(in double x);
        void dummy(in long x, in long y);
    };
};
```

OCL:

```
package ITI
    context Math::dummy(x:Integer, y:Integer)
        pre: x>0
        pre: y>0

    context Math
        inv: value1>0
        inv: value2>0

    context Math::sqrt(x:Real)
        pre: x>=0
        post: return>=0
        post: return*return=x
endpackage
```