46. Invasive Software Composition (ISC)

1

Prof. Dr. Uwe Aßmann

Technische Universität Dresden

Institut für Software- und Multimediatechnik

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

Version 16-0.2, 11.06.16

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

Obligatory Literature

- ISC book Chap 4
- www.the-compost-system.org (now obsolete)
- 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.

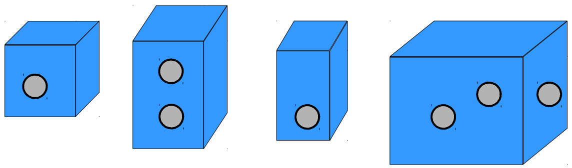
[Henriksson-Thesis] Jakob Henriksson. A Lightweight Framework for Universal Fragment Composition. Technische Universität Dresden, Dec. 2008 http://nbn-resolving.de/urn:nbn:de:bsz:14-ds-1231261831567-11763

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

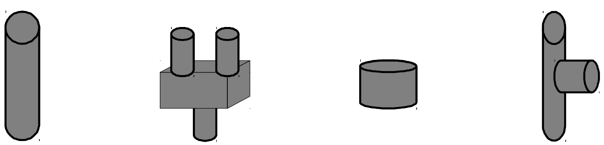
Jendrik Johannes and Uwe Aßmann, Concern-Based (de)composition of Model-Driven Software Development Processes. Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, 2010,Part II, Springer, 2010, LNCS 6395, URL = http://dx.doi.org/10.1007/978-3-642-16129-2

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

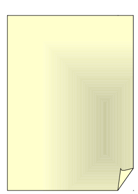
Composition Process in Grey-Box Composition Systems



Grey-box Components

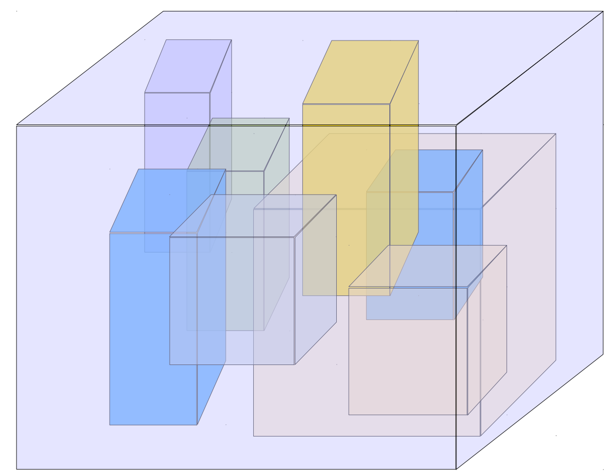


Composition Operators



Composition Recipe

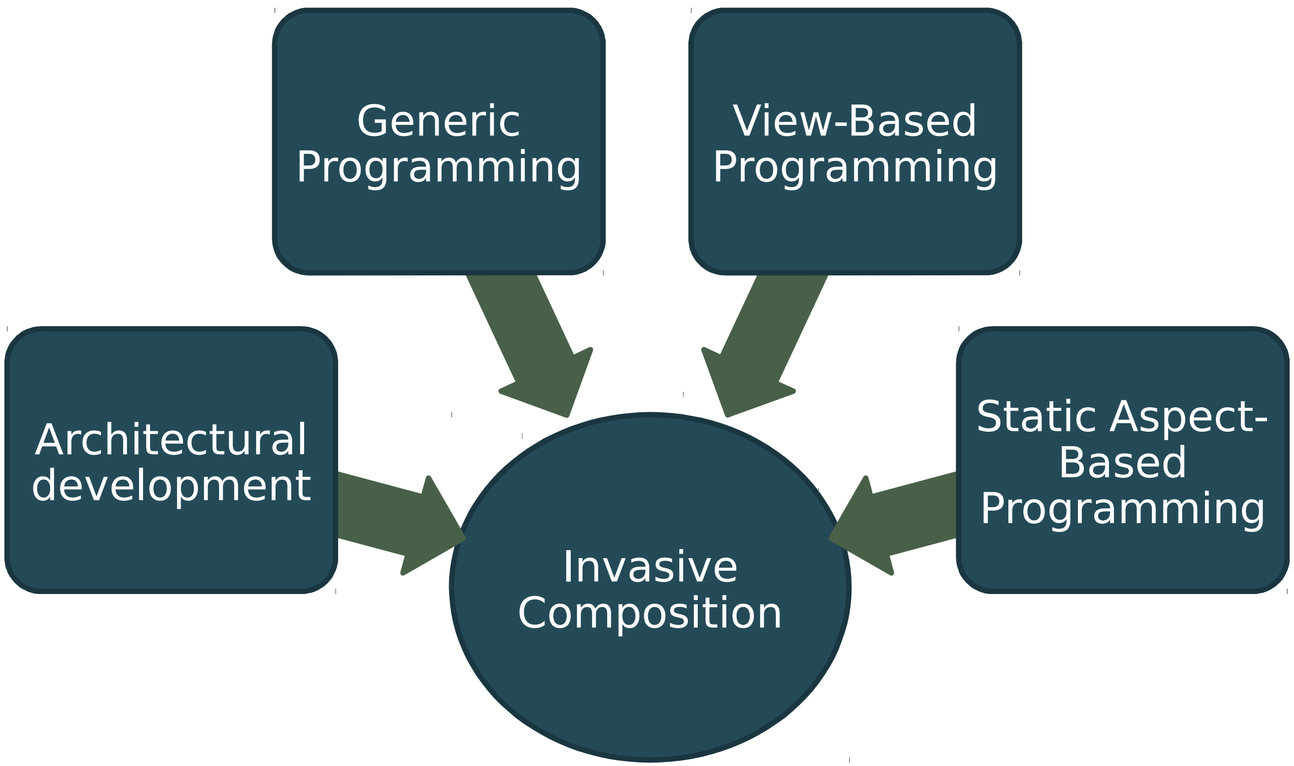
Invasive Software Composition



System Constructed with an Invasive Architecture

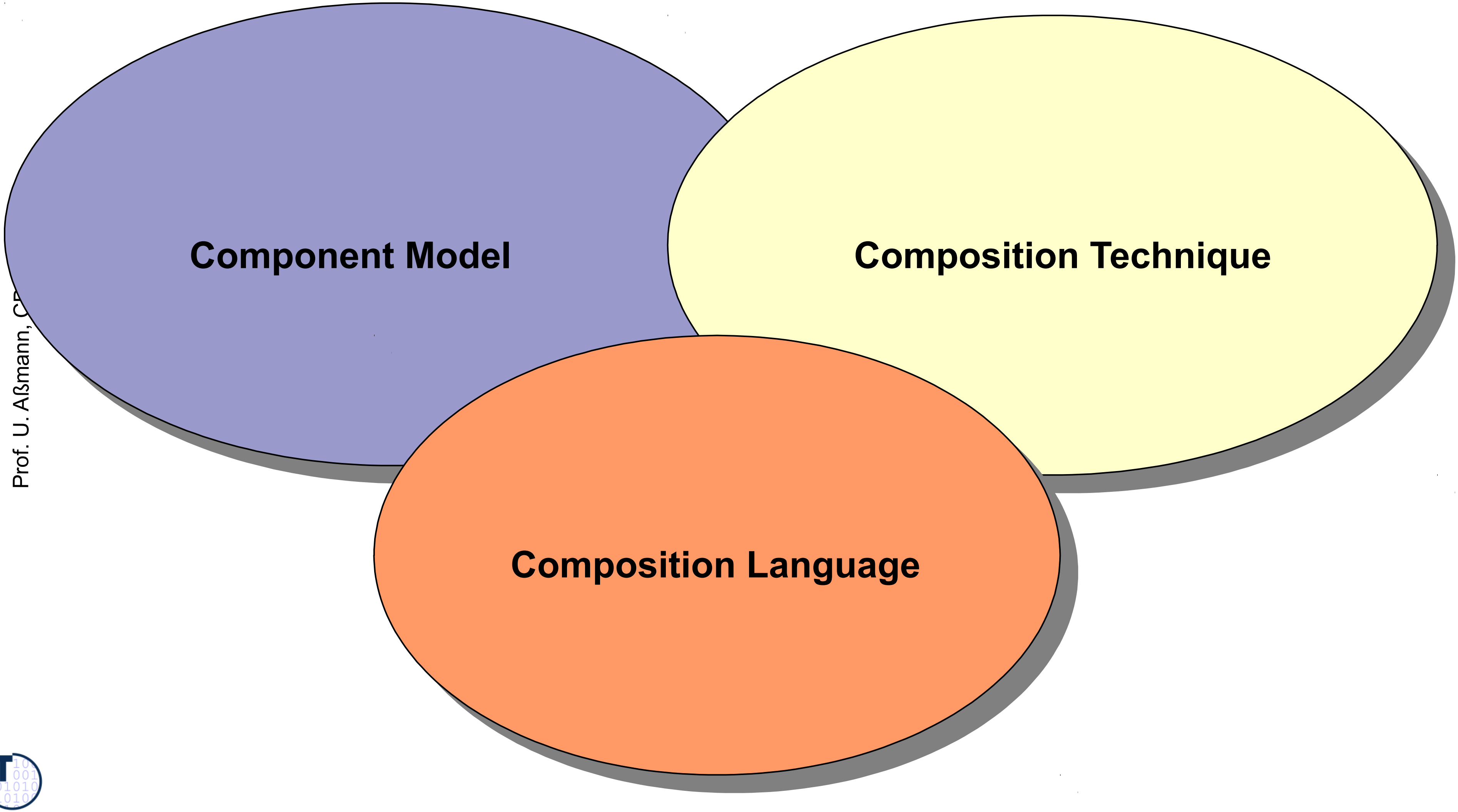
Invasive Software Composition

- Adds a full-fledged composition language to generic and view-based programming
- Combines architectural systems, generic, view-based and aspect-oriented programming



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

Software Composition

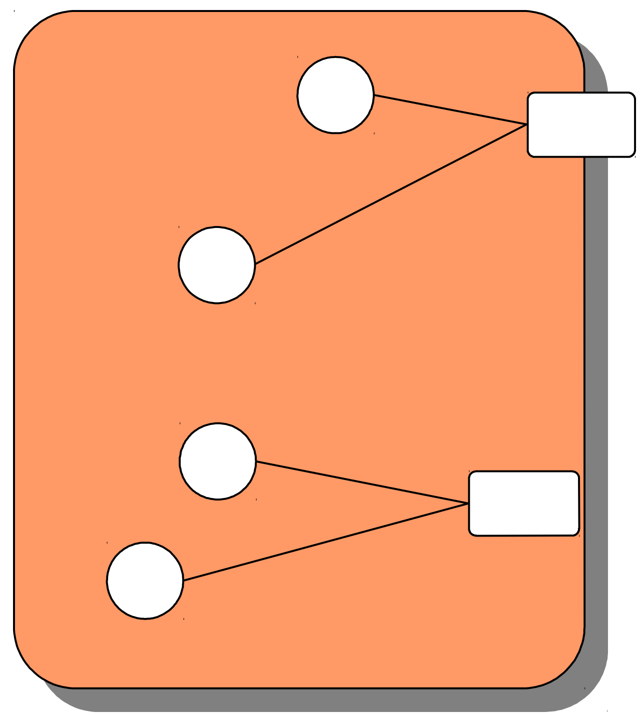


Prof. U. Aßmann, CB

Invasive Software Composition

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

- A **fragment component (snippet components)** is a fragment group (fragment container, fragment box) with a composition interface of change points
- A fragment component is a uniform container for
  - A plain fragment
    - a class, a package, a method
  - A generic fragment (group)
  - A fragment group
    - an advice or an aspect
    - a composition program



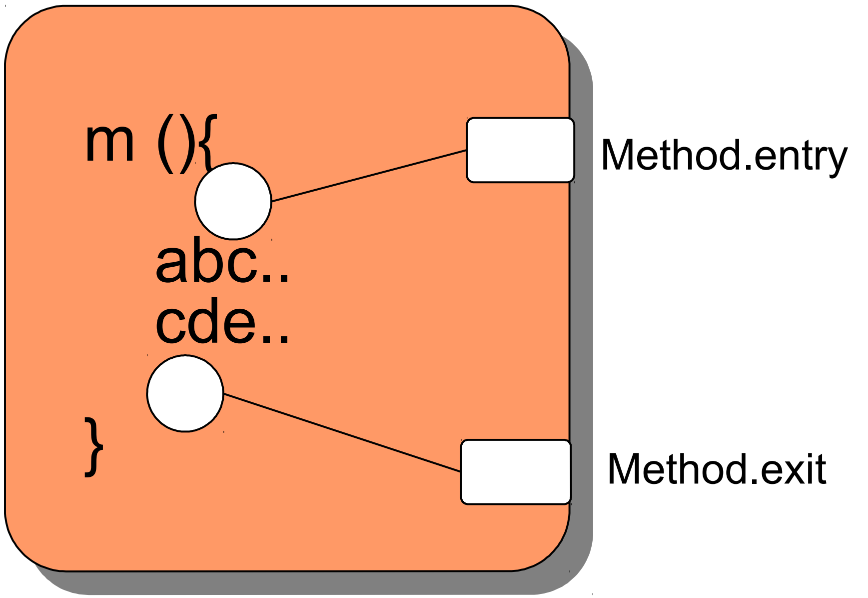
The Component Model of Invasive Composition

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)*
  - A *query point (out port)*
- Example:
  - Extension point: Method entries/exits
  - Variation point: Generic parameters
  - Query point: Contracts that can be queried

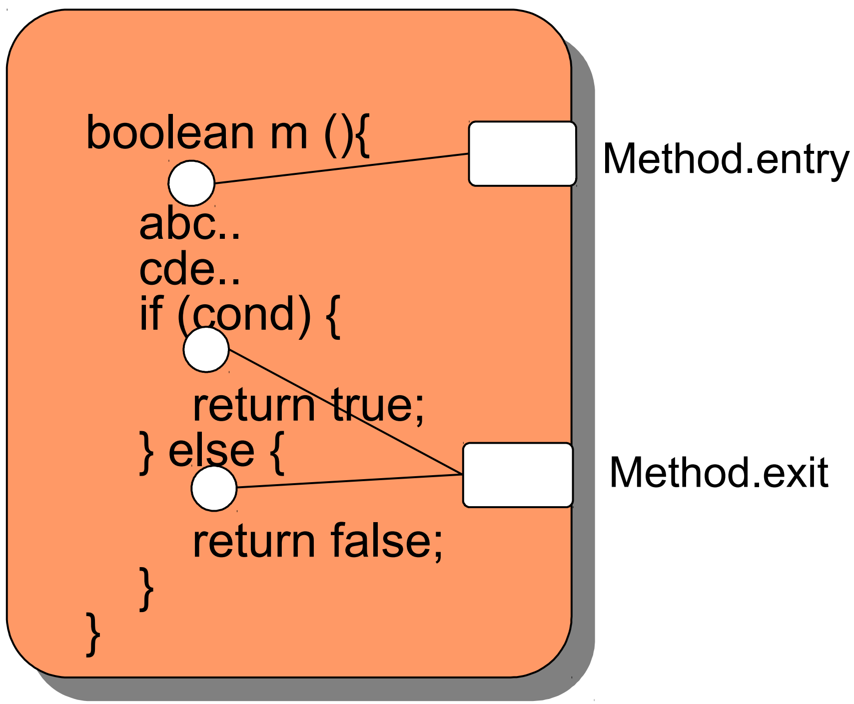
Hooks for Extension

- A hook is an extension point of a fragment component
- Hooks can be implicit or explicit (declared)
- An **implicit hook** is given by the component's language
  - We draw implicit hooks inside the component, at the border
  - Example: Method Entry/Exit
- An **explicit hook** is marked up by the component author
- Between hooks and their positions in the code, there is a **hook-fragment mapping**



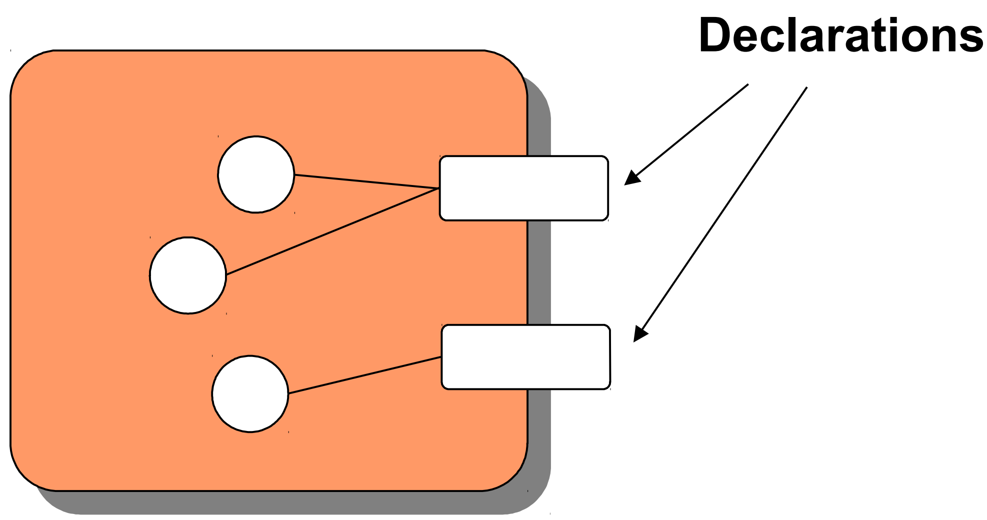
A Hook can Relate to Many Code Points

- A hook can relate to many code points (1:n-hook-fragment mapping)
- Example:
  - Method Entry refers to a code point at the beginning the the method
  - Method Exit refers to n code points *before* return statements



Slots for Parameterization (Declared Hooks)

- A **slot** is a variation point of a component, i.e., a code parameter
- Slots are most often *declared* (*explicit*), 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



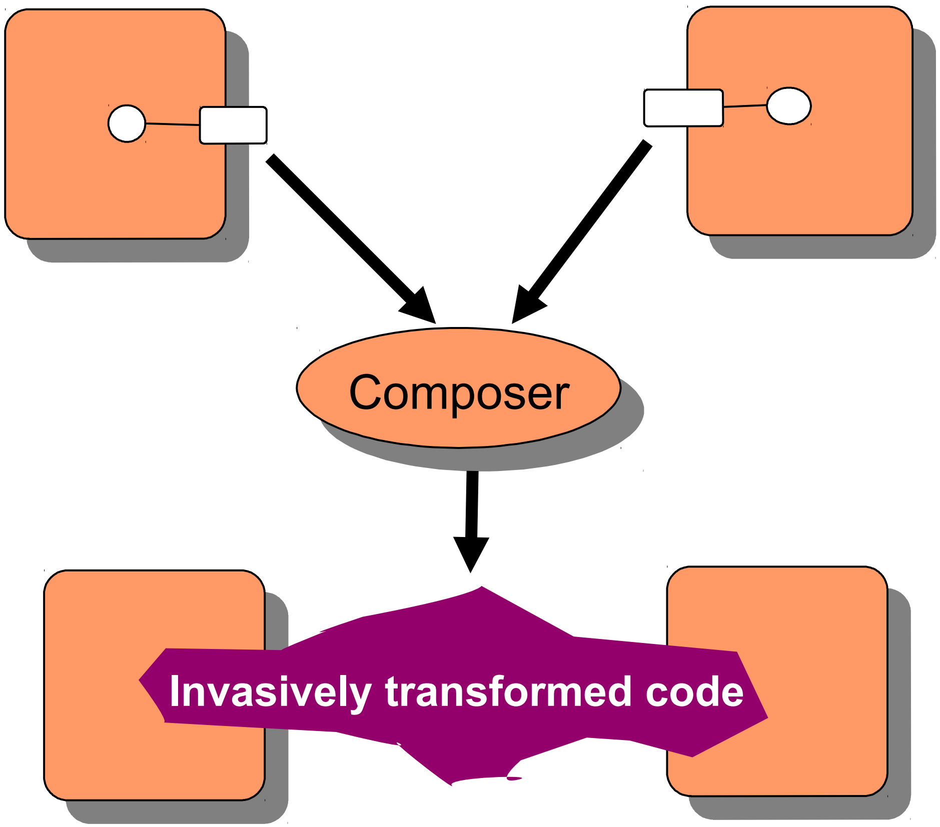
The Composition Technique of Invasive Composition

Invasive Software Composition
queries, parameterizes and extends
fragment components
at implicit and declared change points (hooks and slots)
by transformation

An invasive composition operator treats
declared and implicit slots, hooks, and query points
uniformly

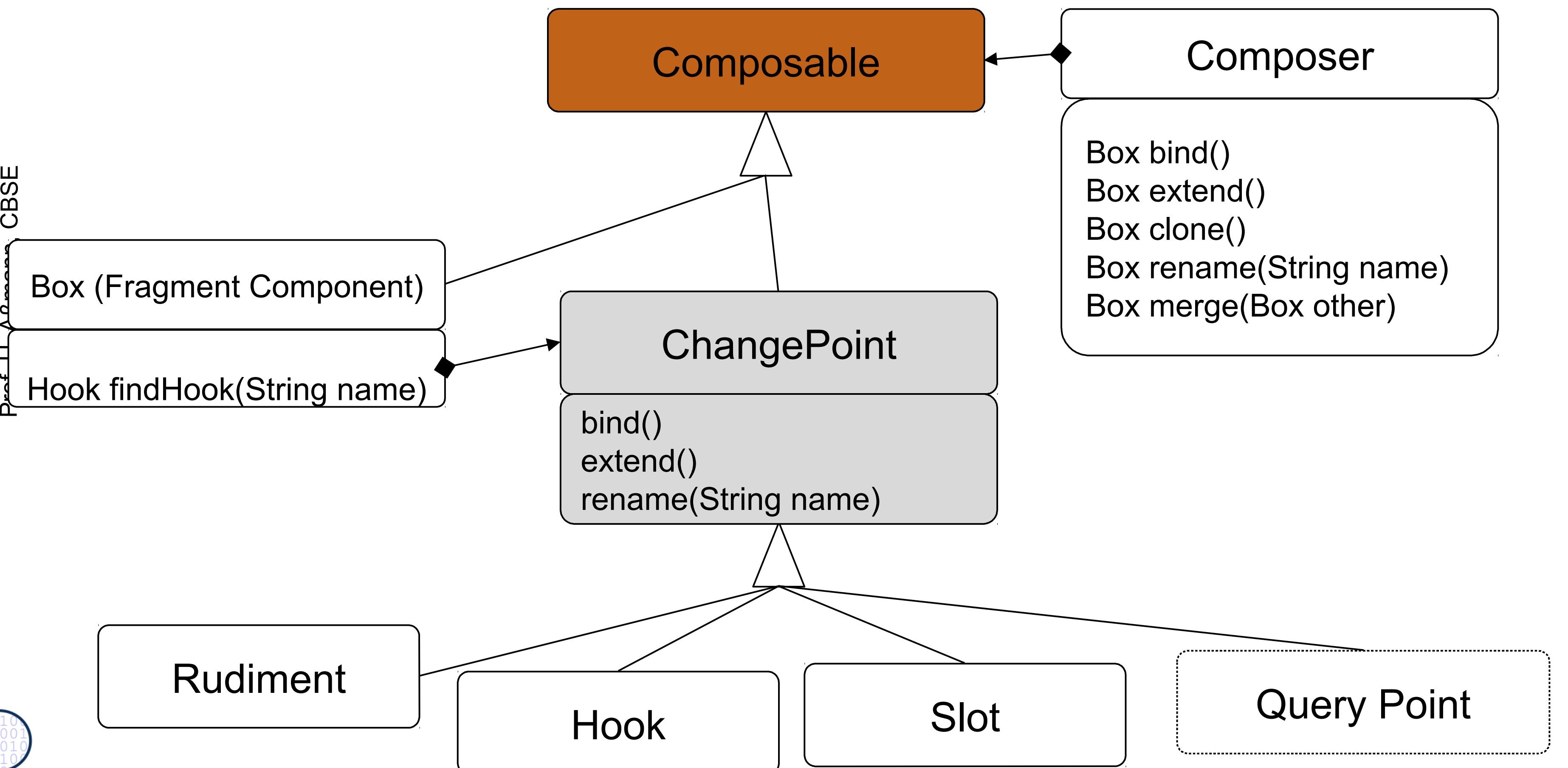
The Composition Technique of Invasive Composition

- A **composer (composition operator)** is a static metaprogram (program transformer) modifying a slot or hook of a fragment component



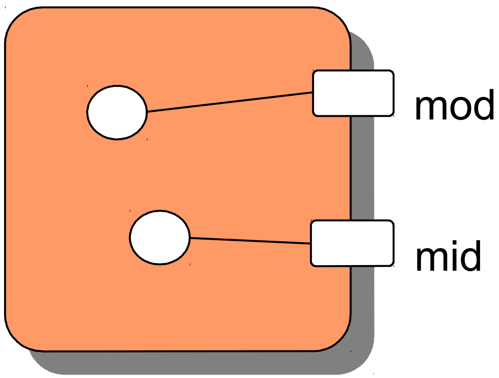
Object-Oriented Metamodeling of Composers

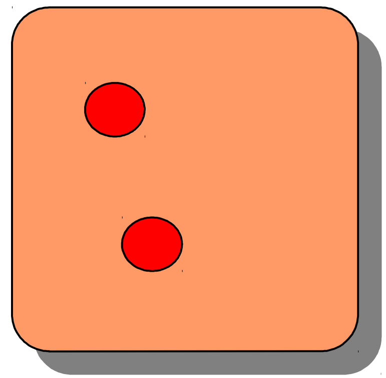
- In the following, we assume an object-oriented metamodel of fragment components, composers, and composition languages.
- The COMPOST library [ISC] has such a metamodel (in Java)
- Composers work on Composables (Changepoints or Boxes)



Bind Composer Parameterizes Fragment Components at Slots

- Like in BETA, for uniformly generic components



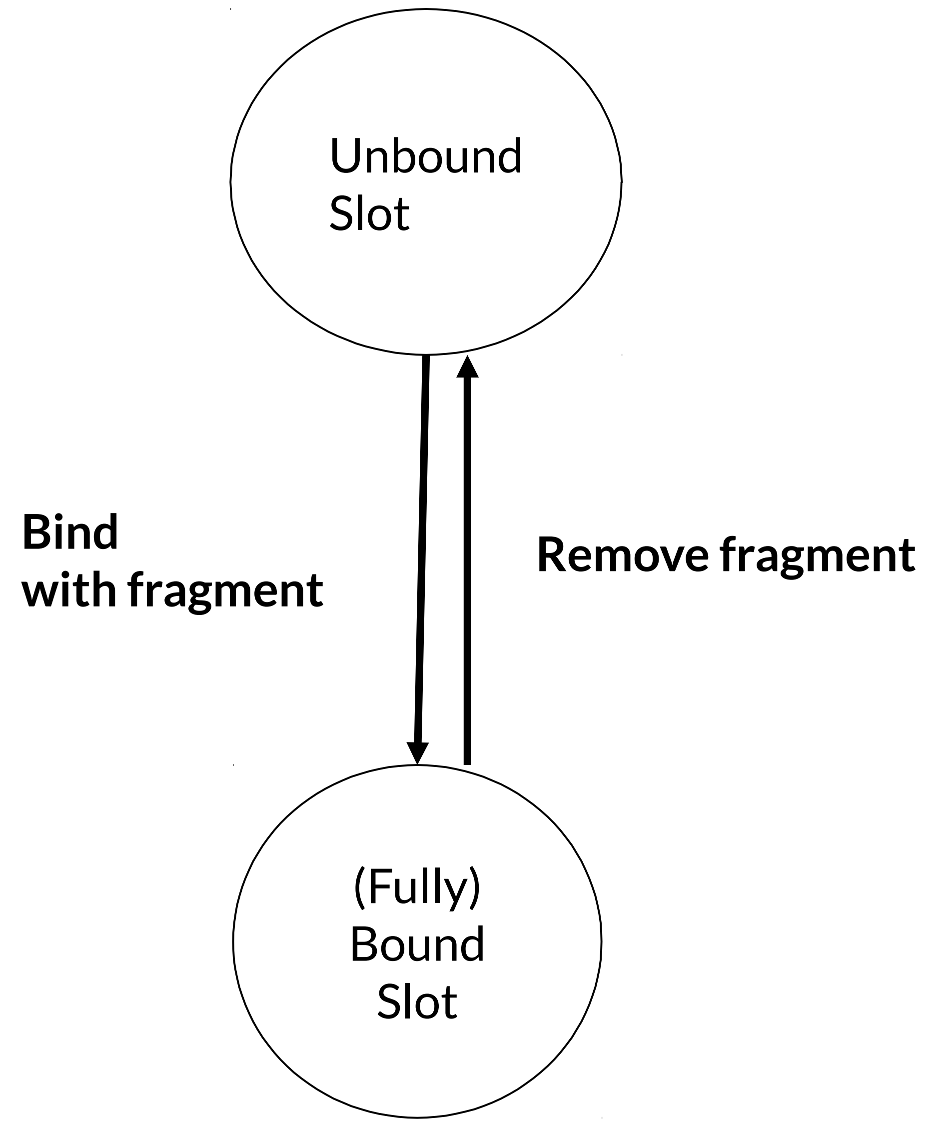


```
<<mod:Modifier>>
m (){

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

}
```

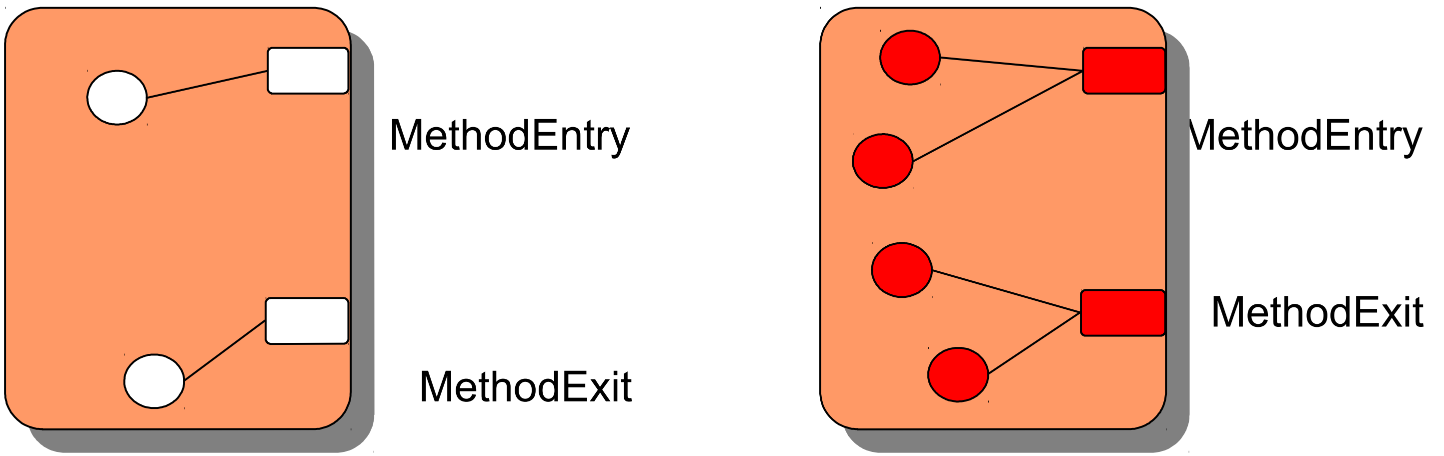
```
synchronized m (){
    abc..
    f();
    cde..
}
```

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

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

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

Slot Binding State Diagram



Extend Operator Universally Extends the Fragment Components at List Hooks




```
m (){

    abc..
    cde..

}
```

→

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

```
Box component = readBoxFromFile(“m.java”);

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

component.findHook(„MethodExit“).extend(“print(\”exit m\”);”);
```

Merge Operator Provides Universal Symmetric Merge

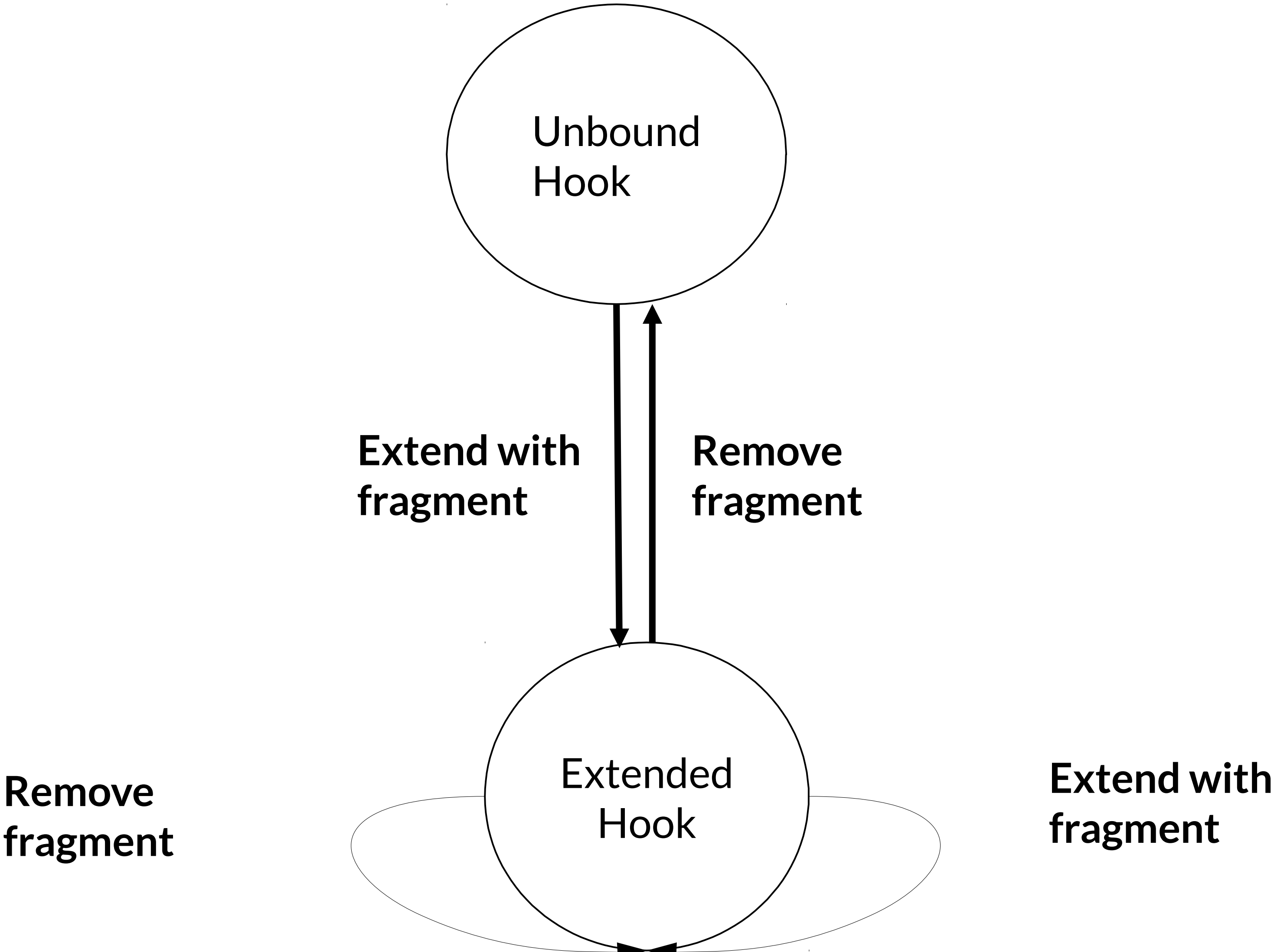
- The **Extend** operator is asymmetric, i.e., extends hooks of a fragment component with new fragment values
- Based on this, a **symmetric Merge** operator can be defined:

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

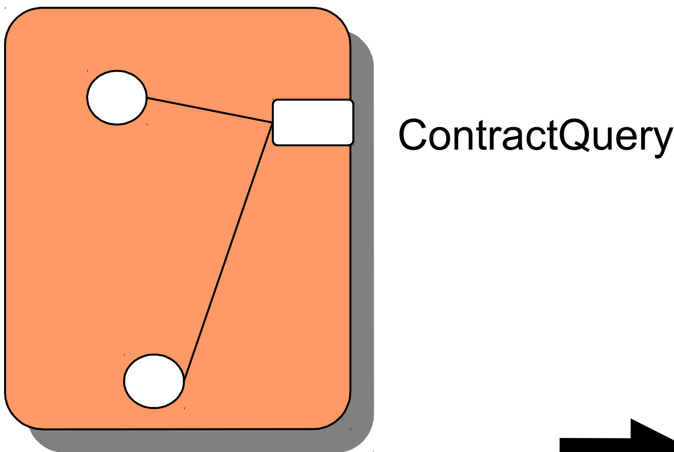
- where list is a list of inner components, inner fragments, etc.

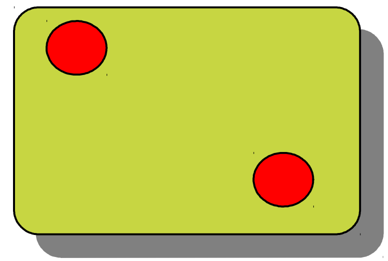
- Both `extend(f)` and `merge(f,g)` work on fragments
  - Extend works on all collection-like language constructs
  - Merge on components with collection-like language constructs

Hook Extension State Diagram



Query Operator Delivers Fragments out of the Fragment Component



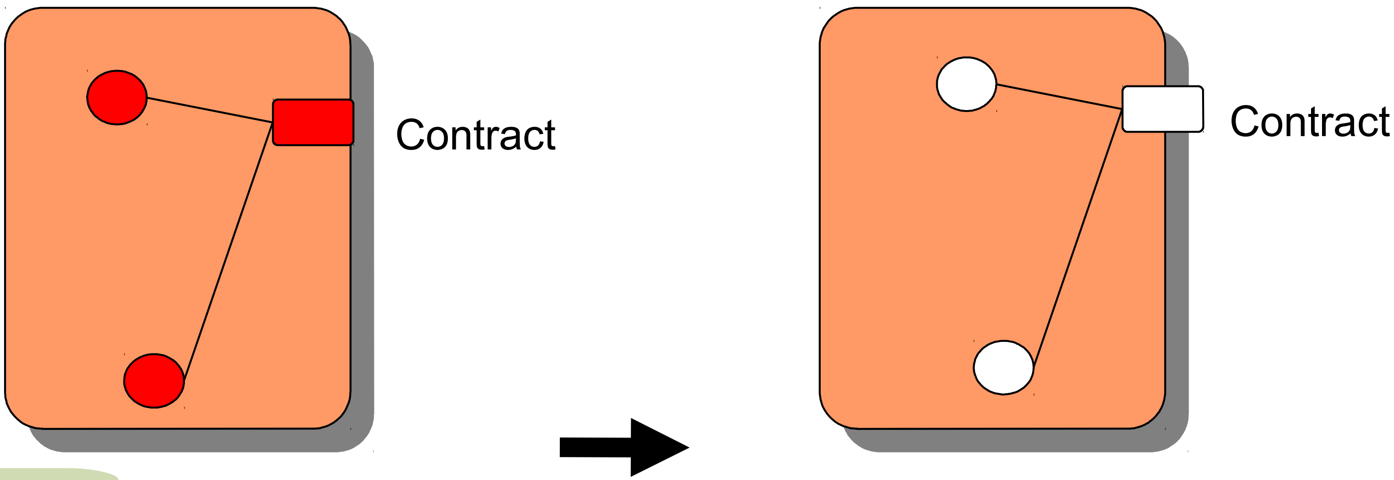


```
int findoutAge(Person p){
    if (p == null) return 19;
    abc..
    result = cde..
    if (result == 0) return 10;
}
```

```
{
if (p == null) return 19;


if (result == 0) return 10;
}
```

```
"{ if (p == null) return 19;
if (result == 0) return 10;}" ←
component.findHook(„ContractQuery").query();
```

Remove Operator Removes Rudiment Fragments out of the Fragment Component



Rudiment

```
int findoutAge(Person p){
    if (p == null) return 19;
    abc..
    result = cde..
    if (result == 0) return 10;
}
```

```
int findoutAge(Person p){

    abc..
    result = cde..

}
```

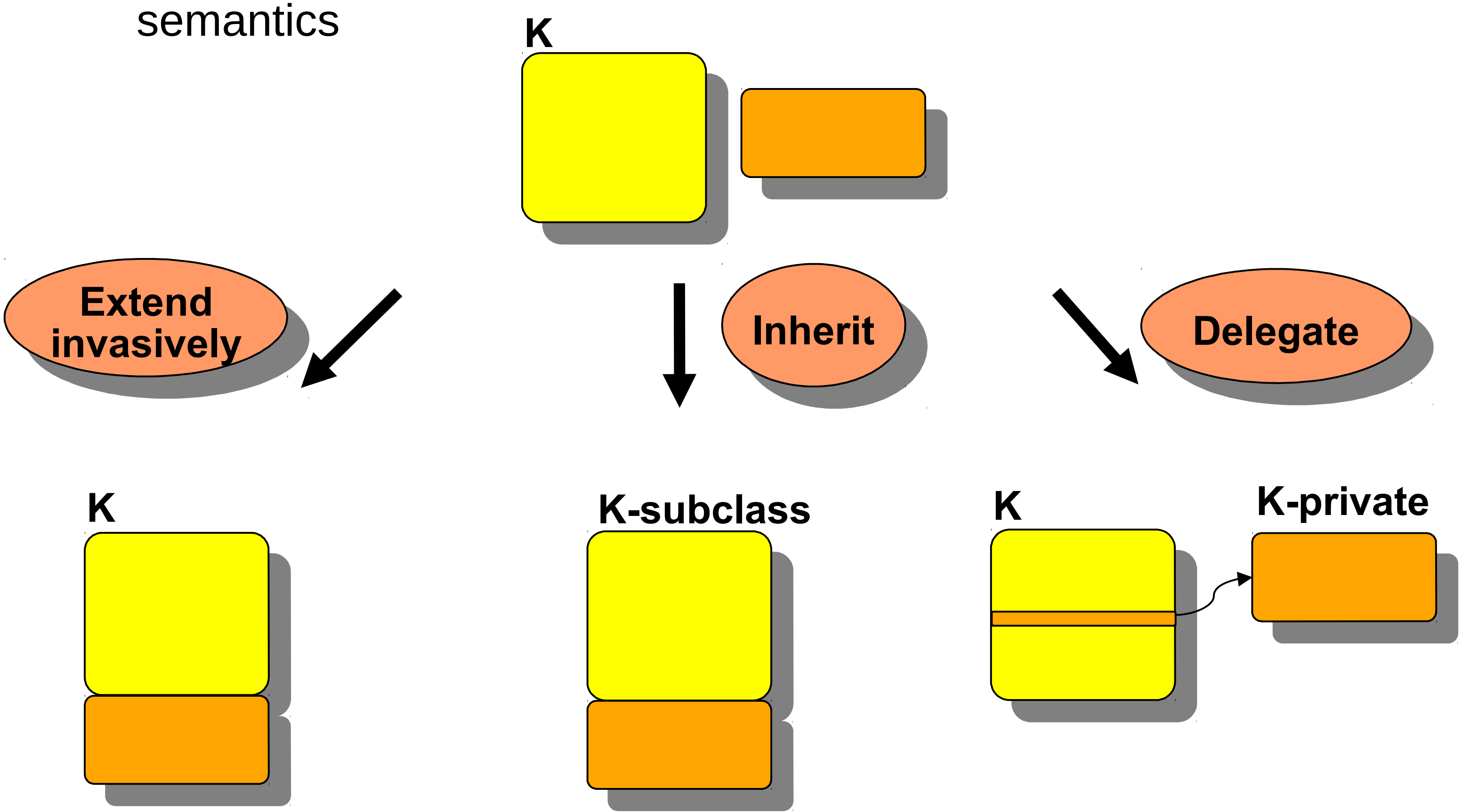
component.findHook(„ContractQuery“).remove();

Basic Composition Operators

Approach	Composables	Composers	Variation/Extension points
	Components	extend	Implicit member list
		merge	Open definitions
	Slots	bind	Variation point
		unbind	
	Hooks	extend	Extension point
	Query port	query	Query point
	Rudiment	remove	

Applied to Classes, Invasive Extension Integrates Feature Groups

- The Extend operator integrates feature groups and roles into classes
  - Delegatee merge: because a delegatee can be merged with delegator
  - Role merge: because a feature group can play a role
- The semantics of extension lies between inheritance and delegation
- This leads to **class caluli** with many inheritance operators with specific semantics



Class Calculi

- [Gilad Bracha and William Cook. Mixin-based inheritance. In N. Meyrowitz, editor, Proceedings of the OOPSLA ECOOP '90, number 25(10) in ACM SIGPLAN NOTICES, pages 303--311. ACM Press, 1990.]
- The CoSy Data Definition Language for data in the repository (fSDL) is a class calculus language
  - [H.R. Walters, J.F.Th. Kamperman and T.B.Dinesh. An extensible language for the generation of parallel data manipulation and control packages. Computer Science/Department of Software Technology. CS-R9575 1995 http://oai.cwi.nl/oai/asset/4931/4931D.pdf]
- A **class calculus** is an algebra with composition operators over classes
  - Different forms of sharing (inheritance) operators (e.g., mixins, generics)
  - Merge operators
    - Sum of classes (+)
  - Associative and commutative operators
  - Distribution operators
    - Product of classes (*)
    - Wrapping of classes
  - Projection operators
    - Differencing of classes
    - Projection of classes

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

- Adaptation of static relations
  - Inheritance relationship: multiple and mixin inheritance
  - Delegation relationship: When delegation pointers have to be inserted
  - Import relationship of packages
  - Definition/use relationships (adding a definition for a use)
  - Type-safe template expansion: When templates have to be expanded 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

46.1.2 Composition Languages

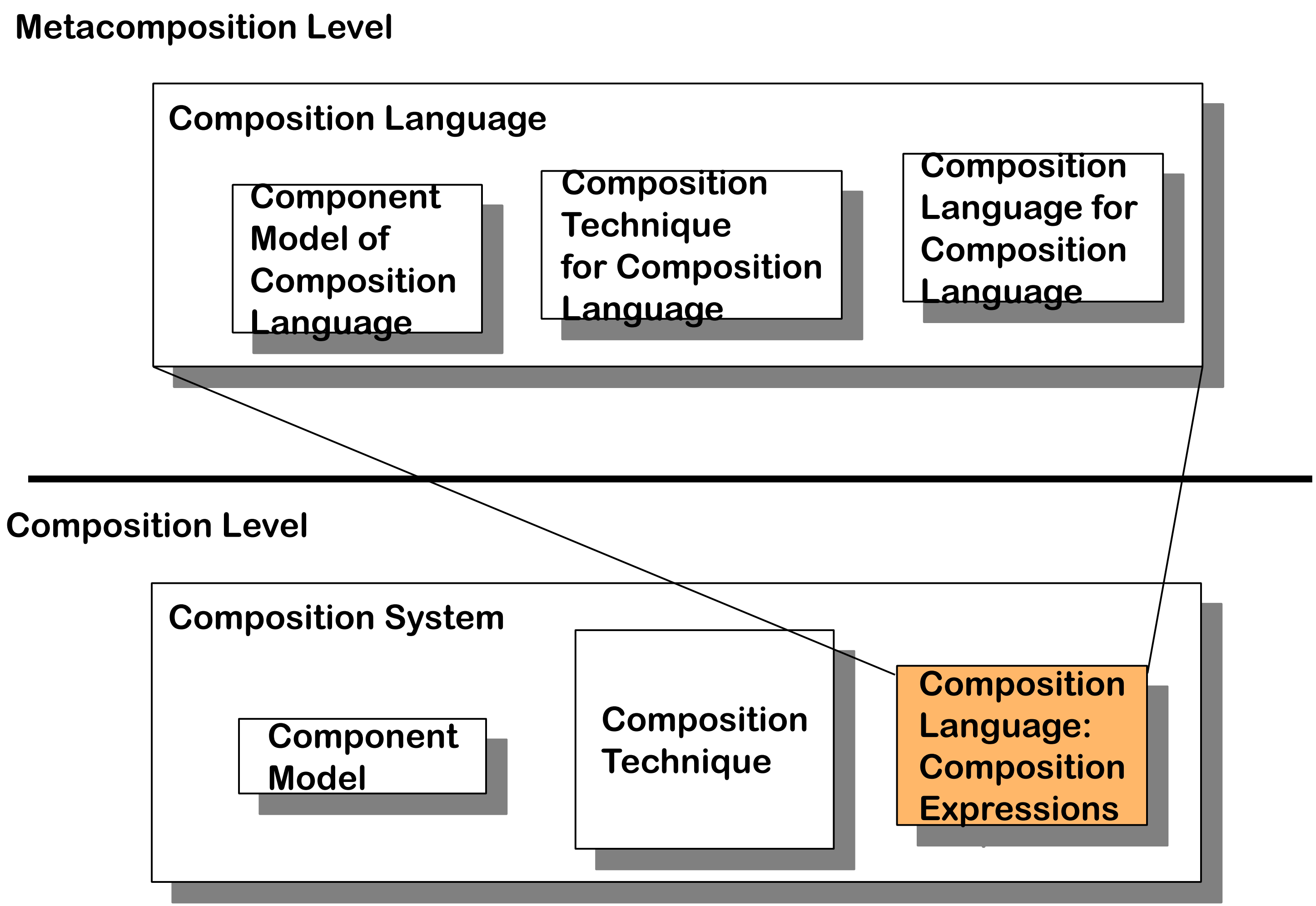
Composition Programs and Their Languages

Basically, every language may act as a composition language, if its supports basic composers like *bind, query,* and *extend.*

- Imperative languages: Java (used in COMPOST and Reusewair), 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 (used in SkAT), Rewrite Systems

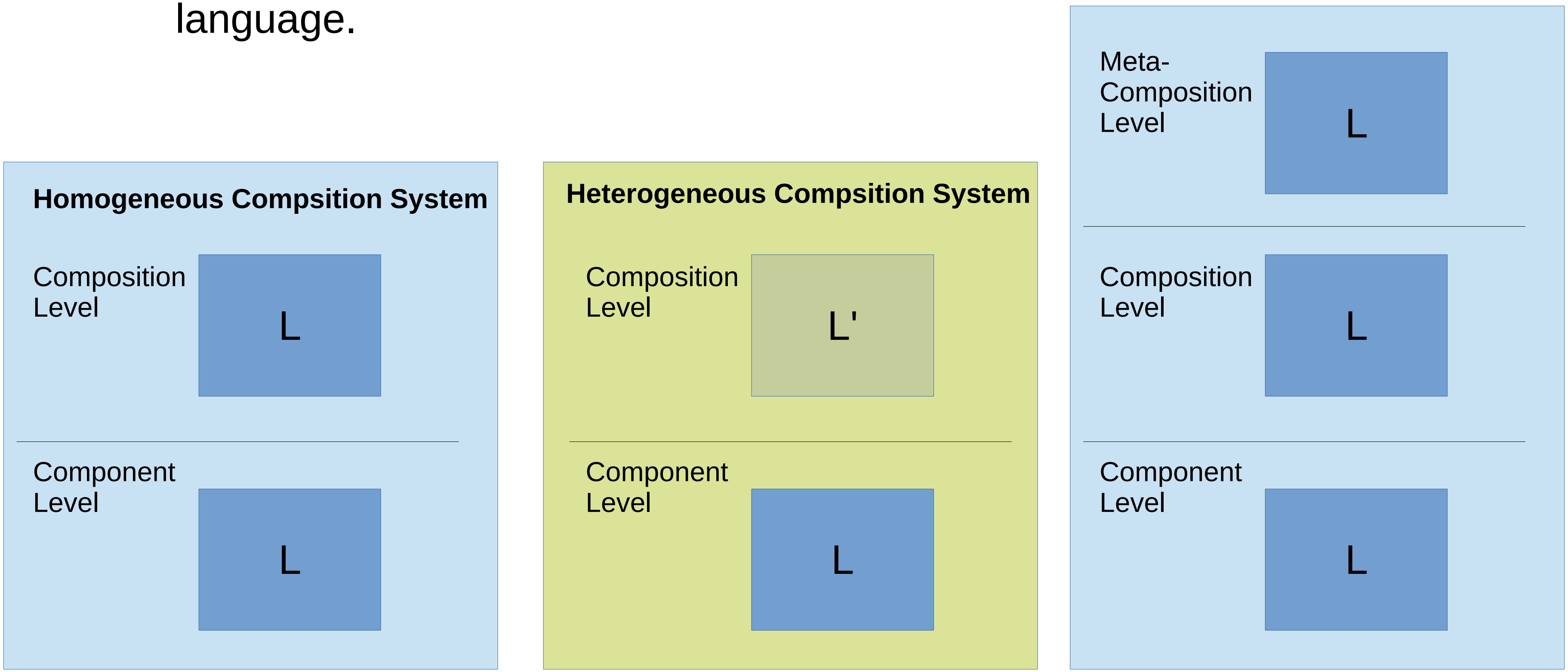
Q2: Component and Composition Language Level

- Acyclic composition programs form *composition expressions*
- Configuration of component systems
- Holds for both black-box and grey-box composition systems



Homogeneous Composition Systems

- A **homogeneous composition system** employs the same composition language and component language.
  - Otherwise, it is called **heterogeneous**
- In a homogeneous composition system, metacomposition is staged composition.
- A point-cut language (cross-cut language) is a simple composition language.

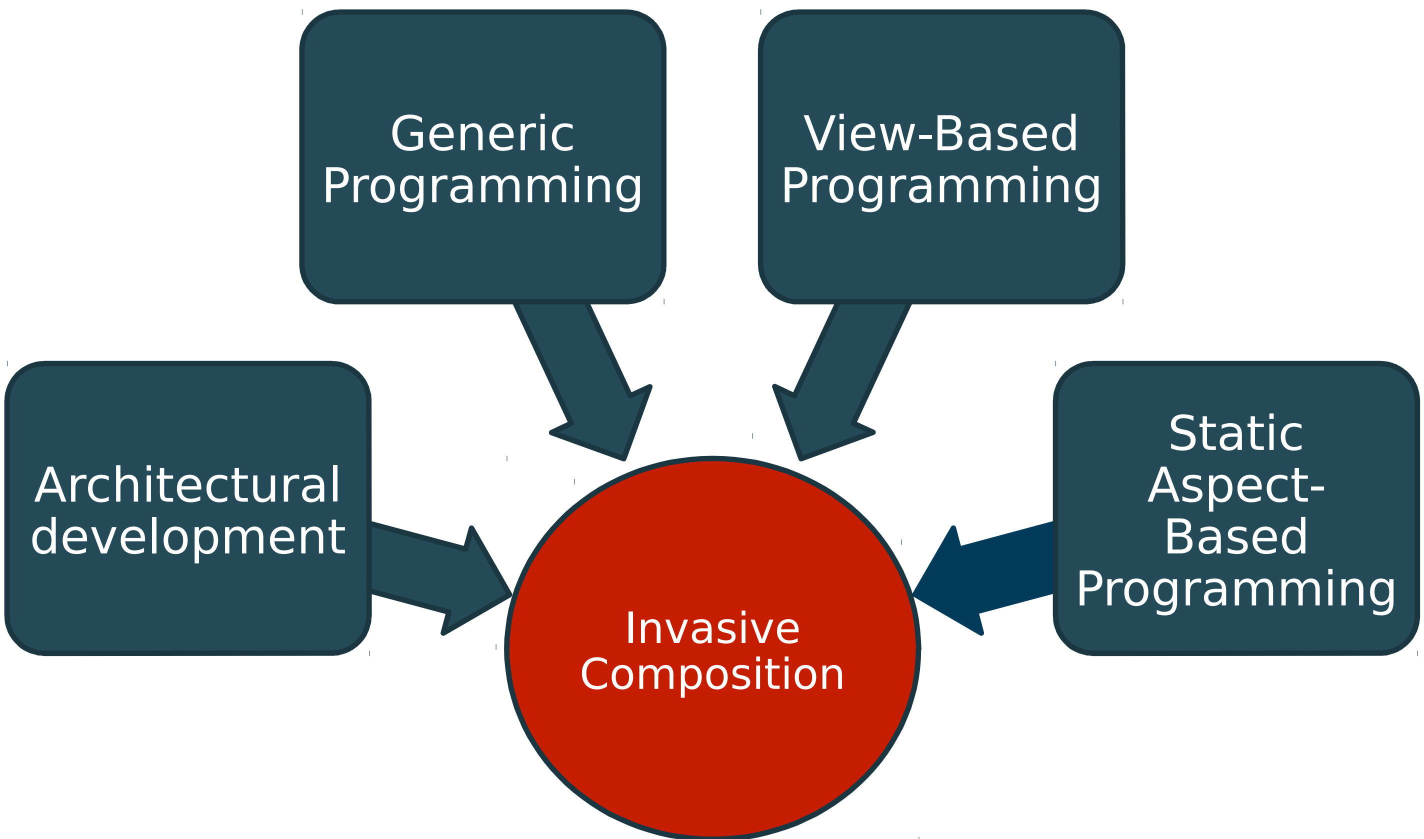


46.2. What Can You Do With Invasive Composition?

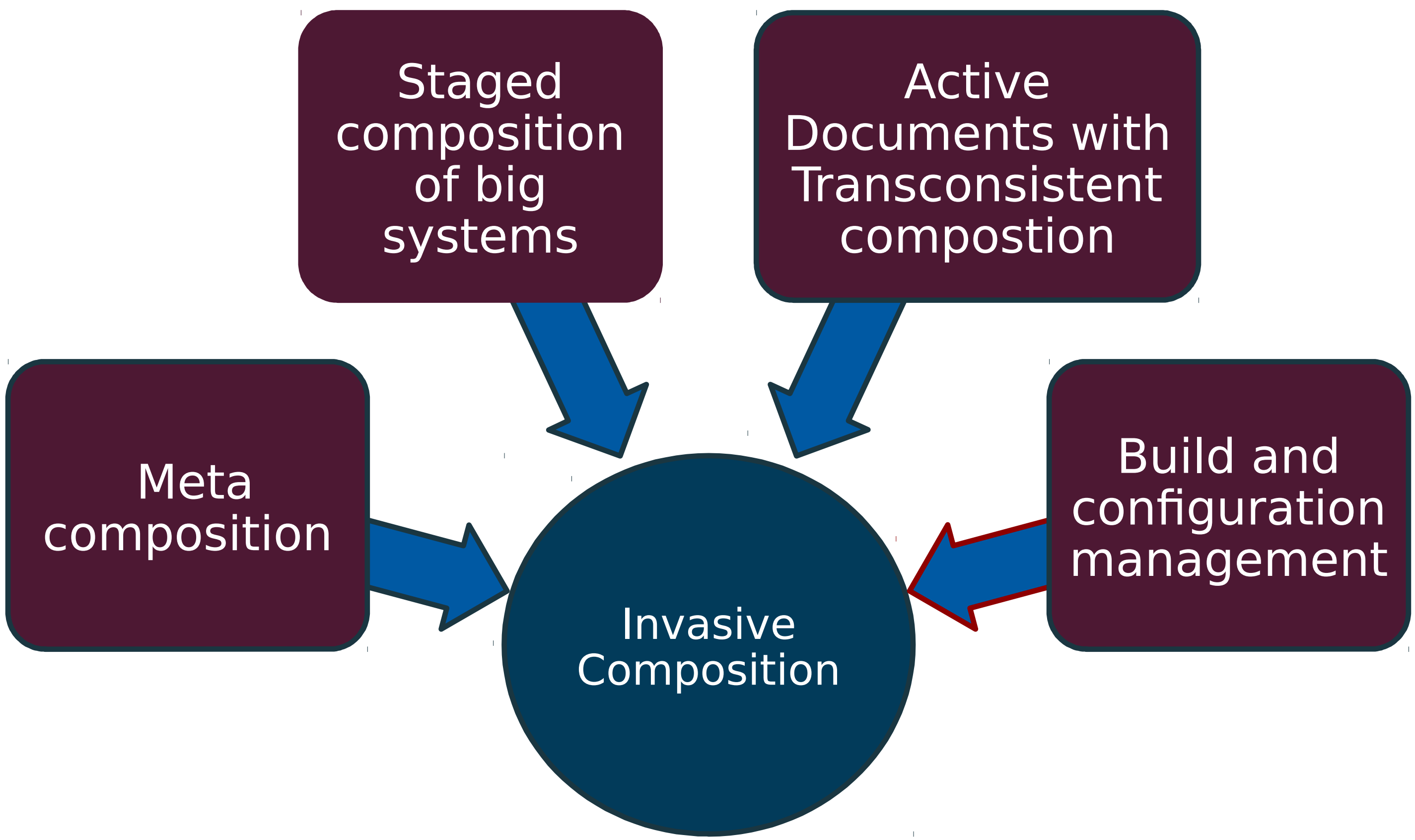
Invasive Composition

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

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



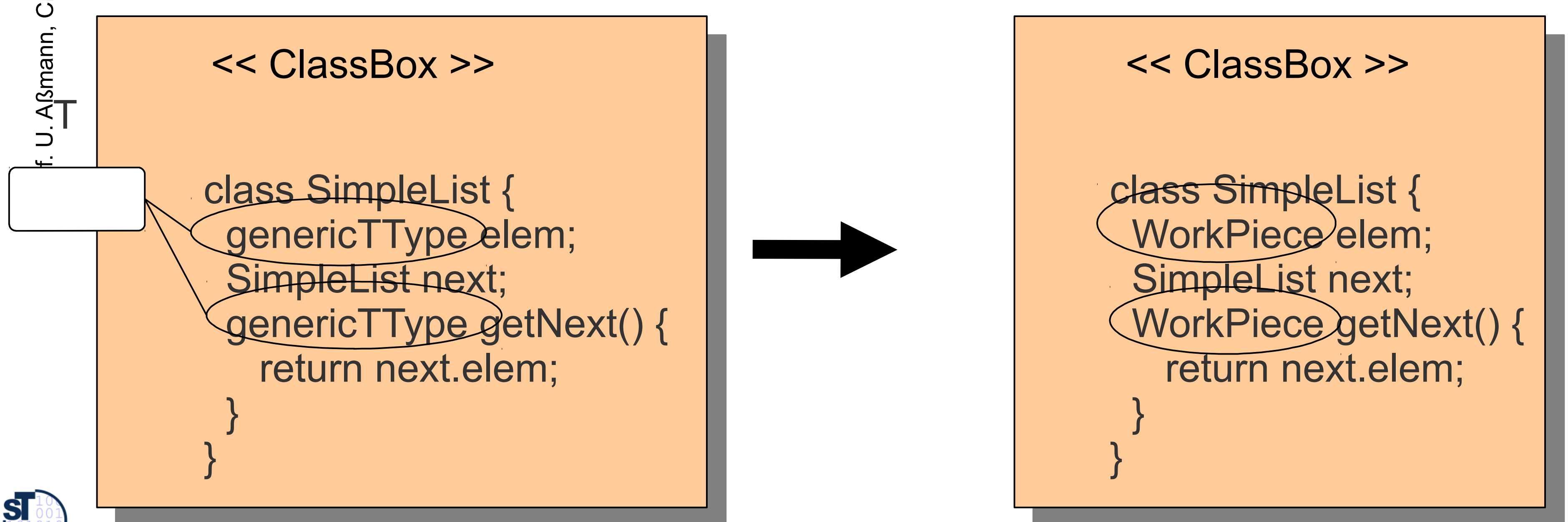
Advanced Applications of Invasive Composition



Components	Composers	Change points
Generic fragments	bind	Slots
Fragments	extend	Hooks
Architectural Components	Connectors, Invasive connectors Encapsulation operators	Ports
Classes	Mixin operators, inheritance operators	Class member lists
Views	Merge operators, extend operators	Open definitions
Core, aspectual components	Weaver (distributor, complex extender)	Join points

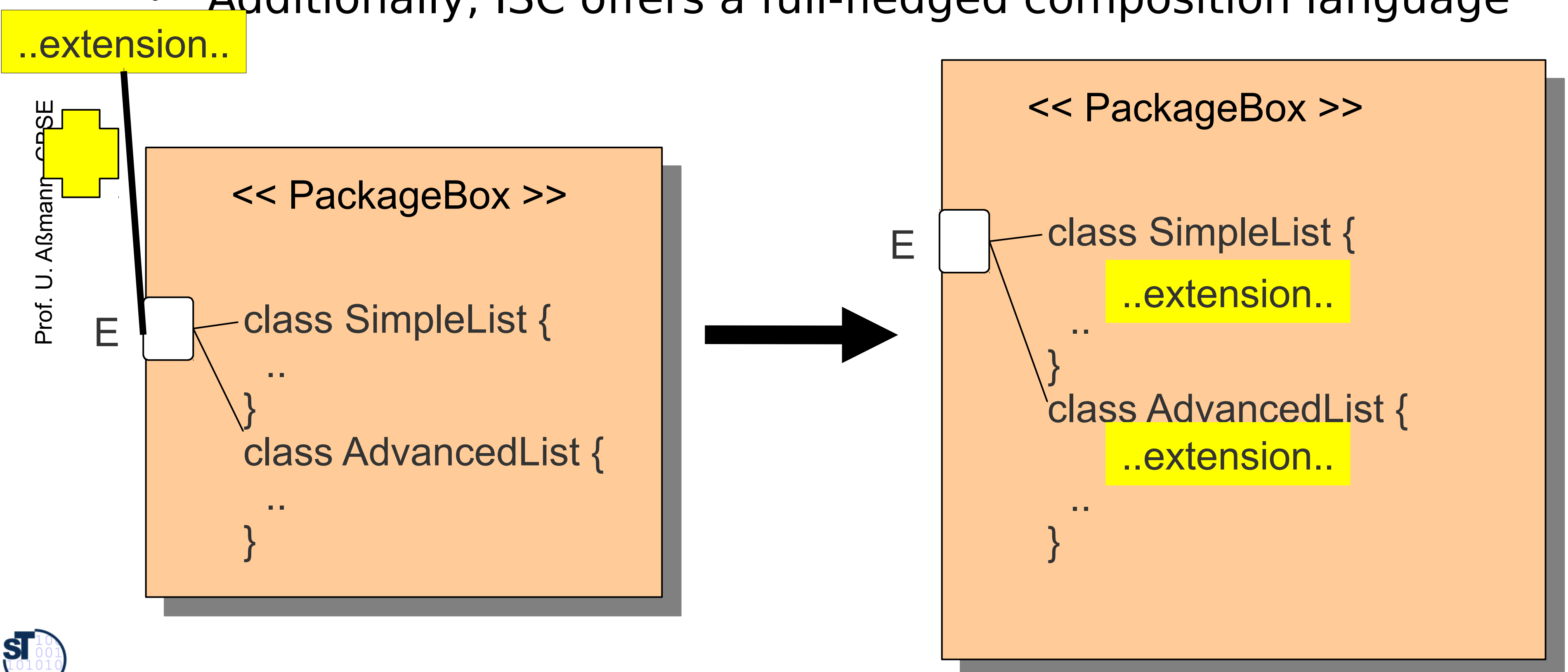
Universally Generic Programming

- 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, ...)



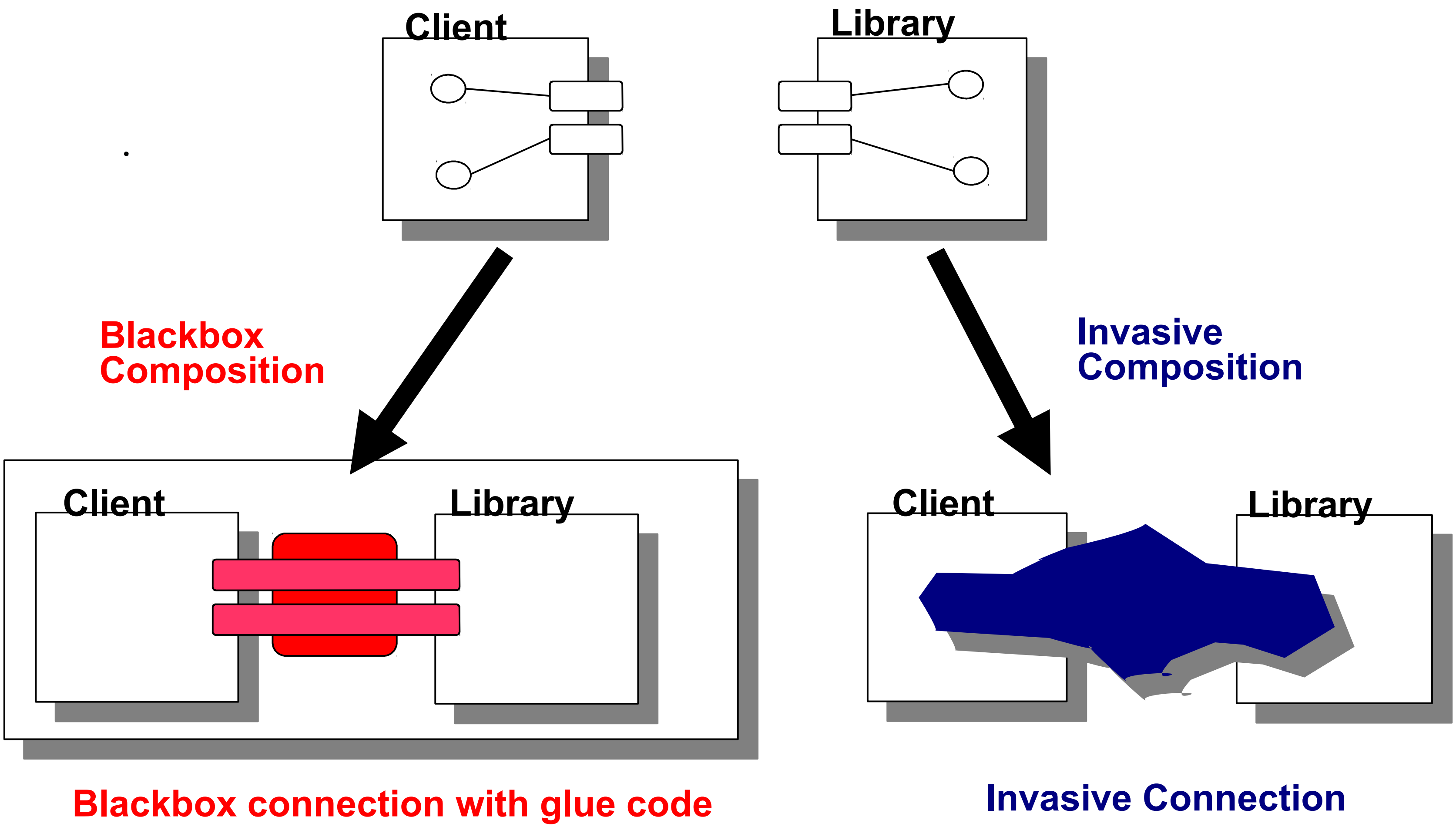
Universal Constructive View Programming

- 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

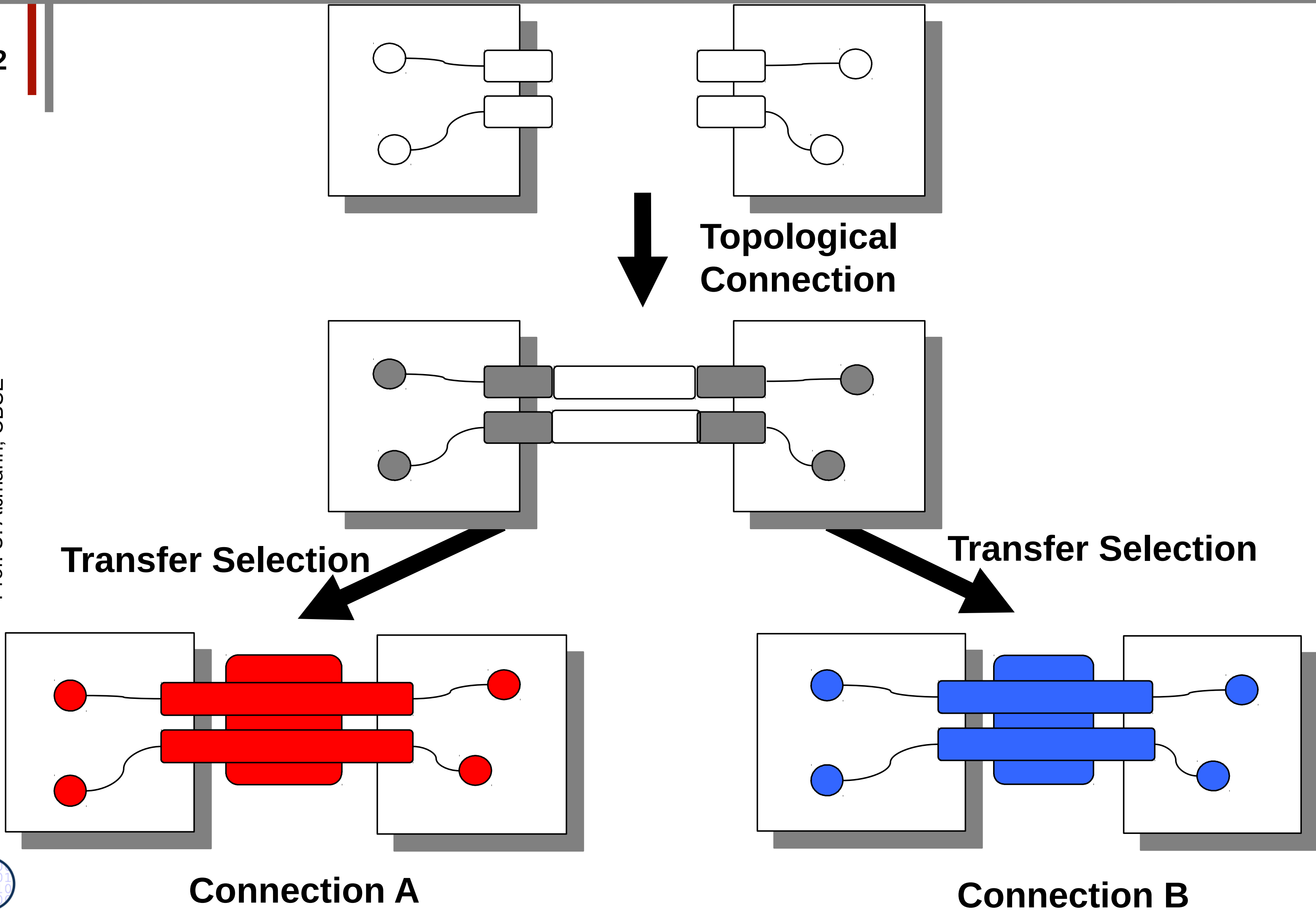
- In contrast to ADL, ISC offers invasive connections [AG00]
- Modification of static relationships between program elements possible (inheritance, delegation relations)



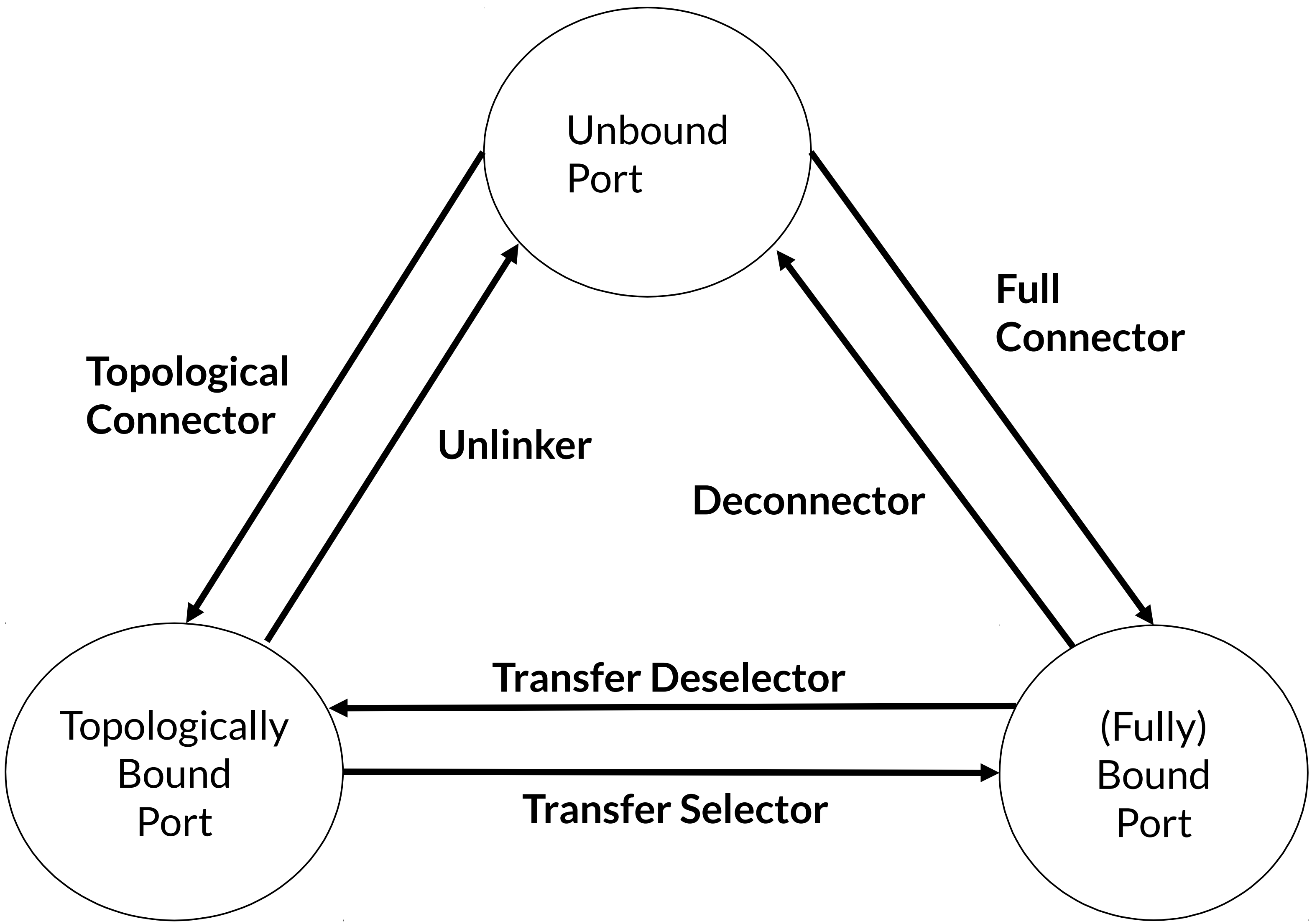
Invasive Architectural Programming

- [ISC] shows how invasive connectors achieve tightly integrated systems by embedding the glue code into senders and receiver components
  - Separation of topological and transfer connectors

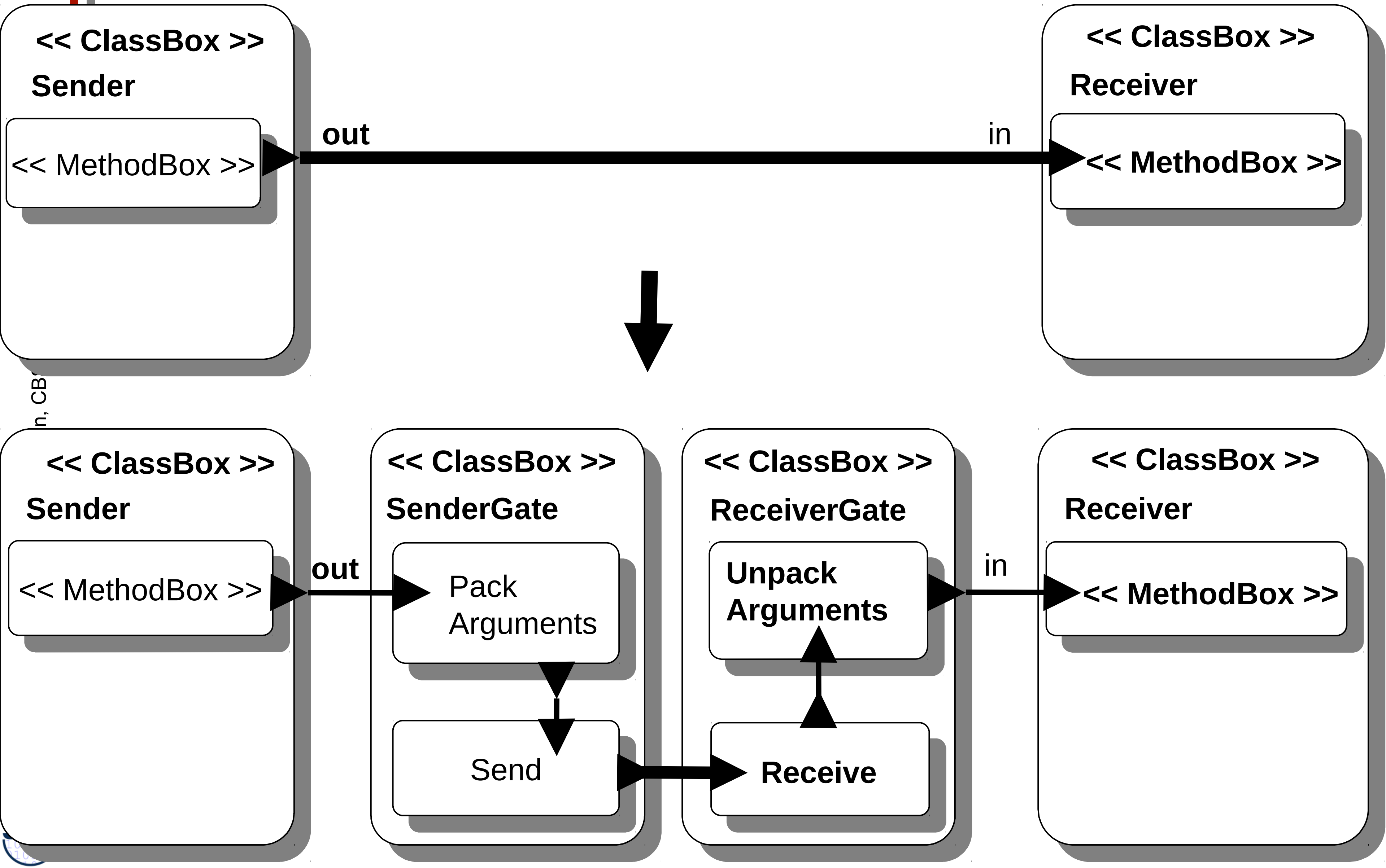
Separation of Topological from Transfer Aspect



Port Binding State Diagram of an Invasive Connector

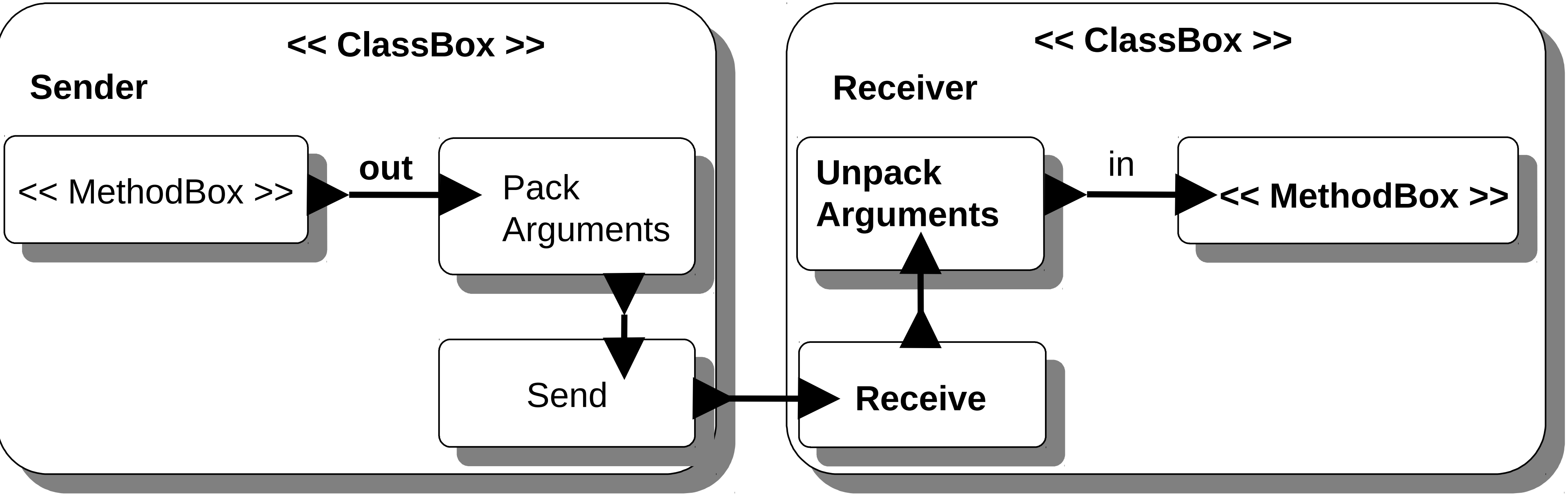


Gate Objects: Glue Separate



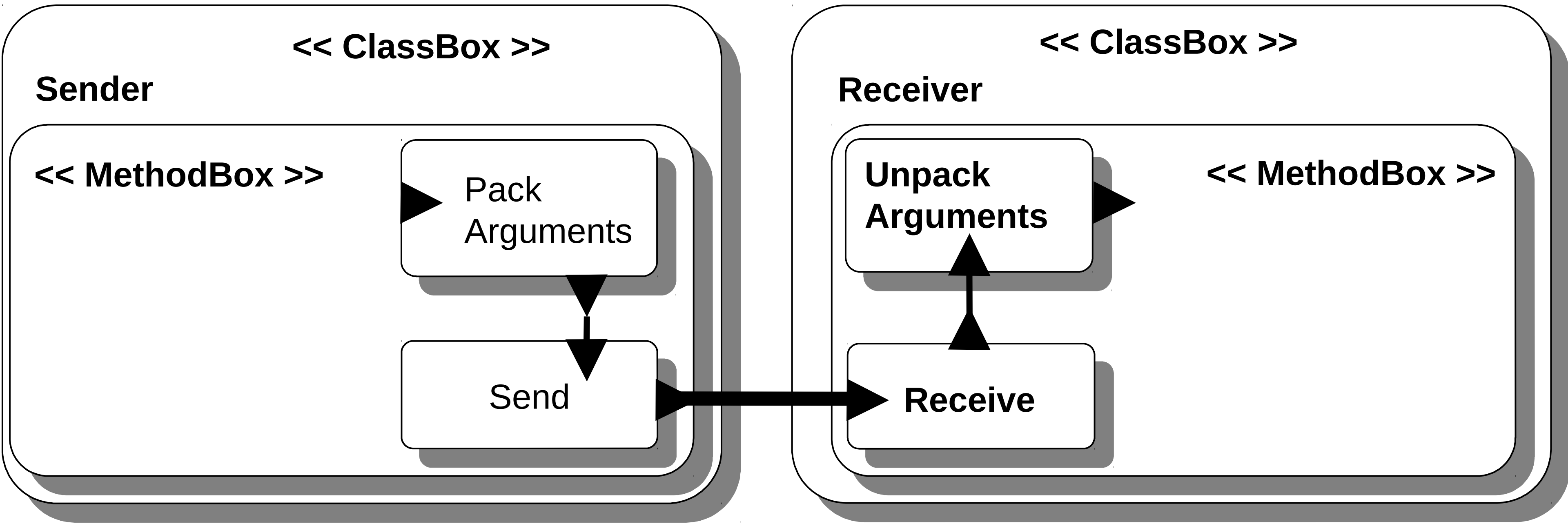
Invasive Connection

- Embedding communication gate methods into a class



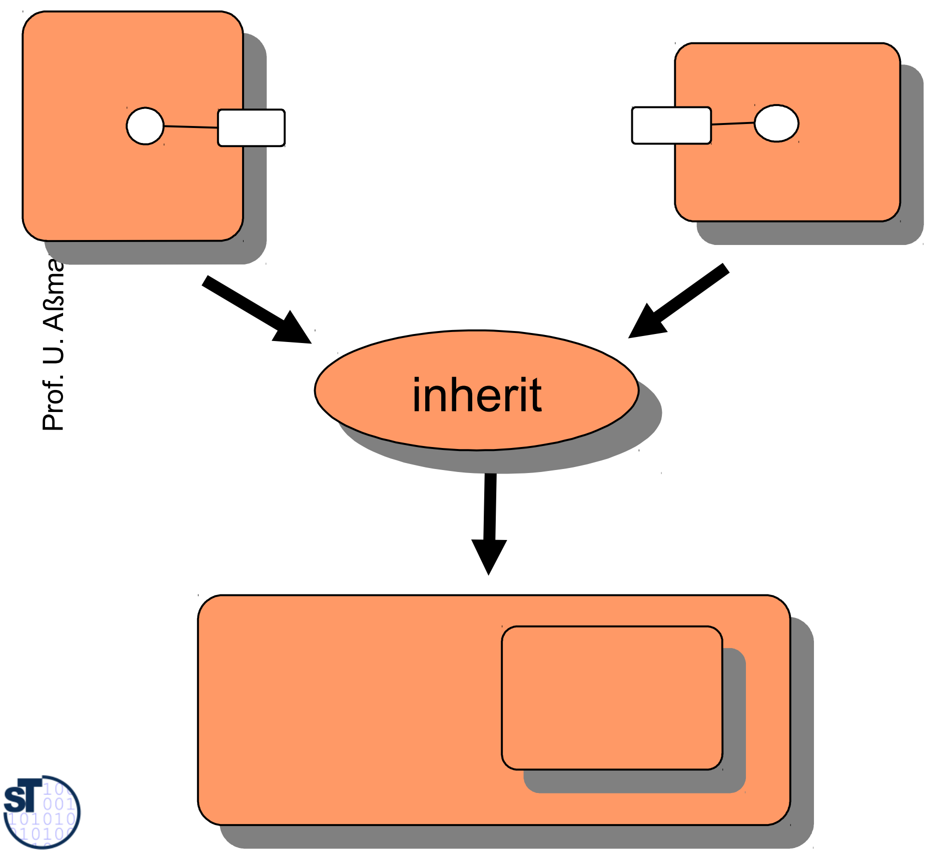
Invasive Connection

- Embedding glue code into sender methods



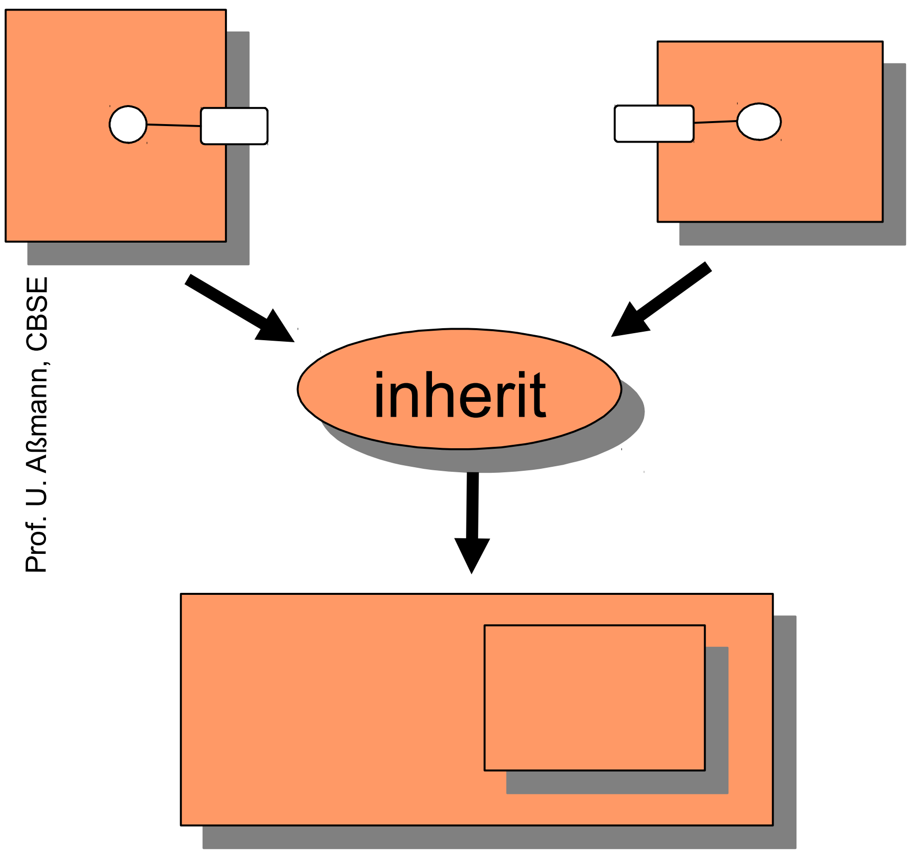
Class Calculi: Universal Inheritance and Mixins

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



- **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

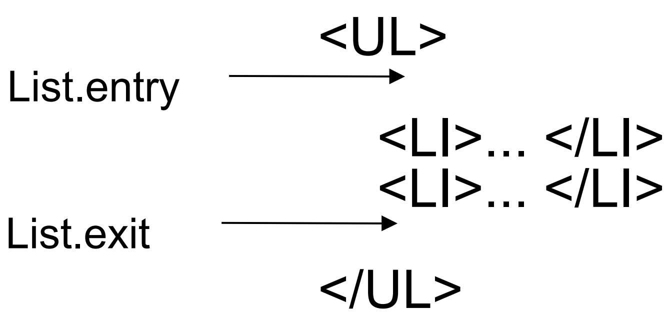


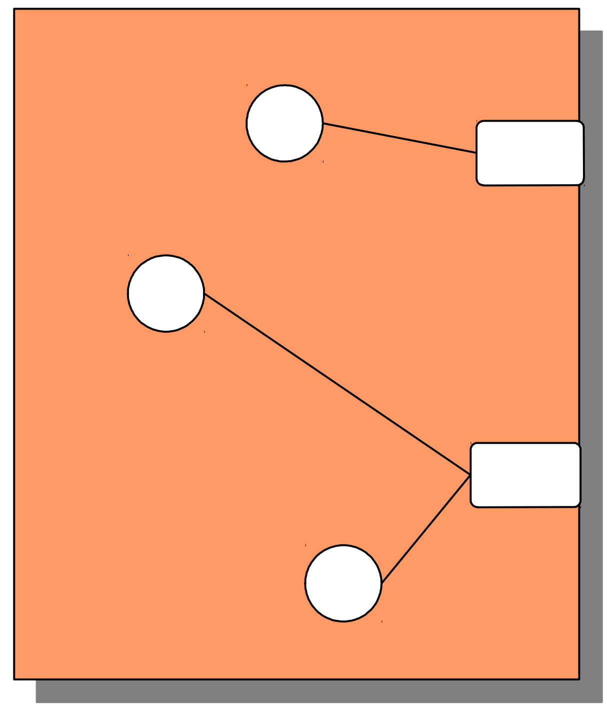
- Invasive composition can model mixin inheritance uniformly for all languages
  - e.g., for XML
- inheritance :=
  - copy first super document
  - extend with second super document

Document Engineering: Invasive Document Composition for XML

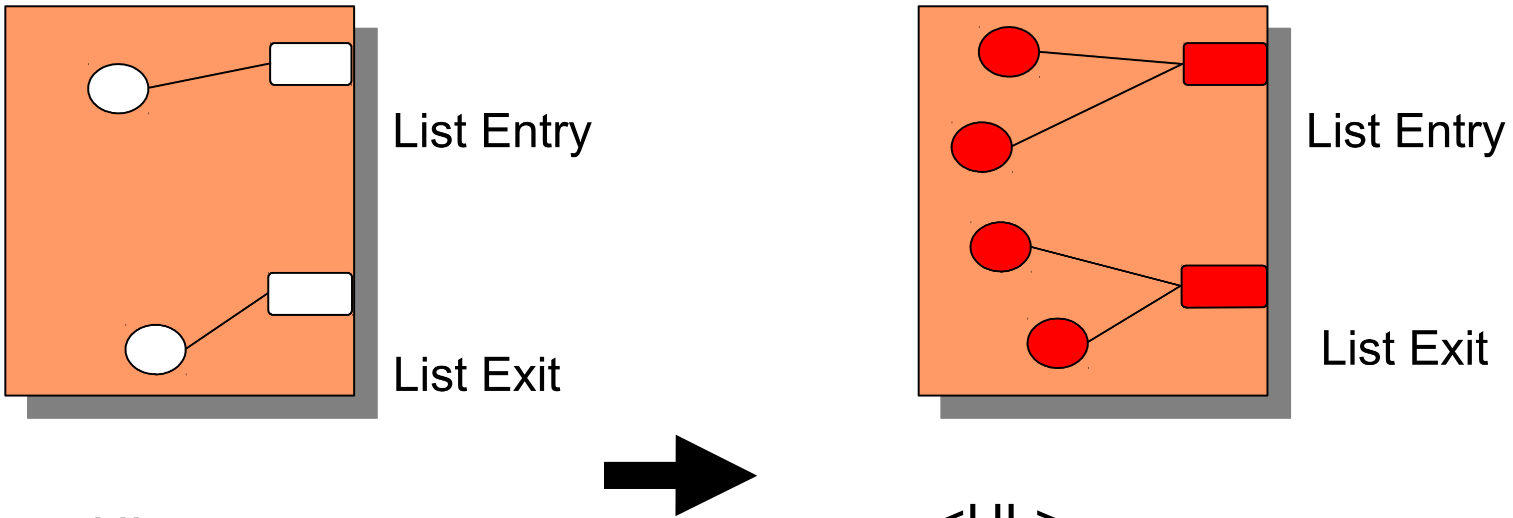
- Invasive composition can be used for document languages, too [Hartmann2011]

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





Hook Manipulation for XML



<UL>

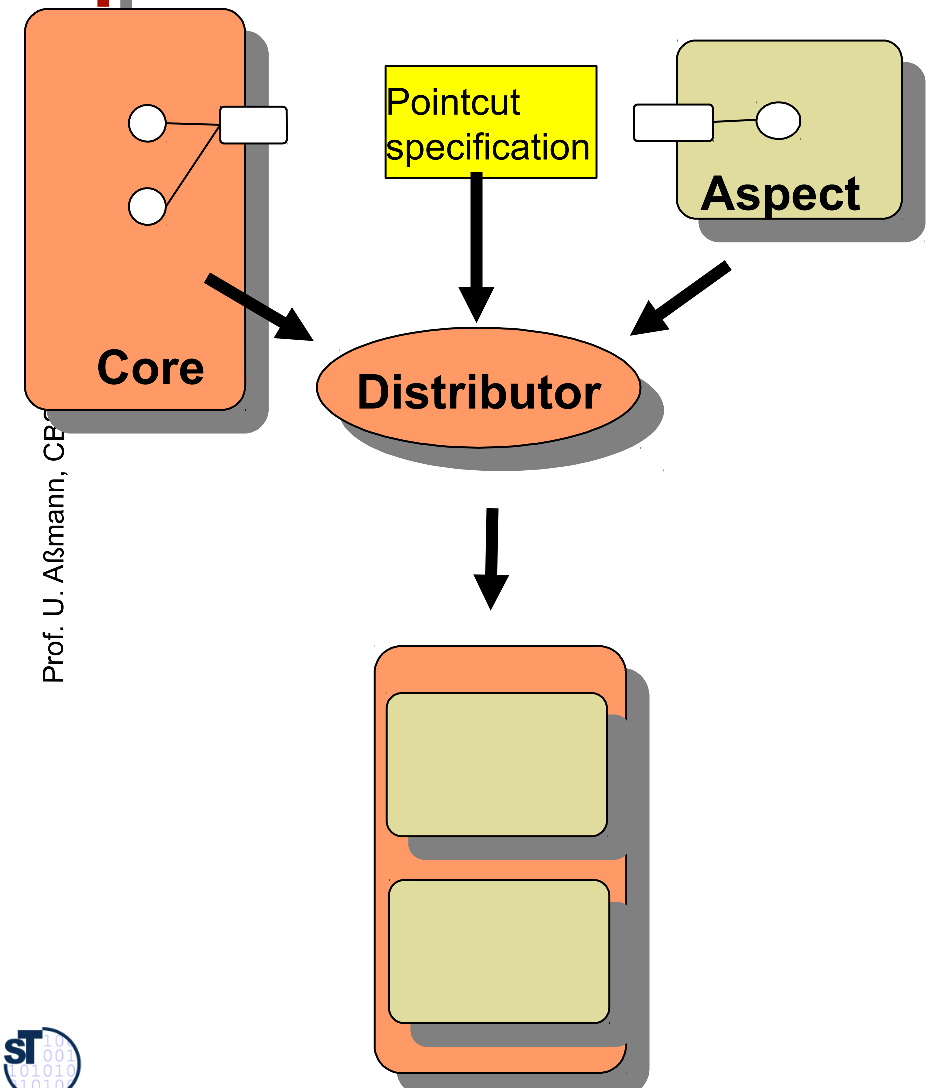
<LI>... </LI>
<LI>... </LI>

</UL>

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

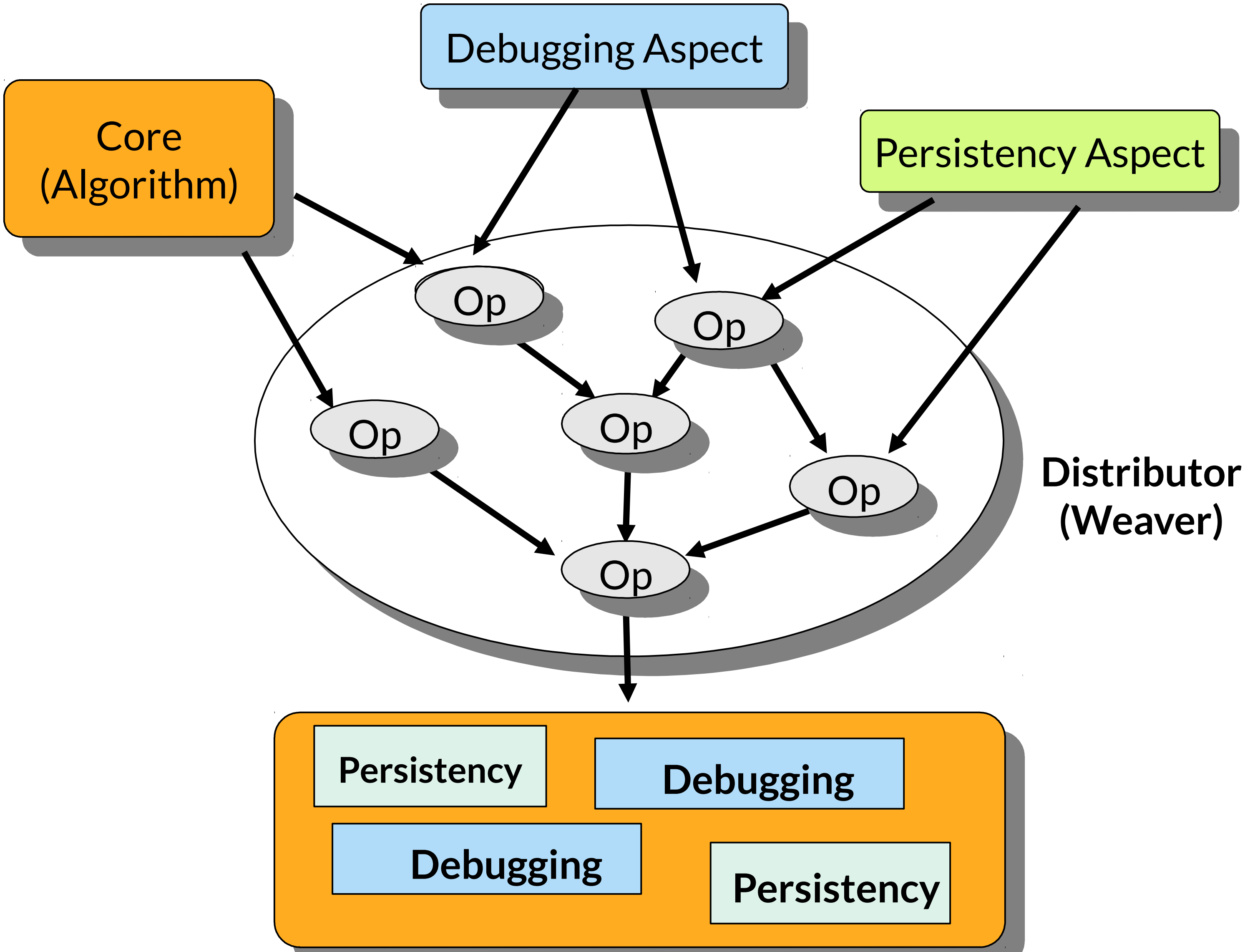
```
XMLcomponent.findHook(„ListEntry“).extend(„<LI>... </LI>”);
XMLcomponent.findHook(„ListExit“).extend(“<LI>... </LI>”);
```

Universal Weaving for AOP (Core and Aspect Components)



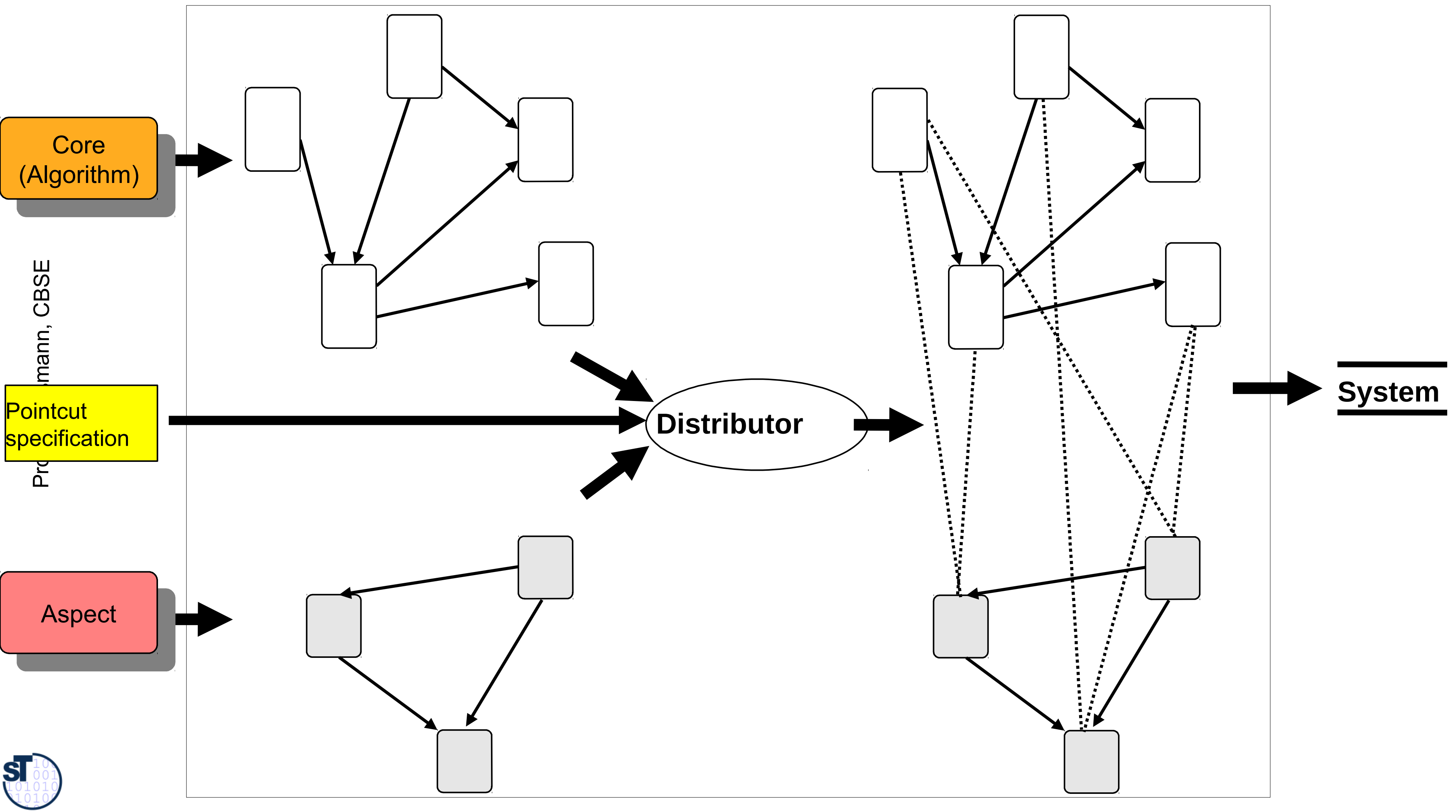
- Complex composers distribute aspect fragments over core fragments
- **Distributors (distribution operators)** extend the core
  - Distributors are more complex operators, defined from basic ones
  - Before, after, around are specific extension operators
- **Static aspect weaving** can be described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

Distributors are Composition Programs



Distributors Weave Relations between Core and Aspect

See optional Chapter “Specifying Crosscut Graphs with Graph Rewriting”



46.3 How to Make a Language Universally Composable

Universally Composable
Languages with for universal type-safe genericity and extension

Meta-Composition Systems
to Design Composition Systems

Universally Composable Languages

[Henriksson-Thesis]

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

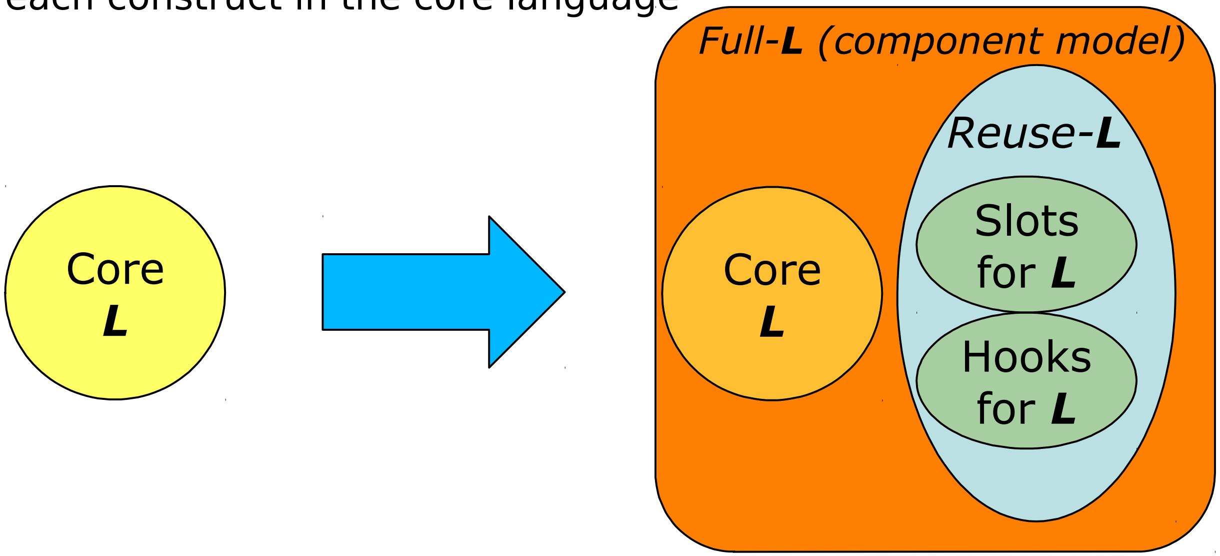
- The language has to be enriched with an invasive component model

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

- The Reuse language describes the composition interfaces of the components, an important part of the component model
- The component model can be composed by metamodel composition

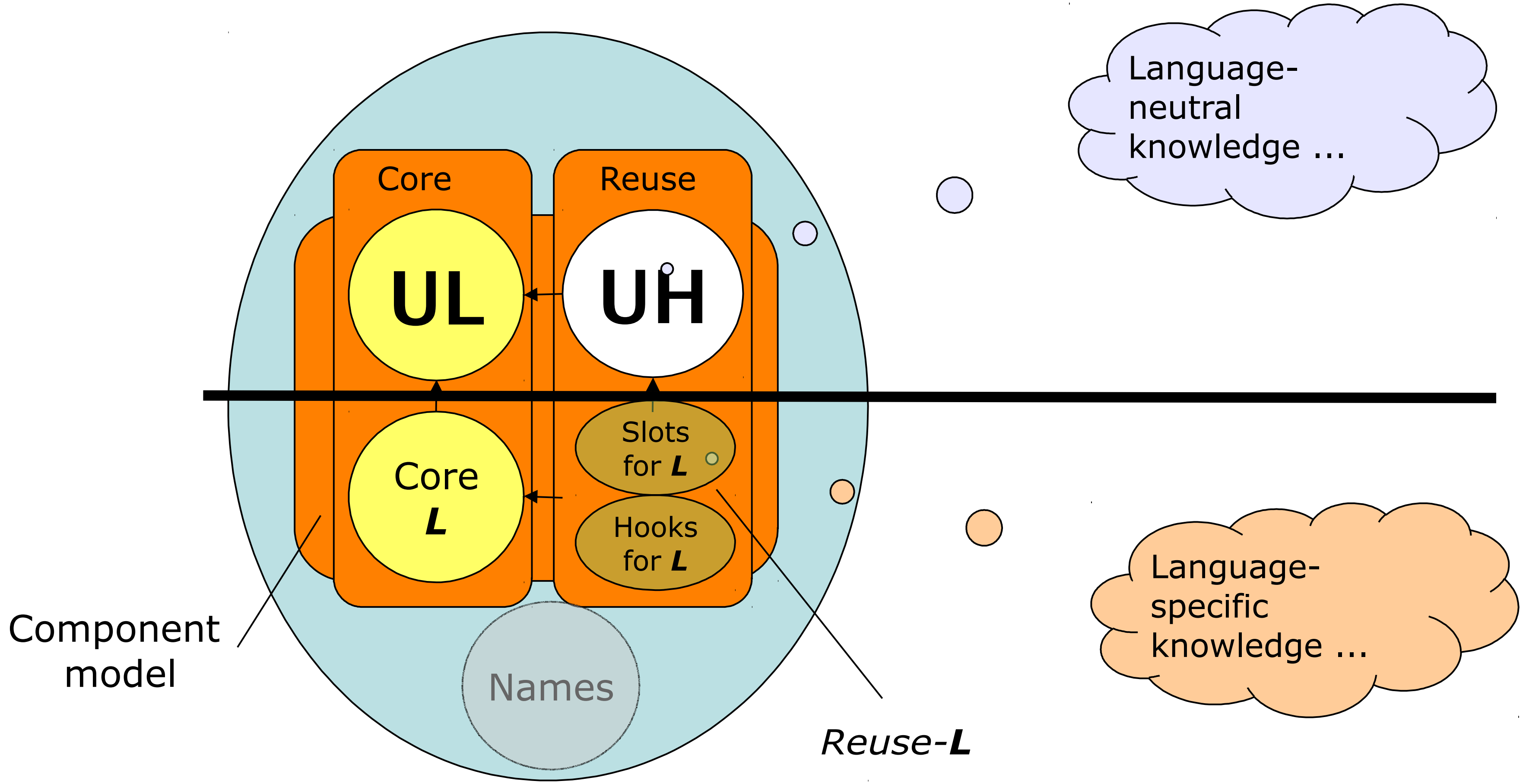
Slot and Hook metamodel: added to the core language metamodel

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



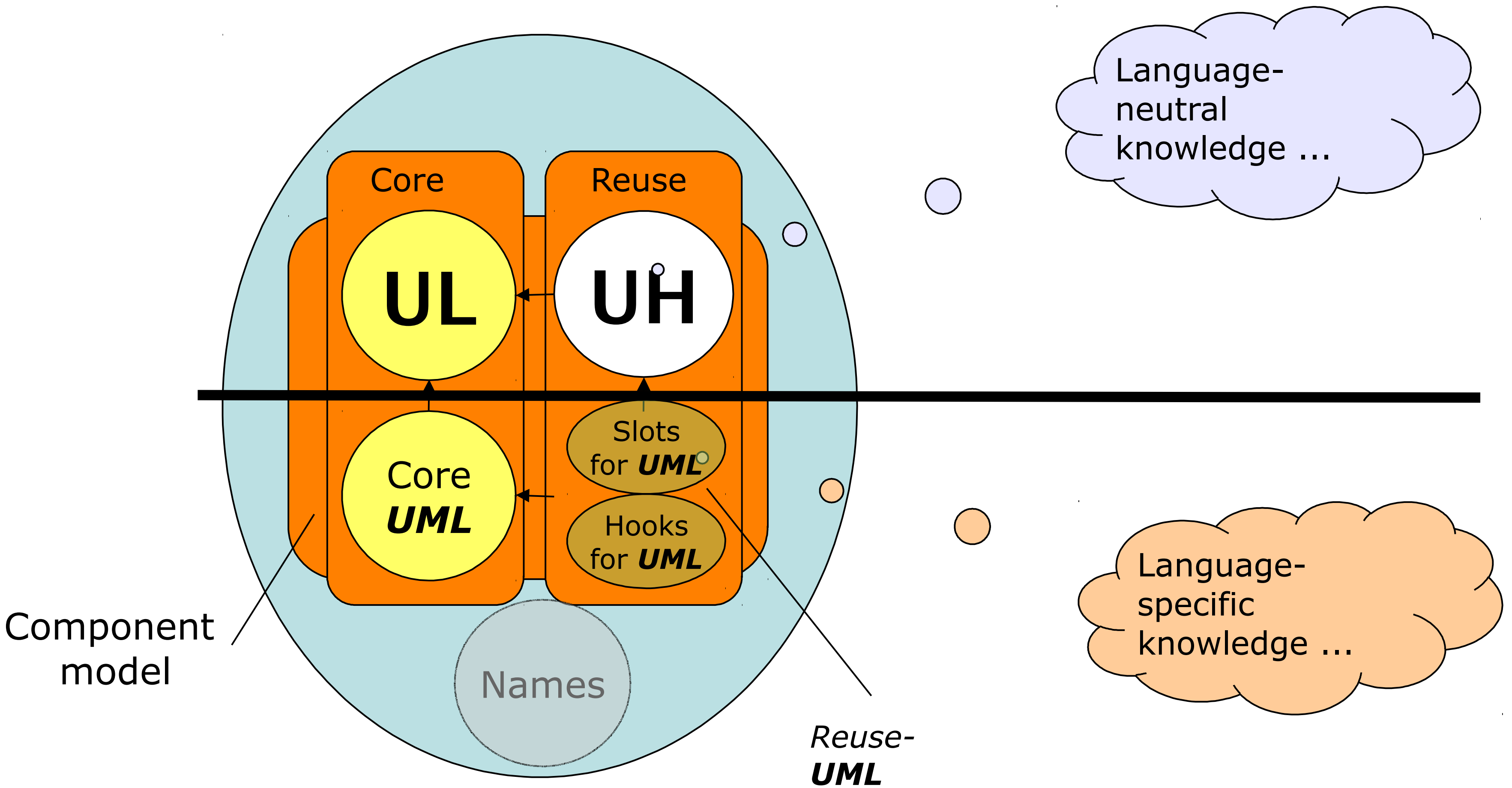
Structure of a Universally Composable Language

- The core and the reuse language have two levels



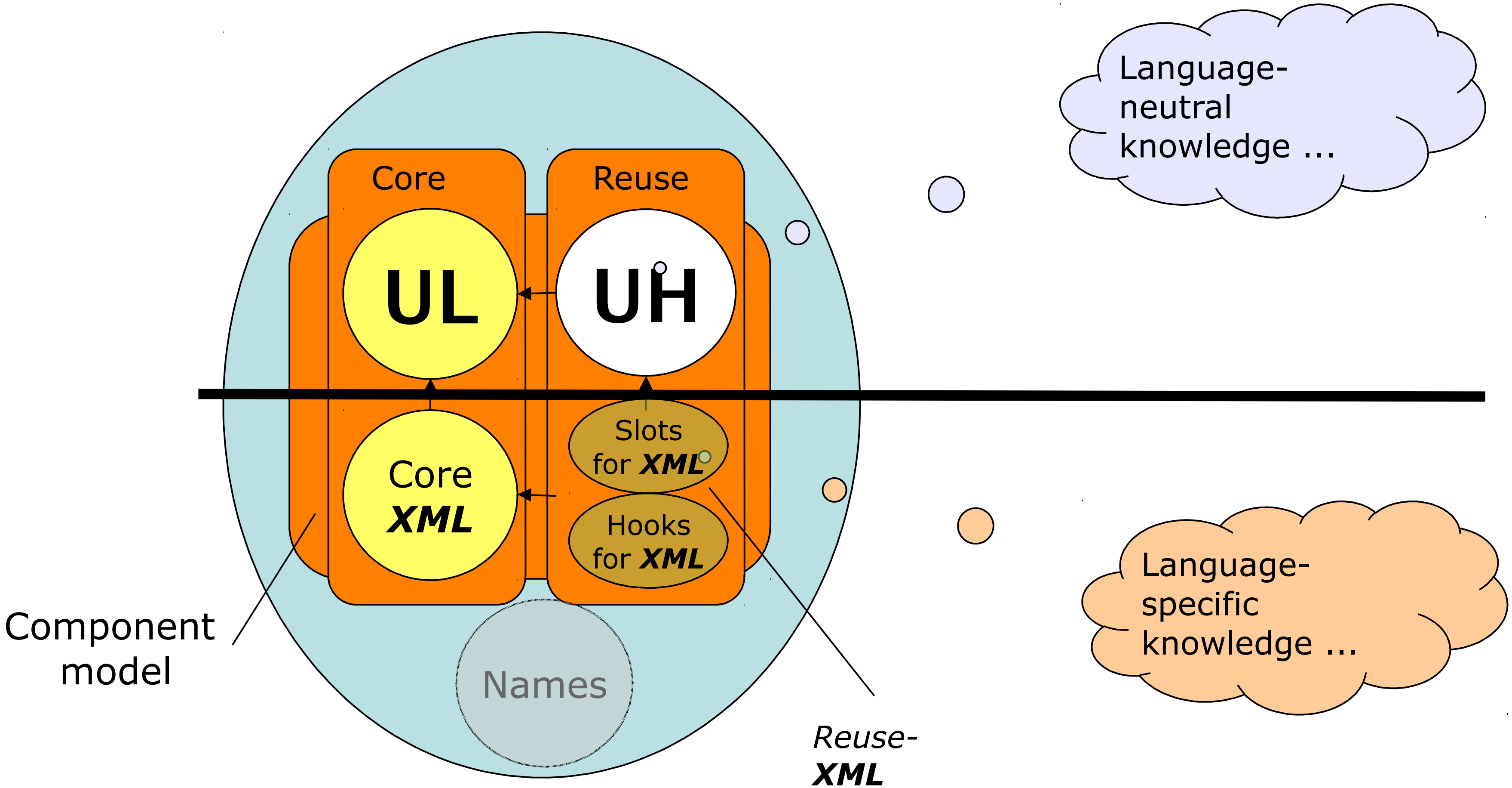
Reuse-UML, a Universally Composable Language

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



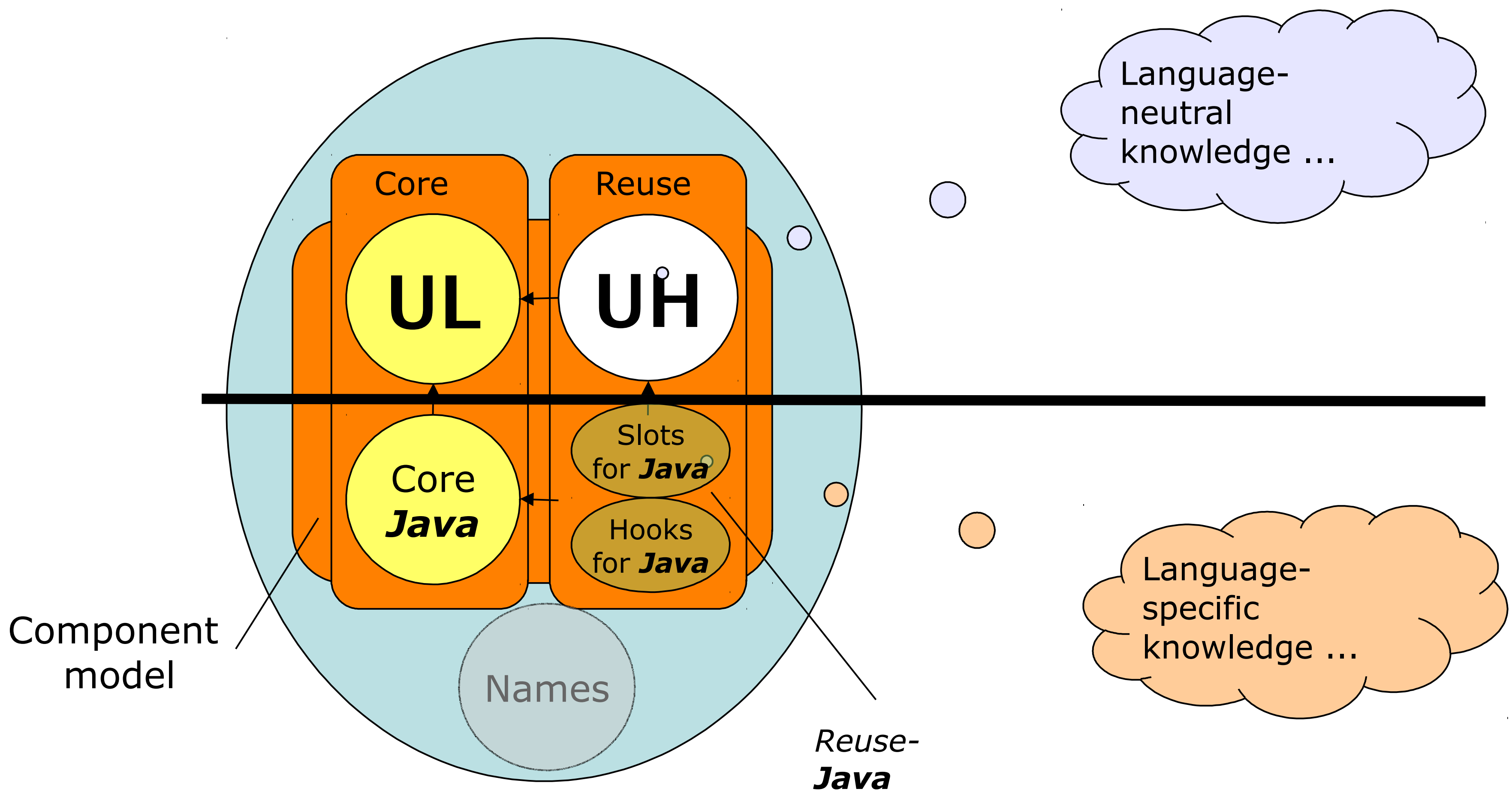
Reuse-XML, a Universally Composable Language

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



Reuse-Java, a Universally Composable Language

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



The Reusewair Technology

- [Henriksson-Thesis] Phd of Jakob Henriksson, 2008

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

- Reusewair was the world-wide first technology and tool to build reuse languages (component models) and composition systems for **any** text-based language
  - Grammar-based (EBNF)
  - Generic strategy for applying composition operators on components (based on Design Pattern Visitor)
  - Composition tools, type checker, come for free

The Reuseware Tool

- www.reuseware.org (Phd of Jendrik Johannes, 2010)
- http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-63986
- Reuseware is a tool to build reuse languages (component models) and composition systems for text-based and diagramm-based languages
    - Eclipse-based
    - metamodel-controlled (metalanguage M3: Eclipse e-core)
    - Plugins are generated for composition
    - Composition tools come for free
    - Textual, graphic, XML languages
- Framework instantiation is supported for variation and extension
- Jobs open!

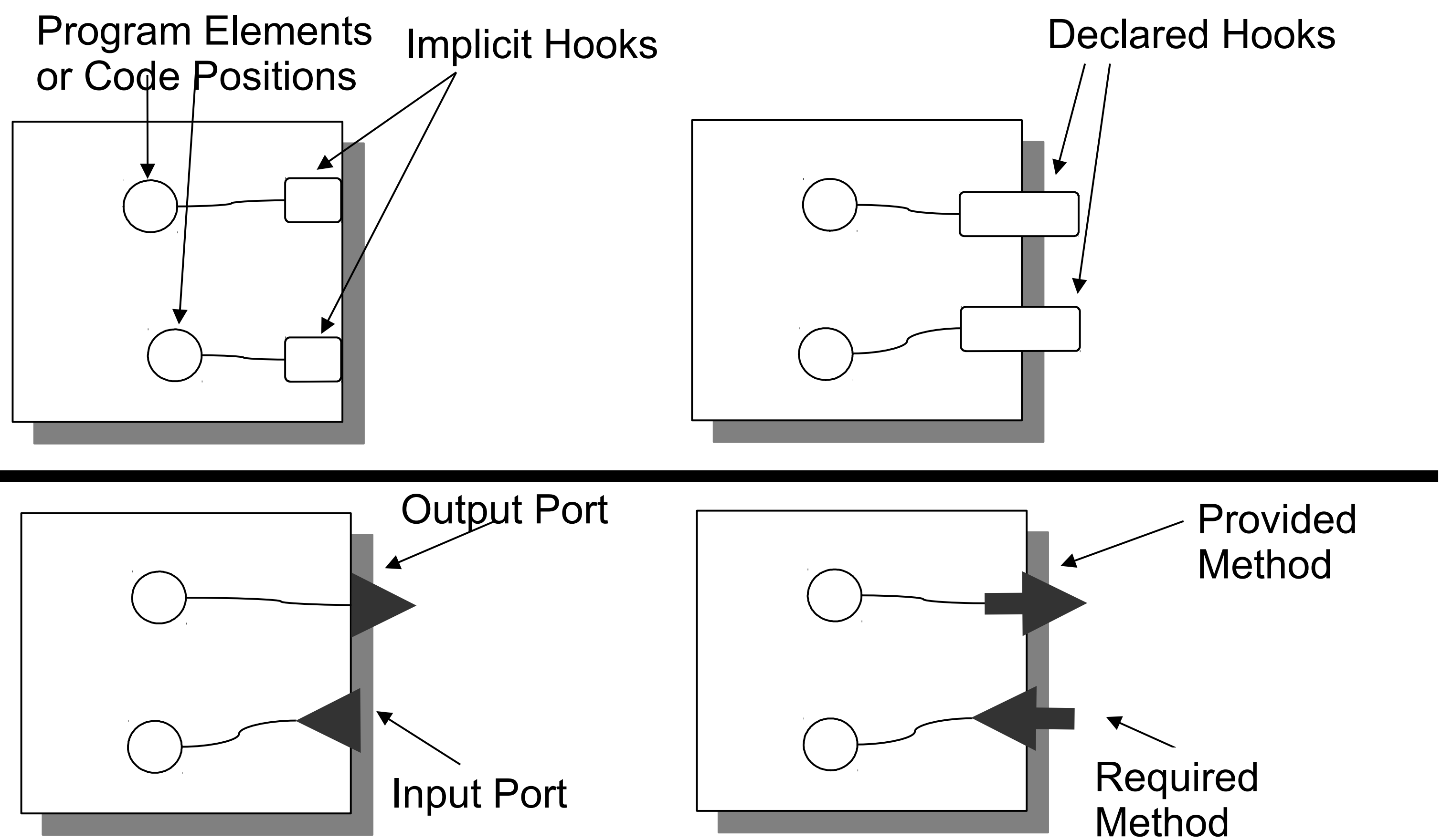
The SkAT Tool

- Phd of Sven Karol, 2014
- Open source project
  - https://bitbucket.org/svenkarol/skat/wiki/Home
- SkAT is a tool to build reuse languages (component models) and composition systems for text-based and diagram-based languages
  - Based on Reference-Attribute-Grammar (RAG)
  - And metamodels (metalanguage M3: Eclipse e-core)
  - Declarative composition constraints control the composition
  - Composition tools come for free
  - Textual, graphic, XML languages
- Framework instantiation is supported for variation and extension
- Jobs open!

46.4. Staging of Composition: Composition and Functional Interfaces

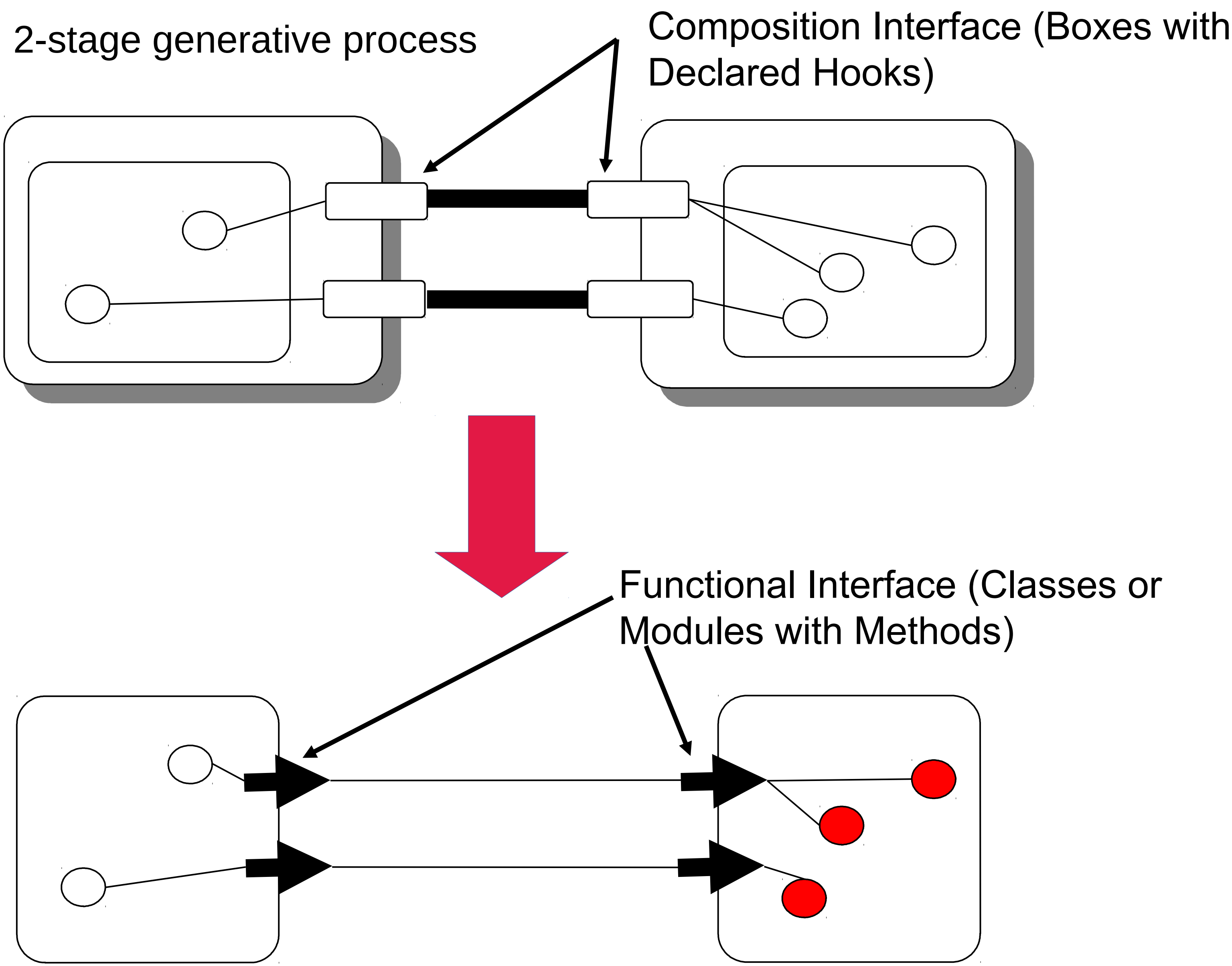
Composition vs Functional Interfaces

- 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



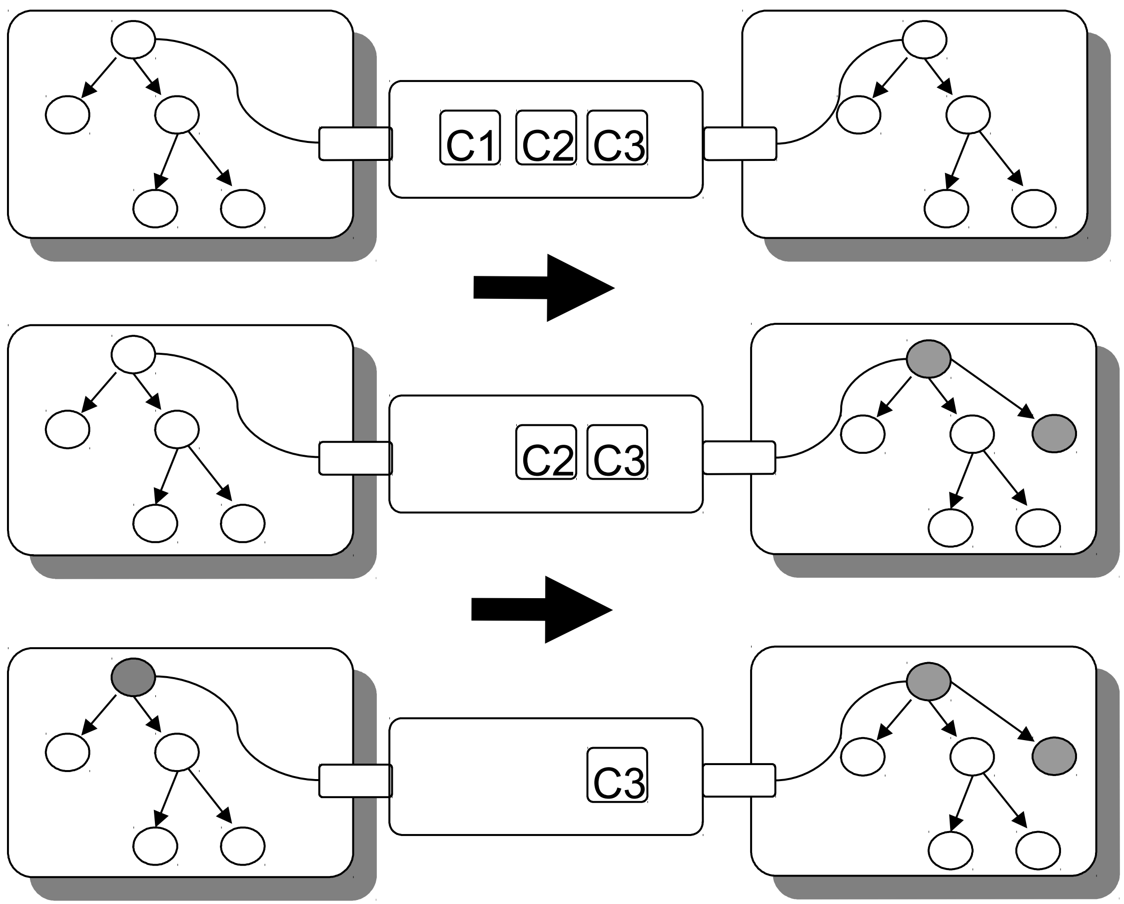
Functional Interfaces are Generated from Composition Interfaces

- 2-stage generative process



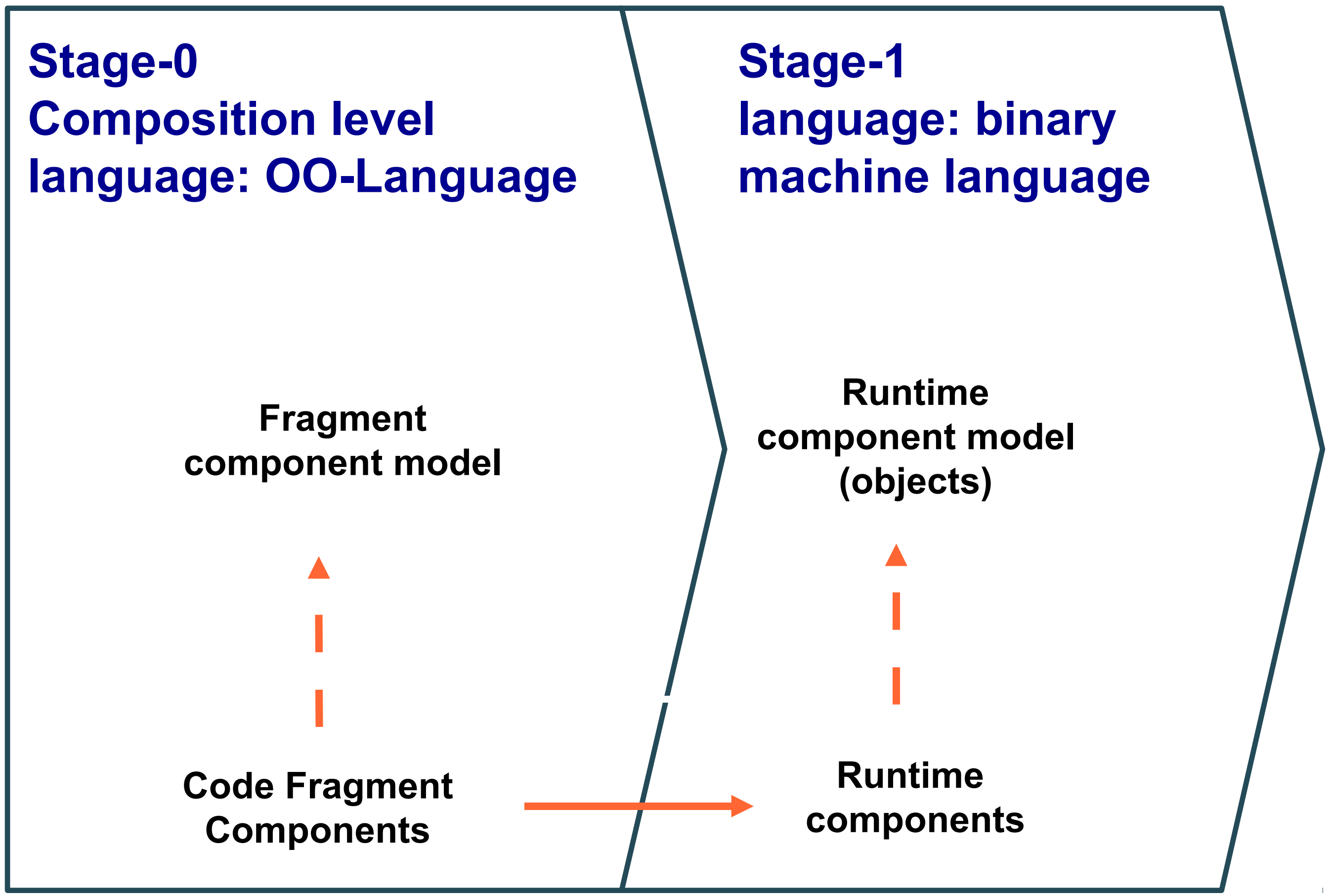
Execution of a Composition Program

- 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



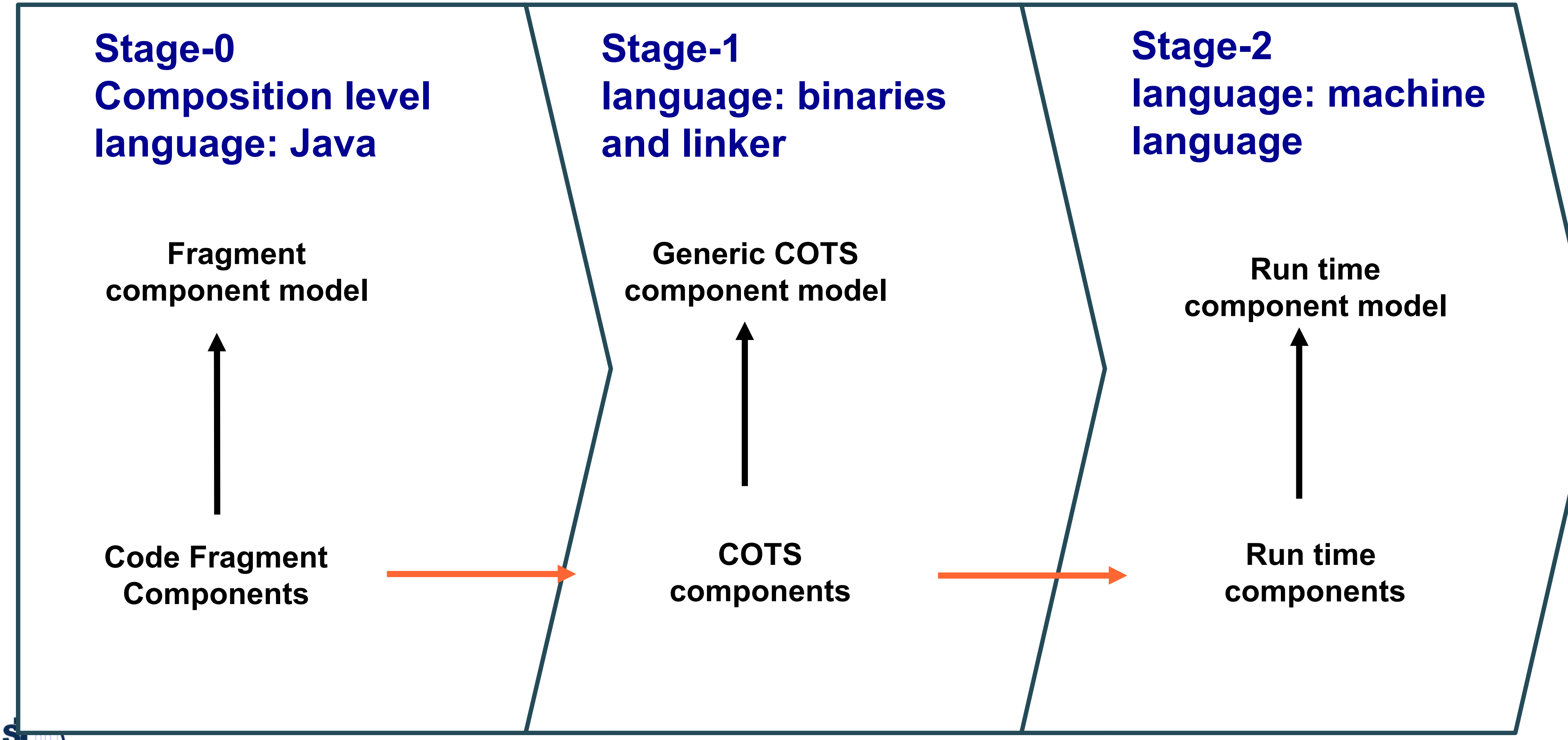
The Stages of Normal O-O Languages

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



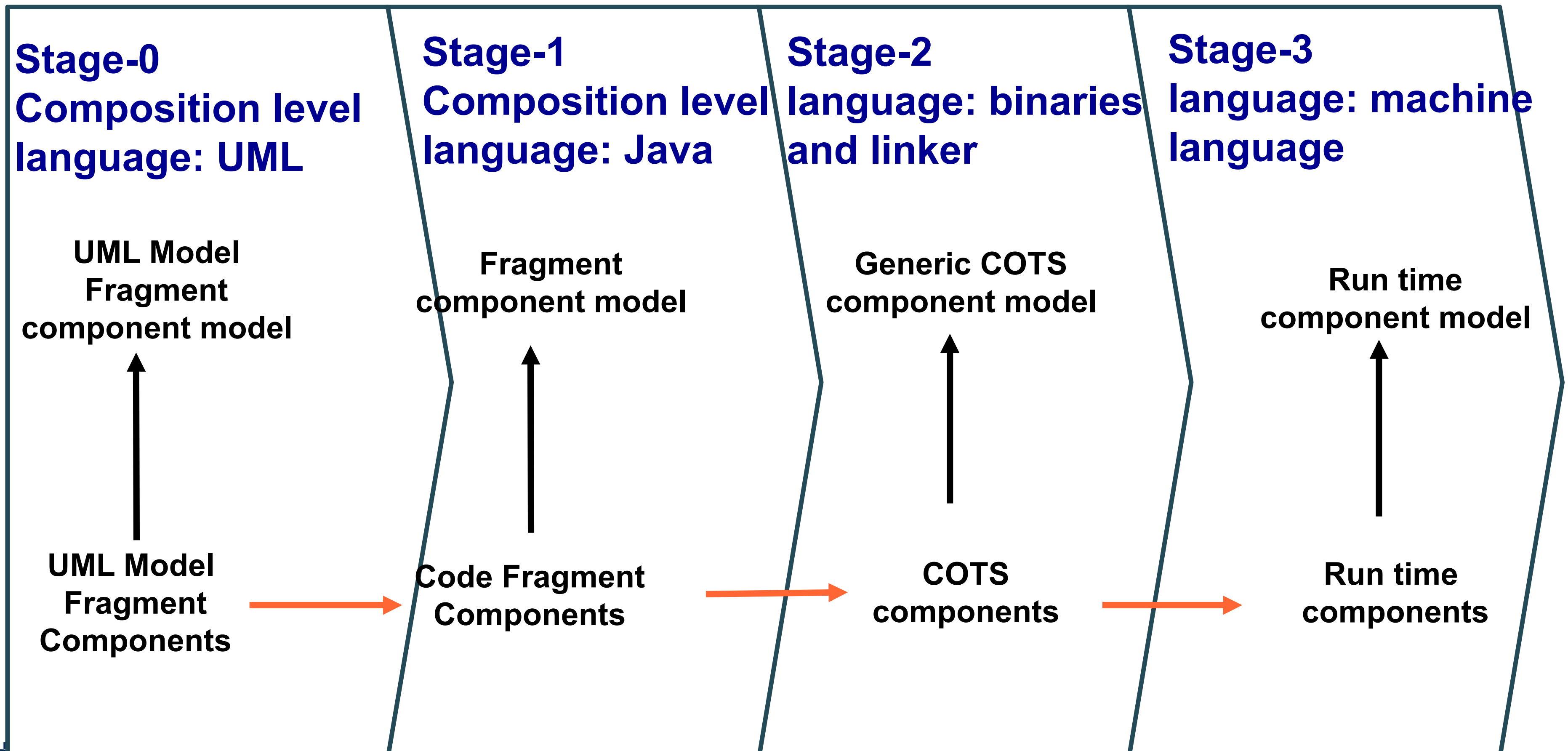
Component Models on Different Levels in the Software Process

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 a universally composable language:

  component models

  composition operators

  composition language

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

46.5. Different Forms of Greyboxes (Shades of Grey)

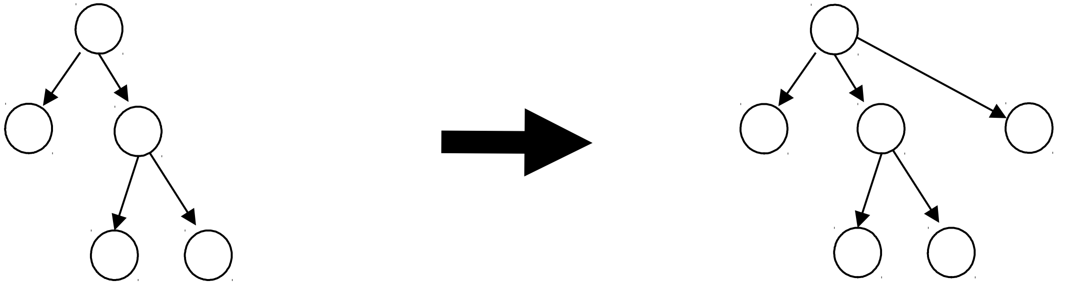
Invasive Composition and Information Hiding

- 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

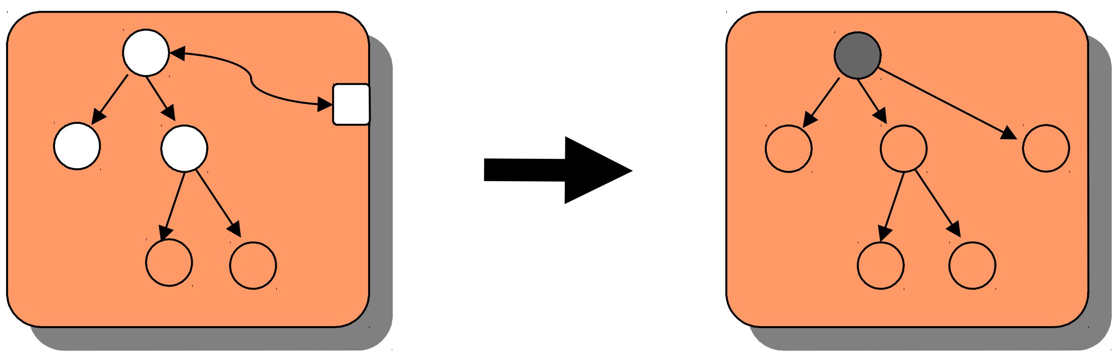
Refactorings
Transformations
Metaprograms



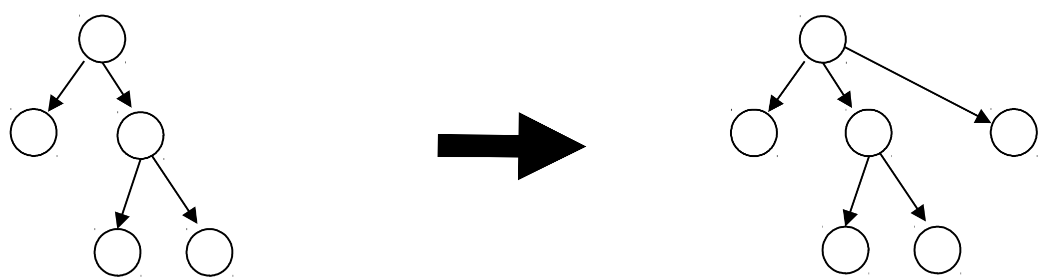
Modifying Implicit Hooks is a Light-Grey Operation

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

Composition with implicit hooks



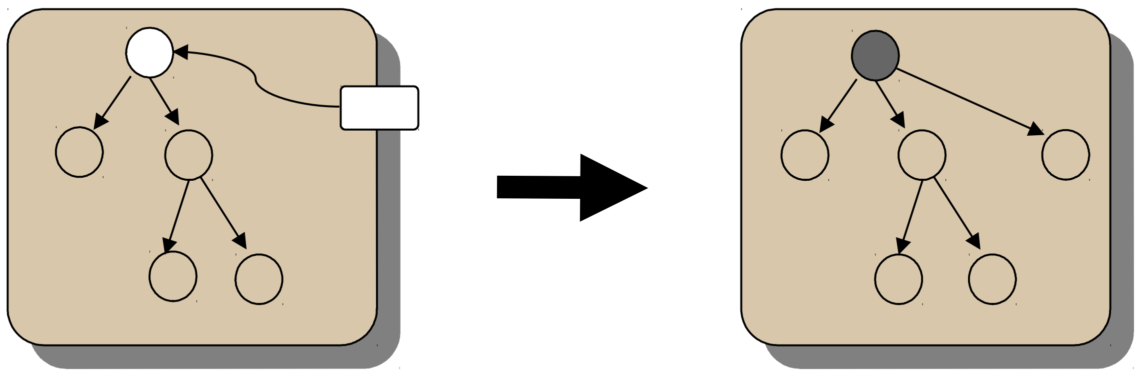
Refactorings
Transformations
Metaprograms



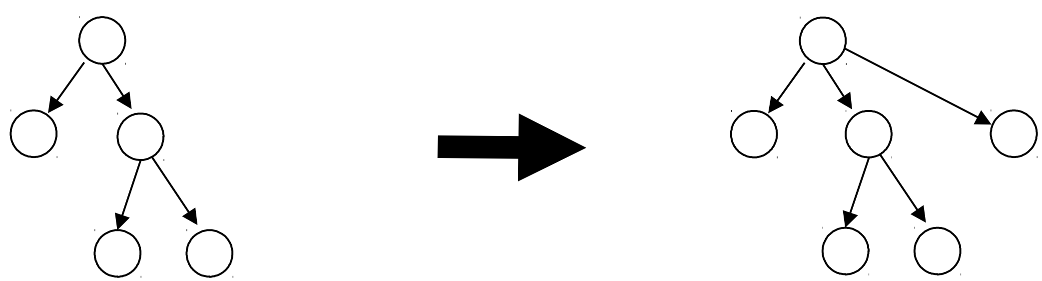
Parameterization as Darker-Grey Operation

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

Composition with declared hooks



Refactorings
Transformations
Metaprograms

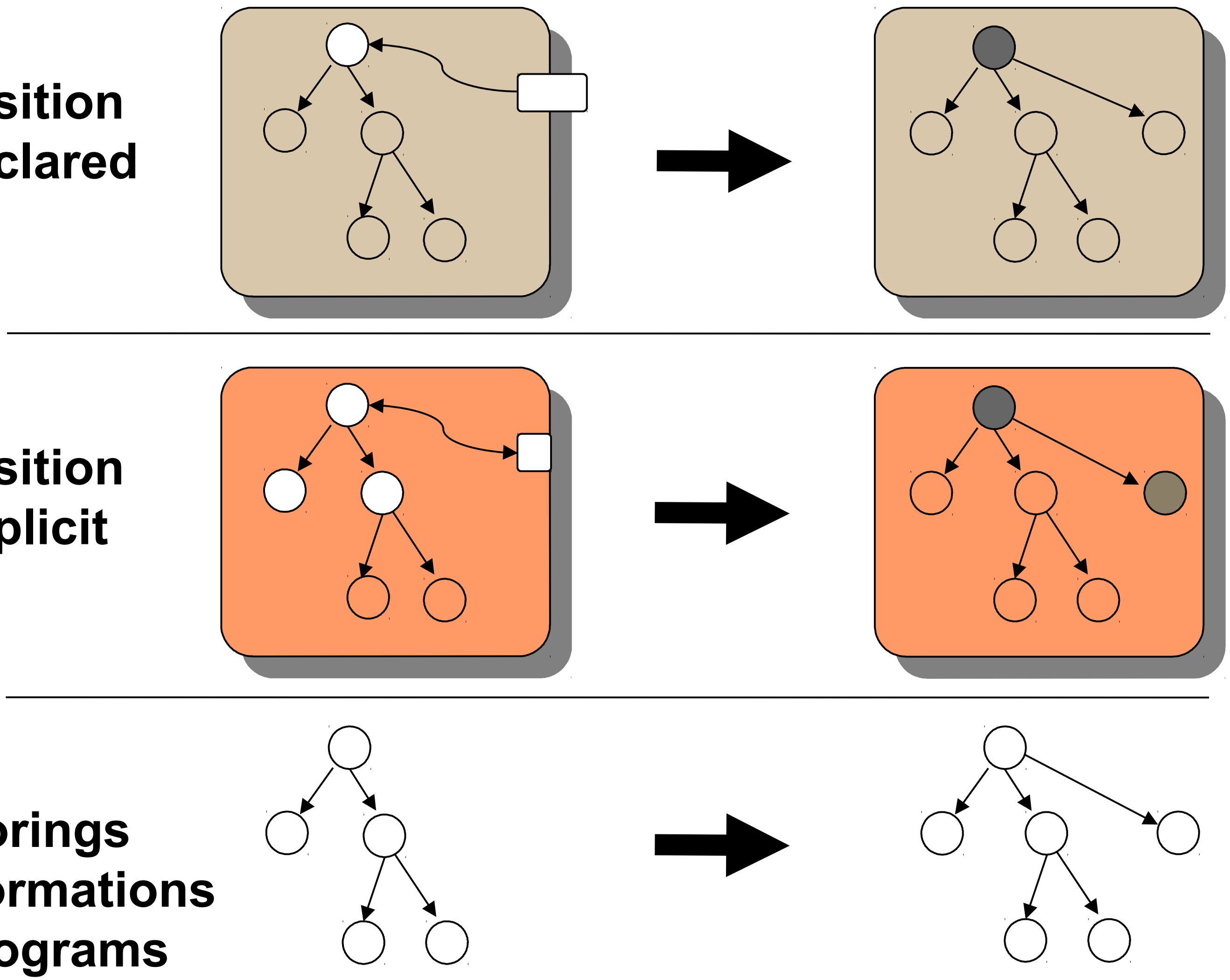


Systematization Towards Greybox Component Models

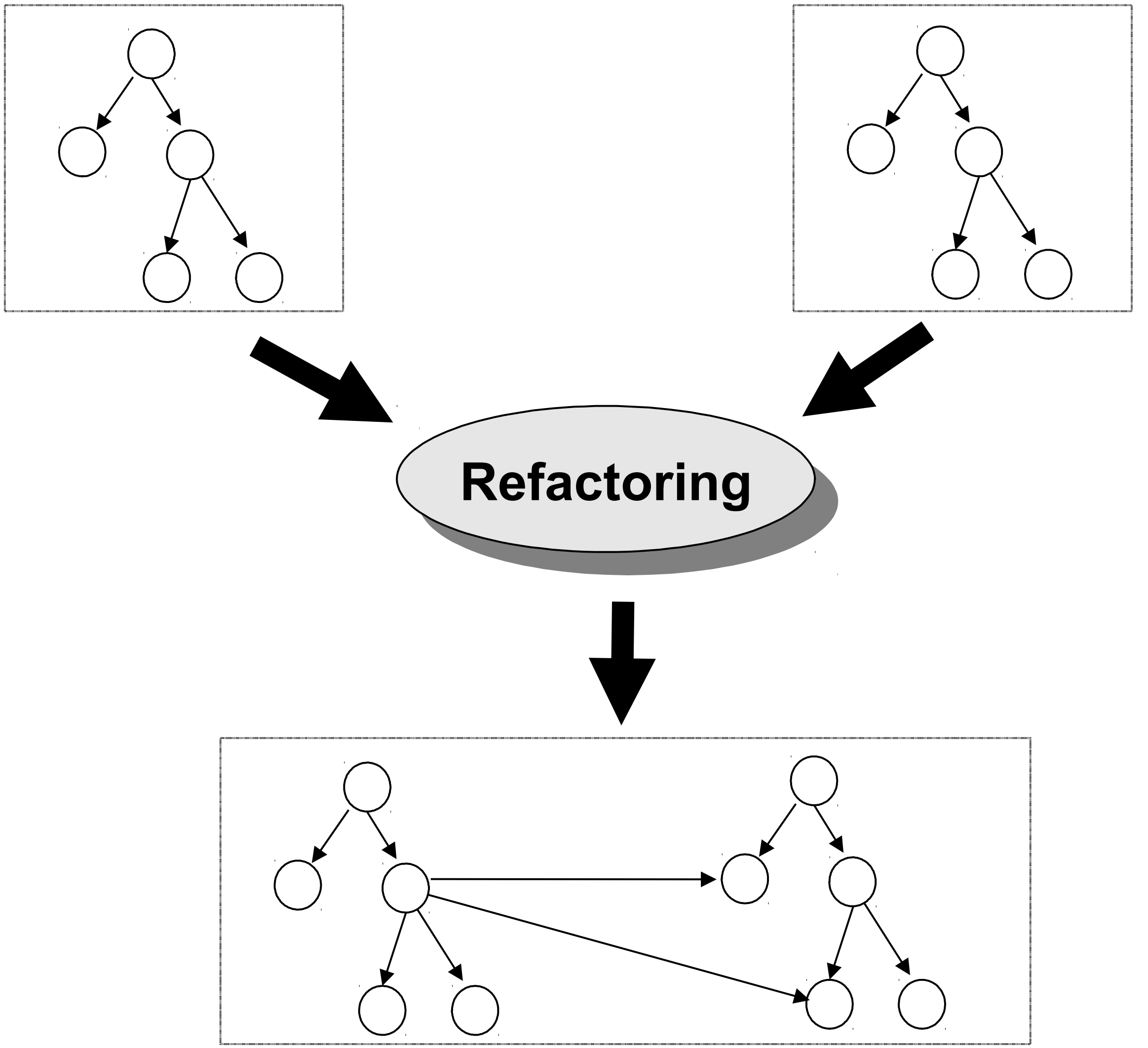
Composition with declared hooks

Composition with implicit hooks

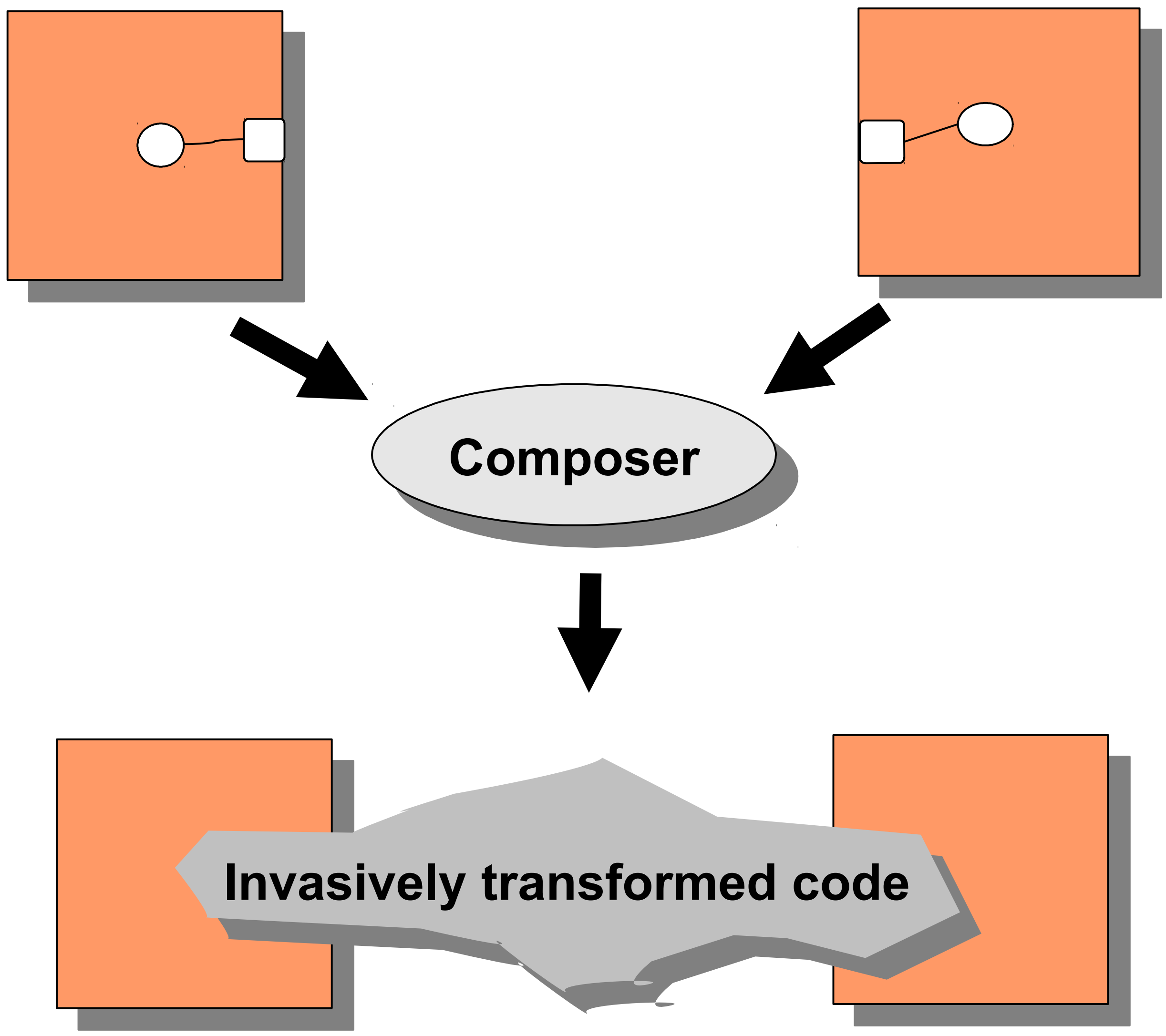
Refactorings
Transformations
Metaprograms



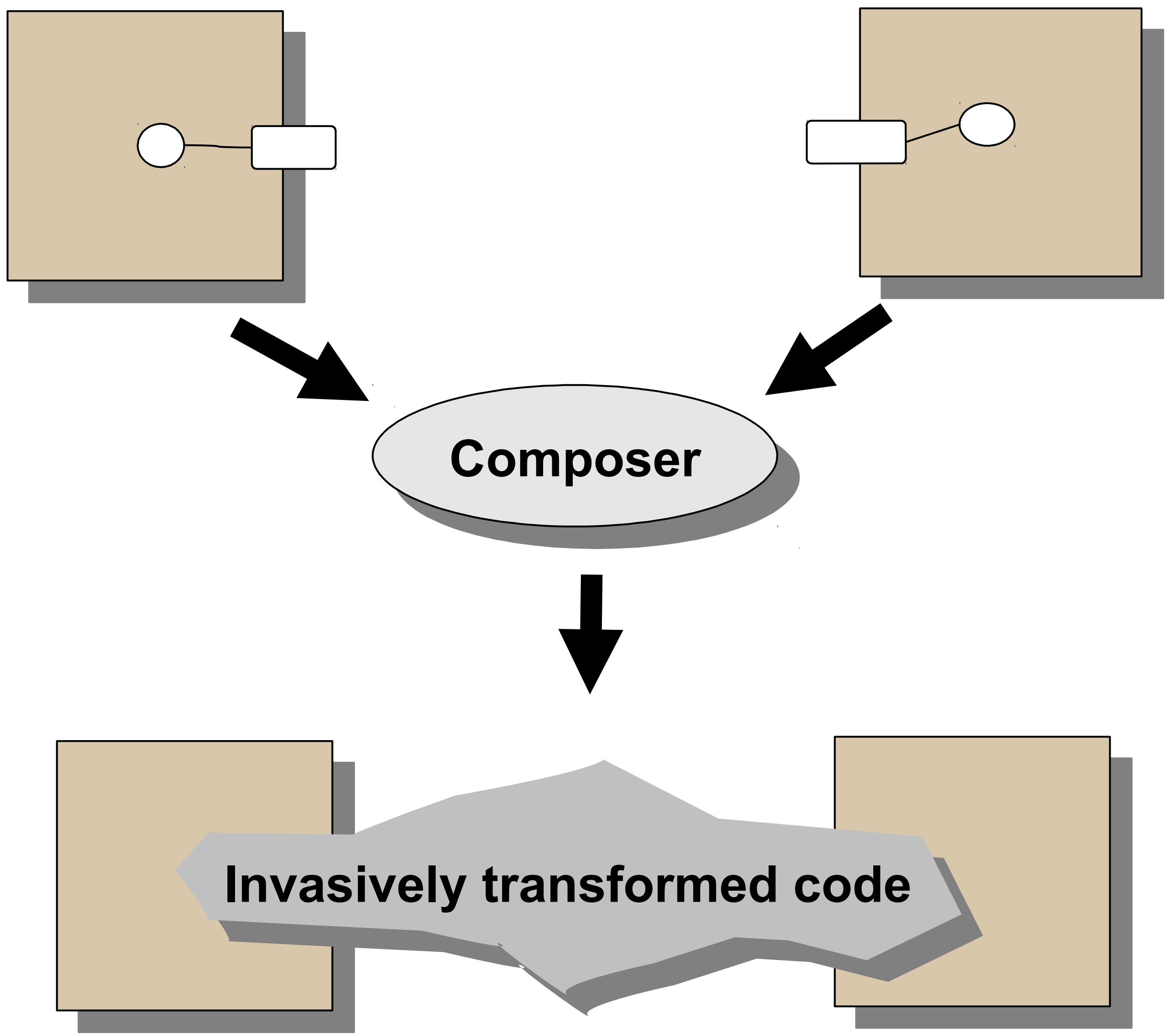
Refactoring Builds On Transformation Of Abstract Syntax



Invasive Composition Builds On Transformation Of Implicit Hooks



Invasive Composition Builds On Transformation on Declared Hooks



46.6 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

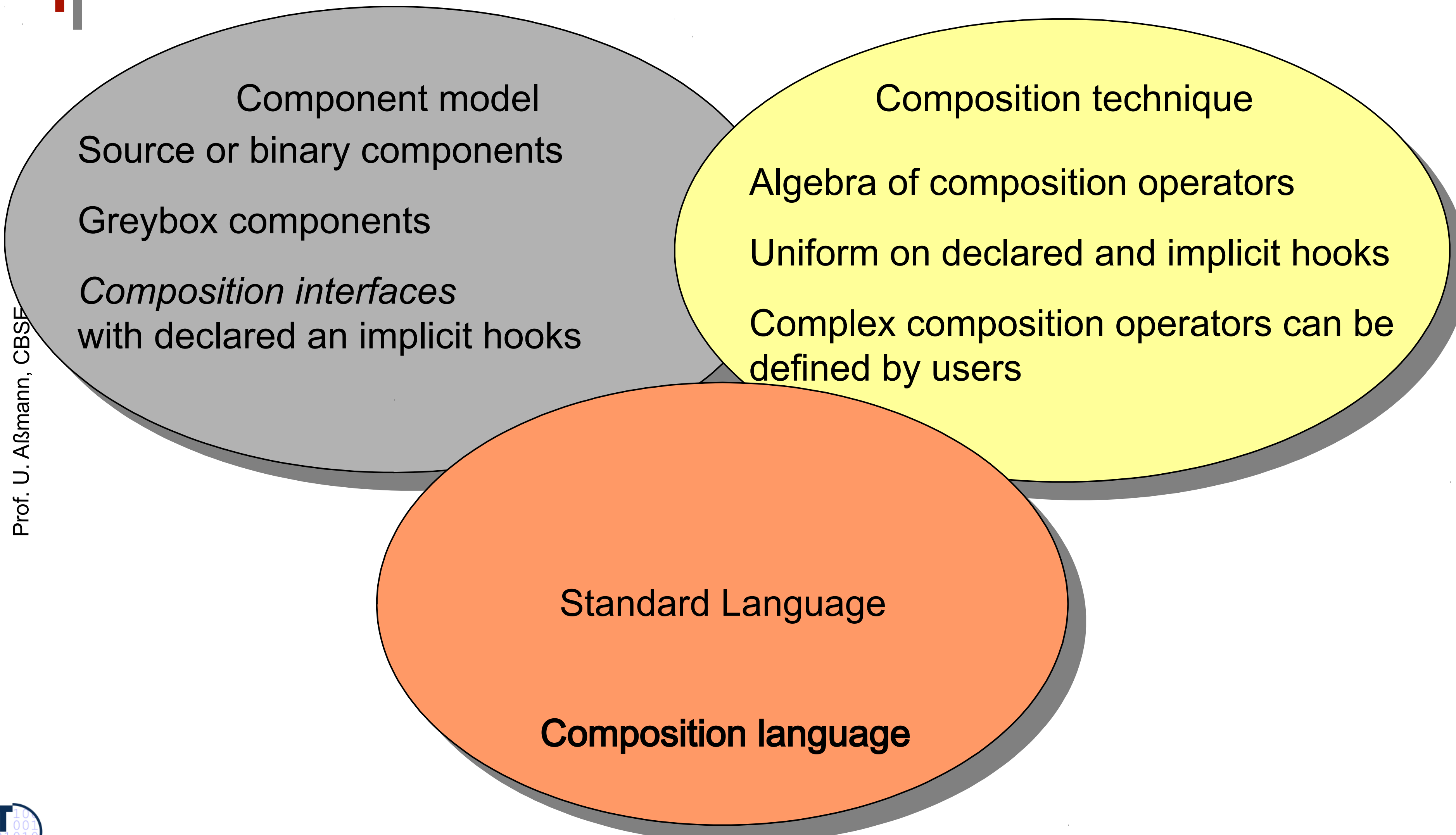
Composition Language

- 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)
- The Reusewair, Reuseware and SkAT Composition Frameworks extends these ideas
  - For arbitrary grammar-based languages
  - For metamodel-based languages
- http://reuseware.org
- https://bitbucket.org/svenkarol/skat/wiki/Home

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

- Why is it good to explicitly specify composition with a composition program ?
- Explain how to write an aspect weaver with an imperative composition language
- Explain the difference of hooks, slots and query points
- Explain invasive connection
- Why can invasive software composition explain so many different programming styles?
- How would you build a composition system for UML activity diagrams?
- Can you imagine the ingredients of a XML composition system?