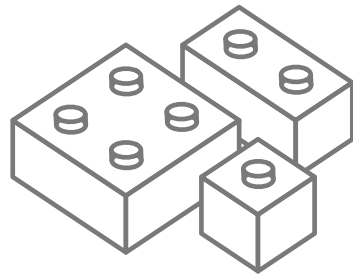


Traits in C#

Stefan Reichhart

stefan_reichhart@student.unibe.ch



Software Composition Group

u^b

b
**UNIVERSITÄT
BERN**

Oscar Nierstrasz, Stephane Ducasse

Roadmap



- ◆ Project Context / Why Traits ?
- ◆ What's a Trait ? Flattening Traits ?
- ◆ Traits in C# and STOO (**s**tatically **t**yped **o**bject **o**riented) Languages

Why Traits ?

Problem / Goal:

avoid *code duplication* and *fragile compositions*
share code easily

Current Reuse-Mechanisms:

Single Inheritance ? -- too *limited*

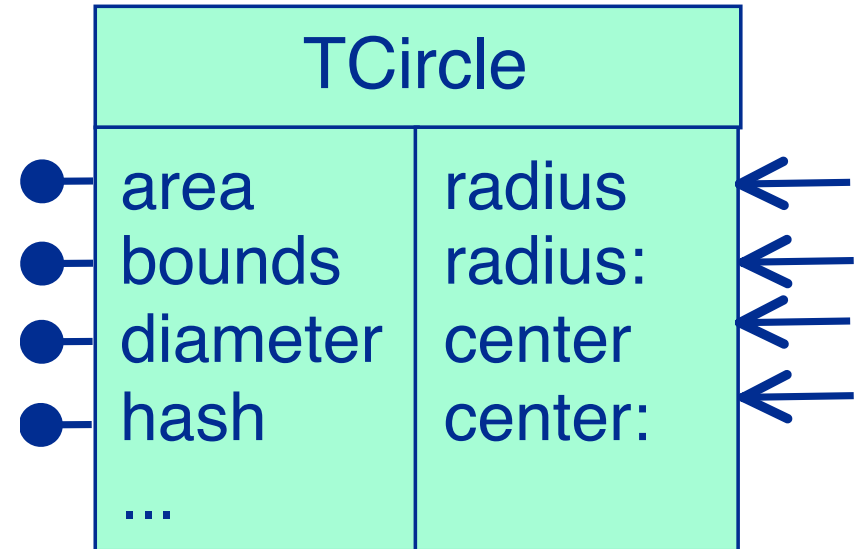
Multiple Inheritance ? } good, but too complex

Mixins ? } fragile composition,
“*ripple effect*”

Solution: Traits -- simple, efficient, cool !

What is a “Trait” ?

- ◆ *first-class* compose-time group of pure methods

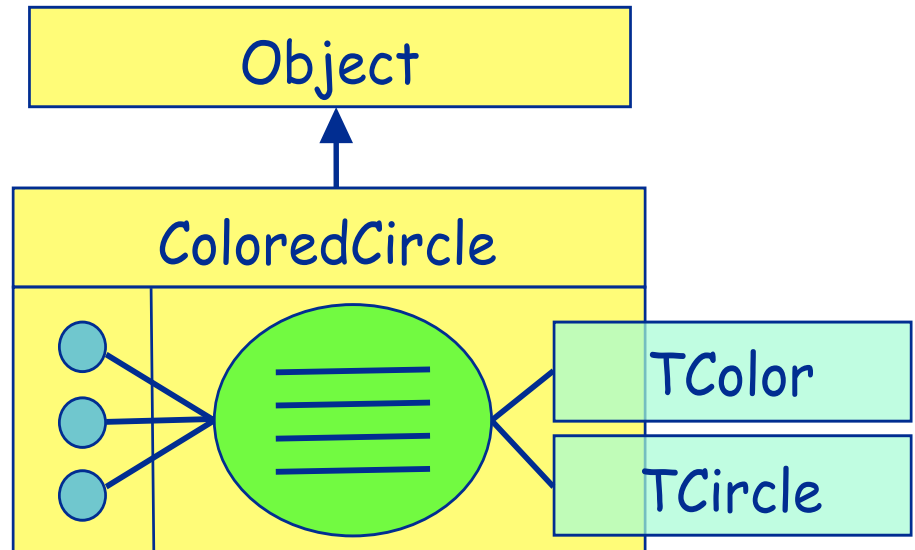


Properties:

- ◆ *stateless, pure behavior*
- ◆ *provides* ● — & *requires* ← a set of methods

Traits & Composition (I)

- ◆ *complements* single inheritance
- ◆ composition order is *irrelevant*
- ◆ *compose time* entity



- ◆ composite entity is in *full control of the composition*

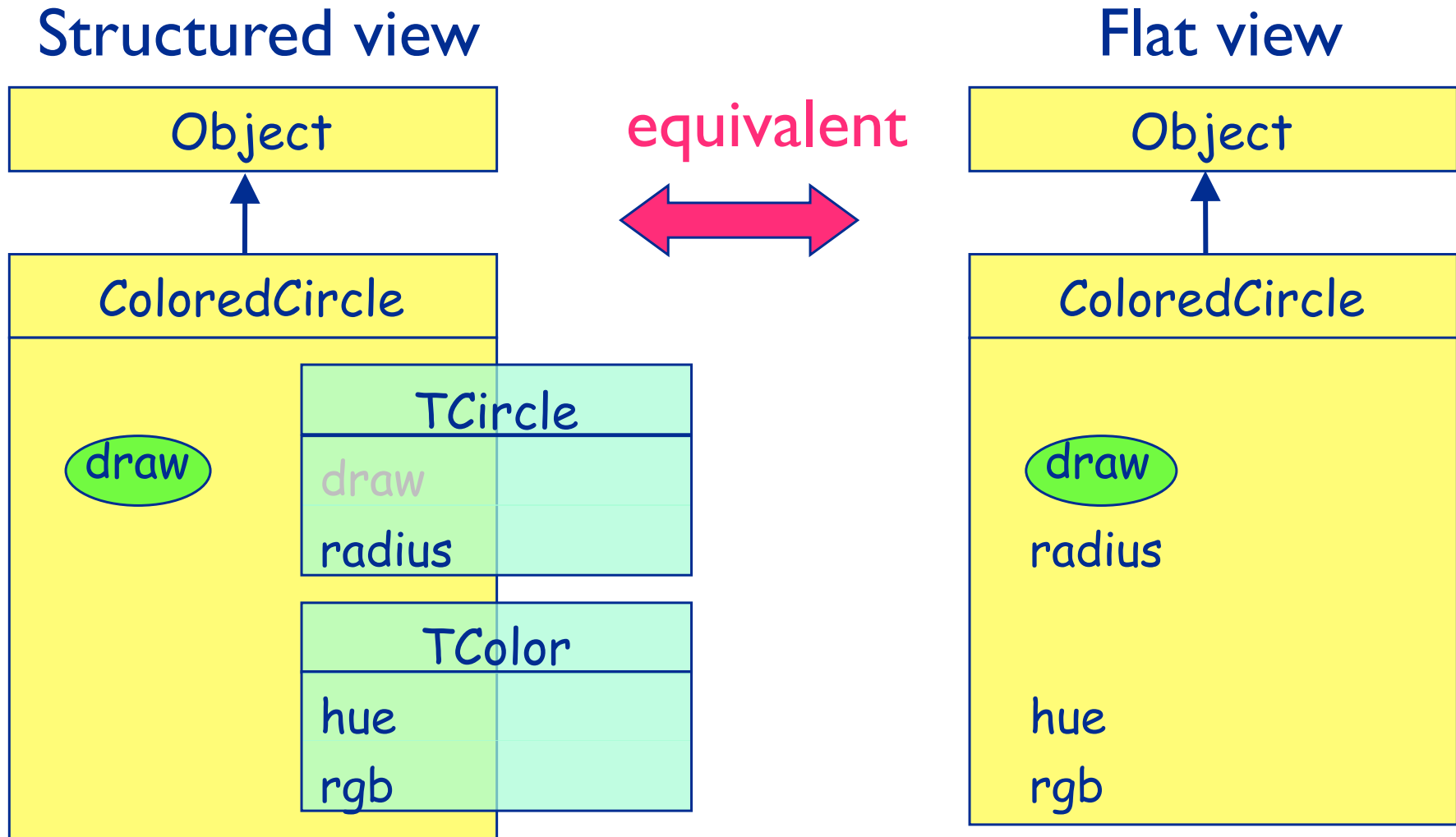
**Class = Superclass + State
+ Traits + Glue methods**

Traits & Composition (2)

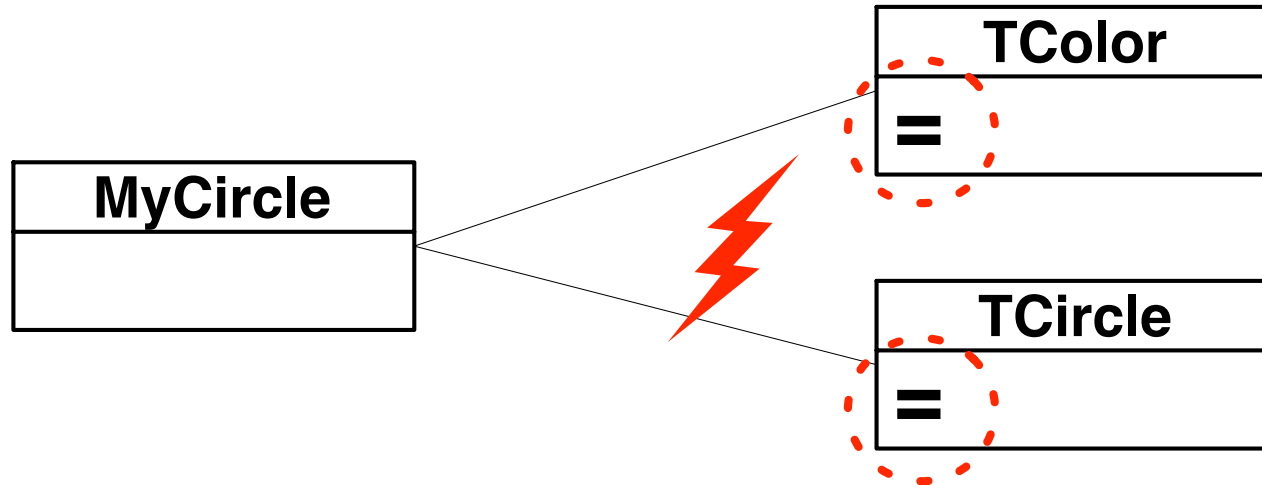
- ◆ class methods >>> trait methods
- ◆ *no change* in overriding !
- ◆ *self/this, super/base* have the *same meaning as before* !

= consequences by the ***flattening property***

Flattening Traits



Traits & Conflicts (I)

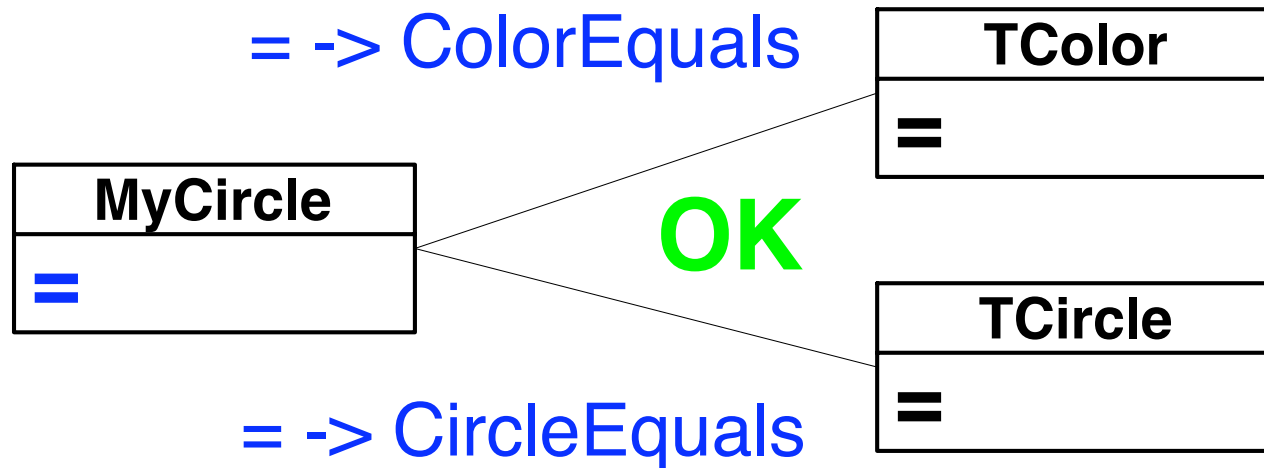


explicit conflict resolution by the composite entity

Alias \rightarrow and **Exclusion** \wedge

Traits & Conflicts (2)

... one possible solution: **aliasing**



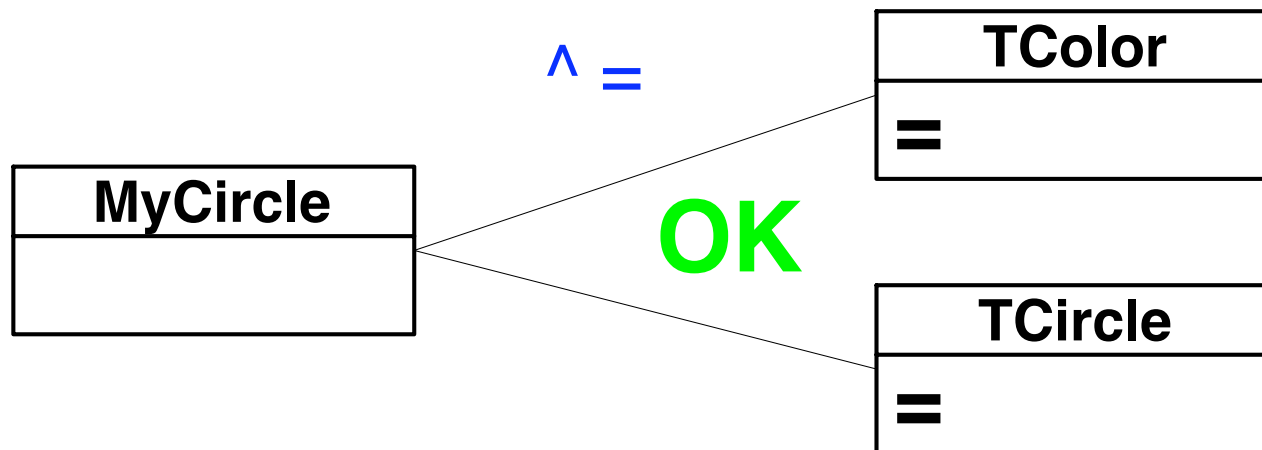
Traits & Conflicts (3)

```
class Circle {  
  uses{  
    TColor { = -> ColorEquals; };  
    TCircle { = -> CircleEquals; };  
  }  
  public boolean operator =(Circle c) {  
    return this.ColorEquals(c)  
      && this.CircleEquals(c);  
  }  
}
```

```
trait TColor { ... }  
trait TCircle { ... }
```

Traits & Conflicts (4)

... an alternative ? **exclusion**



Traits & Conflicts (5)

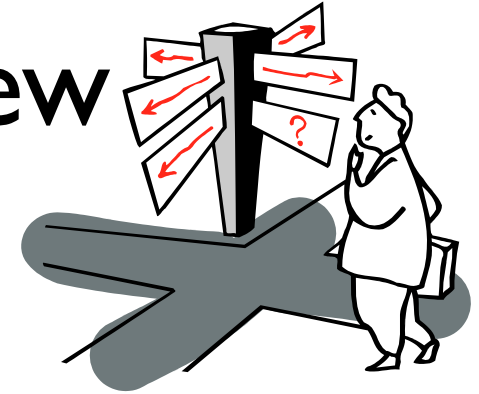
```
class Circle {  
    uses{  
        TColor { ^=; };  
        TCircle;  
    }  
}
```

```
trait TColor { ... }  
trait TCircle { ... }
```

Traits

- ◆ available in *dynamically typed oo languages*
 - Smalltalk, Slate, Perl 6 (role)
 - experiences:** *simple* and *expressive*
- ◆ what about *statically typed oo languages*, e.g. **C#** ???
 - Scala, MiniJava

Traits in C# : Overview



Trait *Container* and *Definition* }

Typed *Syntax/Declaration* }



3 Kinds of Typed Traits in STOO languages

Generics and Generic Constraints

Typing Traits and *Type Problems* ←



Respect *modifiers* and *keywords*

Handling of *libraries* / *packages*

Trait *Interfaces*

... ..

Typed Syntax (I)

Typed Trait Declaration:

- ◆ typed *alias*, *exclusion* and *requirements*

Solution (conceptually simple):

- ◆ alias, exclusion → *argument* types
- ◆ requirements → *argument* (and *return*) types

Squeak / ST

Object subclass: #MyCircle

instanceVariableNames: 'color'

```
uses {  
  TColor @ { #invertAliased -> #invert }.  
  TShape - { #resize: }.  
}
```

```
MyCircle >> color  
  ^ color
```

Trait named: #TColor

```
uses { }  
requires { #color }  
MyColor >> invert  
  ^ ... self color ...
```

C#

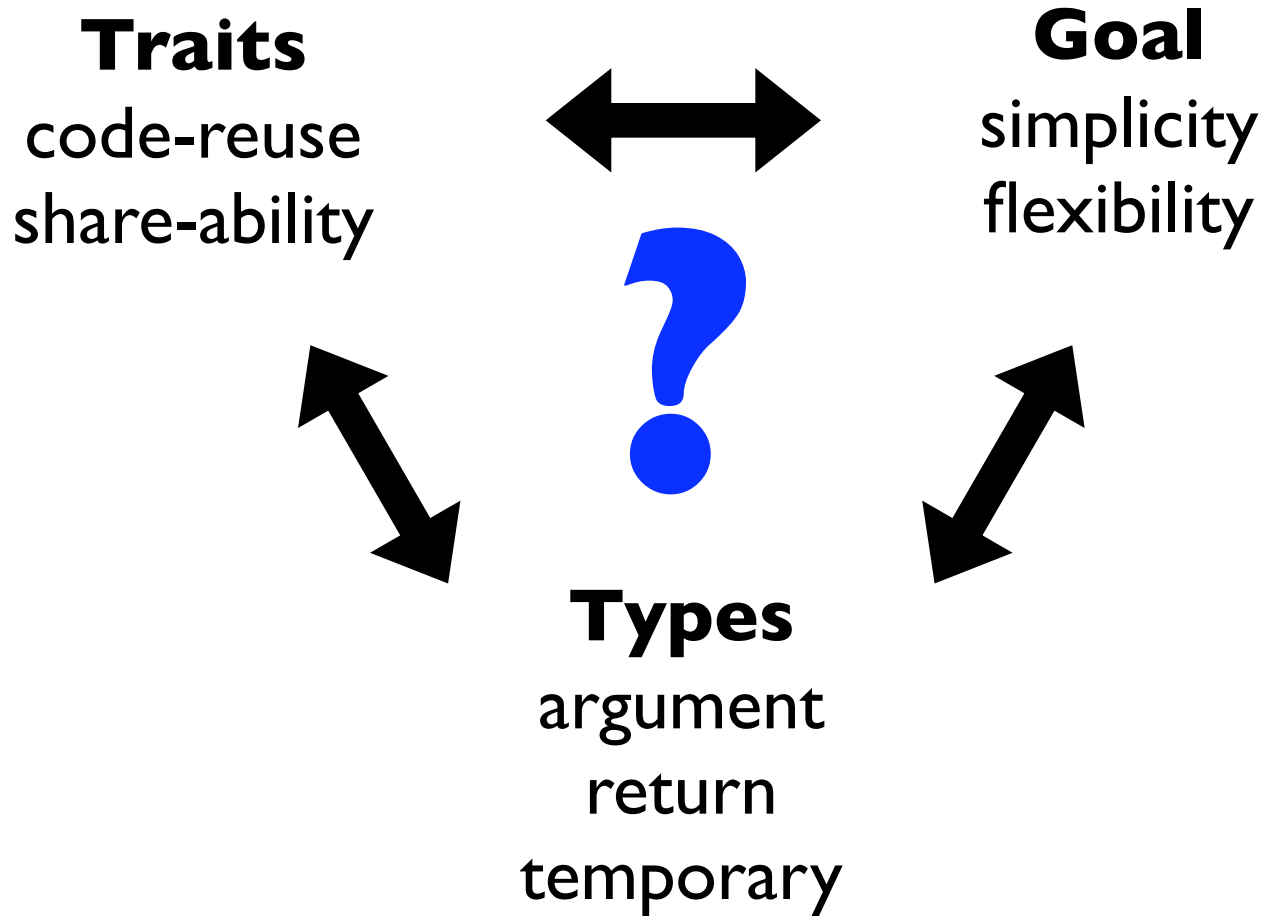
public class MyCircle {

```
  uses {  
    TColor {invert()->invertAliased;};  
    TShape {^resize(int);};  
  }  
  private IColor color;  
  public IColor Color() {  
    return this.color;  
  }  
}
```

trait TColor {

```
  requires {  
    IColor Color();  
  }  
  public IColor Invert() {  
    ... return this.Color();  
  }  
}
```

Typing Traits (I)



Typing Traits (2)

Typed Trait methods

- ◆ what *argument* types ?
- ◆ what types should be *returned* by methods ?
- ◆ how to return the *type of the class* using the trait ?
- ◆ *temporary* types ?



Traits
code-reuse
share-ability
simplicity
flexibility

Typing Traits (3)

```
class MyCollection { uses {TSequenceable;} }

trait TSequenceable {
  public ??? Reverse() {
    ??? reversedList = new ???();
    ...
    return reversedList;
  }
  public void Concat(??? c1, ??? c2) {...}
}
```

Typing Traits (4)

Possible Solutions:

- ◆ Simple/Concrete types
- ◆ Interfaces
- ◆ Explicit / Parameterized ←
- ◆ Implicit / Type Keyword
- ◆ Implicit / Return-Types & Generics ←
- ◆ ...

Explicit / Parameterized

```
class MyCollection {  
    uses { TSequenceable<MyCollection>; }    OR  
    uses { TSequenceable<ICollection>; }  
}
```

```
trait TSequenceable<A> {  
    public A Reverse() {...}  
    public void Concat(A.. c1, A.. c2) {...}  
}
```

Explicit / Parameterized

Advantage:

- ◆ *simple, extremely flexible*
- ◆ *template-like* use, on top of any language

Disadvantage:

- ◆ multiple/variable use of the type parameter
- ◆ *inconsistent syntax/use* when generics are used

Implicit Return Types & Generics

```
class MyCollection<T> {  
    uses { TSequenceable<T>; } ...  
}
```

```
trait TSequenceable<A> {  
    public TSequenceable Reverse() {...}  
    public void Concat(TSeq.. c1, TSeq.. c2)  
    {...}  
}
```

Implicit Return Types & Generics

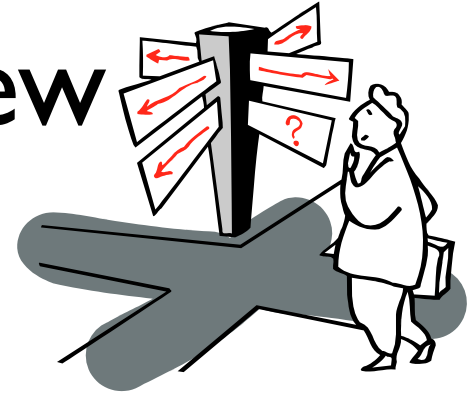
Advantage:

- ◆ *simple*
- ◆ *consistent syntax* for generics and non-generics

Disadvantage:

- ◆ *limited flexibility* for return types
- ◆ *not suitable for any use*

Traits in C# : Overview



Trait *Container* and *Definition* } ~OK
Typed *Syntax/Declaration*

3 Kinds of Typed Traits in STOO languages

Generics and Generic Constraints

Typing Traits and *Type Problems* ~OK

Respect *modifiers* and *keywords*

Handling of *libraries* / *packages*

Trait *Interfaces*

... ..

Prototype Implementation (I)

- ◆ *Preprocessor* based on the *flattening property*
- ◆ *Simple Typed* and *Generic Traits*
(no var-binding, strict-matchings only)
- ◆ Simple and minimal *Typed-Trait-syntax* for declarations and requirements
- ◆

Prototype Implementation (2)

- ◆ *Generic parser* (most C-like languages supported)
- ◆ Simple CodeDOM framework
- ◆ Language-*independant flattening logic*
- ◆

Future for Traits in C# / STOO

- ◆ Finding a *satisfying solution* for *modifier/keyword* and *typing problems*
- ◆ Combining Template-like & Generic Traits (?)
- ◆ Requirement constraints for methods (?)
- ◆
- ◆ Clean Implementation/Integration in *Rotor.NET*

References (I)

N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black.

Traits: Composable units of behavior. In Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming), volume 2743 of LNCS, pages 248–274. Springer Verlag, July 2003.

S. Ducasse, N. Schärli, O. Nierstrasz, R. Wuyts, and A. Black.

Traits: A mechanism for fine-grained reuse.

Transactions on Programming Languages and Systems, 2005.
under revision.

References (2)

N. Schärli.

Traits — Composing Classes from Behavioral Building Blocks. PhD thesis, University of Bern, Feb. 2005.

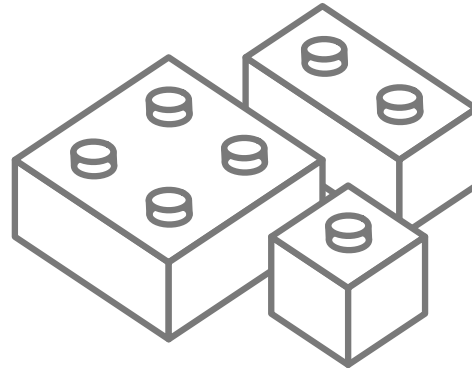
O. Nierstrasz, S. Ducasse, and N. Schärli.

Flattening traits. Journal of Object Technology, 5(3):0–0, May 2006. To appear.

S. Reichhart

Traits in C#, Technical Report and prototype implementation, University of Bern, Switzerland, Sept. 2005

References (3)



Software Composition Group

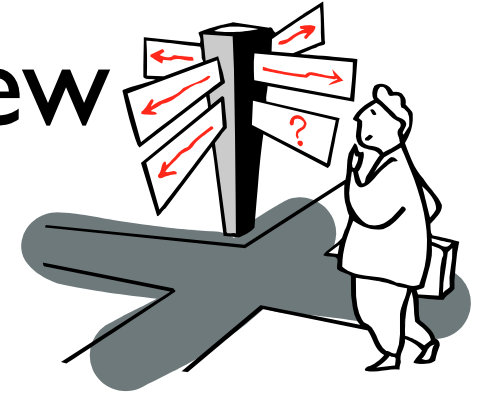
Webpage

<http://www.iam.unibe.ch/~scg/Research/Traits/index.html>

Traits

- ◆ ***simple but effective reuse-mechanism***
- ◆ ***no change* to the existing composition model (overriding, ...)**
- ◆ ***fully controlled composition***

Traits in C# : Overview



Trait *Container* and *Definition* } ~OK
Typed *Syntax/Declaration*

3 Kinds of Typed Traits in STOO languages

Generics and Generic Constraints

Typing Traits and *Type Problems* ~OK

Respect *modifiers* and *keywords*

Handling of *libraries* / *packages*

Trait *Interfaces*

... ..

3 Kinds Of Traits



- ◆ Simple Traits
- ◆ Template-like traits
- ◆ (full) Generic Traits

Simple Traits

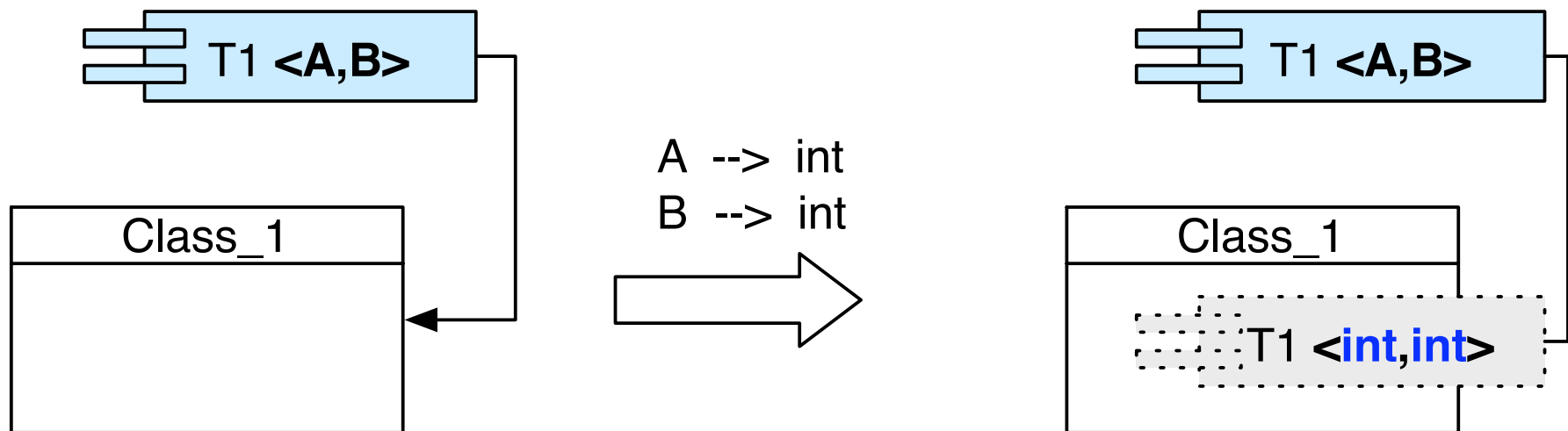
- ◆ uses *identities* of *concrete argument-* and *return types*
- ◆ *type-problems* not handled
- ◆ very simple, but *limited* code reuse

```
class MyCollection {  
    uses { TSequenceable; } ...  
}
```

```
trait TSequenceable {  
    public ICollection Reverse() {...}  
    public void Concat(IC.. c1, IC.. c1) {...}  
}
```

Template-like Traits (I)

- ◆ not like C++ templates !
- ◆ simple, variable, *flexible type parameter*
- ◆ *type-problems* might be handled



Template-like Traits (2)

```
class MyCollection {  
    uses { TSequenceable<ICollection>; } ...  
}
```

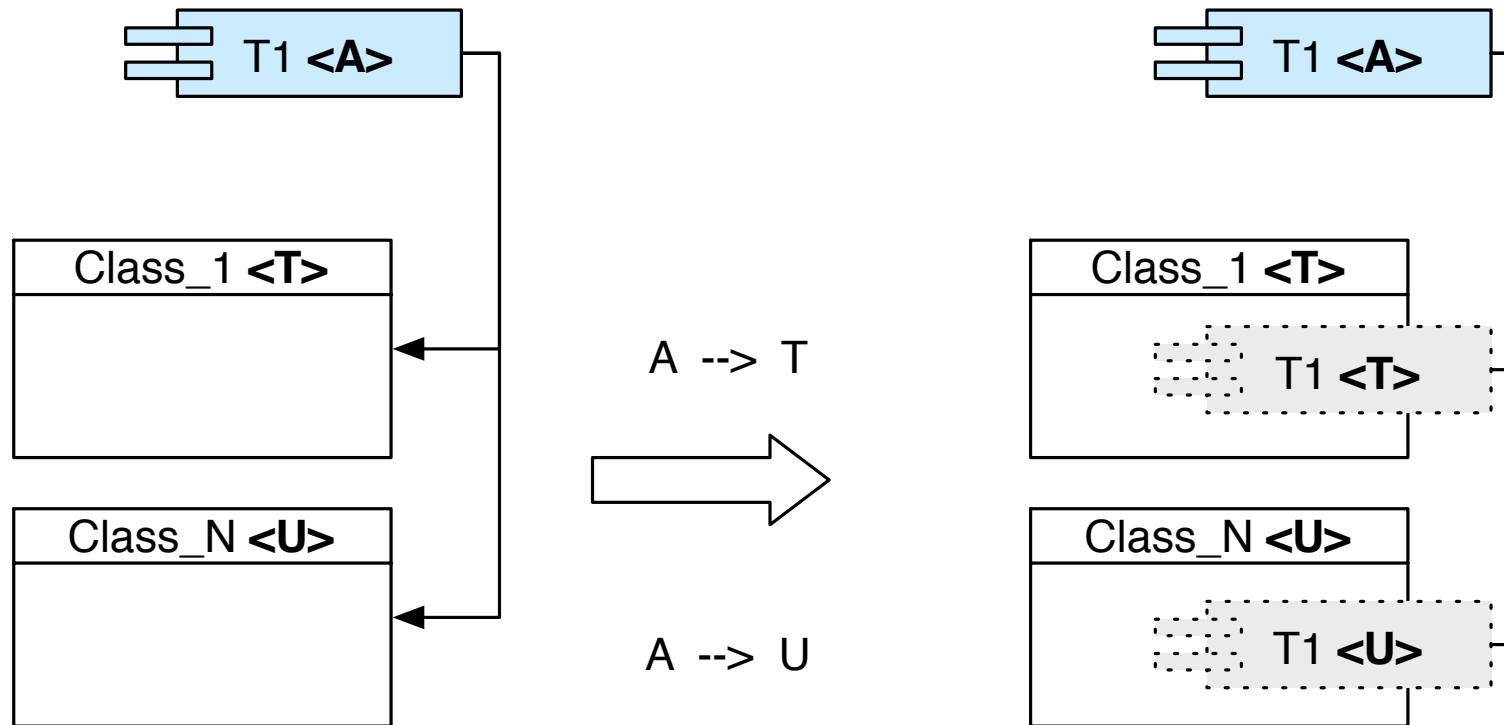
```
trait TSequenceable<A> {  
    public A Reverse() {...}  
    public void Concat(A c1, A c1) {...}  
}
```

or like this ...

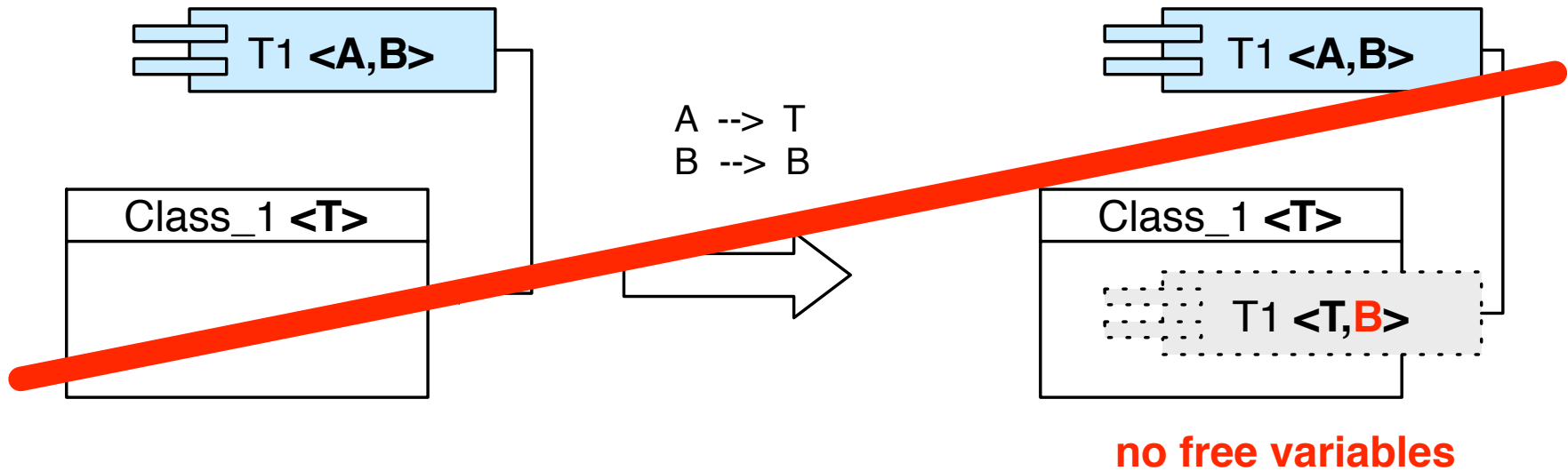
```
class MyCollection {  
    uses { TSequenceable<MyCollection>; } ...  
}
```

Generic Traits (I)

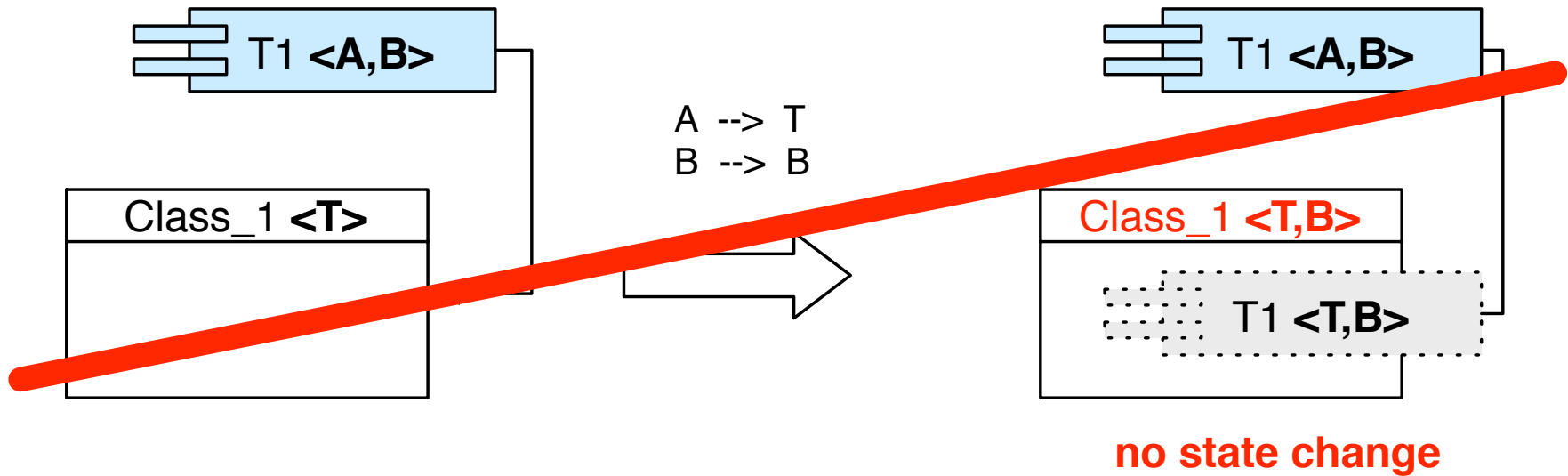
- ◆ parametric Polymorphism, *flexible* code reuse
- ◆ variable binding, a bit *more complex ...*



Generic Traits (2)



Generic Traits (3)



Generic Traits (4)

genericTypeParameters(class)

>=

\sum (genericTypeParameters(Ti))

not more or no 'different' generic type parameters

Generic Traits (5)

```
class MyCollection<T> {  
    uses { TSequenceable<T>; } ...  
}
```

```
trait TSequenceable<A> {  
    public ??? Reverse() {...}  
    public void Concat(??? c1, ??? c1) {...}  
}
```

??? type-problems

Modifiers / Keywords (C# only)

Trait Properties:

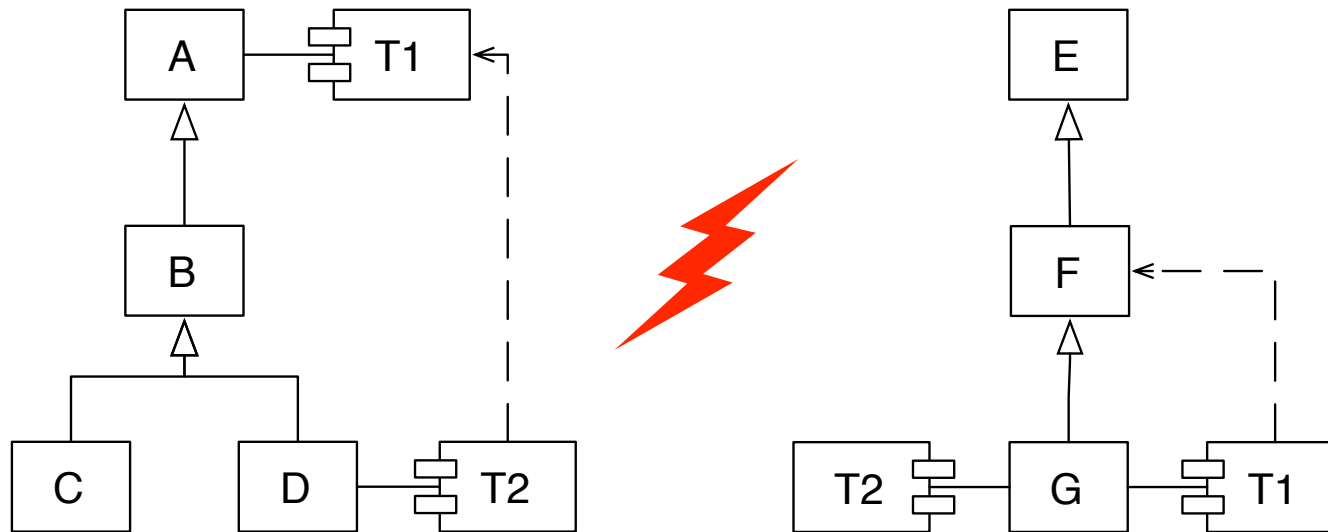
trait methods may want/need to *override/hide* methods of higher levels (classes or traits)

C#'s explicitly:

all methods must declare correct *inheritance* (and *accessibility modifiers*)

virtual, override, new, public, ...

Modifiers / Keywords (2)



Problem & Disadvantage:

- ◆ code sharing is *limited*
- ◆ might lead to *code duplication* (again)

Modifiers / Keywords (3)

Solution Basics:

tests should *catch conflicts* when processing traits
but: not a real solution to the problem ...

Possible solutions:

- ◆ Explicit modifiers for trait declaration ?
- ◆ Implicit resolution ?
- ◆ Avoid overriding traits ?
- ◆ ?

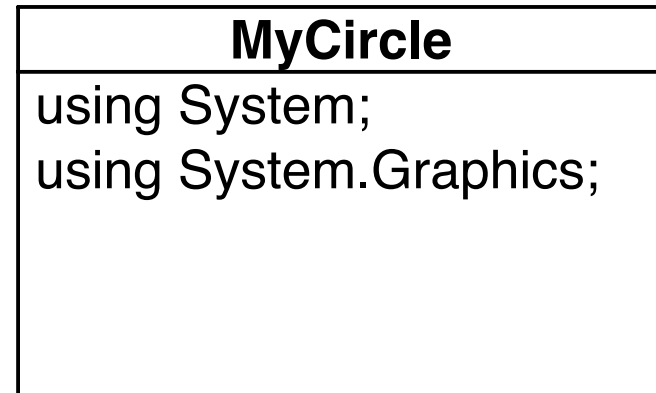
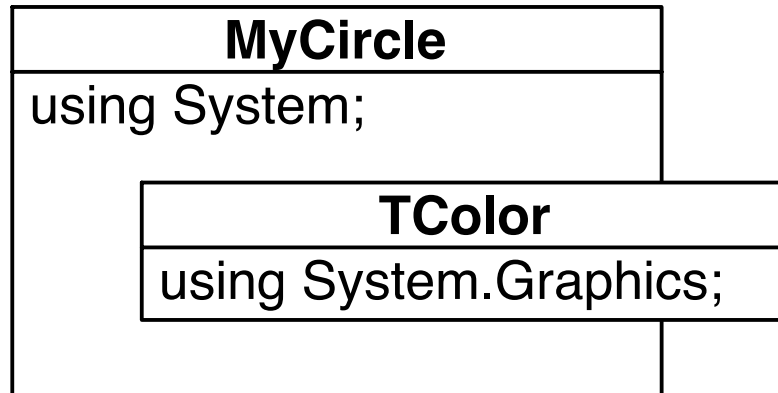
Library / Package propagation

Situation

Traits may also *depend on libraries*

Solution

implicit propagation of libraries to the higher level



Trait Interfaces (I)

Situation:

Traits may also *implement interfaces*

```
class MyShape { uses {TColor; } ... }  
trait TColor : IColorable { ... }
```

Solution:

implicit propagation of interfaces to the higher levels

```
class MyShape : IColorable { ... }
```

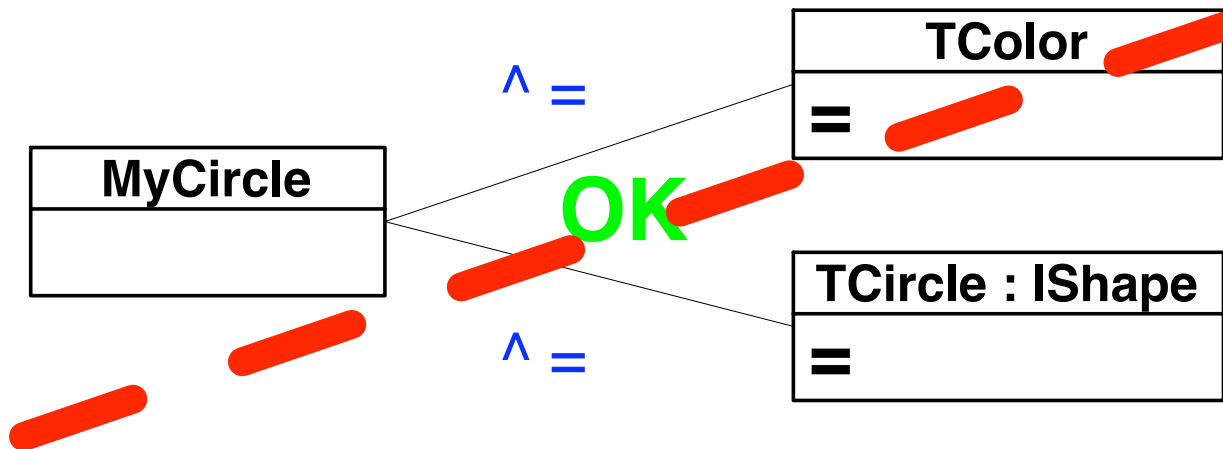
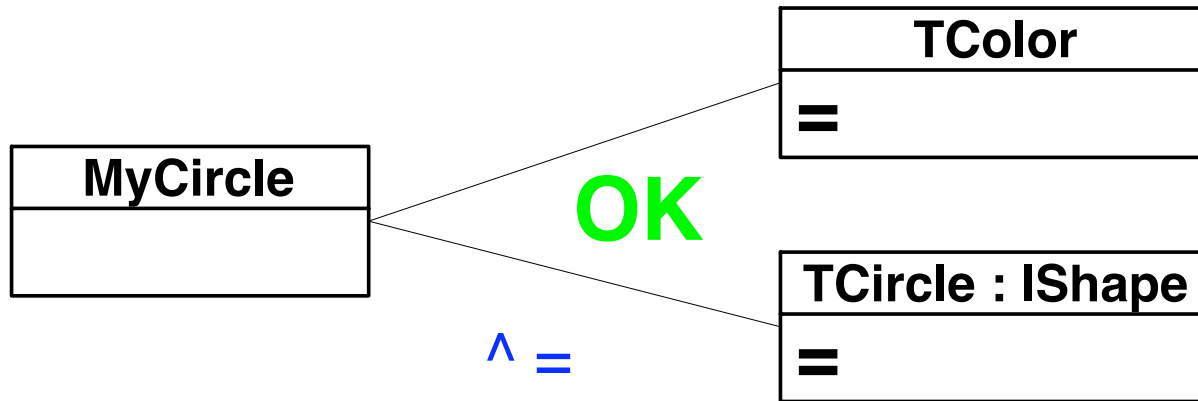
Trait Interfaces (2)

Advantage:

no explicit interface declaration / maintainance

!!! Special Case: Interfaces used with Exclusion
Confusing but correct, no breaking of the interface !

Trait Interfaces (3)



Generic Constraints (C# only)

Situation

Traits may also *use constraints* on generic type parameters

```
trait TSequenceable<T>  
    where T : INumber {...}
```

Generic Constraints (2)

Solution

Test if constraints collide with constraints on higher levels

$$\mathbf{constraints(class) \geq \sum (constraints(T_i))}$$

not more, not more 'restrictive' or different constraints

Generic Constraints (3)

```
class MyRational<T> where T : IInteger {...}  
trait MyAlgebra<A> where A : IFloatingPoint {...}
```

```
class MyDictionary<T> where T : IAssociation {...}  
trait MySequence<A> where A : INumber {...}
```

Generic Constraints (4)

```
class MyDictionary<T> where T : IAssociation {...}  
trait MySequence<A> where A : IAssociation,  
ILockable {...}
```

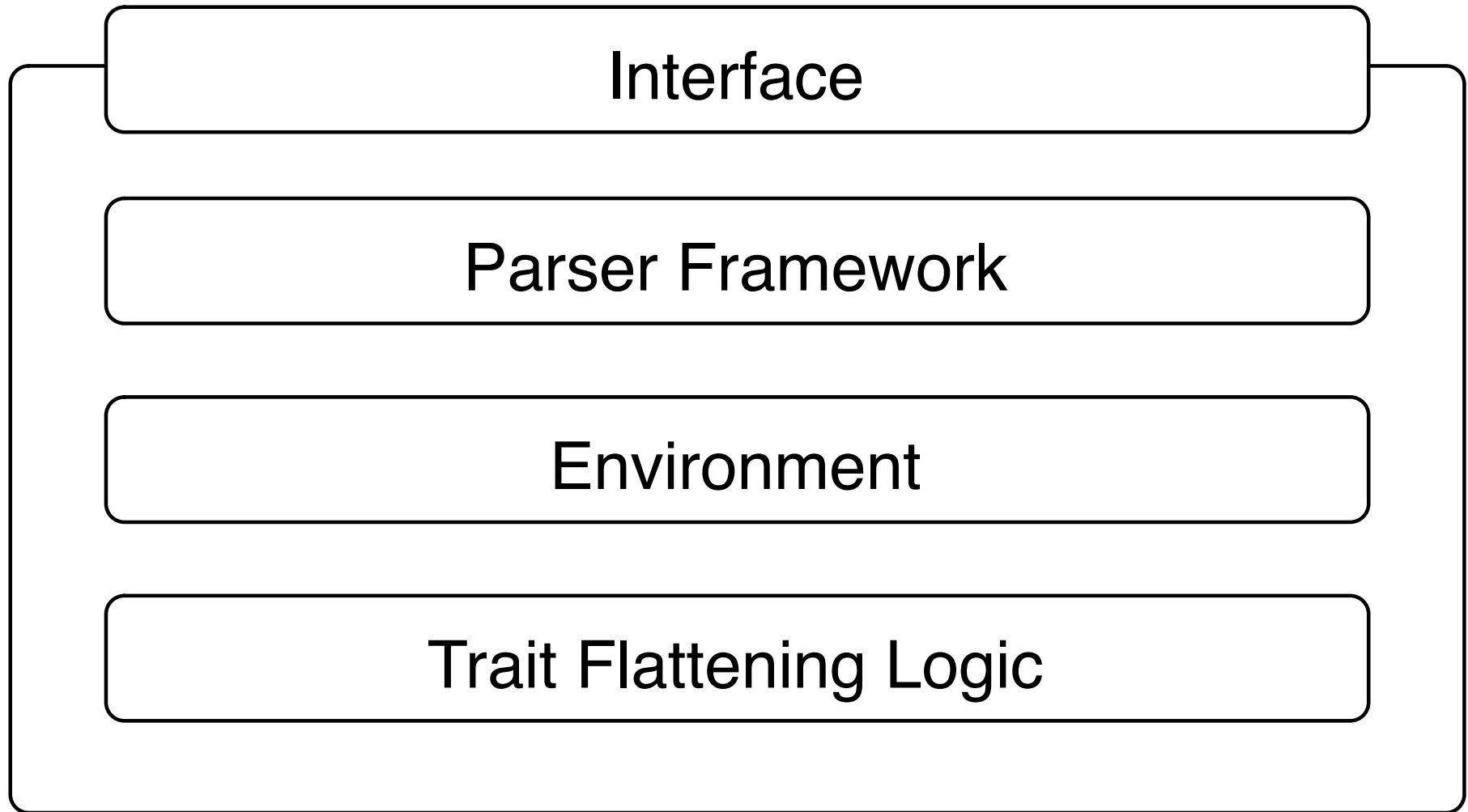
```
class MyRational<T> where T : IFloatingPoint {...}  
trait MyAlgebra<A> where A : IInteger {...}
```

Generic Constraints (5)

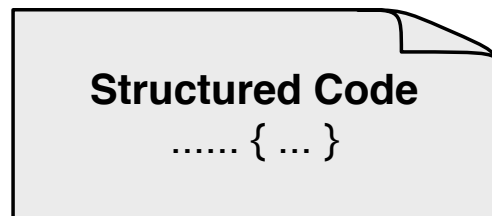
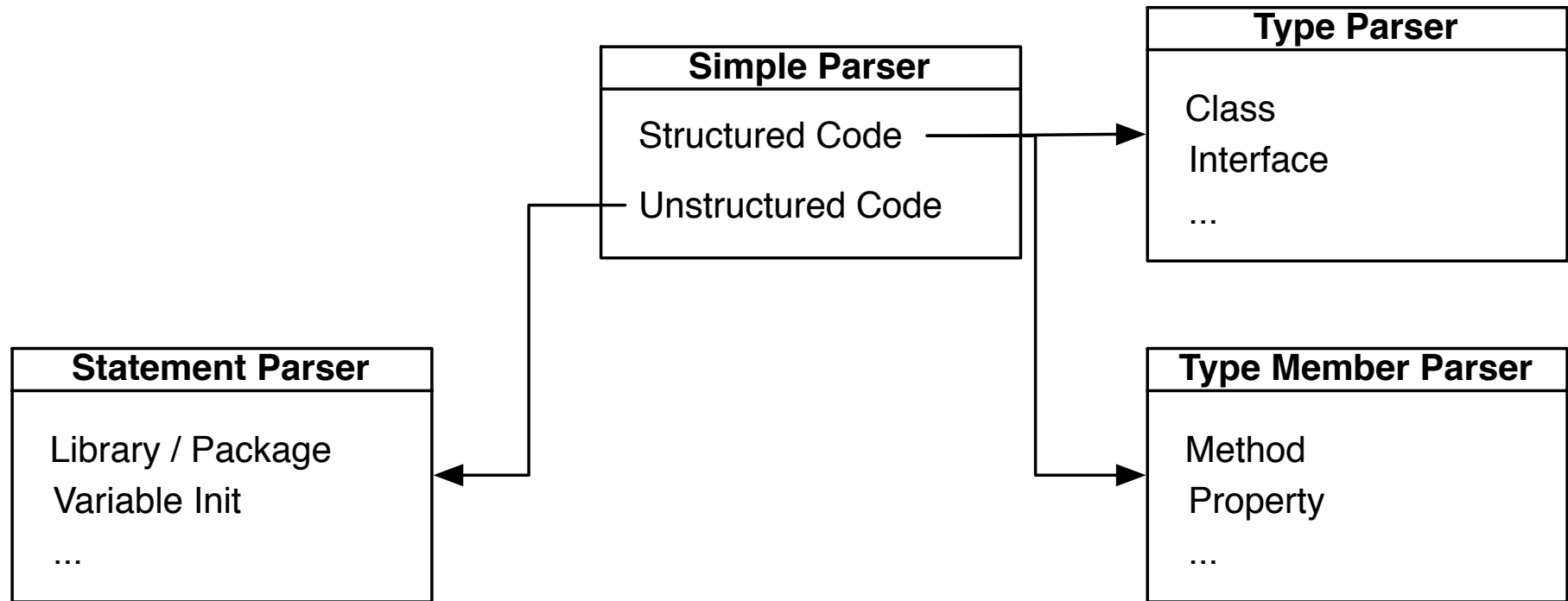
Disadvantage:

- ◆ Constraints *may prevent the ability to share behavior easily*
- ◆ Use is questionable (?)

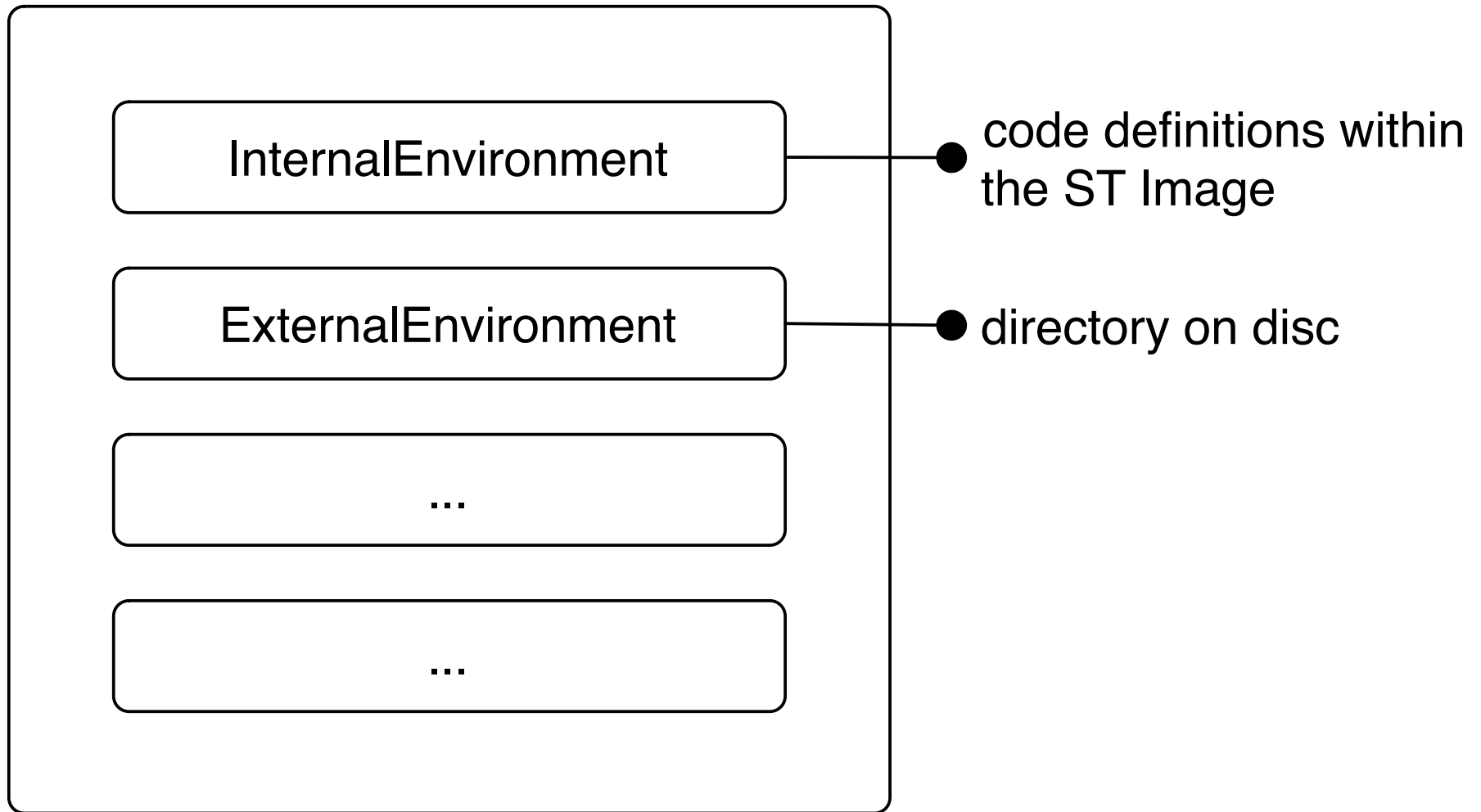
Traits Preprocessor



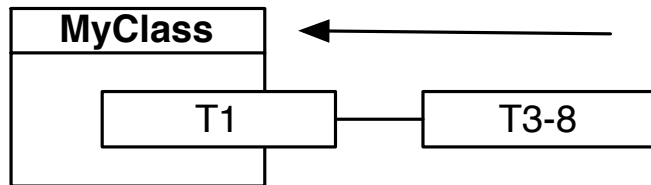
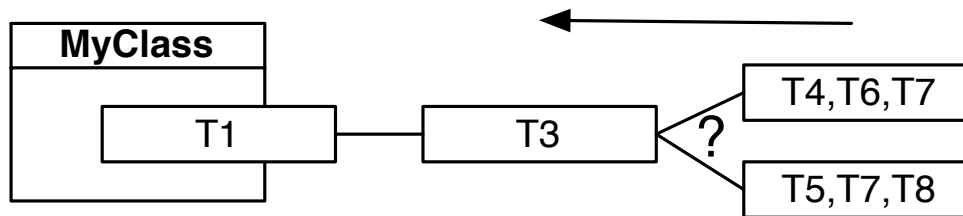
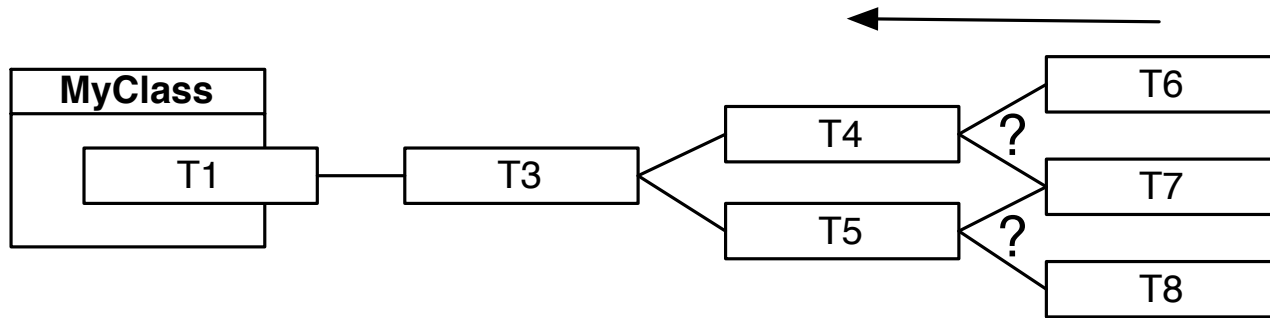
Parser Framework



Environment (singleton)



Trait Flattening Logic (I)



...

Trait Flattening Logic (2)

