

20. Integrational Ways to Decompose and Compose

Prof. Dr. Uwe Aßmann
Florian Heidenreich

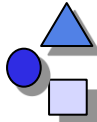
Technische Universität Dresden
Institut für Software- und
Multimediatechnik

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

Version 12-0.1, Mai 15, 2012

1. Decomposition and Composition
 1. Example role modeling
2. Systems with Dimensional Decomposition and Composition
 1. LambdaN calculus
 2. Piccola

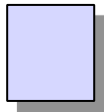
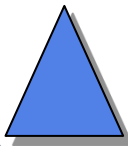
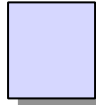


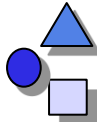


Obligatory Literature

- ▶ [Dami95] Laurent Dami. [Functions, Records and Compatibility in the Lambda N Calculus](http://scg.unibe.ch/archive/oosc/PDF/Dami95aLambdaN.pdf) in Chapter 6 of “Object-oriented Software Composition”.
<http://scg.unibe.ch/archive/oosc/PDF/Dami95aLambdaN.pdf>
- ▶ Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, Object-Based Models and Languages for Concurrent Systems, LNCS 924, pages 147-161. Springer, 1995.
- ▶ Optional:
 - ▶ Dami, Laurent. Software Composition. PhD University Geneva 1997. The centennial work on the Lambda-N calculus
 - ▶ F. Achermann. Forms, Agents, and Channels. Defining Composition Abstraction with Style. PhD thesis. University Berne 2002. Available from Oscar Nierstrasz' Software Composition Group's pages scg.unibe.ch.
 - This web site is great, one of the best sites for composition. Many papers of Nierstrasz and his PhD students show all aspects of composition. Visit it!

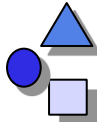
20.1 Decomposition and Composition



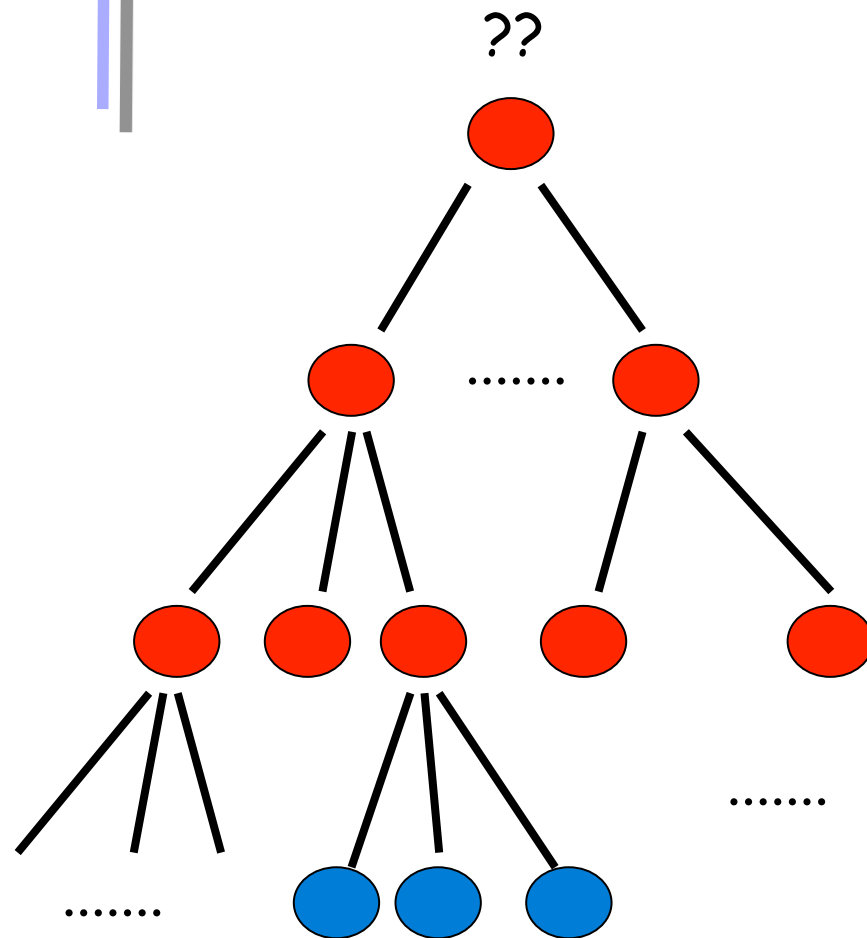


Problem Solving with Divide and Conquer Strategy

- ▶ Divide et impera (from Alexander the Great)
 - divide: problems into subproblems
 - conquer: solve subproblems (hopefully easier)
 - compose (merge): compose the complete solution from the subsolutions
- ▶ However, strategy of decomposition is different
- ▶ Methods of (De)composition. We decompose
 - To simplify the problem
 - To find solutions in terms of the abstract machine we can employ
 - When this mapping is complete, we can compose



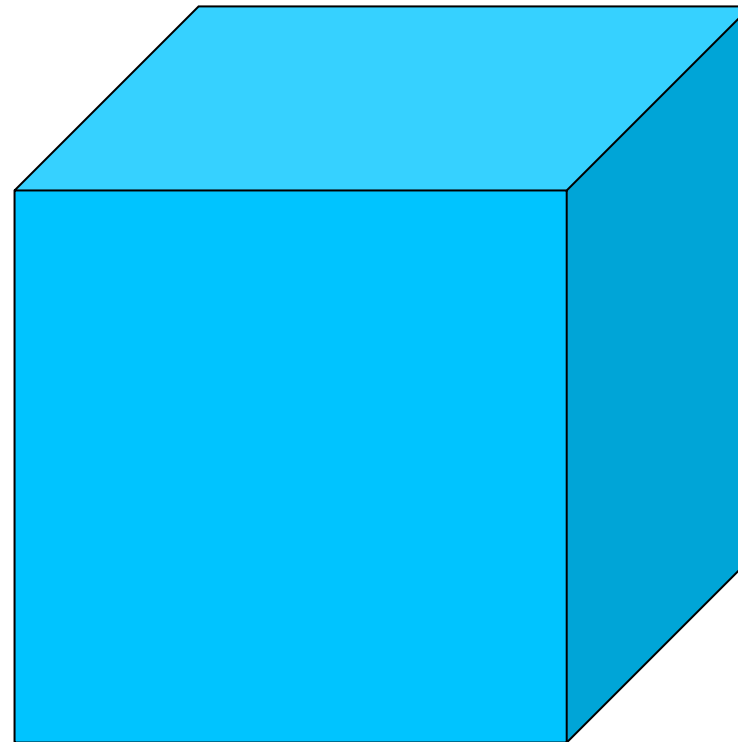
A Decomposition Tree

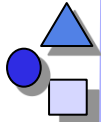


- ▶ Reuse of partial solutions is possible (then the tree is a dag)
- ▶ Leafs are operations of a given abstract machine (may be the software or the chip)



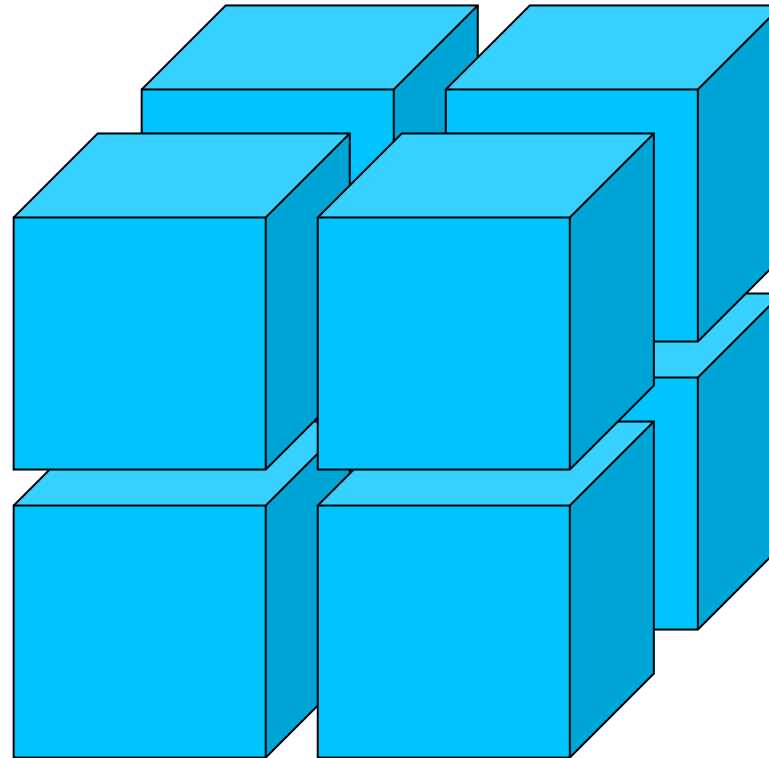
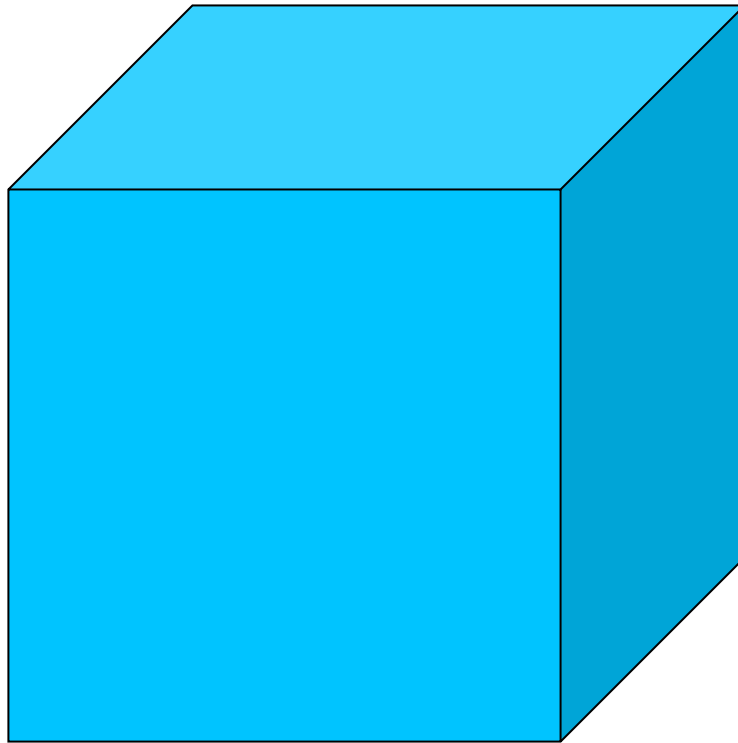
How to Decompose a Cube?





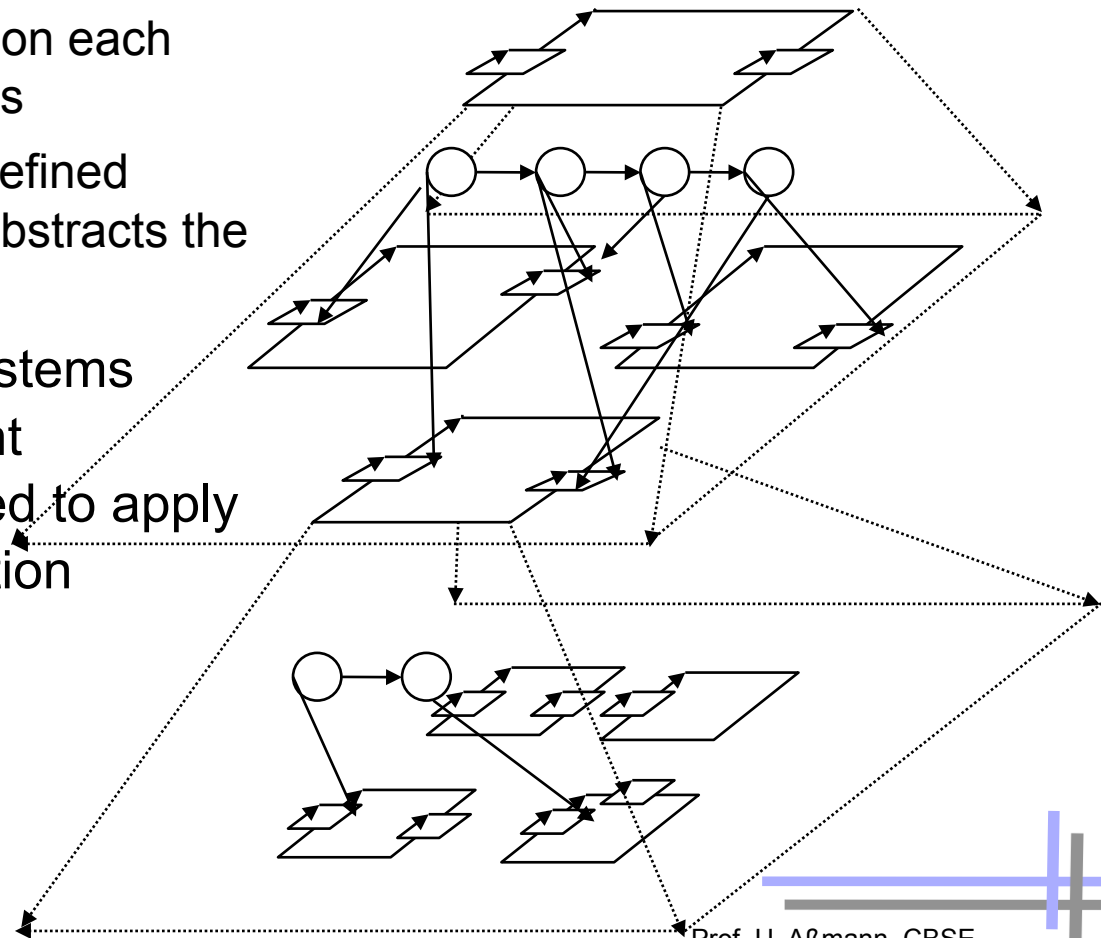
Blockwise Decomposition

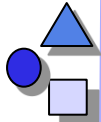
- ▶ Blockwise decomposition is stepwise refinement
 - Problem size is reduced, dimensionality stays the same



Refinement leads to Reducible Hierarchies and Graphs

- ▶ Trees or dags result
 - can be layered
- ▶ Reducible graphs result
 - Can be layered too, on each layer there are cycles
 - Every node can be refined independently and abstracts the lower levels
- ▶ Component-based systems contain the component hierarchy, so they need to apply blockwise decomposition

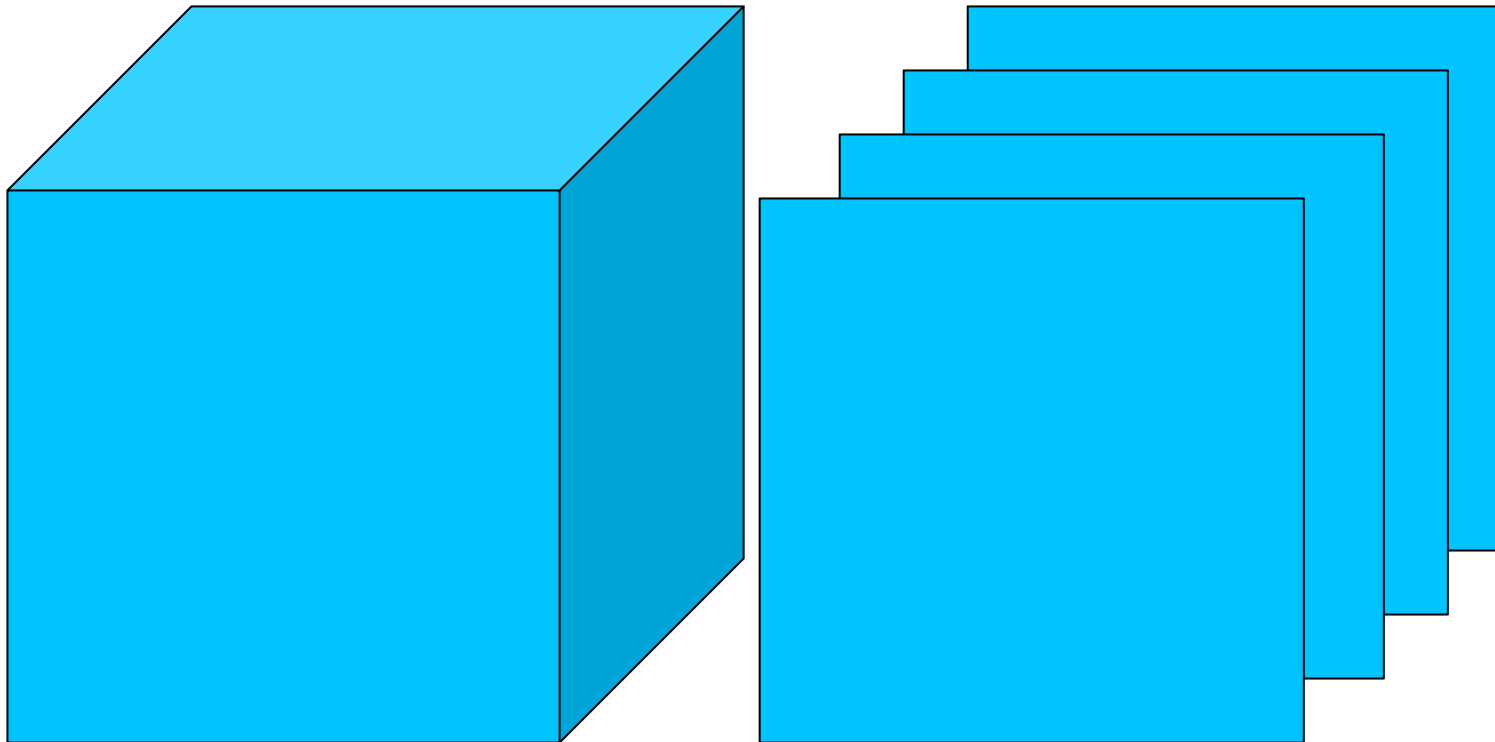


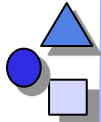


Hyperspace Decomposition (Dimensional Decomposition)

Decomposition is not point-wise

Problem size is retained; number of dimensions is reduced



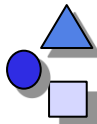


Basic Decomposition Strategy II: Separation of Concerns (SoC)

- ▶ Separation of Concerns (aka Dimensional Divide and Conquer, dimensional (de-)composition)
 - Splitting of hyperplanes (dimensions) of the problem
 - Problem dimension count is reduced
 - Problem size is not reduced
- ▶ If separation of concerns takes place in a component model, we speak of *grey-box composition* or *integrational composition*
- ▶ A *viewpoint* defines a set of related concerns, producing a partial representation of a system (*view*)

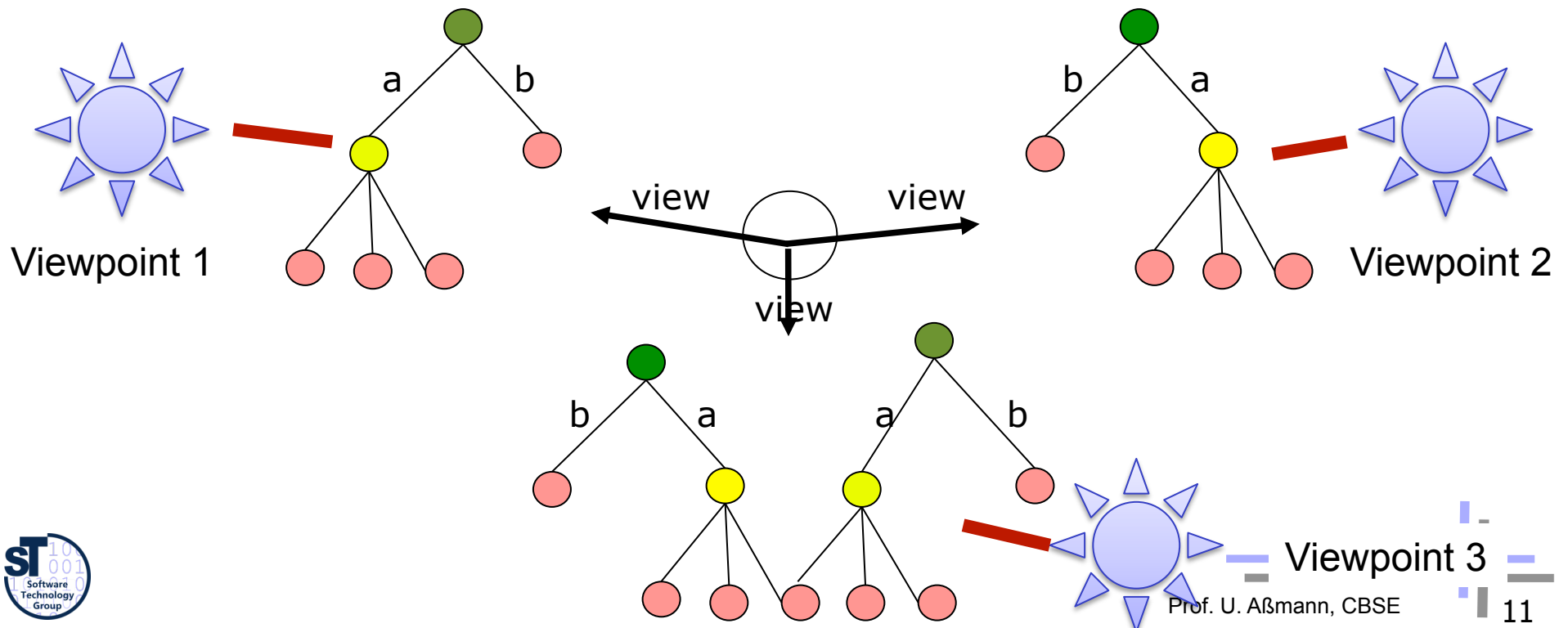
A *view* is a representation of a whole system from the perspective of a related set of concerns

[ISO/IEC 42010:2007, Systems and Software Engineering -- Recommended practice for architectural description of software-intensive systems]



Separation of Concerns leads to Dimensions

Dimensional (de-)composition (separation of concerns) needs *projection operators* for decomposition and *merge operators* for composition

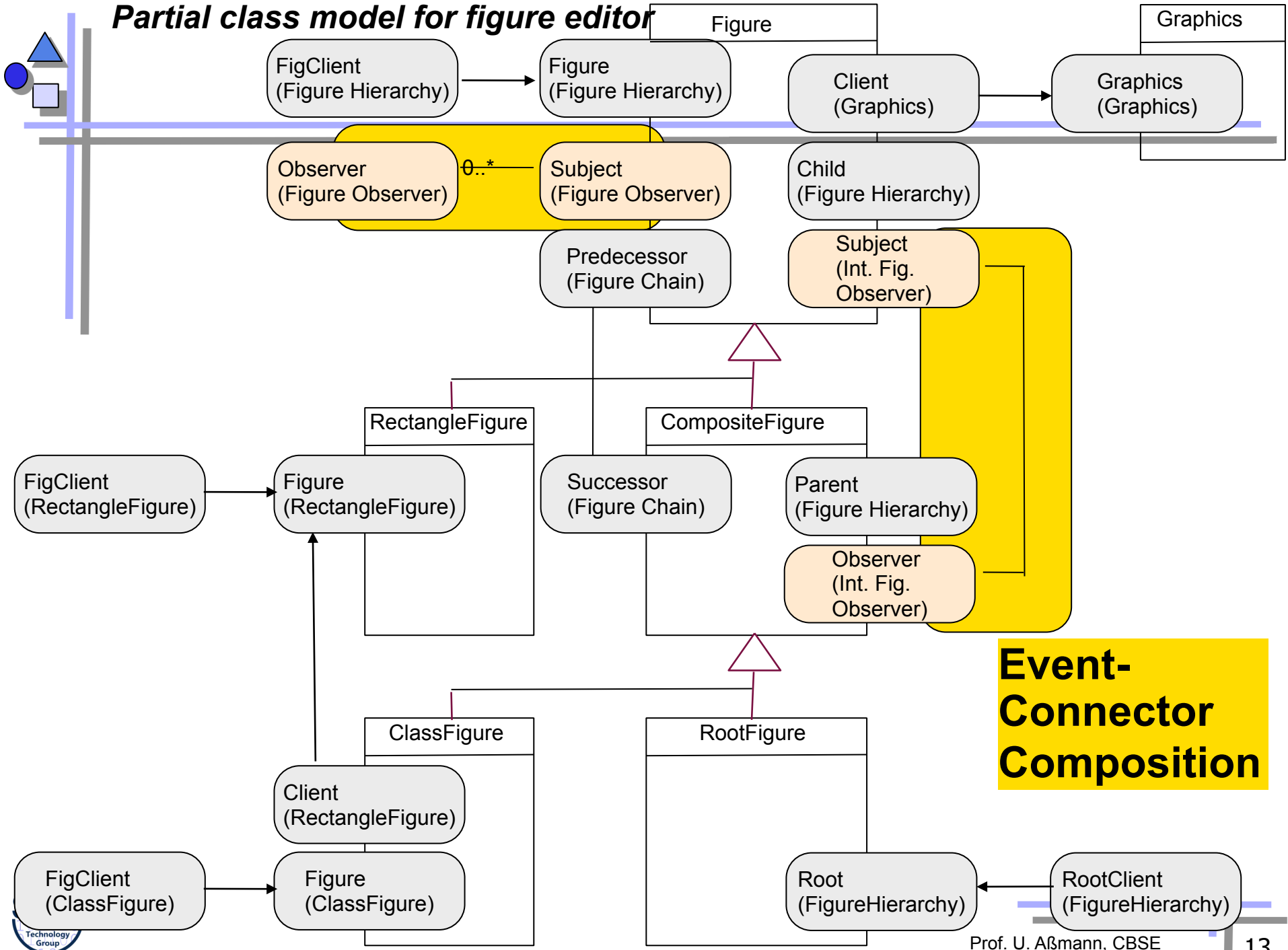


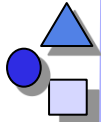
20.1.2 Role Composition and Decomposition in the Role Component Model

Role modeling is a dimensional, view-based
specification technique



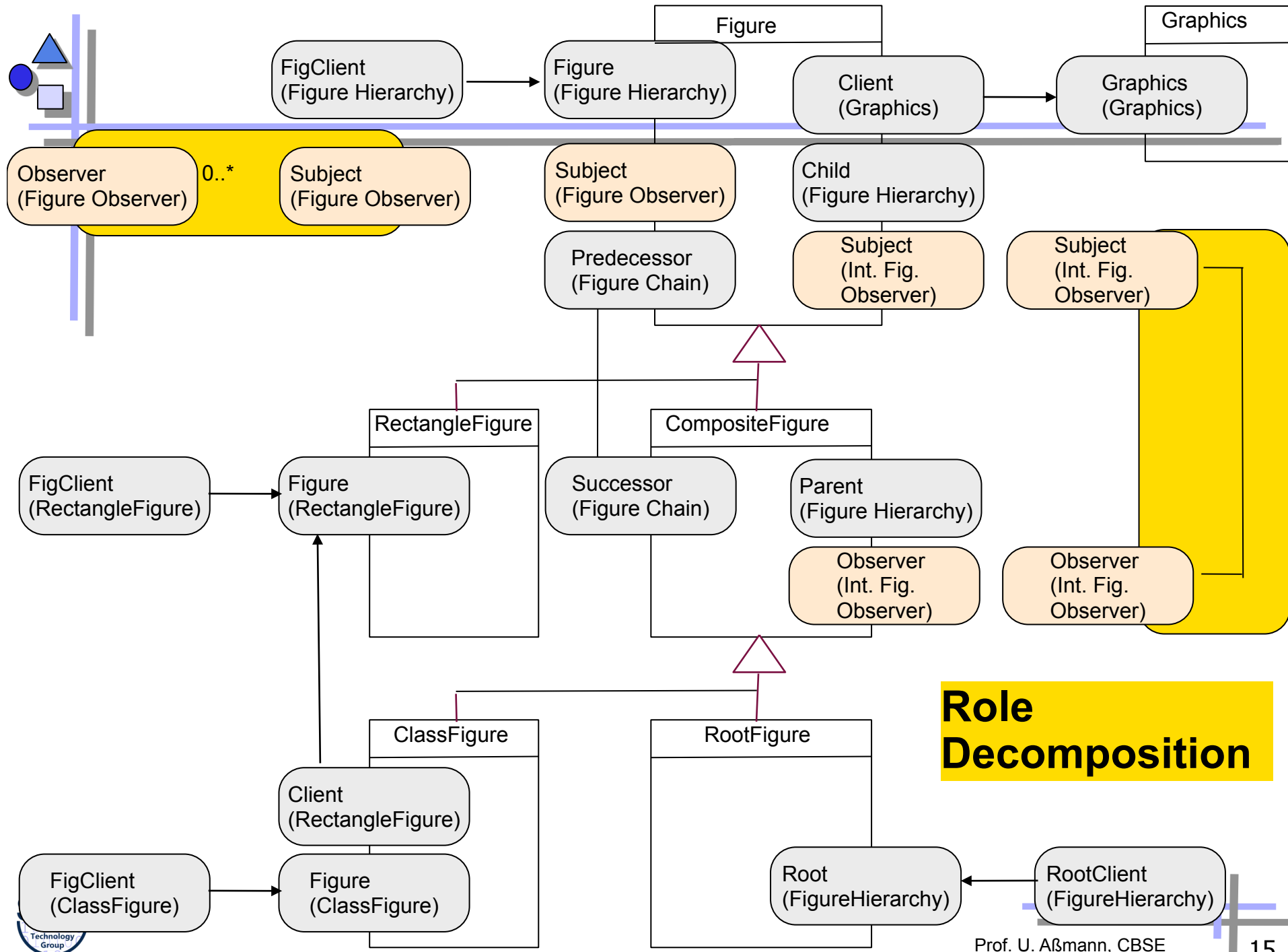
Partial class model for figure editor



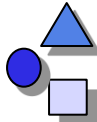


Role Models are Being Composed

- ▶ Roles are *merged to classes*
- ▶ Role models can be decomposed (projected)
 - By role splitting
- ▶ And integrated
 - By role merge or identification



Role Decomposition



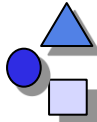
Insight: Role Component Model

- ▶ Because their role models are *integrated* with the role model of the component, connectors work with grey-boxes (Integrating)
- ▶ Roles are a grey-box component model!

**Role-based design relies on a
greybox component model:
composition by role merging
decomposition by role split**

20.2 Systems with Composition Languages for Dimensional De- and Composition





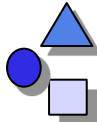
Function Merge in the LambdaN Calculus

- ▶ An extension of the Lambda-calculus [Dami97]
 - Arguments have names by which they are handed over to the callee (as in Ada)
 - No positional parameters as in standard lambda calculus

```
f(p1 => value1, p2 => value2); == f(p2 => value2, p1 => value1);
```

```
f = function (p1, p2) { ... implementation ... }
```

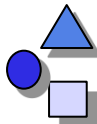
- ▶ Some new reduction rules for the calculus that deal with
 - Name-based argument passing
 - Renaming of names
 - Merging of functions



Function Merge in the LambdaN Calculus

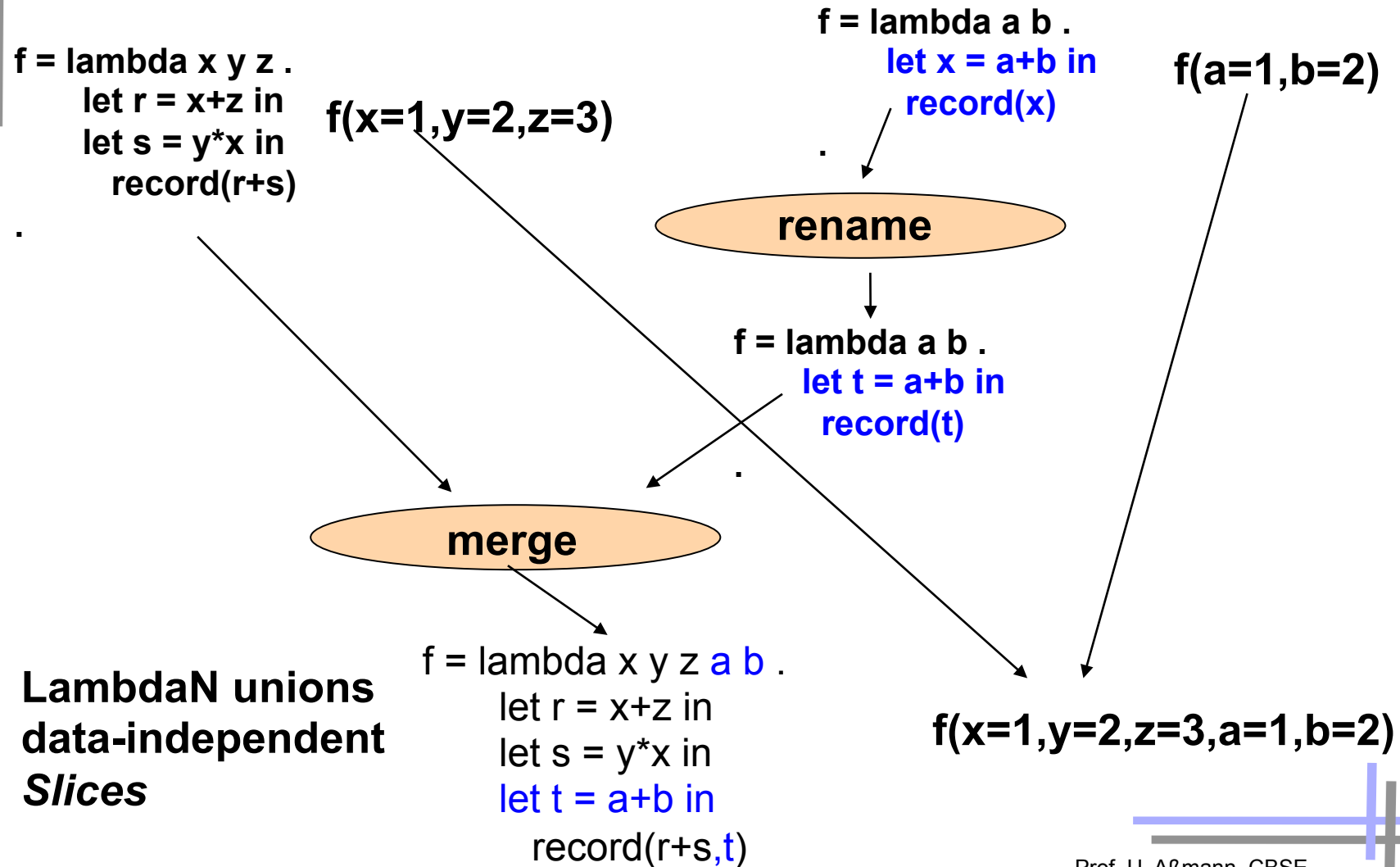
- ▶ Functions can be multiply defined and merged
 - The LambdaN-calculus is based on one simple code merge rule, the merging of lambda expressions (*merge operator* for functions)
 - Currying is possible in arbitrary order

- ▶ LambdaN is the first code calculus for mix of code, i.e., for code composition.



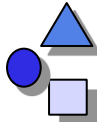
Example

- ▶ Merging of *slices* (black vs blue)



LambdaN unions
data-independent
Slices





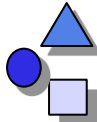
Class Merge in the LambdaN Calculus

- ▶ A class is just a set of functions
 - Classes can be composed by composing the set of functions
- ▶ The merge operator merges implementations, not only of interfaces
 - Role types are partial classes: role model merge can be reduced to lambda merge
- ▶ LambdaN is a higher-order calculus, i.e., is its own composition language
- ▶ Consequence: LambdaN is the perfect calculus to model the semantic base for systems with dimensional decomposition and composition



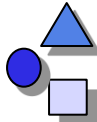
The Power of LambdaN

- ▶ LambdaN can model
 - Role models
 - Classes in object-oriented languages with polymorphism, inheritance, etc.
 - Views
 - Components of any grain size
 - Connectors can be realized, i.e., the calculus subsumes architecture systems
- ▶ Hence, LambdaN can describe all grey-box compositions
 - Composition Filters (wrapping is a merge)
 - Parameterizations (well the calculus is higher order, and functions can be passed as arguments)
 - View-based and aspect-oriented programming (see later)
- ▶ The calculus is *invasive* since functions are merged, i.e., extensions are embedded into extended parts



Sound Composition in the LambdaN

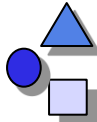
- ▶ A method m is *conformant* to a method n if it can safely replace n in all uses.
- ▶ Merge results of a composition in LambdaN are conformant to their operands (*origins*)!
 - (the resulting f of the previous example is conformant to both of its “ancestors”)
- ▶ Safe composition operations:
 - Extension is safe
 - Adaptation, glueing, aspect weaving is safe



The Composition Language of LambdaN

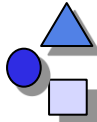
- ▶ The calculus is higher order
 - It's its own composition language
 - It is turing complete
 - It is confluent, i.e., deterministic

- ▶ LambdaN is a sound basis for the next 700 composition languages



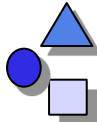
Pi-Calculus

- ▶ The pi-calculus is a calculus for parallel processes (from Milner)
 - A process algebra.
 - Similar to CSP of Hoare
 - Channels (streams) for communication, instead of functional application
- ▶ Pi-calculus scripts model parallel component semantics
 - But also composition semantics
- ▶ The pi-calculus is an “assembler” of composition
 - Non-invasive, i.e., components are black boxes
 - But pi generates glue
 - Higher order, i.e., has its own composition language
- ▶ Pi is another base language for composition



Piccola

- ▶ [Nierstrasz, Schneider, Lumpe, Achermann] from Bern University
- ▶ Derived from Pi-calculus and LambdaN
 - Introduces extensible records for the pi calculus (forms)
 - With these records, all features of LambdaN are inherited
 - Piccola is fully extensible, as LambdaN
 - Higher level language concepts can be mapped to the pi calculus
- ▶ More abstract language, much easier to program
- ▶ Watch out for that group!



History

- ▶ 1988 Aksit Composition Filters
- ▶ Beginning of the 90s: Nierstrasz talks about “Software Composition”
- 1993: Ossher invents subject-oriented programming, an early form of greybox composition
- ▶ 1994: Composition Filters (Bergmans, Aksit)
- ▶ 1996: Invention of AOP (Kiczales)



The End