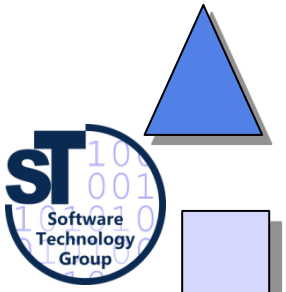
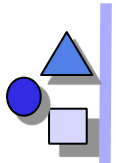
26) Invasive Software Composition (ISC)

Prof. Dr. Uwe Aßmann
Florian Heidenreich
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de
Version 12-1.0, Juni 19, 2012

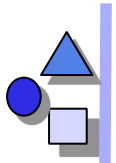


1. Invasive Software Composition - A Fragment-Based Composition Technique
2. What Can You Do With Invasive Composition?
3. Functional and Composition Interfaces
4. Different forms of grey-box components
5. Evaluation as Composition Technique



Obligatory Literature

- ISC book Chap 4
- www.the-compost-system.org
- www.reuseware.org



Other References

[AG00] Uwe Aßmann, Thomas Genßler, and Holger Bär. Meta-programming Grey-box Connectors. In R. Mitchell, editor, Proceedings of the International Conference on Object-Oriented Languages and Systems (TOOLS Europe). IEEE Press, Piscataway, NJ, June 2000.

[HLLA01] Dirk Heuzeroth, Welf Löwe, Andreas Ludwig, and Uwe Aßmann. Aspect-oriented configuration and adaptation of component communication. In J. Bosch, editor, Generative Component-based Software Engineering (GCSE), volume 2186 of Lecture Notes in Computer Science. Springer, Heidelberg, September 2001.

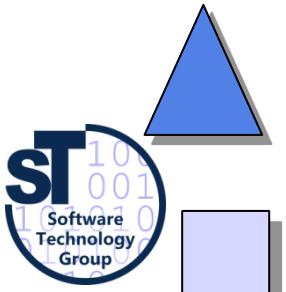
Jakob Henriksson. A Lightweight Framework for Universal Fragment Composition. Technische Universität Dresden, Dec. 2008 http://nbn-resolving.de/urn:nbn:de:bsz:14-ds-1231261831567-11763

Jendrik Johannes. Component-Based Model-Driven Software Development. Technische Universität Dresden, Dec. 2010 http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-63986

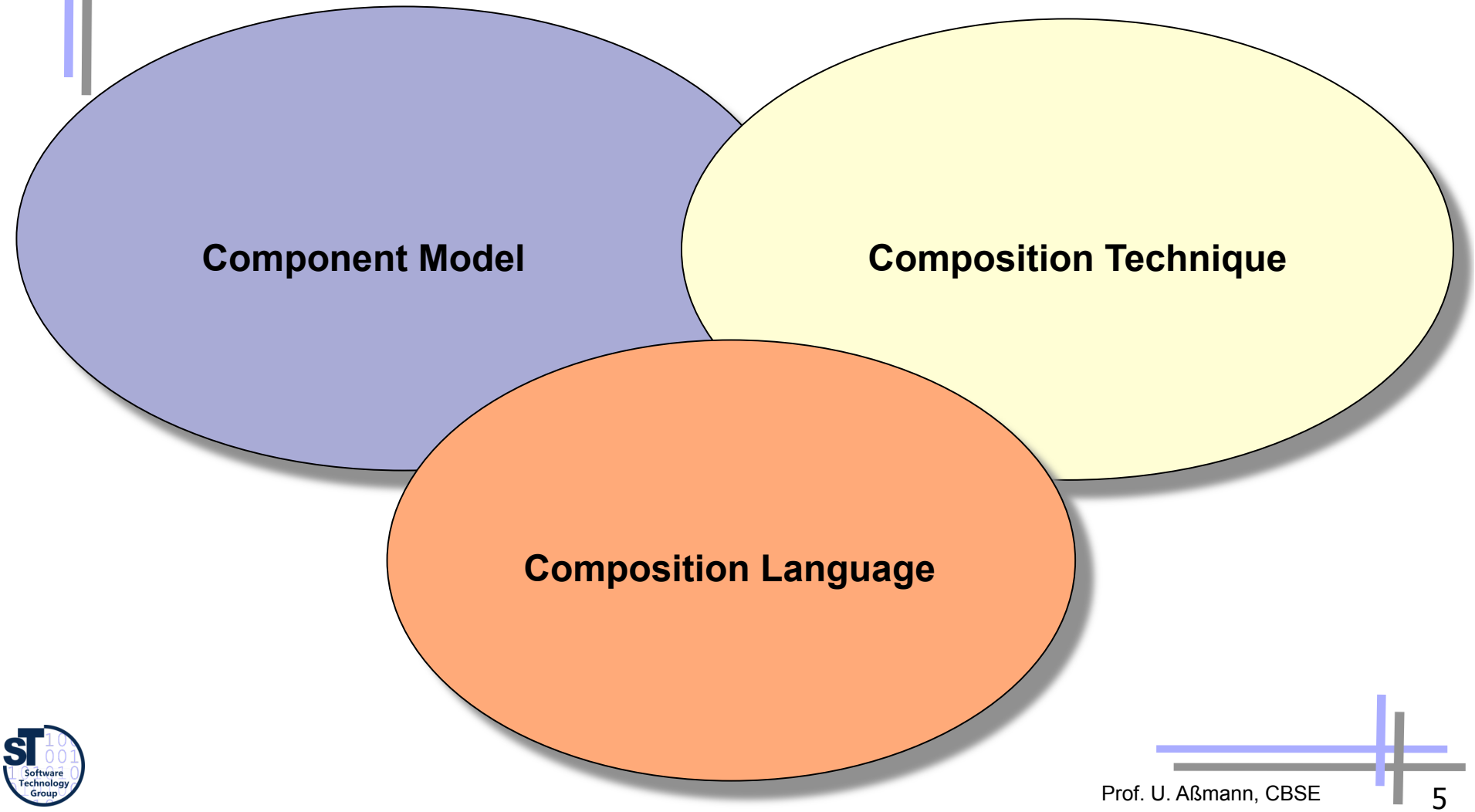
Jendrik Johannes and Uwe Aßmann, Concern-Based (de)composition of Model-Driven Software Development Processes. Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, 2010,Part II, Springer, 2010, LNCS 6395, URL = http://dx.doi.org/10.1007/978-3-642-16129-2

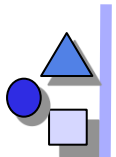
Falk Hartmann. Safe Template Processing of XML Documents. PhD thesis. Technische Universität Dresden, July 2011.

26.1) Invasive Software Composition - A Fragment-Based Composition Technique



Software Composition



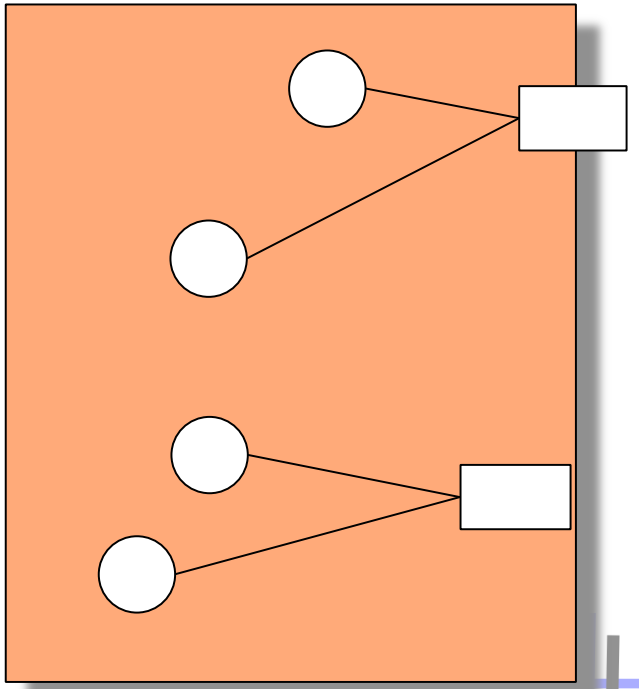


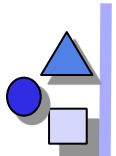
Invasive Software Composition

Invasive software composition parameterizes and extends fragment components at change points (hooks and slots) by transformation

- A **fragment component** is a fragment group (fragment container, fragment box) with a **composition interface** of **change points**

- Uniform container 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)

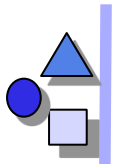




The Component Model of Invasive Composition

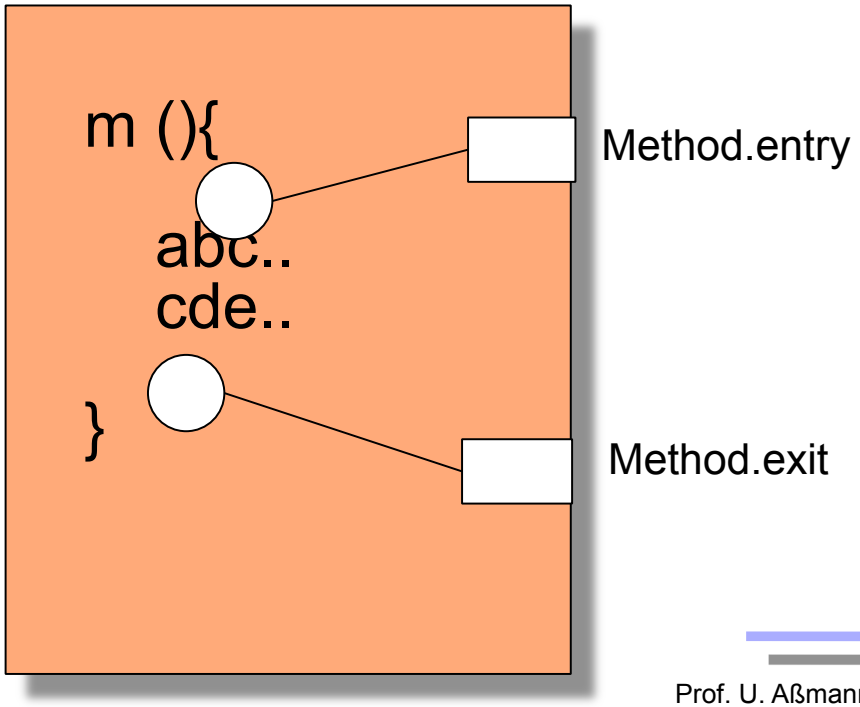
Change points of a fragment component are fragments or positions, which are subject to change

- Fragment components have 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



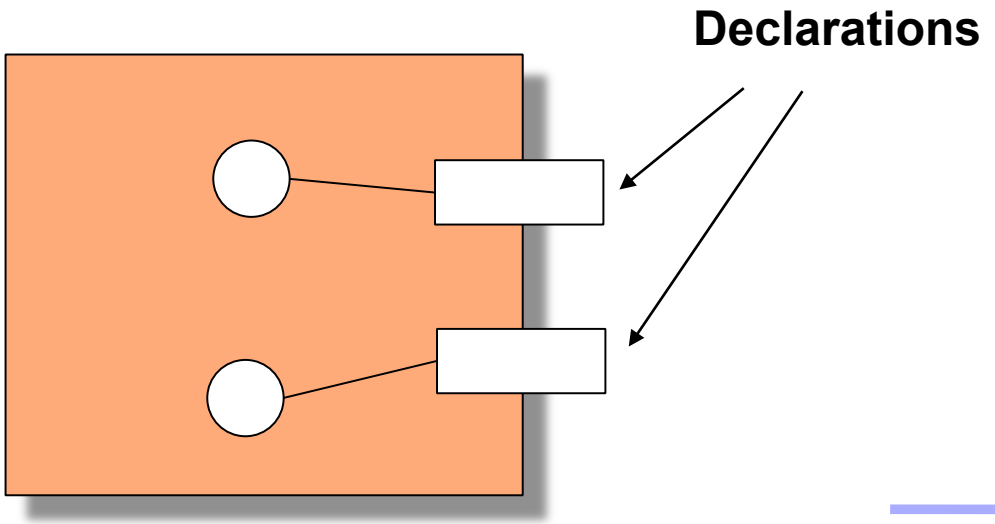
Hooks

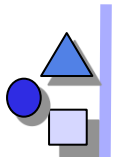
- A **hook (extension point)** is given by the component's language
- Hooks can be *implicit* or *explicit (declared)*
  - We draw implicit hooks *inside* the component, at the border
- Example: Method Entry/Exit



Slots (Declared Hooks)

- A **slot** is a variation point of a component, i.e., a code parameter
- Slots are most often *declared,* i.e., declared or explicit hooks, which must be declared by the component writer
  - They are implicit only if they designate one single program element in a fragment
  - We draw slots as crossing the border of the component
- Between slots and their positions in the code, there is a **slot-fragment mapping**





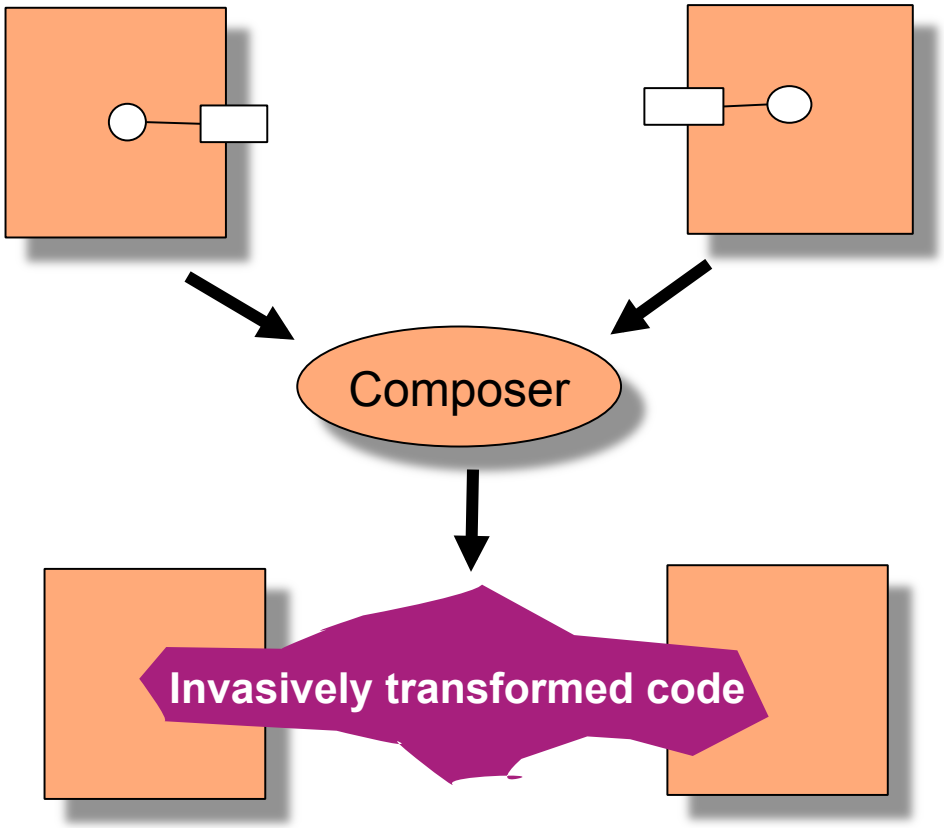
The Composition Technique of Invasive Composition

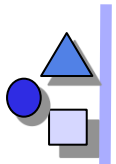
Invasive Software Composition
parameterizes and extends
fragment components
at implicit and declared hooks and slots
by transformation

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)

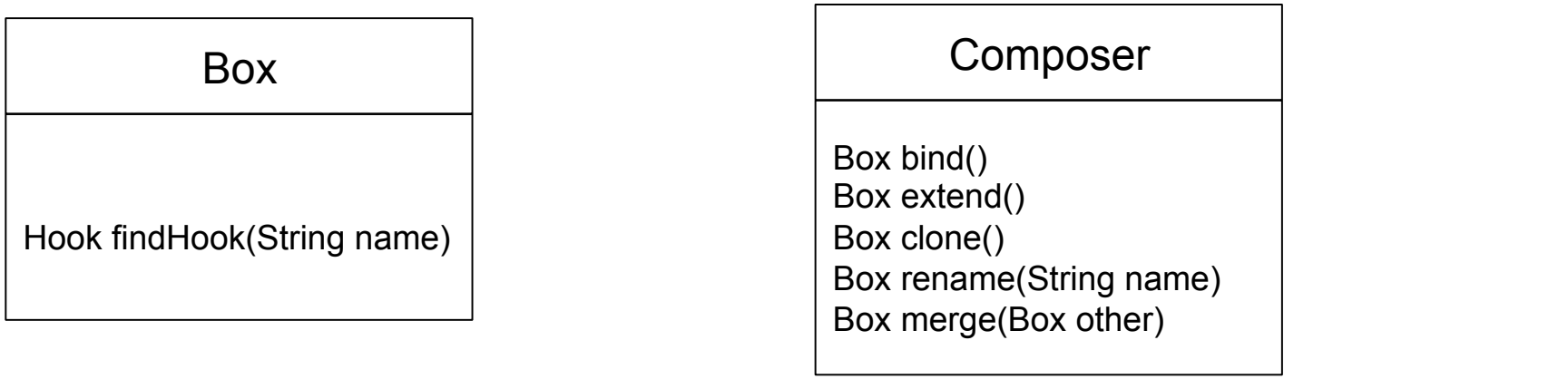


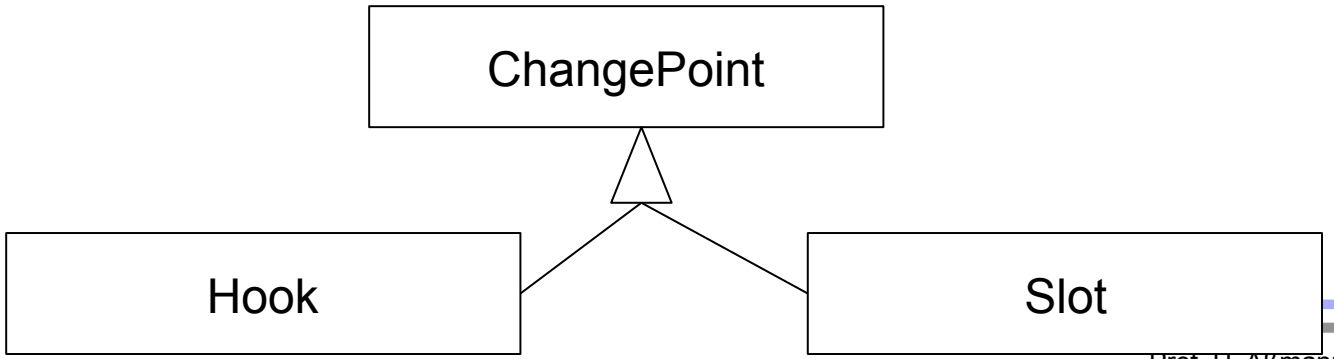


Object-Oriented Metamodeling of Composers

In the following, we assume an object-oriented metamodel of fragment components, composers, and composition languages.

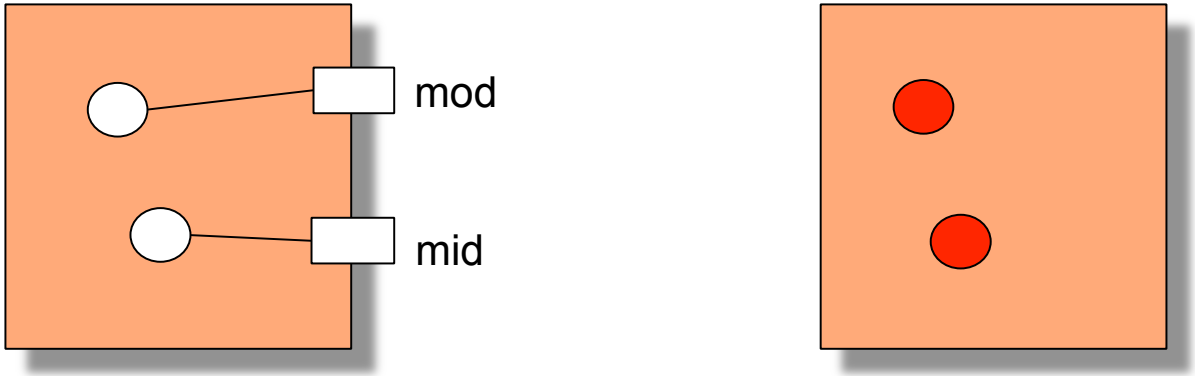
The COMPOST library [ISC] has such a metamodel (in Java)





Bind Composer Universally Parameterizes Fragment Components

- Like in BETA, for uniformly generic components

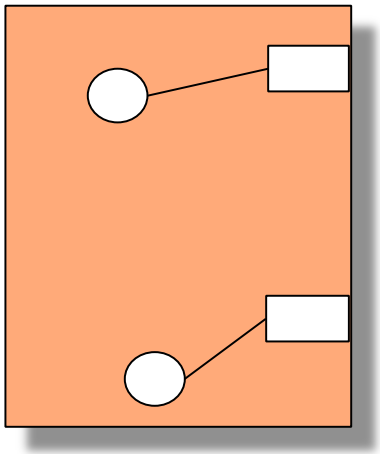



```
<<mod:Modifier>>
m (){

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

}
```

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

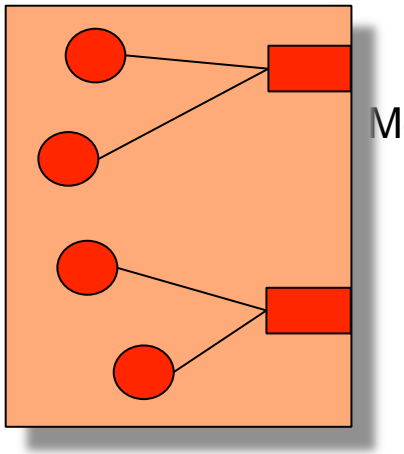
```
Box component = readBoxFromFile("m.java");

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

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

Extend Operator Universally Extends the Fragment Components



MethodEntry

MethodExit



MethodEntry

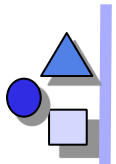
MethodExit

```
m (){
    abc..
    cde..
}
```

```
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 Provides Universal Symmetric Merge

- The **Extend** operator is asymmetric, i.e., extends hooks of a fragment component with new fragment values
- Based on this, a symmetric **Merge** operator can be defined:

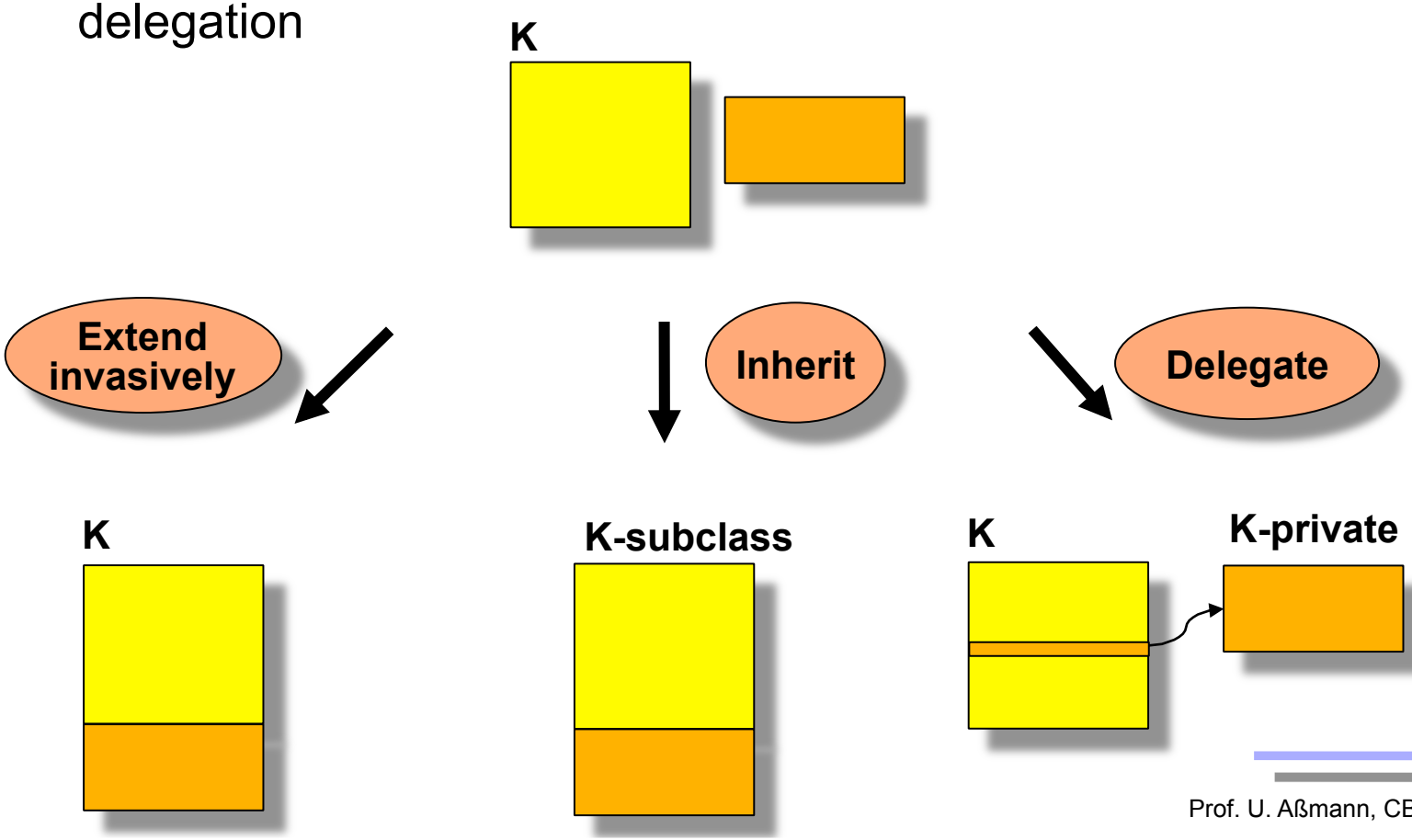
```
merge(Component C1, Component C2) :=
            extend(C1.list, C2.list)
```

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

- Both extend and merge work on fragments
  - Extend works on all collection-like language constructs
  - Merge on components with collection-like language constructs

Applied to Classes, Invasive Extension Integrates Feature Groups

- The Extend operator integrates feature groups and roles into classes (role merge)
  - because a feature group can play a role
- The semantics of invasive extension lies between inheritance and delegation



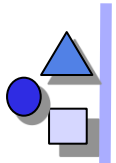
On the Difference of Declared and Implicit Hooks

- 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("");
```

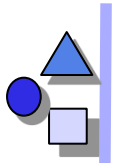
```
/* @genericMYModifier */ public print() {
  // <<MethodEntry>>
   if (1 == 2)
       System.out.println("Hello World");
       // <<MethodExit>>
       return;
   else
       System.out.println("Bye World");
       // <<MethodExit>>
       return;
}
```

```
synchronized public print () {
    if (1 == 2)
        System.out.println("Hello World");
        return;
    else
        System.out.println("Bye World");
        return;
}
```



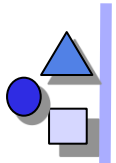
You Need Invasive Composition

- 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 templates have to be expaned in a type-safe way
- When physical unity of logical objects is desired
  - Invasive extension and merges roles into classes
  - No splitting of roles, but integration into one class
- When the resulting system should be highly integrated
  - When views should be integrated constructively



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



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, ..

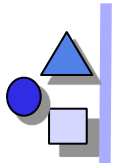
Graphical languages: boxes and lines (used in Reuseware)

Functional languages: Haskell

Scripting languages: TCL, Groovy, ...

Logic languages: Prolog, Datalog, F-Datalog

Declarative Languages: Attribute Grammars, Rewrite Systems



Homogeneous Composition Systems

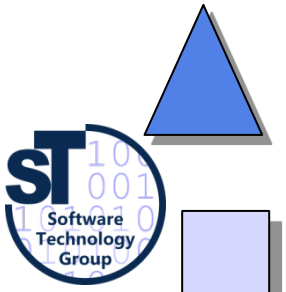
A composition system is called **homogeneous**, if it employs the same composition language and component language.

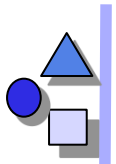
Otherwise, it is called **hegerogeneous**

In a homogeneous composition system, metacomposition is staged composition.

A **point-cut language (cross-cut language)** is a form of composition language.

26.2) What Can You Do With Invasive Composition?

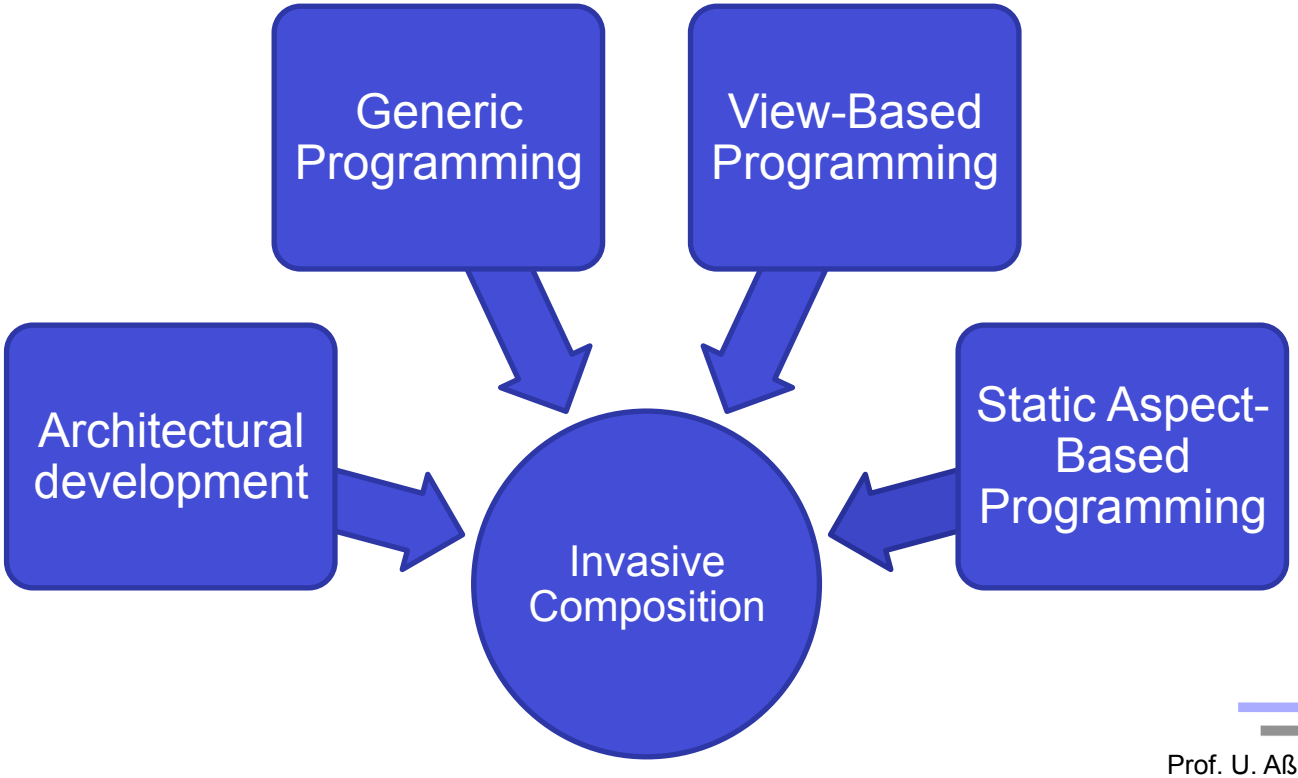




Invasive Composition

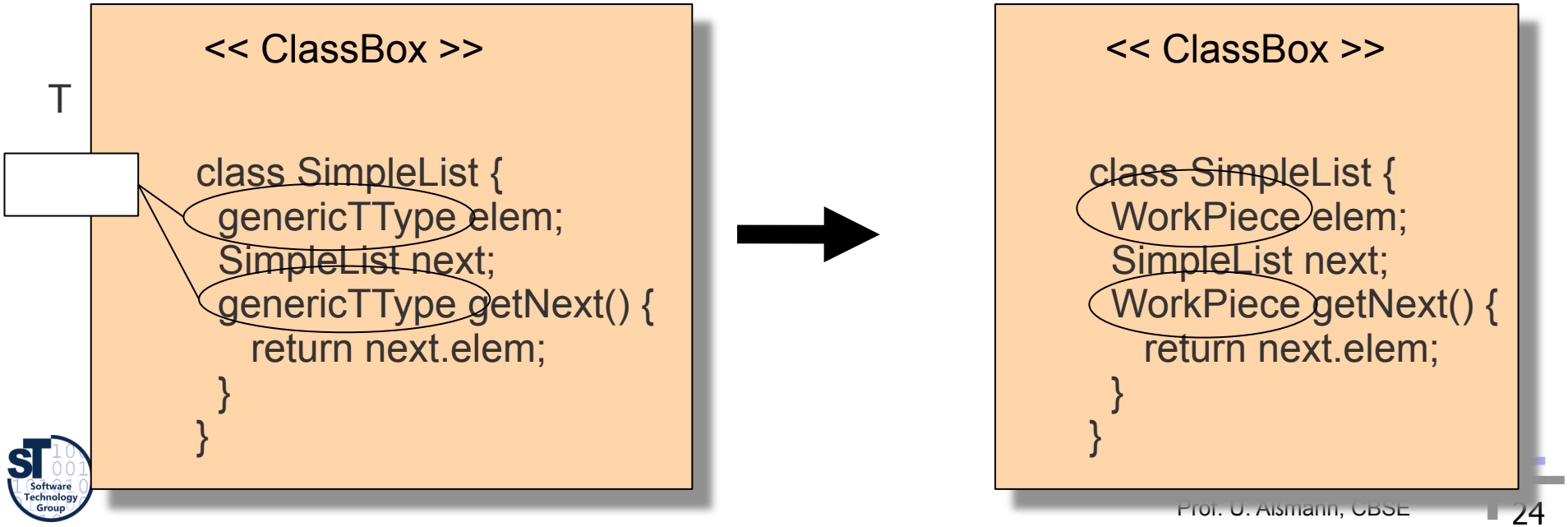
Adds a full-fledged composition language to generic and view-based programming

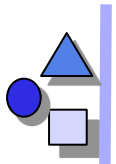
Combines architectural systems, generic, view-based and aspect-oriented programming



Universally Generic Programming

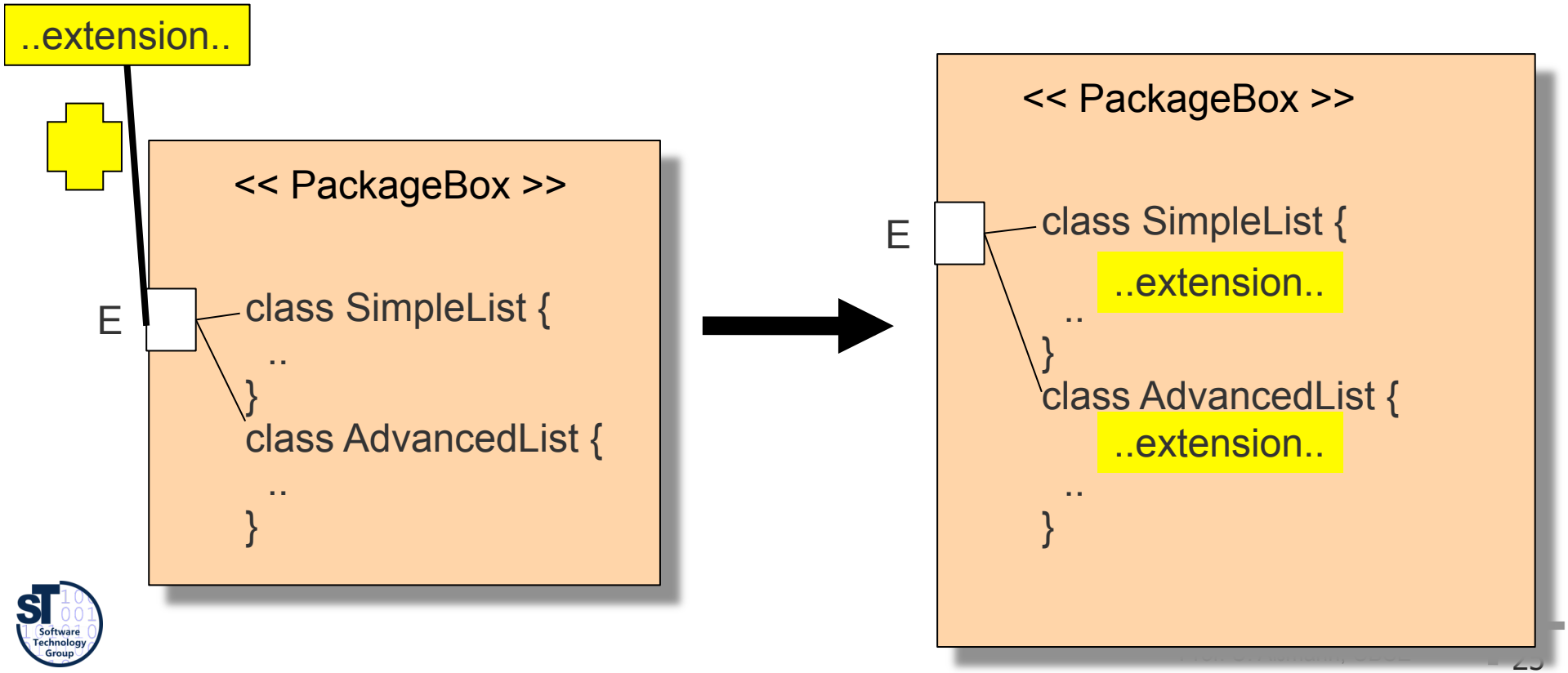
- ISC is a fully generic approach
- In contrast to BETA, ISC offers a full-fledged composition language
- Generic types, modifiers, superclasses, statements, expressions,...
- Any component language (Java, UML, ...)





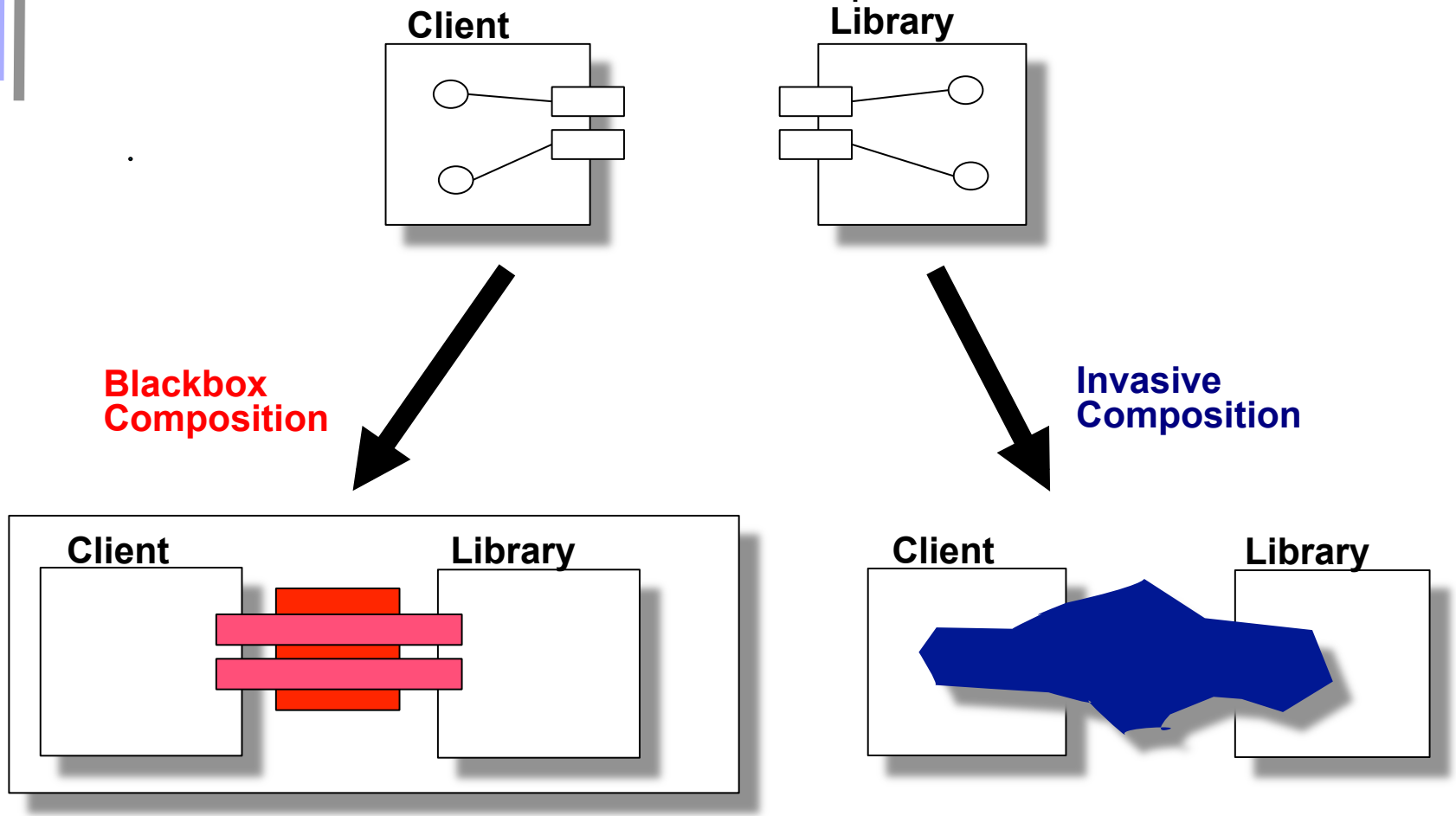
Universal Constructive View Programming

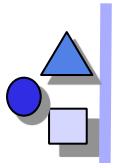
- ISC is a uniform and universal view-programming approach
  - The Extend operator realizes open definitions for *all* language constructs: methods, classes, packages
  - The Merge operator realizes symmetric composition for all language constucts
- Additionally, ISC offers a full-fledged composition language



Invasive Connections

- In contrast to ADL, ISC offers invasive connections [AG00]
- Modification of inheritance relations possible

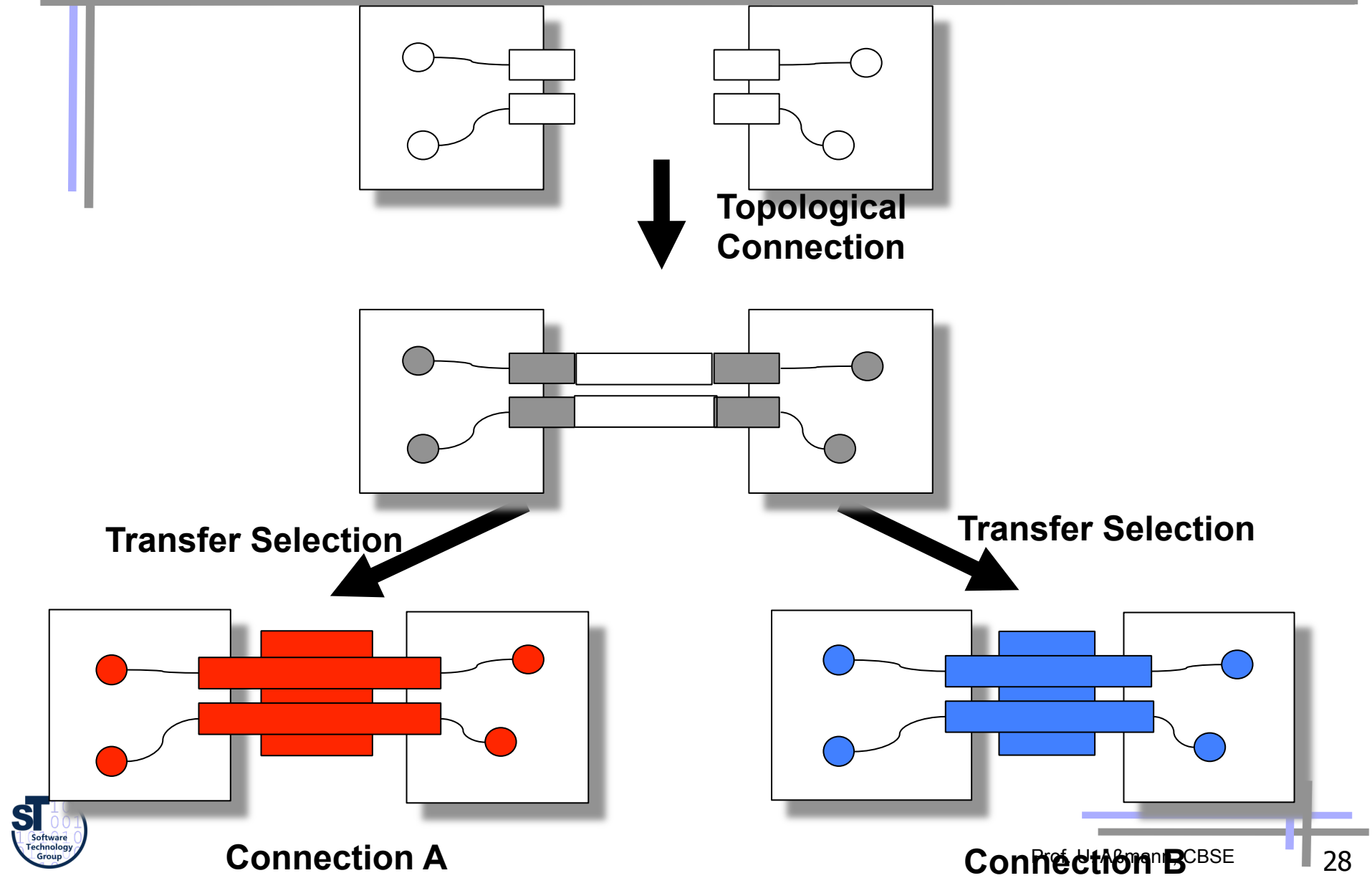




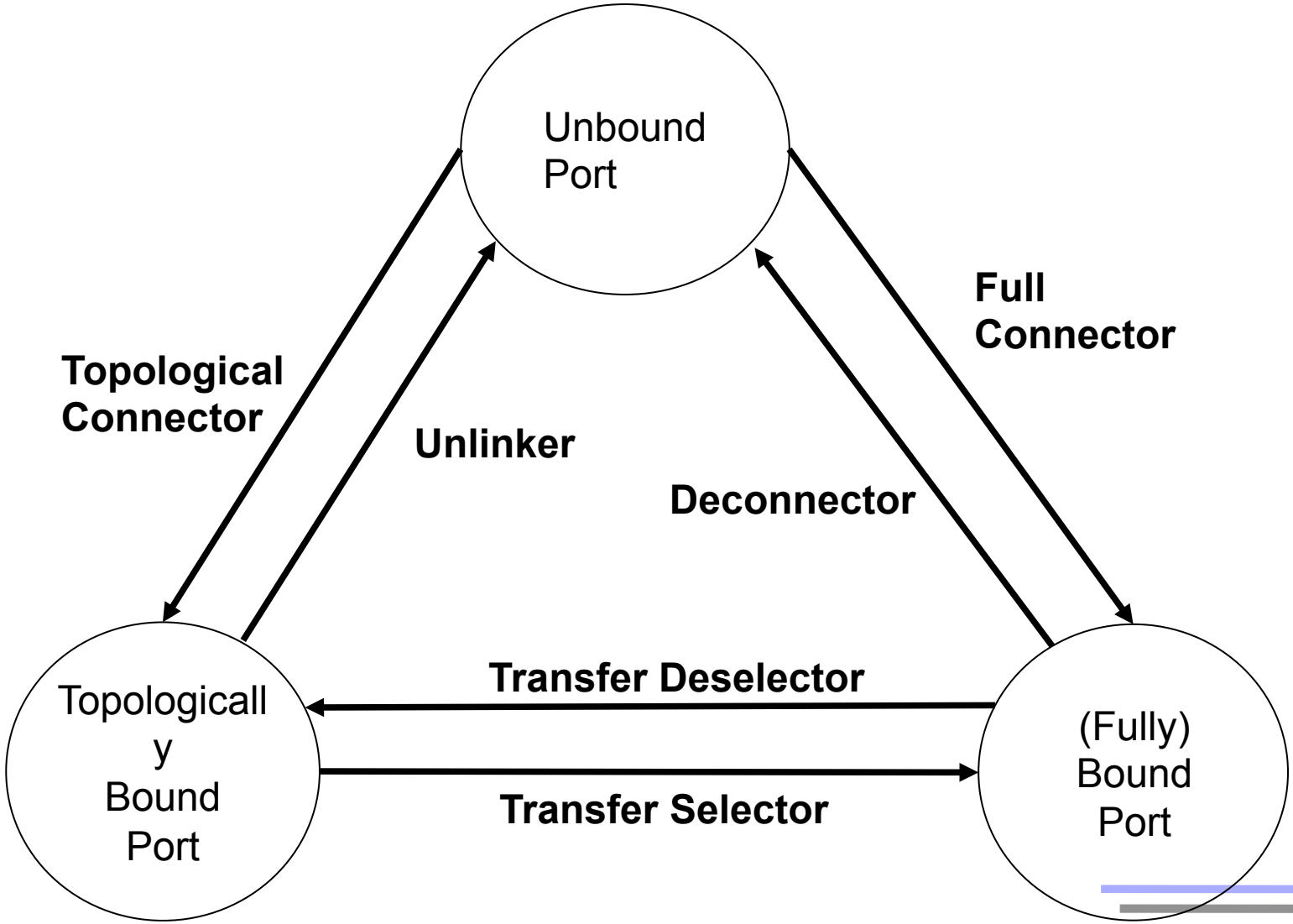
Invasive Architectural Programming

[ISC] shows how *invasive connectors* achieve tightly integrated systems by embedding the glue code into senders and receiver components

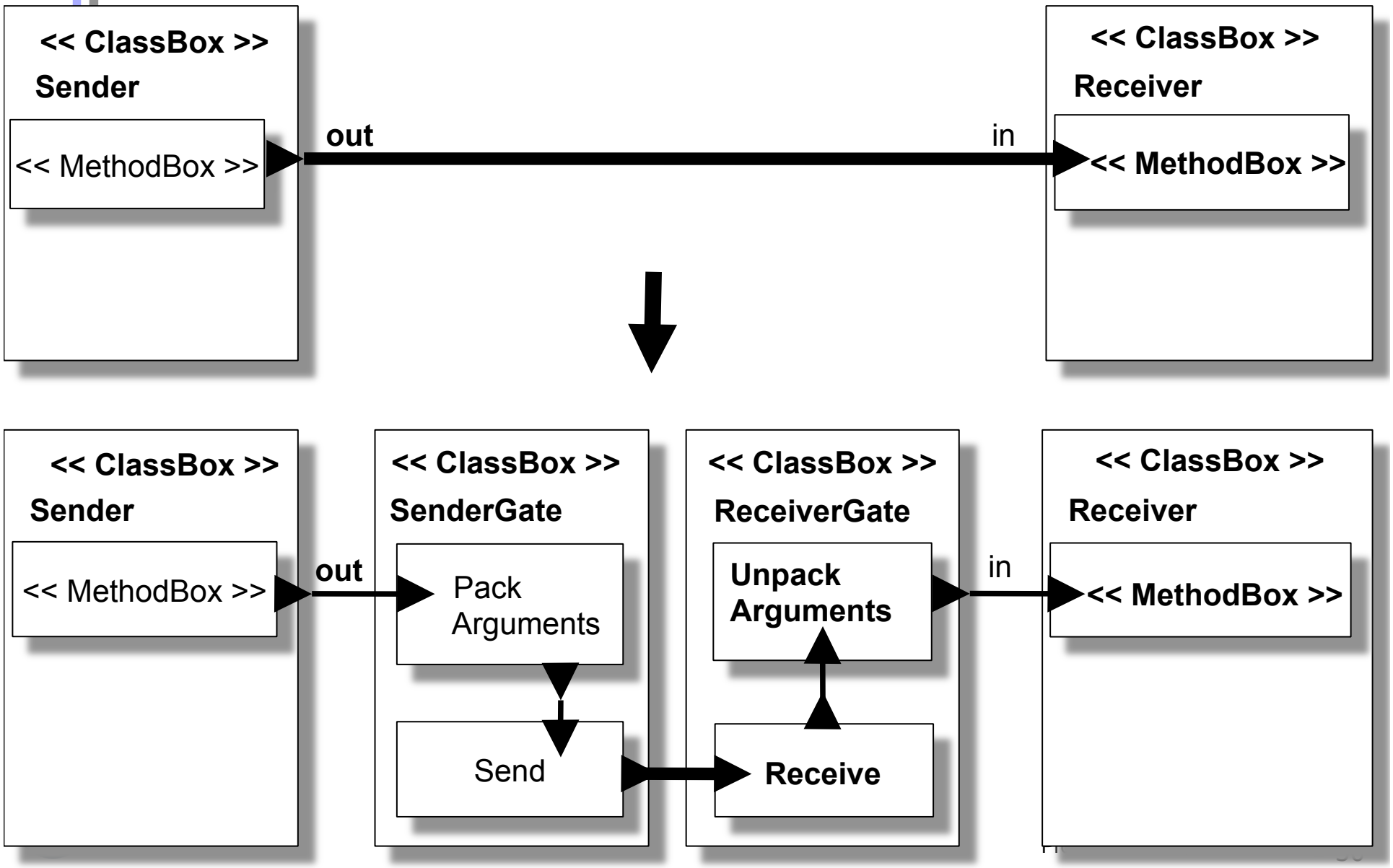
Separation of Topological from Transfer Aspect

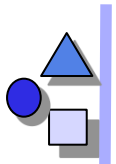


Port Binding State Diagram



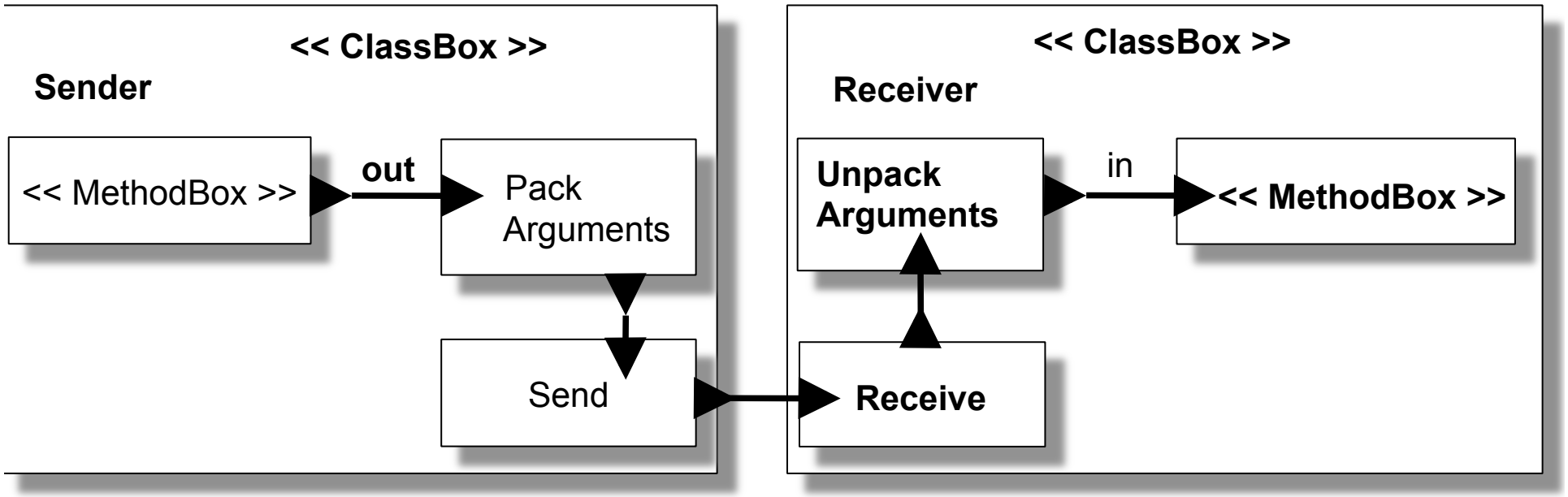
Gate Objects: Glue Separate

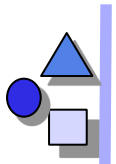




Invasive Connection

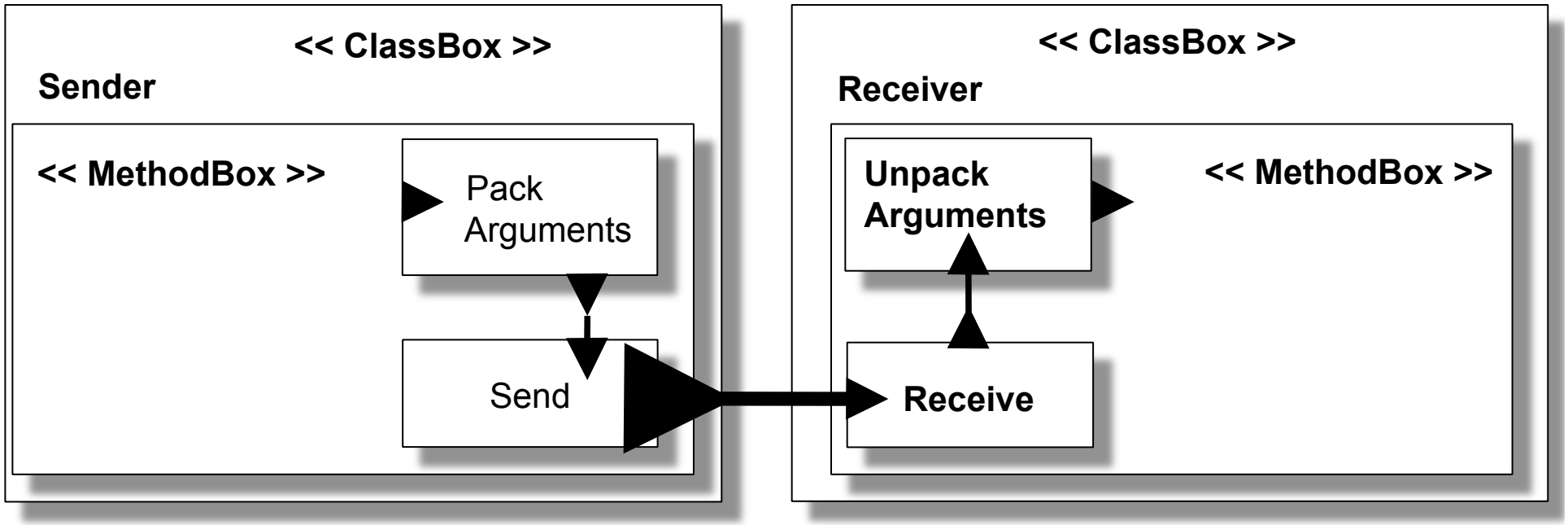
Embedding communication gate methods into a class





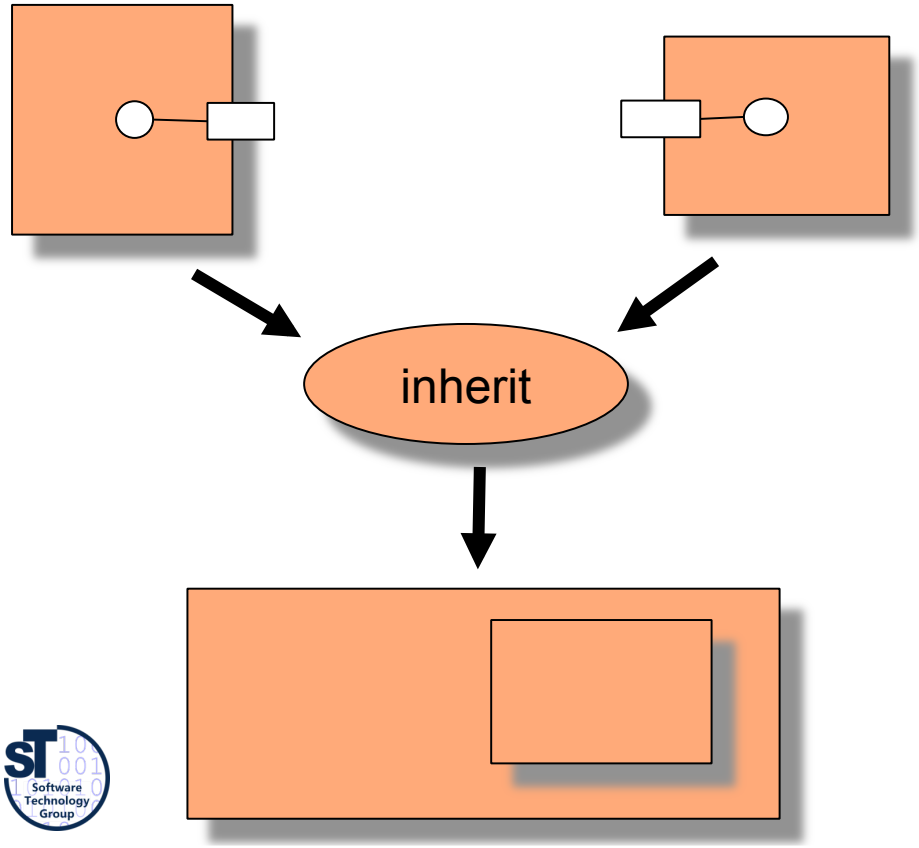
Invasive Connection

Embedding glue code into sender methods



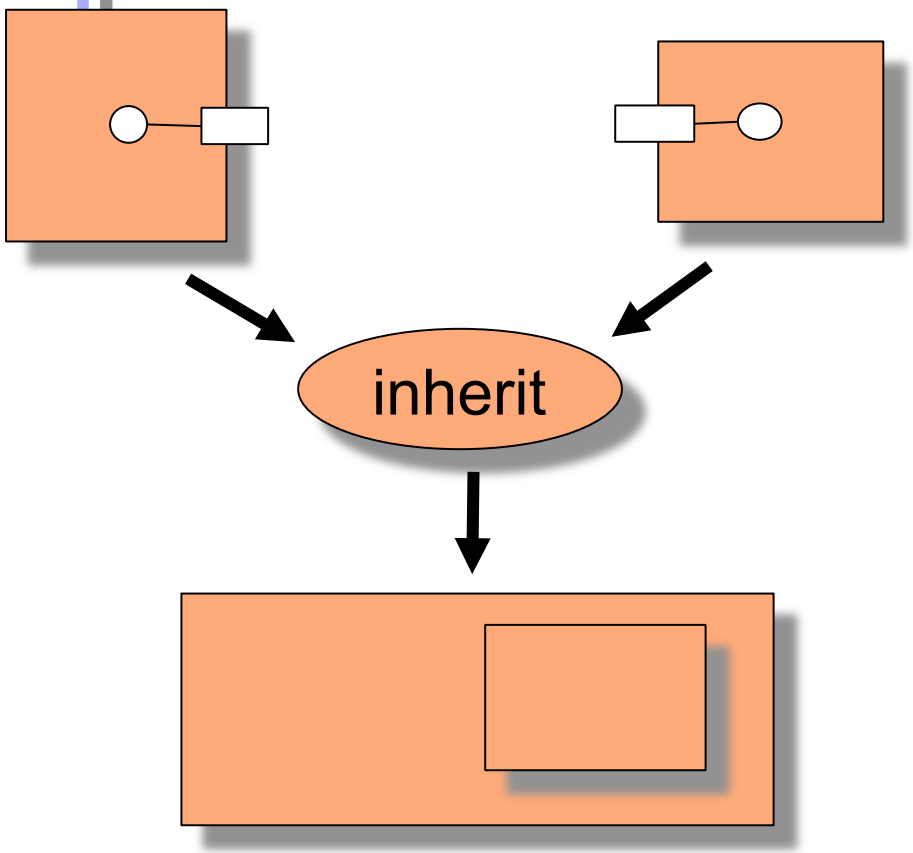
Universal Inheritance and Mixins

- Extension can be used for inheritance and mixins
- In contrast to OO languages, ISC offers tailored inheritance operations, based on the extend operator

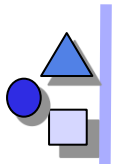


- **inheritance** :=
  - copy first super class
  - extend with second super class
- **mixin_inheritance** :=
  - Bind superclass reference

Mixin Inheritance Works Universally for Languages that don't have it



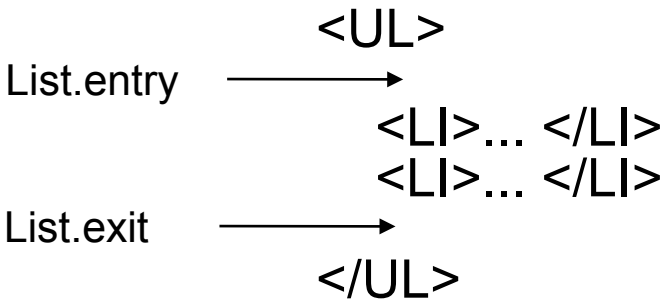
- Invasive composition can model mixin inheritance uniformly for all languages
- e.g., for XML
- inheritance :=
  - copy first super document
  - extend with second super document

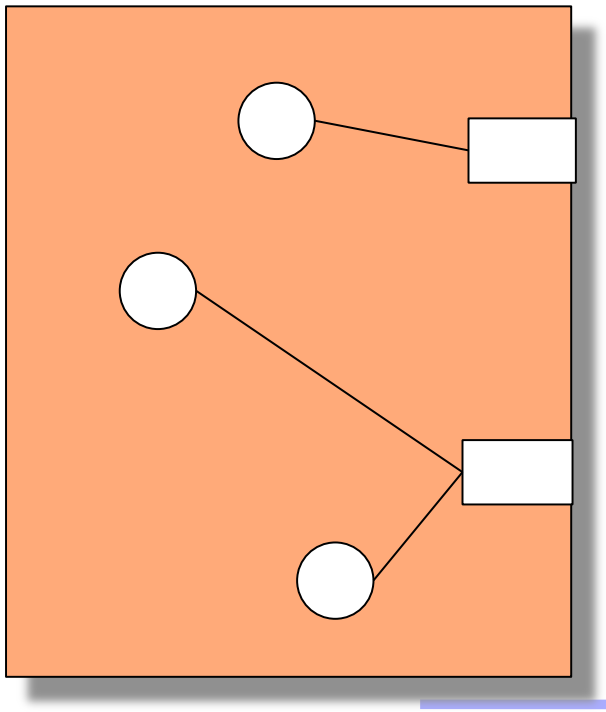


Invasive Document Composition for XML

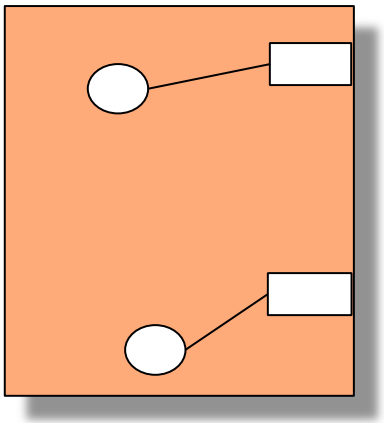
- Invasive composition can be used for document languages, too [Hartmann2011]

- Example List Entry/Exit of an XML list
- Hooks are given by the Xschema



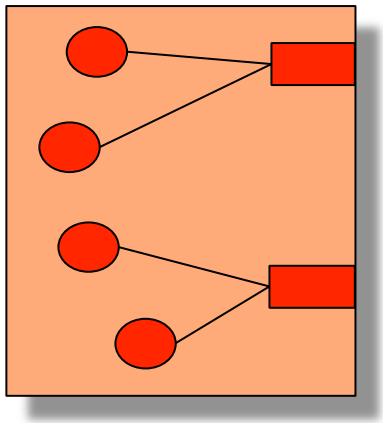


Hook Manipulation for XML



List Entry

List Exit



List Entry

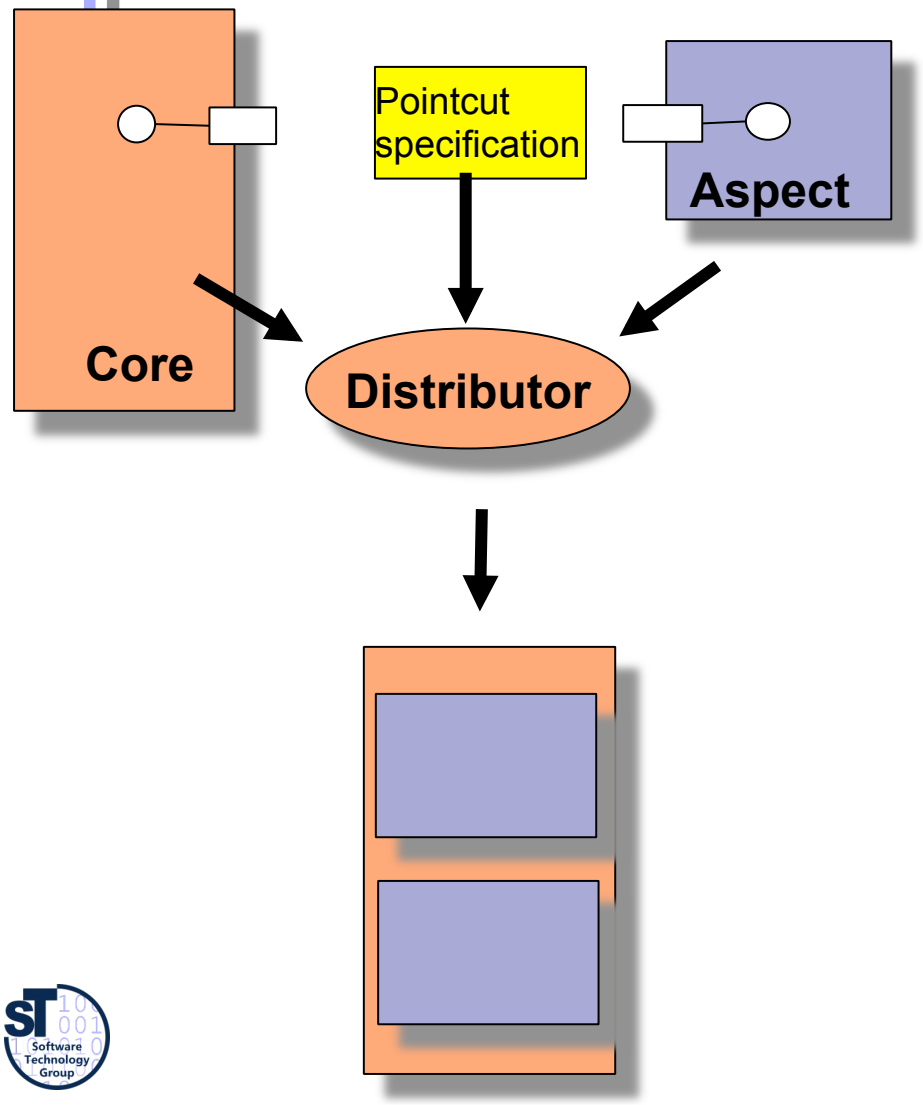
List Exit

```
<UL>

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

</UL>
```

```
<UL>
   <LI>... </LI>
   <LI>... </LI>
   <LI>... </LI>
   <LI>... </LI>
</UL>
```

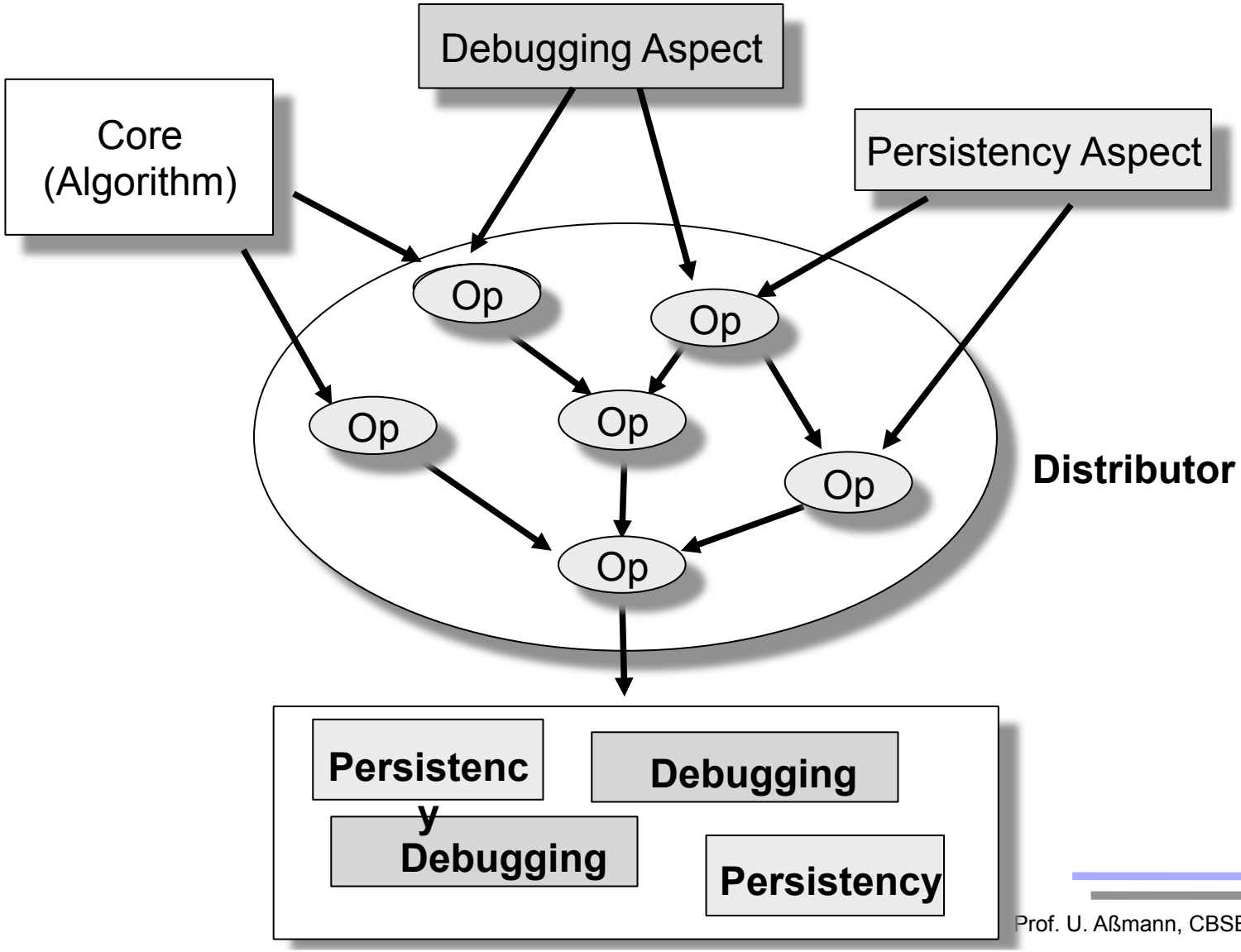
```
XMLcomponent.findHook(„ListEntry“).extend(„<LI>... </LI>”);

XMLcomponent.findHook(„ListExit“).extend(“<LI>... </LI>”);
```

Universal Weaving for AOP (Core and Aspect Components)

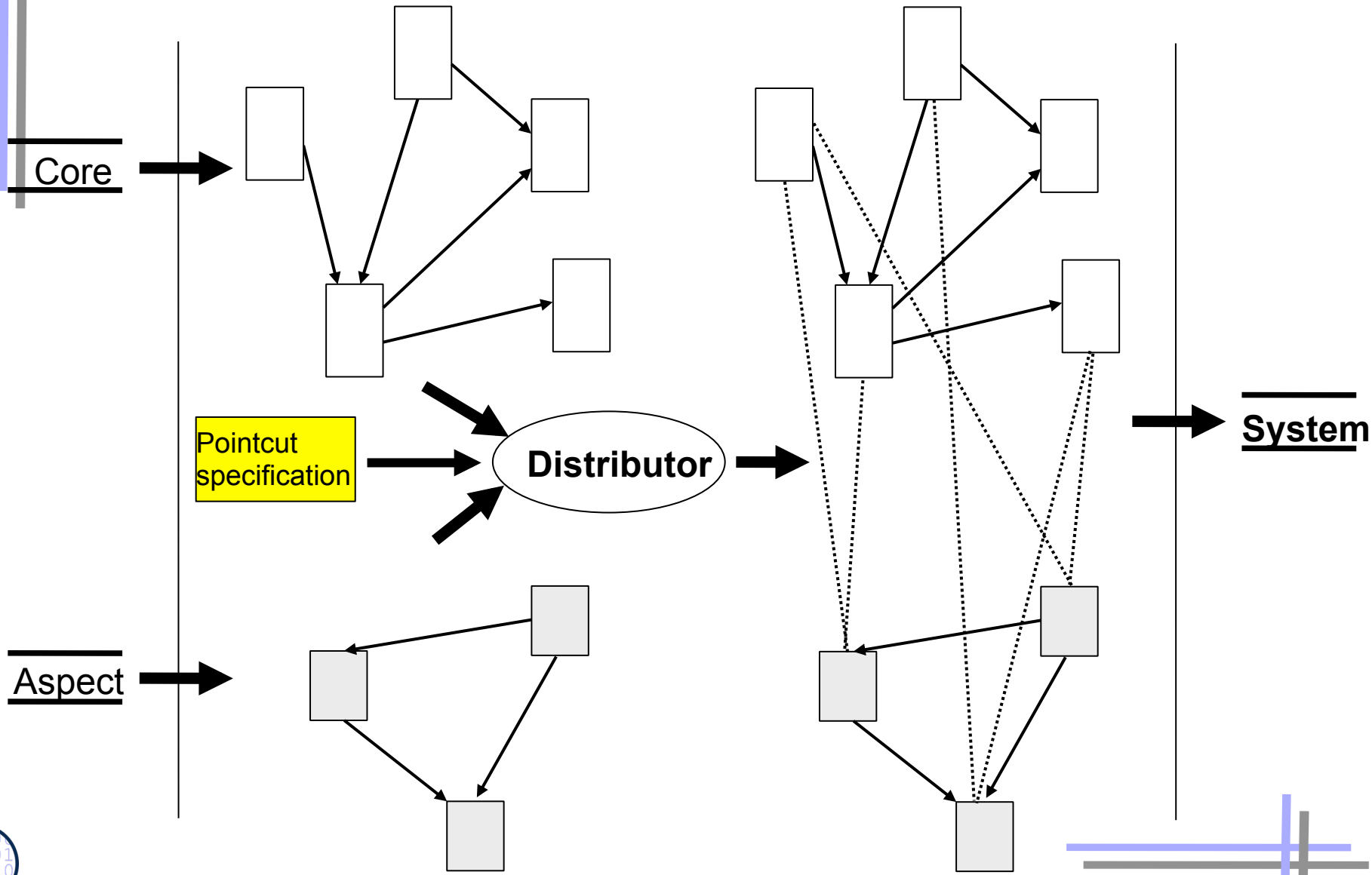


- Complex composers distribute aspect fragments over core fragments
- *Distributors* extend the core
- Distributors are more complex operators, defined from basic ones
- Static aspect weaving can be described by distributors, because hooks are static
  - ISC does not have a dynamic joinpoints
  - Crosscut specifications can be interpreted

Distributors are Composition Programs



Distributors Weave Relations between Core and Aspect

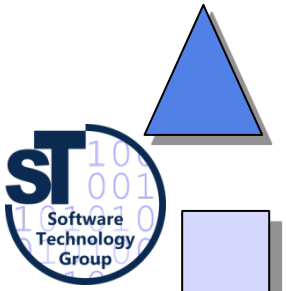


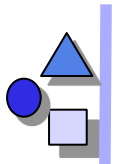
Invasive Model Composition with Reuseware

Editor specification

Universal Reuse Add-On Languages

for universal genericity and extension





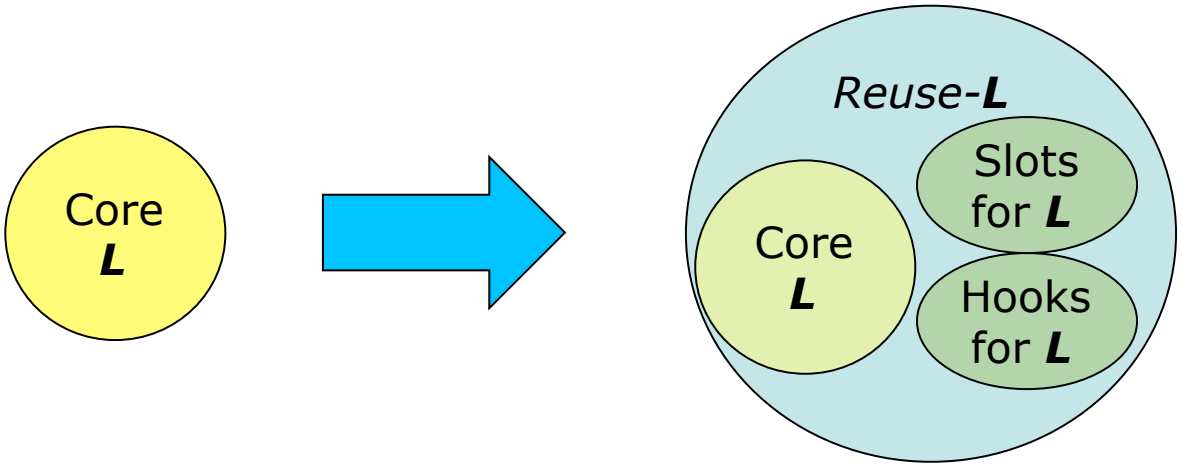
Universally Composable Languages

Universally composable: A language is called *universally composable*, if it provides universal genericity and universal extensibility

Reuse add-on language: Given a metamodel of a *core* language L, a metamodel of a universally composable language can be generated (Reuse-L)

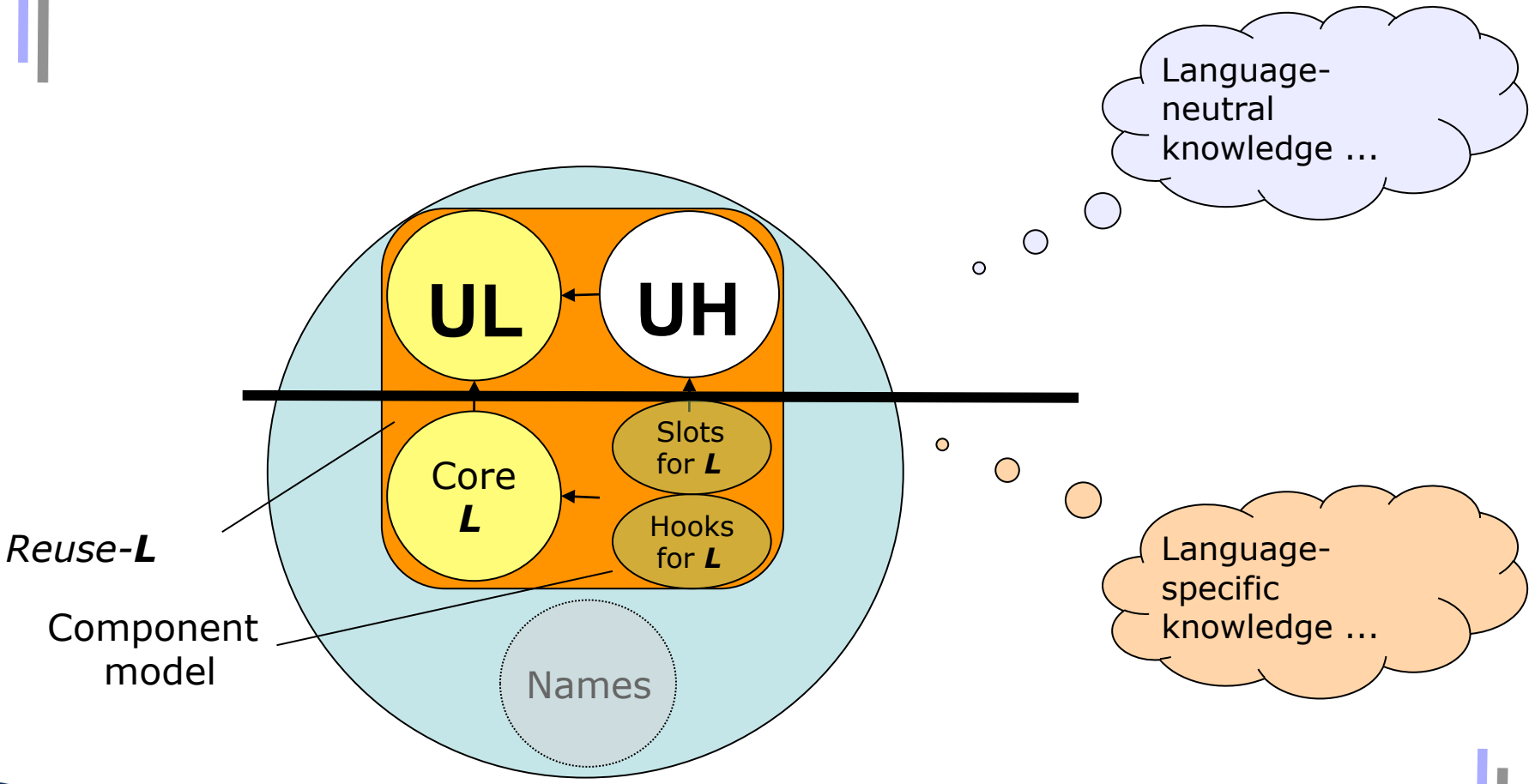
Slot and Hook model: Generated from the core language metamodel

- realizes universal composability by defining *slots* and *hook constructs*, one for each construct in the core language



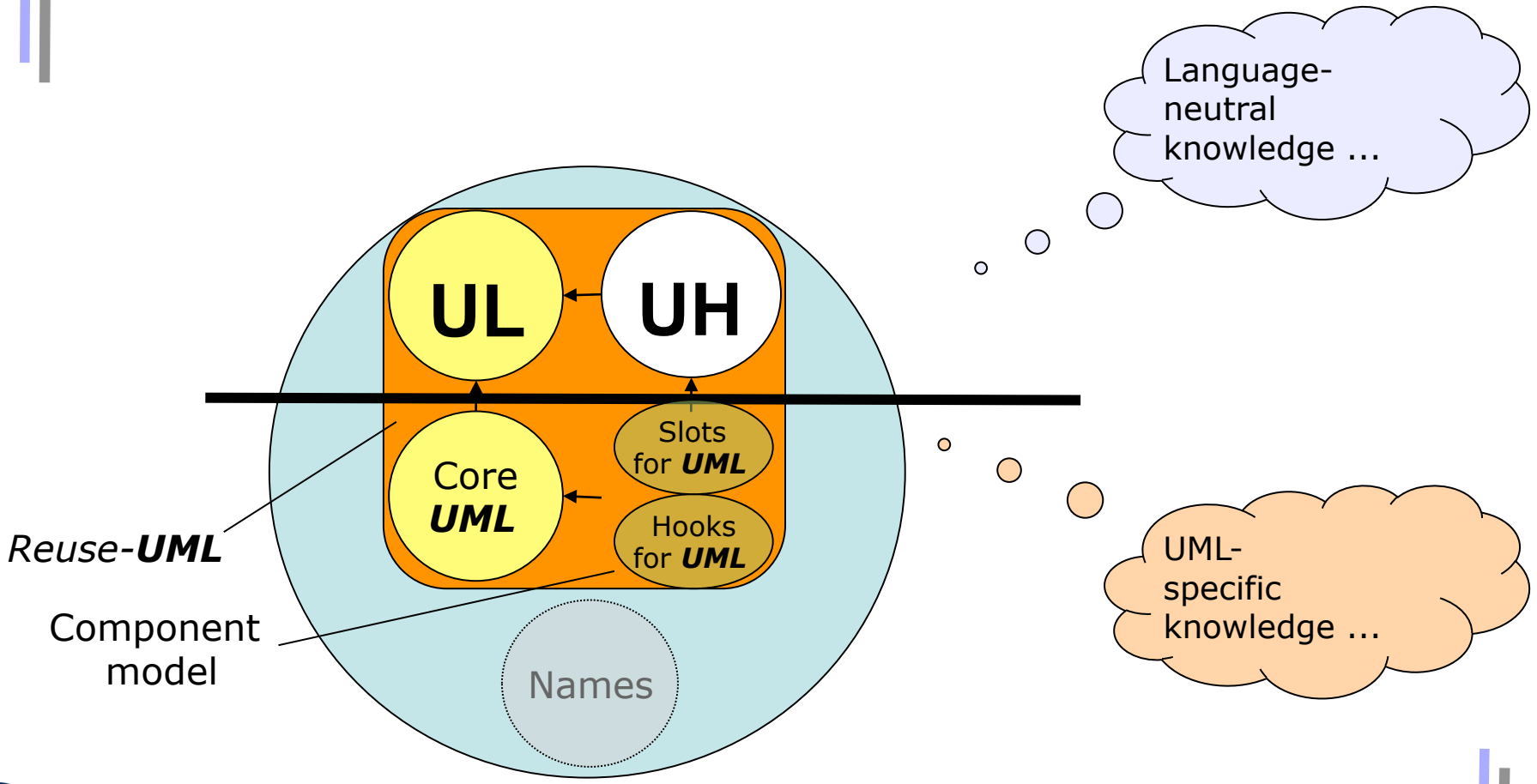
Structure of a Universally Composable Language

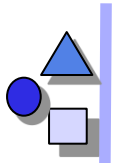
- The reuse language has two levels...



Reuse-UML, a Universally Composable Language

- .. an extension of UML with slot and hook model

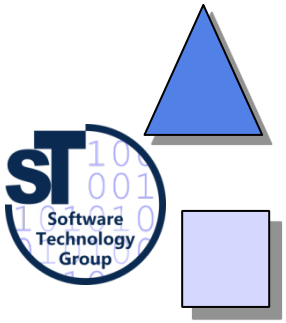




The Reuseware Tool

- www.reuseware.org (Phd of Jendrik Johannes, 2010)
- The ST group develops a tool, *Reuseware,* for reuse languages:
  - Eclipse-based
  - metamodel-controlled (metalanguage M3: Eclipse e-core)
  - Plugins are generated for composition
  - Composition tools come for free
- Framework instantiation is supported for variantion and extension
- Jobs open!

26.3) Composition and Functional Interfaces

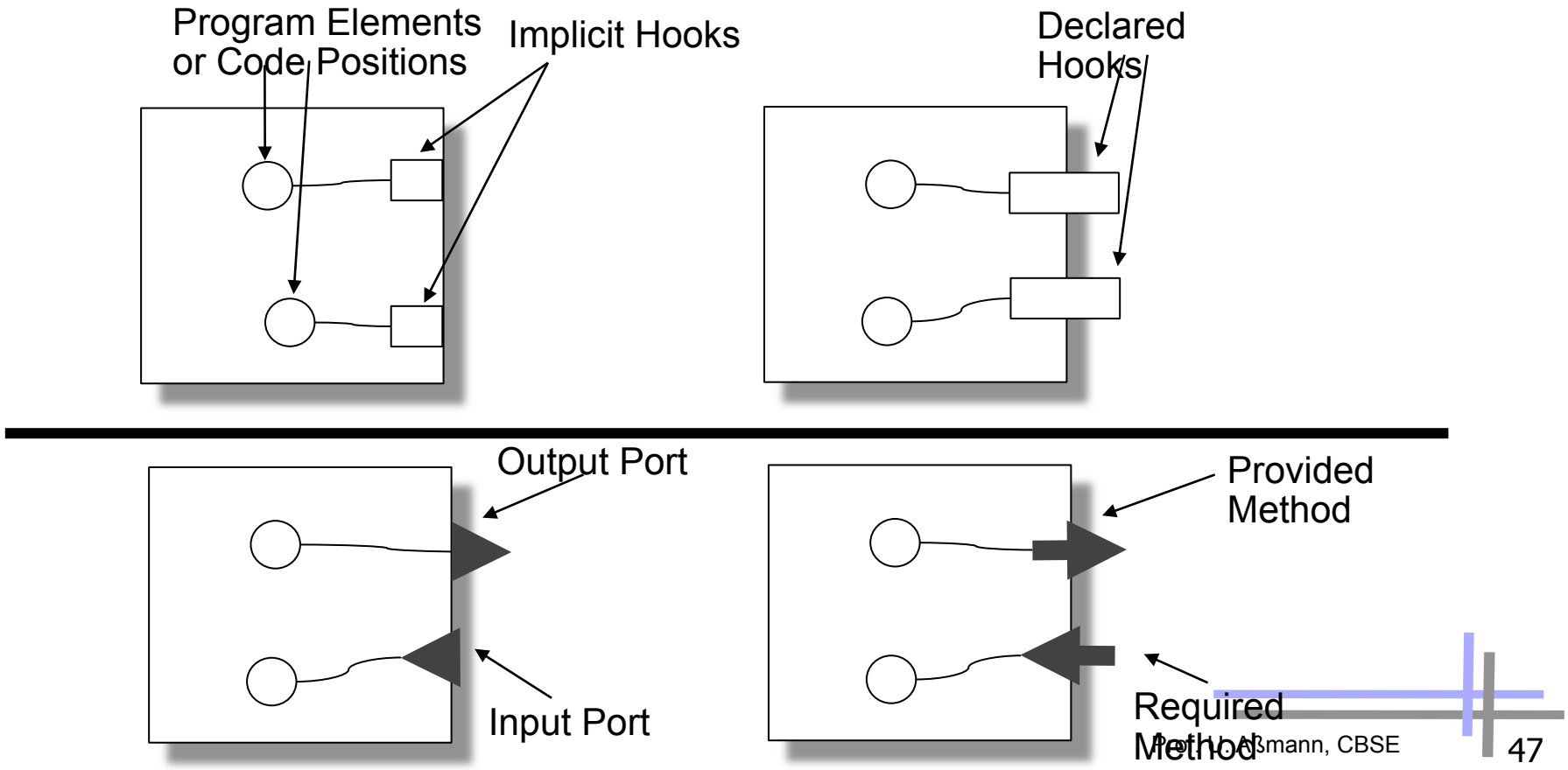


Composition vs Functional Interfaces

Composition interfaces contain hooks and slots

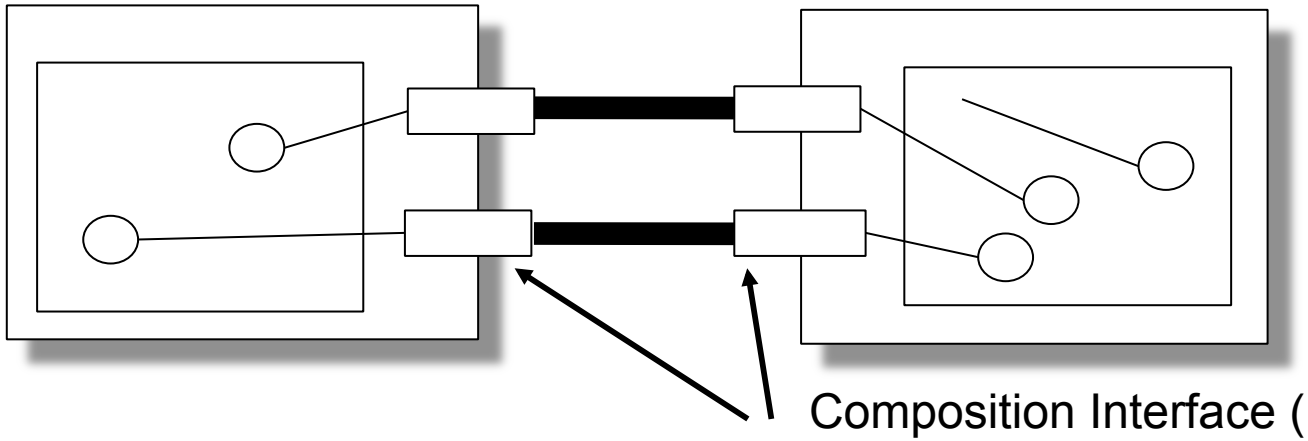
static, based on the component model at design time

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



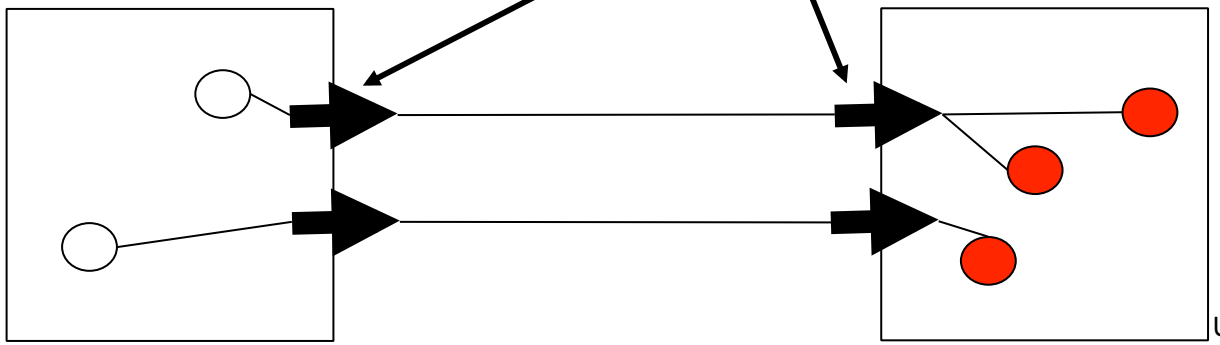
Functional Interfaces are Generated from Composition Interfaces

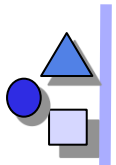
2-stage process



Composition Interface (Boxes with Declared Hooks)

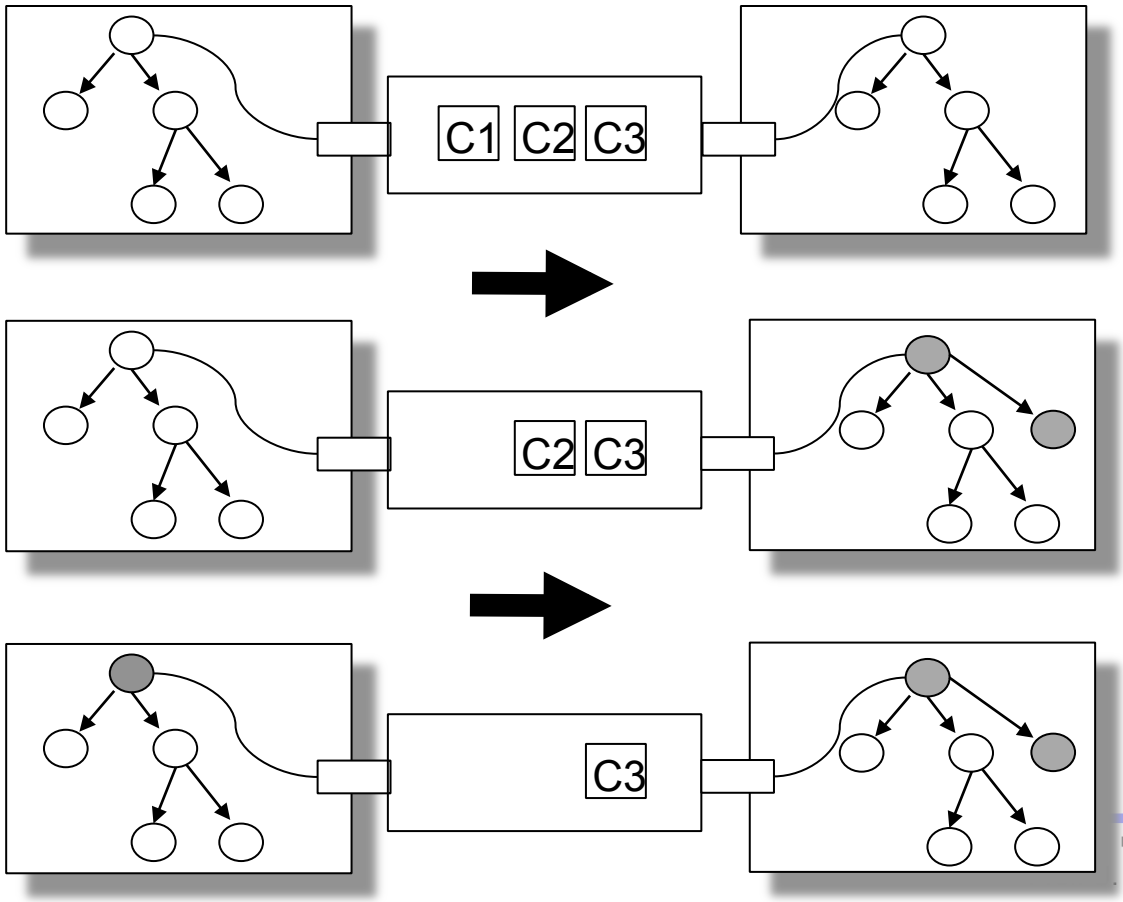
Functional Interface (Classes or Modules with Methods)

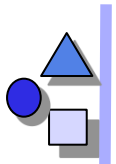




Execution of a Composition Program

- A compostion 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



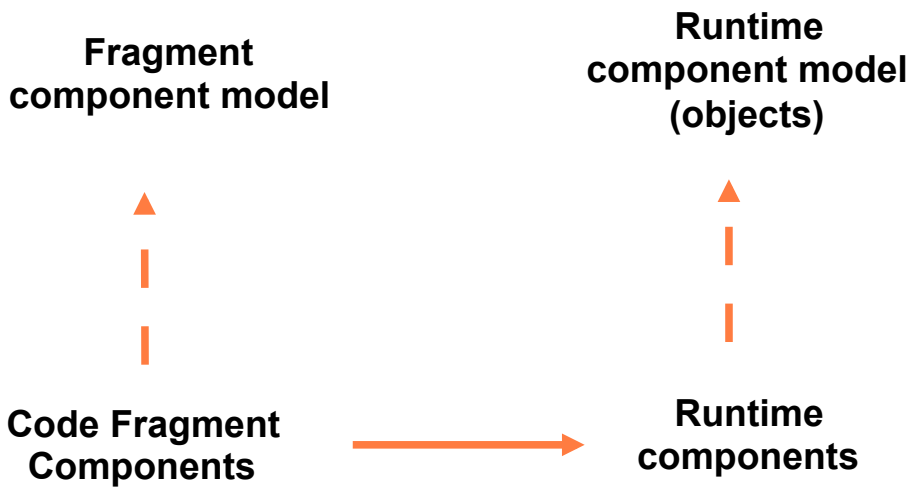


The Stages of ISC

- Produces code from fragment components by parameterization and expansion
- The run-time component model fits to the chip

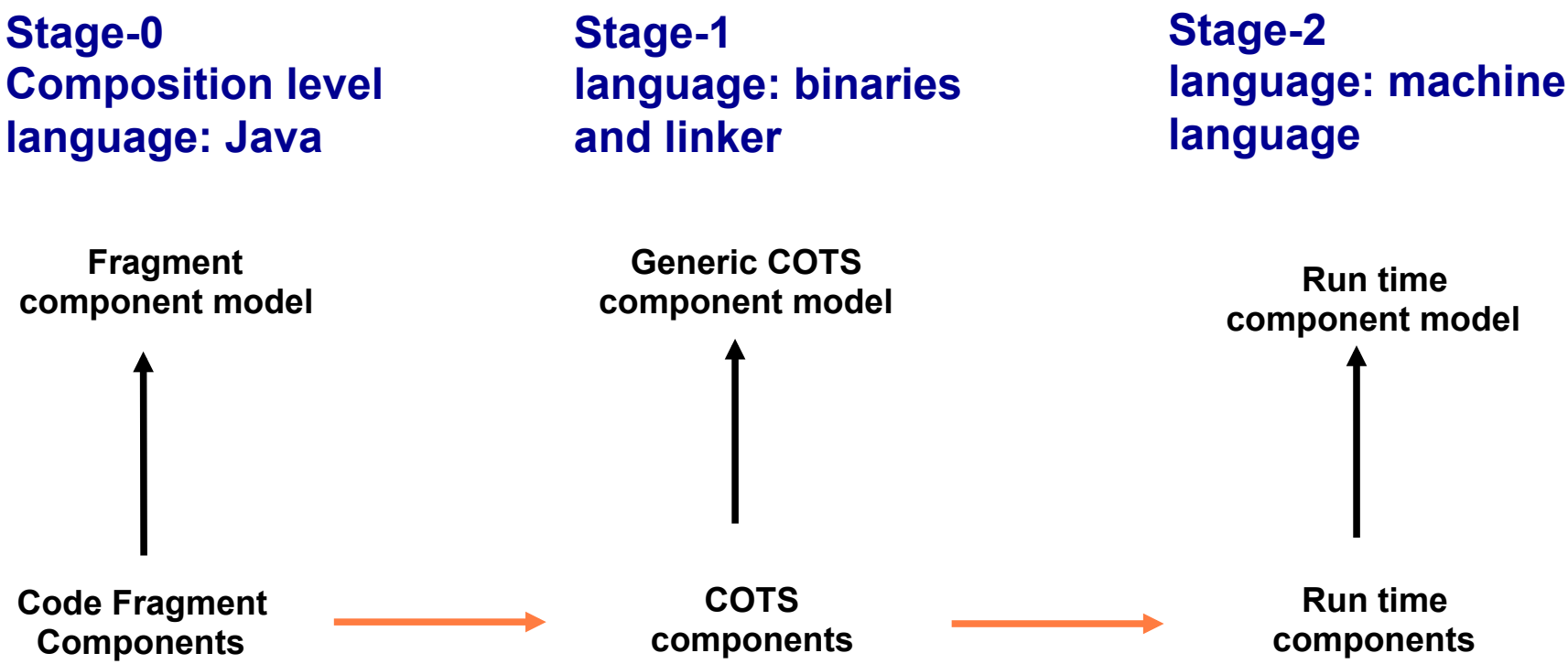
Stage-0
Composition level
language: Java

Stage-1
language: binary
machine language



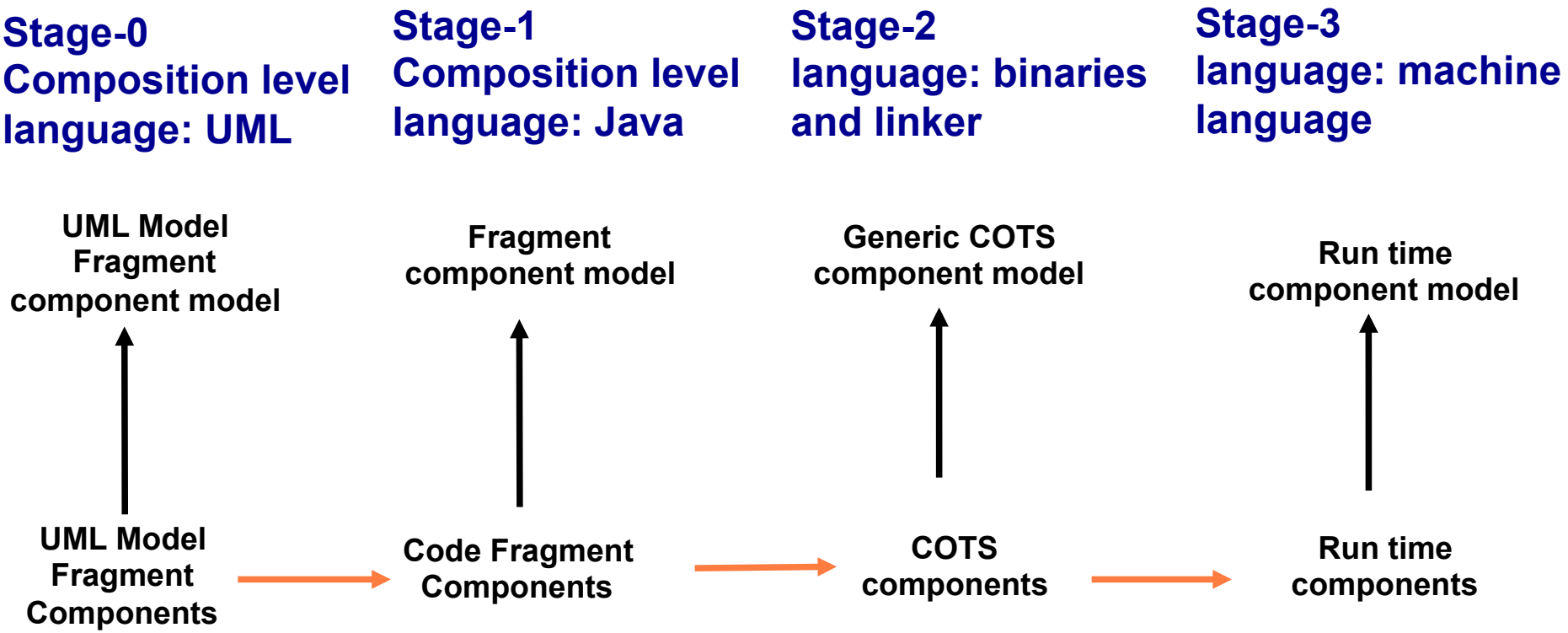
Component Models on Different Levels in the Software Process

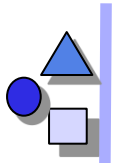
Standard COTS models are just models for binary code components



Component Models on Different Levels in the Software Process

Another stage can be introduced by *UML model composition* from which Java code is generated [Johannes 10]





Staging

- With a universal composition system as Reuseware, stages can be designed (stage design process)
- For each stage, it has to be designed:

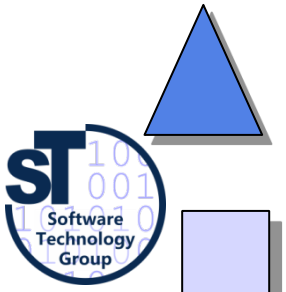
  component models

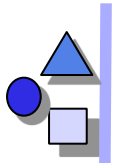
  composition operators

  composition language

  composition tools (editors, well-formedness checkers, component library etc.)

26.4) Different Forms of Greyboxes (Shades of Grey)





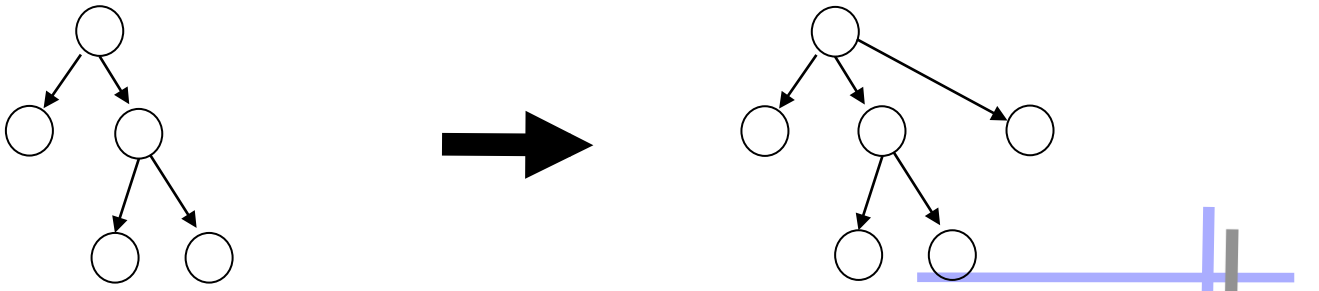
Invasive Composition and Information Hiding

- 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

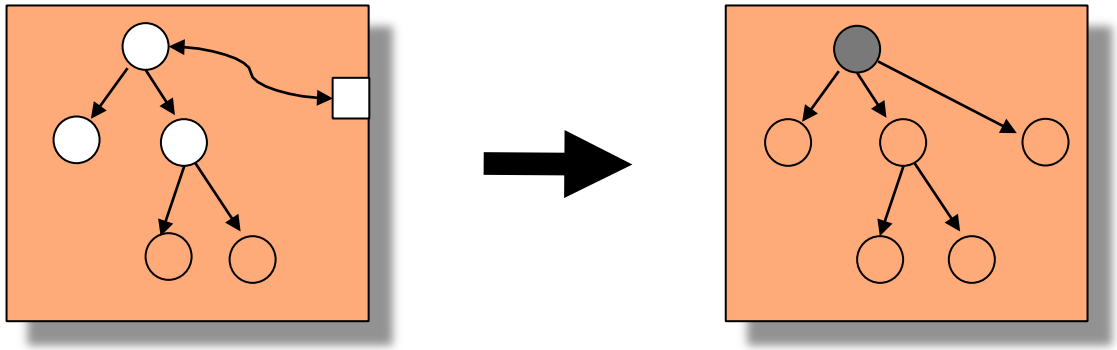
Refactorings
Transformations



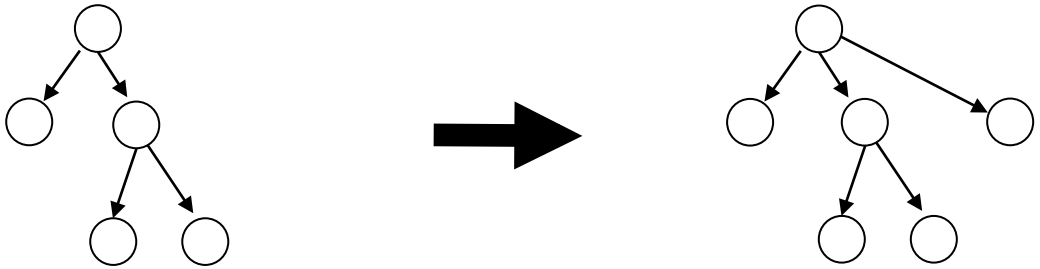
Modifying Implicit Hooks is a Light-Grey Operation

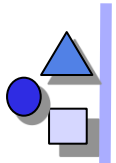
- Aspect weaving and view composition works on implicit hooks *(join points)*
- *Implicit composition interface*

Composition with implicit hooks



Refactorings Transformations

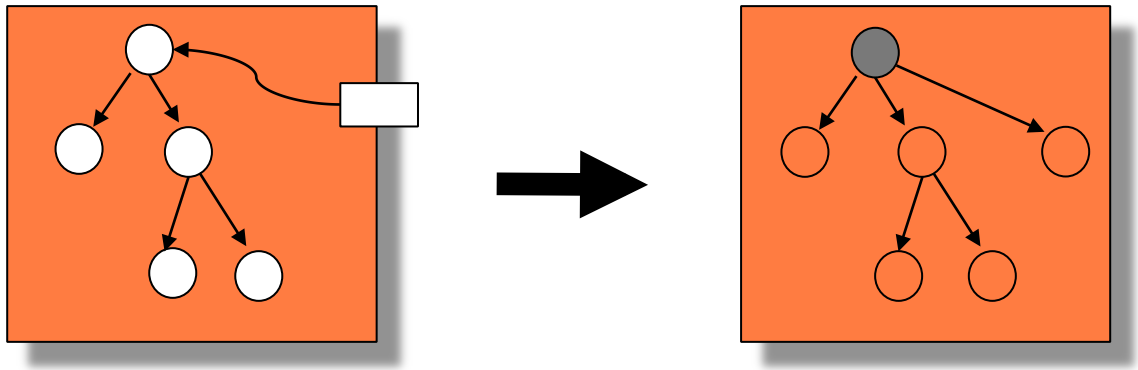




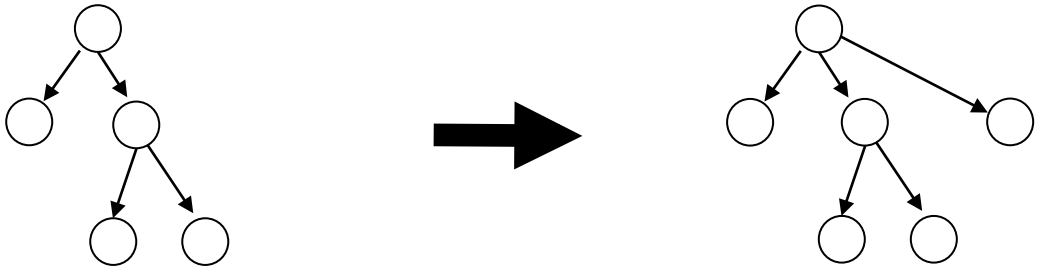
Parameterization as Darker-Grey Operation

- Templates work on *declared hooks*
- *Declared composition interface*

Composition with declared hooks

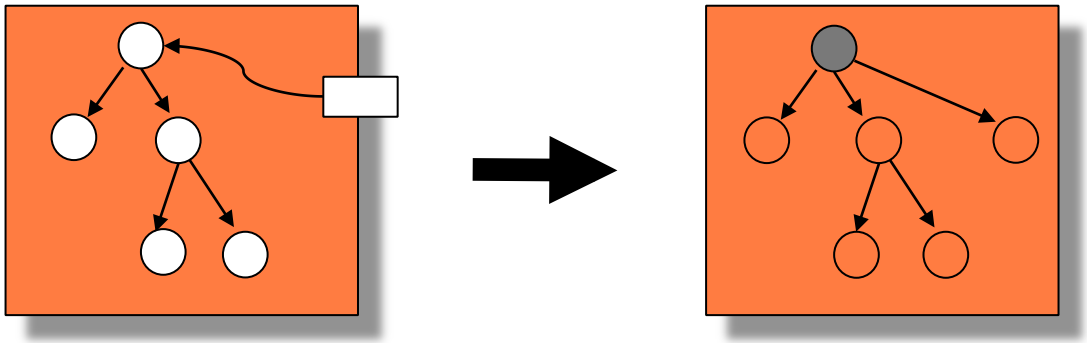


Refactorings Transformations

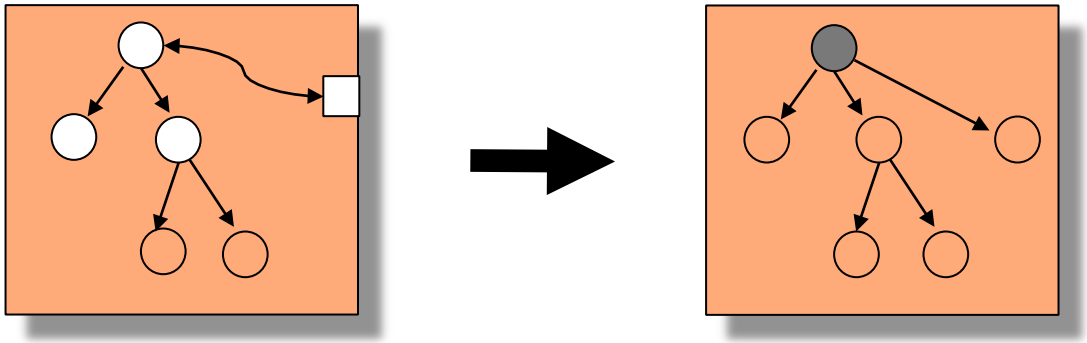


Systematization Towards Greybox Component Models

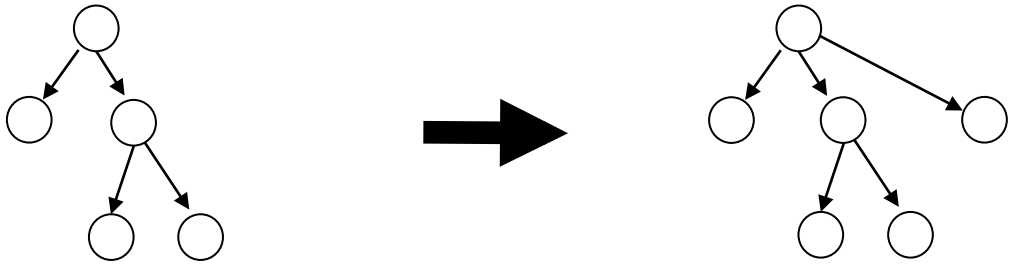
Composition with declared hooks



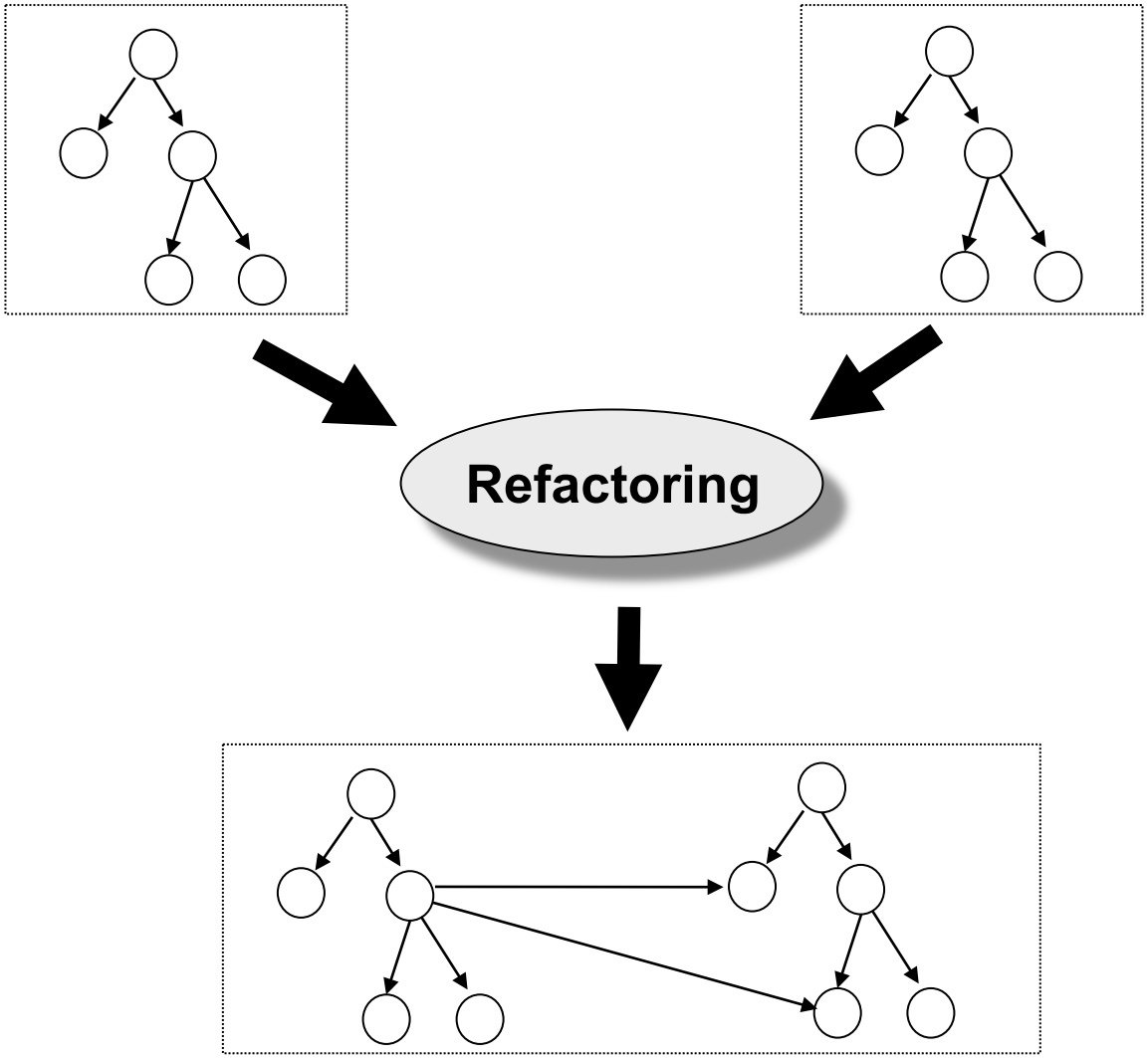
Composition with implicit hooks



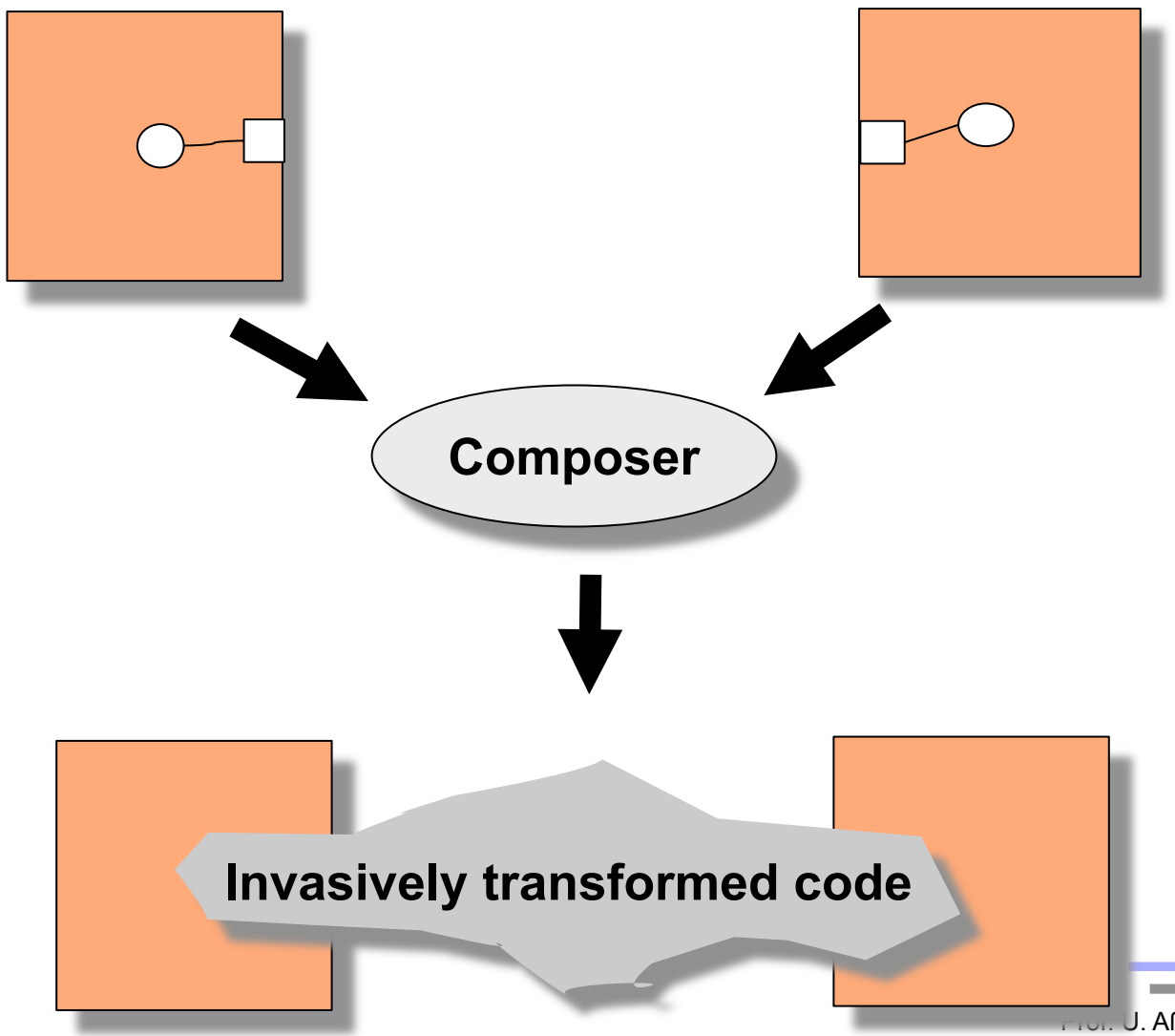
Refactorings Transformations



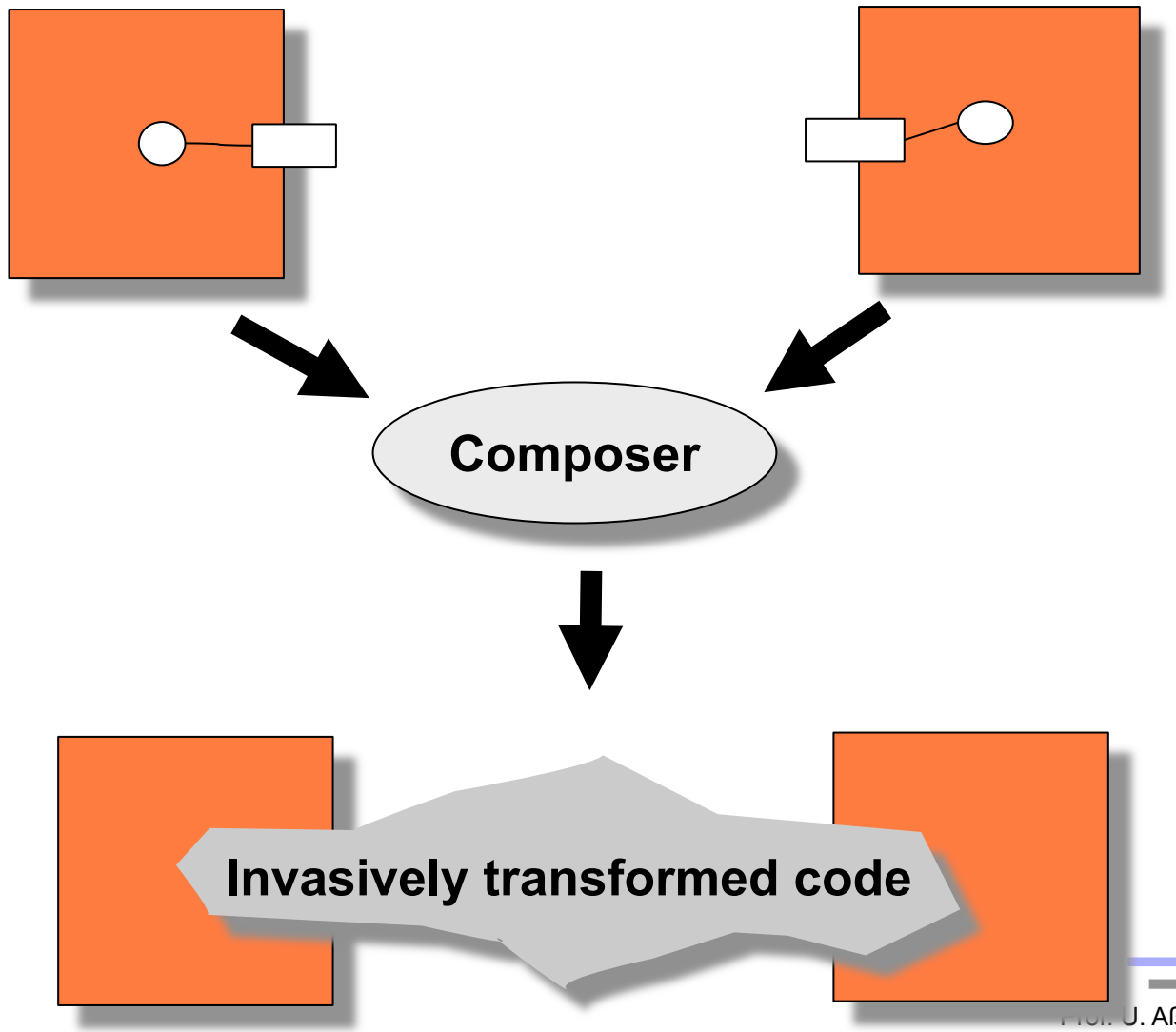
Refactoring Builds On Transformation Of Abstract Syntax



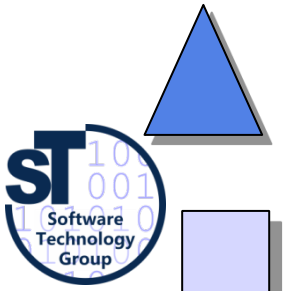
Invasive Composition Builds On Transformation Of Implicit Hooks

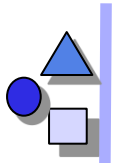


Invasive Composition Builds On Transformation on Declared Hooks



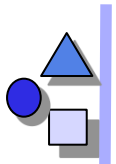
26.5 Invasive Software Composition as Composition Technique





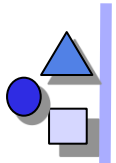
Invasive Composition: Component Model

- 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



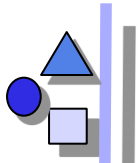
Invasive Composition: Composition Technique

- 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

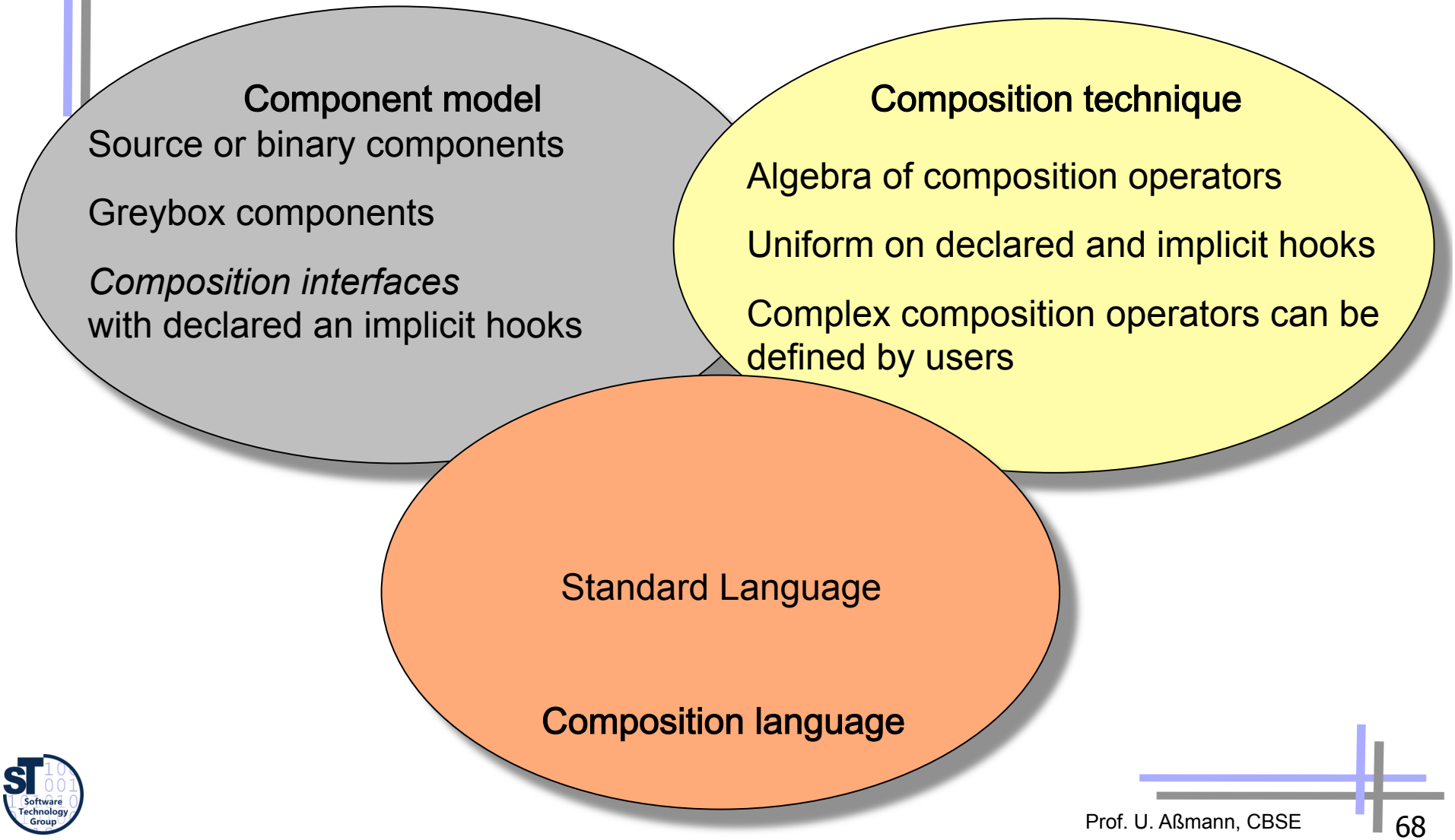
- Various languages can be used
- Product quality improved by metamodel-based typing of compositions
- Metacomposition possible
  - Architectures can be described in a standard object-oriented language and reused
- An *assembler* for composition
  - Other, more adequate composition languages can be compiled

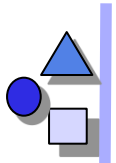


Conclusions for ISC

- 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

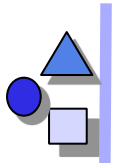
Invasive Composition as Composition System





What Have We Learned

- With the uniform treatment of declared and implicit hooks and slots, several technologies can be unified:
  - Generic programming
  - Connector-based programming
  - View-based programming
    - Inheritance-based programming
  - Aspect-based programming
  - Refactorings



The End