# 23) View-Based Development

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de

Version 12-1.2, Juni 13, 2012

1. View-based development
2. CoSy, and extensible compiler component framework
3. Subject-oriented programming
4. Hyperspaces
5. Evaluation



## Obligatory Literature

- ISC book, chapter 1, 8+9
- H. Ossher and P. Tarr, Multi-Dimensional Separation of Concerns and The Hyperspace Approach, Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2000 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.3807
- Wikipedia::view_model

## Non-obligatory Literature


- Thomas Panas, Jesper Andersson, and Uwe Aßmann. The editing aspect of aspects. In I. Hussain, editor, Software Engineering and Applications (SEA 2002), Cambridge, November 2002. ACTA Press.
- [COSY] M. Alt, U. Aßmann, and H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In P. A. Fritzson, editor, Proceedings of the International Conference on Compiler Construction (CC), volume 786 of Lecture Notes in Computer Science, pages 278-293. Springer, Heidelberg, April 1994.
- [UWE] Daniel Ruiz-Gonzalez[1], Nora Koch[2], Christian Kroiss[2], Jose-Raul Romero[3], and Antonio Vallecillo. Viewpoint Synchronization of UWE Models. Springer.
- [LL95] Claus Lewerentz and Thomas Lindner. Formal development of reactive systems: case study production cell, volume 891 of Lecture Notes in Computer Science. Springer, Heidelberg, 1995.




# 23.1 View-Based Development

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]

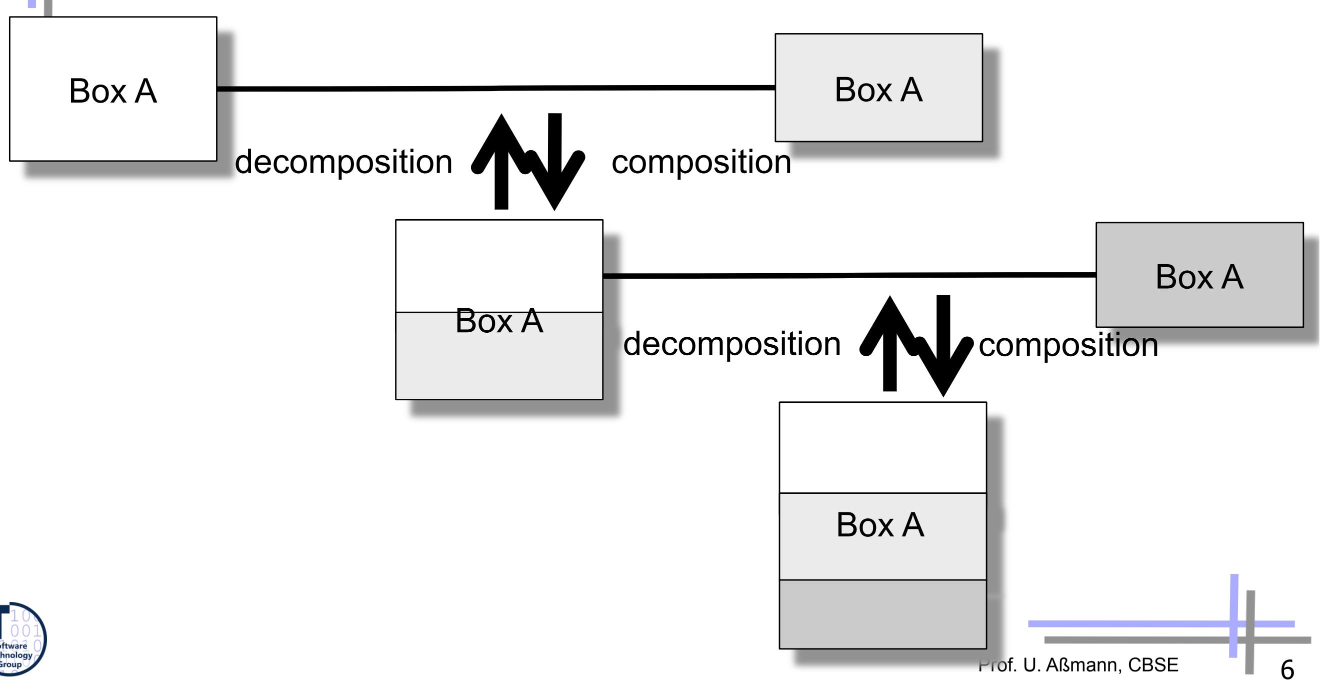
## Constructive and Projective Views

- **Views** are partial representations of a system
  - Views are **constructive** if they can be composed to the full representation of the system
    - Composition needs a merge or extend operator
  - Views are **projective** if they project the full representation of the systen to something simpler
    - Projection extracts a view from the full representation of the system
    - Ex. Views in database query languages
- Views are specified from a **viewpoint (perspective, context)**
  - Viewpoints focus on a set of specific concerns
  - Ex. The architectural viewpoint focuses on
    - The architectural concern
      - the topology and communication
    - The application-specific concern



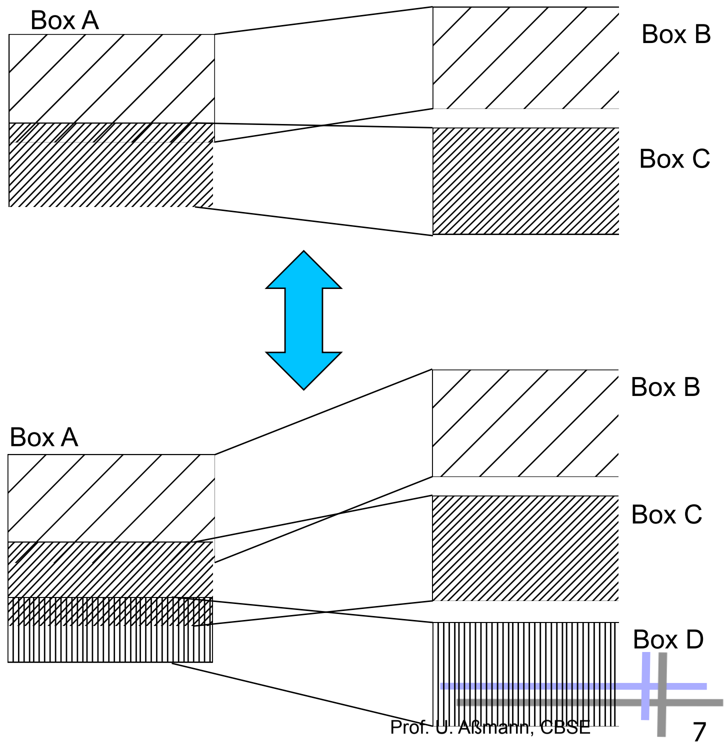
## Constructive vs Projective Views

- Construction (Composition, merge) and projection (decomposition, split) are two sides of one coin



# Constructive Views Require Open Definitions

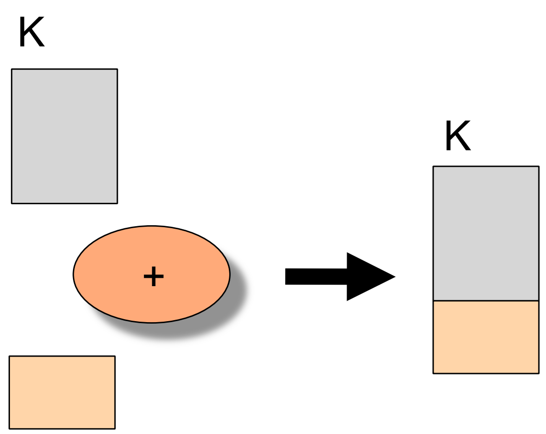
- An **open definition** is a definition of an object that can be re-defined, i.e., extended several times
  - Open definitions can be extended by the *extend* composition operator
- A constructive view contains re-definitions of a set of open definitions
  - Every definition contains partial information





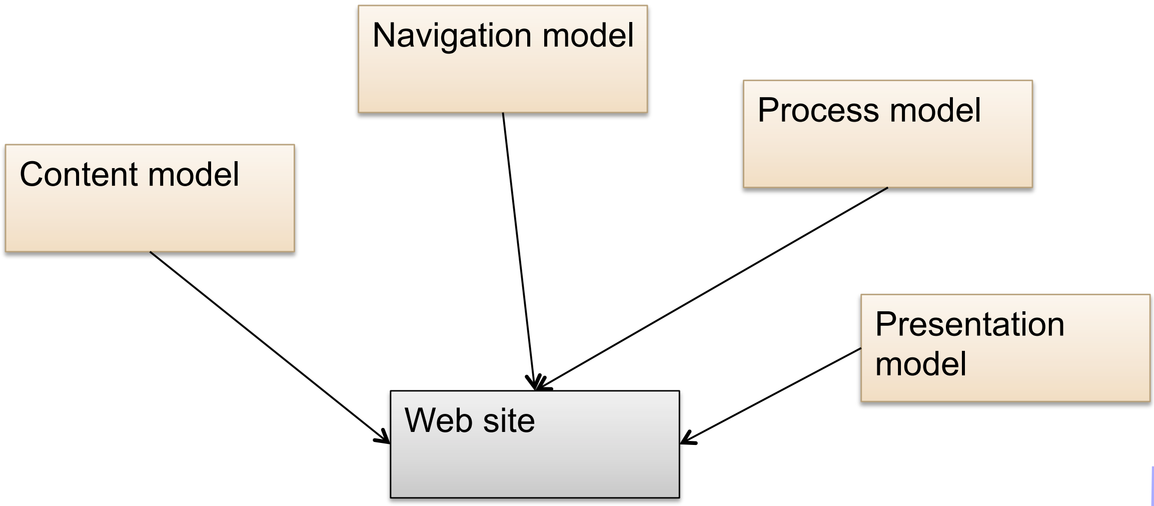

# Merge vs. Extend: Symmetric vs. Asymmetric Composition

- View composition operators can be **symmetric** or **asymmetric**
  - Symmetric composition is commutative
  - Merge of views is symmetric
  - Extend of components is asymmetric
- Both can be implemented in terms of each other



## Example Model-Driven Web Engineering (MDWE)

- [UWE] “This approach has been adopted by most MDWE methodologies that propose the construction of different *views* (i.e., models) which comprise at least a content model, a navigation and a presentation model”





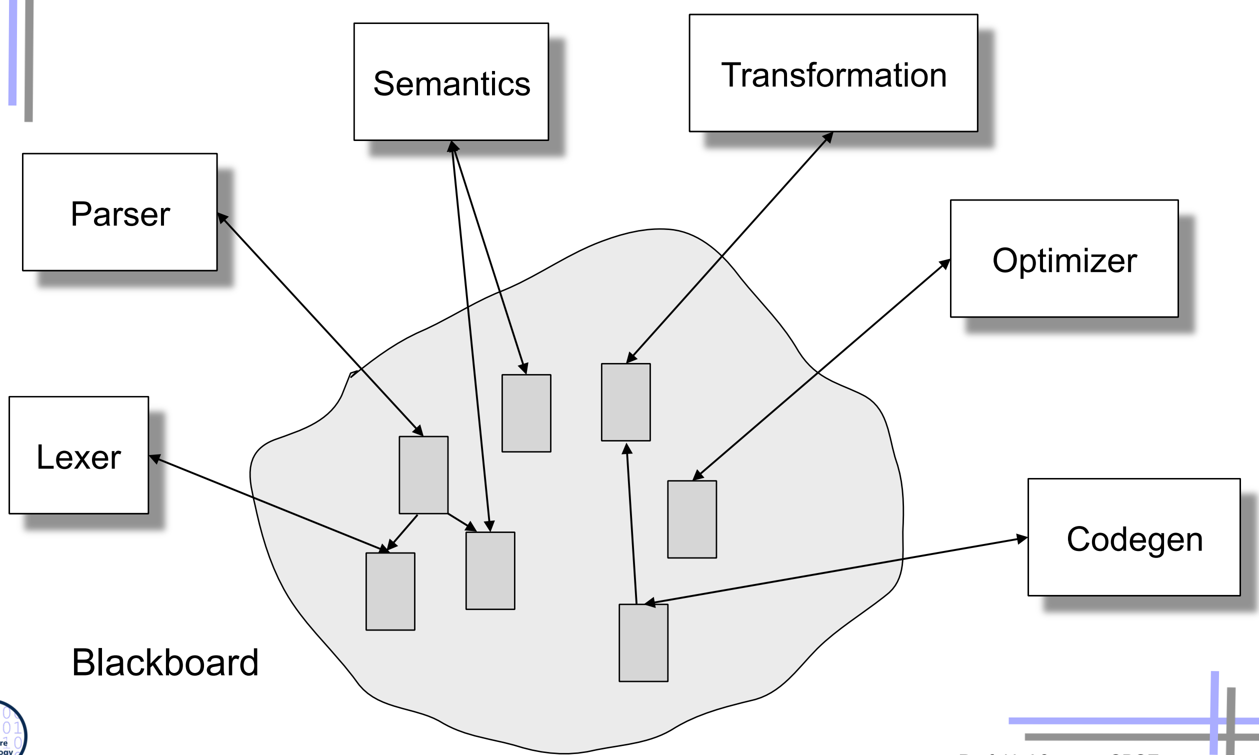

# 23.1. A Composition System based on Constructive Views: CoSy

# Problem: Extensibility (here Compilers)

- CoSy is a modular component framework for compiler construction [Alt/Aßmann/vanSomeren94]
  - Built in 90-95 in Esprit Project COMPARE
  - Sucessfully marketed by ACE bV, Amsterdam
- Goal: *extensible*, easily configurable compilers
  - Extensions without changing other components
  - Plugging from binary components without recompilations
  - New compilers within half an hour
  - Extensible repository by extensible data structures
- Very popular in the market of compilers for embedded systems
  - Many processors with strange chip instruction sets
  - Old designs are kept alive because of maturity and cheap production

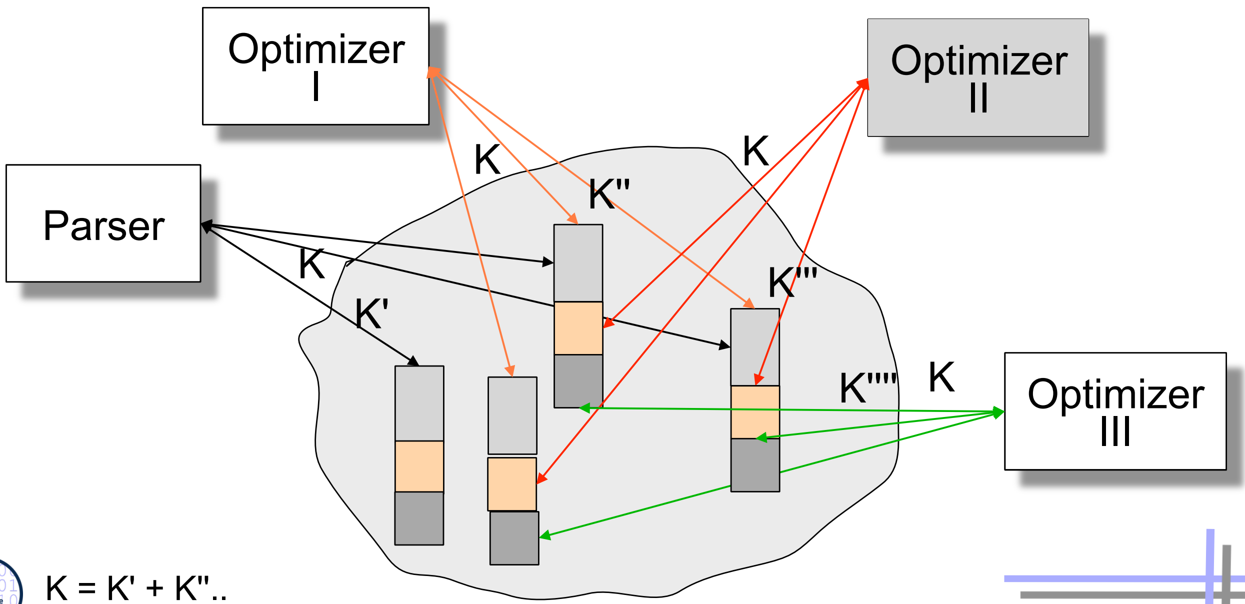


# CoSy Extensible Repository-Architecture



# O-O Technology doesn't fit

- Objects have to be allocated by the parser in base class format, but new components introduce new attributes into the base class

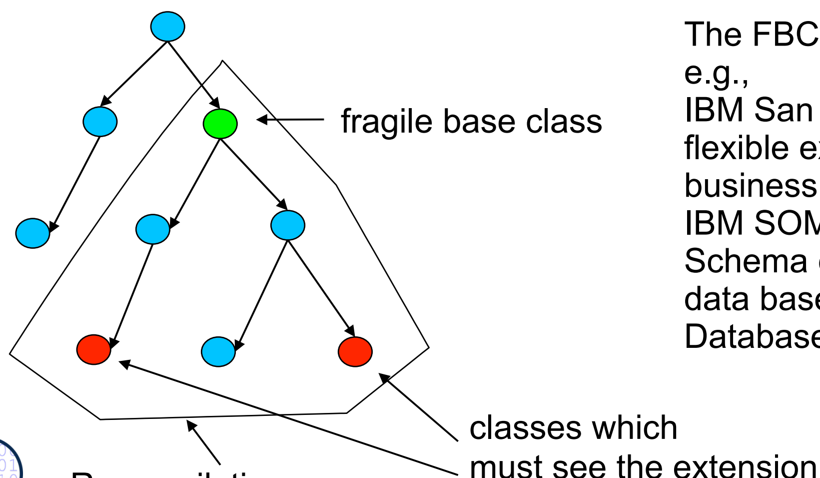



K = K' + K''..



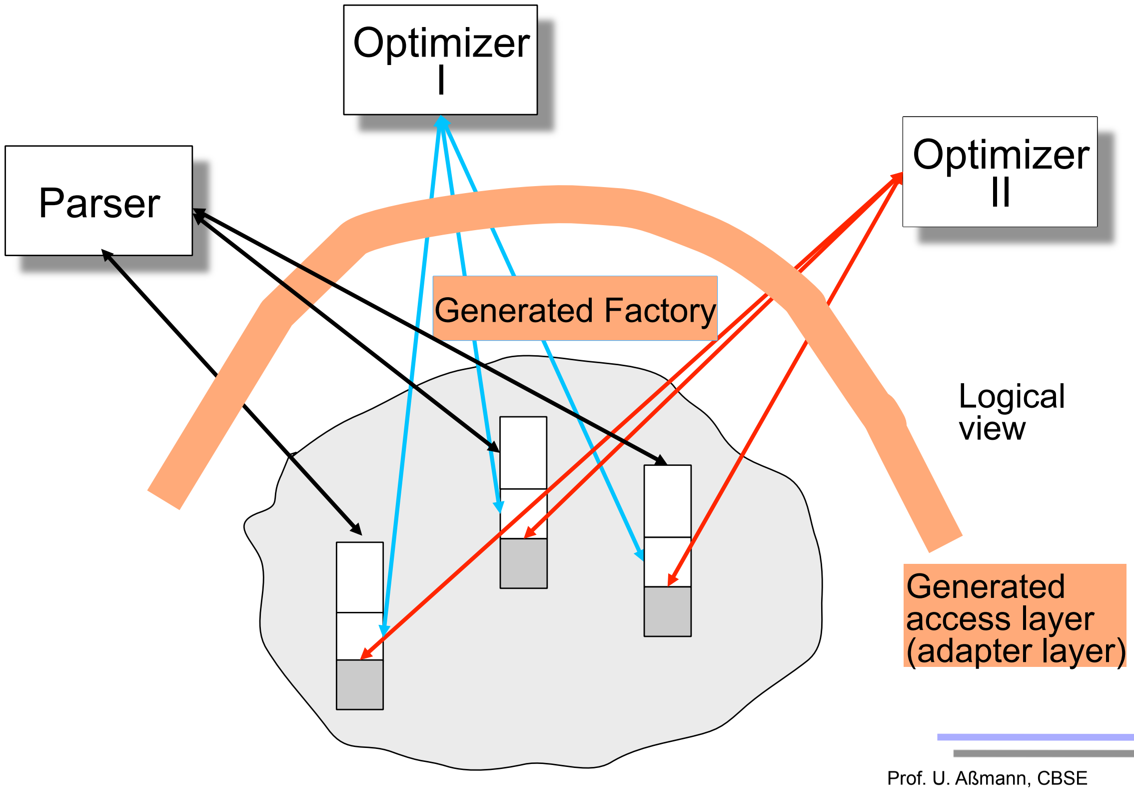
# Syntactic Fragile Base Class Problem

- In unforeseen extension of a system, a base class has to be extended, which is the smallest common ancestor of all subclasses, which must know the extension
- Re-compilation of the class sub-tree required (i.e., the base class is *syntactic fragile*)




The FBCP problem was described in e.g.,
IBM San Francisco: a library with flexible extensible classes and business objects
IBM SOM: release of new versions
Schema changes in object-oriented data bases
Database OBST, FZI, PhD B. Schiefer

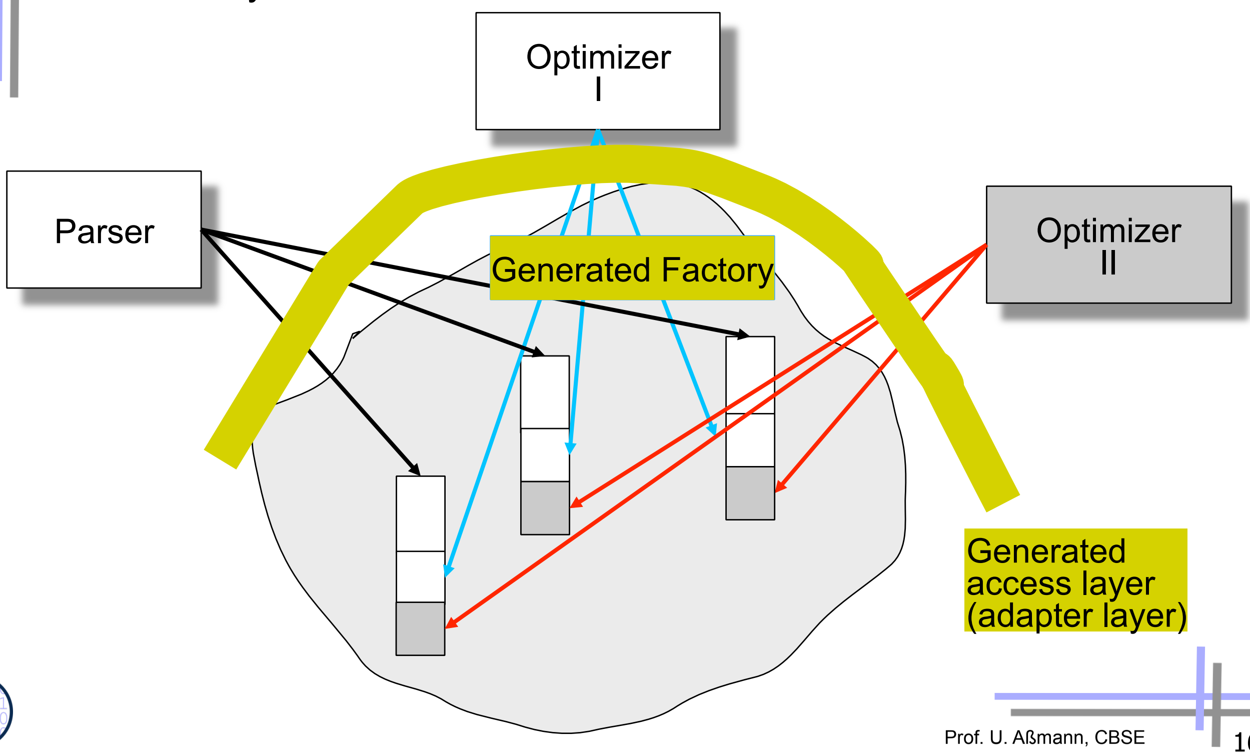
## A CoSy Compiler is Extensible by Constructive Views



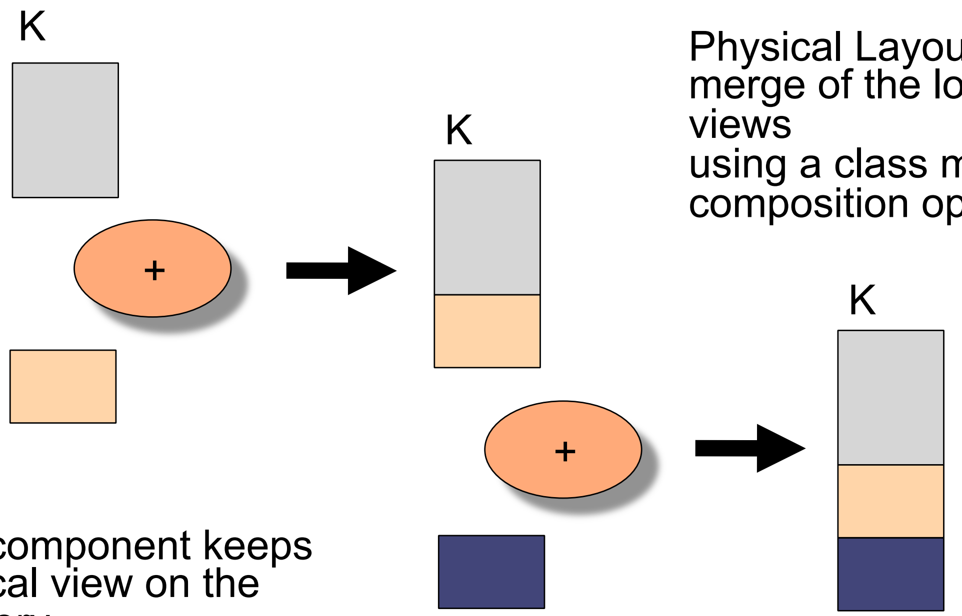



## Extension with Constructive Views

- Extension leads to new repository structure and regeneration of access layer and factories



# CoSy Solution: Construtive Views on the Repository with Extension Operators for Classes



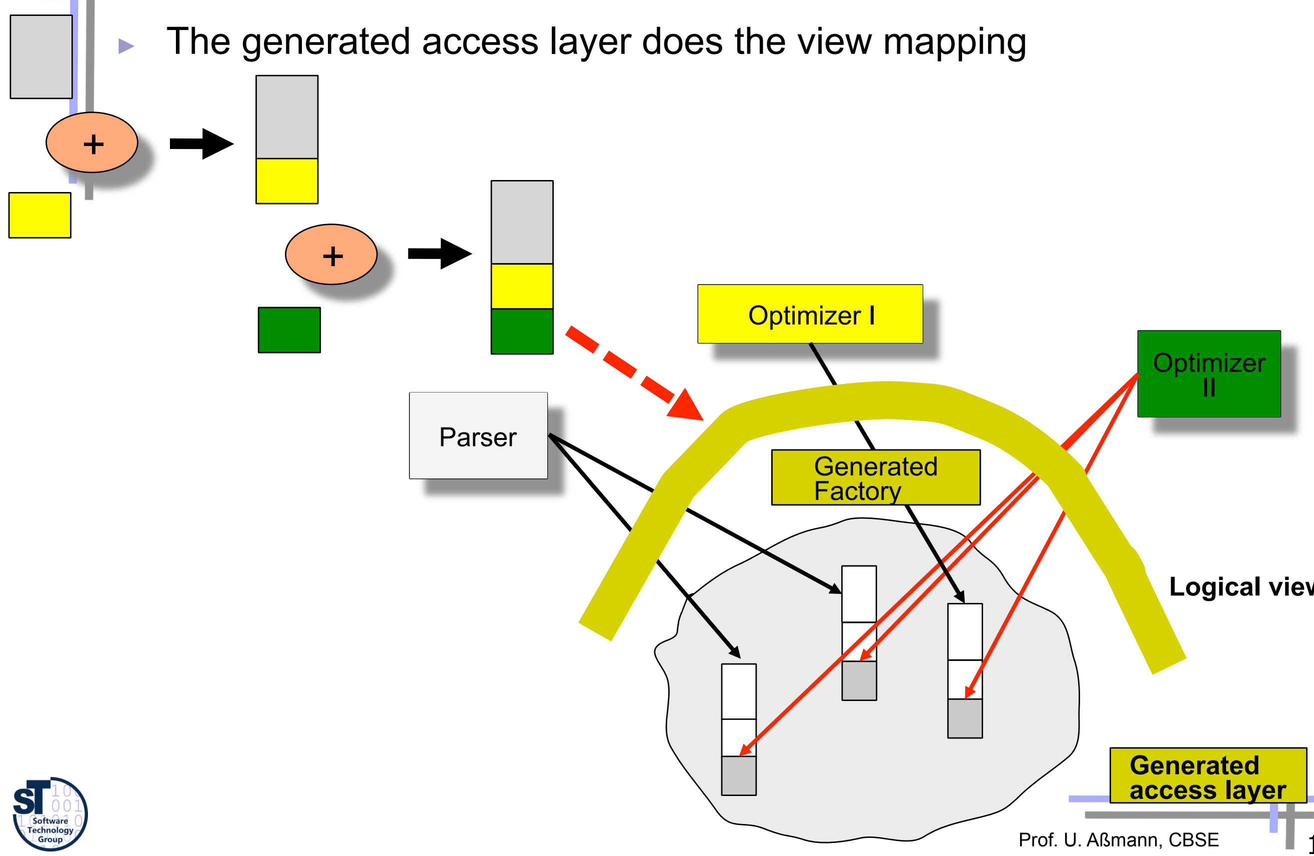

Physical Layout is a merge of the logical views using a class merge composition operator

Every component keeps its logical view on the repository



# Compute from View Specifications the View Mapping Layer

- The generated access layer does the view mapping



# Implementations of Extensions (Views)

- By delegation to view-specific delegatees
  - Uses Role-object pattern: every view defines a role for an object
  - Flexible, extensible at run-time
  - Slow in navigations
  - Splits logical object into physical ones (may suffer from object schizophrenia, if ROP is not carefully followed)
- By extension of base classes (mixin inheritance, GenVoca pattern)
  - Efficient
  - Addresses of fields in subclasses change
  - Leads to hand-initiated recompilations, also at customers' sites (syntactic FBCP)
- By a view mapping layer (the CoSy solution)
  - Fast access to the repository
  - Generative (syntactic FBCP leads to automatic regenerations)



# Advantages of CoSy

- Access level must be efficient
  - Macro implementation is generated
- Due to views, Cosy compilers can be extended easily $$
- Companies reduce costs (e.g. when migrating to a new chip) by improved reuse

Is there a general solution to the extensibility problem?

# 23.2 Subject-Oriented Programming



## Subject-Oriented Programming (SOP)

- SOP provides constructive views by open definitions of classes [Ossher, Harrison, IBM]
- Component model: **subjects** are views on C++ classes
  - Subjects are *partial classes* and consist of
    - Operations (generic methods)
    - Classes with instance variables (members)
  - Mapping of classes and operations to each other
    - (class,operation) realization-poset: describes how to generate the methods of the real class from the compositions and the subjects
- Composition technique:
  - Assemble subjects by composition operators (*mix rules,* composition rules)
- By composition of the subjects the mapping of classes and operations is changed
  - The result of the composition is a C++ class system

# A Simple Subject

Subjects are views on classes
.. and these subjects can be mixed with composition operators

```
Subject: PAYROLL
   Operations: print()
   Classes: Employee()
           with InstanceVariables: _emplName;
   Mapping:
        Class Employee, Operation Print() implemented by
             &Employee::Print()
             // others..
```



# Composition Operators of SOP (Mix Rules)

- **Correspondence operators**: declare equivalence of views of classes
  - Equate: equate method-implementations and subject parts)
  - Correspond: Delegation of delegator and delegatee)
- **Combination operators**
  - Replace: override of features)
  - Join: linking of subject parts)
- **Composed composition operators**
  - Merge := (Join; Equate)
  - Override

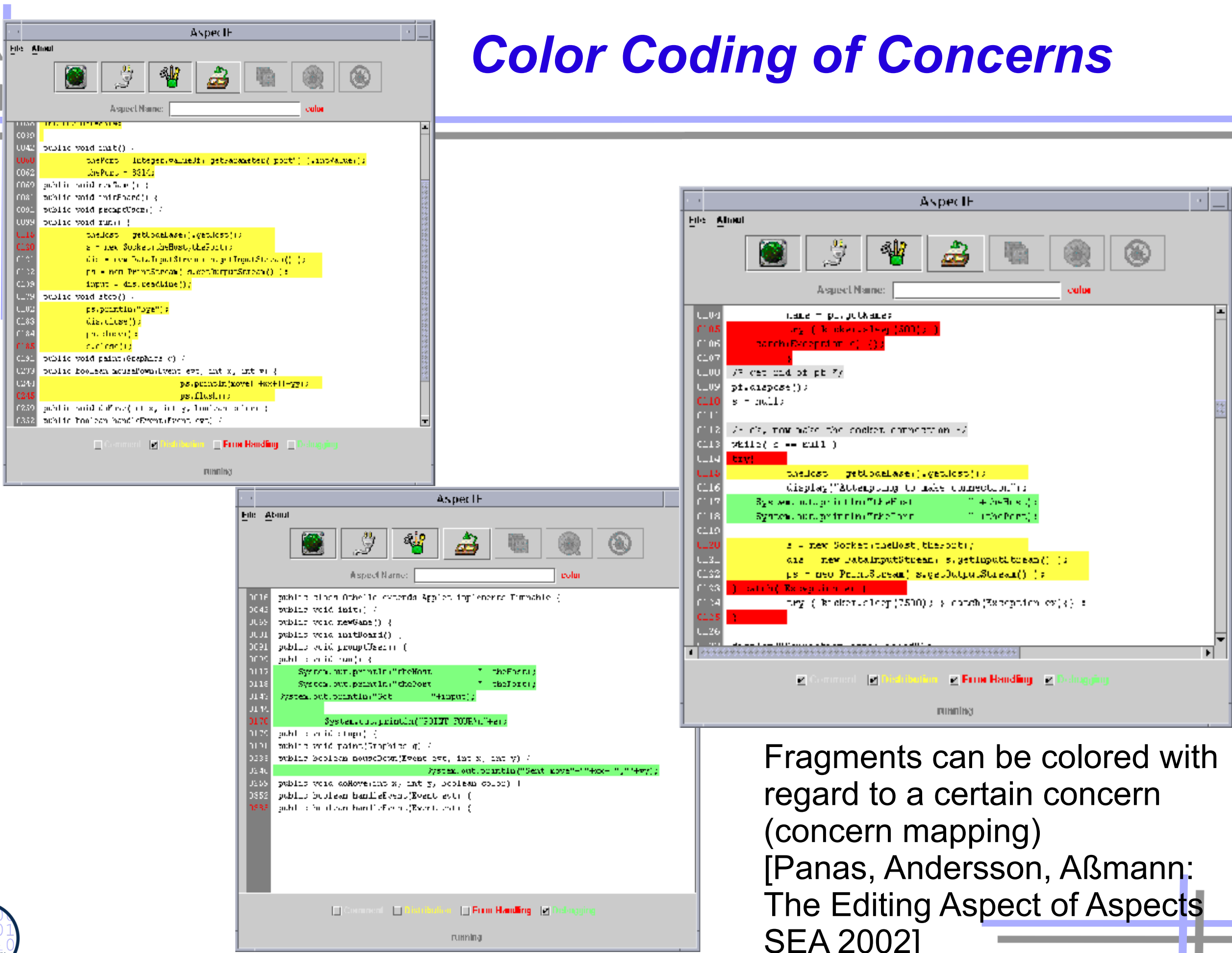
# Evaluation of SOP as Composition System

- Advantage
  - C++ applications become simply extensible with new views that can be merged into existing ones by the extension operators
- Disadvantage:
  - No real composition language: the set of composition operators is fixed!
  - No control flow on compositions



# 23.4 Hyperspaces

# Color Coding of Concerns



Fragments can be colored with regard to a certain concern (concern mapping)
[Panas, Andersson, Aßmann: The Editing Aspect of Aspects SEA 2002]

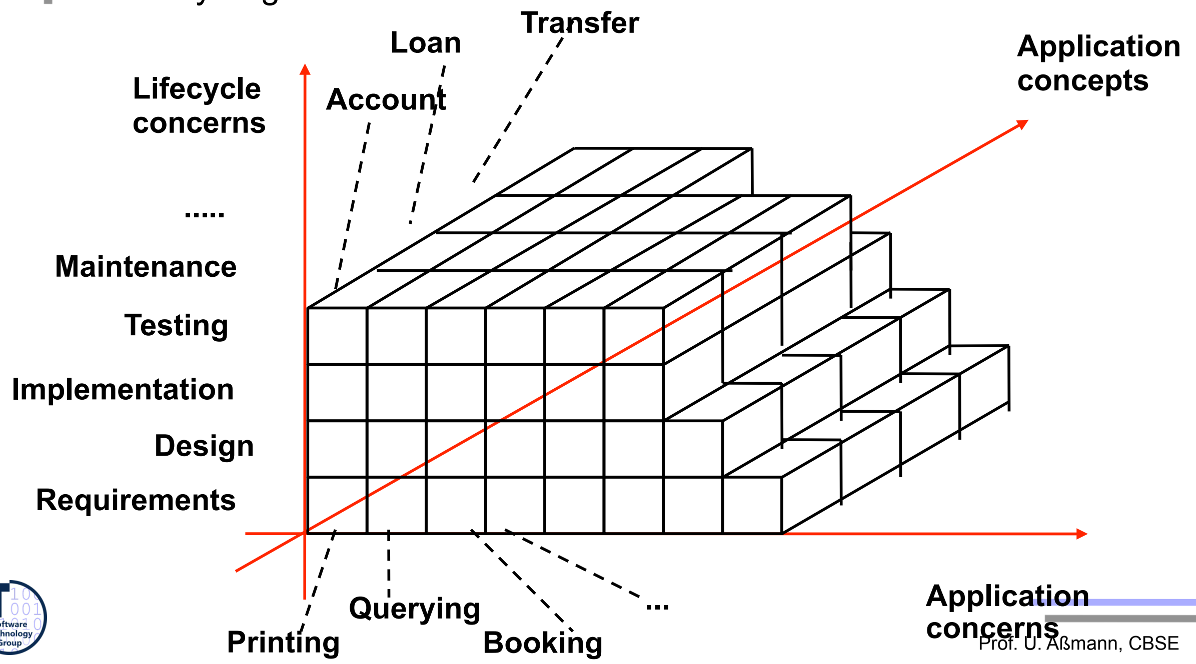


# Hyperspaces

- Hyperspaces generalize SOP
  - Instead of classes, hyperspaces work on sets of *fragments* (aka *units)*, i.e, fragment groups, fragment boxes
  - Open definitions for classes, methods, and all kinds of other definitions
- A hyperspace represents an environment for dimensional development (view-based development)
- A **hyperspace** is a multi-dimensional space over fragment groups
  - Each axis (dimension) is a dimension of software concerns
- Each point on the axis is a *concern*
  - A *concern* groups semantically related fragments

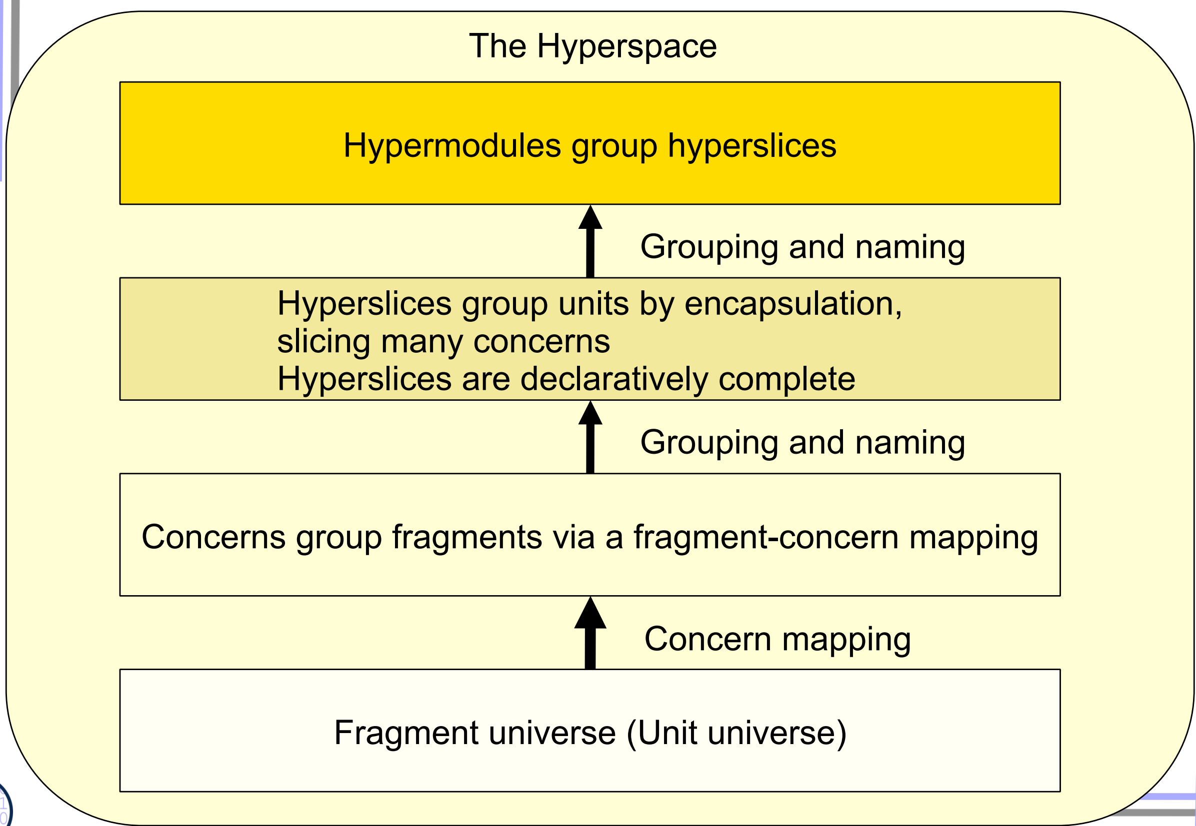
## The Concern Matrix Describes the Artifact Universe, i.e., All Fragments

- Fragments are grouped into an *n-dimensional space of concerns*, arranged in *concern dimensions*
- A point of the space relates to a set of fragments, attached to n concerns
- Every fragment is related to n concerns



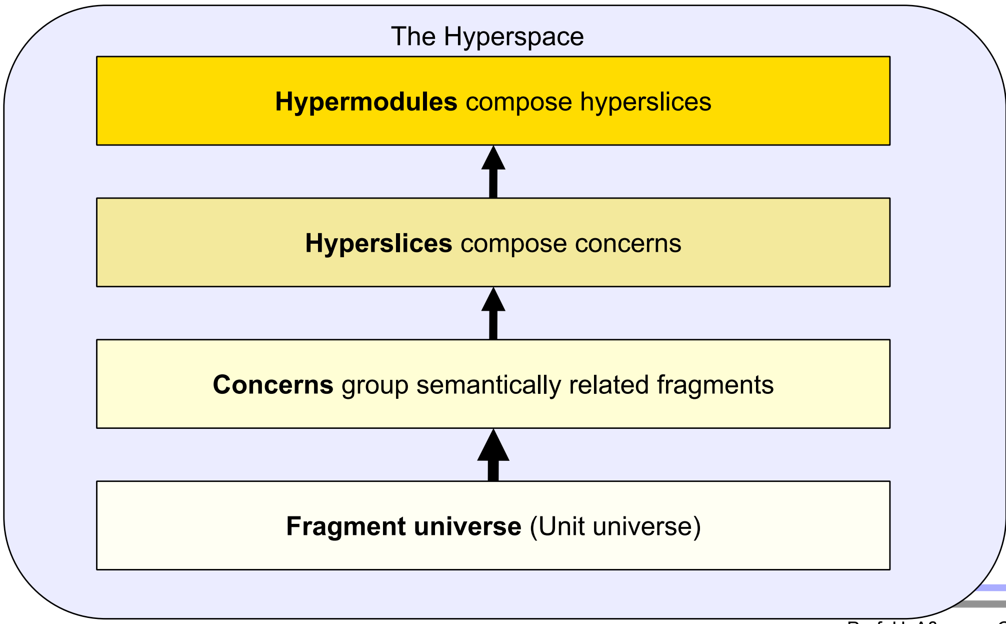



## The Layering in a Hyperspace

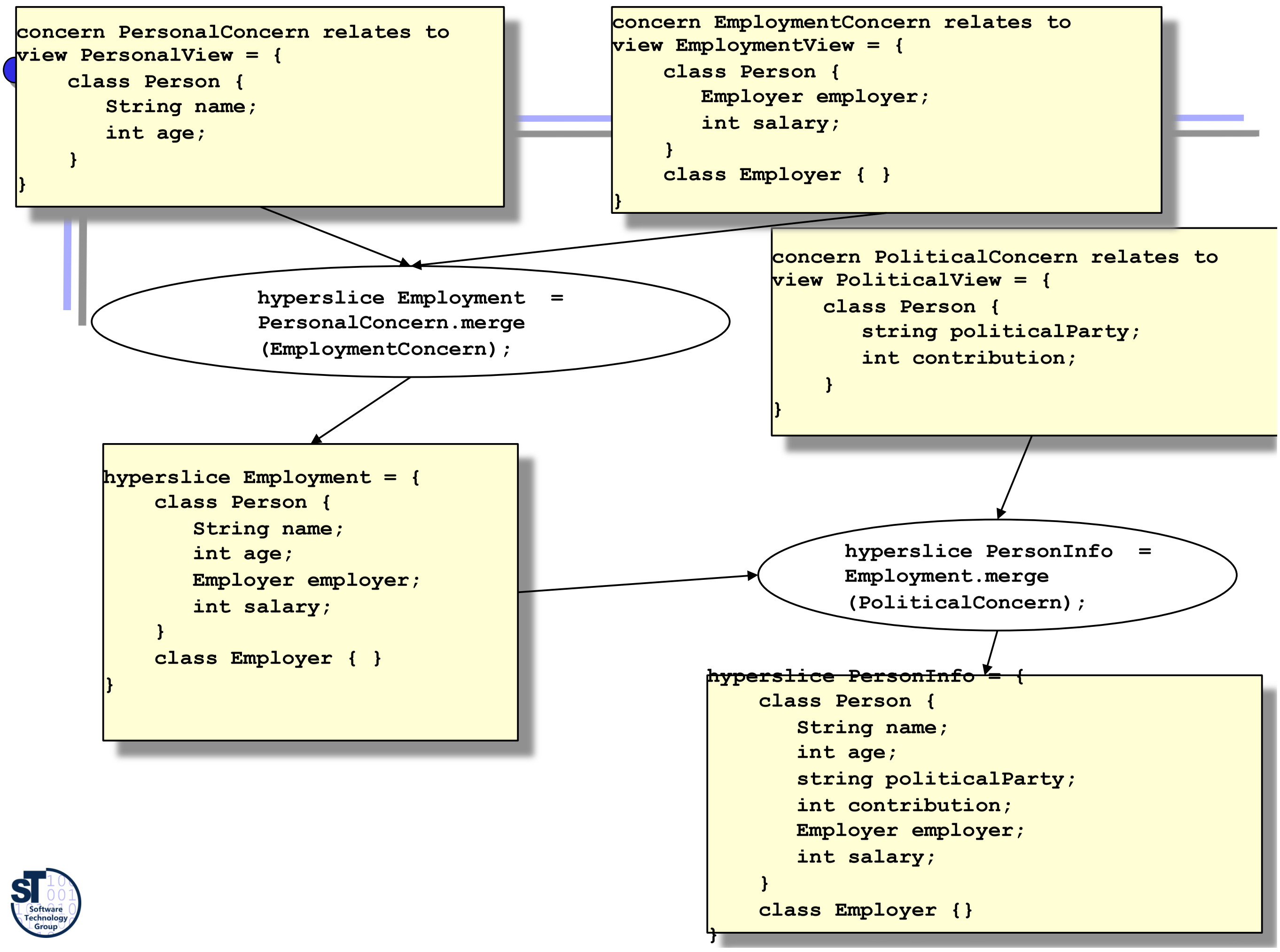


# The Hyperspace, a Fragment Space

- A **hyperslice** is a view (slice) of a system
- A **basic hyperslice** is a view related to one concern of every dimension
- Composition operation: *merge of fragments* in concerns and hyperslices

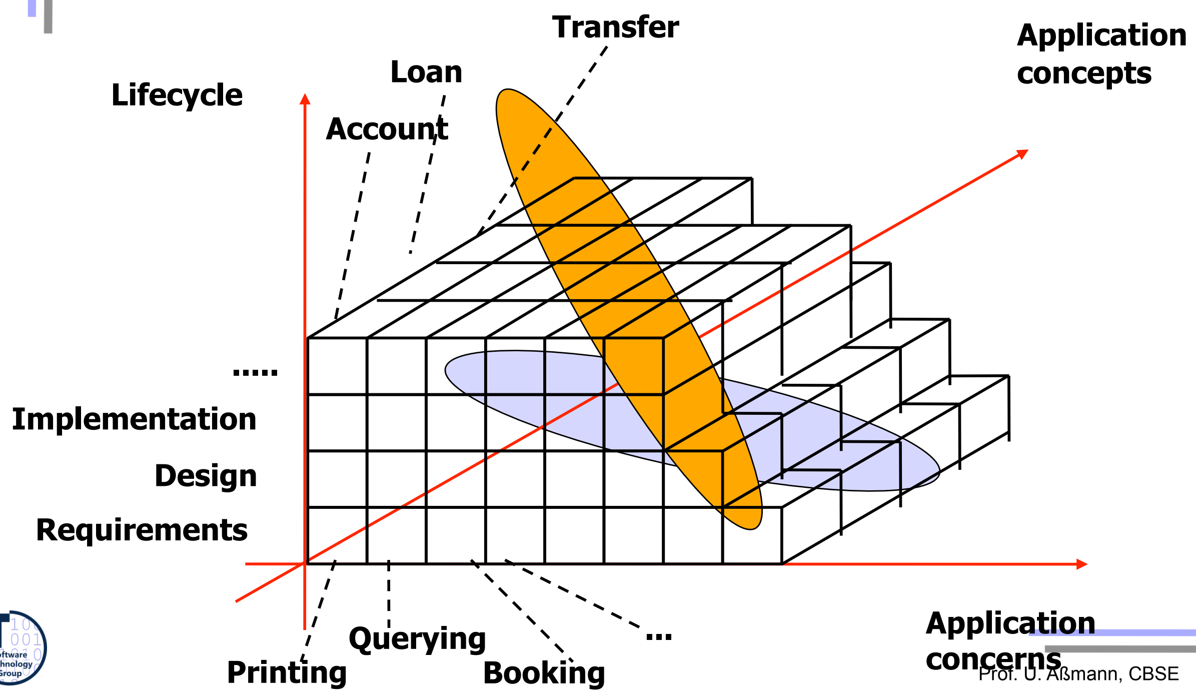




# Hyperslices are Composed out of Concerns

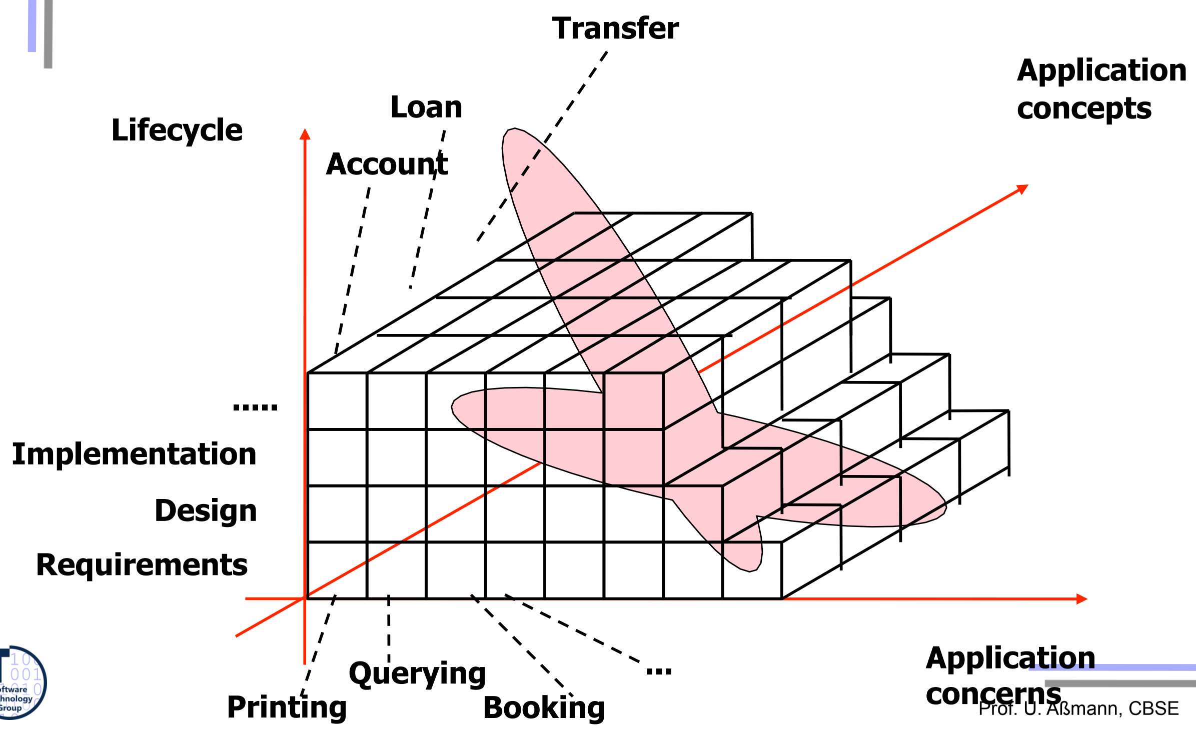
- Hyperslices are named slices through the concern matrix
- A hyperslice is **declaratively complete:** every use has a definition
  - A hyperslice can be compiled and executed





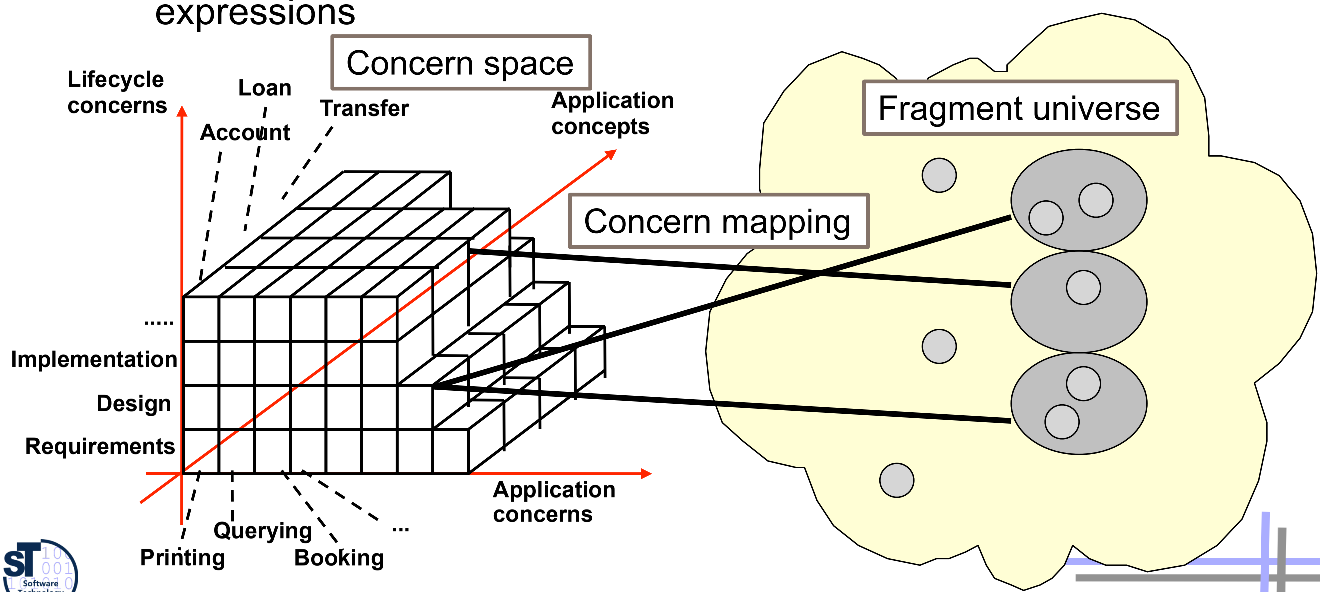

# Hypermodules are Named Compositions of Hyperslices

- Hypermodules are deployable products



## *The Concern Matrix maps Concerns to the Sets of Fragments*

- via a *concern mapping (crosscut graph)*
- one fragment can relate to many concerns:
  - (concern_1, .., concern_n) x fragment
- The concern mapping results from hand-selection and selection/query expressions

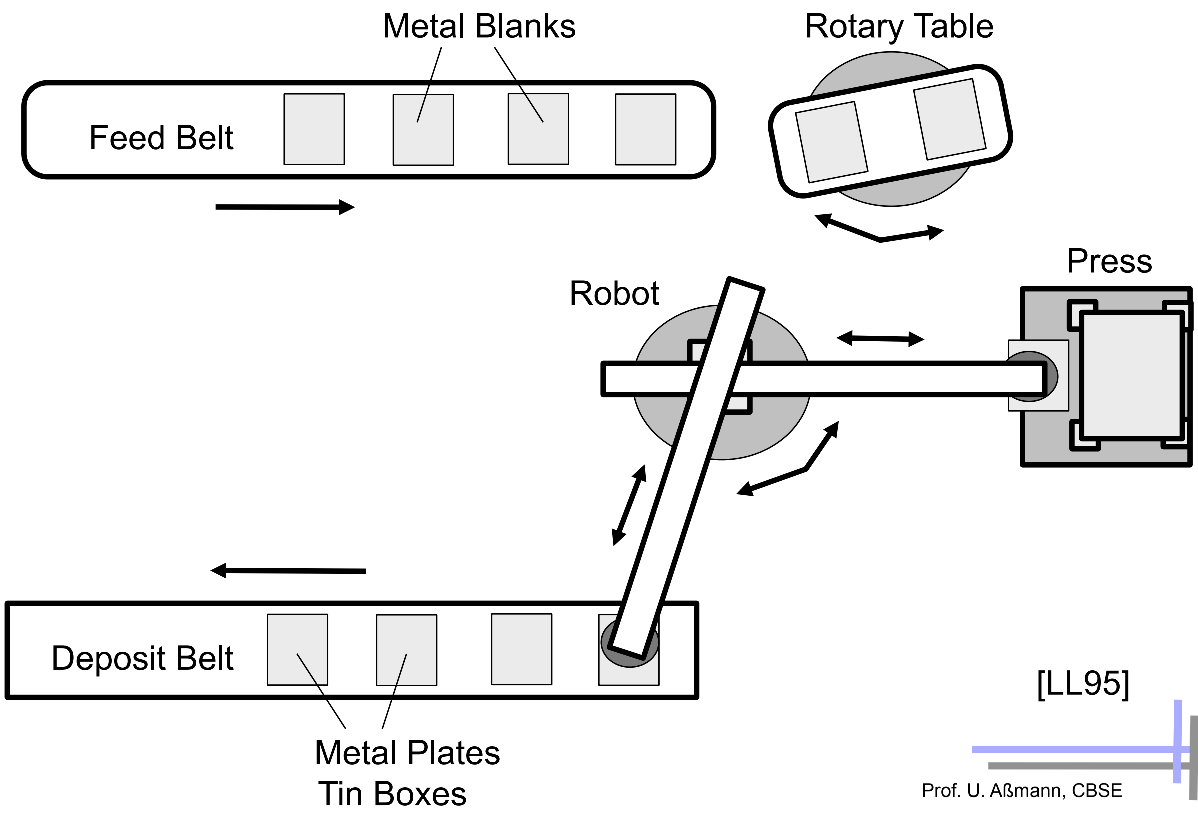





# *23.4.1 Hyperspace Programming*

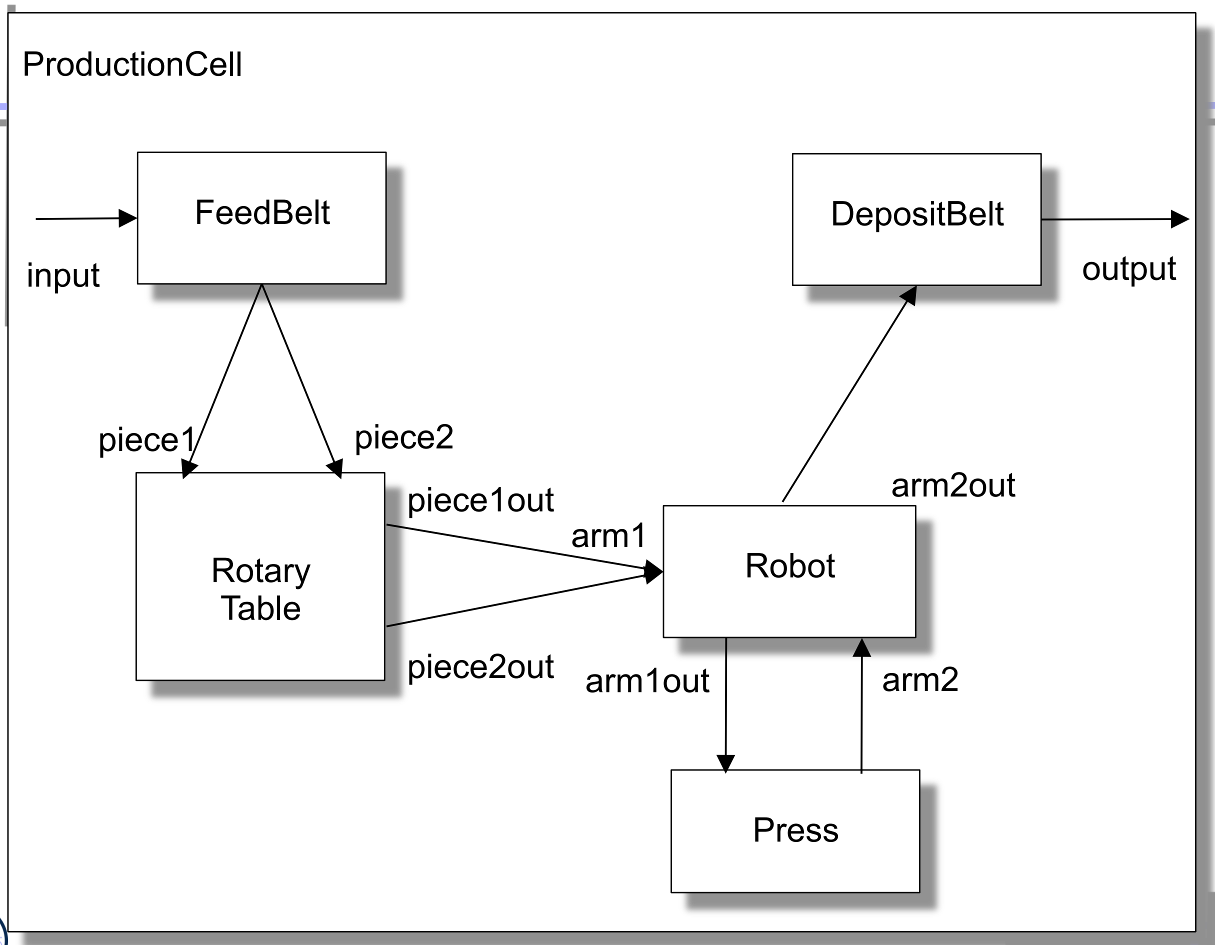
Example

# The Production Cell Case Study




[LL95]





# Component Model

- The components of Hyperspace Programming are *concerns, hyperslices* and *hypermodules*
- The product is a hypermodule

- **Domain concerns** will group the machines and materials of the production cell
- **Technical concerns** group issues with regard to software technology
- **Lifecycle concerns** group issues with life cycle of the software



# Composition Technology – Description of the Artifact Universe

- The following treats only Hyper/J, an instance of Hyperspaces for Java
  - The fragment universe (hyperspace) is a subset of some Java packages, classes and methods
  - Hyper/J supports a selection language to describe the hyperspace
  - Java methods are the fragment unit
- Here, example ProductionCell
  - The hyperspace, ProductionCell, is a selection of classes from some packages:

```
// define a hyperspace in Hyper/J by „sucking in" some Java
// packages
hyperspace ProductionCell
   composable class passiveDevices.*
   composable class activeDevices.*
   composable class tracing.*
```

# Composition Technology – Concern Mapping

- For package **passiveDevices**, we define the following *concern mapping* between concerns and Java fragments
  - First, we define a default concern, **Feature.WorkPieces**, which includes by default every member in the package.
  - Then, the mapping specifies for specific members that they belong to a second concern, **Feature.Transfer**.
  - All features belong to one of two concerns of dimension Feature
    - Concerns are named **<dimension>.<concern>**

```
// Decompose the package passiveDevices
// into concerns
package passiveDevices:               Feature.WorkPieces
   operation lifeCycle:               Feature.Transfer
   field ConveyorBelt.pieces:         Feature.Transfer
   operation setPieces:               Feature.Transfer
   operation setPiecesNumber:         Feature.Transfer
   operation getPiecesNumber:         Feature.Transfer
```

Default mapping for the entire package

Dimensions and concerns

Specific mappings

Fragments

Mapping



# Composition Technology – Concern Mapping

- A second package, **activeDevices**, models the behavior of active devices.
  - It contains the classes **Press** and **Robot**.
- The package is grouped into three domain concerns,
  - **Feature.ActiveDeviceBehavior**, **Feature.Transfer**, and **Feature.Action**

Default mapping for the entire package

```
// Decompose the package activeDevices into concerns
package activeDevices:          Feature.ActiveDeviceBehavior
   operation Press.takeUp:      Feature.Transfer
   operation Robot.takeUp:      Feature.Transfer
   operation lifeCycle:         Feature.Action
```

Specific mappings

# Composition Technology – Concern Mapping

A third *technical* concern, **`Logging.Tracing`**, groups all methods from class **`TracingAttribute`**

Default mapping for the entire package

Specific mappings

```
// Decompose the package tracing into concerns
package tracing: Logging.Tracing
class TracingAttribute: Logging.Tracing, Logging.Data
// This implies:
// operation TracingAttribute.enterAttribute : Logging.Tracing
// operation TracingAttribute.leaveAttribute : Logging.Tracing
```



# Composition Language: Grouping Concerns/Views to Hyperslices

- Now, we can define the hyperslices of transfer, workpieces, and tracing
  - They are declaratively complete concerns
- and compose a hypermodule
  - that groups the hyperslices of transfer, workpieces, and tracing, describing the transfer of workpieces in the production cell
- This hypermodule merges the three hyperslices by name, and brackets all operations of all classes with tracing code.
  - It doesn't contain code that is concerned with actions.

```
hypermodule TracedProductionCellTransfer
  hyperslices: Feature.Transfer, Feature.WorkPieces, Logging.Tracing
  relationships: mergeByName
  bracket "*"."*"
    before Logging.Tracing.TracingAttribute.enterAttribute()
    after Logging.Tracing.TracingAttribute.leaveAttribute()
```

# Finally, a System is a Hypermodule

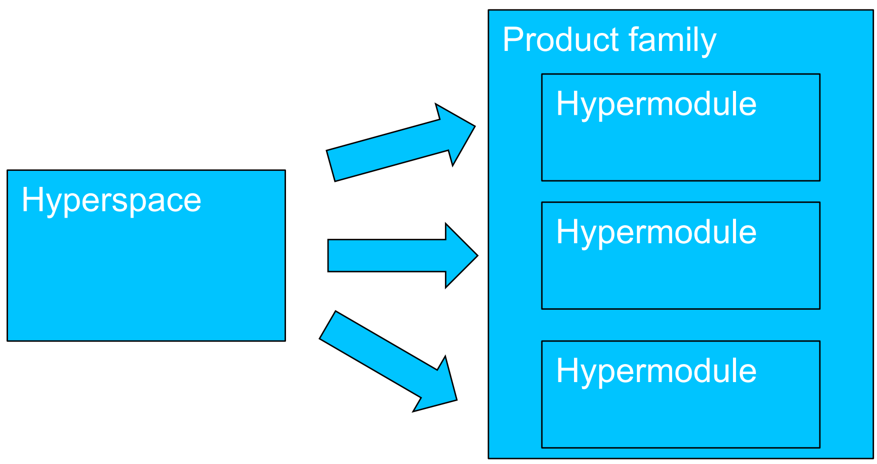
- Another hypermodule groups active devices without tracing
- Features can override features in other hyperslices
  - Here, features of active devices override transfer features
  - Although the method **`lifeCycle`** from package **`passiveDevices`** is contained in concern **`Feature.Transfer`**, the version of concern **`Feature.ActiveDeviceBehavior`** overrides it,
  - and the resulting hypermodule will act in the style of active devices.

```
hypermodule ProductionCell
   hyperslices: Feature.Transfer, Feature.WorkPieces,
        Feature.ActiveDeviceBehavior
   relationships: overrideByName
```



# Variability in Hyperspaces

- With Hyper/J, variants of a system can be described easily by grouping and composing the hyperslices, and -modules together differently
- Different selection of concerns and hyperslices makes up different products in a product family
- Hyperspaces can include software documentation, requirements specifications and design models



# Advantages of the Hyperspace Approach

- Compositional merge resp. extension of fragment sets
  - Classes
  - Packages
  - Methods
  - Hyperslices

**Universal extensibility:** A language is called *universally extensible*, if it provides extensibility for every collection-like language construct.

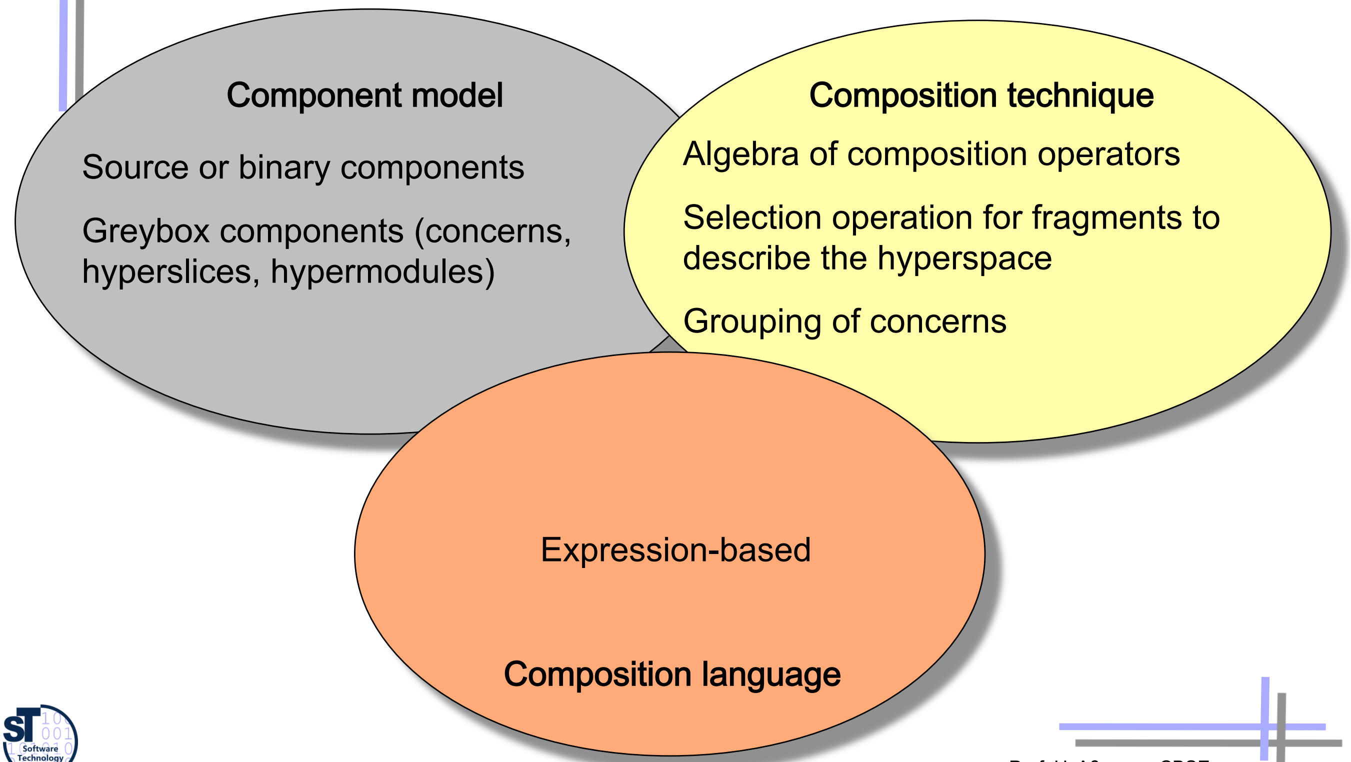


# Universal Composability: Universal Genericity vs Universal Extension

- BETA and hyperspaces look really similar
  - Fragment components
  - *slots* vs *hooks* (parameterization vs extension interface)
  - *bind* vs *merge* composition operations
- BETA is a *generic* component approach
- Hyperspaces is an *extensible* component approach

**Universal composability:** A language is called *universally composable*, if it provides universal genericity and extension.

# 23.5 Evaluation: Hyperspaces as Composition System






# The End - Appendix

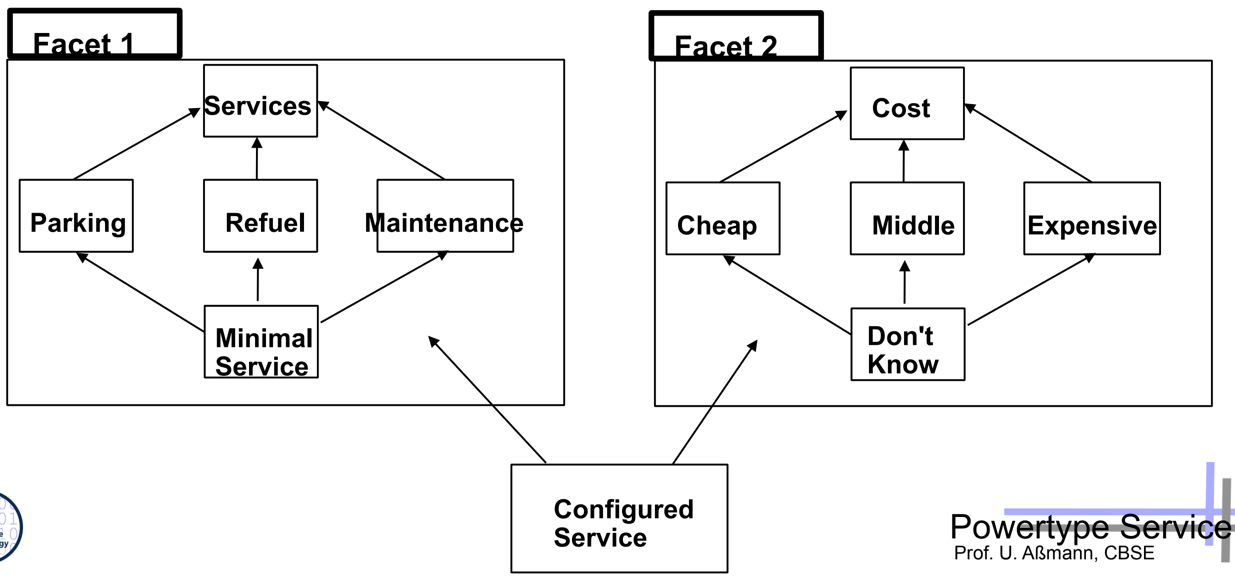
# Side Remark: Concern Matrix and Facet Matrix

- The concern matrix is similar to a facet space
  - Dimensions correspond to facets
    - Dimensions *partition* the universe differently (n dimensions == n partitions)
  - Concern dimensions correspond to *flat facets*, lattices of height 3
    - Concerns in one dimension *partition* the facet
- Difference of concern matrix and facet matrices
  - Facets describe an object; concerns do not describe an object, but describe all objects and subjects in the univers
  - Concerns are more like *attributes*



# (remember DPF) Facet Spaces are Dimensional Spaces over Objects

- describing *one object,* not a fragment space
- When the facets are *flat*, every facet makes up a dimension
- Bottom is 0
- Top is infinity



# Side Remark: The Facet Matrix Describes Objects Dimensionally