Classes and Types in an Ideal Object-Oriented Programming Language

Roland Ducournau

Classes and Types in an Ideal Object-Oriented Programming Language

Roland Ducournau

LIRMM – Université de Montpellier & CNRS

东北大学, 沈阳市 – April 2016

Motivation: the good news

Object-orientation

is now universal for programming, modelling, ..

Mature theory and technology

- ≈ 24 centuries after Aristotle (350 BC),
- ≈ half a century after Simula (1967),
- ≈ 3 decades ago: first *mainstream* languages (Eiffel, C++)
- ≈ 2 decades ago: Java, then C♯ and Scala

☛ Likely the greatest success of the last century!

LIRMM

Motivation: the good news

Object-orientation

is now universal for programming, modelling, ..

Mature theory and technology

- ≈ 24 centuries after Aristotle (350 BC),
- ≈ half a century after Simula (1967),
- ≈ 3 decades ago: first *mainstream* languages (Eiffel, C++)
- ≈ 2 decades ago: Java, then C♯ and Scala

☛ Likely the greatest success of the last century!

LIRMM

Motivation: the good news

Object-orientation

is now universal for programming, modelling, ..

Mature theory and technology

- ≈ 24 centuries after Aristotle (350 BC),
- ≈ half a century after Simula (1967),
- ≈ 3 decades ago: first *mainstream* languages (Eiffel, C++)
- ≈ 2 decades ago: Java, then C♯ and Scala

☛ Likely the greatest success of the last century!

LIRMM

Motivation: the bad news

The greatest failure of the last century?

The object-oriented programming languages!

- Each one, individually!
- All together!

The same features

- are specified differently,
- as if programming languages were works of art!

☛ The Babel Tower!

Motivation: the bad news

The greatest failure of the last century?

The object-oriented programming languages!

- Each one, individually!
- All together!

The same features

- are specified differently,
- as if programming languages were works of art!

☛ The Babel Tower!

LIRMM

Motivation: the bad news

The greatest failure of the last century?

The object-oriented programming languages!

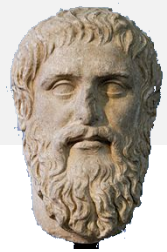
- Each one, individually!
- All together!

The same features

- are specified differently,
- as if programming languages were works of art!

☛ The Babel Tower!

My thesis



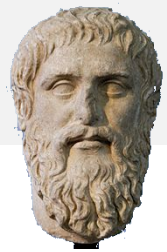
Plato's ideals

- apply to Circle, Tree, ..
- apply to Programming Languages, too

☛ The ideal Object-Oriented Programming Language exists

LIRMM

My thesis



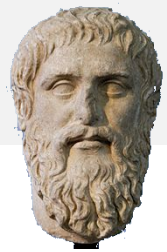
Plato's ideals

- apply to Circle, Tree, ..
- apply to Programming Languages, too

☛ The ideal Object-Oriented Programming Language exists

LIRMM

My thesis



Arguments taken from ...

- philosophy (Aristotle)
- ontology (object metamodel)
- necessity (Occam's razor)
- mathematics (type and set theory, logic)
- empiricism
- common sense

LIRMM

Plan

1. Classes and inheritance
2. Types and subtyping
3. Genericity
4. Conclusions and propects

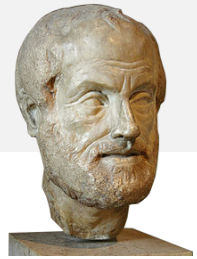
Plan

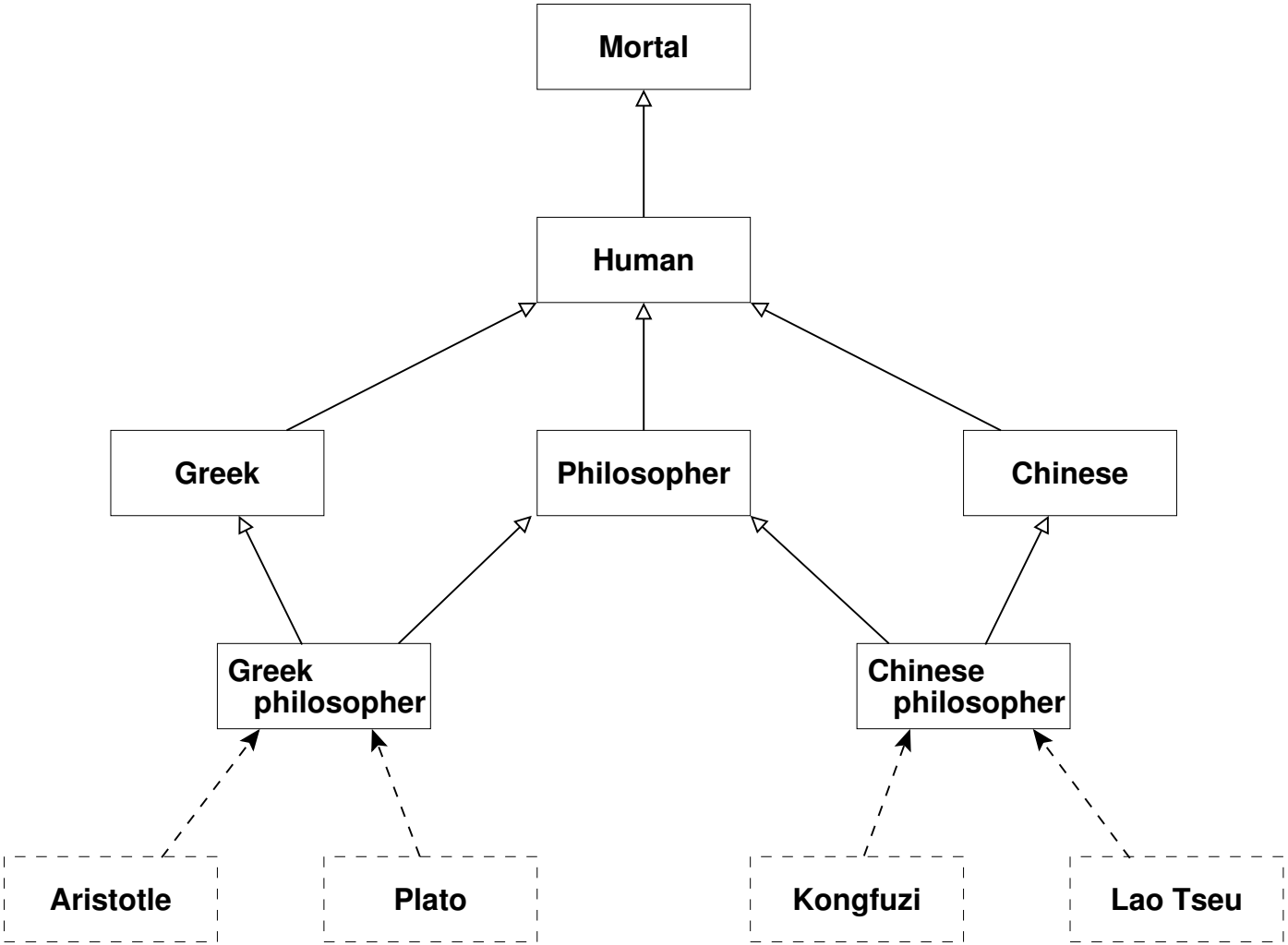
LIRMM

Object-orientation vs knowledge representation

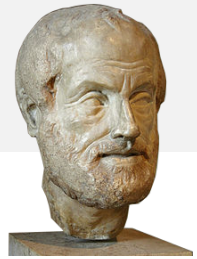
☛ an object-oriented model is a representation of the real-world™

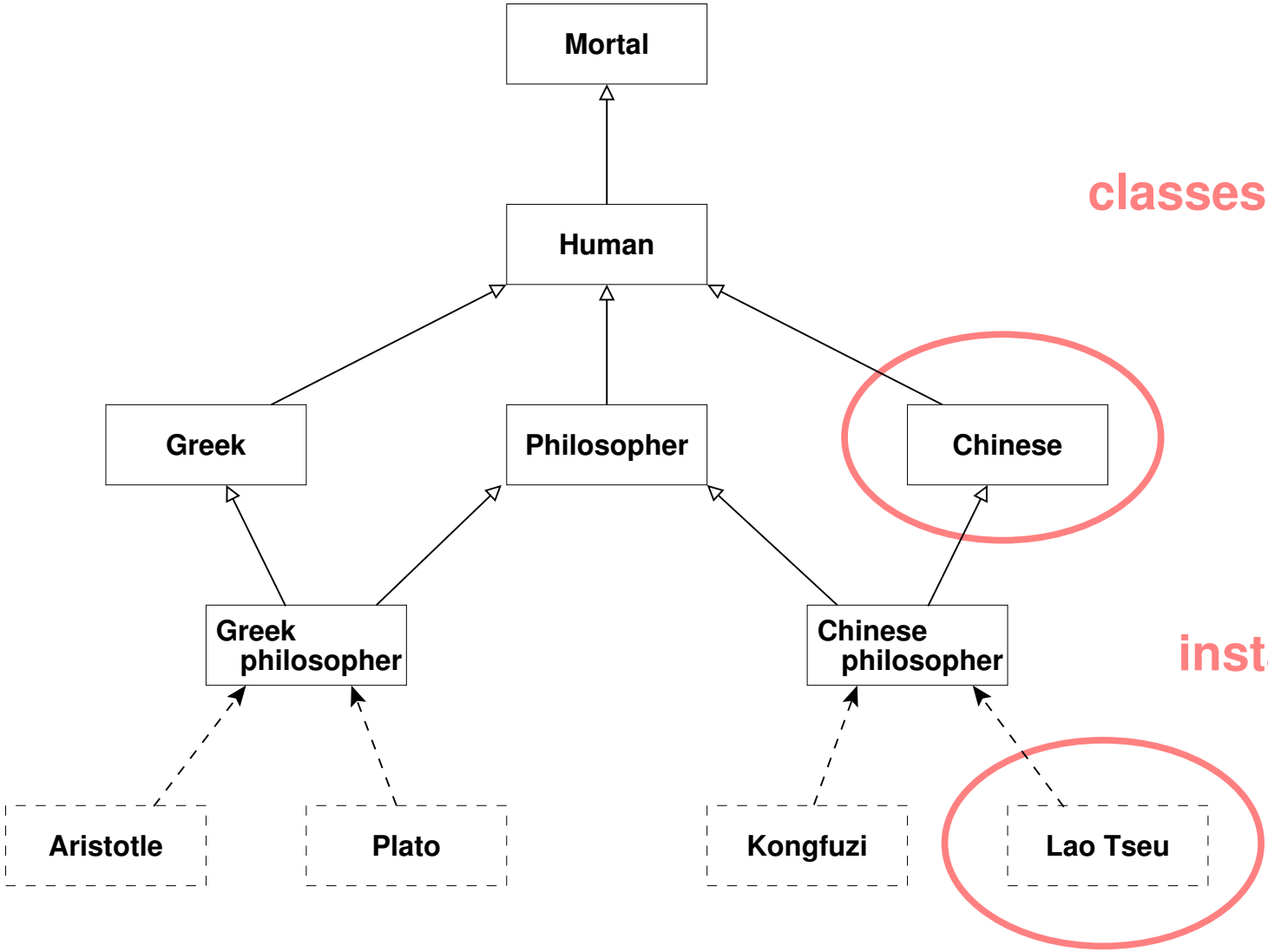
OO vs KR: Philosophers



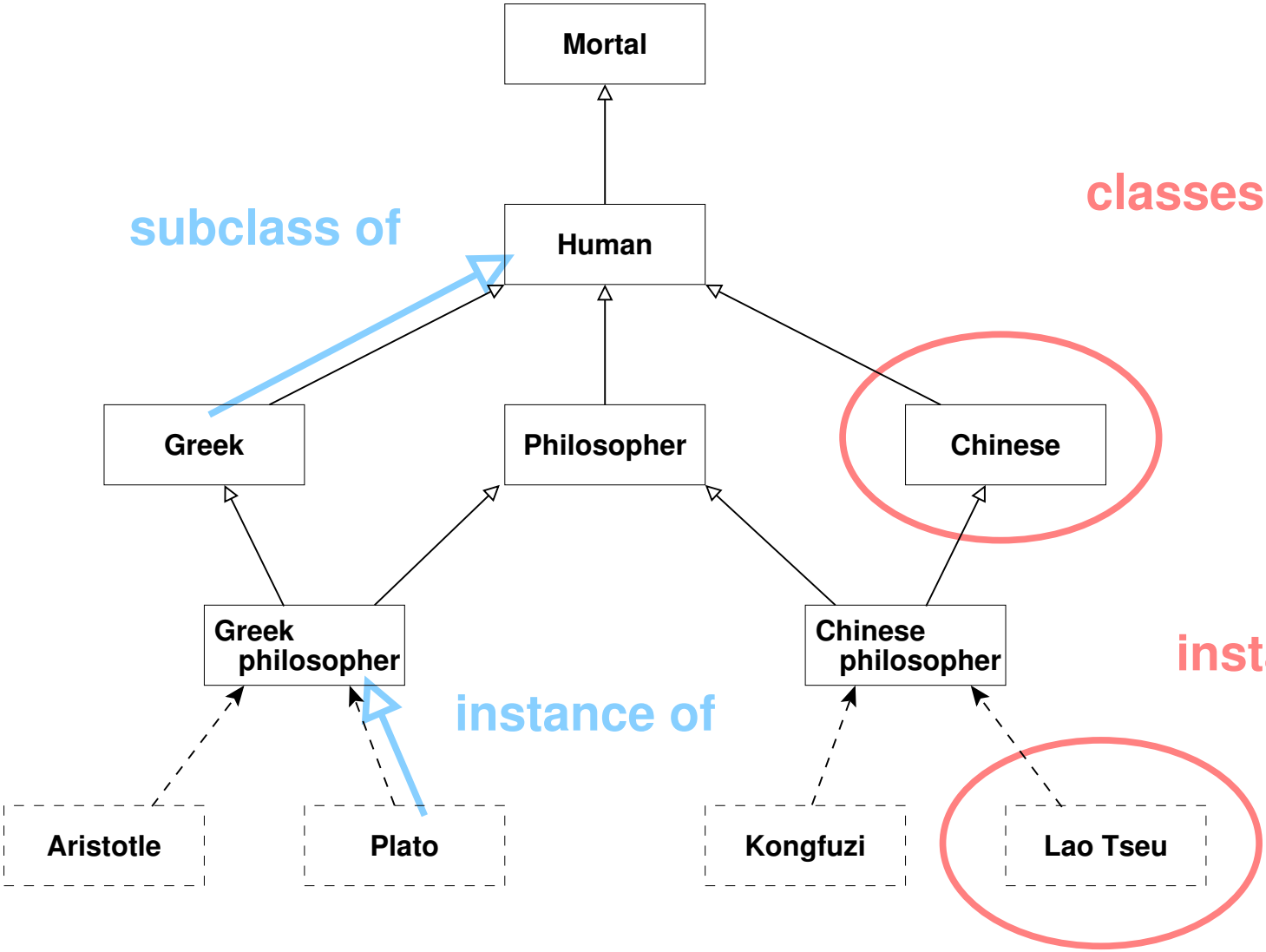


OO vs KR: Philosophers

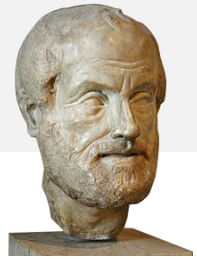




OO vs KR: Philosophers



Philosophers



Aristotle (Ἀριστοτέλης , 384-322 BC) founded logic

Plato (Πλάτων , 427-348 BC) promoted the existence of ideas

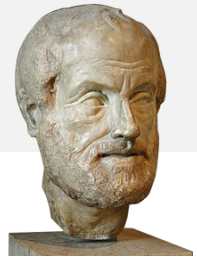
孔子 (Kongfuzi, 551-479 BC)

老子 (Lao Tseu)

庄子 (Zhuangzi) a butterfly dream

☛ in the XX° century jargon, ideas are first-class objects

Philosophers



Aristotle (Ἀριστοτέλης , 384-322 BC) founded logic

Plato (Πλάτων , 427-348 BC) promoted the existence of ideas

孔子 (Kongfuzi, 551-479 BC)

老子 (Lao Tseu)

庄子 (Zhuangzi) a butterfly dream

☛ in the XX° century jargon, ideas are first-class objects

Aristotelian semantics (1/3)

The extension of a class is the set of its instances

Foundation syllogism

Humans are Mortals 孔子 is a Human 孔子 is a Mortal	$\text{Human} \prec \text{Mortal}$ $\text{孔子} \in Ext(\text{Human})$ $\text{孔子} \in Ext(\text{Mortal})$

Subclassing = specialization = inclusion of extensions

☛ Instances of the subclass are instances of the superclass

$$B \prec A \Rightarrow Ext(B) \subset Ext(A)$$

LIRMM

Aristotelian semantics (1/3)

The extension of a class is the set of its instances

Foundation syllogism

$$\frac{\text{Humans are Mortals} \quad \text{孔子 is a Human}}{\text{孔子 is a Mortal}}$$

$$\frac{\text{Human} \prec \text{Mortal} \quad \text{孔子} \in \mathit{Ext}(\text{Human})}{\text{孔子} \in \mathit{Ext}(\text{Mortal})}$$

Subclassing = specialization = inclusion of extensions

- Instances of the subclass are instances of the superclass

$$B \prec A \Rightarrow \mathit{Ext}(B) \subset \mathit{Ext}(A)$$

Aristotelian semantics (2/3)

The intension of a class is a set of properties declared for its instances

Inheritance of properties

- an instance of a class has all the properties declared by the class
- as a Human, 孔子 is a Mortal

☛ 孔子 has all the properties declared by Mortal

Inheritance is implied by specialization

☛ The subclass inherits the properties declared in the superclass

$$B \prec A \Rightarrow Int(A) \subset Int(B)$$

LIRMM

Aristotelian semantics (2/3)

The intension of a class is a set of properties declared for its instances

Inheritance of properties

- an instance of a class has all the properties declared by the class
- as a Human, 孔子 is a Mortal

☛ 孔子 has all the properties declared by Mortal

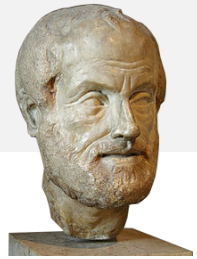
Inheritance is implied by specialization

☛ The subclass inherits the properties declared in the superclass

$$B \prec A \Rightarrow Int(A) \subset Int(B)$$

LIRMM

Aristotelian semantics (3/3)



My answers to objections

- in Logo, a Turtle is a Point
  there is no specialization in the real world™
- ☛ but specialization is in the artefact
- so-called implementation inheritance
- ☛ a bad practice resulting from an erroneous model

An object model of the object model (1/3)

Object-orientation is part of the real world™

☛ object-orientation can be used for representing object-orientation

An object meta-model

- a UML model
- modelling the entities of object-orientation,
  i.e. classes, associations, attributes, methods, ...
- with classes, associations, attributes, methods, ...

An object model of the object model (2/3)

Motivations

- mandatory for all metaprograms (e.g. compilers, VMs, IDEs)
- provides an ontology of object orientation
- with unambiguous specifications
- by getting rid of names

An object model of the object model (3/3)

Language ambiguities

- natural languages are inherently ambiguous
  - ☛ plays on words
- programming languages, although formal, are ambiguous, too
  - because they serve as man-machine interfaces
  - through various names
  - ☛ compilers don't joke!

Foundation requirement

☛ in the modelled program, each occurrence of the name of a modelled entity must denote a single instance of the metamodel

An object model of the object model (3/3)

Language ambiguities

- natural languages are inherently ambiguous
  - ☛ plays on words
- programming languages, although formal, are ambiguous, too
  - because they serve as man-machine interfaces
  - through various names
  - ☛ compilers don't joke!

Foundation requirement

☛ in the modelled program, each occurrence of the name of a modelled entity must denote a single instance of the metamodel

```
1   class A {
2       foo() {...}
    }

3   class B extends A {
4       foo() {...}
5       bar() {...}
    }
```

a **class**, named **A**

a **method** named **foo()**, defined in **A**

a class named **B**, subclass of **A**

a method named **foo()**, defined in **B**
redefining **foo()** of **A**

a method named **bar()**, defined in **B**

```
1     A x;
3     B y;
6     x.foo();
7     y.bar();
```

a type annotation with class **A**

a type annotation with class **B**

a **message** **foo()** introduced in **A**
sent to **x** with late binding

a message **bar()** introduced in **B**

```
1  class A {
2      foo() {...}
   }

3  class B extends A {
4      foo() {...}
5      bar() {...}
   }
```

a **class**, named **A**

a **method** named **foo()**, defined in **A**

a class named **B**, subclass of **A**

a method named **foo()**, defined in **B**
redefining **foo()** of **A**

a method named **bar()**, defined in **B**

```
1    A x;
3    B y;
6    x.foo();
7    y.bar();
```

a type annotation with class **A**

a type annotation with class **B**

a **message** **foo()** introduced in **A**
sent to **x** with late binding

a message **bar()** introduced in **B**

```
1   class A {
2           foo() {...}
    }

3   class B extends A {
4           foo() {...}
5           bar() {...}
    }
```

a **class**, named **A**

a **method** named **foo()**, defined in **A**

a class named **B**, subclass of **A**

a method named **foo()**, defined in **B**
redefining **foo()** of **A**

a method named **bar()**, defined in **B**

```
1       A x;
3       B y;
6       x.foo();
7       y.bar();
```

a type annotation with class **A**

a type annotation with class **B**

a **message foo()** introduced in **A**
sent to **x** with late binding

a message **bar()** introduced in **B**

An object model of the object model (3/3)

A single class for classes

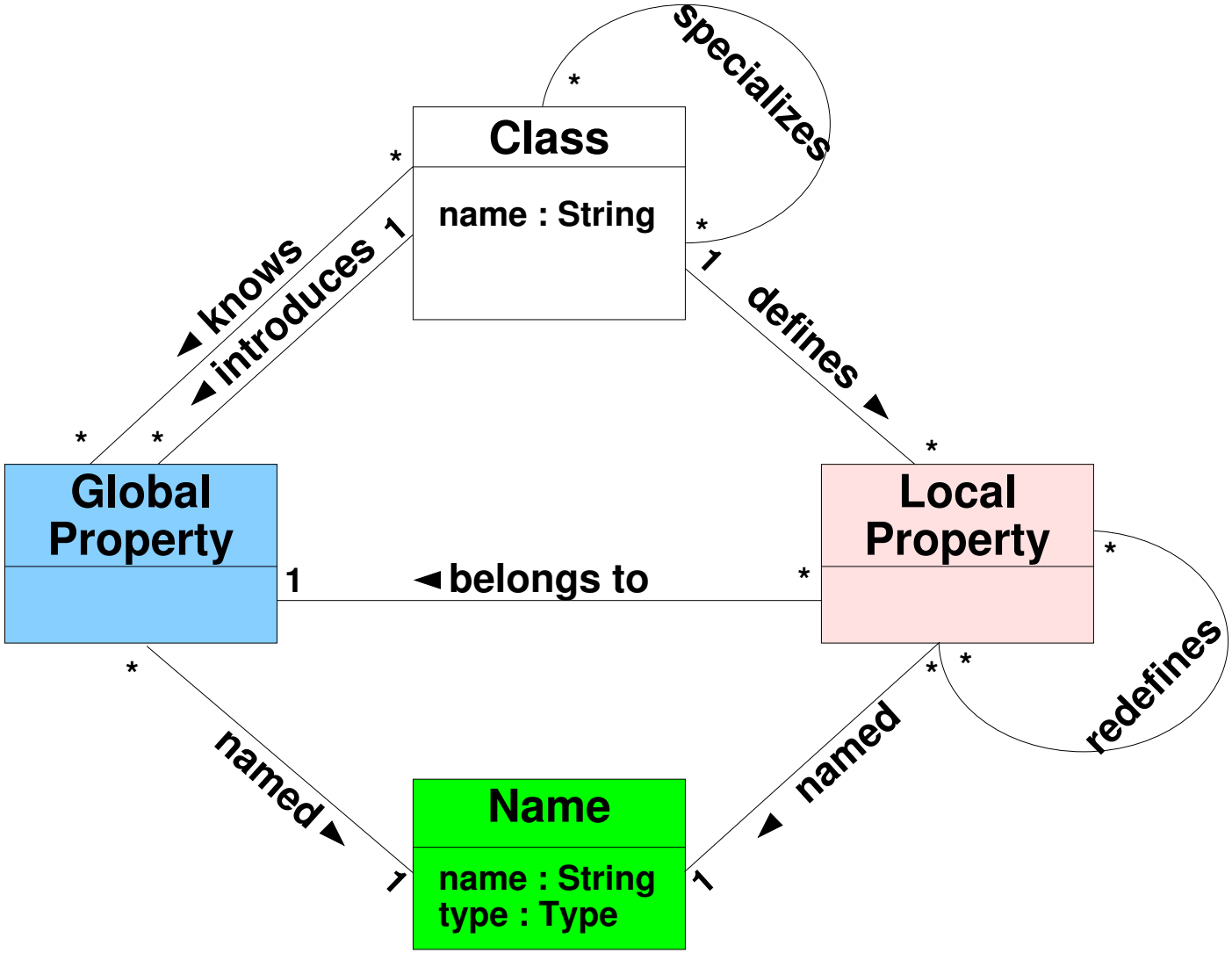
- for all usages: declarations, type annotations and **new**

Two classes for properties

- a class for local properties, implementations defined in a class
- a class for global properties, messages invoked from the code

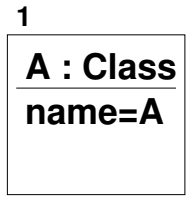
What are properties?

- methods, attributes
- formal type parameters, virtual types, ...


Class
name : String
specializes
*
*
knows
introduces
*
1
1
defines
*
Global Property
Local Property
1
belongs to
*
*
redefines
named
named
Name
name : String
type : Type
1
1

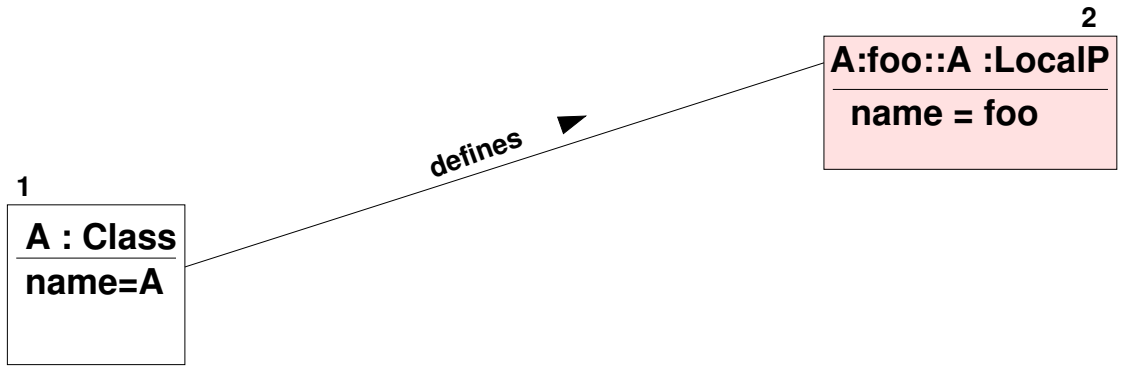
```
1   class A {
2           foo() {...}
    }

3   class B extends A {
4           foo() {...}
5           bar() {...}
    }
```

```
1       A x;
3       B y;
6       x.foo();
7       y.bar();
```



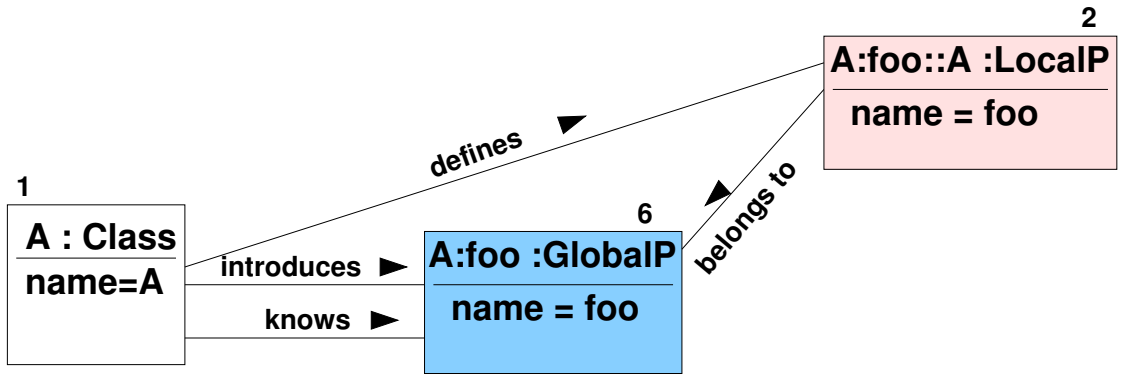
```
1   class A {
2           foo() {...}
    }

3   class B extends A {
4           foo() {...}
5           bar() {...}
    }
```

```
1       A x;
3       B y;
6       x.foo();
7       y.bar();
```



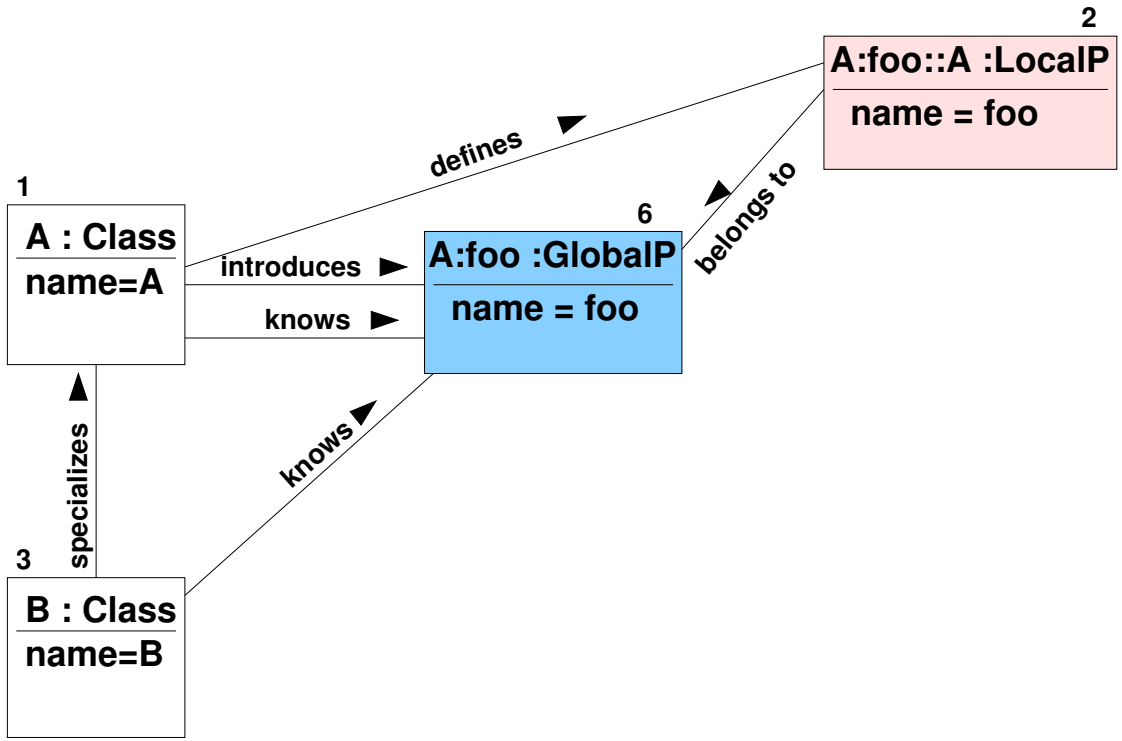
```
1   class A {
2         foo() {...}
    }

3   class B extends A {
4         foo() {...}
5         bar() {...}
    }
```

```
1     A x;
3     B y;
6     x.foo();
7     y.bar();
```



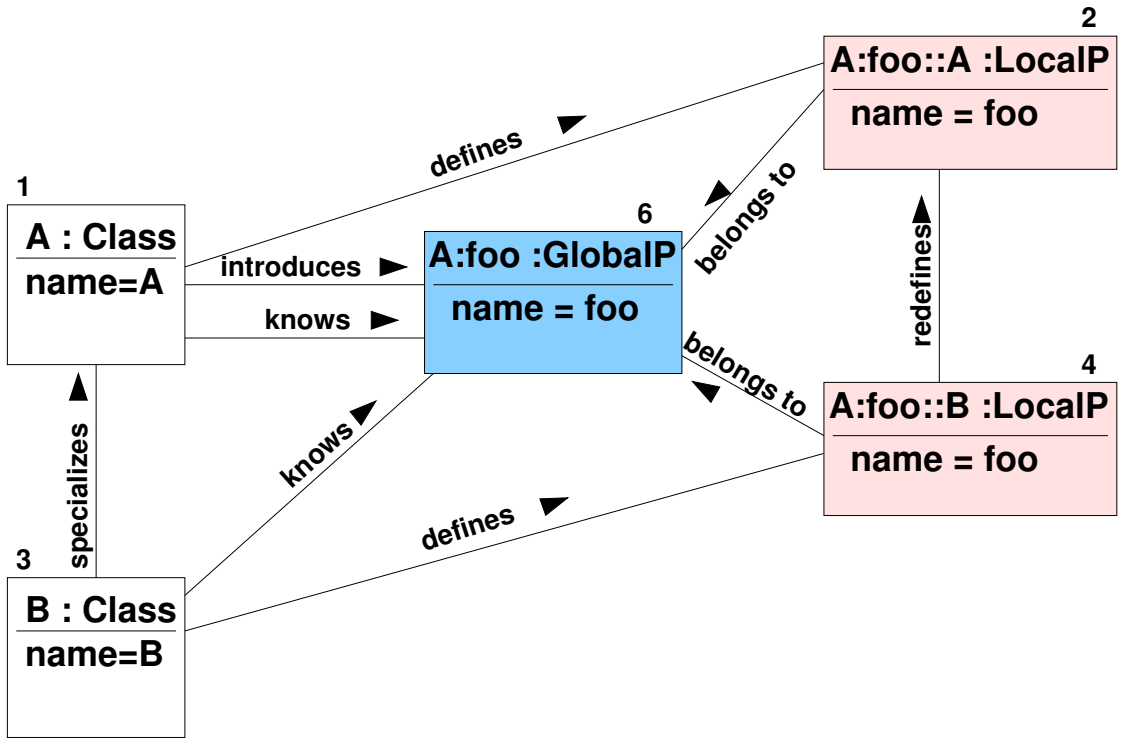
```
1   class A {
2         foo() {...}
    }

3   class B extends A {
4         foo() {...}
5         bar() {...}
    }
```

```
1     A x;
3     B y;
6     x.foo();
7     y.bar();
```



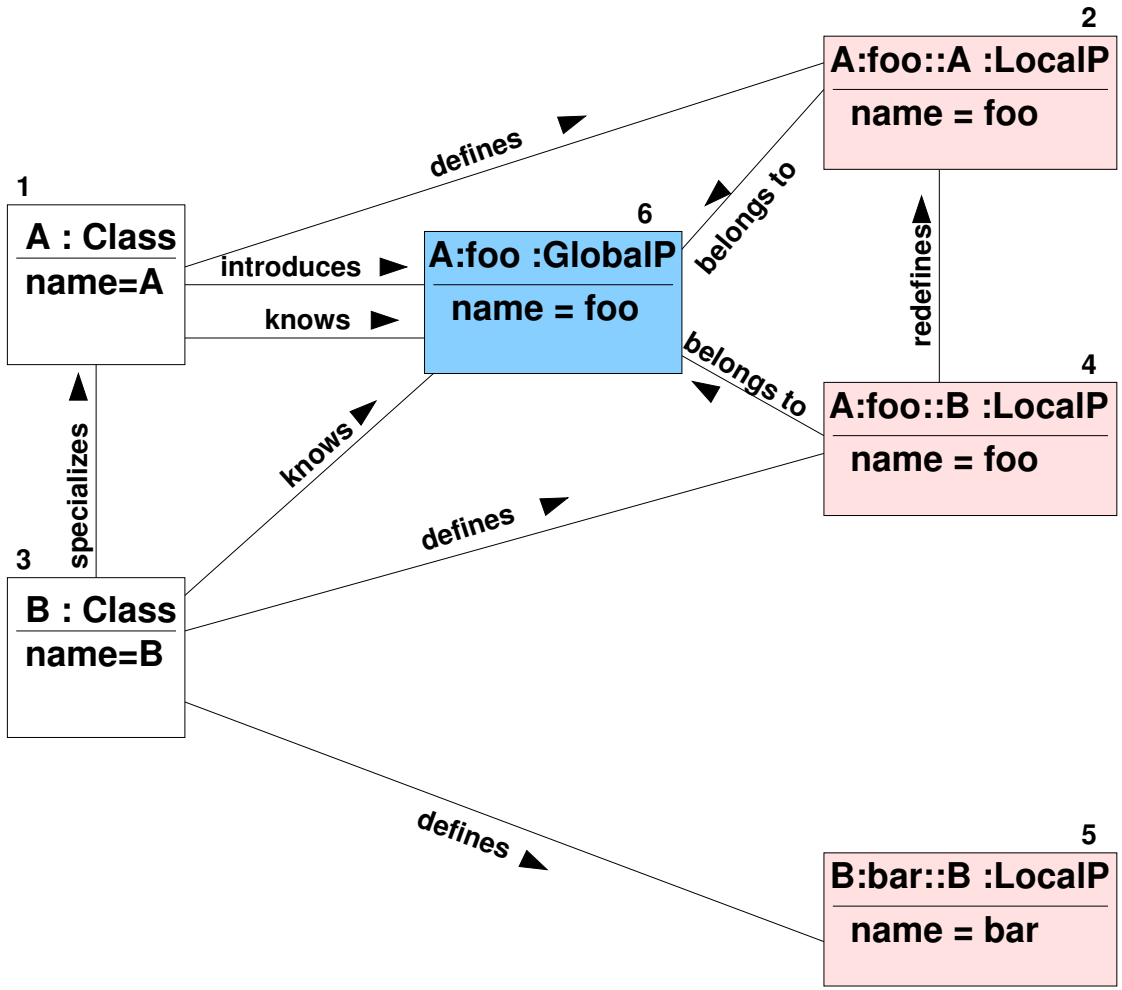
```
1   class A {
2         foo() {...}
    }

3   class B extends A {
4         foo() {...}
5         bar() {...}
    }
```

```
1     A x;
3     B y;
6     x.foo();
7     y.bar();
```



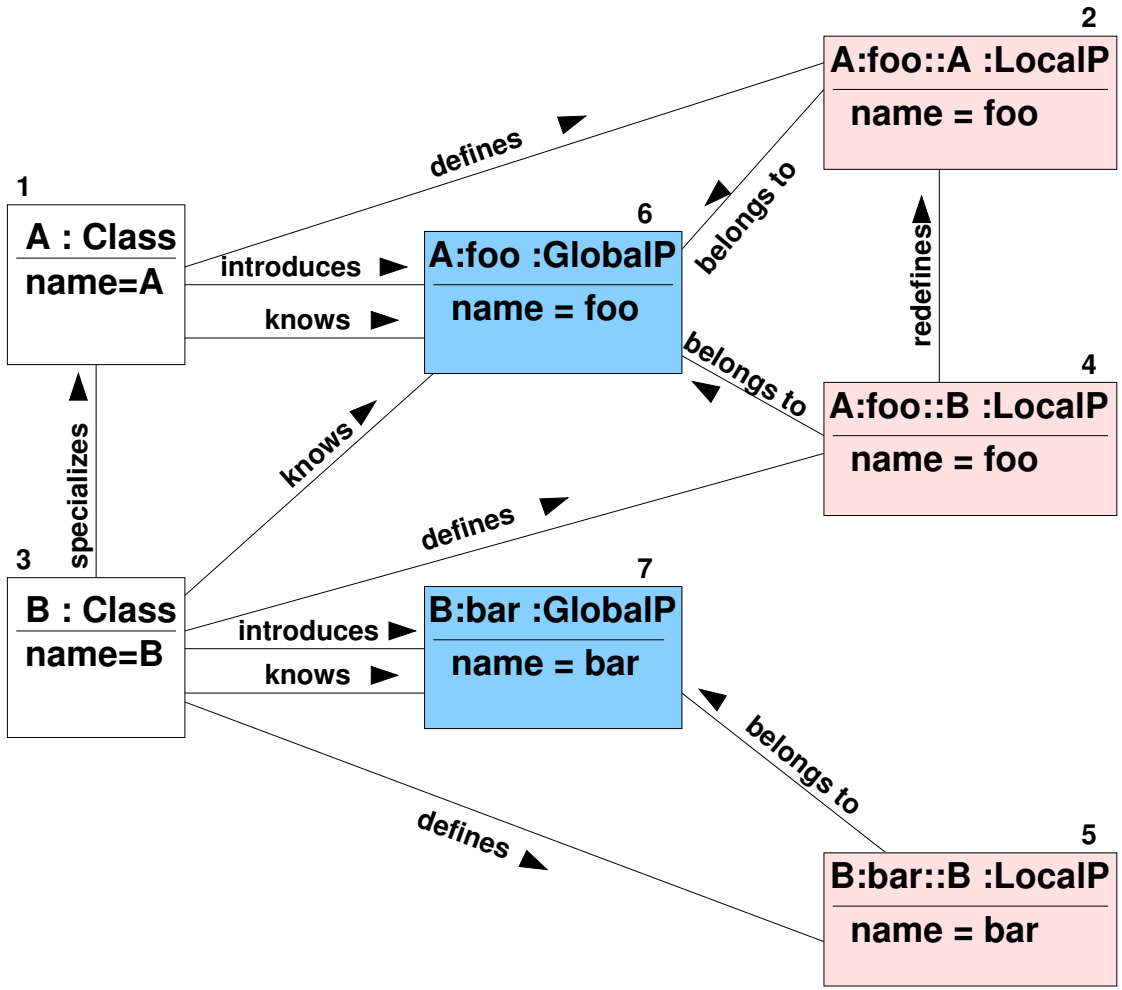
```
1   class A {
2         foo() {...}
    }

3   class B extends A {
4         foo() {...}
5         bar() {...}
    }
```

```
1     A x;
3     B y;
6     x.foo();
7     y.bar();
```



```
1   class A {
2         foo() {...}
    }

3   class B extends A {
4         foo() {...}
5         bar() {...}
    }
```

```
1     A x;
3     B y;
6     x.foo();
7     y.bar();
```



Metamodeling semantics

Disambiguating name conflicts

In two situations

- with multiple inheritance
- with static overloading

☛ by substituting an instance of local/global property to each property name, even when it seems ambiguous

Actual ambiguities = compiler errors

When this substitution is not possible (several candidates)

Metamodeling semantics

Disambiguating name conflicts

In two situations

- with multiple inheritance
- with static overloading

☛ by substituting an instance of local/global property to each property name, even when it seems ambiguous

Actual ambiguities = compiler errors

When this substitution is not possible (several candidates)

Motivation for multiple inheritance (1/2)

Multiple inheritance provides

- increased expressivity
- improved reuse

- given a class **A** providing a service **foo**
- and a class **B** providing a service **bar**
- both developped independently of each other (apart from common superclasses)

☛ define a common subclass **C** providing both services

Motivation for multiple inheritance (1/2)

Multiple inheritance provides

- increased expressivity
- improved reuse

- given a class **A** providing a service **foo**
- and a class **B** providing a service **bar**
- both developped independently of each other (apart from common superclasses)

☛ define a common subclass **C** providing both services

Motivation for multiple inheritance (2/2)

In static typing

There is no language without

- full multiple inheritance (C++, Eiffel),
- or mixins (Scala),
- or at least multiple subtyping (Java, C♯, Ada 2005)

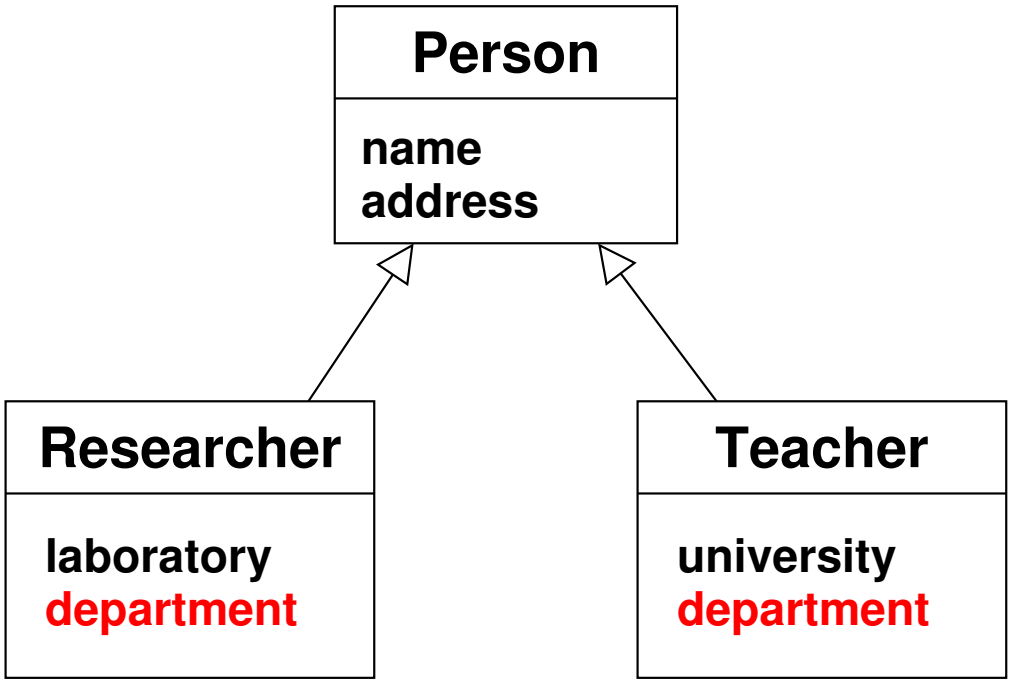
Multiple inheritance conflicts (1/5)

Conflicts of two kinds

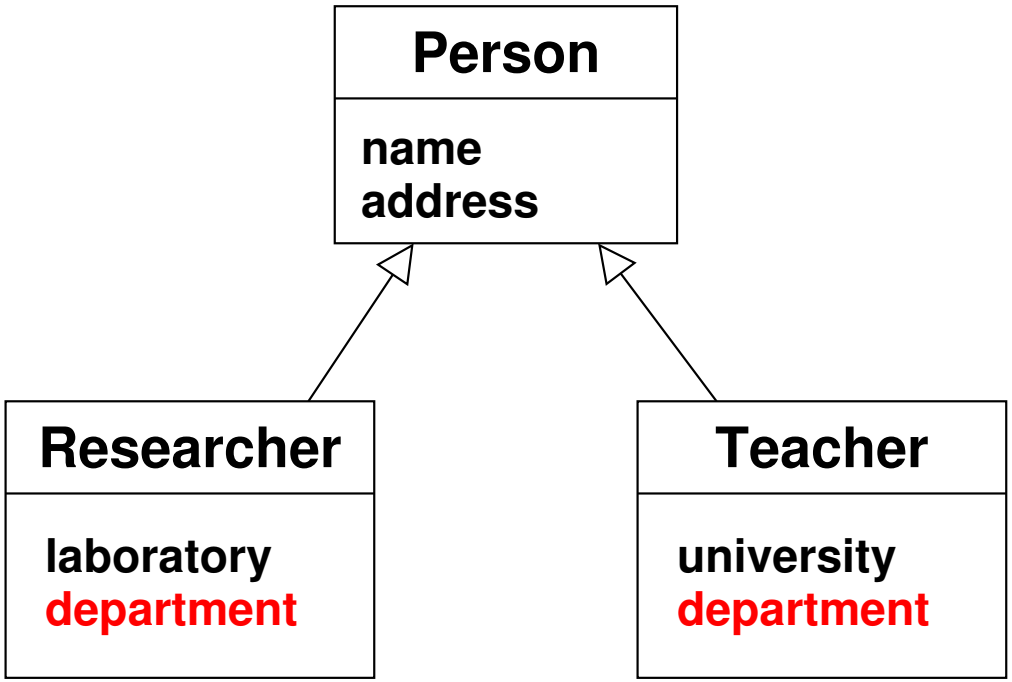
- between two global properties with the same name
- between two local properties of the same global property

☛ plus the method combination case

Multiple inheritance conflicts (2/5)

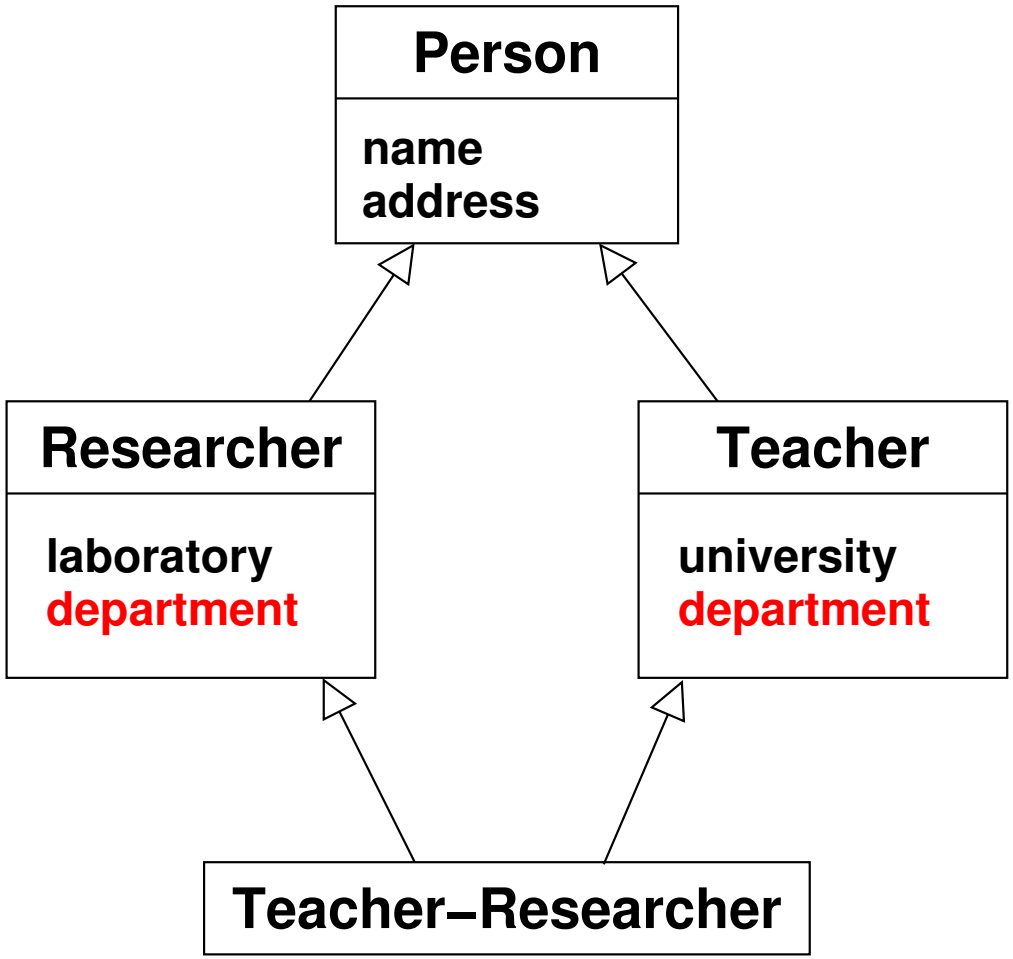


Multiple inheritance conflicts (2/5)



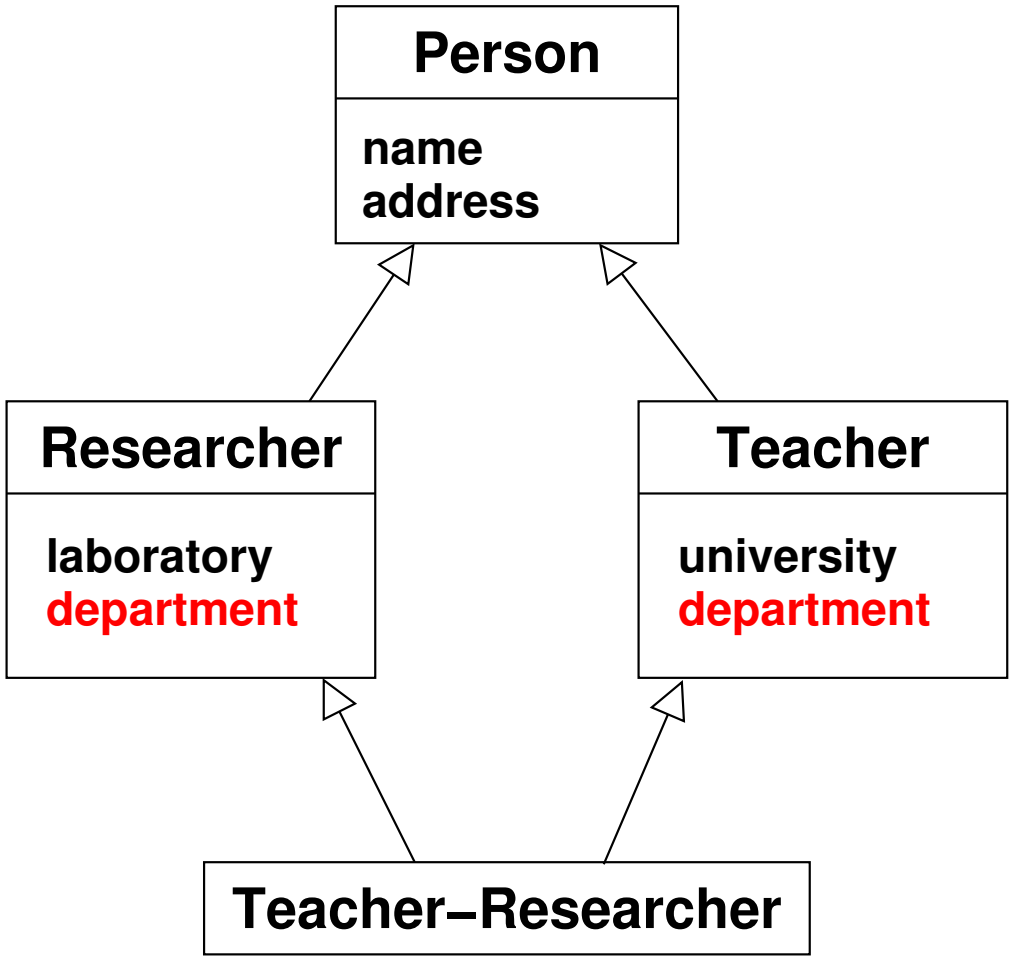

```
Researcher x;
x.department;      //  OK

Teacher y;
y.department       //  OK
```

Multiple inheritance conflicts (2/5)

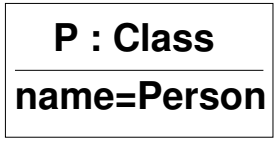


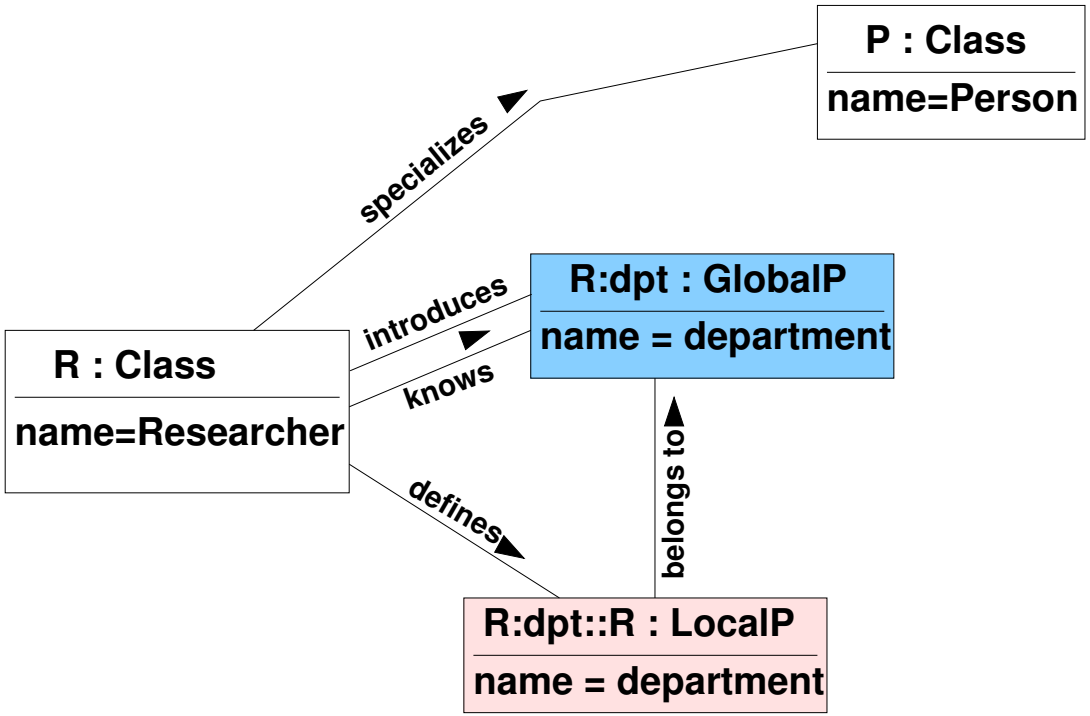

```
Researcher x;
x.department;      //  OK

Teacher y;
y.department       //  OK
```

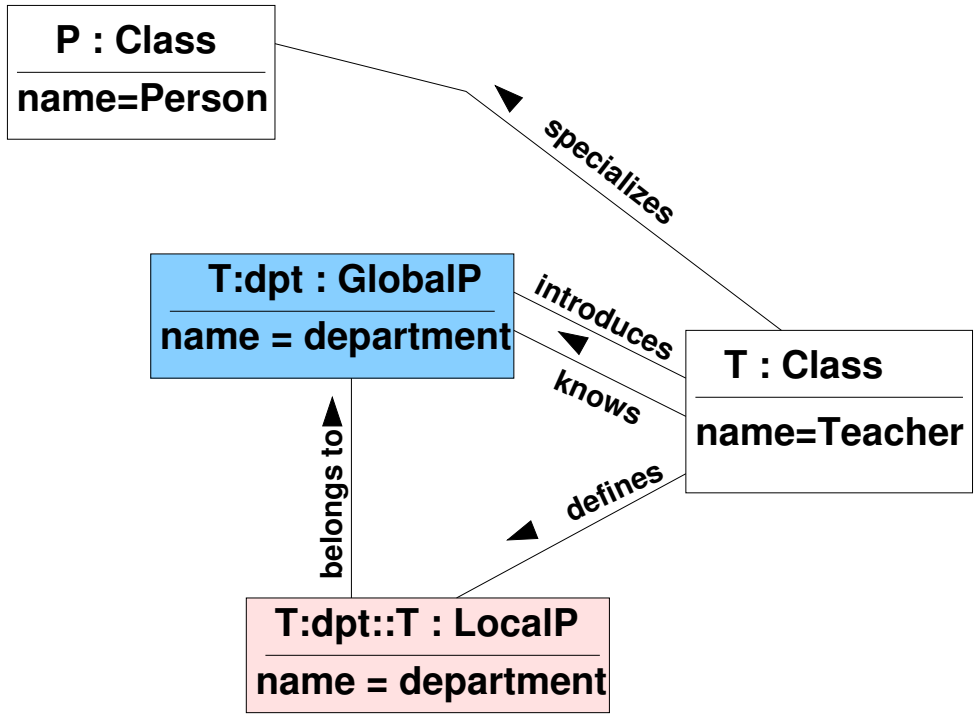
Multiple inheritance conflicts (2/5)

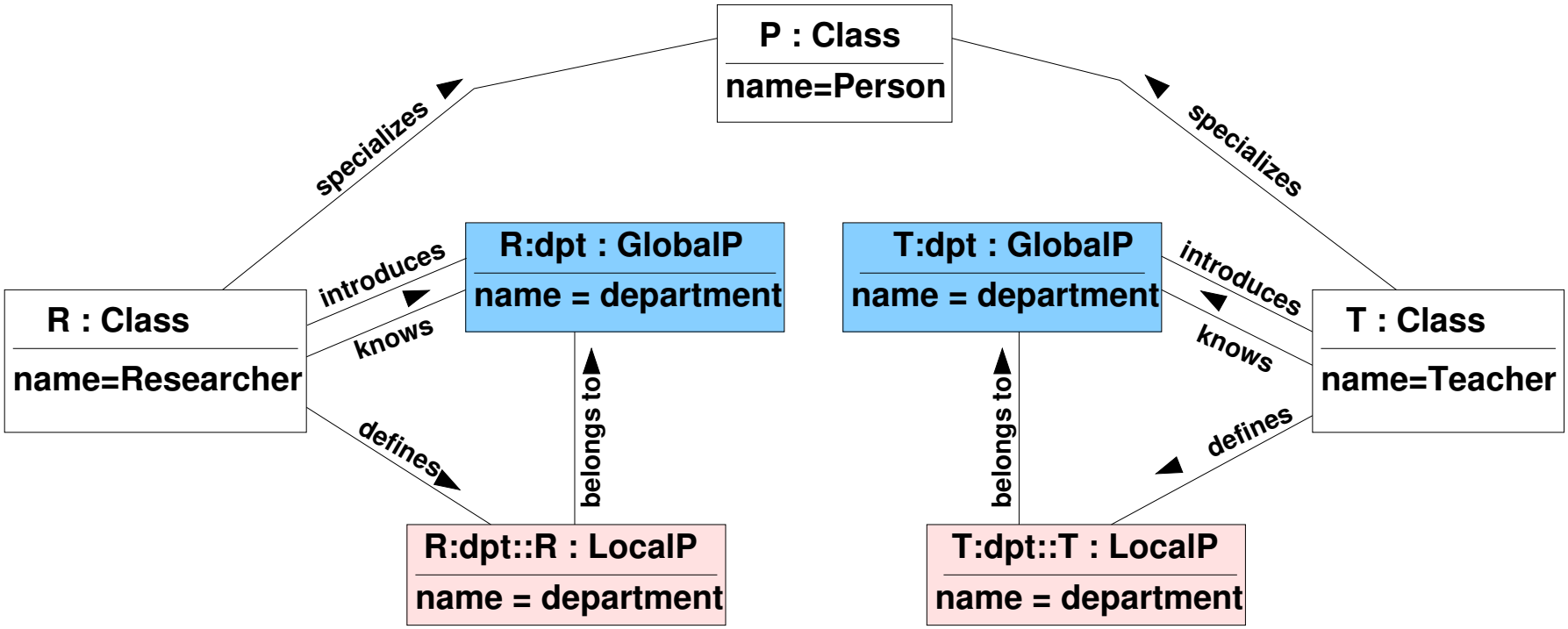


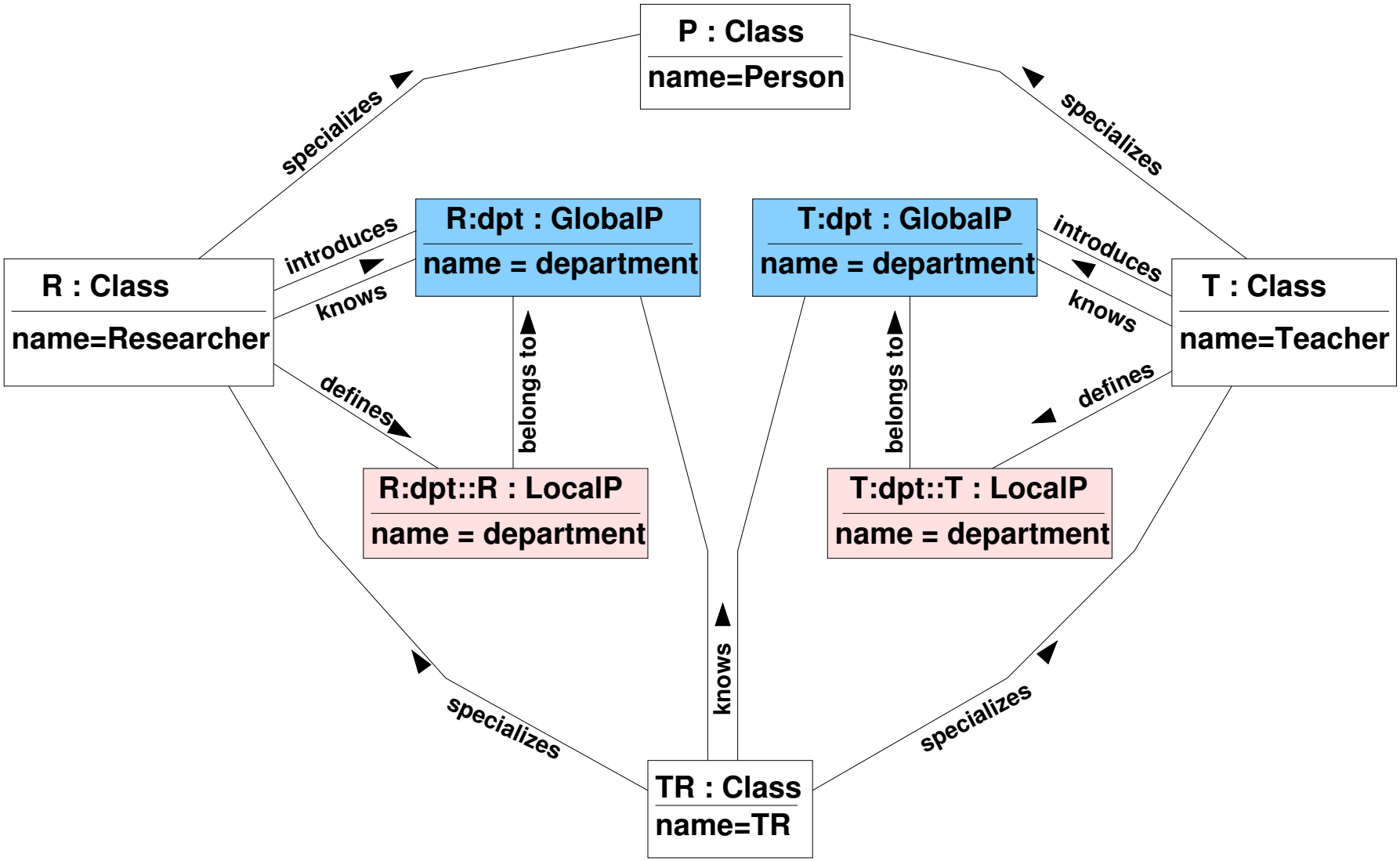

```
Teacher-Researcher z;

Researcher x=z;
x.department;      //  OK

Teacher y=z;
y.department       //  OK

z.department       //  KO
```


P : Class
name=Person


P : Class
name=Person
specializes
R:dpt : GlobalP
name = department
introduces
knows
R : Class
name=Researcher
belongs to
defines
R:dpt::R : LocalP
name = department


P : Class
name=Person
specializes
T:dpt : GlobalP
name = department
introduces
knows
T : Class
name=Teacher
belongs to
defines
T:dpt::T : LocalP
name = department


P : Class
name=Person
specializes
specializes
R : Class
name=Researcher
introduces
knows
defines
R:dpt : GlobalP
name = department
belongs to
R:dpt::R : LocalP
name = department
T:dpt : GlobalP
name = department
introduces
knows
defines
belongs to
T : Class
name=Teacher
T:dpt::T : LocalP
name = department


P : Class
name=Person
specializes
specializes
R : Class
name=Researcher
introduces
knows
defines
R:dpt : GlobalP
name = department
belongs to
R:dpt::R : LocalP
name = department
T:dpt : GlobalP
name = department
introduces
knows
defines
belongs to
T : Class
name=Teacher
T:dpt::T : LocalP
name = department
specializes
knows
specializes
TR : Class
name=TR

Multiple inheritance conflicts (3/5)

Diagnosis

- conflict between two global properties with the same name

Solution: Fully Qualified Names

- short names are used in most situations
- names qualified with the introduction class
  used when a conflict occurs

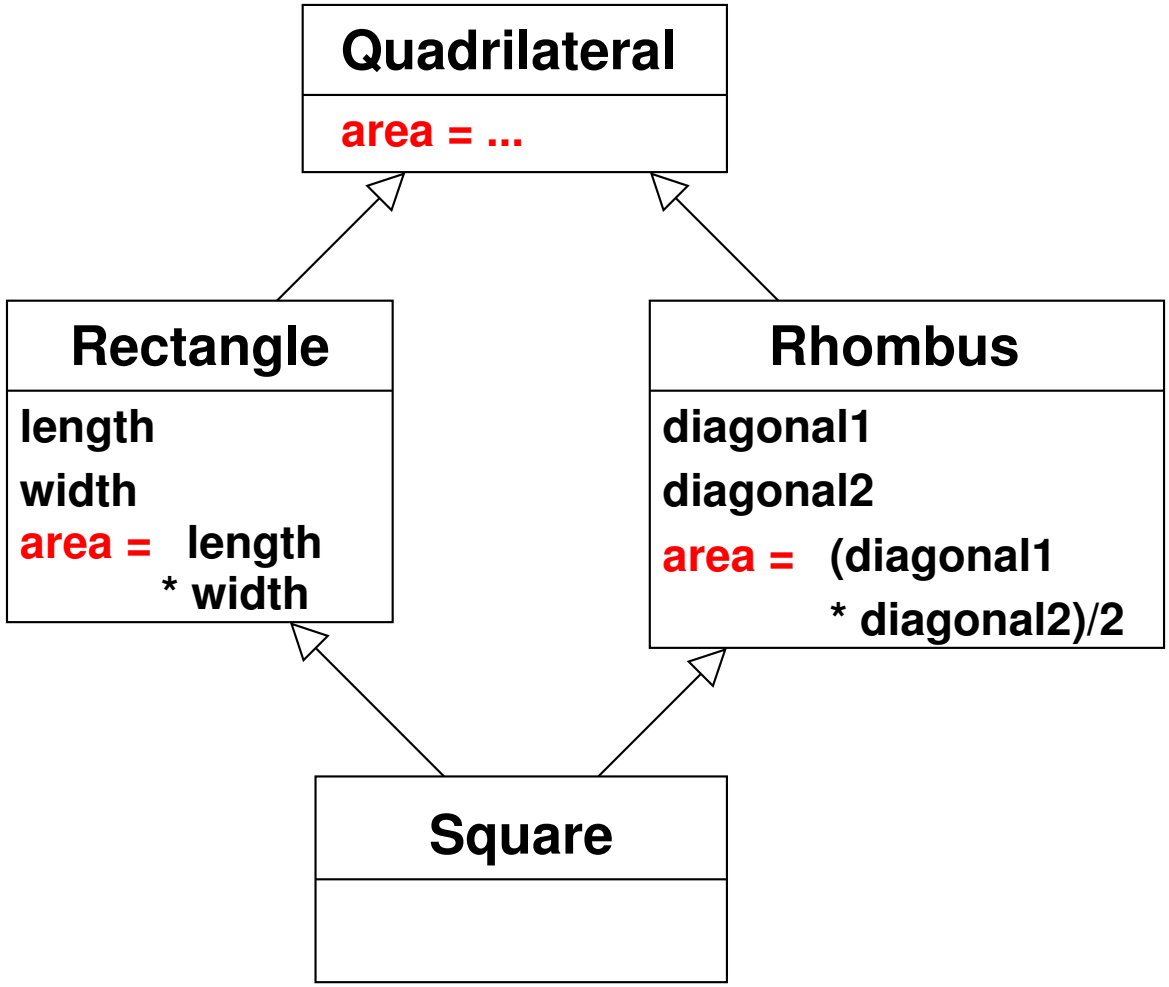
☛ a global property is introduced by a single class
static typing required

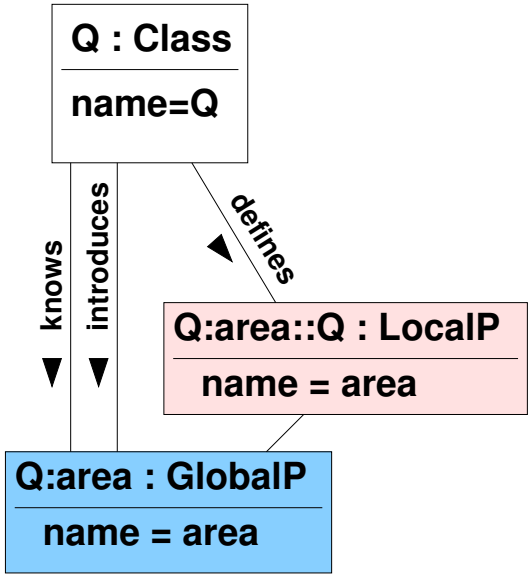
short names

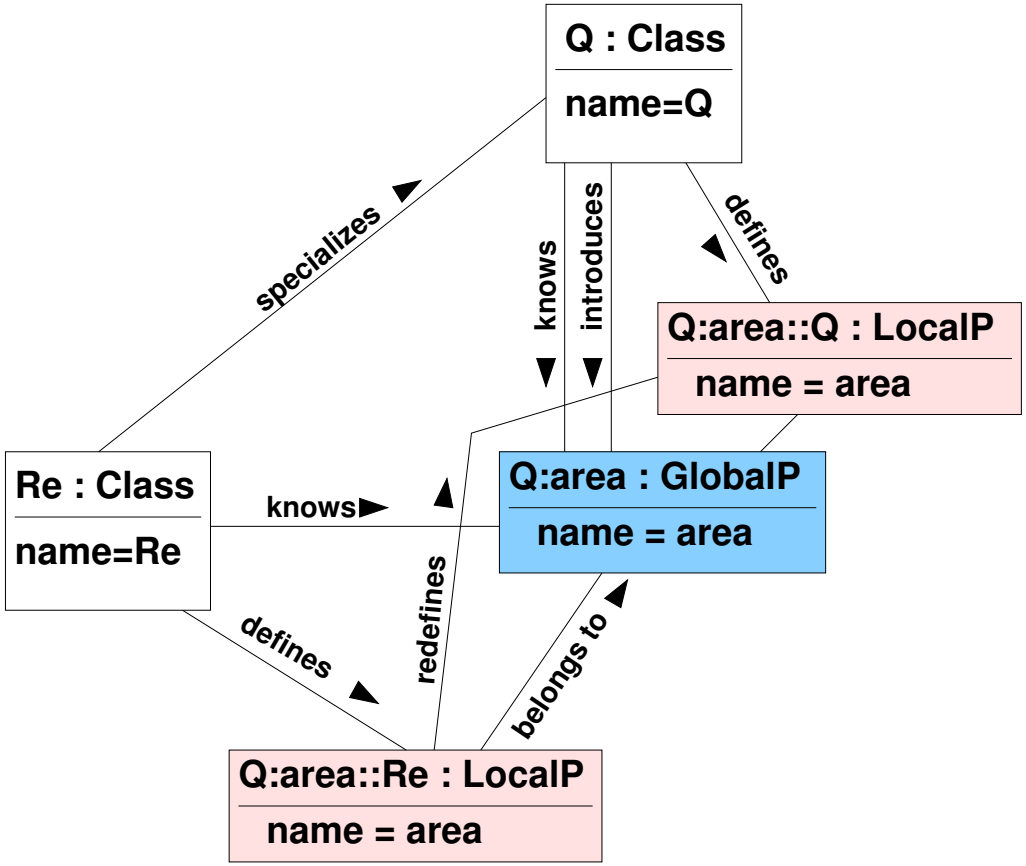
```
Teacher-Researcher z;

Researcher x=z;
x.department;      //   OK


Teacher y=z;
y.department       //   OK


z.department       //   KO
```

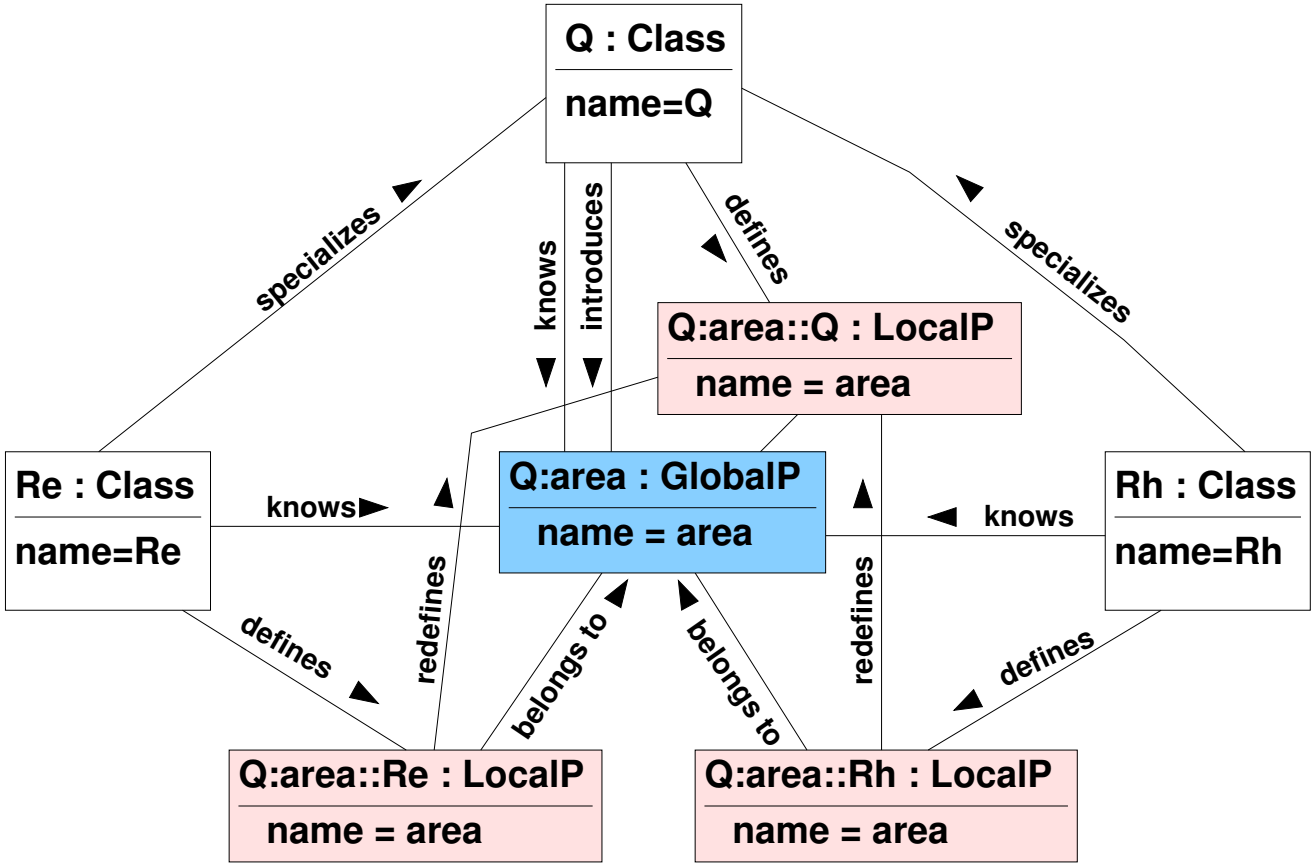
short names	fully qualified names
Teacher-Researcher z;	
Researcher x=z;	
x.department; // OK	means x.Researcher:department
Teacher y=z;	
y.department // OK	means y.Teacher:department
z.department // KO	must be disambiguated with z.Teacher:department or z.Researcher:department the programmer should know!

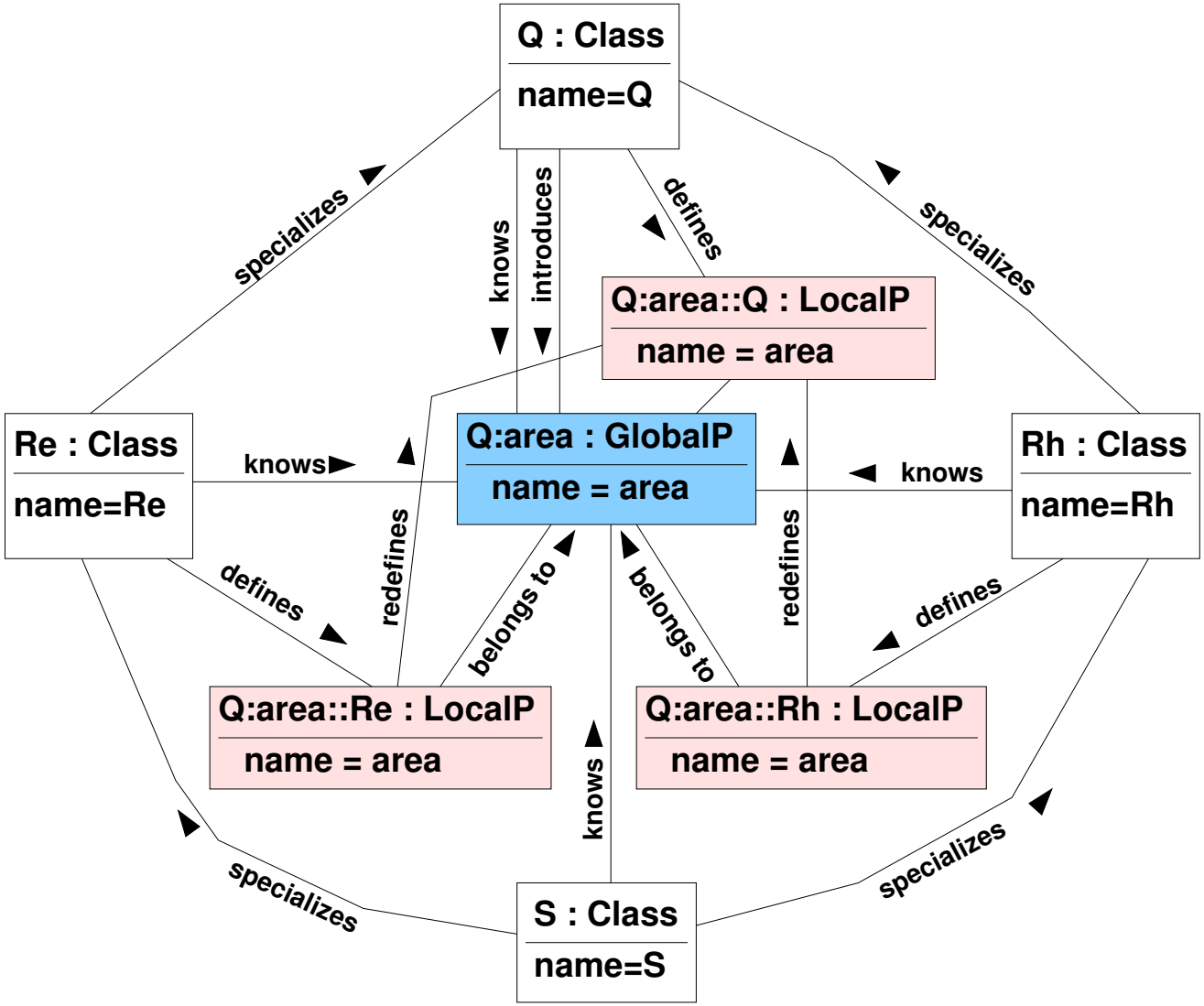
Multiple inheritance conflicts (4/5)




Q : Class
name=Q
knows
introduces
defines
Q:area::Q : LocalP
name = area
Q:area : GlobalP
name = area


Q : Class
name=Q
specializes
knows
introduces
defines
Q:area::Q : LocalP
name = area
Re : Class
name=Re
knows
Q:area : GlobalP
name = area
defines
redefines
belongs to
Q:area::Re : LocalP
name = area


Q : Class
name=Q
specializes
knows
introduces
defines
specializes
Q:area::Q : LocalP
name = area
Re : Class
name=Re
knows
Q:area : GlobalP
name = area
knows
Rh : Class
name=Rh
defines
redefines
belongs to
belongs to
redefines
defines
Q:area::Re : LocalP
name = area
Q:area::Rh : LocalP
name = area


Q : Class
name=Q
specializes
knows
introduces
defines
specializes
Q:area::Q : LocalP
name = area
Re : Class
name=Re
knows
Q:area : GlobalP
name = area
knows
Rh : Class
name=Rh
defines
redefines
belongs to
belongs to
redefines
defines
Q:area::Re : LocalP
name = area
Q:area::Rh : LocalP
name = area
knows
specializes
specializes
S : Class
name=S

Multiple inheritance conflicts (5/5)

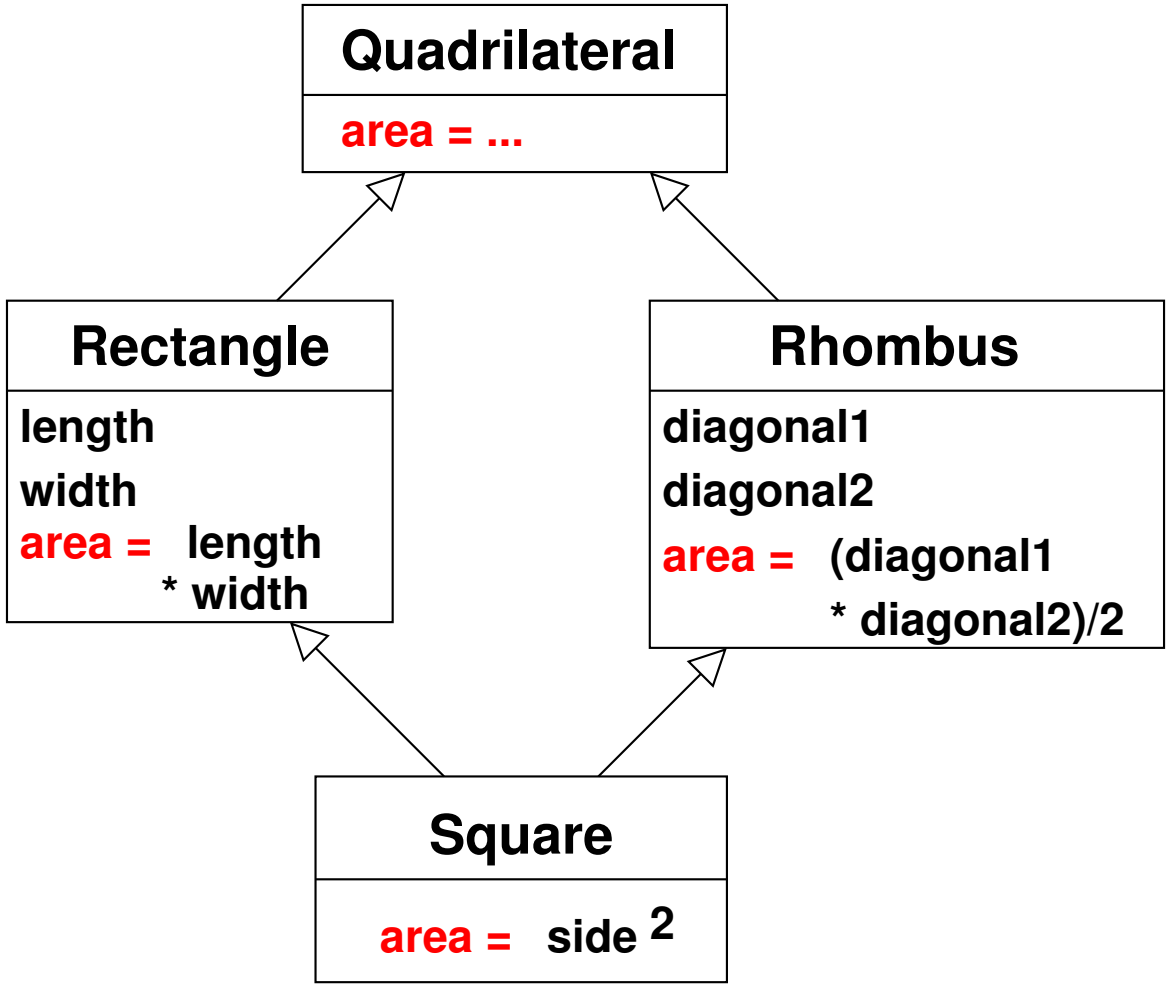
Diagnosis

- conflict between two local properties of the same global property
- none more specific than the other

Solution

☛ redefinition in the class where the conflict occurs

Multiple inheritance conflicts (5/5)



Monotonicity vs redefinition

Aristotelian logic is monotonic

- a Human must behave like a Mortal

Redefinition is non-monotonic

- redefining a method yields non-monotonicity

Method combination = Call to super

- a way to recover monotonicity
- a Human behaves like a Mortal, with extra behaviour

Monotonicity vs redefinition

Aristotelian logic is monotonic

- a Human must behave like a Mortal

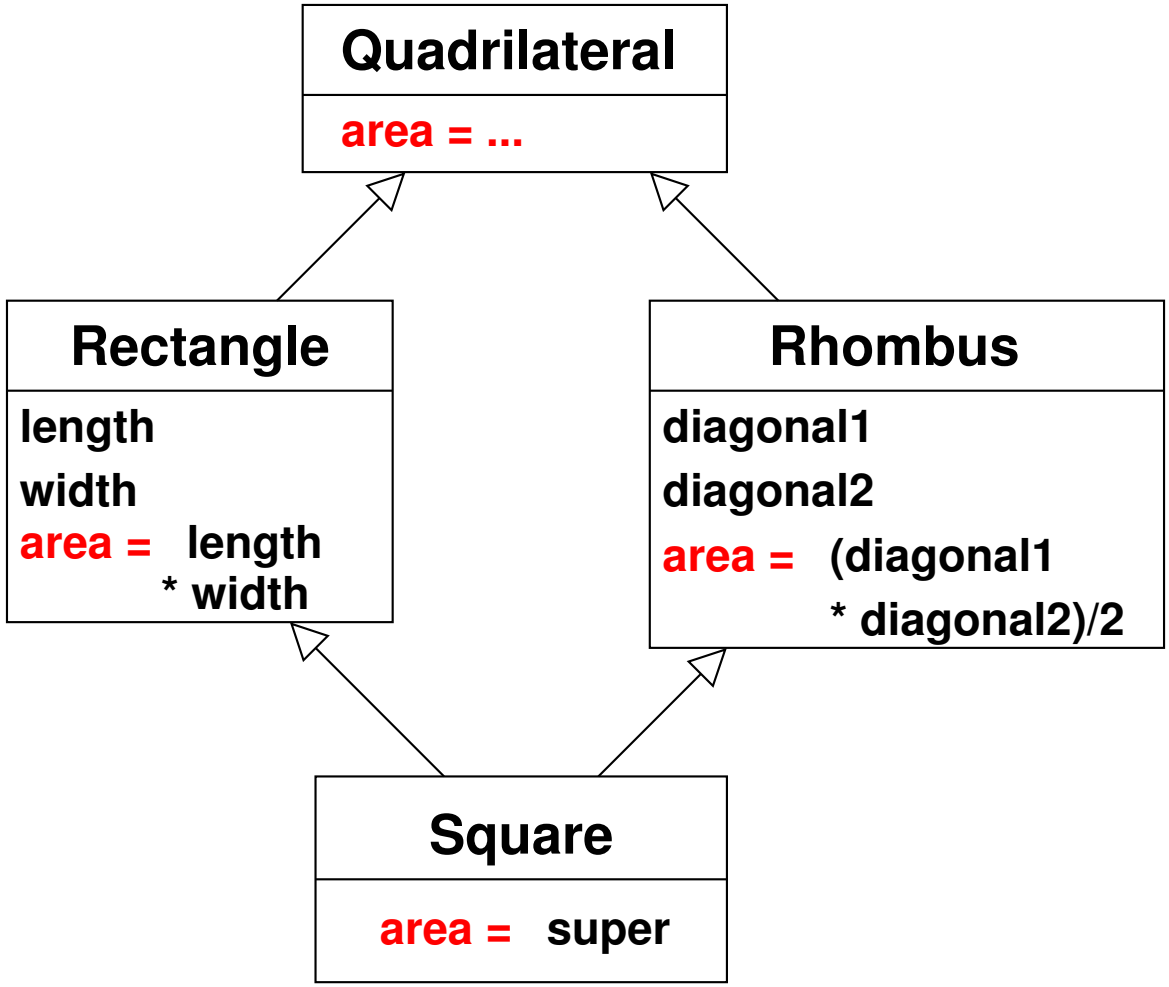
Redefinition is non-monotonic

- redefining a method yields non-monotonicity

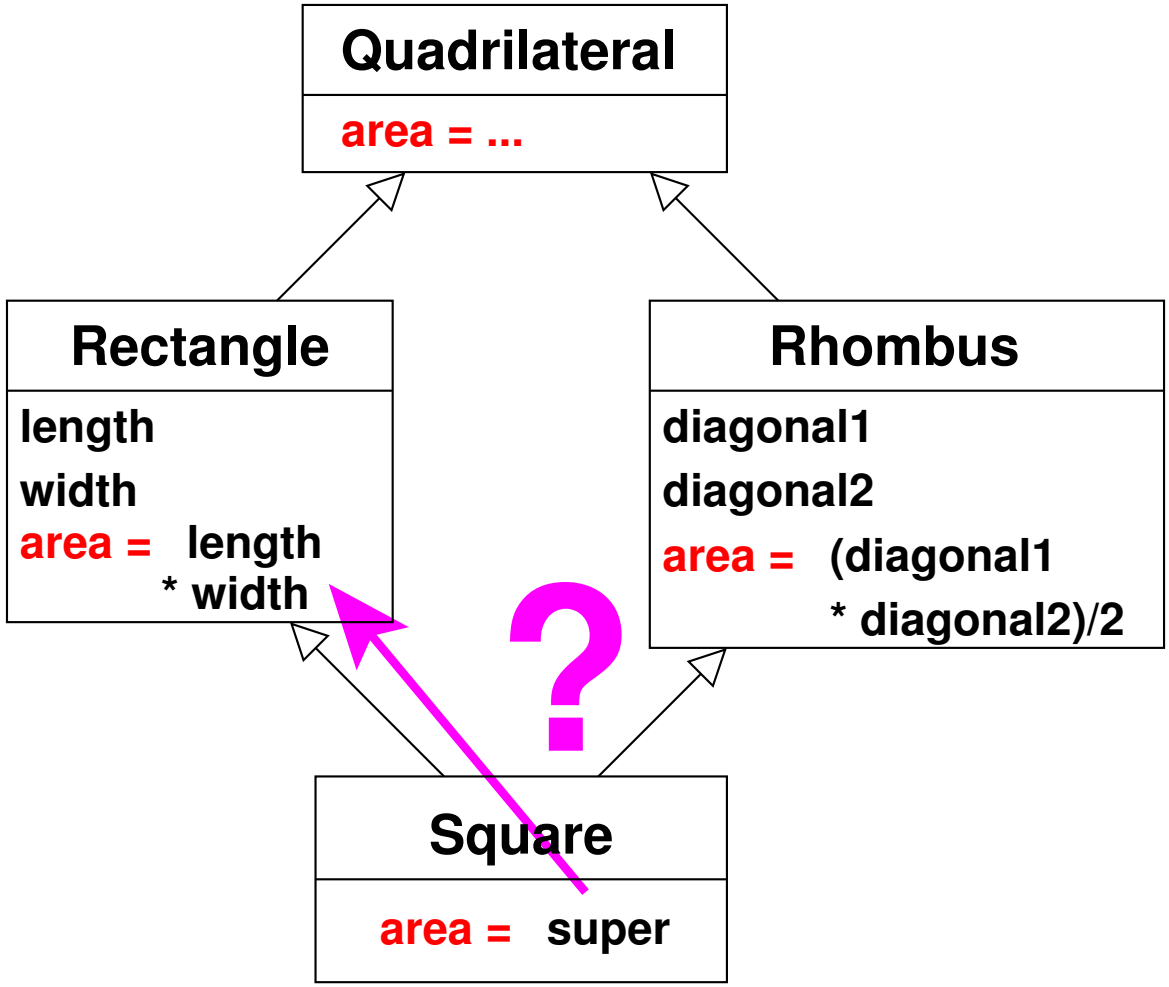
Method combination = Call to super

- a way to recover monotonicity
- a Human behaves like a Mortal, with extra behaviour

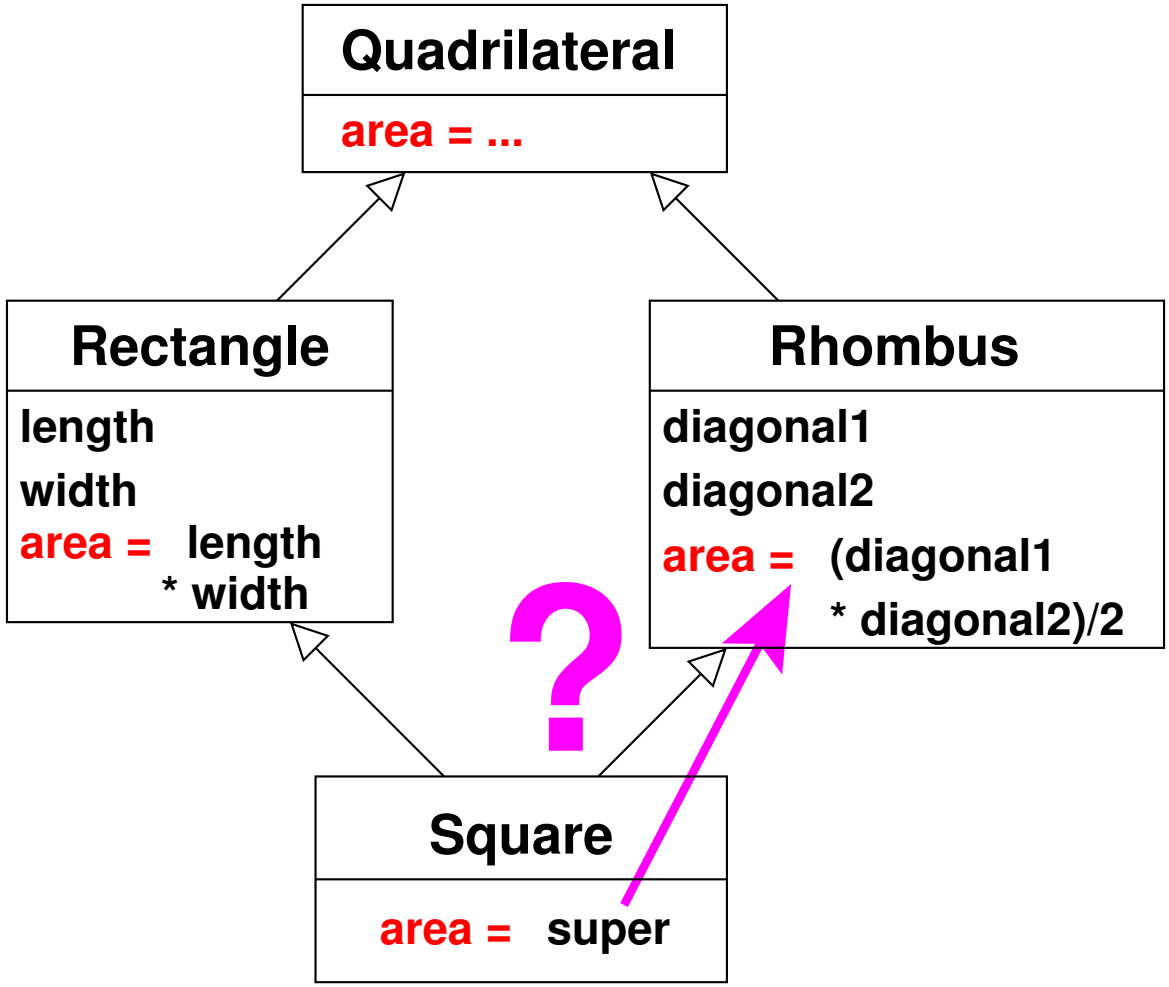
Method combination conflicts



Method combination conflicts

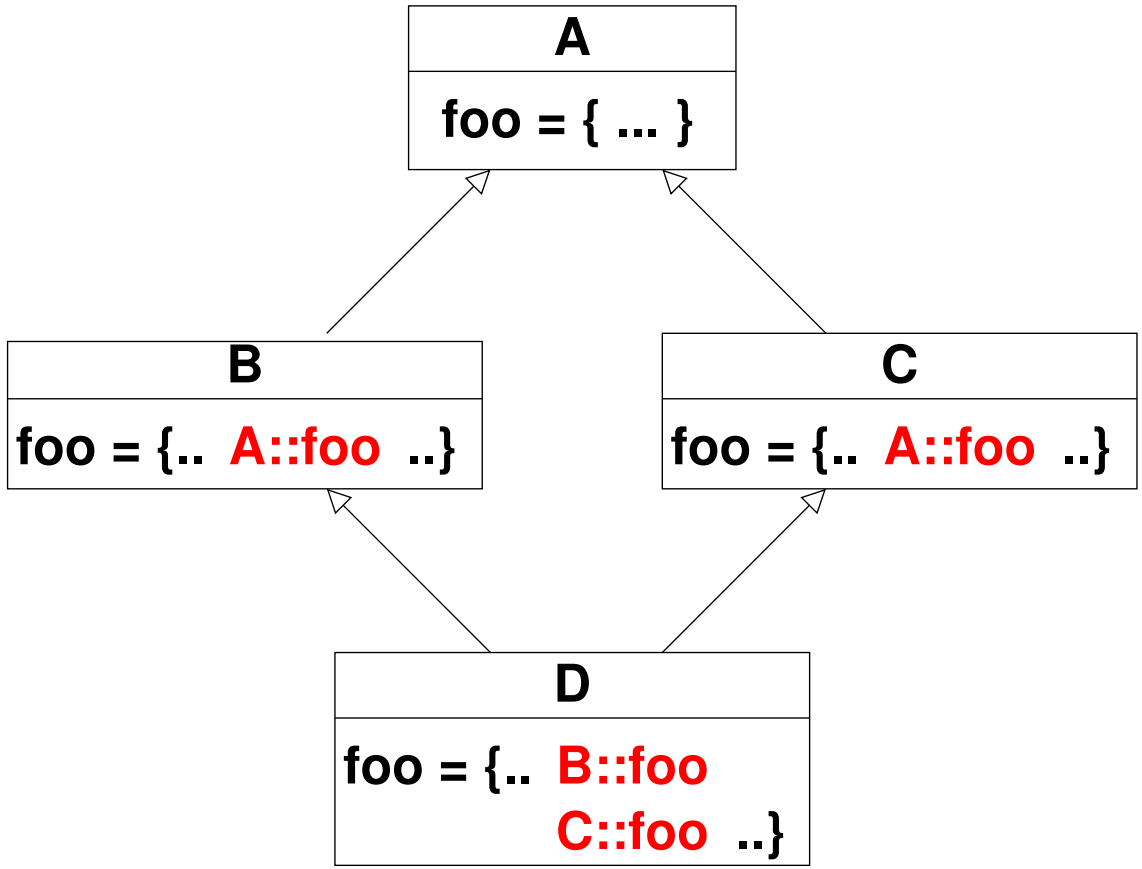


Method combination conflicts

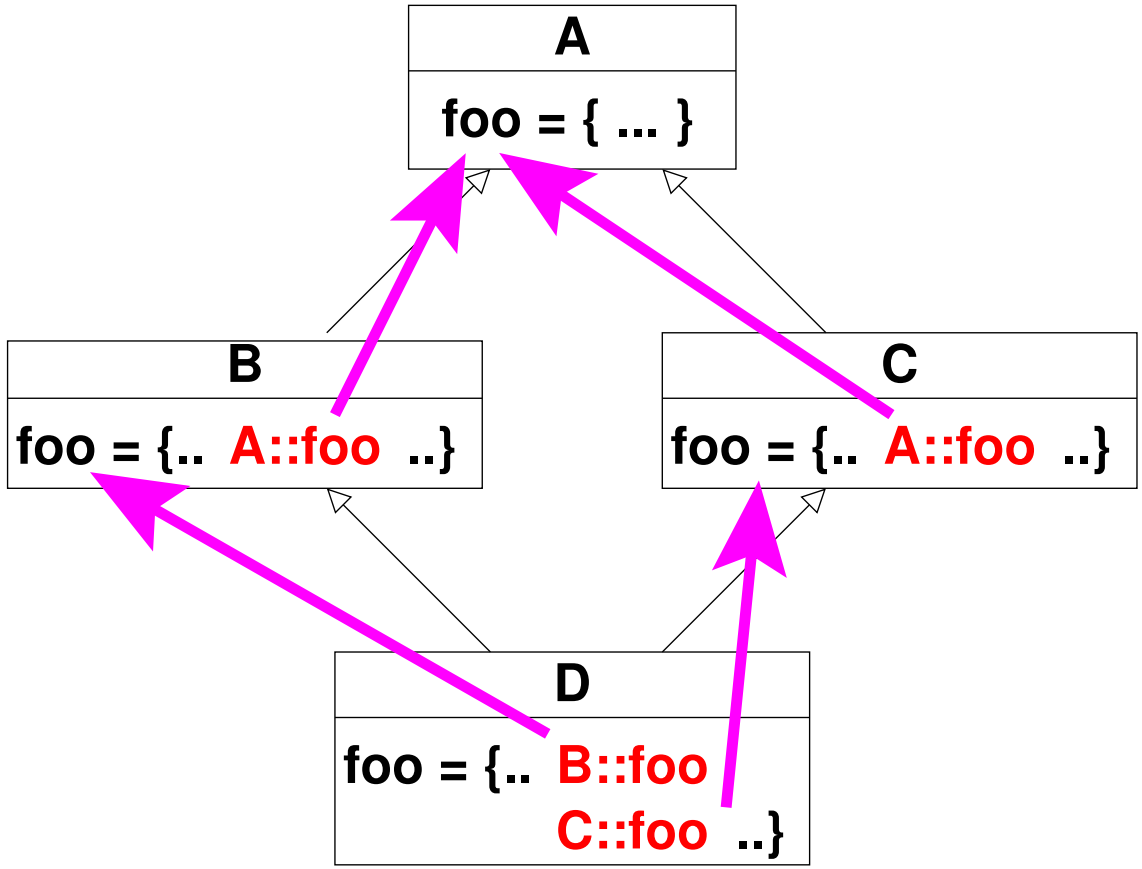


A wrong solution: static super calls

- like C++ or Eiffel

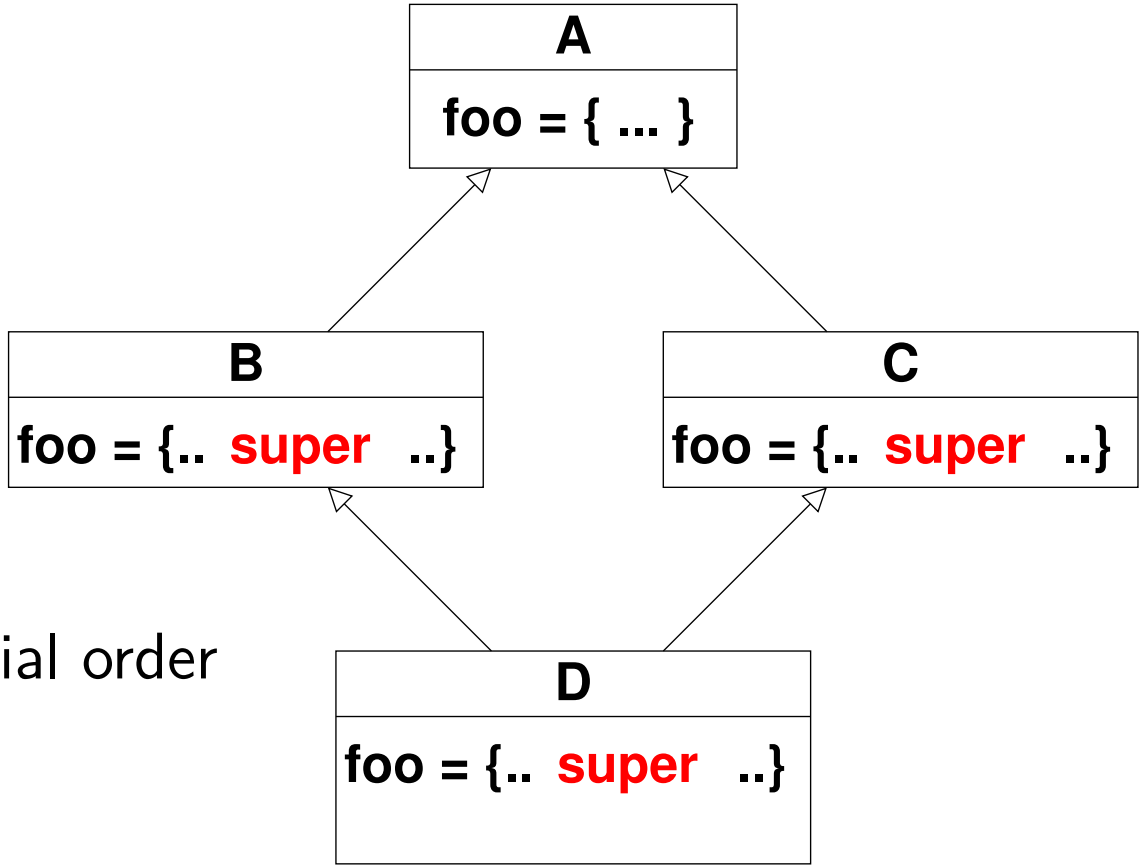


A wrong solution: static super calls



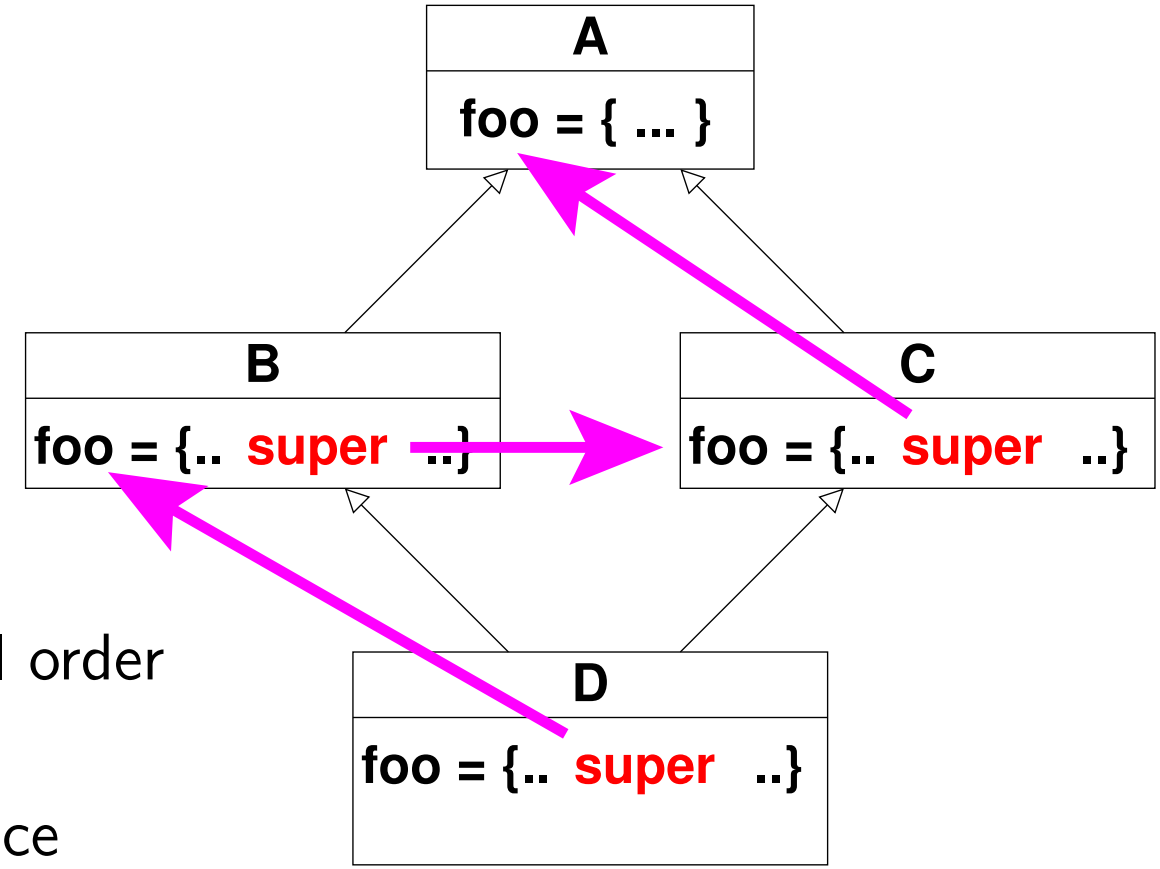
- like C++ or Eiffel
- ☛ A::foo evaluated twice
  repeated inheritance

The right solution: linearization (1/2)



- Specialization = a partial order

The right solution: linearization (1/2)



- Linearization = a total order
- ☛ A::foo evaluated once
  non-repeated inheritance

The right solution: linearization (2/2)

Principle

- linear extension of the specialization partial order
- monotonic
- ☛ order preserved by specialization
- an algorithm called C3 (used in Python)

With restricted multiple inheritance (1/3)

Multiple subtyping (Java, C♯)

- no problem with local property conflicts, nor method combination
- ☛ except with default methods in Java 8
- ad hoc solution for global property conflicts in C♯
- no solution for global property conflicts in Java

Multiple subtyping could be well-specified

With restricted multiple inheritance (1/3)

Multiple subtyping (Java, C♯)

- no problem with local property conflicts, nor method combination
- ☛ except with default methods in Java 8
- ad hoc solution for global property conflicts in C♯
- no solution for global property conflicts in Java

Multiple subtyping could be well-specified

With restricted multiple inheritance (2/3)

Mixins/Traits

- same problems as with full multiple inheritance
  - global property conflicts
  - method combination

☛ if mixins were the answer, what was the question?

☛ just add unnecessary asymmetry between classes and traits

With restricted multiple inheritance (3/3)

The Scala case

- no solution for global property conflicts (like in Java)
- linearization-based method combination, but not C3

The question

How to easily implement "multiple inheritance" in multiple-subtyping runtime systems?

With restricted multiple inheritance (3/3)

The Scala case

- no solution for global property conflicts (like in Java)
- linearization-based method combination, but not C3

The question

How to easily implement "multiple inheritance" in multiple-subtyping runtime systems?

With full multiple inheritance (1/2)

C++ virtual inheritance

- right solution for global attribute conflicts
- no solution for global method conflicts

☛ two distinct attributes for a single accessor!

- method combination with static calls and repeated inheritance
- linearization used for constructor/destructor combination but not C3

C++ non-virtual inheritance

- repeated inheritance for non-conflicting global attributes

☛ an abomination

With full multiple inheritance (1/2)

C++ virtual inheritance

- right solution for global attribute conflicts
- no solution for global method conflicts

☛ two distinct attributes for a single accessor!

- method combination with static calls and repeated inheritance
- linearization used for constructor/destructor combination but not C3

C++ non-virtual inheritance

- repeated inheritance for non-conflicting global attributes

☛ an abomination

With full multiple inheritance (2/2)

Eiffel

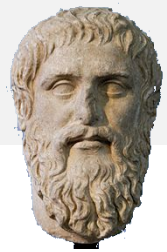
- ad hoc solution for global property conflicts, with renaming
- method combination with static calls and repeated inheritance

With dynamic typing (CLOS, Python)

- no solution for global property conflicts
- linearization-based method combination
- C3 default linearization only in Python
- possibility to define metaclasses using C3 in CLOS

LIRMM

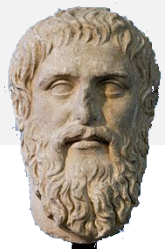
The ideal of multiple inheritance



- fully symmetric
  - no distinction between classes and traits
  - the same for methods, attributes, virtual types, type parameters
- metamodeling semantics
  - with fully qualified names for global properties
  - without any repetition
- local property conflicts solved by redefinition
- method combination using the C3 linearization

LIRMM

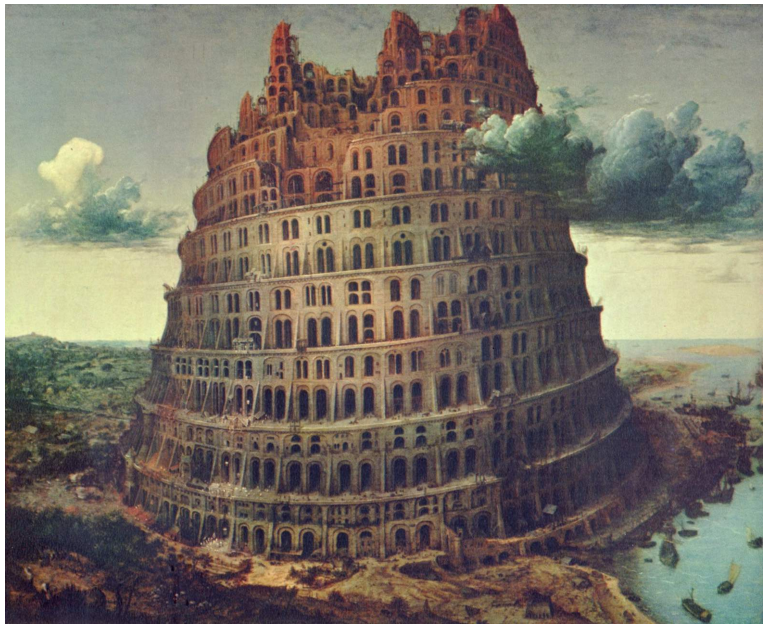
References on multiple inheritance



- with Jean Privat: Meta-Modeling Semantics of Multiple Inheritance *Science of Computing Programming*, 2011.
- with Michel Habib, Marianne Huchard and Marie-Laure Mugnier: Proposal for a monotonic multiple inheritance linearization. *In Proc. OOPSLA'94*. 1994.
- Monotonic conflict resolution mechanisms for inheritance. *In Proc. OOPSLA'92*, 1992

LIRMM

Plan



LIRMM

Classes vs types (1/2)

Different roles

- classes declare properties and create instances
- types serve as annotations in the code
- ☛ allow the compiler to ensure the code is type safe

Nominal vs structural types

- a nominal type is a symbol with explicit subtyping
- a structural type is a record of named signatures, with implicit subtyping

Classes vs types (2/2)

Mainstream position (C++, Java, C♯, Scala, Eiffel, ..)

Besides higher-order types:

- classes are nominal types
- subtyping is implied by class specialization

The OCAML exception

☛ types are purely structural

Specialization vs subtyping (1/5)

Subtyping is substitutability (B. Liskov)

Specialization implies subtyping

IFF redefinition satisfies the contravariance rule

- return type must be redefined covariantly
- parameter types must be redefined contravariantly

☛ type safe

Specialization vs subtyping (2/5)

Mainstream position (C++, Java, Scala, ...)

- return types are covariant
- parameter types are invariant

Exceptions

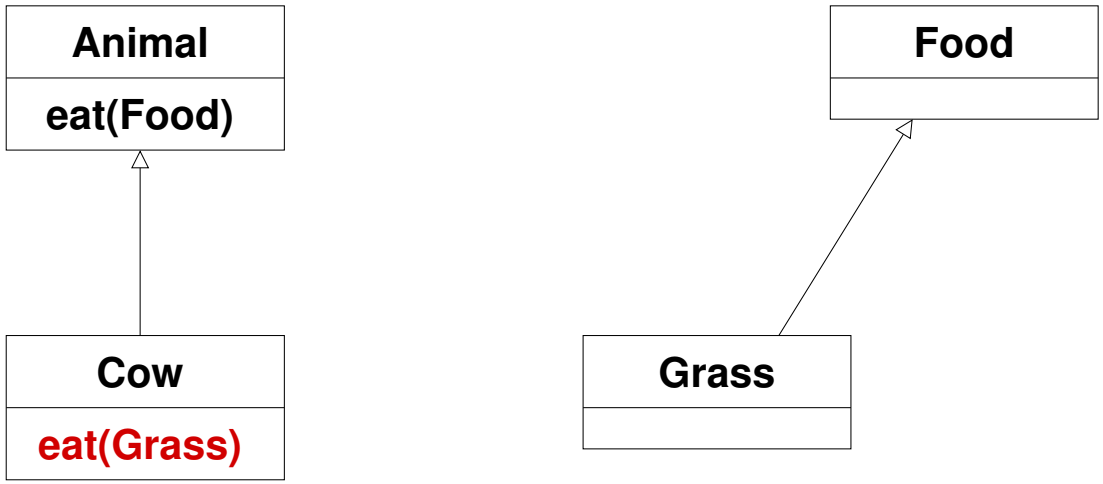
- C♯ & Java 1.4: return types are invariant
  ☛ no reason at all
- Eiffel: parameter types are covariant
  ☛ type unsafe
- OCAML: parameter types are contravariant

Specialization vs subtyping (3/5)

Motivations of the choice

- covariance because of the mad cow example
- invariance because
  - contravariance is useless in practice
  - static overloading was preexisting object-orientation
- contravariance because of structural types

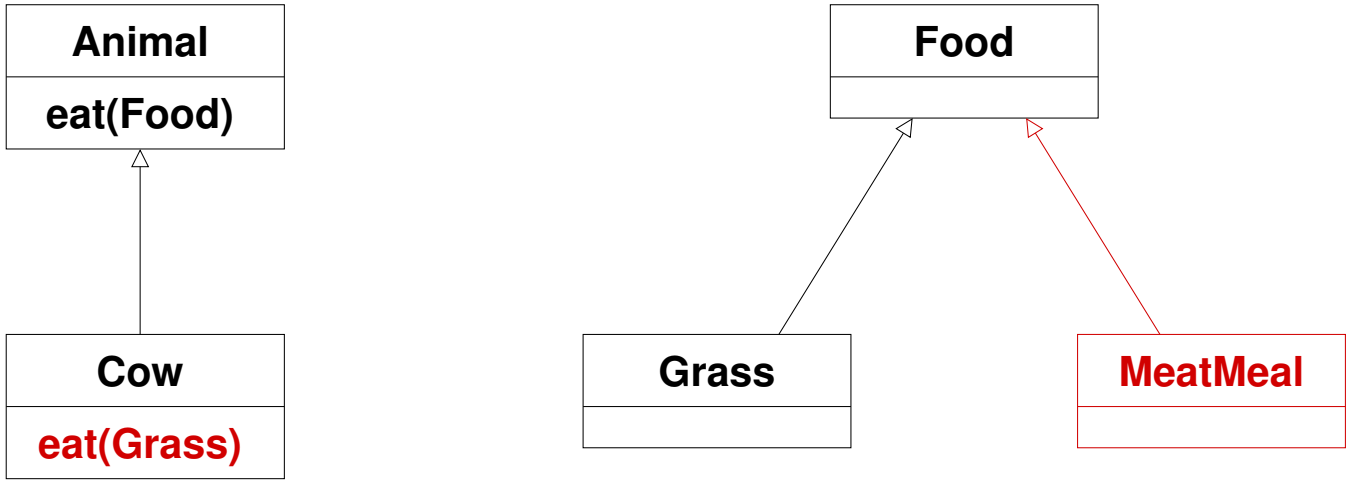
Specialization vs subtyping (4/5)




```
Animal x ;
Food y ;
x.eat(y);
```

☛ don't panic, but type errors are in the real world™

Specialization vs subtyping (4/5)




```
Animal x = new Cow();
Food y = new MeatMeal();
x.eat(y); // runtime type error
```

☛ don't panic, but type errors are in the real world™

Specialization vs subtyping (5/5)

What is my ideal?

- invariance of both parameter and return types
- covariance through virtual types

Static overloading (1/3)

When parameter types are invariant,

☛ there is room for static overloading

Principle

- a name denoting different entities in a common context
- disambiguated with static types
- originates in pre-object languages like PL/1 and C

Static overloading (1/3)

When parameter types are invariant,

☛ there is room for static overloading

Principle

- a name denoting different entities in a common context
- disambiguated with static types
- originates in pre-object languages like PL/1 and C

Mal nommer les choses,
c'est ajouter au malheur du monde

Misnaming things adds to the world's misfortunes

(Albert Camus)

Albert Camus a French writer and philosopher (1913-60)

Peyo a Belgian author of comic strips (1928-92)
creator of the Schtroumpfs

Schtroumpfs small characters whose language
has a single noun and verb: "schtroumpf"

Mal nommer les choses,
c'est ajouter au malheur du monde

Misnaming things adds to the world's misfortunes

(Albert Camus)

Albert Camus a French writer and philosopher (1913-60)

Peyo a Belgian author of comic strips (1928-92)
creator of the Schtroumpfs

Schtroumpfs small characters whose language
has a single noun and verb: "schtroumpf"

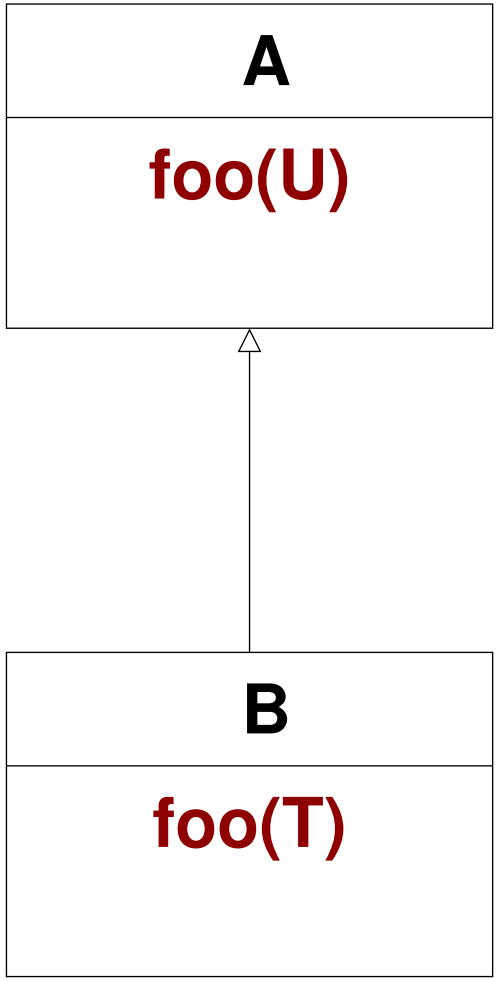
Misnaming things adds
to the world's misfortunes

in the context of a picture, even small children can understand!

Schtroumpfing schtroumpfs schtroumpfs
to the schtroumpf's schtroumpfs

in the context of a picture, even small children can understand!

Static overloading (2/3)

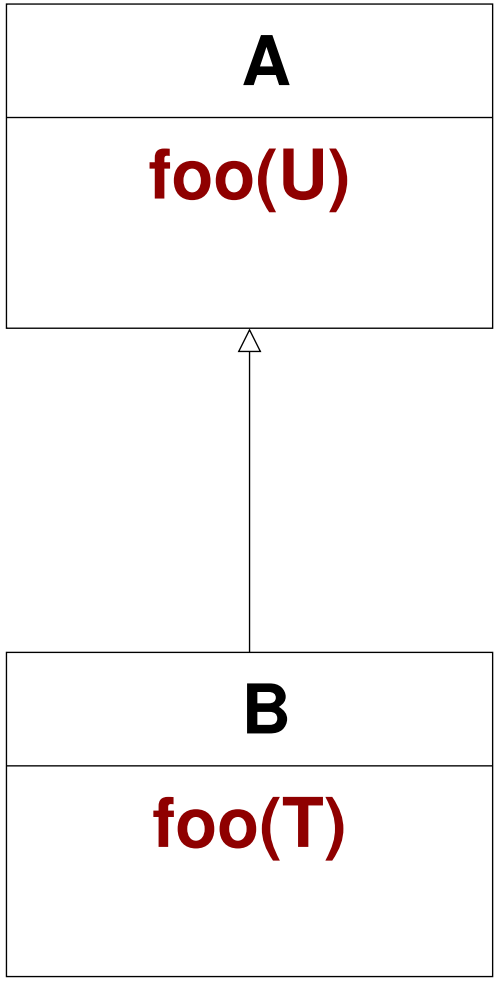


U <: T

B x
U y

	x.foo(y)	
C++		
Java 1.4		
Java 1.5		
Scala		
C♯		

Static overloading (2/3)



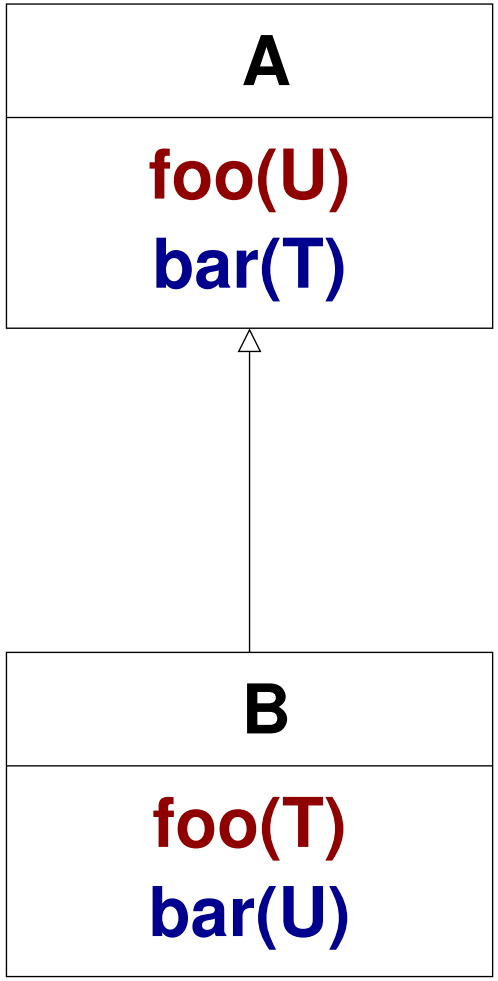
U <: **T**

B x
U y

	x.foo(y)
C++	foo(**T**)
Java 1.4	error
Java 1.5	foo(**U**)
Scala	error
C♯	foo(**T**)

LIRMM

Static overloading (2/3)



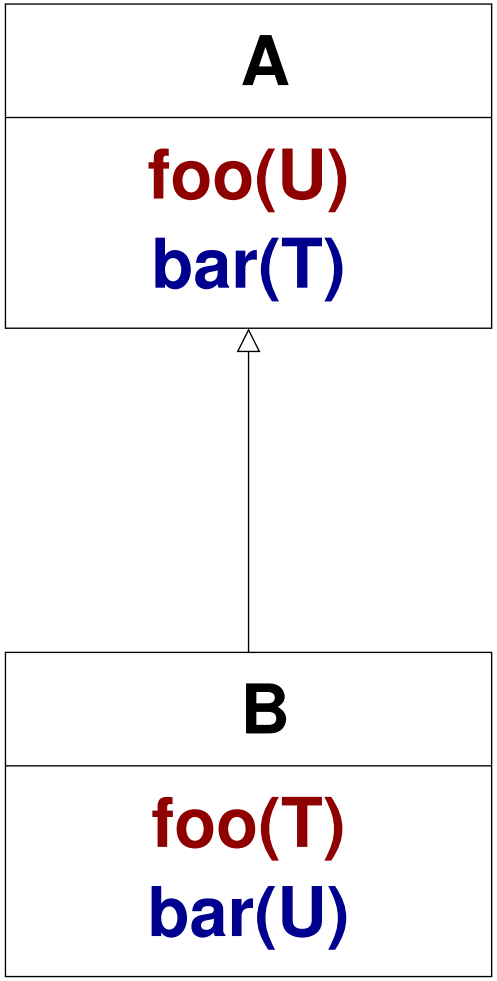
U <: **T**

B x
U y
T z = new U

	x.foo(y)	x.bar(z)
C++	foo(**T**)	error
Java 1.4	error	bar(**T**)
Java 1.5	foo(**U**)	bar(**T**)
Scala	error	bar(**T**)
C♯	foo(**T**)	bar(**T**)

LIRMM

Static overloading (2/3)



$U <: T$

```
B x
U y
T z = new U
```

Confusion with covariance contravariance or multiple selection

	x.foo(y)	x.bar(z)
Eiffel	error	error
OCAML	foo(**T**)	error
intuition	—	bar(**U**)

Static overloading (3/3)

6 languages

- 6 different specifications + the intuition

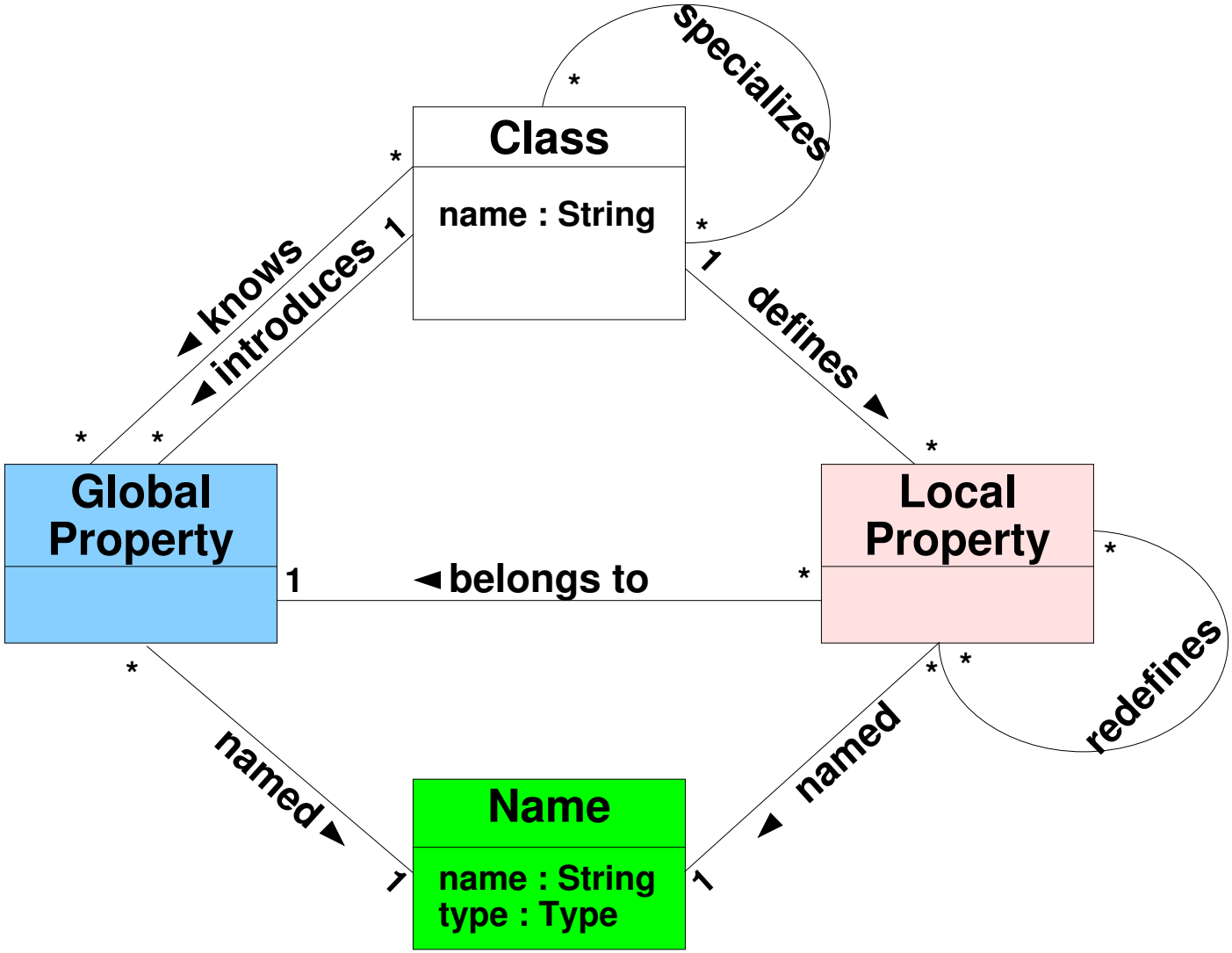
- it cannot be a sane language feature

Static overloading (3/3)

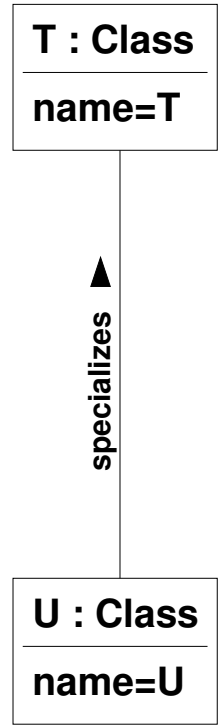
6 languages

☛ 6 different specifications + the intuition

☛ it cannot be a sane language feature


Class
name : String
specializes
*
*
knows
introduces
*
*
1
1
*
1
defines
*
Global Property
Local Property
1
belongs to
*
*
redefines
*
*
named
*
*
named
Name
name : String
type : Type
1
1

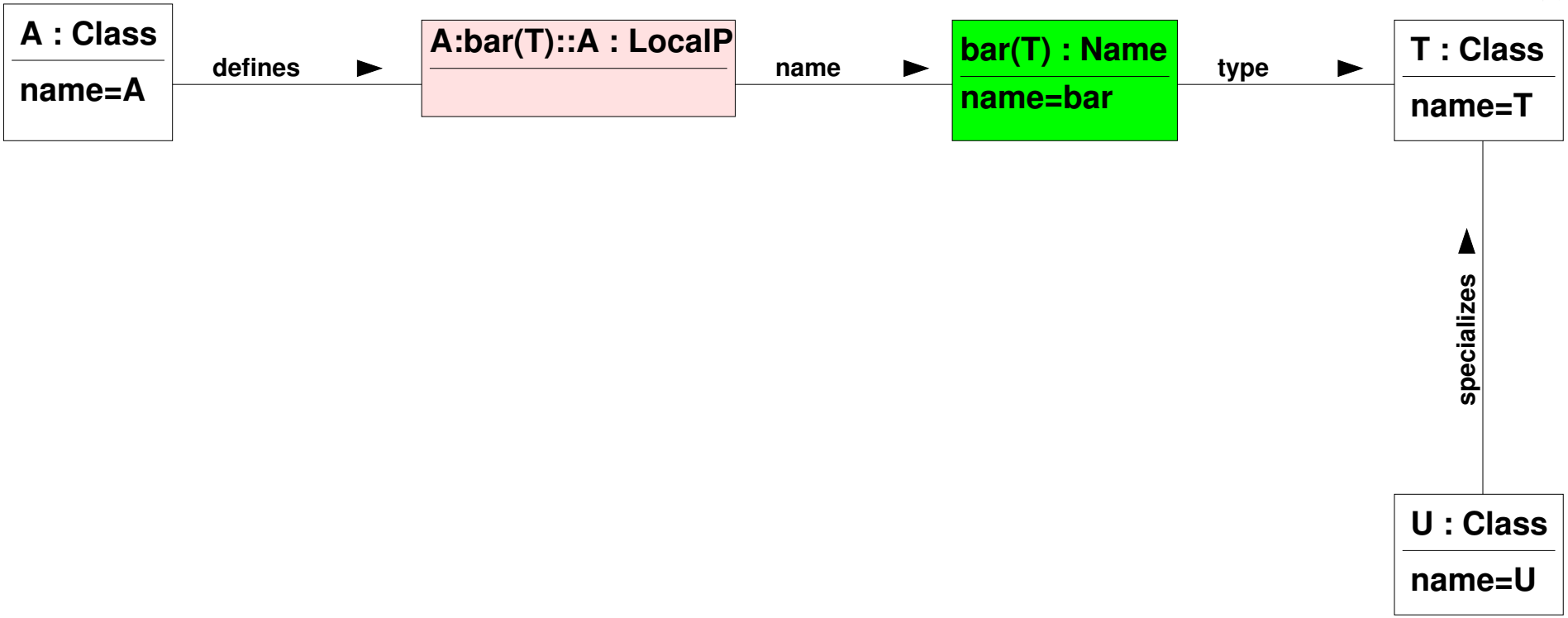
Static overloading in the metamodel



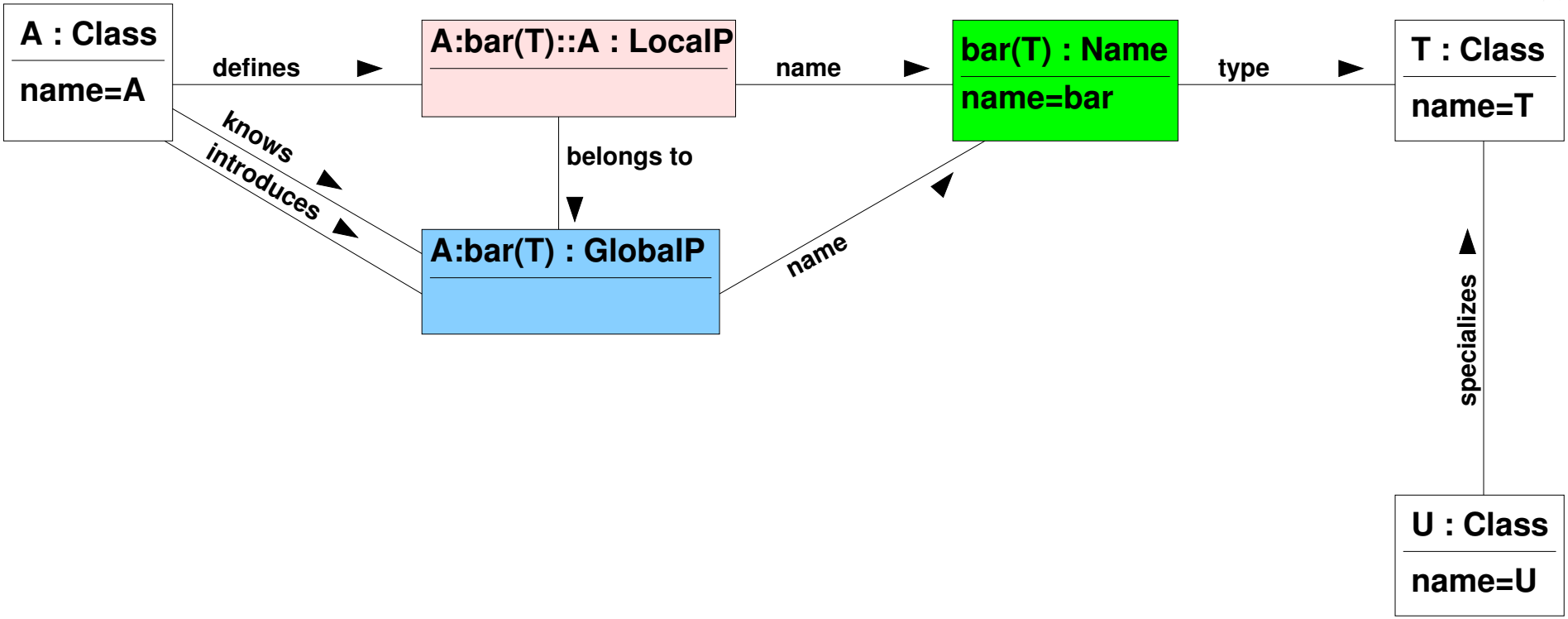
Static overloading in the metamodel

Static overloading in the metamodel

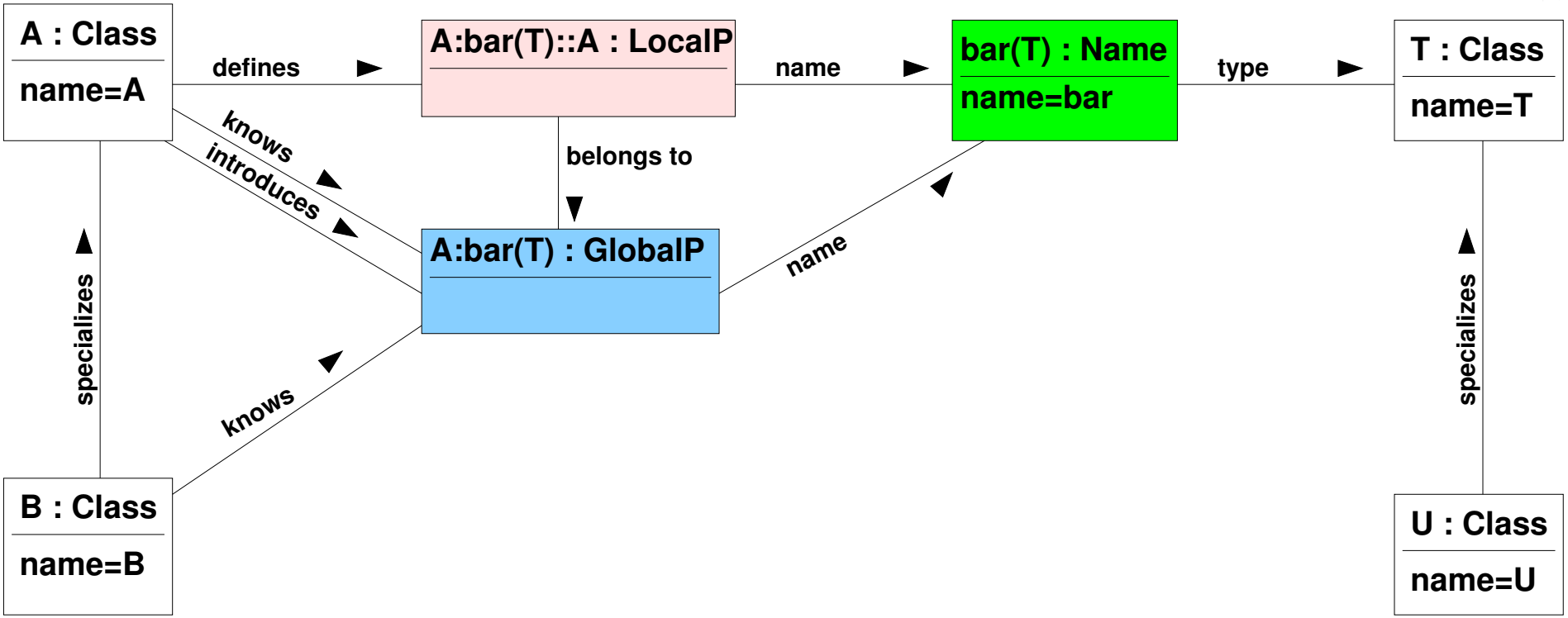
Static overloading in the metamodel



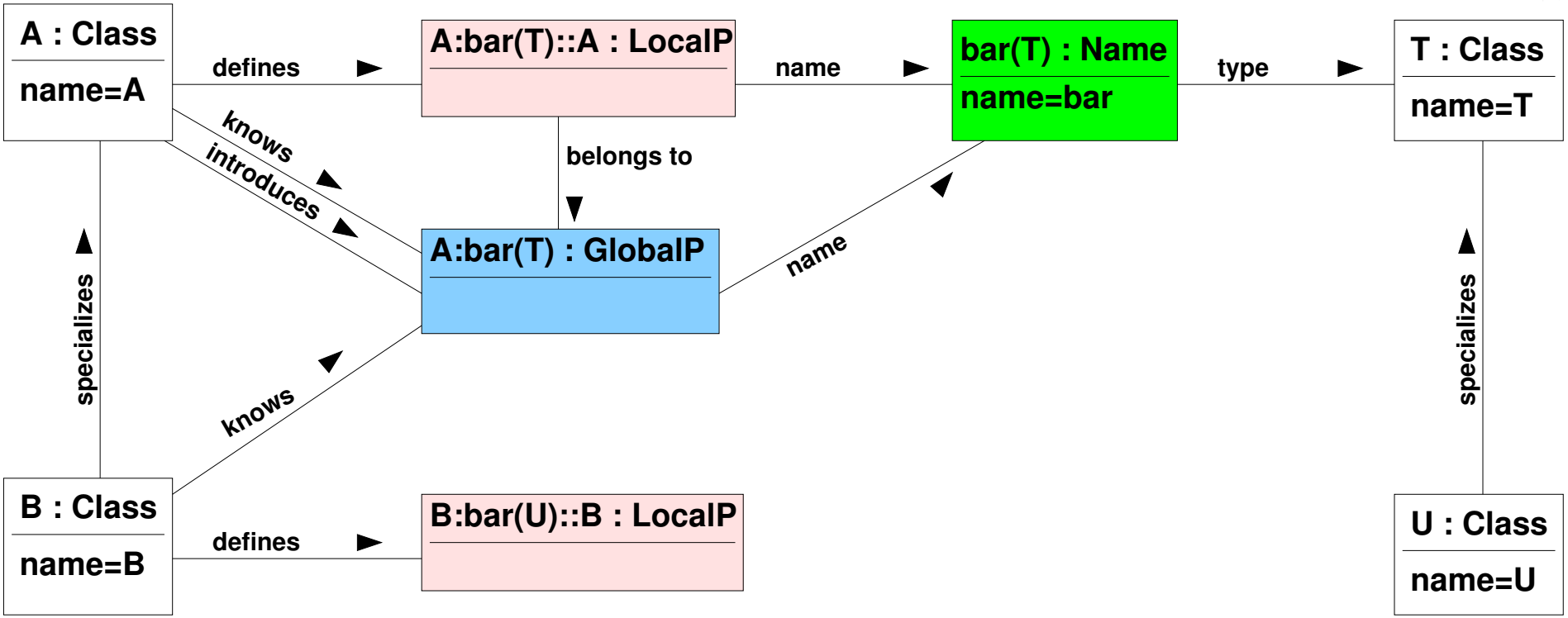
Static overloading in the metamodel



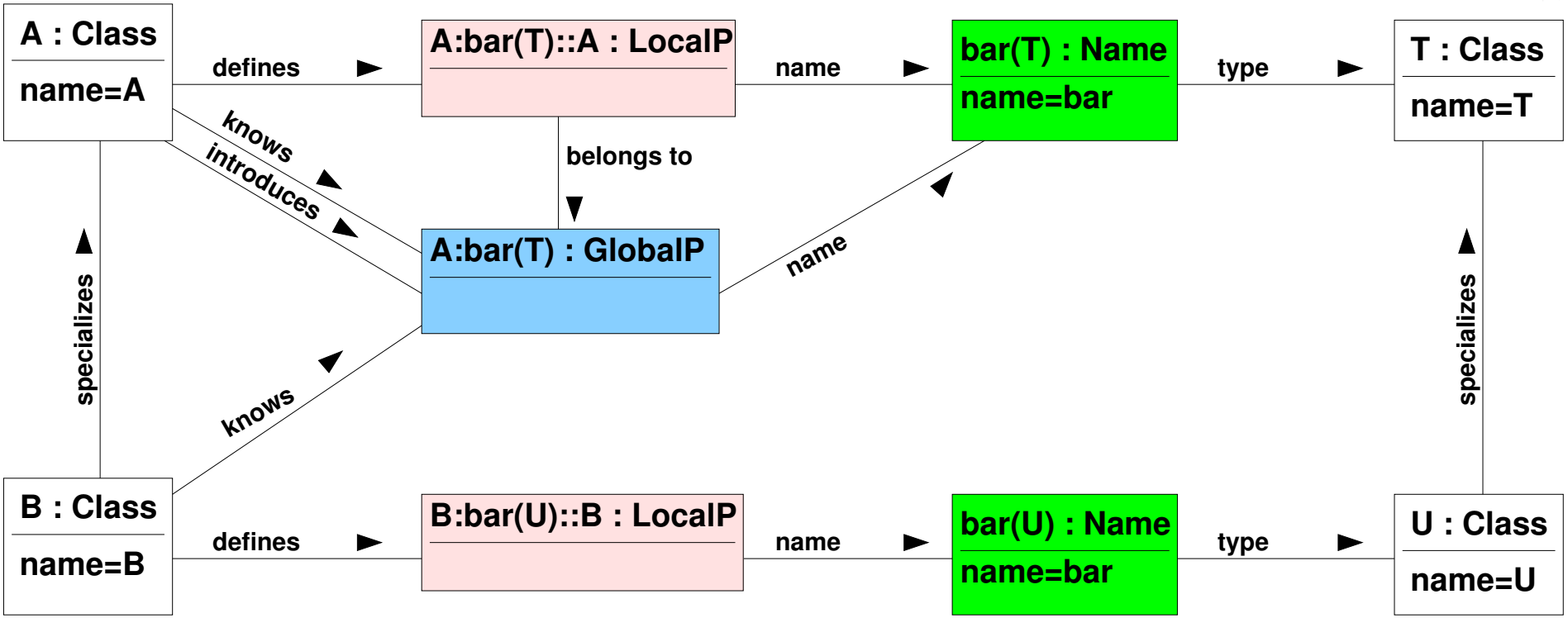
Static overloading in the metamodel



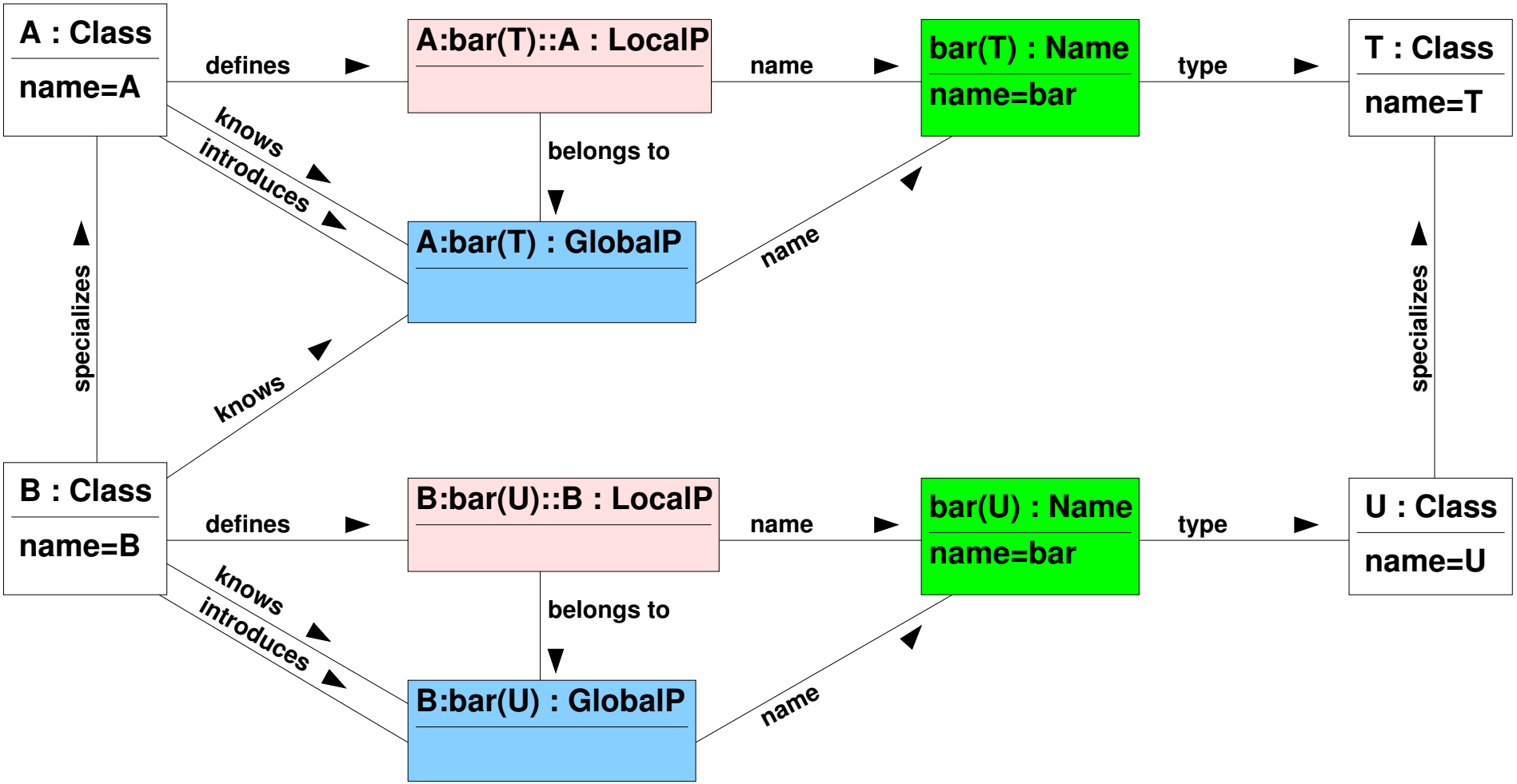
Static overloading in the metamodel



Static overloading in the metamodel



Static overloading in the metamodel



The right semantics (1/2)

At compile-type

1. select all the global methods, known by the receiver's static type, with parameter static types compatible with the call
   say foo(T) and foo(U)
2. select among them the single most specific
   if U <: T, foo(U) is more specific than foo(T)
3. compilation error when there are several most specific
   eg baz(T,U) and baz(U,T)

At run-type = late binding

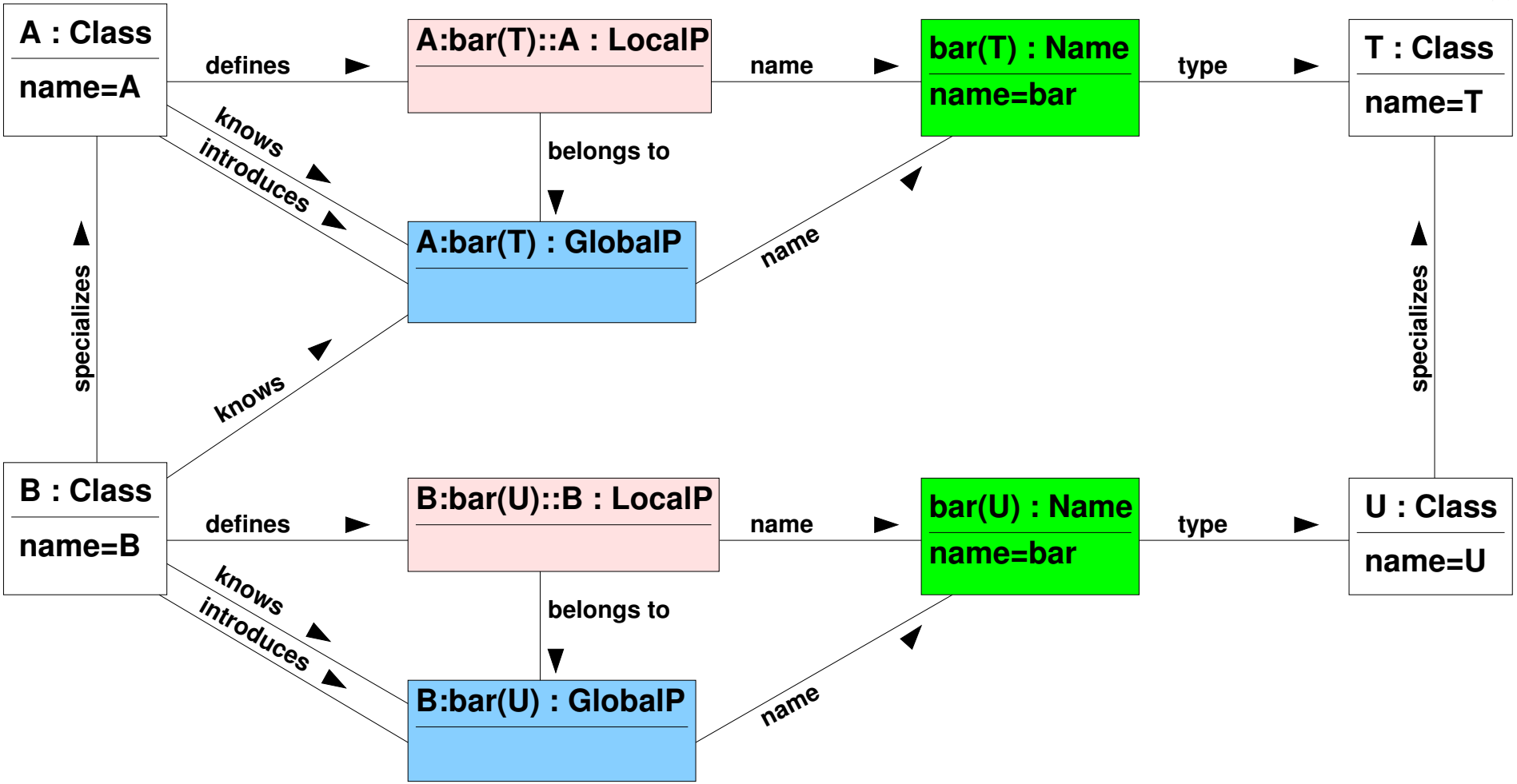
- select in the global property the most specific local property for the receiver's dynamic type

The right semantics (2/2)

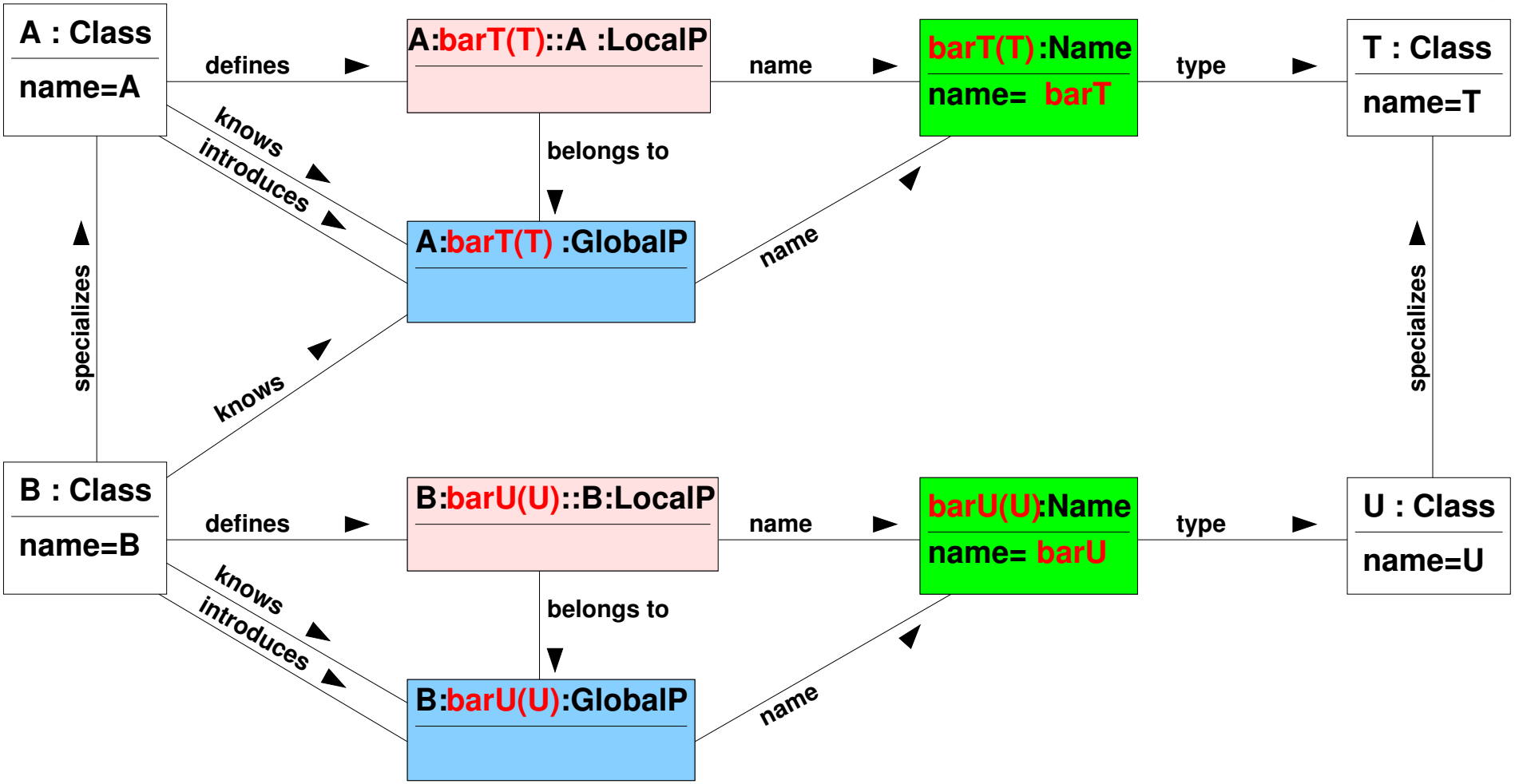
That of Java 1.5

- specificity does not involve the introduction or definition classes (as in Java 1.4 or Scala)
- a local property of a given global property doesn't mask a local property of another global property, as in C++ and C♯
  eg foo(T) in B doesn't mask foo(U) in A

Avoid overloading by renaming



Avoid overloading by renaming



The right specifications

- exclude overloading from the language specifications (as in Eiffel or Nit),
- otherwise, apply the right semantics (Java 1.5)
- but don't use it
- ☛ instead, rename!

The Schtroumpf project

Reductio ad absurdum

- select some mainstream language with static overloading
  - C++, Java, C♯, Scala, not Eiffel
- select some large-scale project written in this language
- rename all methods in the project classes as either foo or bar

☛ in case of conflict, add an extra, unused parameter

Variant

☛ develop an Eclipse plugin that does this renaming, automatically

Plan

Generic vs object-oriented programmin

Genericity is not object-oriented

- two almost orthogonal constructs

Genericity is now universal

In static typing

- object-oriented languages are now generic (Eiffel, C++, Java, C♯, ...)
- generic languages (Ada) are now object-oriented

LIRMM

Generic vs object-oriented programmin

Genericity is not object-oriented

- two almost orthogonal constructs

Genericity is now universal

In static typing

- object-oriented languages are now generic (Eiffel, C++, Java, C♯, ...)
- generic languages (Ada) are now object-oriented

Generic vs object-oriented programmin

Genericity + subtyping ⇒ troubles

- hard to specify
- hard to understand and use
- hard to implement efficiently

LIRMM

Specifications by implementation

At least 3 versions

heterogeneous pure textual substitution (C++)

- ☛ no recursive types + code explosion

homogeneous type erasure and code sharing (Java 1.5, Scala)

- ☛ limited expressivity, unsafe *casts*, inefficient boxing

mixed code shared or specialized, with runtime types (C♯)

- ☛ best tradeoff expressivity-efficiency-safety

LIRMM

Constrained genericity

At least 3 specifications of contraints

- none (C++)
  - ☛ no checking before instantiation
  - ☛ (recently) notion of concept
- formal type parameters bounded by subtyping
  - ☛ simple to understand and use
- recursive bound (*F-bounded*) (Java, C♯, Scala, Eiffel, ...)
  - ☛ powerful but harder to understand

  allows to clone isomorphic structures

Genericity and (co)variance

Principle

Cup<茶> is not a subtype of **Cup<Drink>**

But many unsafeties ...

- generalized covariance (Eiffel)
- covariance of arrays (Java, C♯)
- casts with type erasure (Java, Scala)

Safe variance annotations

- at definition-time (Scala, C♯ only for interfaces)
- at use-time (*wildcards* Java, Scala)

Genericity and (co)variance

Principle

Cup<茶> is not a subtype of **Cup<Drink>**

But many unsafeties ...

- generalized covariance (Eiffel)
- covariance of arrays (Java, C♯)
- casts with type erasure (Java, Scala)

Safe variance annotations

- at definition-time (Scala, C♯ only for interfaces)
- at use-time (*wildcards* Java, Scala)

Array covariance

Cats are not dogs

```
Cat[] x;
...
Animal[] y;
y = x;                  // dangerous but compiled
y[i] = new Dog();       // compiled but runtime exception
```

Type erasure = Alzheimer

Cats are not dogs (re)

```
Stack<Cat> x;
...
Stack<Dog> y;
y = (Stack<Dog>)x;      // stupid but compiled
y.push(new Dog);        // type is erased! Alzheimer
...
Cat z = x.pop();        // late exception
```

☛ bad traceability of errors

Type erasure = Alzheimer

Cats are not dogs (re)

```
Stack<Cat> x;
...
Stack<Dog> y;
y = (Stack<Dog>)x;      // stupid but compiled
y.push(new Dog);        // type is erased! Alzheimer
...
Cat z = x.pop();        // late exception
```

☛ bad traceability of errors

Variance

Variance positions

```
class Stack<T> {
    T pop () {..}          // covariant position
    push(T t) {..}         // contravariant position
}
```

- more complex rules when T is nested

Variance

-variance can be considered

- if T does not occur in a -variant position
- if such occurrences are excluded from the type interface

LIRMM

Variance

Variance positions

```
class Stack<T> {
    T pop () {..}          // covariant position
    push(T t) {..}         // contravariant position
}
```

- more complex rules when **T** is nested

Variance

Co-variance can be considered

- if **T** does not occur in a contra-variant position
- if such occurrences are excluded from the type interface

Variance

Variance positions

```
class Stack<T> {
    T pop () {..}          // covariant position
    push(T t) {..}         // contravariant position
}
```

- more complex rules when **T** is nested

Variance

Contra-variance can be considered

- if **T** does not occur in a co-variant position
- if such occurrences are excluded from the type interface

Variance annotations (1/3)

Use-site covariance

```
Stack<? extends Animal> s = new Stack<Cat>();
Animal a = s.pop();          // OK
s.push(a);                   // KO
```

- interface restricted to methods where the type parameter is not in a contravariant position

- useful for exporting "almost read-only" collections

Variance annotations (1/3)

Use-site covariance

```
Stack<? extends Animal> s = new Stack<Cat>();
Animal a = s.pop();          // OK
s.push(a);                   // KO
```

- interface restricted to methods where the type parameter is not in a contravariant position

- useful for exporting "almost read-only" collections

Variance annotations (2/3)

Use-site contravariance

```
Stack<? super Cat> s = new Stack<Animal>();
Animal a = s.pop();                    // KO
Object o = s.pop();                    // OK
s.push(new Cat());                     // OK
```

- interface restricted to methods where the type parameter is not in a covariant position or is replaced by the parameter bound

- counter-intuitive and rarely used, apart from Comparable

Variance annotations (2/3)

Use-site contravariance

```
Stack<? super Cat> s = new Stack<Animal>();
Animal a = s.pop();                    // KO
Object o = s.pop();                    // OK
s.push(new Cat());                     // OK
```

- interface restricted to methods where the type parameter is not in a covariant position or is replaced by the parameter bound

- counter-intuitive and rarely used, apart from Comparable

Variance annotations (3/3)

Definition-site variance

- class ImmutableContainer<+ T>
  { T get() ;}
- class Container<T> inherit ImmutableContainer<T>
  { put(T) ;}

☛ with + the type parameter is **co**variant
and cannot be used in a **contra**variant position

useful for exporting "actual read-only" collections

Variance annotations (3/3)

Definition-site variance

- class ImmutableContainer<+ T>
  { T get() ;}
- class Container<T> inherit ImmutableContainer<T>
  { put(T) ;}

☛ with – the type parameter is **contra**variant and cannot be used in a **co**variant position

counter-intuitive and rarely used

Contravariance and recursive bound

- interface Comparable<**T** extends Comparable<**T**>>
- class OrderedSet<**T** extends Comparable<**T**>>
- class **A** implements Comparable<**A**>
- OrderedSet<**A**> // OK

- class B extends A
  // **B** implements Comparable<**A**>
- OrderedSet<**B**> //
  // **B** doesn't implement Comparable<**B**>

Contravariance and recursive bound

- interface Comparable<**T** extends Comparable<**T**>>
- class OrderedSet<**T** extends Comparable<**T**>>
- class **A** implements Comparable<**A**>
- OrderedSet<**A**> // OK

- class B extends A // OK
  // **B** implements Comparable<**A**>
- OrderedSet<**B**> // KO
  // **B** doesn't implement Comparable<**B**>

Contravariance and recursive bound

- interface Comparable<**T** extends Comparable<**T**>>
- class OrderedSet<**T** extends Comparable<**T**>>
- class **A** implements Comparable<**A**>
- OrderedSet<**A**> // OK

- class B extends A implements Comparable<B> // KO
  // B cannot implement both
- OrderedSet<**B**> // KO

Contravariance and recursive bound

- interface Comparable<**T** extends Comparable<**T**>>
- class OrderedSet<**T** extends Comparable<**? super T**>> Java
- class **A** implements Comparable<**A**>
- OrderedSet<**A**> // OK

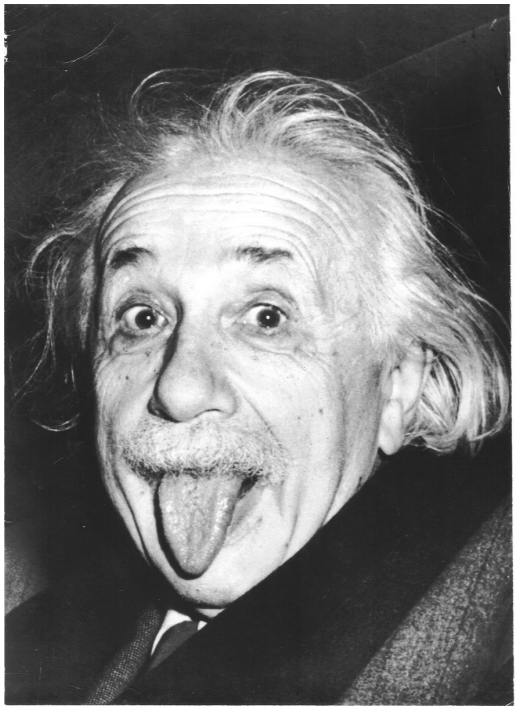
- class B extends A
  // **B** implements Comparable<A> <: Comparable<? super **B**>
- OrderedSet<**B**> // OK

Contravariance and recursive bound

- interface Comparable<**T** extends Comparable<**–T**>> Scala
- class OrderedSet<**T** extends Comparable<**T**>>
- class **A** implements Comparable<**A**>
- OrderedSet<**A**> // OK

- class B extends A
  // **B** implements Comparable<A> <: Comparable<**B**>
- OrderedSet<**B**> // OK

A programmer hierarchy

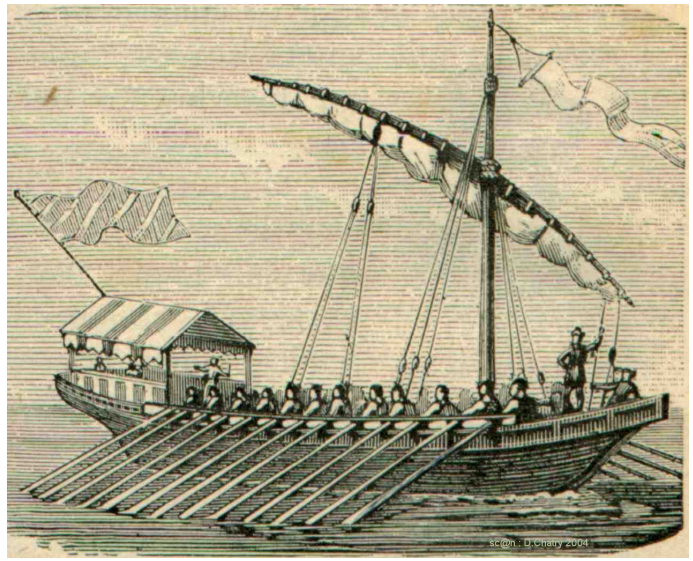


language designer

a Pure Light of programming

language or library implementer

an ingenious engineer



base programmer

an obscure rower

Variance annotations

Use-site variance more general than definition-site

- instead of defining **A<+T>** once,
- use **A<? extends T>** everywhere!

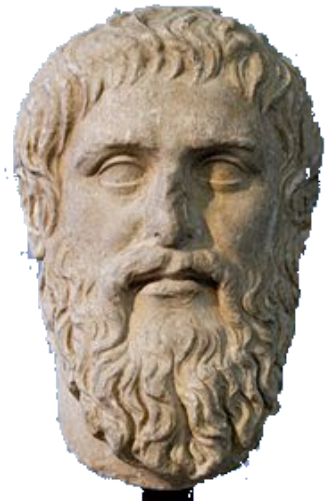
☛ It's unfair to impose to the base programmer the difficulties that could have been adressed by language designers or implementers

- using classes, especially generics, is easier that defining them
- definition-site variance should be added to Java and enlarged to all classes in C♯
- most of the burden of use-site variance would be avoided

Genericity (the end)

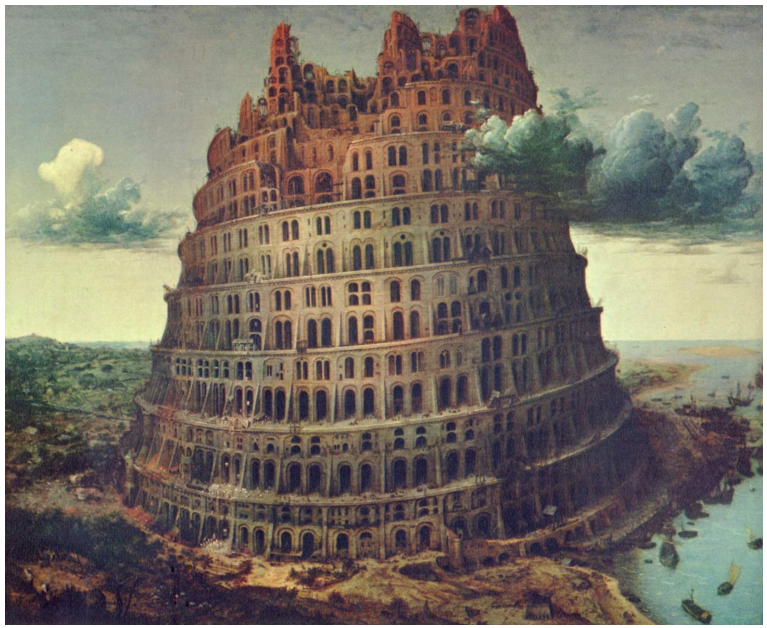
Ideal specifications

- no type erasure
- mixed implementation à la C♯
- definition- and use-site variance (Scala)
- array invariance
- recursive bound (*F-bounded*)
- no specialization of multiple generic instances
- no static overloading on the formal type
- better support of the IDEs



LIRMM

Plan



LIRMM

Ideal specifications of OO languages

In static typing

- metamodeling semantics of multiple inheritance
- generics with variance and runtime types
- without overloading

Ideal specifications of OO languages

Why static typing?

because

- dynamic typing is unsafe
- dynamic typing is too difficult for most programmers
- programming in dynamic typing is an art, not an industry
- ☛ static typing allows for exoskeletons like Eclipse

Before boarding an aircraft,
be sure that all the avionics is statically typed!

LIRMM

a programmer dreams that he commands Eclipse,
is it not Eclipse dreaming it is a programmer? (after 庄子)

Ideal specifications for other features

Constructors, i.e. initialization methods

- an open problem
- no satisfying specification

Reflection

- first-class metaclasses, based on variant generics
- as an extension of Java Class class

- language-level UML associations
- ... and certainly a few other object-oriented features

LIRMM

Ideal specifications for other features

Constructors, i.e. initialization methods

- an open problem
- no satisfying specification

Reflection

- first-class metaclasses, based on variant generics
- as an extension of Java Class class

- language-level UML associations
- ... and certainly a few other object-oriented features

LIRMM

What about existing languages?

Marginal evolution in Java and C♯

- global property conflicts (Java)
- covariant return types (C♯)
- definition-site variance (both)

Too many backwards incompatibilities

☛ type erasure seems to be definitive (Java, Scala)

☛ a complete solution involves new languages

LIRMM

Just do it!

- the specifications are state-of-the-art
- solutions exist for implementing it (another story...)

末尾