25) Invasive Software Composition (ISC)

Prof. Dr. Uwe Aßmann

Florian Heidenreich

Technische Universität Dresden

Institut für Software- und Multimediatechnik

http://st.inf.tu-dresden.de

Version 11-0.5, Juli 6, 2011

1. Invasive Software Composition - A Fragment-Based Composition Technique

2. What Can You Do With Invasive Composition?

3. Functional and Composition Interfaces

4. Different forms of grey-box components

5. Evaluation as Composition Technique

Obligatory Literature

- ► ISC book Chap 4

- ► www.the-compost-system.org

- ► www.reuseware.org

Other References

Jakob Henriksson. A Lightweight Framework for Universal Fragment Composition. Technische Universität Dresden, Dec. 2008

http://nbn-resolving.de/urn:nbn:de:bsz:14-ds-1231251831567-11763

Jendrik Johannes. Component-Based Model-Driven Software Development. Technische Universität Dresden, Dec. 2010

http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-63986

Jendrik Johannes and Uwe Aßmann, Concern-Based (de)composition of Model-Driven Software Development Processes. Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, 2010,Part II, Springer, 2010, LNCS 6395, URL = http://dx.doi.org/10.1007/978-3-642-16129-2
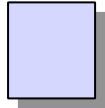
Falk Hartmann. Safe Template Processing of XML Documents. PhD thesis. Technische Universität Dresden, July 2011.

Software Composition

Component Model

Composition Technique

Composition Language
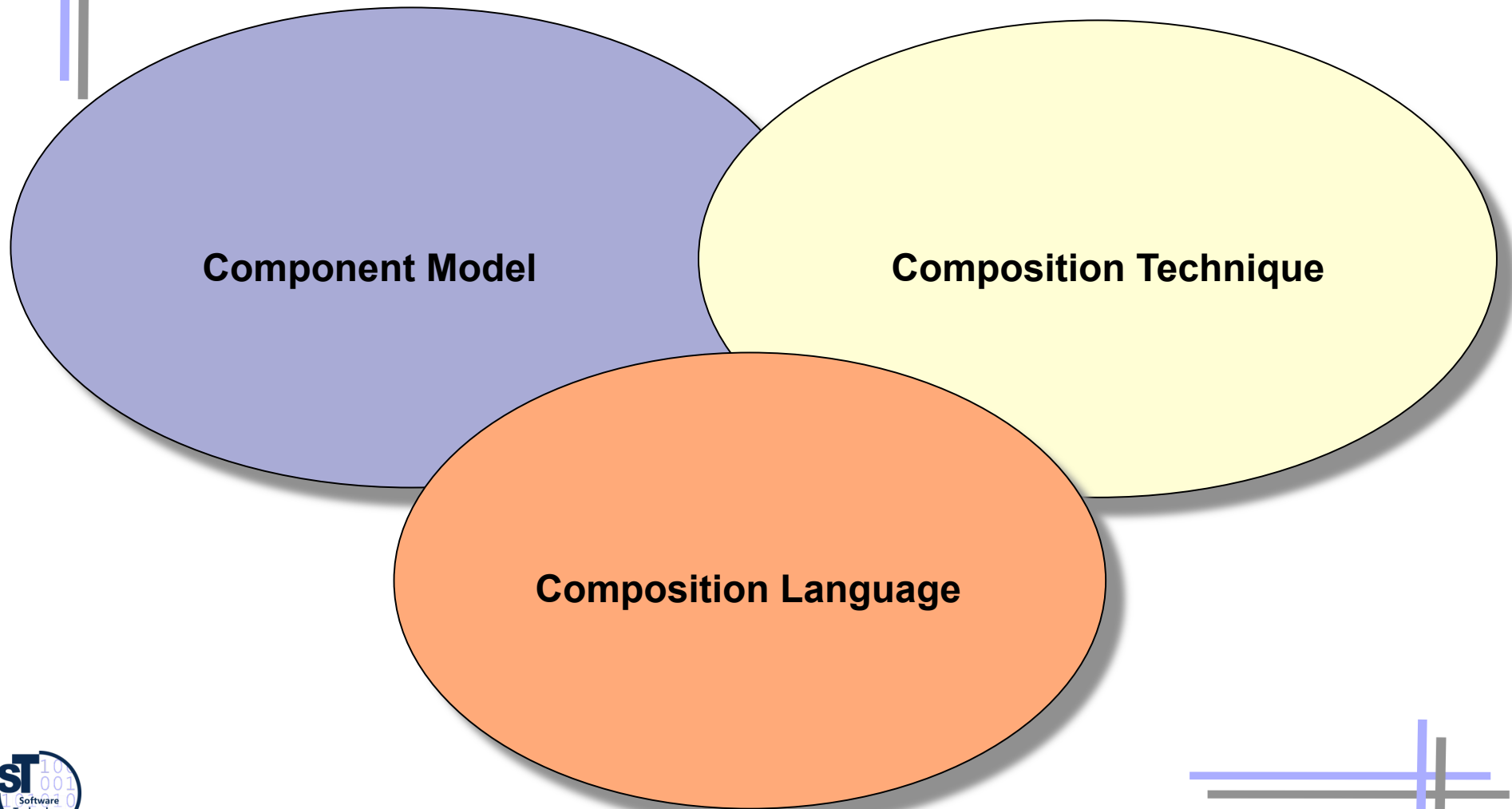
# 25.1) Invasive Software Composition - A Fragment-Based Composition Technique

Invasive Software Composition

Invasive software composition **parameterizes** and **extends**
fragment components
at **hooks**
by transformation

► A **fragment component** is a fragment group (fragment container, fragment component, fragment box)

- a set of fragments or fragment forms

► Uniform representation for

- a fragment
  - a class, a package, a method
- a fragment group
  - an advice or an aspect
  - some metadata
  - a composition program
- A generic fragment (group)
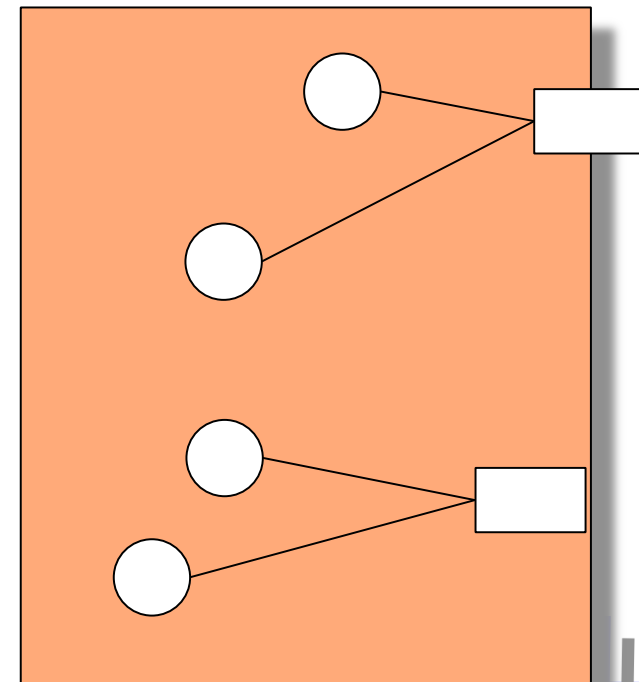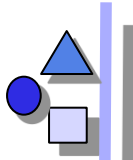
# The Component Model of Invasive Composition

Hooks are *change points* of a fragment component:

**fragments or positions,
which are subject to change**

► Fragment components have hooks (change points)

► A *change point* can be

- An *extension point (hook)*

- A *variation point (slot)*

► Example:

- Extension point: method entries/exits

- Variation point: Generic parameters

Implicit Hooks

- ► A **hook (extension point)** is given by the component's language
- ► Hooks can be *implicit* or *explicit (declared)*
  - We draw implicit hooks *inside* the component, at the border
- ► Example: Method Entry/Exit

m (){

abc..
cde..

}

Method.entry

Method.exit
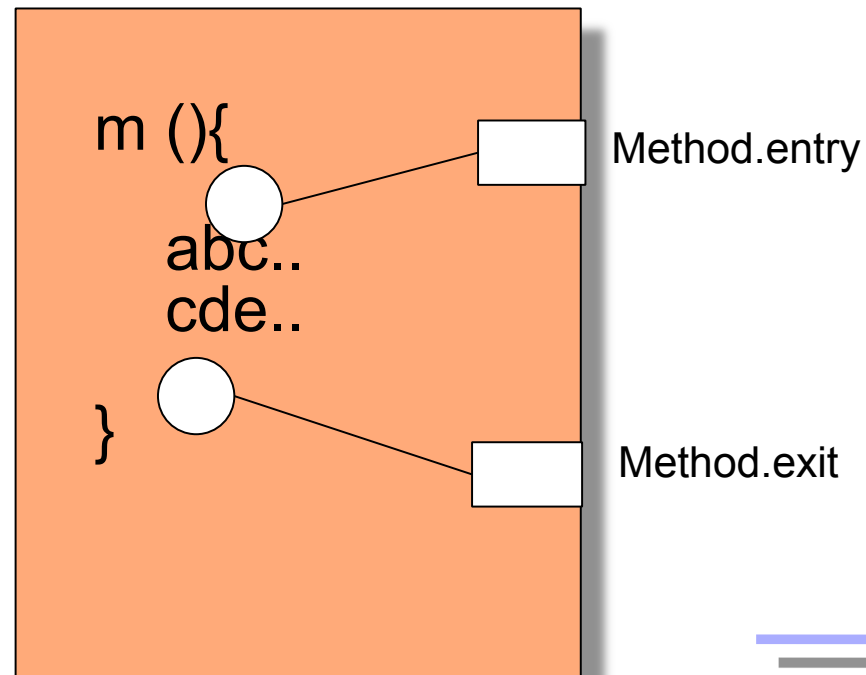
Slots (Declared Hooks)

► A **slot** is a variation point (a code parameter)

► Slots are always *declared,* i.e., declared or explicit hooks

  ▪ They are never implicit, i.e., must be declared by the component writer

 ▪ We draw slots as crossing the border of the component

Declarations
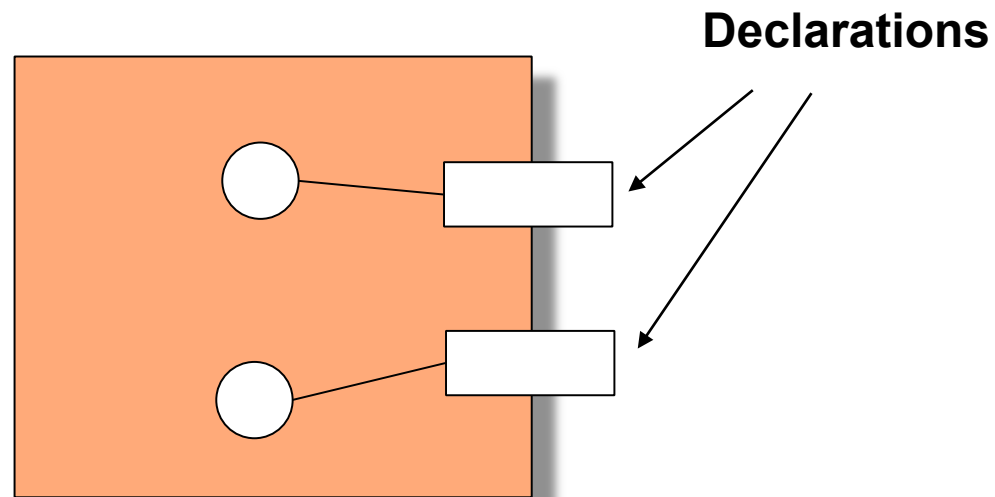
The Composition Technique of Invasive Composition

Invasive Software Composition
parameterizes and **extends**
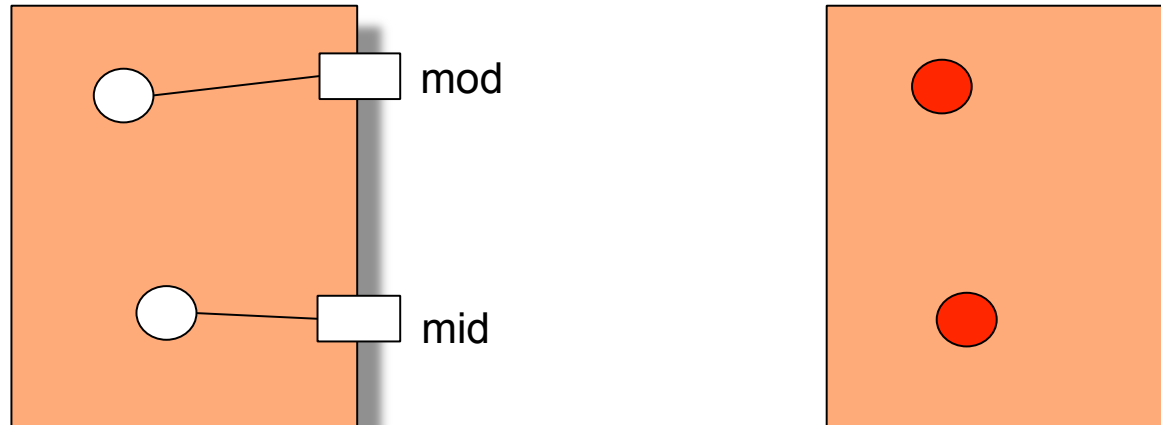**fragment components**
at implicit and declared **hooks**
by transformation

An invasive composition operator treats
declared and implicit hooks uniformly

The Composition Technique of Invasive Composition

► A *composer (composition operator)* is a static metaprogram (program transformer)
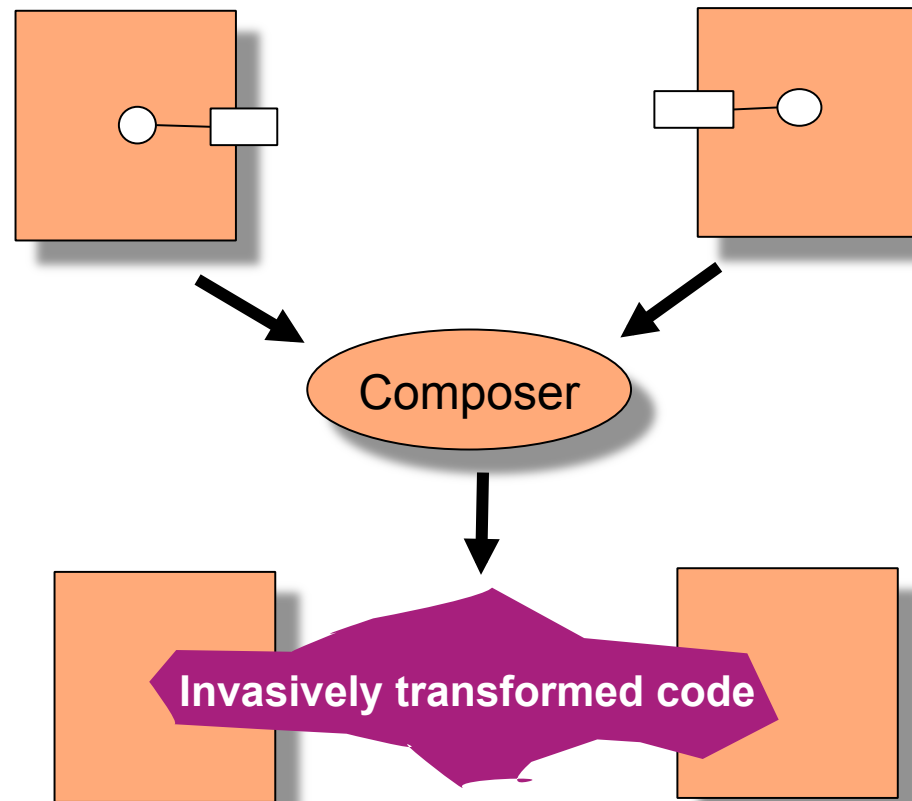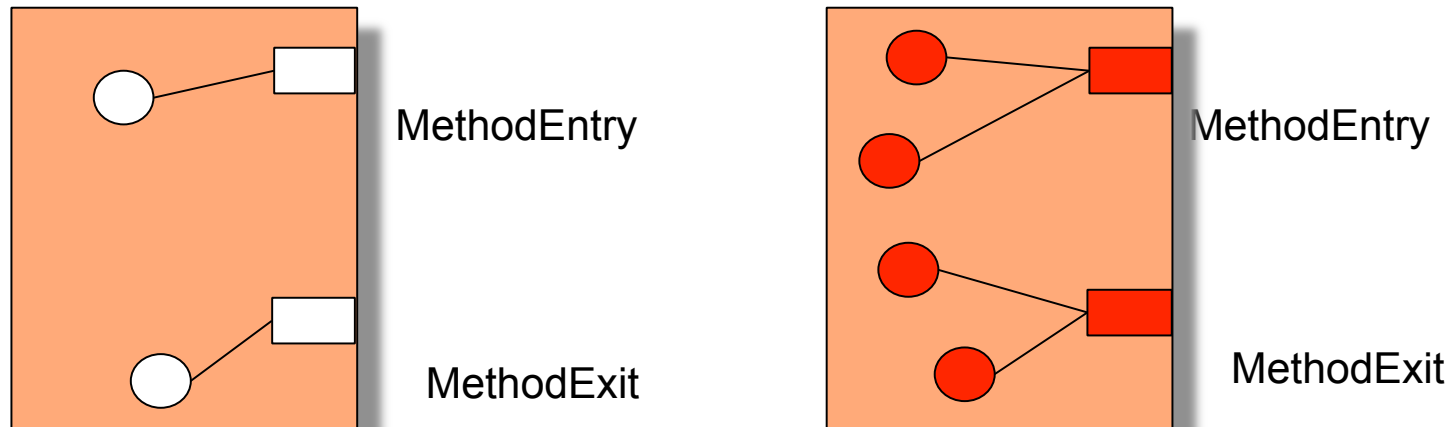
Bind Composer Parameterizes Fragment Components


```
<<mod:Modifier>>
m (){

    abc..
    <<mid:Statement>>
    cde..

}
```

```
synchronized m (){
    abc..
    f();
    cde..
}
```

component.findHook(„mod").bind("synchronized");

component.findHook(„mid").bind("f();");

Extend Operator Extends the Fragment Components

MethodEntry

MethodExit

```
m (){

    abc..
    cde..

}
```

MethodEntry

MethodExit

```
m (){
    print("enter m");
    abc..
    cde..
    print("exit m");
}
```

component.findHook(„MethodEntry").extend("print(\"enter m\");");

component.findHook(„MethodExit").extend("print(\"exit m\");");

Merge Operator

merge(Component C1, Component C2) :=

   extend(C1.list, C2.list)

where list is a list of inner components, inner fragments, etc.

On the Difference of Declared and Implicit Hooks

► Invasive composition unifies generic programming (BETA) and view-based programming (merge composition operators)

▪ By providing *bind* (parameterization) and *extend* for all language constructs

```
Hook h = methodComponent.findHook("MY");
if (parallel)
    h.bind("synchronized");
else
    h.bind(" ");
methodComponent.findHook("MethodEntry").bind("");
methodComponent.findHook("MethodExit").bind("");
```

```
/* @genericMYModifier */ public print() {

  // <<MethodEntry>>

    if (1 == 2)

        System.out.println("Hello World");

        // <<MethodExit>>

        return;

    else

        System.out.println("Bye World");

        // <<MethodExit>>

        return;

}
```

```
synchronized public print () {
        if (1 == 2)
            System.out.println("Hello World");
            return;
        else
            System.out.println("Bye World");
            return;
}
```
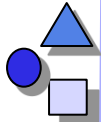
When Do you Need Invasive Composition

► When static relations have to be adapted

  ▪ Inheritance relationship: multiple and mixin inheritance

  ▪ Delegation relationship:;When delegation pointers have to be inserted

  ▪ Import relationship

  ▪ Definition/use relationships (adding a definition for a use)

► When physical unity of logical objects is desired

  ▪ No splitting of roles, but integration into one class

► When the resulting system should be highly integrated

Invasive Extension Integrates Feature Groups

▶ ... and roles, because a feature group can play a role

▶ The invasive extension lies between inheritance and delegation

K

Extend invasively   **Inherit**   **Delegate**

K   **K-subclass**   **K**   **K-private**
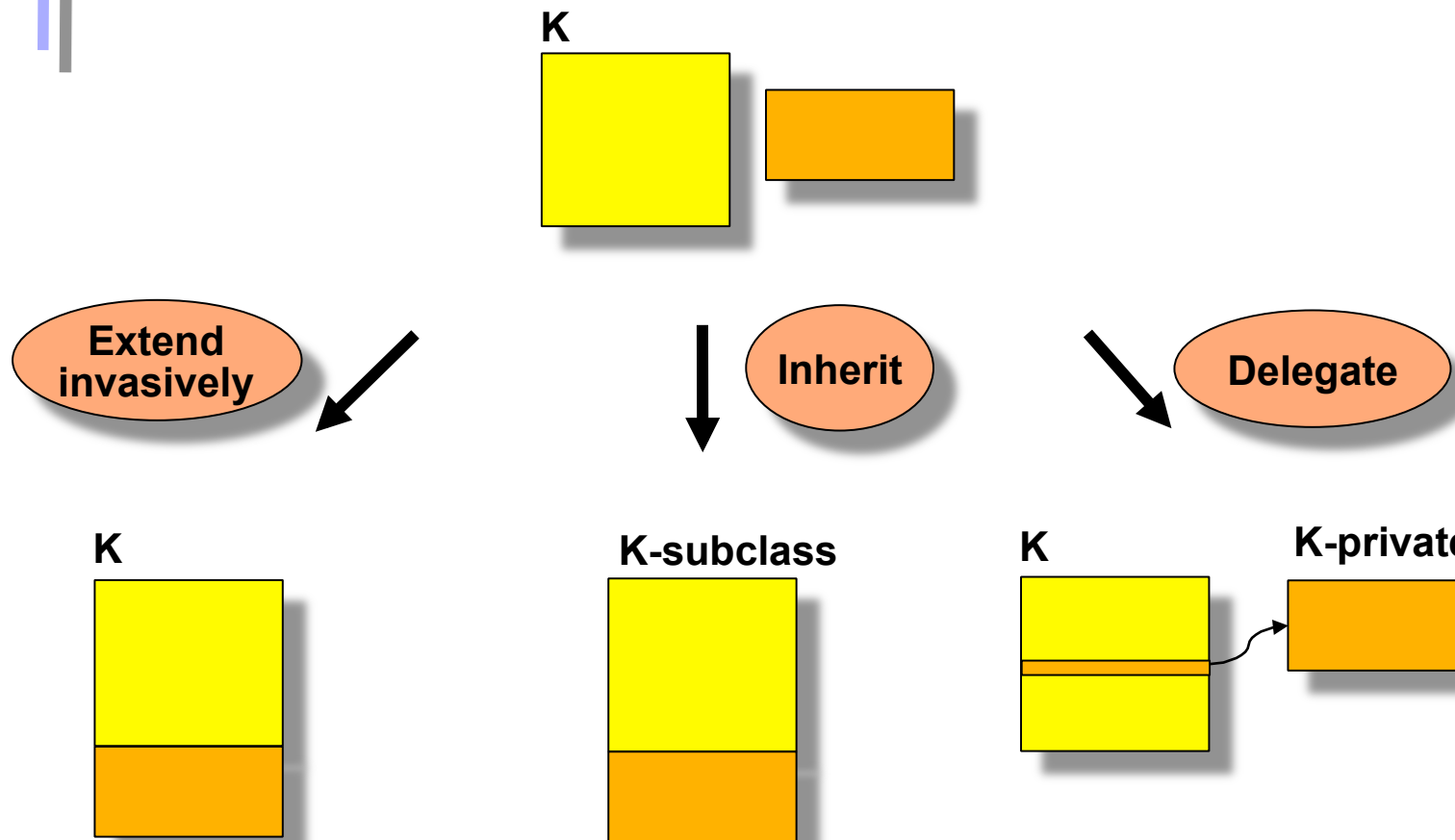
When To Use What?

- ► Deploy Inheritance
  - for consistent side-effect free composition

- ► Deploy Delegation
  - for dynamic variation
  - Suffers from object schizophrenia

- ► Deploy Invasive Extension
  - for non-foreseen extensions that should be *integrated*
  - to develop aspect-orientedly
  - to adapt without delegation
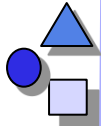
Invasive Composition

Adds a full-fledged composition language to generic and view-based programming

Combines architectural systems, generic, view-based and aspect-oriented programming
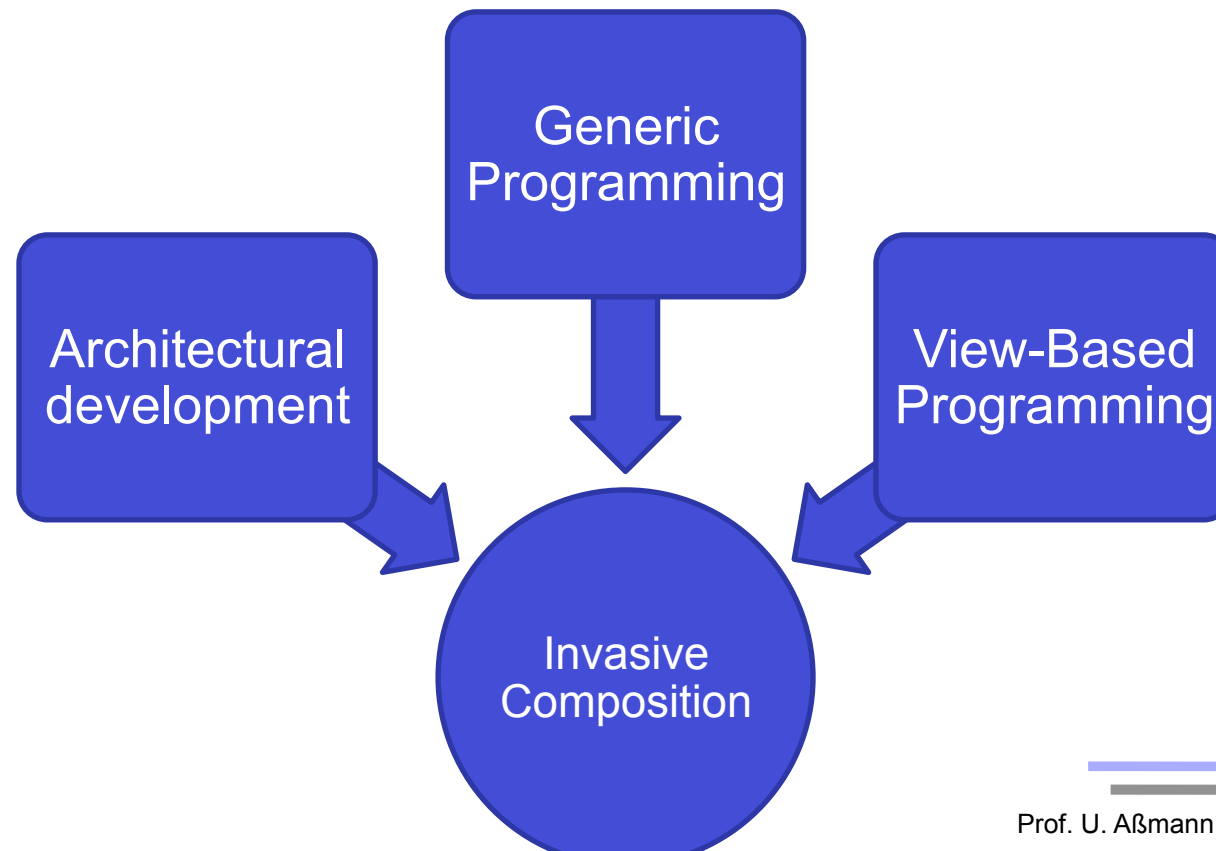
Composition Programs

Basically, every language may act as a composition language, if its basic operators are *bind* and *extend.*

Imperative languages: Java (used in COMPOST), C, ..

Graphical languages: boxes and lines (used in Reuseware)

Functional languages: Haskell

Scripting languages: TCL, Groovy, ...

Logic languages: Prolog, Datalog, F-Datalog

Declarative Languages: Attribute Grammars, Rewrite Systems

25.2) What Can You Do With Invasive Composition?
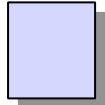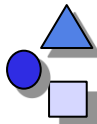
Universally Generic Programming

In contrast to BETA, ISC offers a full-fledged composition language

T

```
<< ClassBox >>


class SimpleList {
  genericTType elem;
  SimpleList next;
  genericTType getNext() {
     return next.elem;
  }
}
```

➡

```
<< ClassBox >>


class SimpleList {
  WorkPiece elem;
  SimpleList next;
  WorkPiece getNext() {
     return next.elem;
  }
}
```
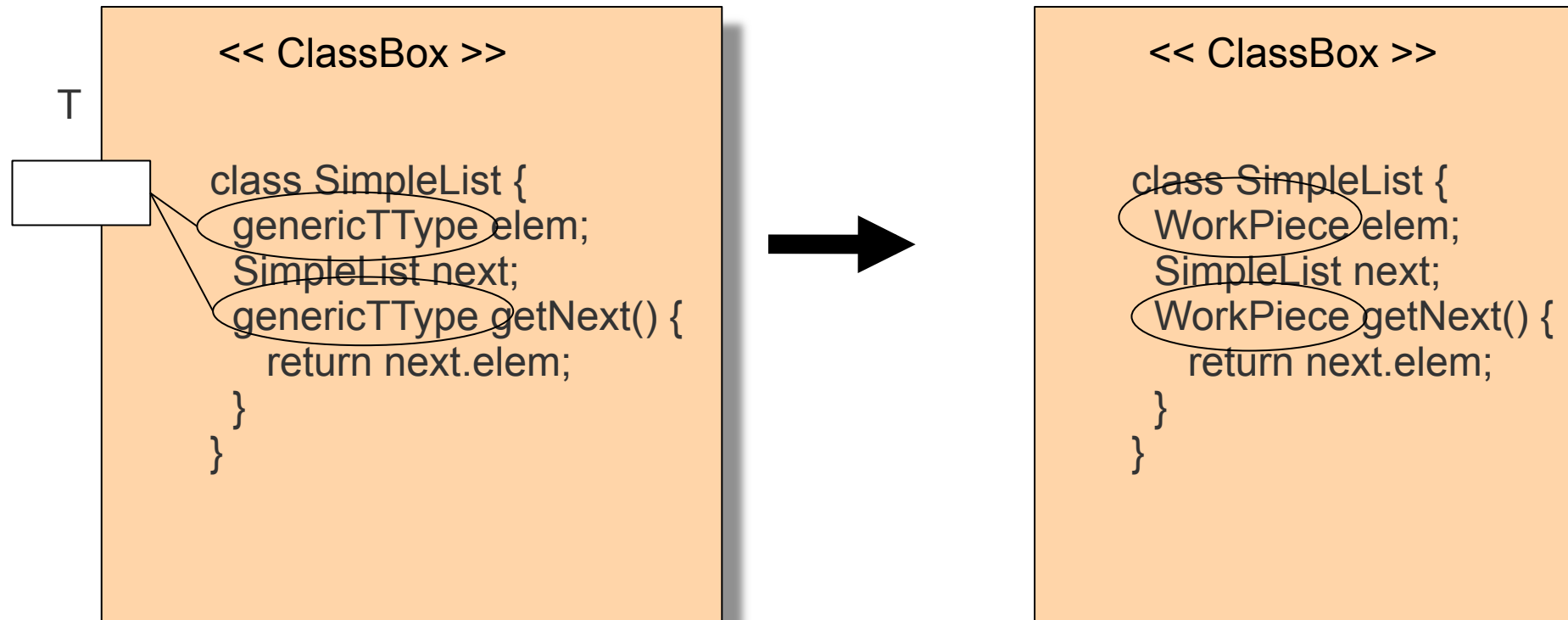
Invasive Connections

In contrast to ADL, ISC offers invasive connections

Client

Library

Blackbox Composition

Invasive Composition

Client **Library**

Client **Library**

Blackbox connection with glue code

Invasive Connection
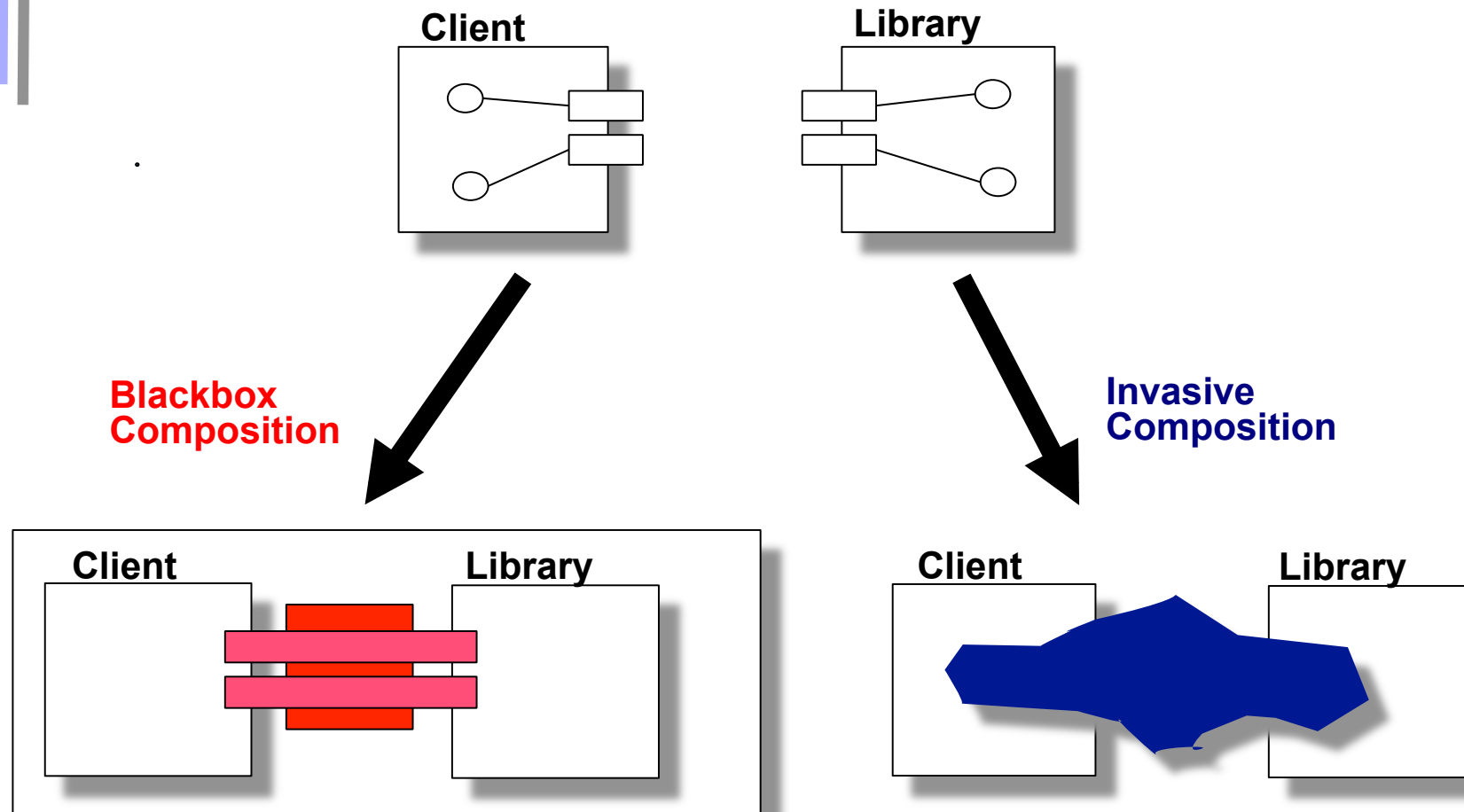
Mixin Inheritance

▶ Extension can be used for inheritance (mixins)

▶ In contrast to OO languages, ISC offers tailored inheritance operations, based on the extend operator
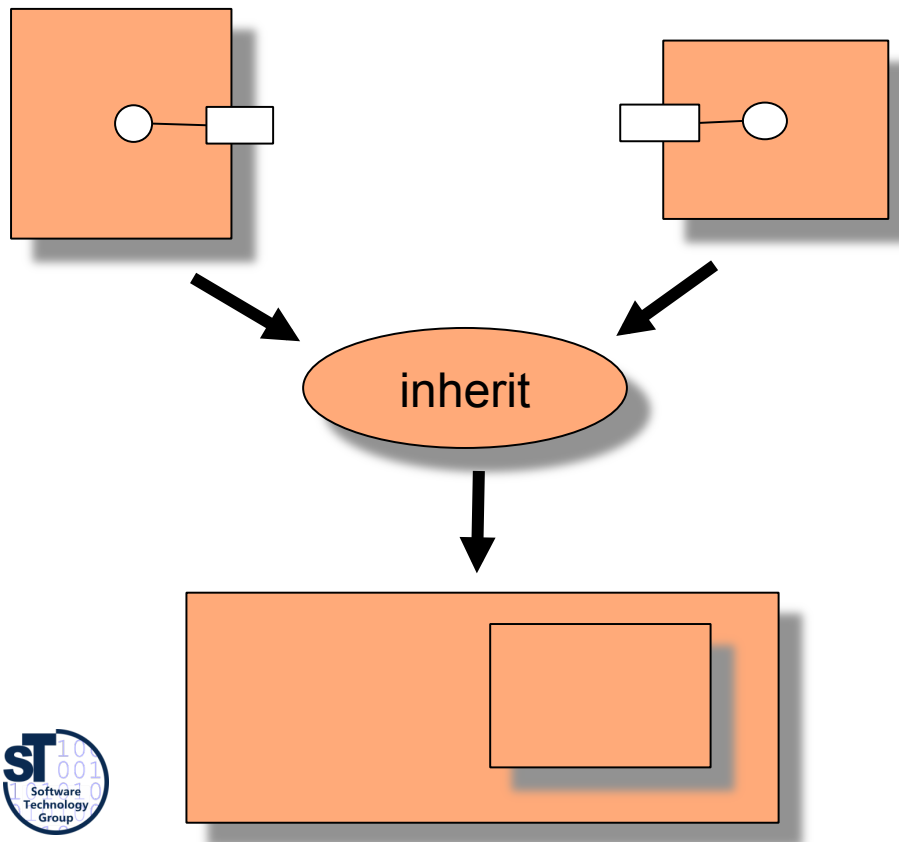
- **inheritance :=**
  - copy first super class
  - extend with second super class

Mixin Inheritance Works Uniformly for Languages that don't have it
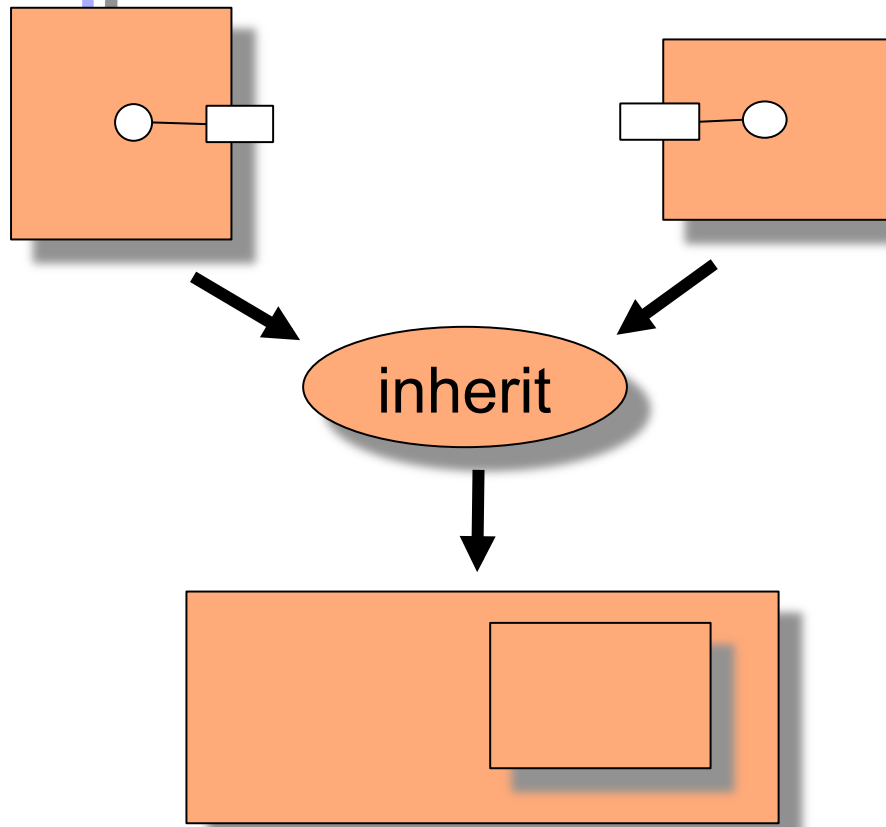
- ▶ Invasive composition can model mixin inheritance uniformly for all languages
- ▶ e.g., for XML
- ▶ inheritance :=
  - copy first super document
  - extend with second super document

Invasive Document Composition for XML

▶ Invasive composition can be used for document languages, too [Hartmann2011]

▶ Example List Entry/Exit of an XML list

▶ Hooks are given by the Xschema

<UL>

List.entry ⟶

    <LI>... </LI>
    <LI>... </LI>

List.exit ⟶

</UL>
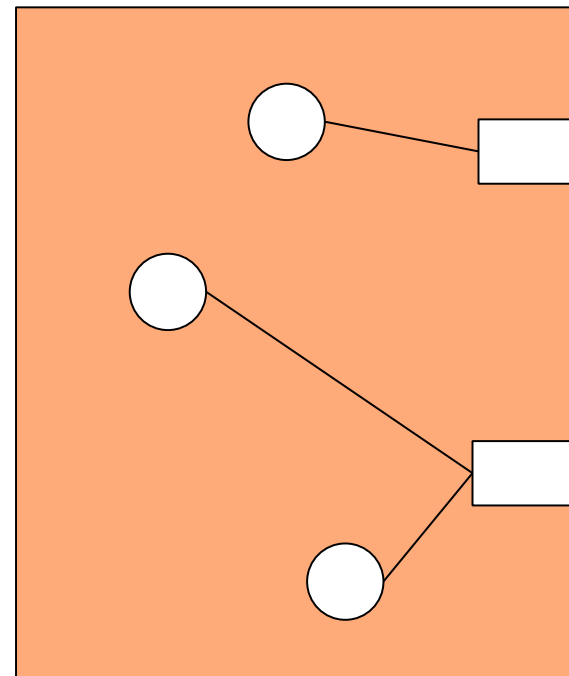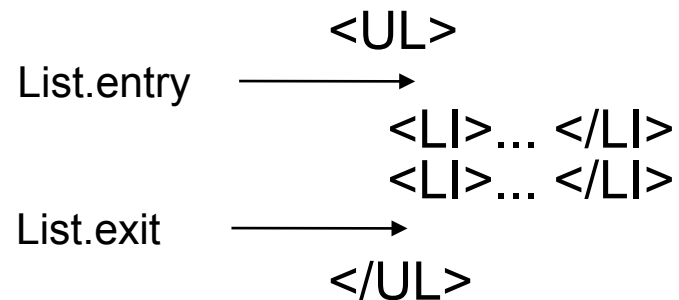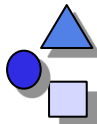
Hook Manipulation for XML

List Entry

List Exit

List Entry

List Exit

<UL>

  <LI>... </LI>
  <LI>... </LI>

</UL>

<UL>
  <LI>... </LI>
  <LI>... </LI>
  <LI>... </LI>
  <LI>... </LI>
</UL>
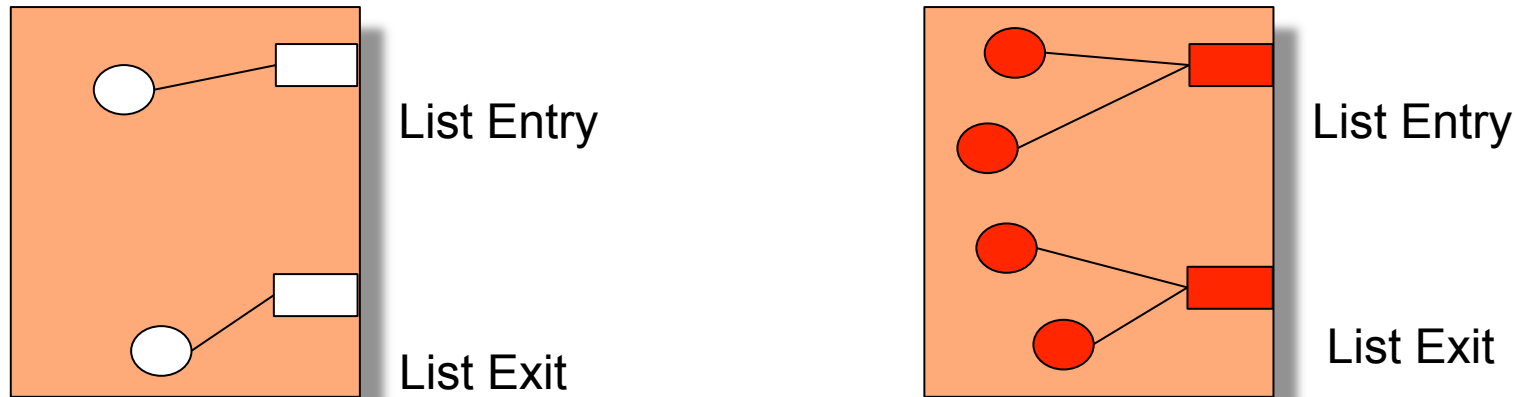
XMLcomponent.findHook(„ListEntry").extend(„<LI>... </LI>");

XMLcomponent.findHook(„ListExit").extend("<LI>... </LI>");

Composers can be Used as Weavers in AOP (Core and Aspect Components)

Core

Aspect
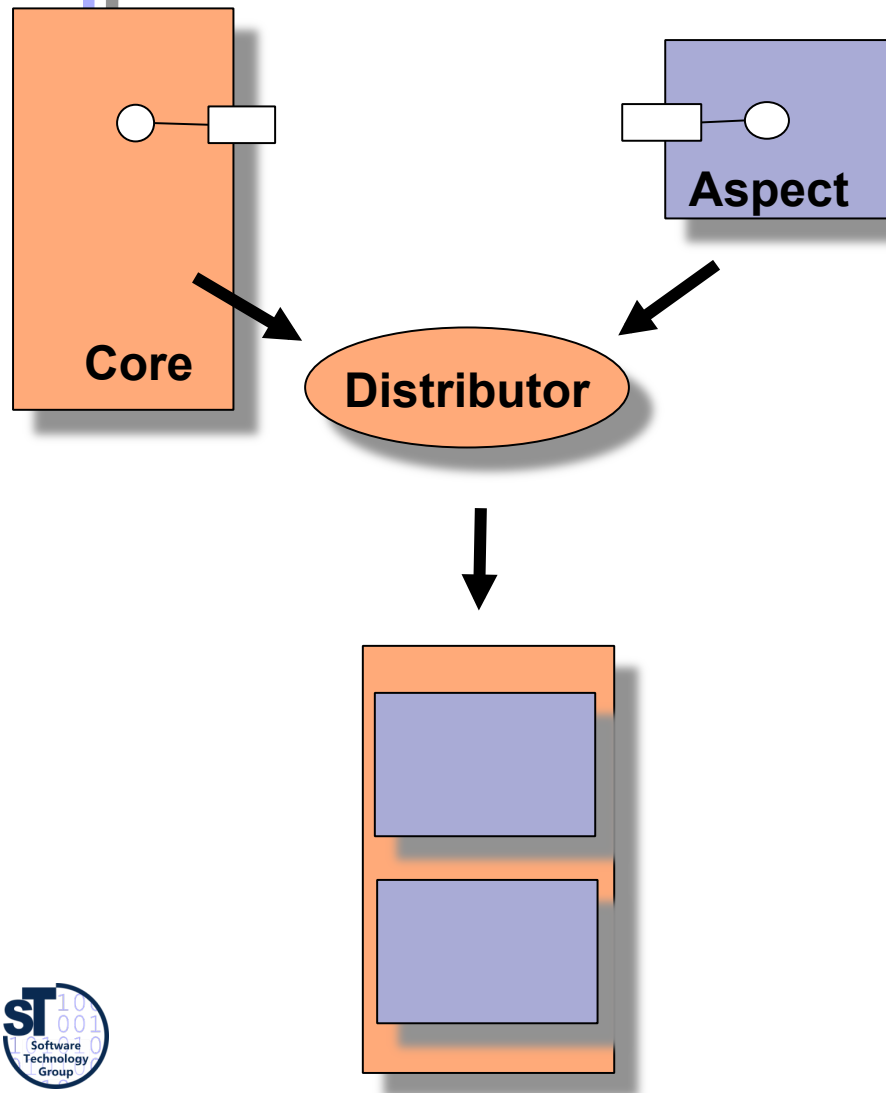
Distributor

- ► Complex composers distribute aspect fragments over core fragments
- ► *Distributors* extend the core
- ► Distributors are more complex operators, defined from basic ones
- ► Static aspect weaving can be descirbed by distributors

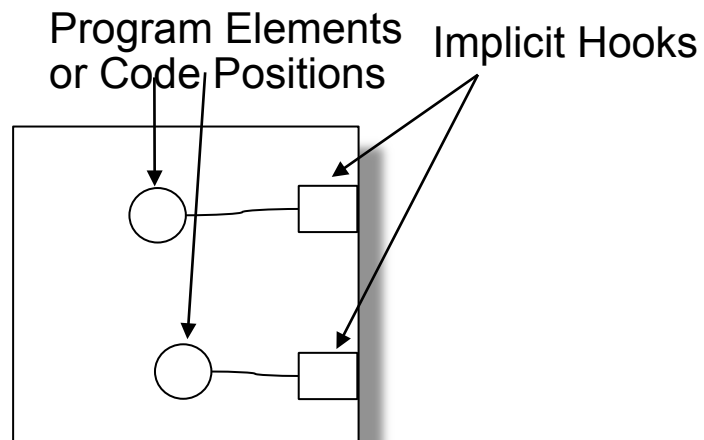# *Invasive Model Composition with Reuseware*

25.3) Composition and Functional Interfaces
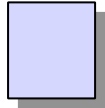
Composition vs Functional Interfaces

Composition interfaces contain hooks and slots

  static, based on the component model at design time

Functional interfaces are based on the component model at run time and contain slots and hooks of it

Program Elements or Code Positions   Implicit Hooks

Declared Hooks

Output Port

Input Port

Provided Method

Required Method
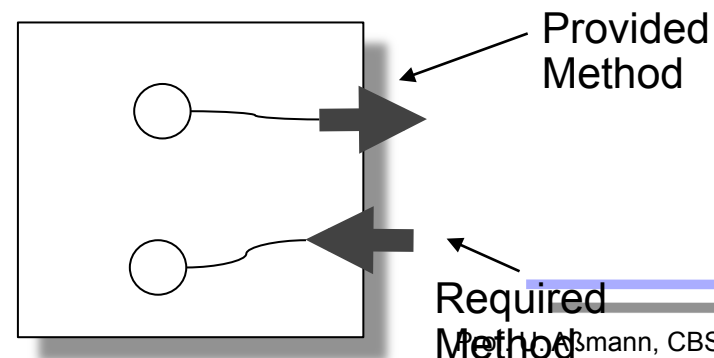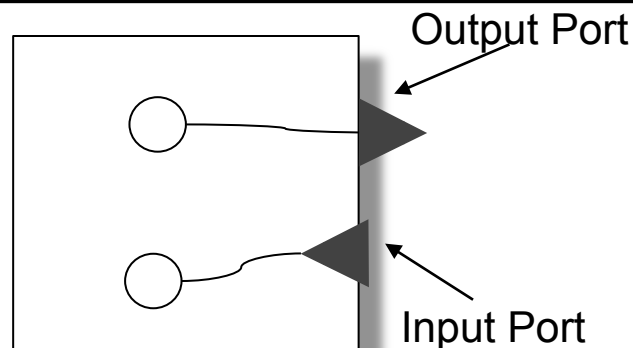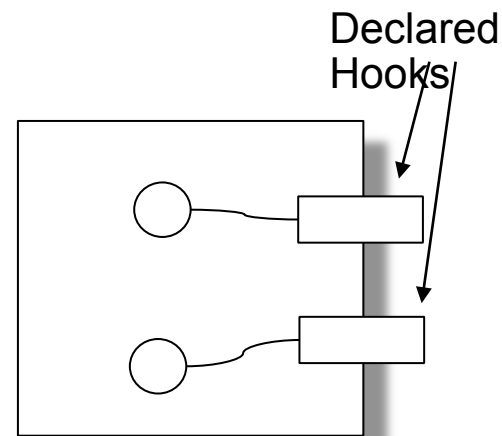
# *Functional Interfaces are Generated from Composition Interfaces*

2-stage process

Composition Interface (Boxes with Declared Hooks)

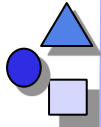Functional Interface (Classes or Modules with Methods)
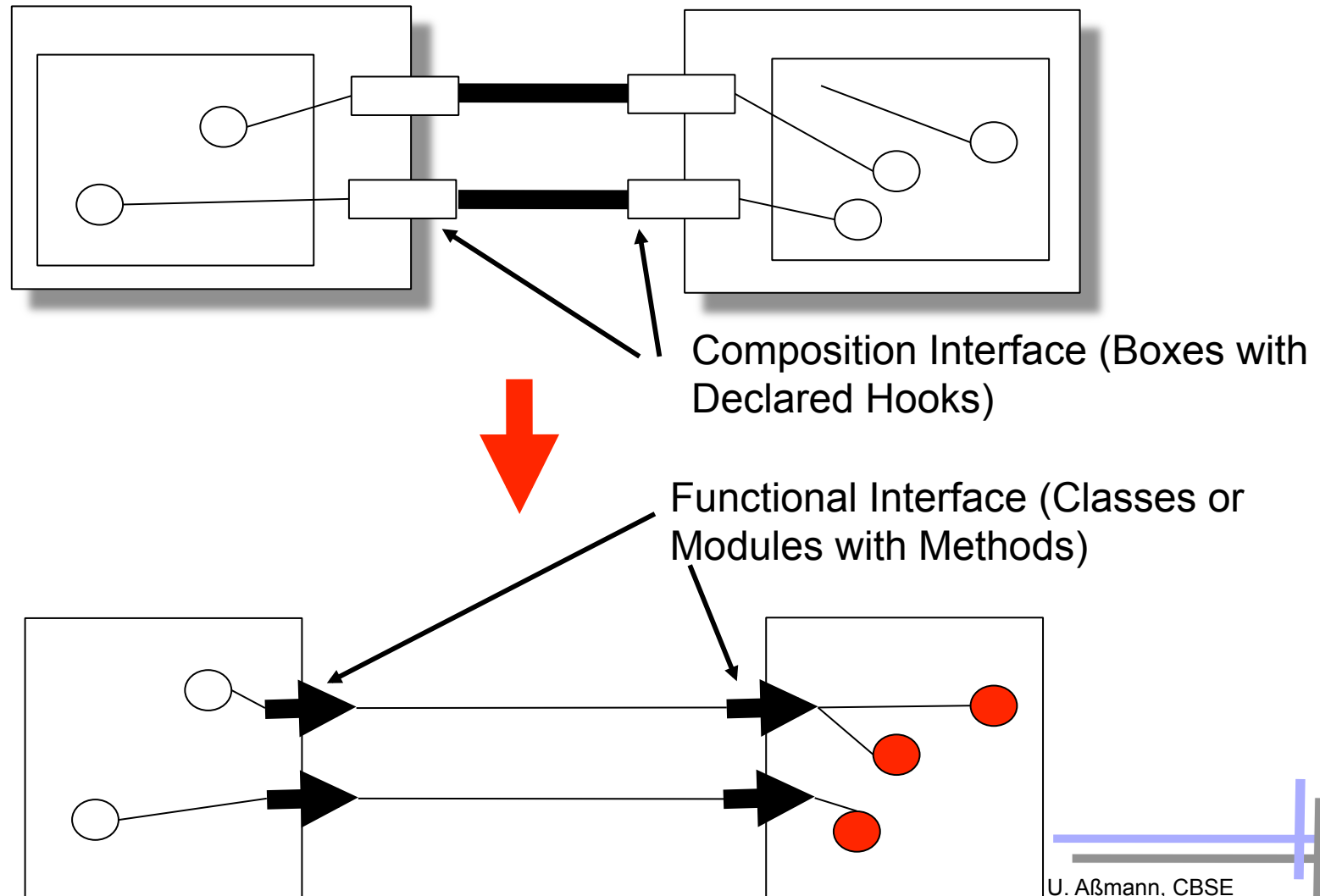
Execution of a Composition Program

► transforms a set of fragment components step by step, binding their composition interfaces (filling their slots and hooks), resulting in an integrated program with functional interfaces
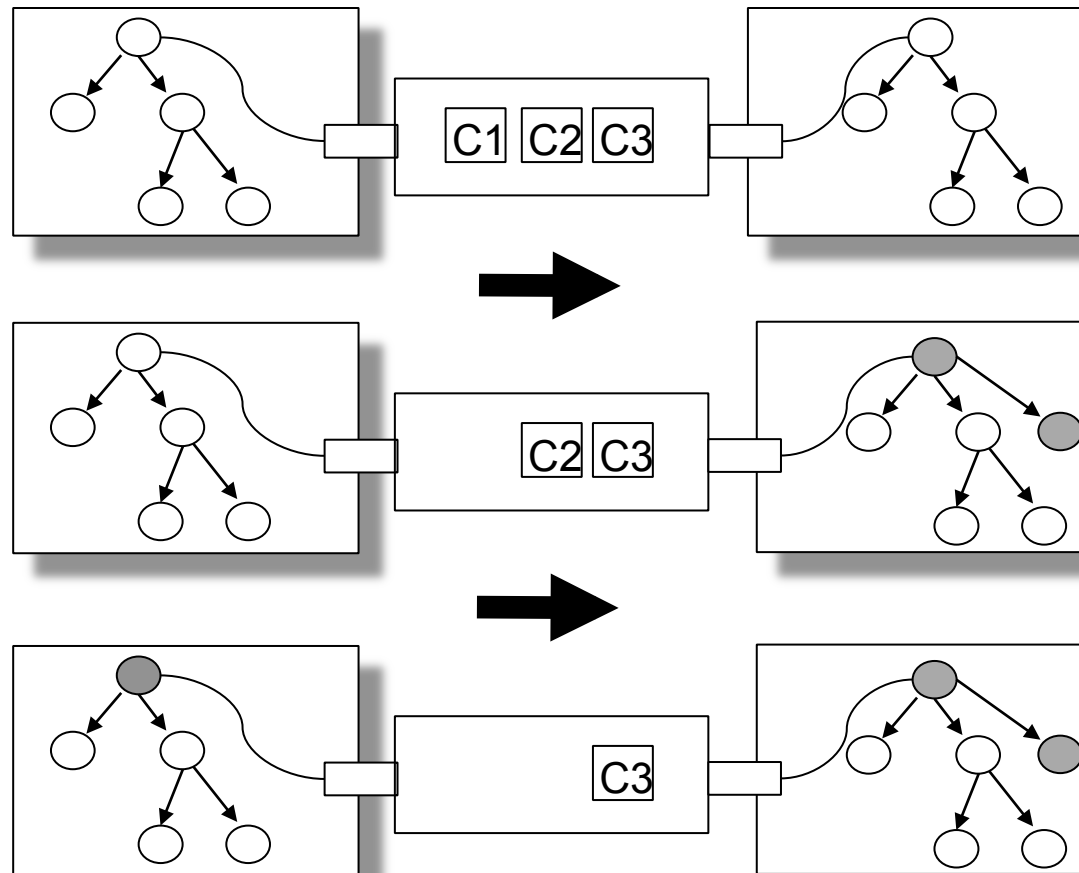
25.4) Different Forms of Greyboxes (Shades of Grey)
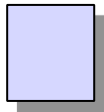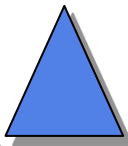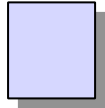
Invasive Composition and Information Hiding

► Invasive Composition modifies components at well-defined places during composition

- There is less information hiding than in blackbox approaches

- But there is...

- ... that leads to greybox components

Refactoring is a Whitebox Operation

► Refactoring works directly on the AST/ASG

► Attaching/removing/replacing fragments

► Whitebox reuse

Refactorings Transformations
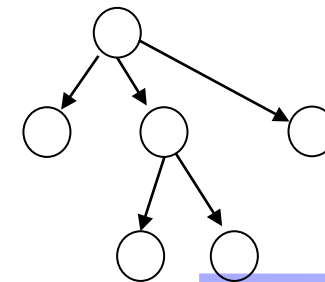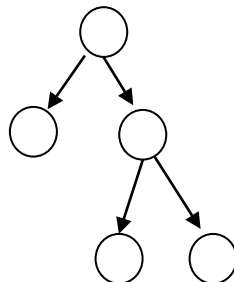
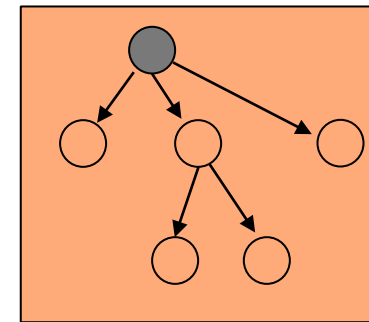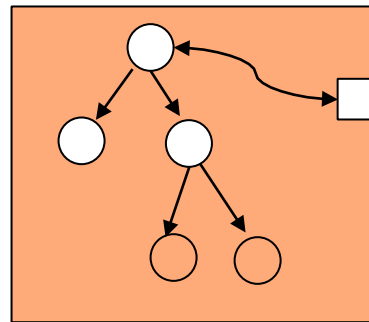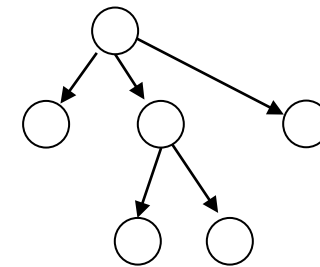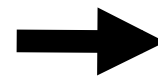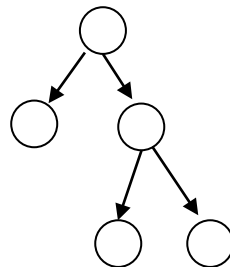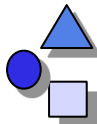# Modifying Implicit Hooks is a Light-Grey Operation

- ► Aspect weaving and view composition works on implicit hooks *(join points)*

- ► Implicit composition interface

Composition with implicit hooks
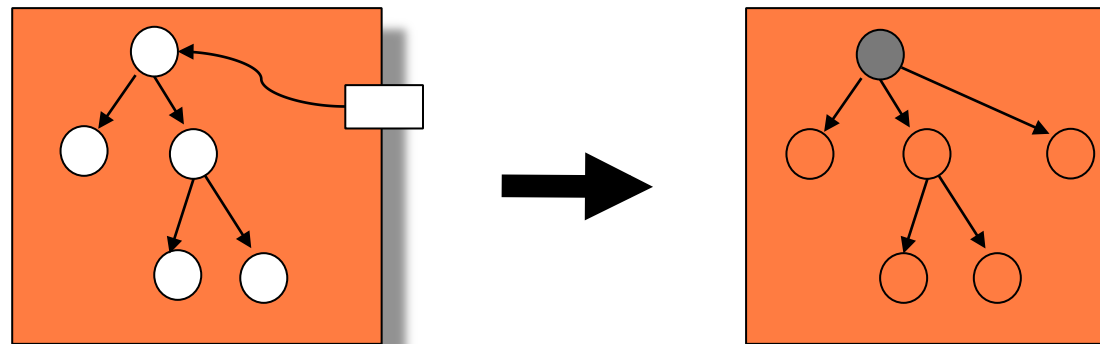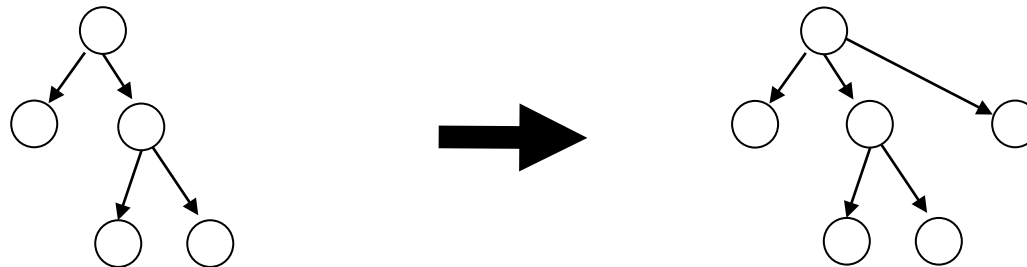

Refactorings Transformations

Parameterization as Darker-Grey Operation

- ► Templates work on *declared hooks*

- ► *Declared composition interface*

**Composition
with declared
hooks**
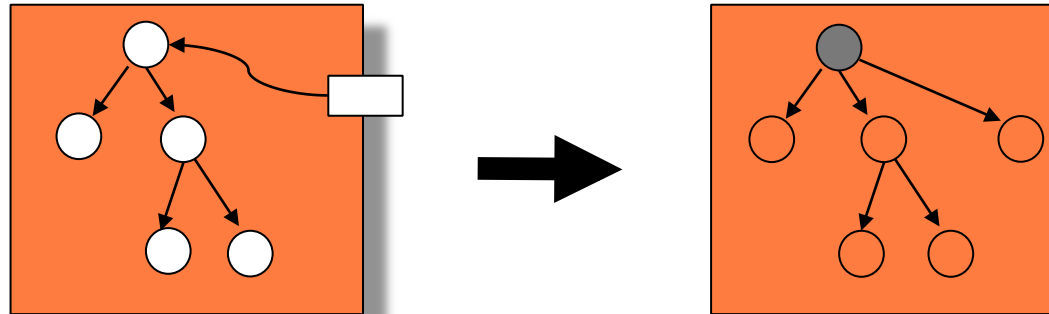

**Refactorings
Transformations**

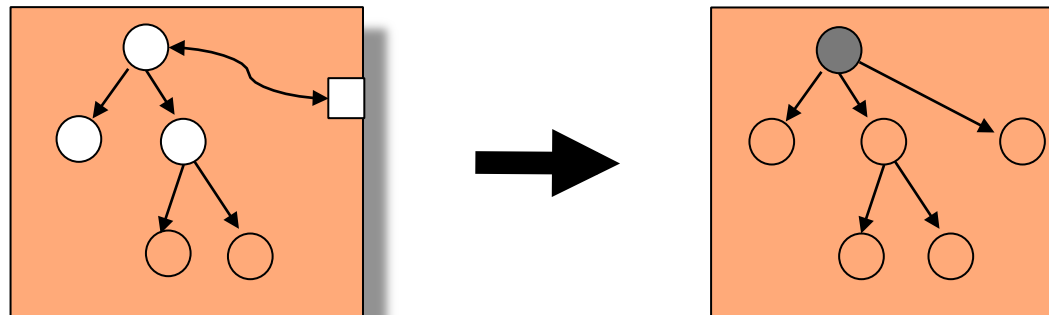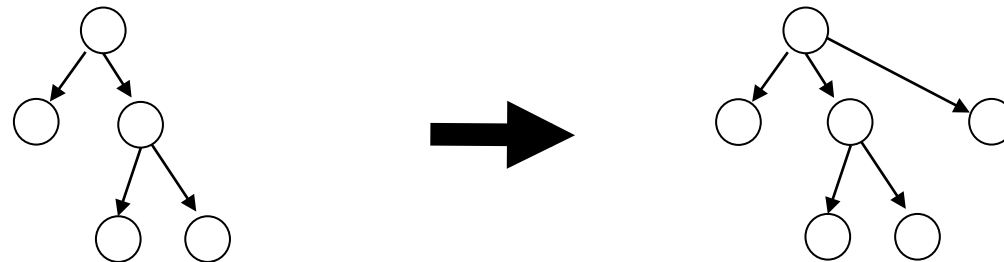# Systematization Towards Greybox Component Models

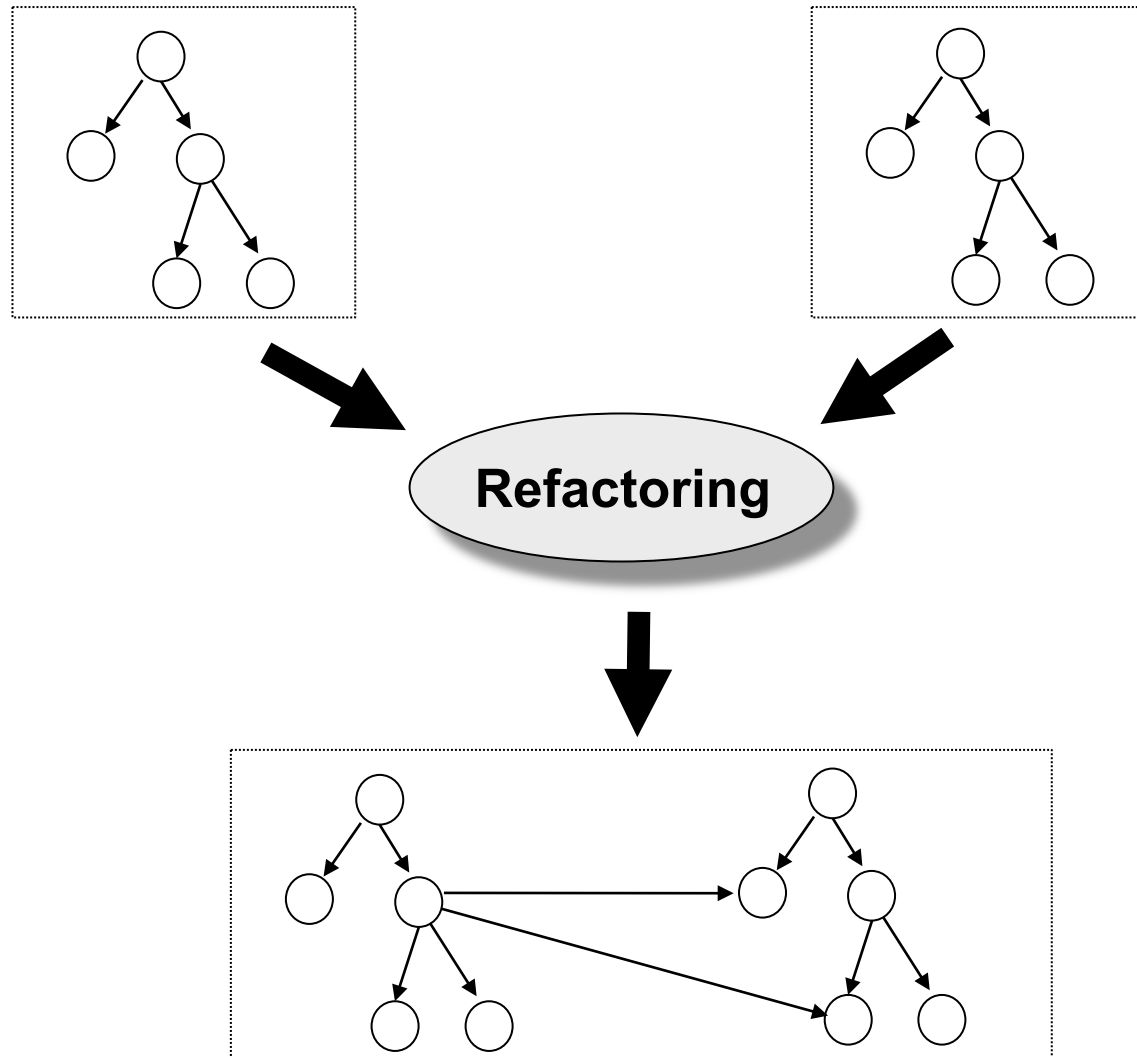**Composition with declared hooks**
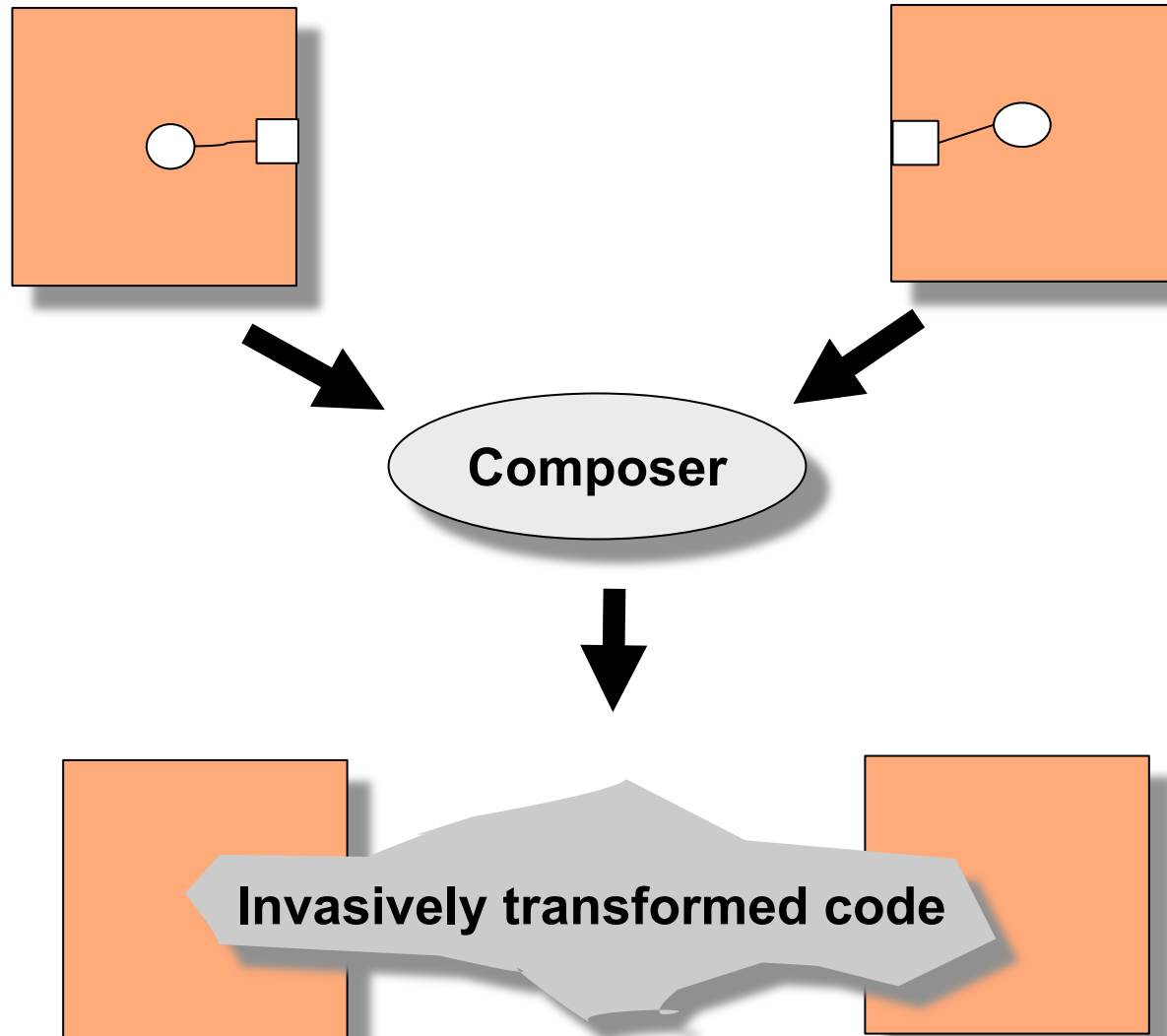

Composition with implicit hooks

Refactorings Transformations

Refactoring Builds On Transformation Of Abstract Syntax

Invasive Composition Builds On Transformation Of Implicit Hooks

Invasive Composition Builds On Transformation on Declared Hooks
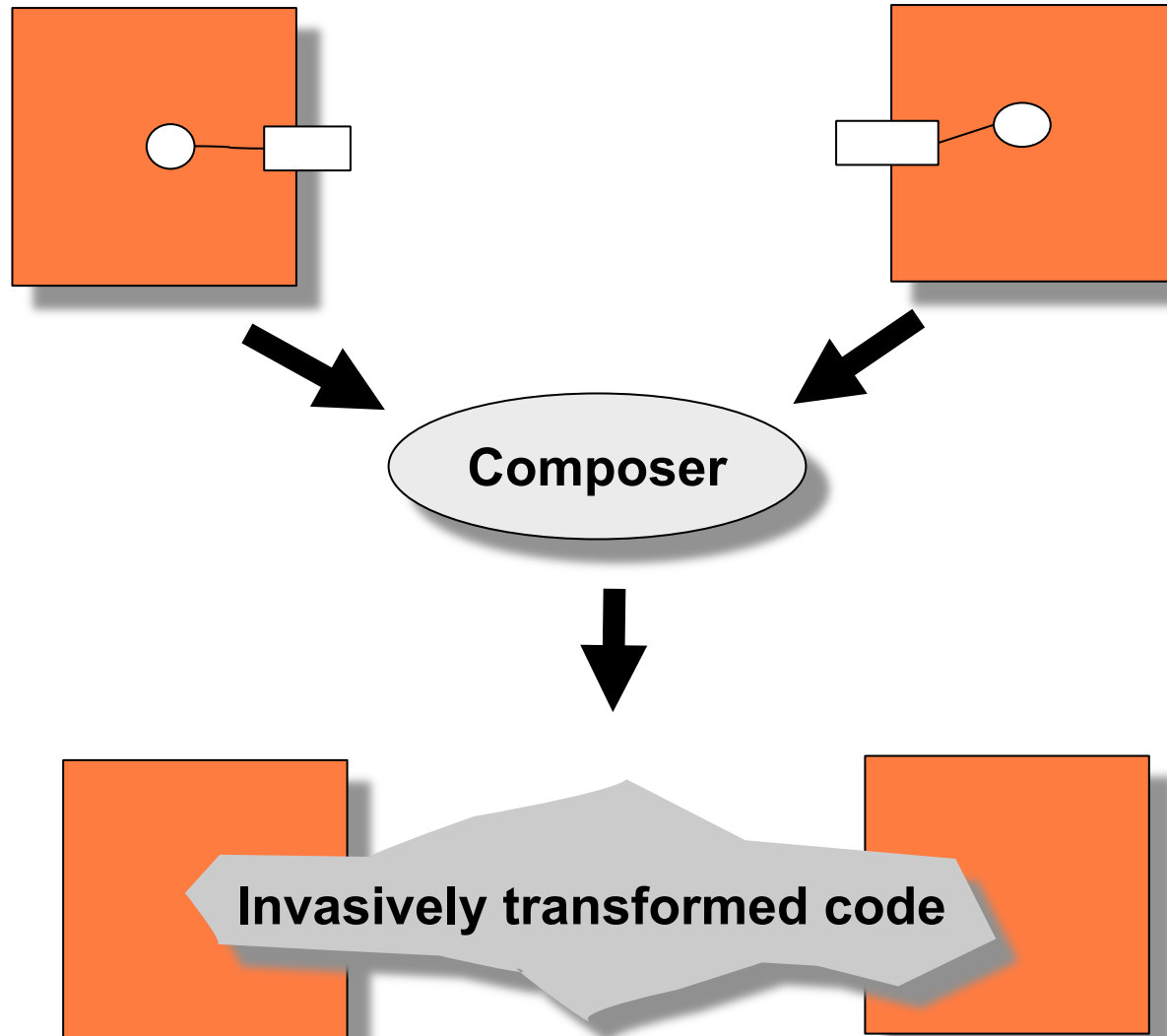
25.5 Invasive Software Composition as Composition Technique

Invasive Composition: Component Model

► Fragment components are graybox components

- Composition interfaces with declared hooks

- Implicit composition interfaces with implicit hooks

- The composition programs produce the functional interfaces

  - Resulting in efficient systems, because superfluous functional interfaces are removed from the system

- Content: source code

  - binary components also possible, poorer metamodel

► Aspects are just a special type of component

► Fragment-based parameterisation a la BETA

- Type-safe parameterization on all kinds of fragments
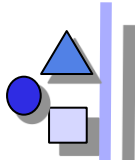
Invasive Composition: Composition Technique

► Adaptation and glue code: good, composers are program transformers and generators

► Aspect weaving

  ▪ Parties may write their own weavers

  ▪ No special languages

► Extensions:

  ▪ Hooks can be extended

  ▪ Soundness criteria of lambdaN still apply

  ▪ Metamodelling employed

► Not yet scalable to run time

Composition Language

► Various languages can be used

► Product quality improved by metamodel-based typing of compositions

► Metacomposition possible

  ▪ Architectures can be described in a standard object-oriented language and reused

► An *assembler* for composition

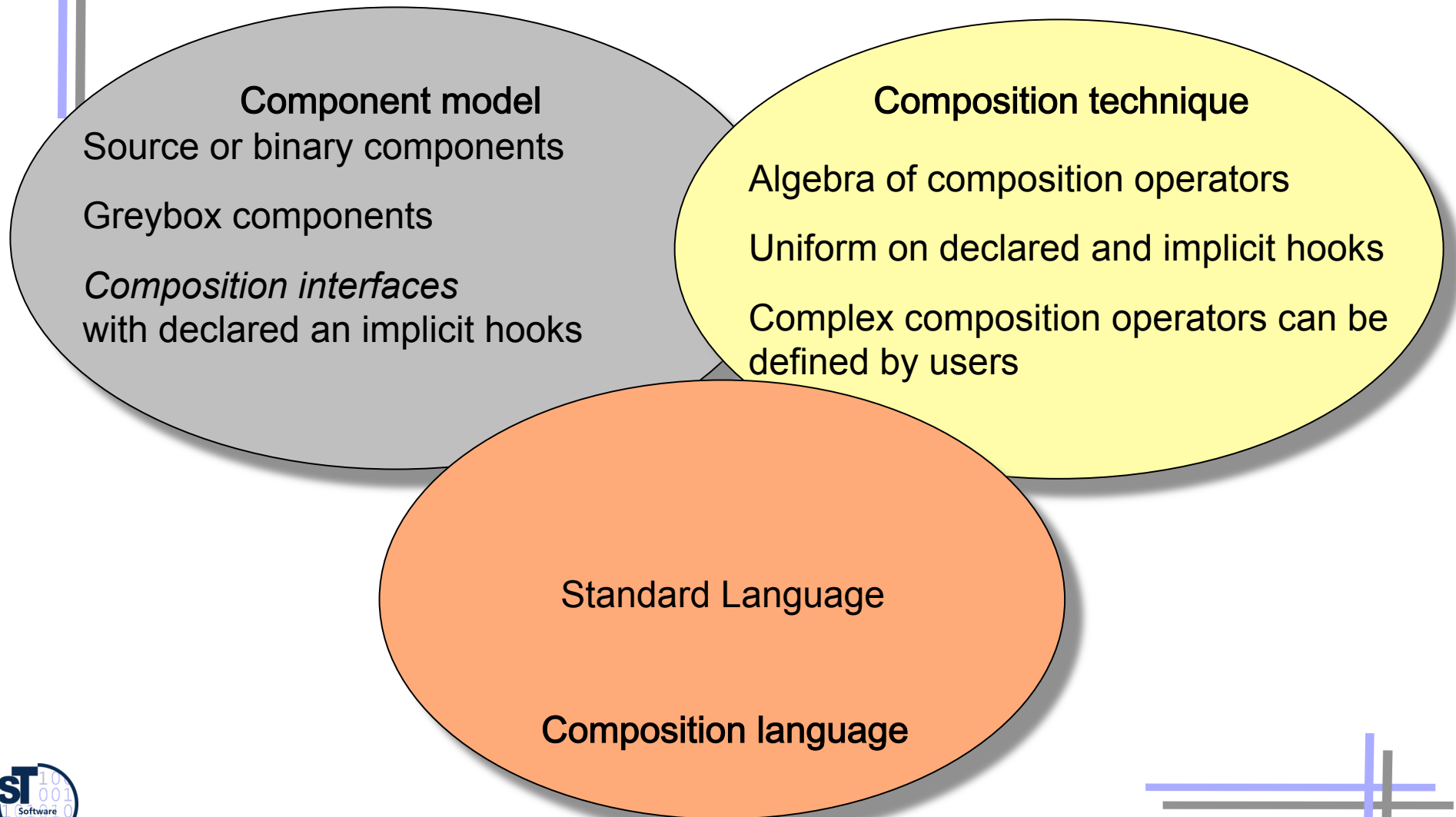 ▪ Other, more adequate composition languages can be compiled

Conclusions for ISC

- ▶ Fragment-based composition technology
  - Graybox components
  - Producing tightly integrated systems

- ▶ Components have *composition interface*
  - From the composition interface, the functional interface is derived
  - Composition interface is different from functional interface
  - Overlaying of classes (role model composition)

- COMPOST framework showed applicability of ISC for Java
  - (ISC book)

- Reuseware Composition Framework extends these ideas
  - For arbitrary grammar-based languages
  - For metamodel-based languages

- http://reuseware.org

Invasive Composition as Composition System

Component model

Source or binary components

Greybox components

Composition interfaces
with declared an implicit hooks

Composition technique

Algebra of composition operators

Uniform on declared and implicit hooks

Complex composition operators can be defined by users

Standard Language

Composition language

What Have We Learned

► With the uniform treatment of declared and implicit hooks, several technologies can be unified:

- Generic programming
- Connector-based programming
- Refactorings
- Inheritance-based programming
- View-based programming
- Aspect-based programming

The End