ModSoft

Modellbasierte Software-Entwicklung mit UML 2
im WS 2014/15

Teil II b: Strukturmodellierung: Klassen,…

Prof. Dr. Joachim Fischer
Dr. Markus Scheidgen
Dipl.-Inf. Andreas Blunk

fischer@informatik.hu-berlin.de

Klassendiagramme - eine vertiefende Betrachtung

1. Attribute, Assoziationen und Links
2. Operationen und Methoden
   - Metaklasse Operation
   - Allgemeine Charakterisierung
   - Namensräume, Überladung von Operationen
   - Konstrukturen, Destruktoren
   - Allgemeine konkrete Syntax
   - Standard-Eigenschaften einer Operation:
     Leaf, Abstract, Query, Static, Guarded, Sequential, Concurrent
   - Parameter-Eigenschaften
   - Weitere verhaltenseinschränkende Operationseigenschaften
   - Redefinition als Operationseigenschaft

Zum Inhalt der letzten Vorlesung

- Attribute und Assoziationsenden
- Operationen einer Klasse
- Spezialisierung von Klassen bei Betrachtung von Attributen und Operationen

1. Problem
Attribute gleichen Namens in Basisklasse und Ableitung

2. Problem
Operationen gleichen Namens in einer Klasse (ein Namensraum) mit unterschiedlichen Signaturen (Overloading)

Namensraum einer Klasse

… gilt allgemein für alle Classifier

- Jede **Klasse** ist ein **Namensraum**/ stellt einen Namensraum dar

A **Namespace** is an Element in a model that contains a set of NamedElements that can be identified by name

- Bei einer Vererbungsbeziehung/Spezialisierung wird der **Namensraum** der abgeleiteten **Klasse** um alle nicht-privaten Elemente des Namensraumes der Basisklasse erweitert.

  Also: Keine separaten Namensräume(!).

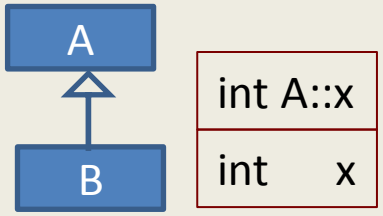
- Ein Namensraum setzt die Eindeutigkeit der Namen voraus.

A **NamedElement** may, in addition to having an explicit name, be associated with a **StringExpression**

A **StringExpression** is an Expression that specifies a String value that is derived by concatenating a sequence of operands with String values or a sequence of subExpressions, …

- Zur Sicherung der Eindeutigkeit können damit Umbenennungen vorgenommen werden.

  Der Einsatz eines Scope-Resolution-Operators (Wie in Simula) wäre eine konkrete Art der Umbenennung



Fortsetzung …

NamedElements may appear within a Namespace according to rules that specify how one NamedElement is **distinguishable** from another.

The default rule is that two members are distinguishable if
- they have different names or
- if they have the same names, but their metaclasses are different and neither is a (direct or indirect) subclass of the other.

This rule may be overridden for particular cases, such as **Operations** that are distinguished by their **signature** (see sub clause 9.6).

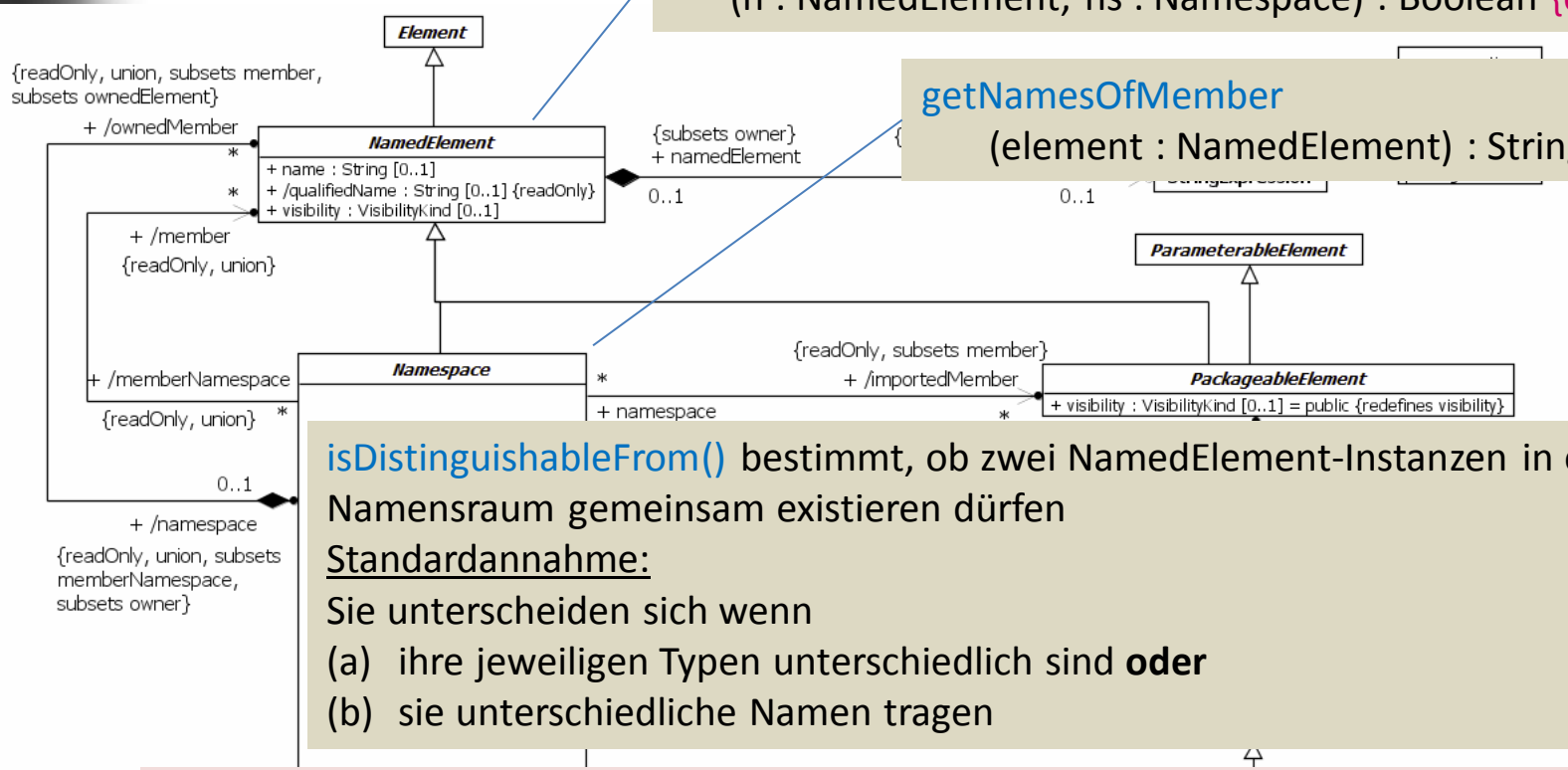
Expliziter Hinweis,
dass das Konzept der Überladung unterstützt wird.

bleibt zu klären:
1. Wie sind die Regeln formuliert?
2. Was gehört zu einer Operations-Signatur (Rückgabewert, Exceptions)?

ad 1) Eindeutigkeit von Namen in einem Namensraum

isDistinguishableFrom
(n : NamedElement, ns : Namespace) : Boolean {query}

getNamesOfMember
(element : NamedElement) : String [0..*]



isDistinguishableFrom() bestimmt, ob zwei NamedElement-Instanzen in einem Namensraum gemeinsam existieren dürfen

<u>Standardannahme:</u>

Sie unterscheiden sich wenn

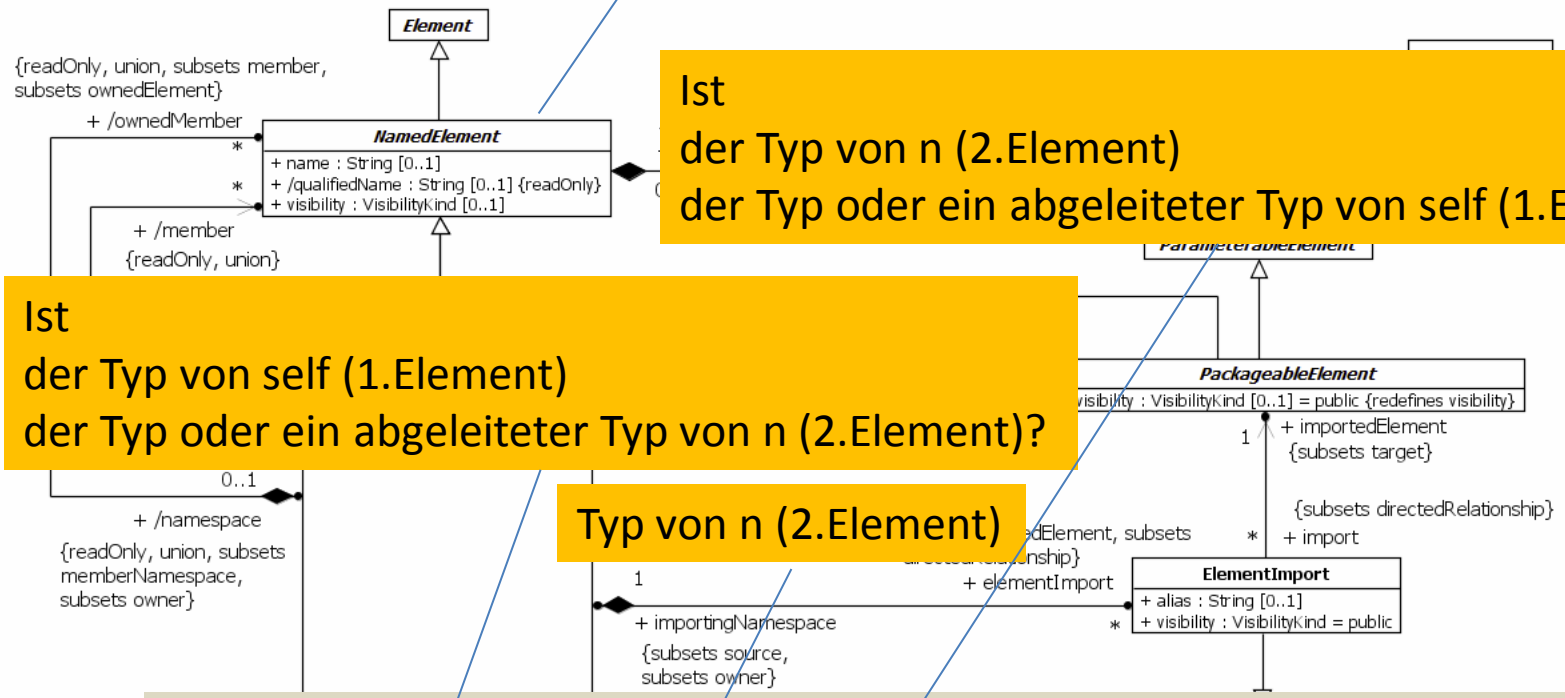
(a) ihre jeweiligen Typen unterschiedlich sind **oder**

(b) sie unterschiedliche Namen tragen

```
body: (self.oclIsKindOf(n.oclType()) or
            n.oclIsKindOf(self.oclType())
       )
       implies
       ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->isEmpty()
```

...Fortsetzung

isDistinguishableFrom (...)



Ist
der Typ von n (2.Element)
der Typ oder ein abgeleiteter Typ von self (1.Element)?

Ist
der Typ von self (1.Element)
der Typ oder ein abgeleiteter Typ von n (2.Element)?

Typ von n (2.Element)

isDistinguishableFrom (n : NamedElement, ns : Namespace) : Boolean

body: (self.oclIsKindOf(n.oclType()) **or**
n.oclIsKindOf(self.oclType())
)
implies
ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->isEmpty()

gilt eine von beiden Annahen dann muss auch folg. Bedingung gelten:

OCL-Standard-Funktionen:

- self — Bezug zur Instanz des Kontext-Classifierers (hier: BehavioralFeature)
- object.oclType(): OCLType — Typ der Classifiers-Instanz object
- object.oclIskindOf (type:OCLType): Boolean — **true**, falls Typ von objekt gleich type oder davon abgeleitet ist
- set->intersection(set2: Set(T)): Set(T) — **Durchschnitt** von set und set2 als Instanzen von Set ergibt eine Menge vom Typ Set(T)
- set->isEmpty: Boolean — **true**, falls set leer ist

isDistinguishableFrom (n : NamedElement, ns : Namespace) : Boolean

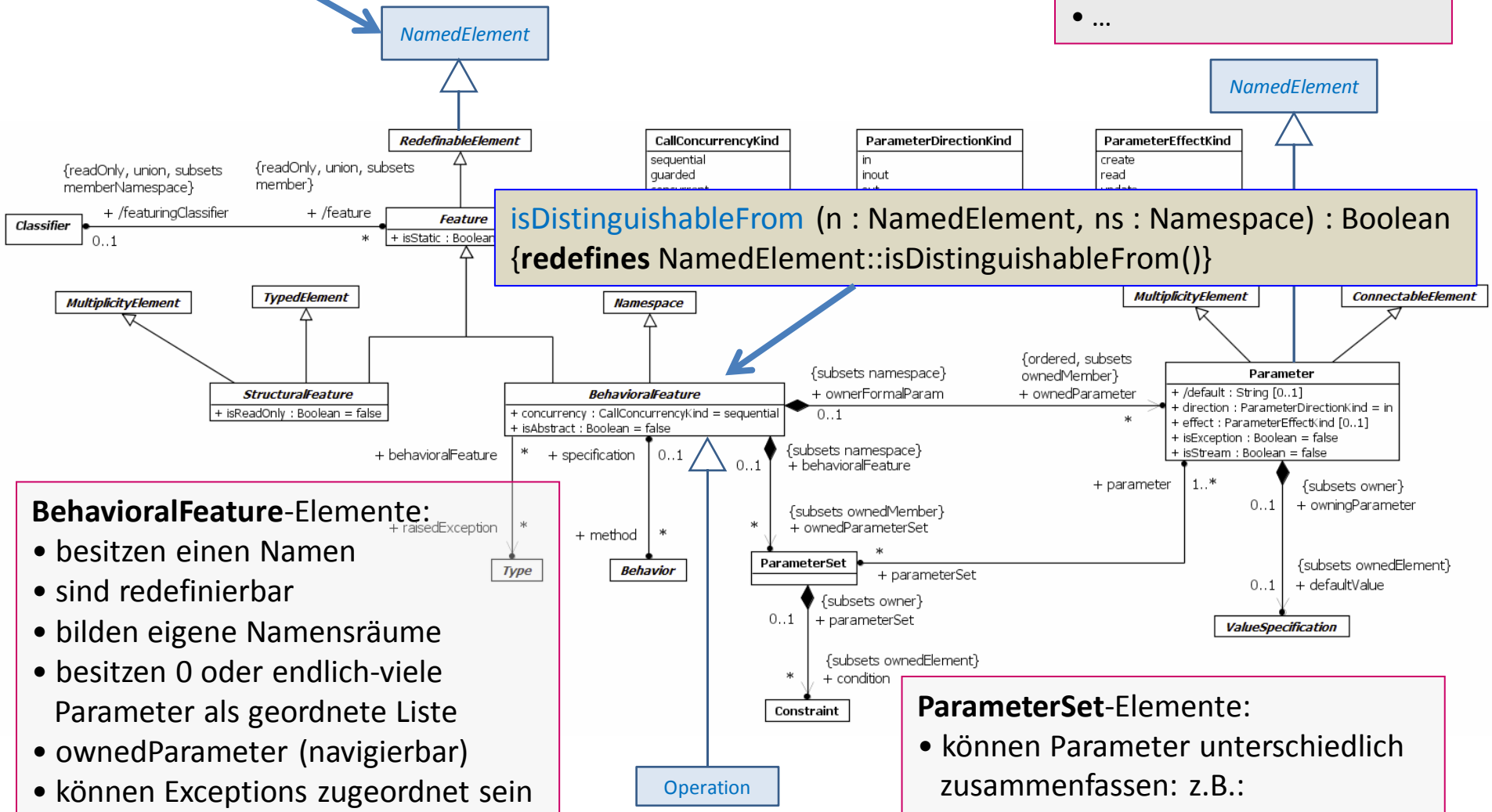
```
body: (self.oclIsKindOf(n.oclType()) or
            n.oclIsKindOf(self.oclType())
      )
      implies
      ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->isEmpty()
```

gilt eine von beiden Annahmen bzgl. Typen beiden NamedElement-Instanzen,
dann muss die Durchnittsfremdheitsforderung gelten,
was bedeutet, dass sich die Namen unterscheiden müssen

ad 2) Operator-Overloading

Parameter
- haben einen Namen und Typ
- eine Richtung der Übergabe (in als Default)
- eine Multiplizität
- einen optionalen Wert
- …

isDistinguishableFrom(n : NamedElement, ns : Namespace) : Boolean)

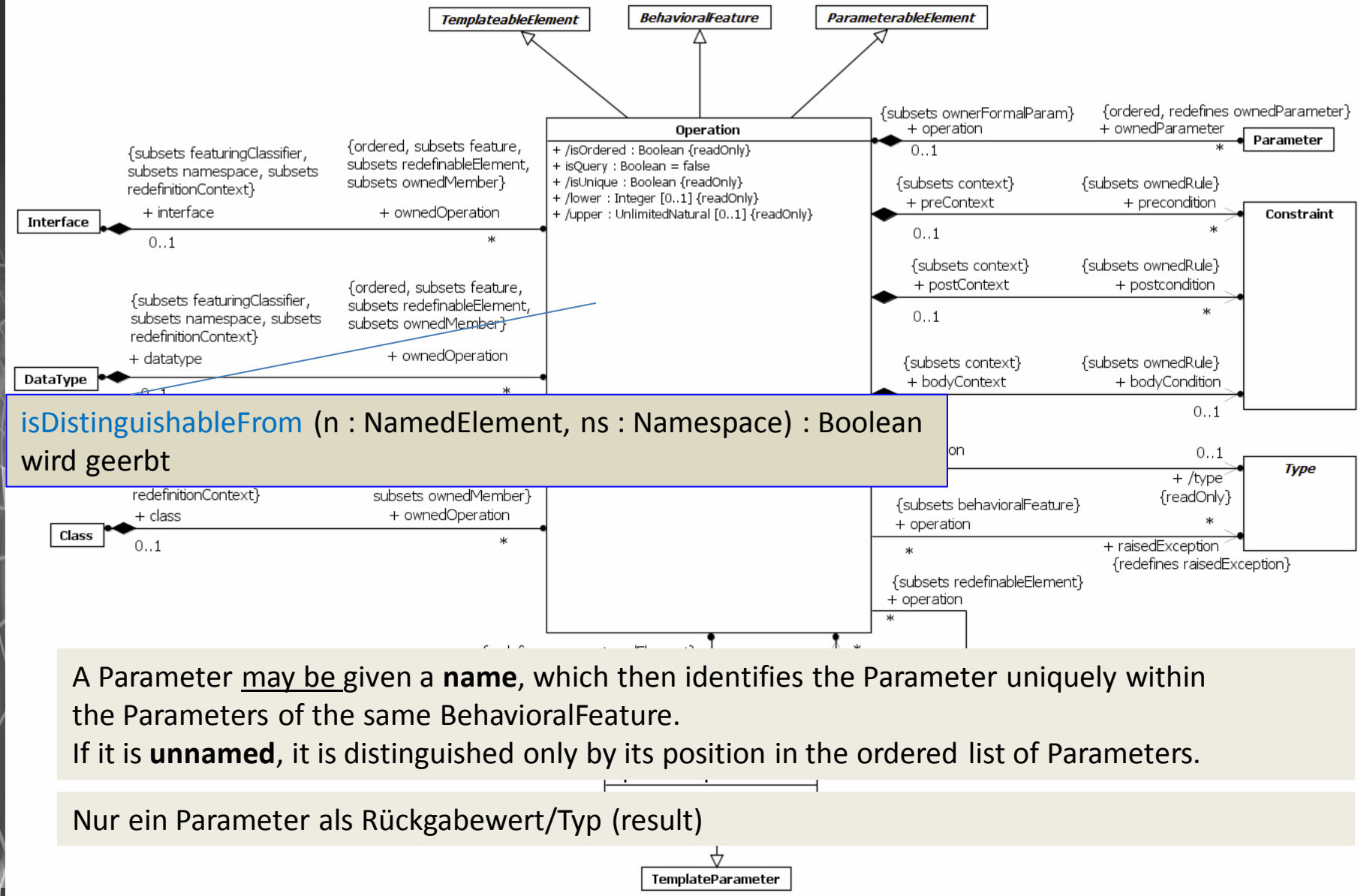


BehavioralFeature-Elemente:
- besitzen einen Namen
- sind redefinierbar
- bilden eigene Namensräume
- besitzen 0 oder endlich-viele Parameter als geordnete Liste
- ownedParameter (navigierbar)
- können Exceptions zugeordnet sein

ParameterSet-Elemente:
- können Parameter unterschiedlich zusammenfassen: z.B.:

Input-Parameter

…Fortsetzung



isDistinguishableFrom (n : NamedElement, ns : Namespace) : Boolean wird geerbt

A Parameter may be given a **name**, which then identifies the Parameter uniquely within the Parameters of the same BehavioralFeature.
If it is **unnamed**, it is distinguished only by its position in the ordered list of Parameters.

Nur ein Parameter als Rückgabewert/Typ (result)

BehavioralFeature::isDistinguishableFrom

object.isDistinguishableFrom(n : NamedElement, ns : Namespace) : **Boolean**
{**redefines** NamedElement::isDistinguishableFrom()}

```
body: (n.oclIsKindOf(BehavioralFeature) and
        ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n)
    )->notEmpty()) 1)
    implies
        Set{self}->including(n.oclAsType(BehavioralFeature))
                ->isUnique 2)
                (ownedParameter->collect(p|Tuple {
                    3)                          4)
                                                name=p.name,
                                                type=p.type,
                                                effect=p.effect,
                                                direction=p.direction,
                                                isException=p.isException,
                                                isStream=p.isStream,
                                                isOrdered=p.isOrdered,
                                                isUnique=p.isUnique,
                                                lower=p.lower,
                                                upper=p.upper
                                                }
```

Menge aller eingeführten Namen des betreffenden BehavioralFeature-Elementes (inkl. Alias-Bezeichner)

1. ***Bei Annahme der Aussagen**:*
   Der Typ von n ist eine Ableitung von BehavioralFeature und zum Namensraum ns gehört ein und derselbe Name zweier BehavioralFeature-Instanzen
   muss gelten:
2. *dass die Menge (der zusammengefassten Signaturen) beider Elemente duplikationsfrei sein muss,*
3. *wobei diese Menge so konstruiert wird, indem nur die Parametersätze der beiden BehavioralFeature-Instanzen zur neuen Elementebildung herangezogen werden und*
4. *dabei so 10-Tupel als Elemente entstehen*

BehavioralFeature::isDistinguishableFrom

object.isDistinguishableFrom(n : NamedElement, ns : Namespace) : **Boolean**
{**redefines** NamedElement::isDistinguishableFrom()}

```
body: (n.oclIsKindOf(BehavioralFeature) and
        ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n)
    )->notEmpty())
    implies
        Set{self}->including(n.oclAsType(BehavioralFeature))
                ->isUnique
                  (ownedParameter->collect(p|Tuple {
                                name=p.name,
                                type=p.type,
                                effect=p.effect,
                                direction=p.direction,
                                isException=p.isException,
                                isStream=p.isStream,
                                isOrdered=p.isOrdered,
                                isUnique=p.isUnique,
                                lower=p.lower,
                                upper=p.upper
                                }
```

Zwei Signaturen müssen sich mindestens in einer der folg. Angaben

1. *Namen*
2. *Typ*
3. *Effect*
4. *Ein-Ausgabe-Richtung*
5. *ExceptionMarkierung eines AusgabeParameters*
6. *StreamEigenschaft*
7. *Odnungseigenschaft des Rückgabewertes*
8. *Duplikationsausschluss des Rückgabewertes*
9. *untere und*
10. *obere Multiplizitätsgrenze des Rückgabewertes*

unterscheiden

Unterschied zur früheren UML-Versionen

gemeinsam

Im Metamodell von UML wird die Unterscheidbarkeit von **BehavioralFeature**-Signaturen generell geregelt (und damit auch für Operationen)

früher galt:

Operationen müssen sich unterscheiden

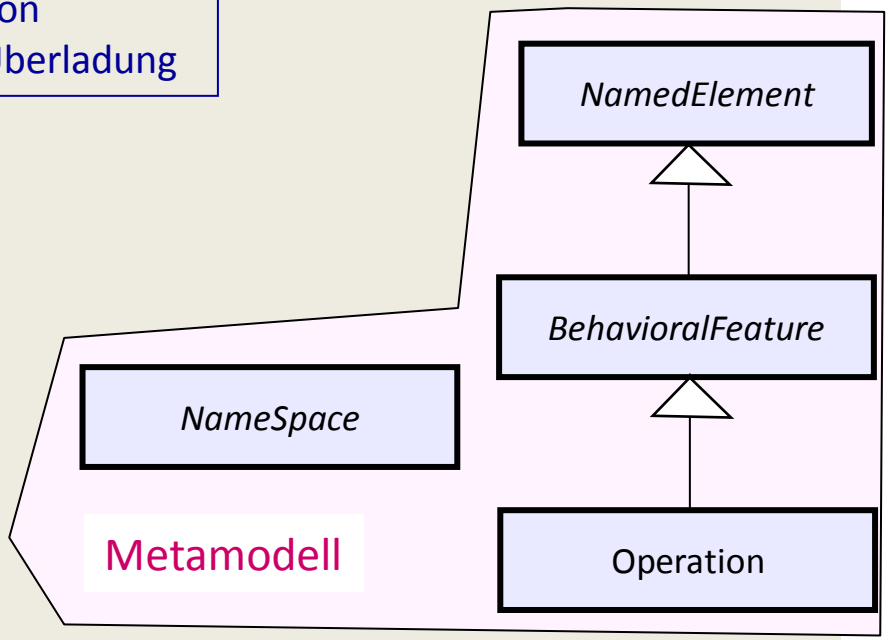
- im Namen oder
- in einem der Parametertypen oder
- im Rückgabetyp

} Grundlage von Überladung

sie sind nicht unterscheidbar, wenn nur

- Parameternamen,
- die Art von Parametern,
- Programmausnahmen (Exception) oder
- Bedingungen (Pre- und Postcondition)

unterschiedlich sind



Klassendiagramme - eine vertiefende Betrachtung

1. Attribute, Assoziationen und Links
2. Operationen und Methoden
   - Metaklasse Operation
   - Allgemeine Charakterisierung
   - Namensräume, Überladung von Operationen
   - Konstrukturen, Destruktoren
   - Allgemeine konkrete Syntax
   - Standard-Eigenschaften einer Operation:
     Leaf, Abstract, Query, Static, Guarded, Sequential, Concurrent
   - Parameter-Eigenschaften
   - Weitere verhaltenseinschränkende Operationseigenschaften
   - Redefinition als Operationseigenschaft

Operationen (allgemein)

Grundsyntax
[«stereotype»]
[visibility]
name ([paramList]) [: returnType]
[{ propertyStringList }]

Normal Operations:

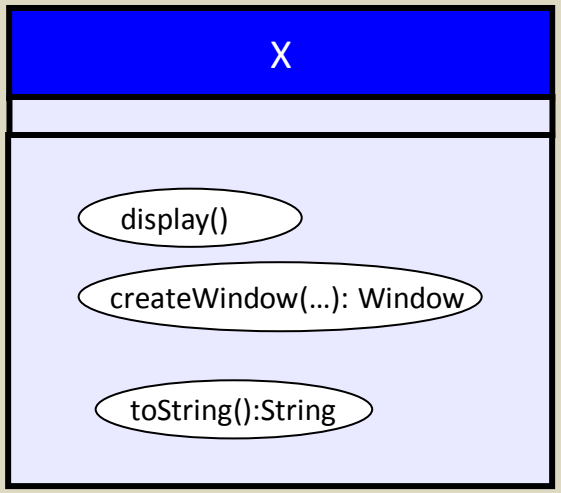
- display ()
- -hide ()
- +createWindow (location: Coordinates, container: Container [0..1]): Window
- +toString (): String

A template Operation:

- f <T:Class>(x : T)

A binding of that template Operation:

- f << T -> Window >>(x : Window)



syntaktische Variation

Konstrukturen /Destruktoren

UML:

- A constructor is an Operation having a **single return result** parameter of the type of the owning Class, and marked with the standard stereotype
  «Create».
  Bem.: Der Name ist i.allg. frei wählbar.

- Destructor is defined with the
  «Destroy» stereotype
  to annotate a destructor operation

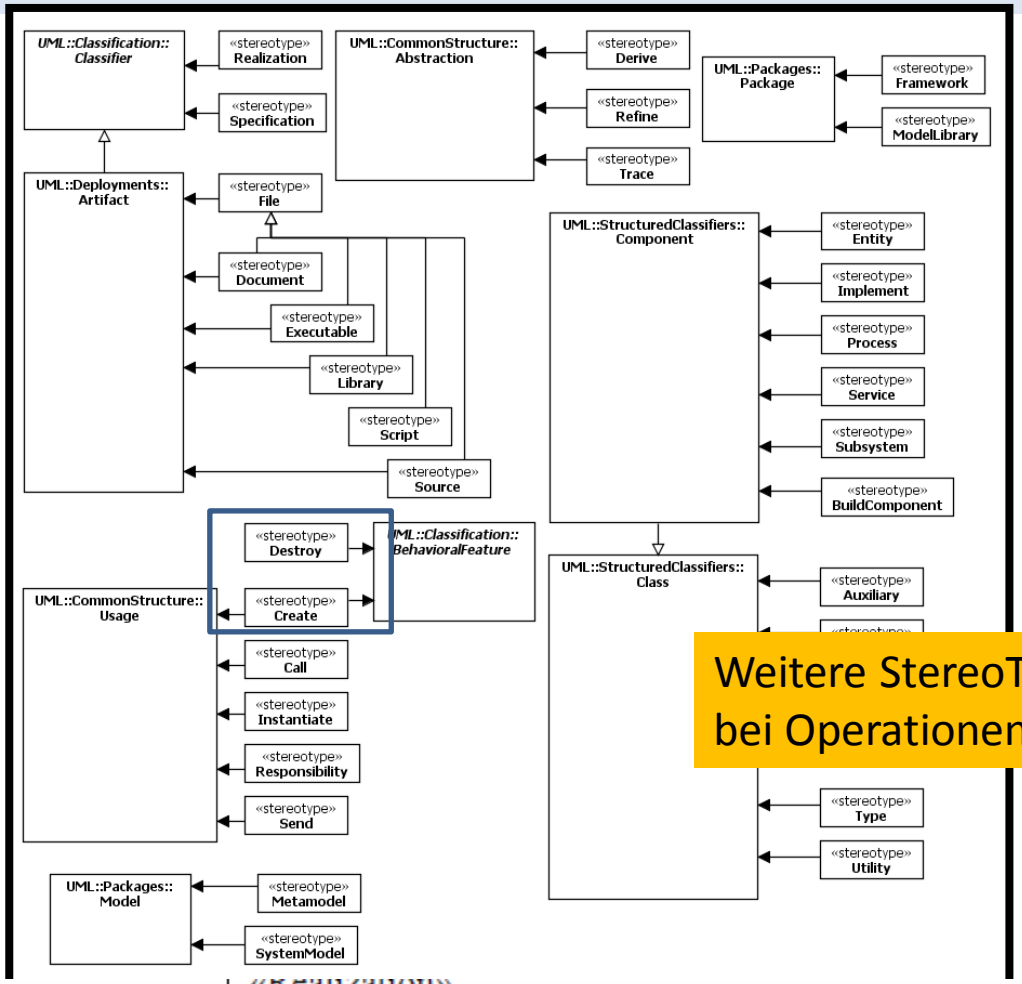
Alf:

- Every class represented is always mapped as having at least one constructor.
  Bem.: Es kann mehrere geben: (Overloading)
- If no constructor is explicitly defined for the class, then the class is assumed to have a *default* constructor. The name of the default constructor is the same as the name of the class.

- Destructor maps to an operation with the stereotype «Destroy» .

FRAGE: Gibt es weitere mit Stereo-Typen markierte Anweisungen?
→ Suche einem Package **Predefined**

Package Predefined



Weitere StereoTypes kommen bei Operationen **nicht** zum Einsatz

		«Realization».
«SystemModel»	Model	A SystemModel is a stereotyped model that contains a collection of models of the same system. A SystemModel also contains all relationships and constraints between model elements contained in different models.

A-ha-Effekt: Es gibt doch ein Modellelement für System in UML

Klassendiagramme - eine vertiefende Betrachtung

1. Attribute, Assoziationen und Links
2. Operationen und Methoden
   - Metaklasse Operation
   - Allgemeine Charakterisierung
   - Namensräume, Überladung von Operationen
   - Konstrukturen, Destruktoren
   - Allgemeine konkrete Syntax
   - Standard-Eigenschaften einer Operation:
     Leaf, Abstract, Query, Static, Guarded, Sequential, Concurrent
   - Parameter-Eigenschaften
   - Weitere verhaltenseinschränkende Operationseigenschaften
   - Redefinition als Operationseigenschaft

Operations-Syntax

[<visibility>] <name> '(' [<parameter-list>] ')' [':'
[<return-type>] ['[' <multiplicity-range> ']'] ['{' <oper-property> [',' <oper-property>] '}']]*

where:

- <*visibility*> is the visibility of the Operation (see 7.4). <*visibility*> ::= '+' | '-' | '#' | '~'
- <*name*> is the name of the Operation.
- <*parameter-list*> is a list of Parameters of the Operation in the following format:
  <*parameter-list*> ::= <*parameter*> [','<*parameter*>]*

where <*parameter*> is defined in 9.4.4.

- <*return-type*> is the type of the return result Parameter if the Operation has one defined.
- <*multiplicity-range*> is the multiplicity of the return type (see 7.5).
- <*oper-property*> indicates the properties of the Operation.
  <*oper-property*> ::= 'redefines' <*oper-name*> | 'query' | 'ordered' | 'unordered' | 'unique' | 'nonunique' | 'seq' | 'sequence' | <*oper-constraint*>

where:

- 'redefines' <*oper-name*> means that the Operation redefines an inherited Operation identified by <*oper-name*>, where <*oper-name*> may be qualified.
- 'query' means that the Operation does not change the state of the system.
- 'ordered' applies when there is a multi-valued return Parameter and means that its values are ordered.
- 'unordered' applies when there is a multi-valued return Parameter and means that its values are not ordered.
- 'unique' applies when there is a multi-valued return Parameter and means that its values have no duplicates.
- 'nonunique' applies when there is a multi-valued return Parameter and means that its values may have duplicates.
- 'seq' or 'sequence' applies when there is a multi-valued return Parameter and means that its values constitute an ordered bag, i.e., isUnique = false and isOrdered = true.
- <*oper-constraint*> is a constraint that applies to the Operation. The parameter list may be suppressed.

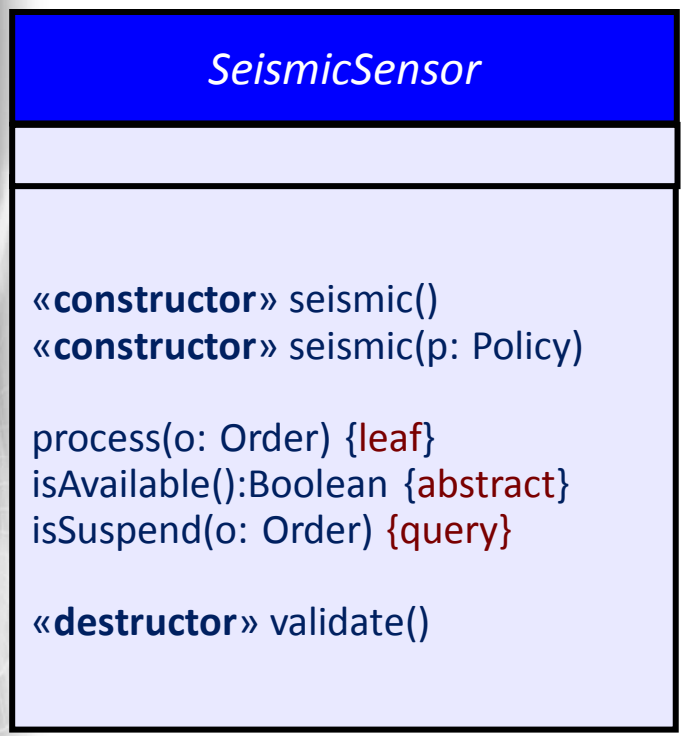
Klassendiagramme - eine vertiefende Betrachtung

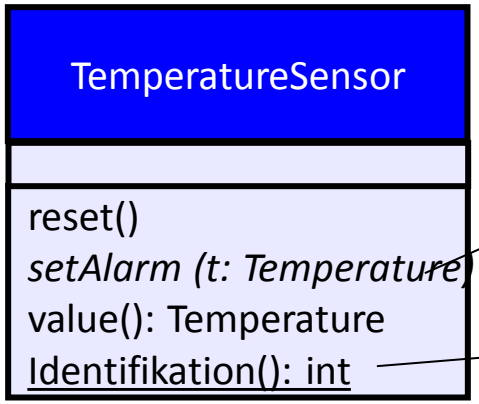
1. Attribute, Assoziationen und Links
2. Operationen und Methoden
   - Metaklasse Operation
   - Allgemeine Charakterisierung
   - Namensräume, Überladung von Operationen
   - Konstrukturen, Destruktoren
   - Allgemeine konkrete Syntax
   - Standard-Eigenschaften einer Operation:
     Leaf, Abstract, Query, Static, Guarded, Sequential, Concurrent
   - Parameter-Eigenschaften
   - Weitere verhaltenseinschränkende Operationseigenschaften
   - Redefinition als Operationseigenschaft

Operation einer Klasse

Grundsyntax
[«stereotype»]
[visibility]
name ([paramList]) [: returnType]
[{ propertyStringList }]

- Operationen sind Abstraktionen von Aktionen,
  die über Objekte der Klasse (Classifiers) ausgeführt werden kö
- werden per Namen und Signatur identifiziert



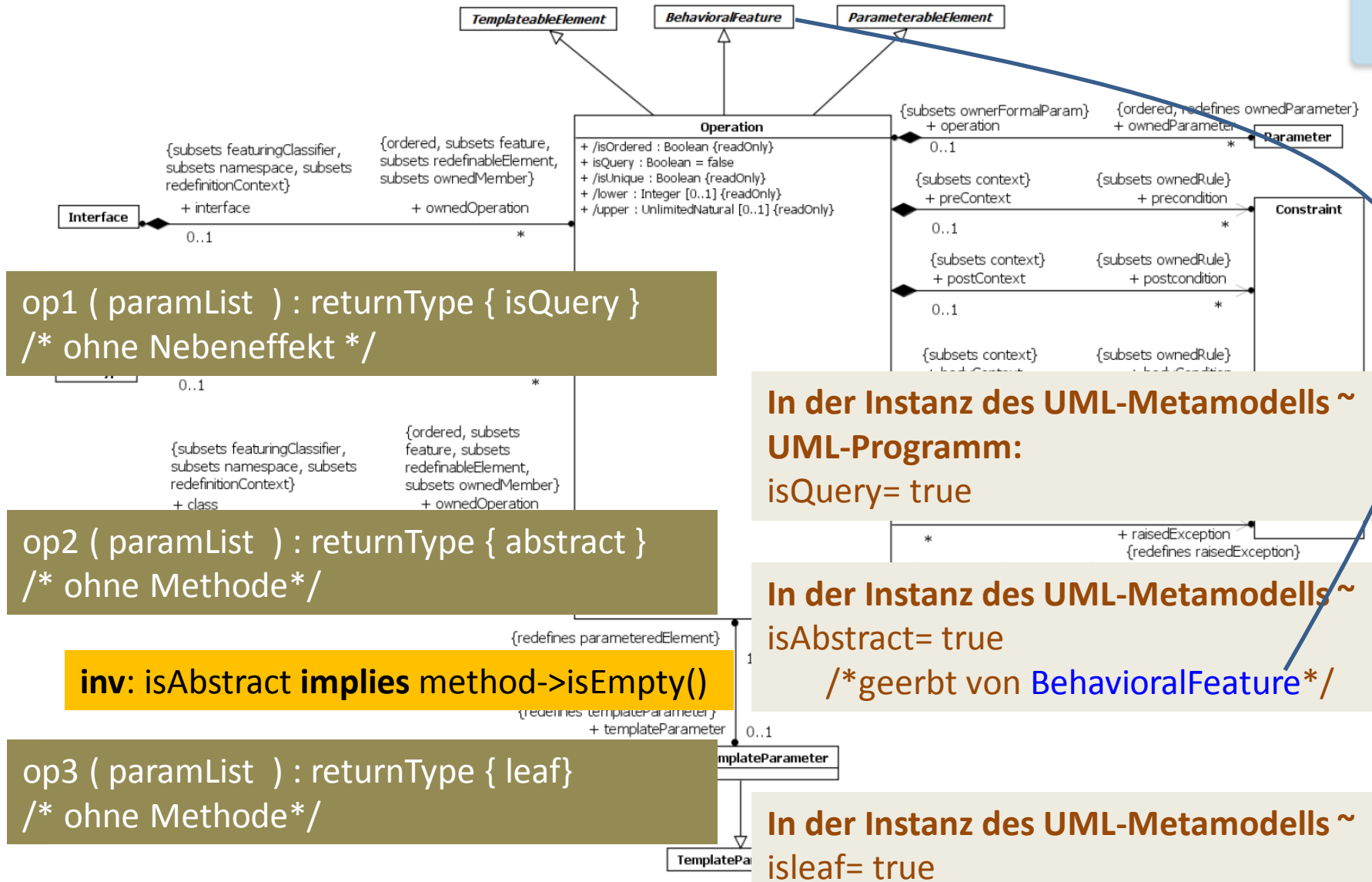


abstrakte Operation (ohne Methode)

Klassenoperation (Operation ohne Objektbezug)

Als Eigenschaft sind aber auch Verhaltens-Constraints definierbar

- Operationen legen nicht die Definition der Verhaltensaktionen fest,
  nur deren **Signatur** und weitere **Eigenschaften** der Operation
- Implementationen der Operationen werden durch **Methoden**spezifikationen geliefert:
  Aktionsdiagramme, Zustandsdiagramme, …



op1 (paramList) : returnType { isQuery }
/* ohne Nebeneffekt */

In der Instanz des UML-Metamodells ~ UML-Programm:
isQuery= true

op2 (paramList) : returnType { abstract }
/* ohne Methode*/

In der Instanz des UML-Metamodells ~
isAbstract= true
/*geerbt von BehavioralFeature*/

inv: isAbstract **implies** method->isEmpty()

op3 (paramList) : returnType { leaf}
/* ohne Methode*/

In der Instanz des UML-Metamodells ~
isleaf= true
/*geerbt über mehrere Stufen von RedefinableElement*/

Figure 9.13 Operations

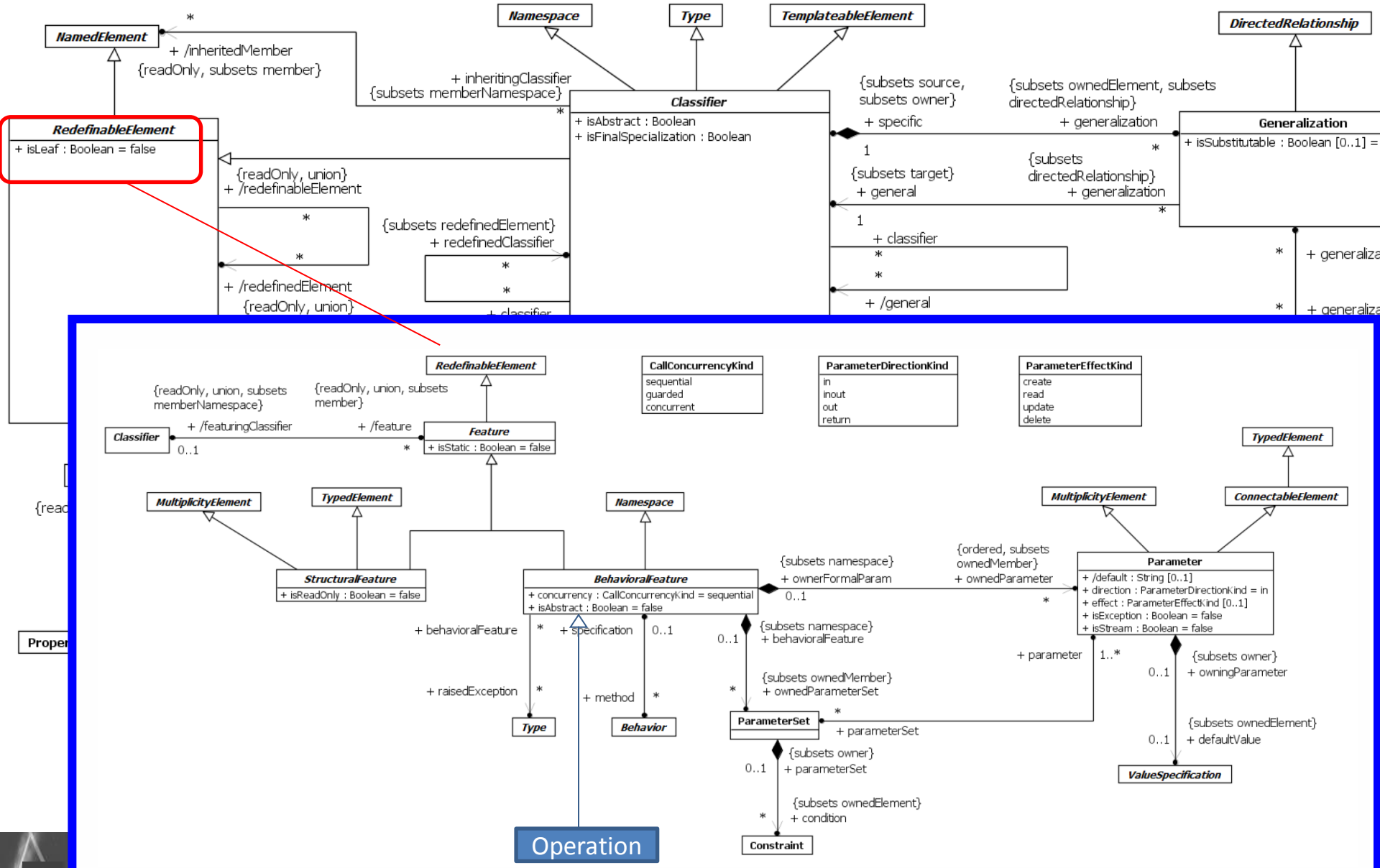


Figure 9.9 Features

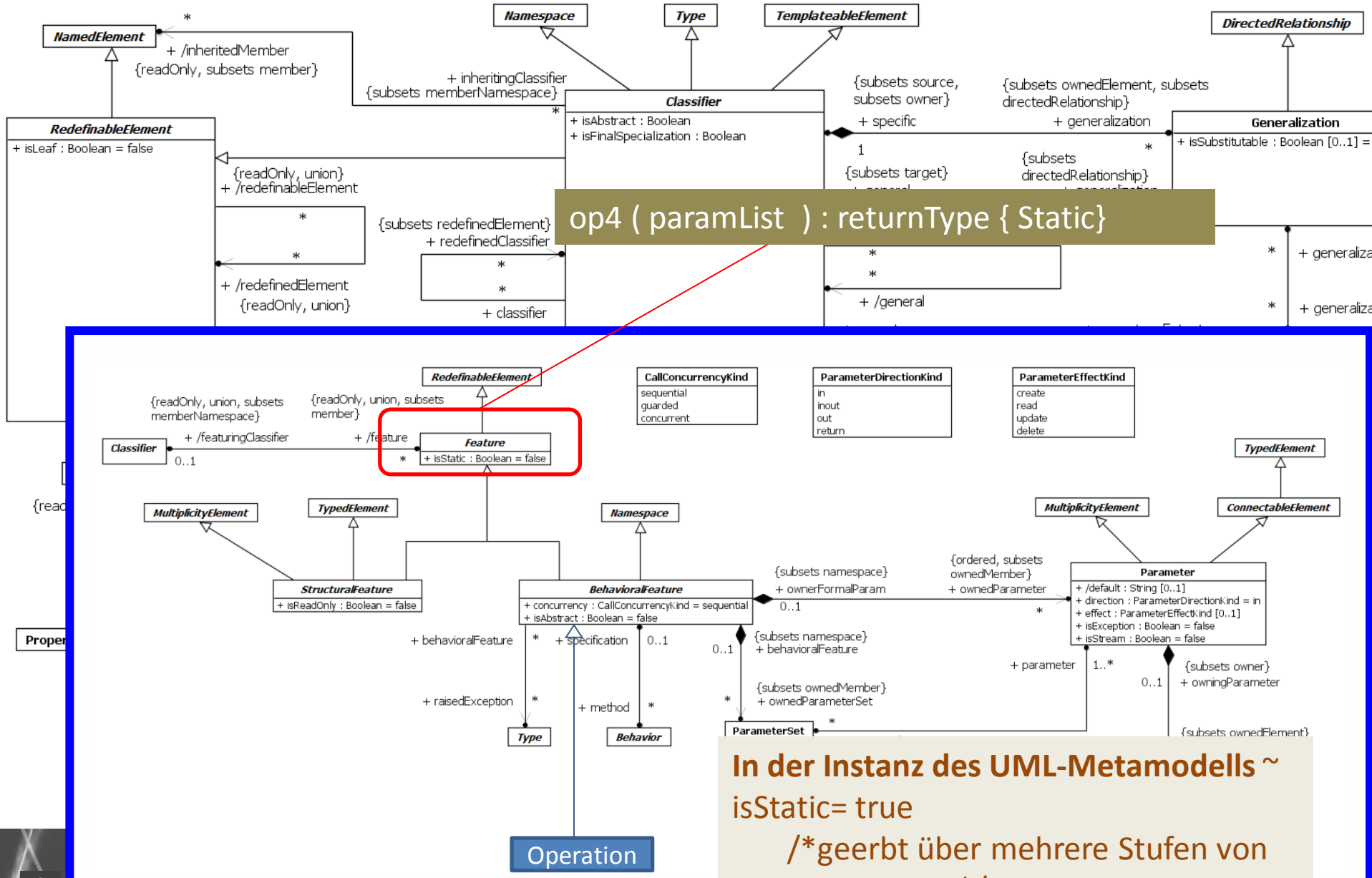


Figure 9.9 Features

Weitere Eigenschaften von Operationen

op5 (paramList) : returnType {guarded}

Default-Einstellung: *sequential*

- *sequential* (standardfall)
  Keine Unterstützung für parallele Aufrufe
  solche müssen außerhalb des Objektes koordiniert werden

Achtung: Angabe nur möglich, wenn die besitzende Klasse eine **aktive** Klasse ist

- *guarded*
  Zwangssequentialisierung bei zeitweiliger Blockierung von Kontrollflüssen

- *concurrent*
  gleichzeitige mehrfache Ausführung von Kontrollflüssen wird unterstützt

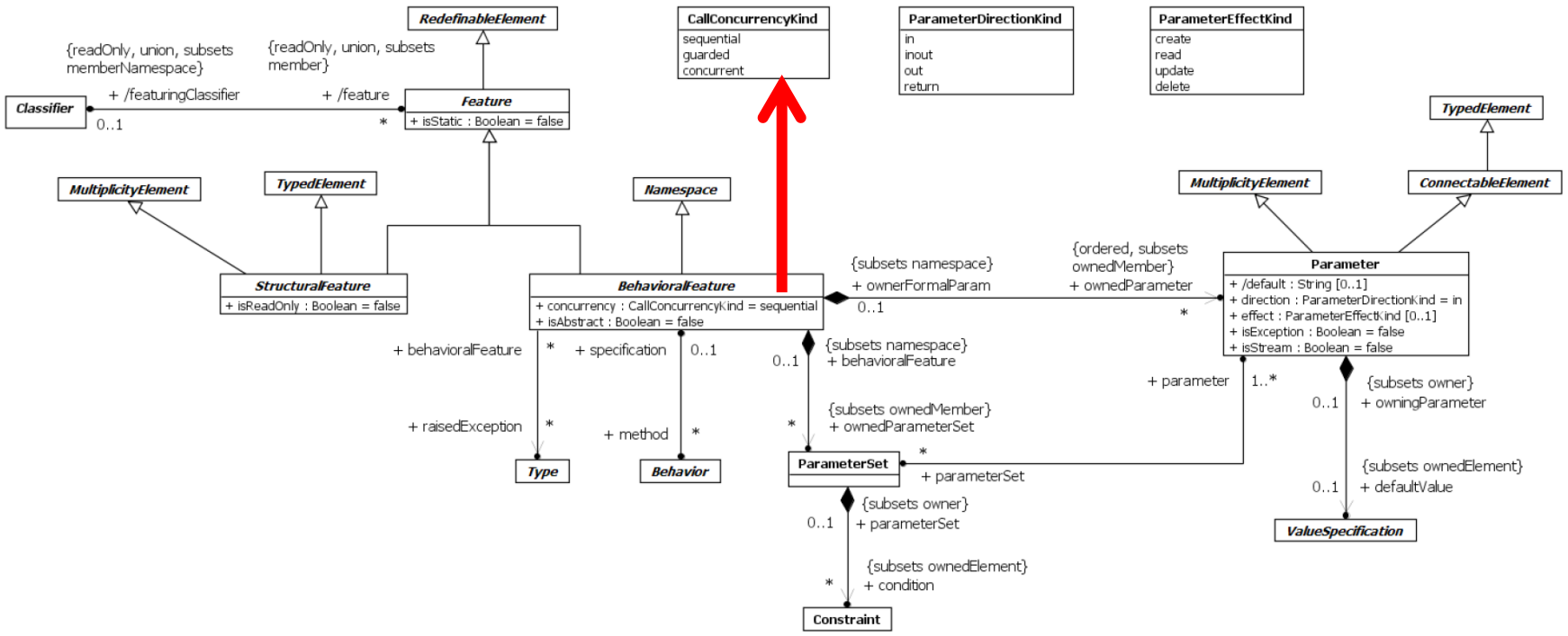


Figure 9.9 Features

Klassendiagramme - eine vertiefende Betrachtung

1. Attribute, Assoziationen und Links
2. Operationen und Methoden
   - Metaklasse Operation
   - Allgemeine Charakterisierung
   - Namensräume, Überladung von Operationen
   - Konstrukturen, Destruktoren
   - Allgemeine konkrete Syntax
   - Standard-Eigenschaften einer Operation:
     Leaf, Abstract, Query, Static, Guarded, Sequential, Concurrent
   - Parameter-Eigenschaften
   - Weitere verhaltenseinschränkende Operationseigenschaften
   - Redefinition als Operationseigenschaft

Parameter-Syntax

<*parameter*> ::= [<*direction*>] <*parameter-name*> ':'
<*type-expression*> ['['<*multiplicity-range*>']'] ['=' <*default*>]
['{' <*parm-property*> [',' <*parm-property*>]* '}']

where:

- <*direction*> ::= 'in' | 'out' | 'inout' (defaults to 'in' if omitted).
- <*parameter-name*> is the name of the Parameter.
- <*type-expression*> is an expression that specifies the type of the Parameter.
- <*multiplicity-range*> is the multiplicity of the Parameter.
  (See MultiplicityElement — sub clause 7.5).
- <*default*> is an expression that defines the value specification for the default value of the Parameter.
- <*parm-property*> indicates additional property values that apply to the Parameter.

<*parm-property*> ::= 'ordered' | 'unordered' | 'unique' | 'nonunique' | 'seq' | 'sequence'

where

- 'ordered' applies when there is a multi-valued Parameter and means that its values are ordered.
- 'unordered' applies when there is a multi-valued Parameter and means that its values are not ordered.
- 'unique' applies when there is a multi-valued Parameter and means that its values have no duplicates.
- 'nonunique' applies when there is a multi-valued Parameter and means that its values may have duplicates.
- 'seq' or 'sequence' applies when there is a multi-valued Parameter and means that its values constitute an ordered bag, i.e., isUnique = false and isOrdered = true.

Keine Henkel für **Exceptions** und **Effect** vorhanden

Effect-Property

The **effect property** may be used
to specify what happens to **objects** passed
in or
out
of a Parameter.

create	Objects passed out of executions of the behavior as values of the parameter do not exist before those executions start.
read	Objects that are values of the parameter have values of their properties, or links in which they participate, or their classifiers retrieved during executions of the behavior.
update	Objects that are values of the parameter have values of their properties, or links in which they participate, or their classification changed during executions of the behavior.
delete	Objects that are values of the parameter do not exist after executions of the behavior are finished.

Ausnahmen

The **isException** property applies to output Parameters.
An output posted to a Parameter with isException true during an invocation of a BehavioralFeature excludes outputs from being posted to any other outputs of the BehavioralFeature during the same invocation.
The type of such an exception Parameter may be included in the **raisedException set**, but does not have to be included.

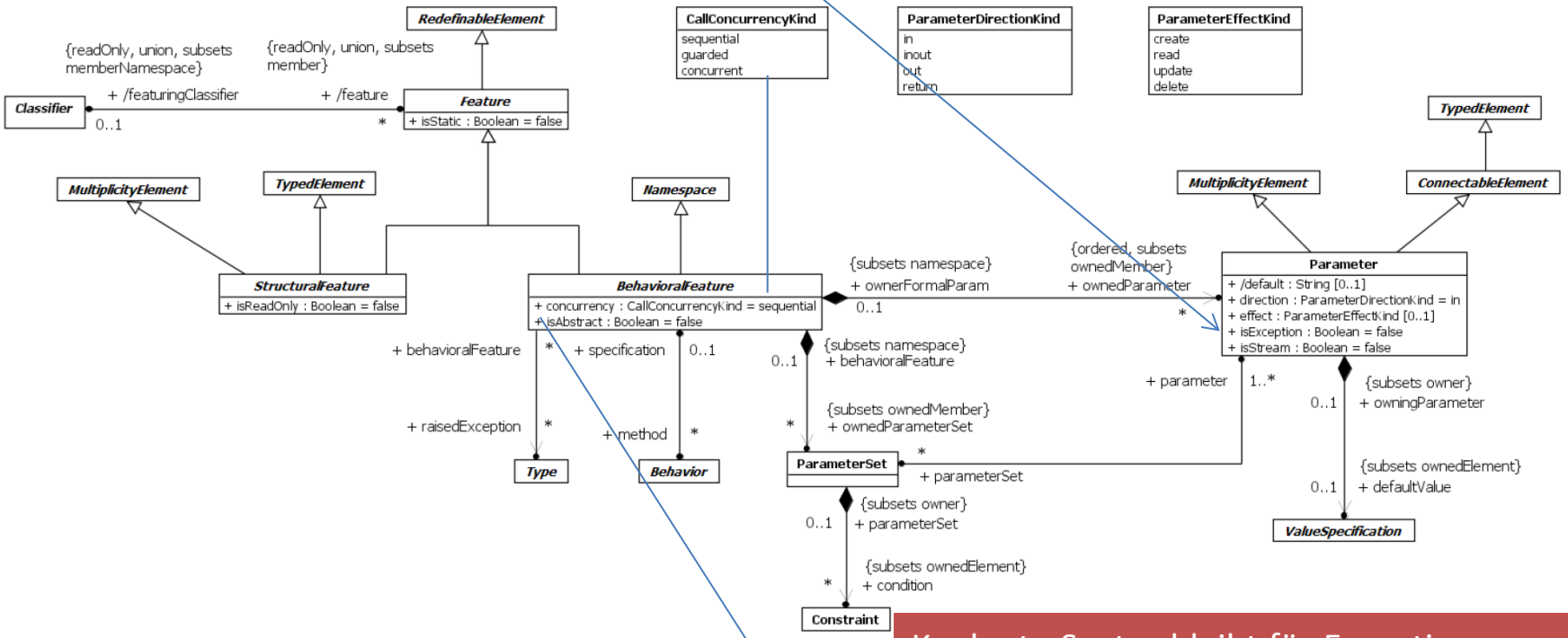


Figure 9.9 Features

Konkrete Syntax bleibt für Exceptions dennoch offen

A BehavioralFeature may raise an **exception** during its invocation. **Possible exception types** may be specified by attaching them to the BehavioralFeature using the **raisedException** association.

Ausnahmen

name (paramList) : returnType { propertyStringList }

UML (=2.5):

- Ausgabeparameter können
  als Exceptions markiert werden, aber wie?
- Typen können in **raisedException set**
  auftauchen

Ausnahmen gehören zur Signatur

Operation: Eigenschaften

Laufzeitbedingung
als Einschränkung (constraint) …

[«stereotype»]
[visibility]
name ([paramList]) [: returnType]
[{ propertyStringList }]

- betrifft das Verhalten bei der Interaktion von Ruferobjekten und gerufenen Objekten
- stellt einen Vertrag dar, der von möglichen **Methoden** einzuhalten ist
- zwei Notationsmöglichkeiten { … }
  - nach der Operationssignatur
  - als Kommentar mit Verbindungslinie
- Vorbedingung (*precondition*)
  - Zustandscharakterisierung vor Aufruf
- Nachbedingung (*postcondition*)
  - Zustandscharaketrisierung mit Beendigung
- Ergebnisbedingung (*bodycondition*)
  - Einschränkung des Rückgabewertes

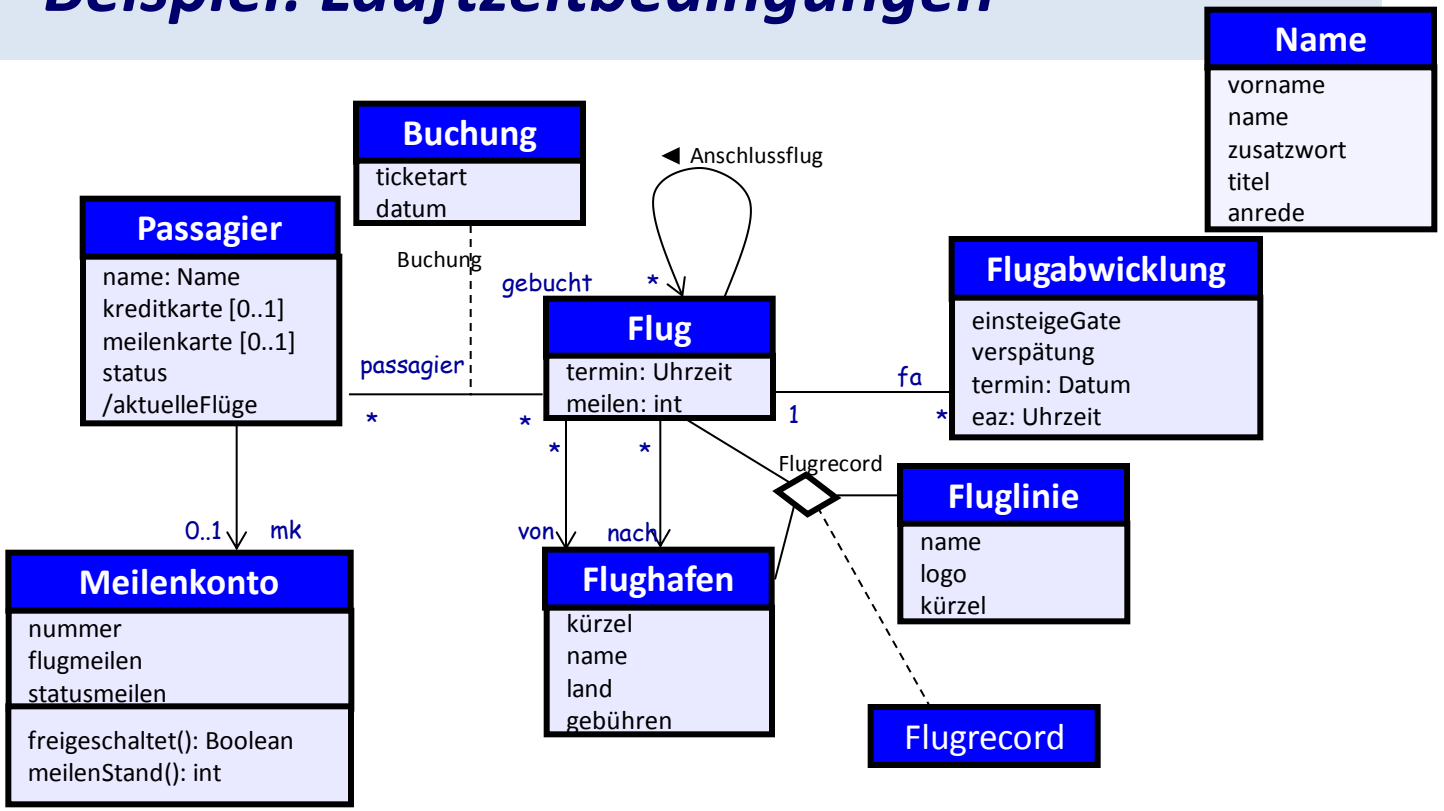
Bedingungen formuliert
- in Pseudocode
- als informaler Text
- in OCL

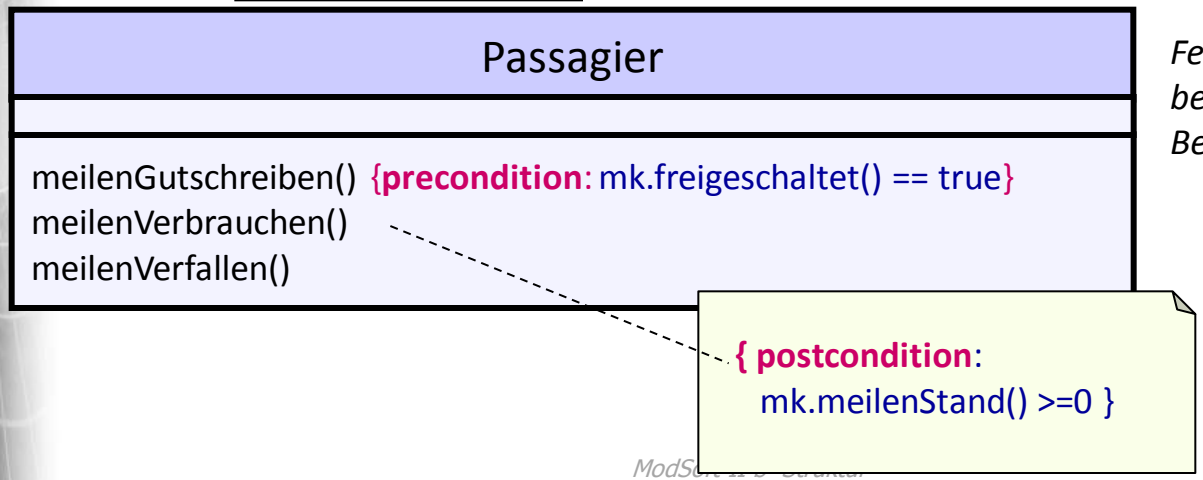
BodyCondition:
kann im Gegensatz zu
Pre- und Post-Bedingungen bei
Spezialisierung redefiniert werden

Klassendiagramme - eine vertiefende Betrachtung

1. Attribute, Assoziationen und Links
2. Operationen und Methoden
   - Metaklasse Operation
   - Allgemeine Charakterisierung
   - Namensräume, Überladung von Operationen
   - Konstrukturen, Destruktoren
   - Allgemeine konkrete Syntax
   - Standard-Eigenschaften einer Operation:
     Leaf, Abstract, Query, Static, Guarded, Sequential, Concurrent
   - Parameter-Eigenschaften
   - Weitere verhaltenseinschränkende Operationseigenschaften
   - Redefinition als Operationseigenschaft

Beispiel: Lauftzeitbedingungen





Festlegung des Kontextes bei der Angabe von Bedingungen ist zwingend

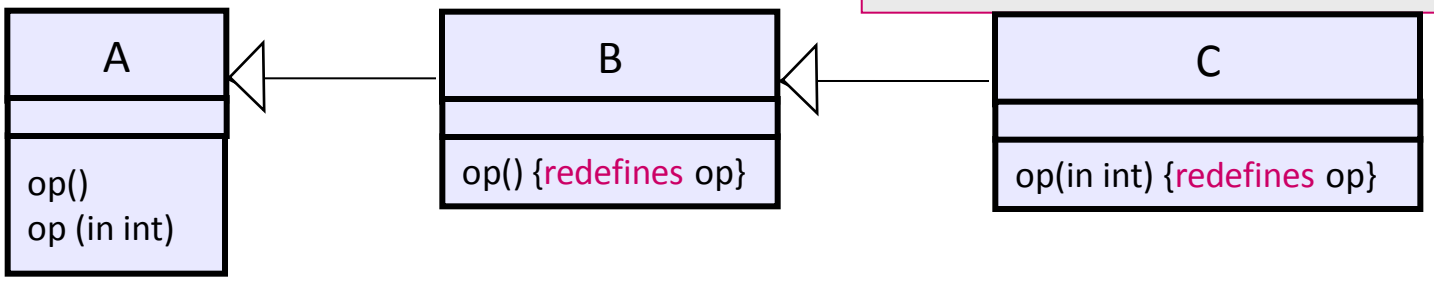
Klassendiagramme - eine vertiefende Betrachtung

1. Attribute, Assoziationen und Links
2. Operationen und Methoden
   - Metaklasse Operation
   - Allgemeine Charakterisierung
   - Namensräume, Überladung von Operationen
   - Konstrukturen, Destruktoren
   - Allgemeine konkrete Syntax
   - Standard-Eigenschaften einer Operation:
     Leaf, Abstract, Query, Static, Guarded, Sequential, Concurrent
   - Parameter-Eigenschaften
   - Weitere verhaltenseinschränkende Operationseigenschaften
   - Redefinition als Operationseigenschaft

Operation: Eigenschaften

[«stereotype»]
[visibility]
name ([paramList]) [: returnType]
[{ propertyStringList }]

Redefinition (redefines)



zwei Operationen mit **gleicher Signatur** sind über Spezialisierungsgrenzen hinweg nur erlaubt,
wenn die Spezialisierung dabei die Operation als *redefines* kennzeichnet

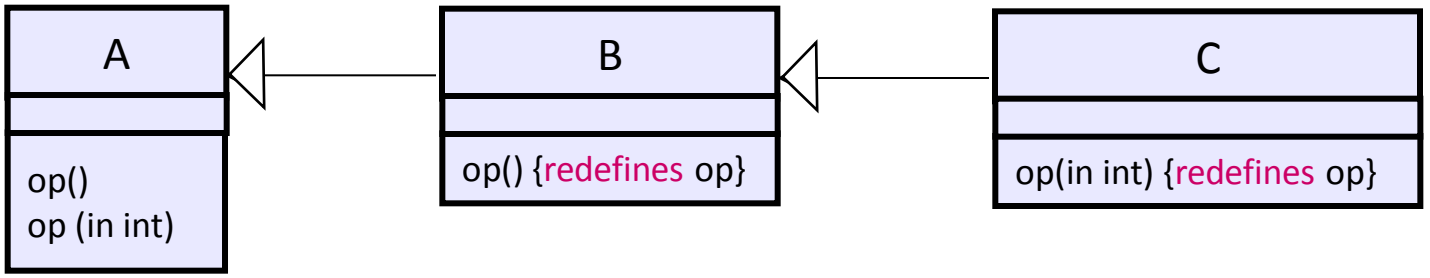
An Operation may be redefined in a specialization of the featuringClassifier.
This redefinition may **add** new preconditions or postconditions,
add new raisedExceptions, or otherwise refine the specification of the Operation.

Aber wie sind Parameter zu interpretieren, deren Typen in Vererbungsrelation stehen?

Operation: Eigenschaften

Redefinition (redefines)

[«stereotype»]
[visibility]
name ([paramList]) [: returnType]
[{ propertyStringList }]



Im Kontext einer Classifier-Spezialisierung ist die Redefinition von Operationen in UML mit semantischen Variationspunkten belegt, und zwar für die Überwachung von

- Invarianz (Preconditions)
- Covarianz (Redefinition der Ausgabe- und Rückgabeparameter) und
- Contravarianz (Redefinition von Eingabeparametern)

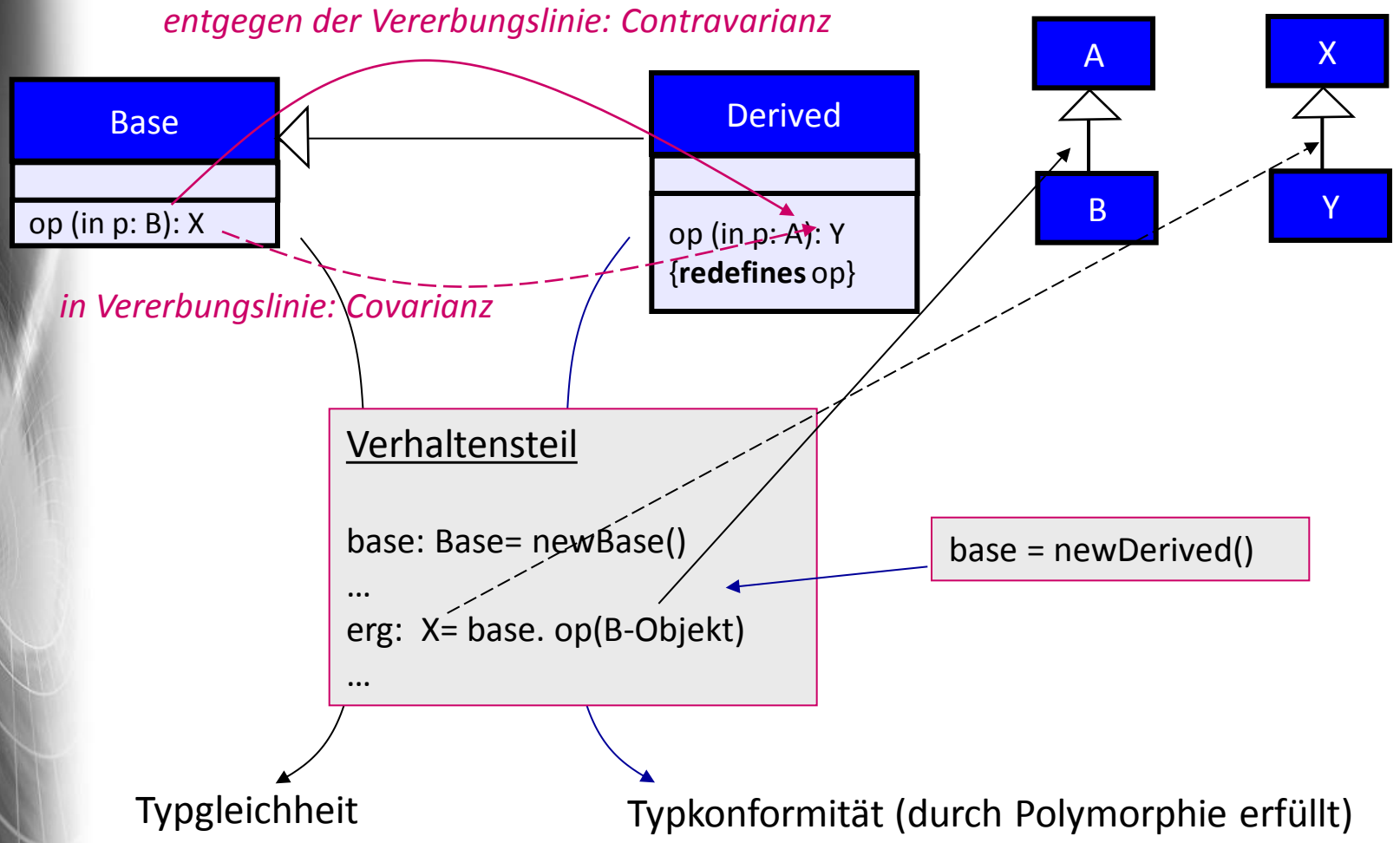
Zielsprachabhängige Typkonformität

Different type-conformance systems adopt different schemes
for how the types of parameters and results may vary
when an Operation is **redefined** in a specialization.

- When the type may not vary, it is called **invariance**.
- When the parameter type may be specialized in a specialized type, it is called **covariance**.
- When the parameter type may be generalized in a specialized type, it is called **contravariance**.

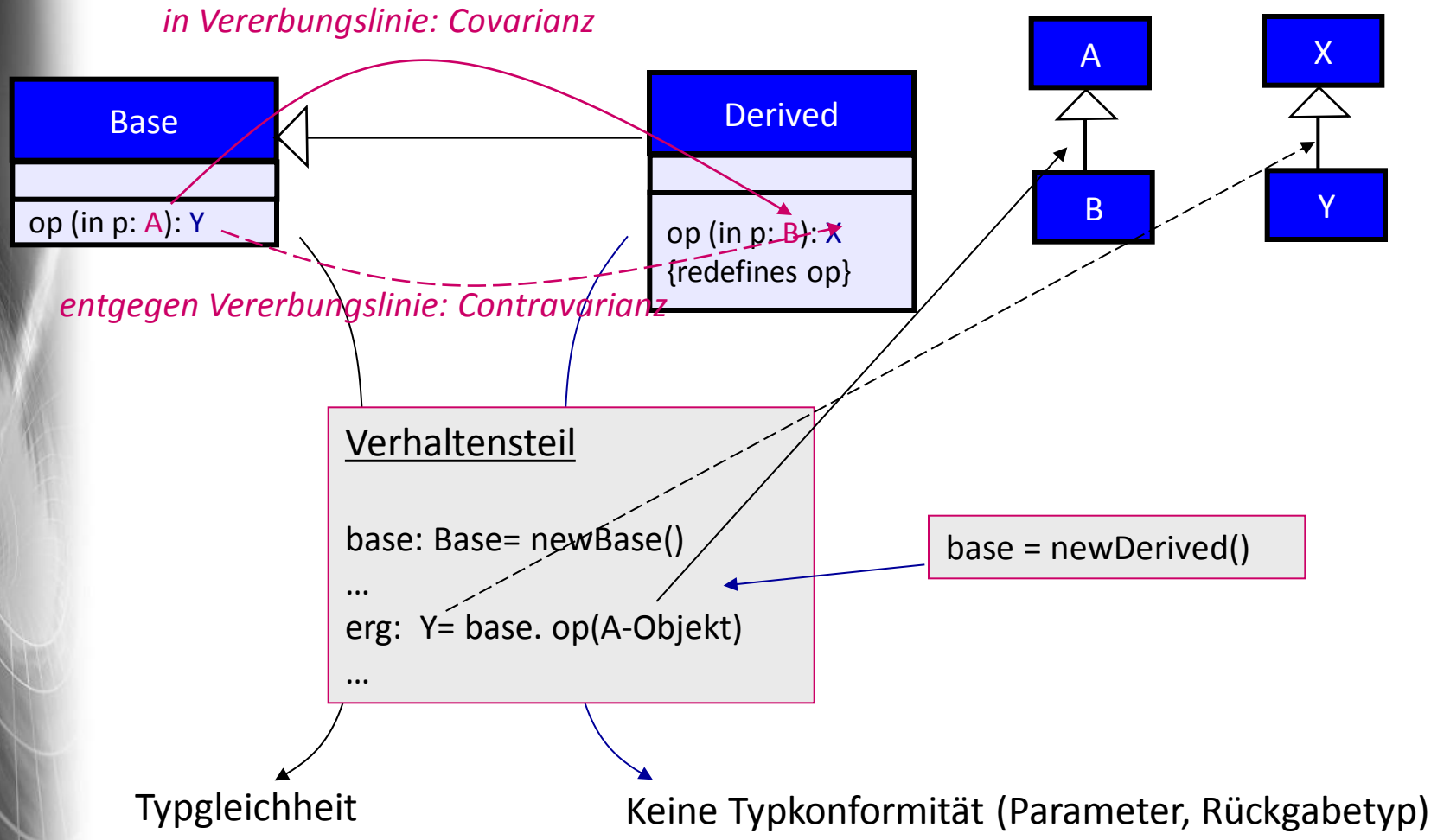
Redefinition: Covarianz und Contravarianz

Frage: ist folgende Redefinition sinnvoll?



Redefinition: Covarianz und Contravarianz

Frage: was liefert die Umkehrung der Verhältnisse



Operation: redefines (Diskussion von Typkonformität)

Im Kontext einer Classifier-Spezialisierung ist die Redefinition von Operationen mit semantischen Variationspunkten für die Überwachung von
- Invarianz (nicht erfüllte Preconditions)
- Covarianz (Redefinition der Ausgabe- und Rückgabeparameter) und
- Contravarianz (Redefinition von Eingabeparametern)

belegt.

vernünftig wären Konformitätsregeln auf der Basis folgender Forderungen :
- Contravarianz für Eingabeparameter
- Covarianz für Ausgabe- und Return-Parameter