43. View-Based Development

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

Version 16-0.1, Juni 4, 2016

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.

43.1 View-Based Development

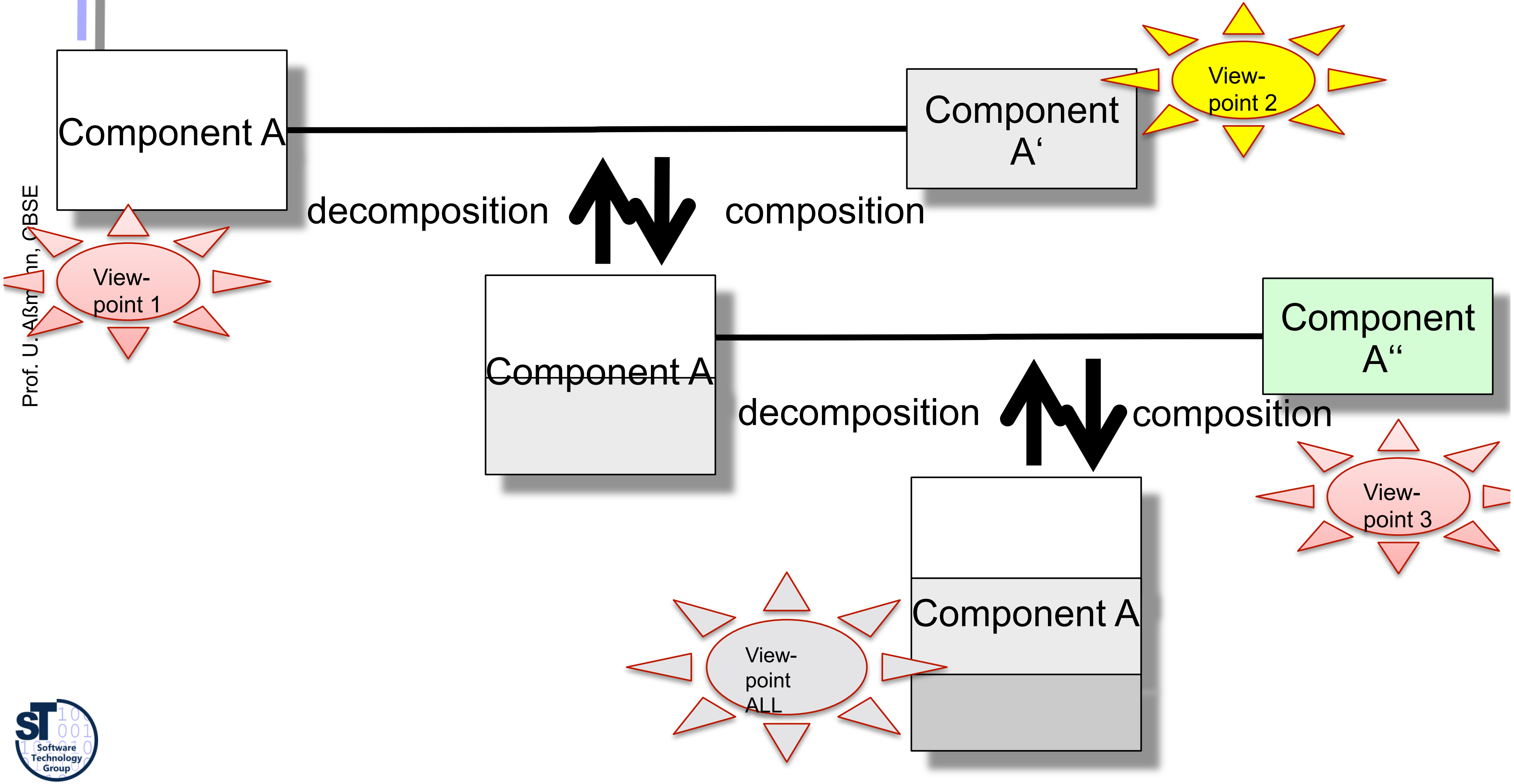
A *view* is a representation of a whole system from the perspective of a related set of concerns
[ISO/IEC 42010:2007, Systems and Software Engineering -- Recommended practice for architectural description of software-intensive systems]

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* (symmetric composition) or extend (asymmetric composition)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

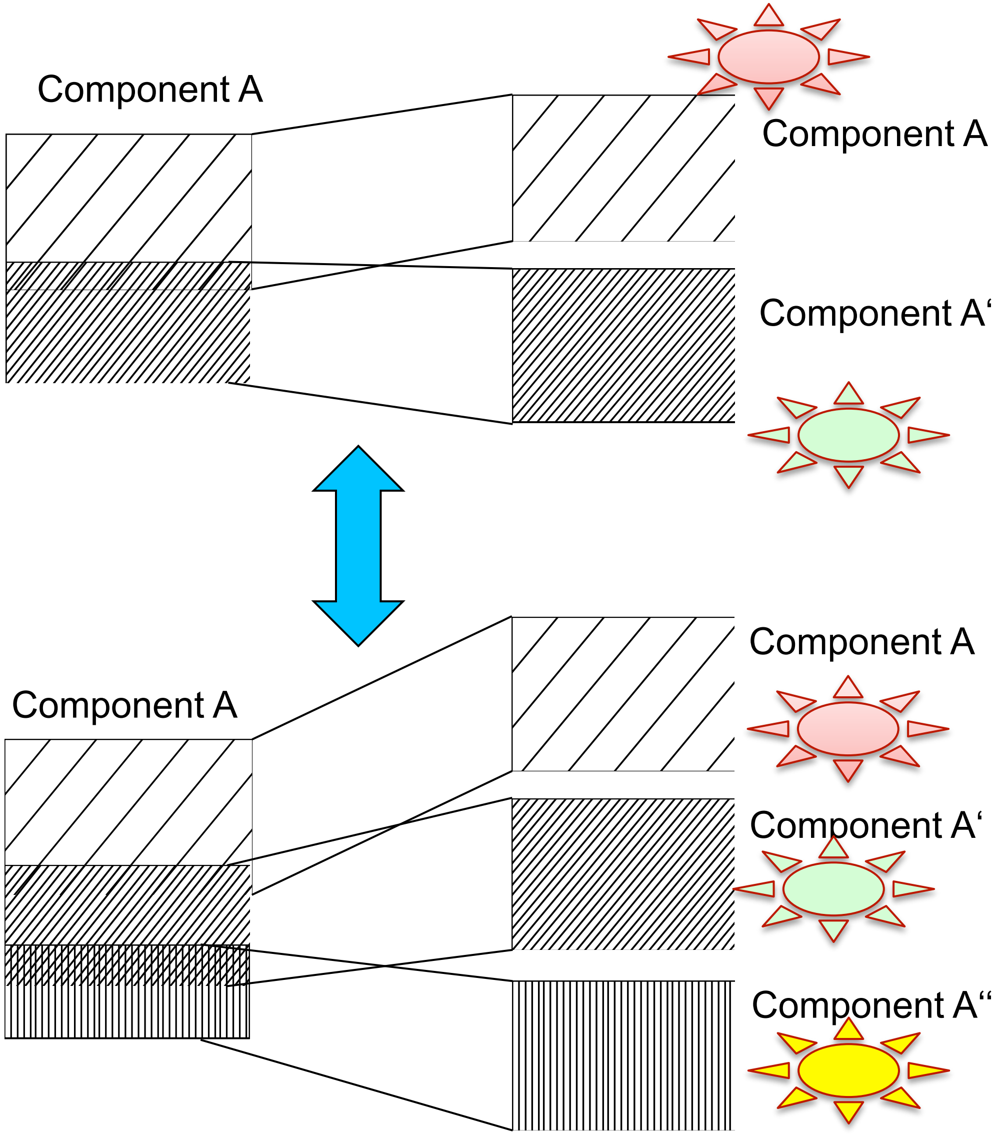
Constructive vs Projective Views

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



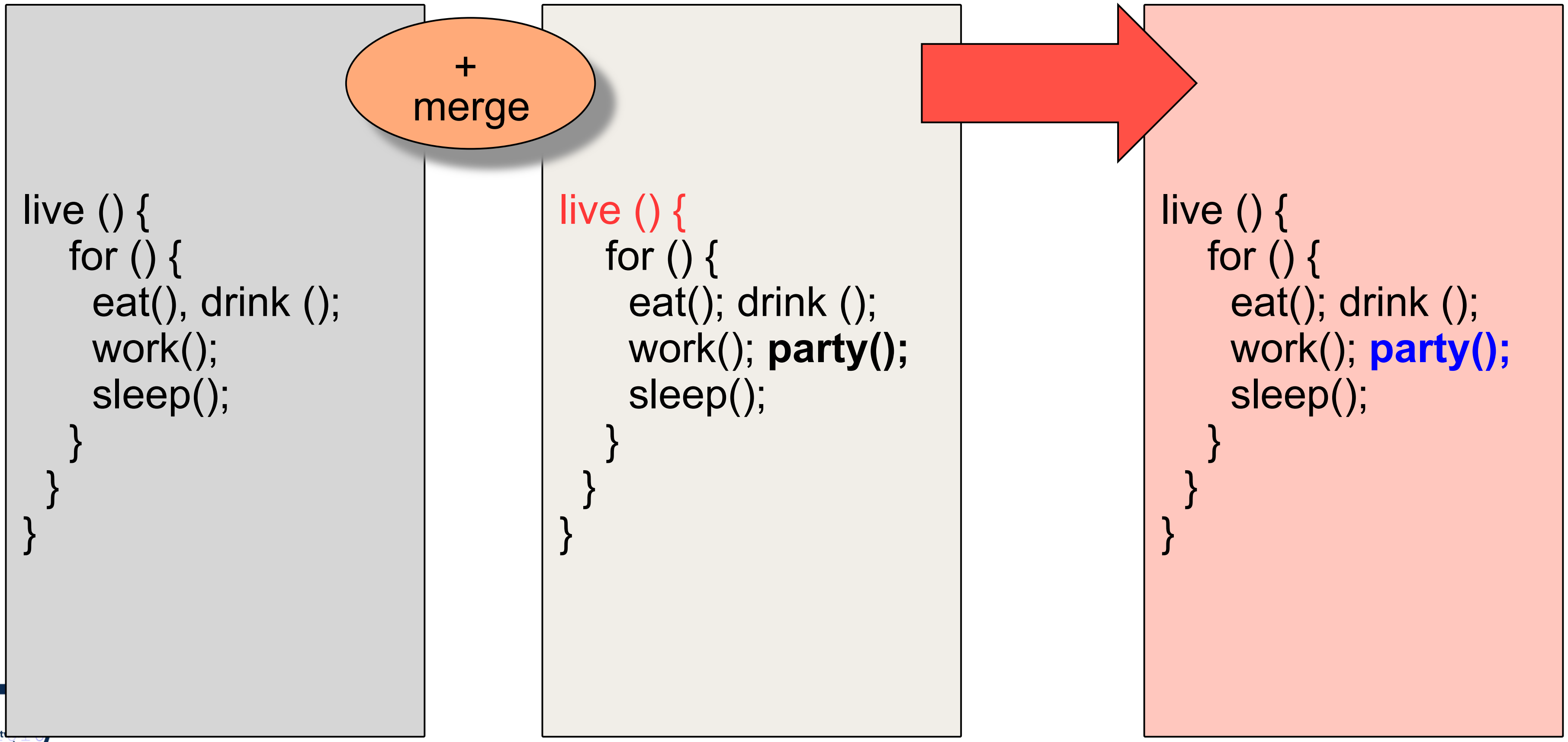
Constructive Views Require Open Definitions

- An **open definition** is a view definition of an object that can be re-defined, i.e., extended several times by different viewpoints
  - 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



Remember: The Lambda-N Calculus Merges Functions

- Functions in Lambda-N are open definitions
  - Redefinitions are possible
  - Merge is automatic



```
live () {
    for () {
      eat(), drink ();
      work();
      sleep();
    }
  }
}
```

+ merge

```
live () {
    for () {
      eat(); drink ();
      work(); party();
      sleep();
    }
  }
}
```

```
live () {
    for () {
      eat(); drink ();
      work(); party();
      sleep();
    }
  }
}
```

Example: Merging Classes

- Merging means Unification (merge by name): Identify
  - **Common elements:** merge
  - *Disjoint elements:* union
  - conflicting elements: try to resolve conflicts

```
class Person {
  String name;
  int salary;
  work() { .. }
  drink { .. }
  eat() { .. }
  live () {
    for () {
      eat(), drink ();
      work();
      sleep();
    }
  }
}
```

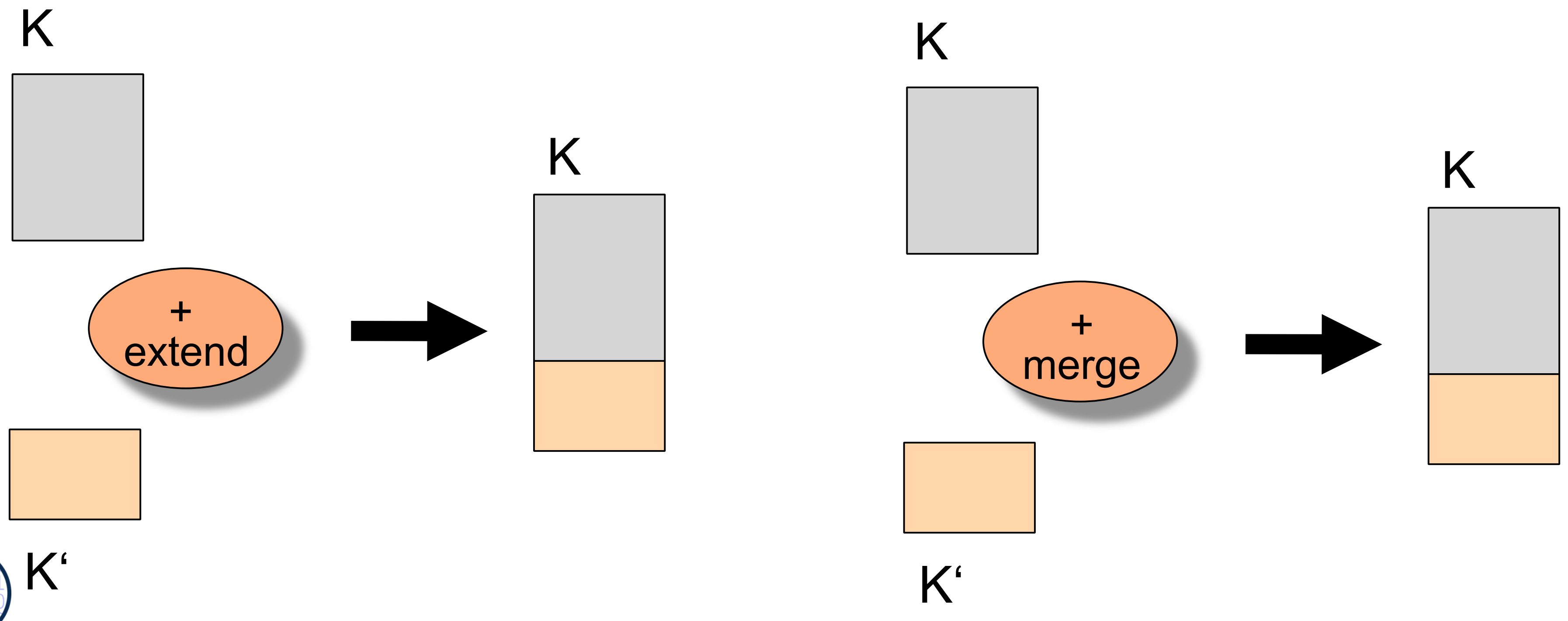
```
class Person {
  String name;
  real salary;
  work() { .. }
  party{ .. }
  breathe() { .. }
  live () {
    for () {
      eat(); drink ();
      work(); party();
      sleep();
    }
  }
}
```

\+ merge

```
class Person {
  String name;
  real salary;
  work() { .. }
  party{ .. }
  breathe() { .. }
  drink() { .. }
  eat() { .. }
  live () {
    for () {
      eat(); drink ();
      work(); party();
      sleep();
    }
  }
}
```

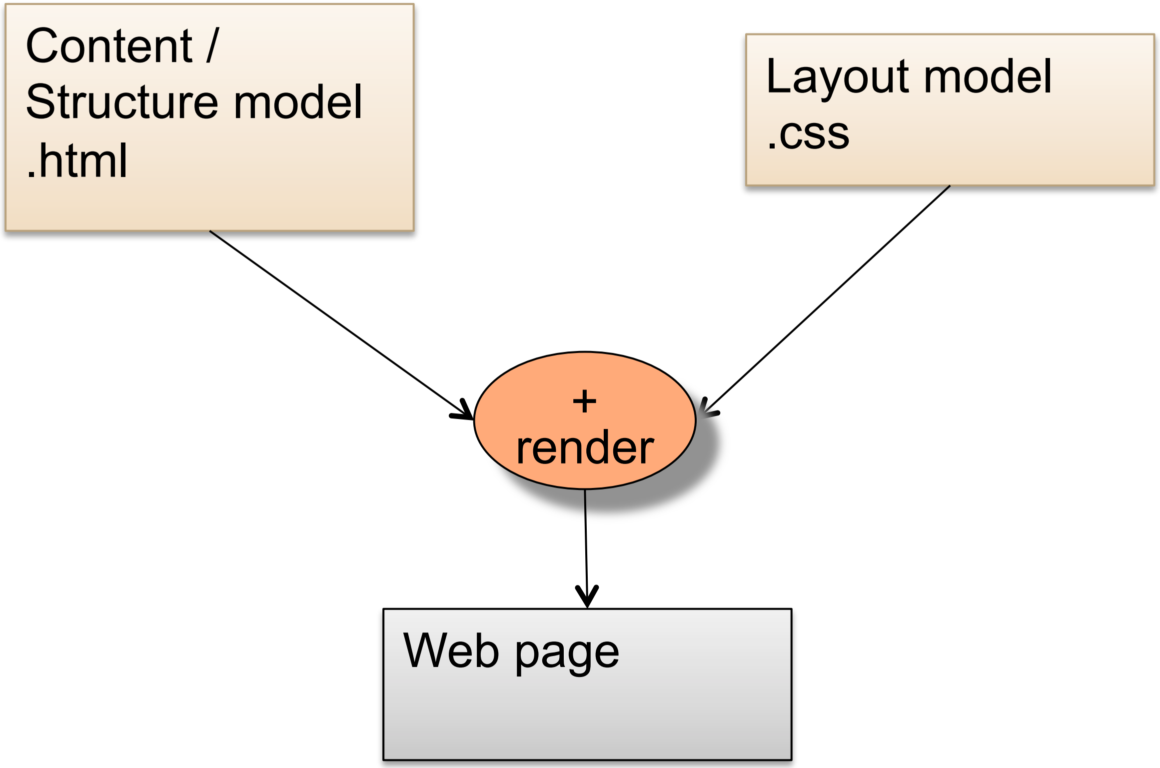
Merge vs. Extend: Symmetric vs. Asymmetric Composition

- 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: html+css

- From the beginning, SGML, XML and html separated structure from layout



Content / Structure model .html

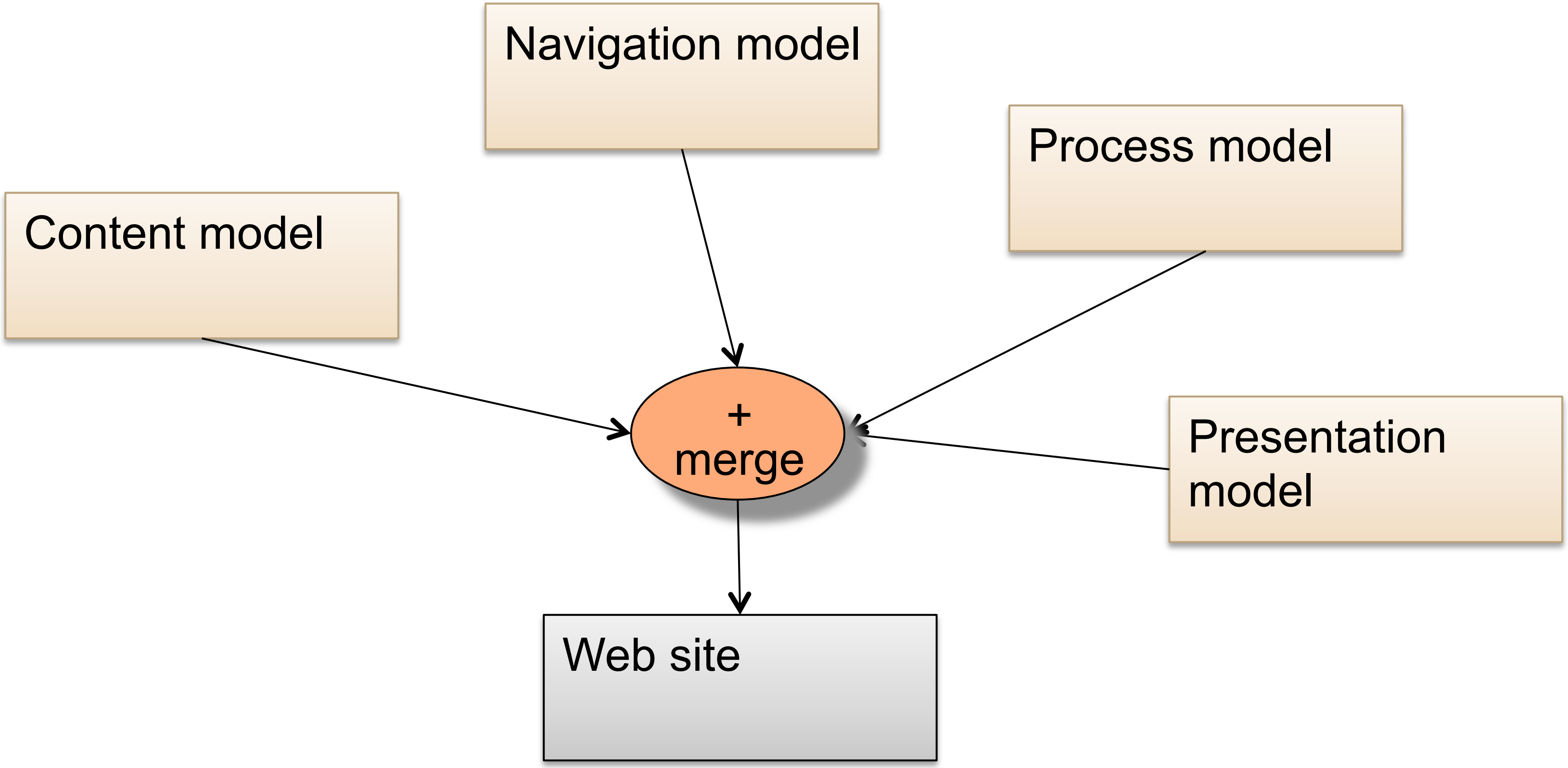
Layout model .css

+ render

Web page

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”

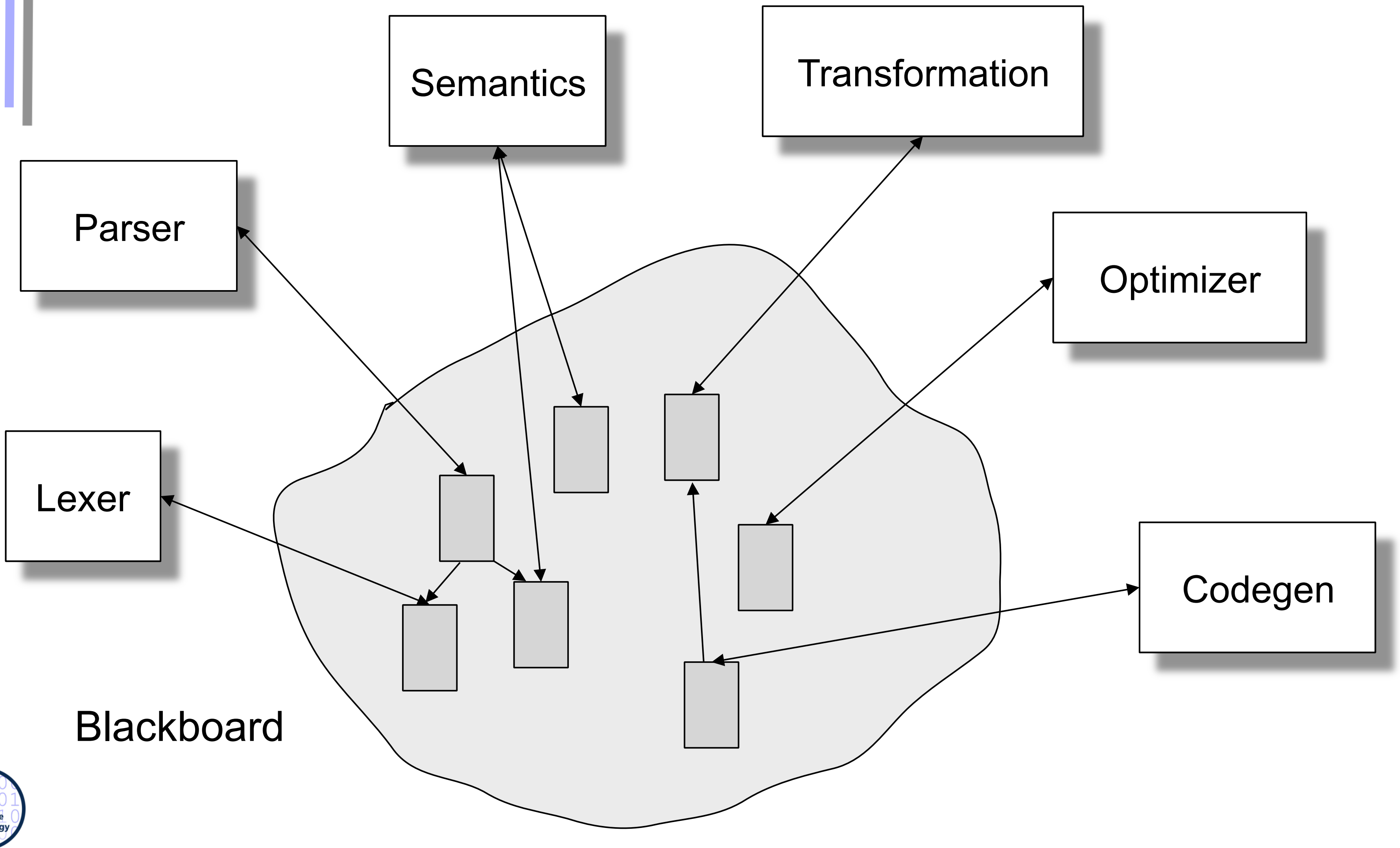


43.2. A Composition System based on Constructive Views: CoSy

Problem: Extensibility (here Compilers)

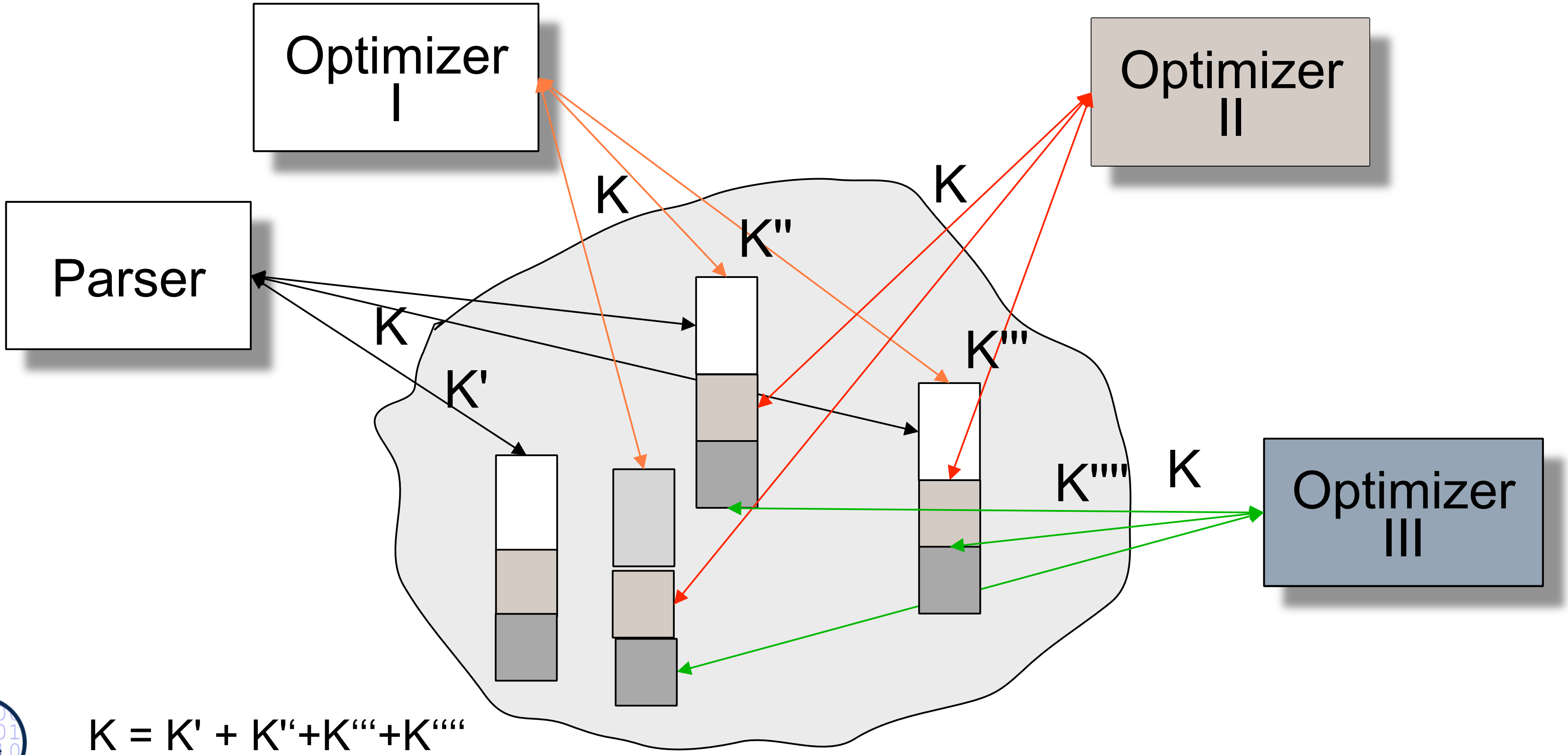
- **CoSy** is a modular component framework for compiler construction [Alt/Aßmann/vanSomeren94]
  - Built in 90-95 in Esprit Project COMPARE
  - Sucessfully marketed by ACE bV, Amsterdam
- Goal: *extensible*, easily configurable compilers
  - Extensions without changing other components
  - Plugging 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

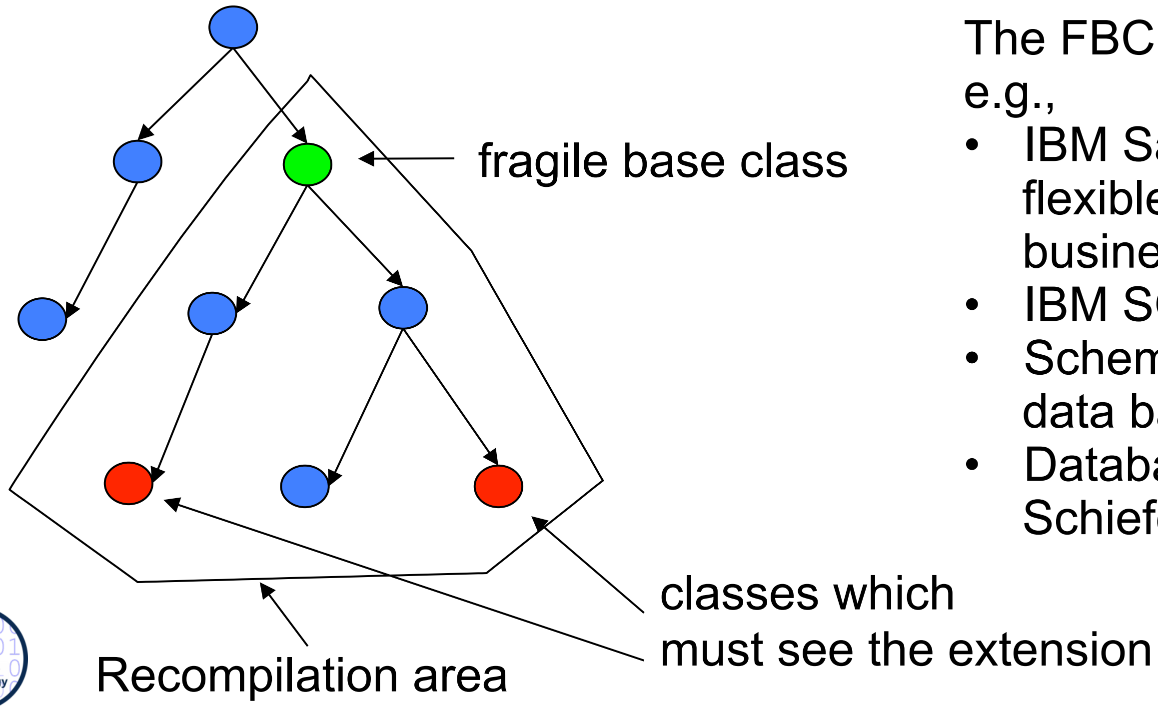
- Objects have to be allocated by the parser in base class format, but new components introduce new attributes into the base class



K = K' + K"+K"'+K""

Syntactic Fragile Base Class Problem in Object-Oriented Languages

- In unforeseen extension of a object-oriented 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*)

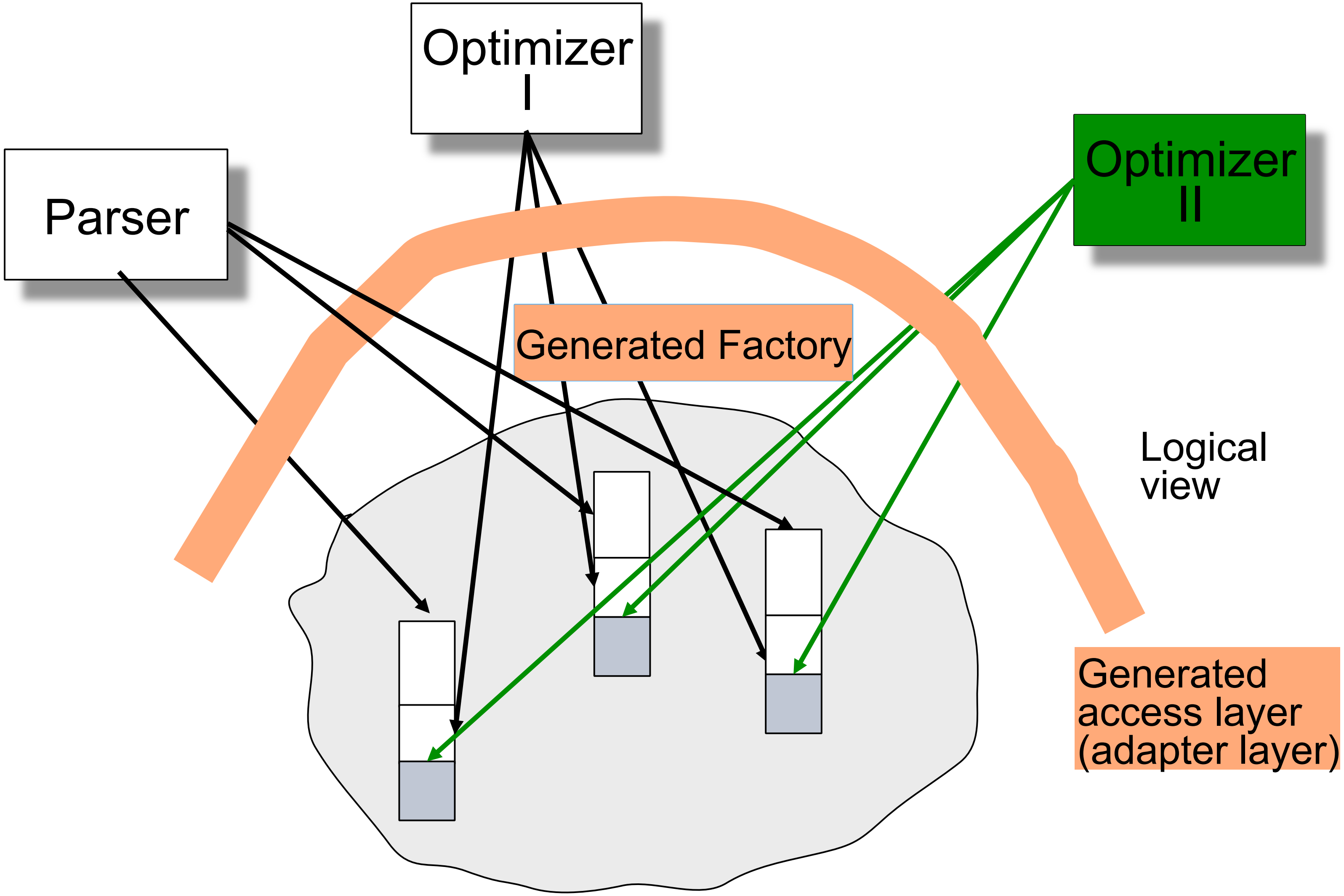


The FBCP problem was described in e.g.,

- 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

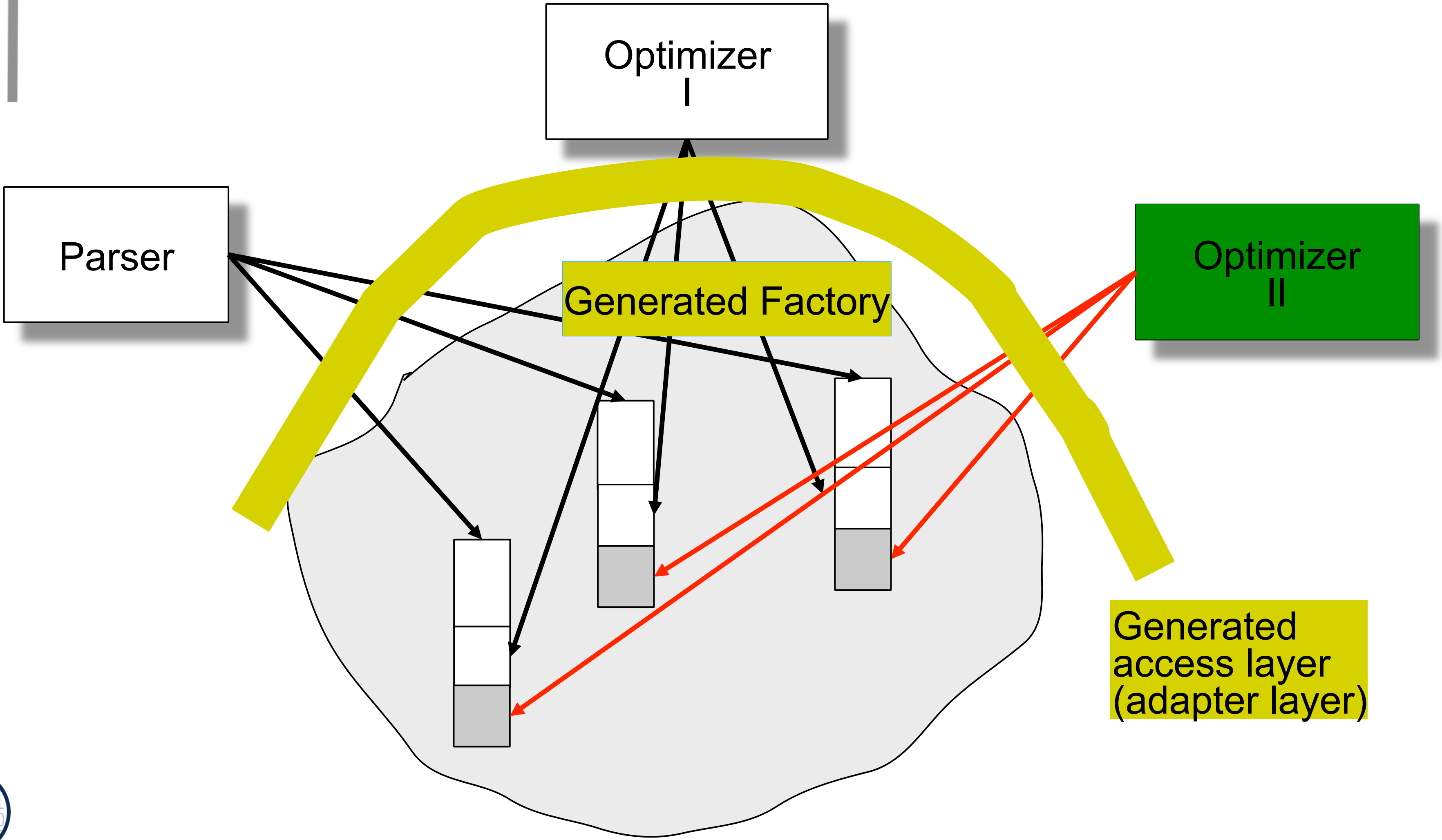
A CoSy Compiler is Extensible by Constructive Views

- Similar in IBM SOM



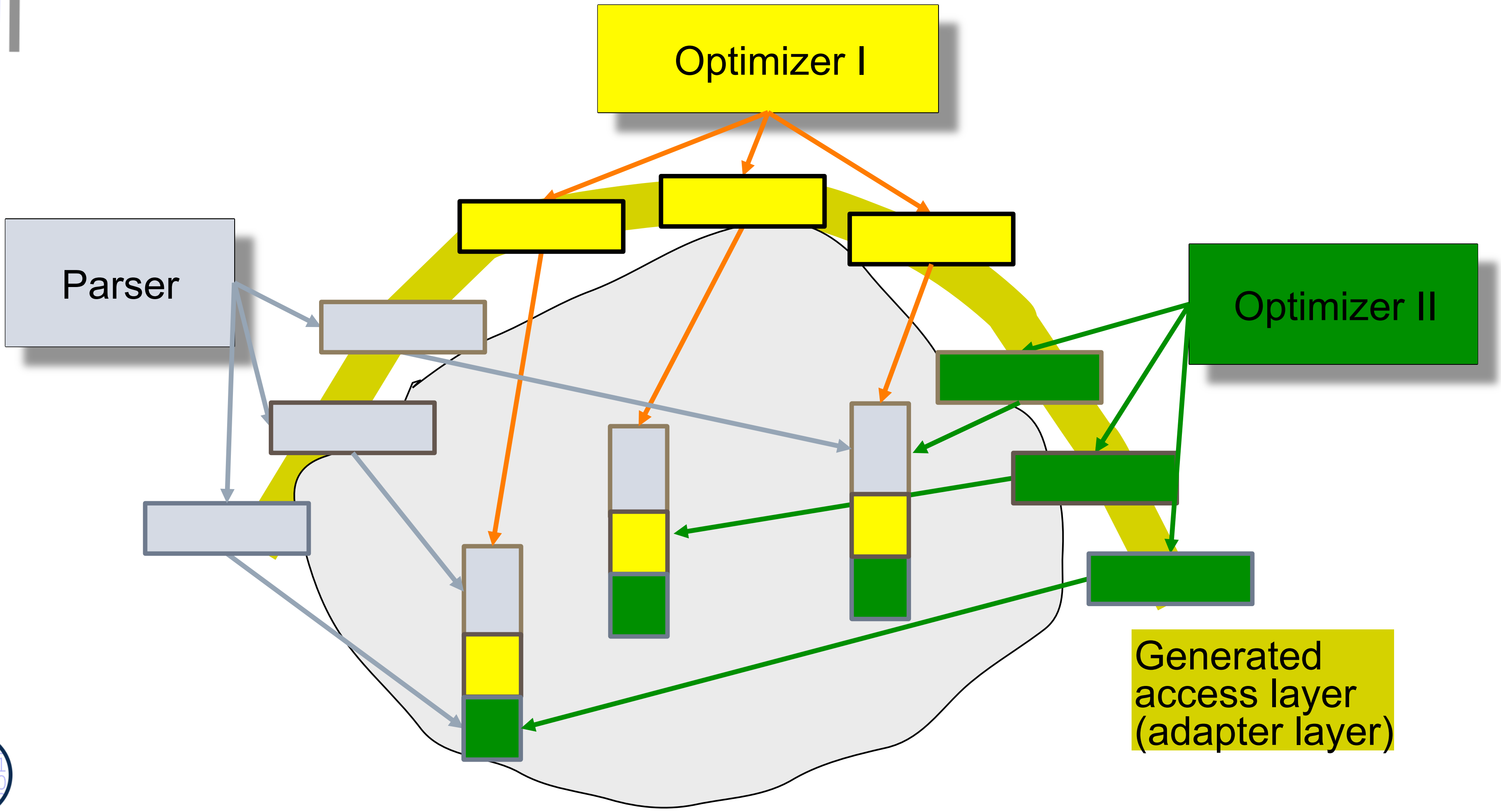
Extension with Constructive Views

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

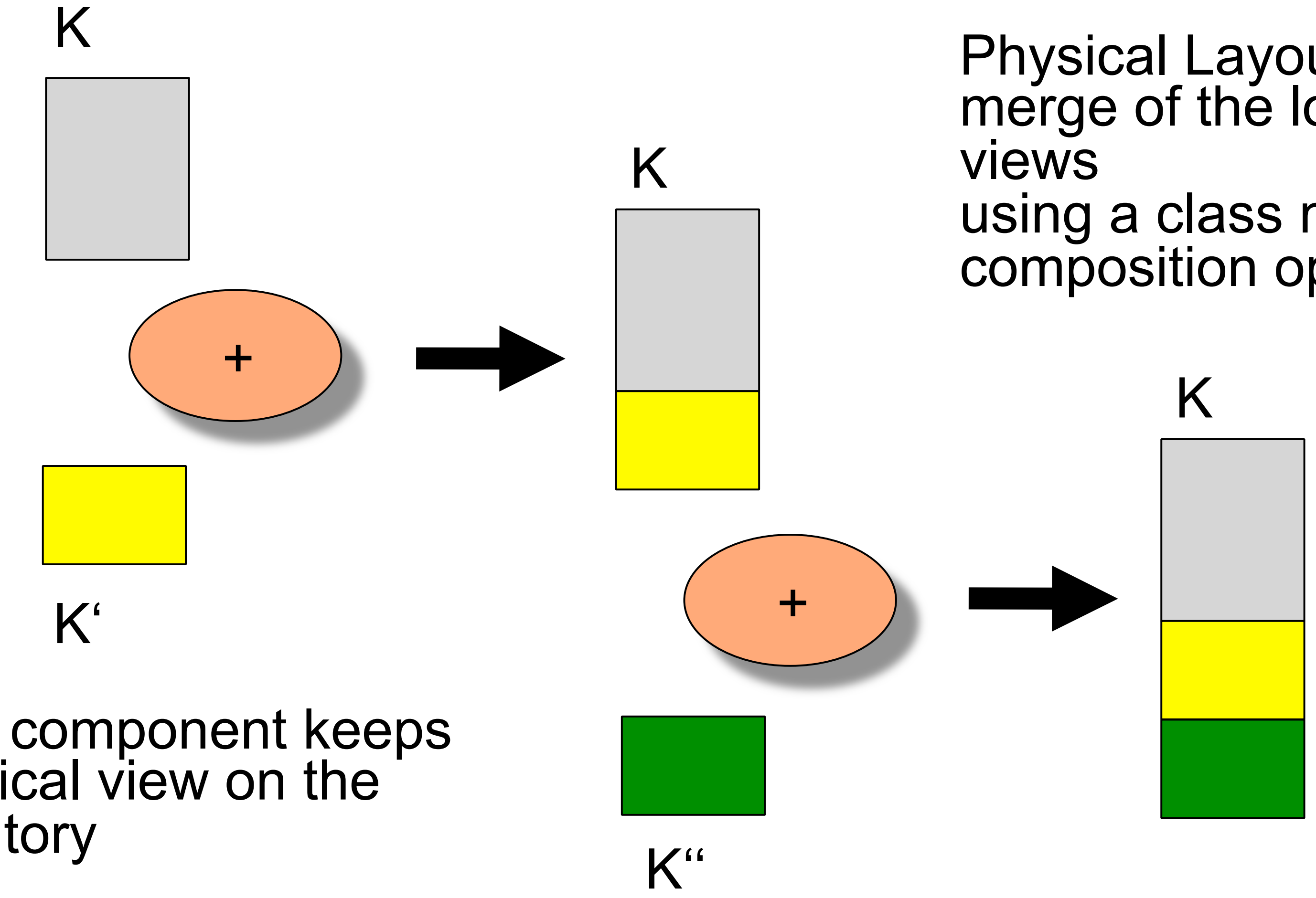


Extension with Constructive Views (Detail)

- Extension wraps all material classes in the repository by specific composition filters (decorators)
- The access layer is a decorator (filter layer)



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

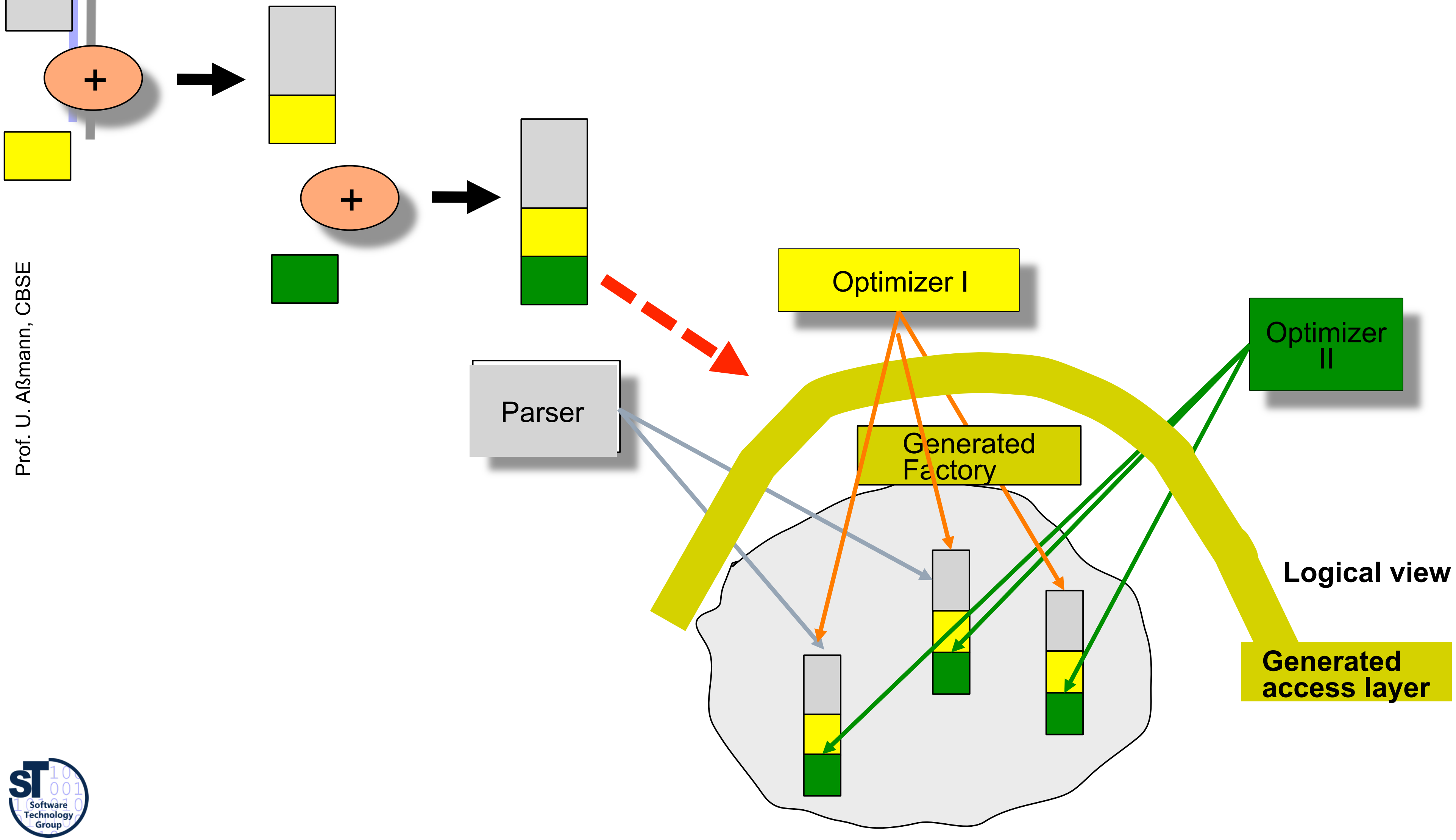


Physical Layout is a merge of the logical views
using a class merge composition operator

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 Role-Object Pattern 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, generated adapter 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?

43.3 Subject-Oriented Programming

A C++-based class calculus for view-based 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* consisting of
    - Operations (generic methods)
    - Classes with instance variables (members)
    - Mapping of classes and operations to each other
      - (class,operation) realization-relation: describes how to generate the methods of the real class from the compositions and the subjects
- **Composition technique:**
  - Assemble subjects by composition with *composition operators* (*mix rules,* composition rules)
  - By composition of the subjects the classes are completed step by step and the mapping of classes and operations is changed
  - The result of the composition is a C++ class system

A Subject is a View on a Class

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

```
// Subject PAYROLL defines a view on class Employee
Subject: PAYROLL {
   Operations: { print() }
   Classes: { Employee()
           with InstanceVariables: _emplName;
   }
   Mapping: {
        Class Employee, Operation print() implemented by
            &Employee::Print()
            // others..
   }
}
```

SOP for MDA and Refinement

Composition Operators of SOP (Mix Rules)

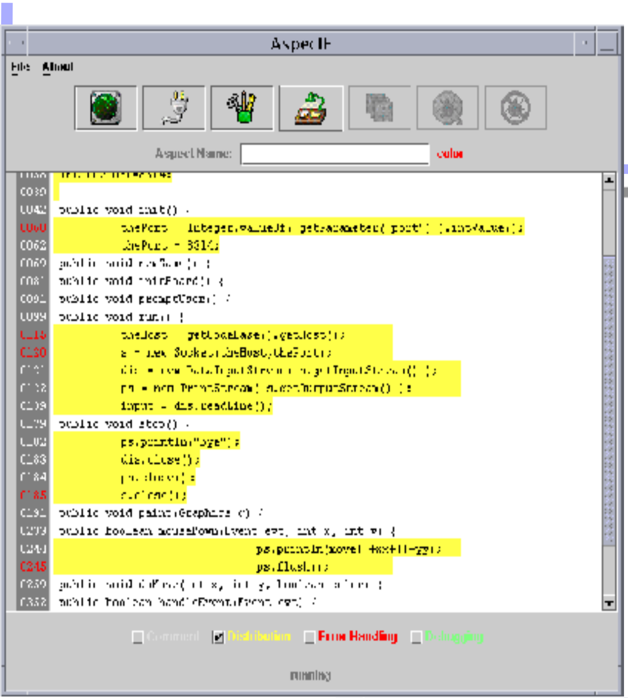
- **Correspondence operators**: declare equivalence of views of classes
  - *Equate*: equate method-implementations and method interfaces in subjects
  - *Correspond*: Introduce delegation between delegator and delegatee
- **Combination operators**
  - *Replace*: override of features of all classes of a subject
  - *Join*: linking of parts of subjects
- **Composed composition operators**
  - *Merge := (Join; Equate):* After Join equate implementations and interfaces
  - *Override:* override features in subject

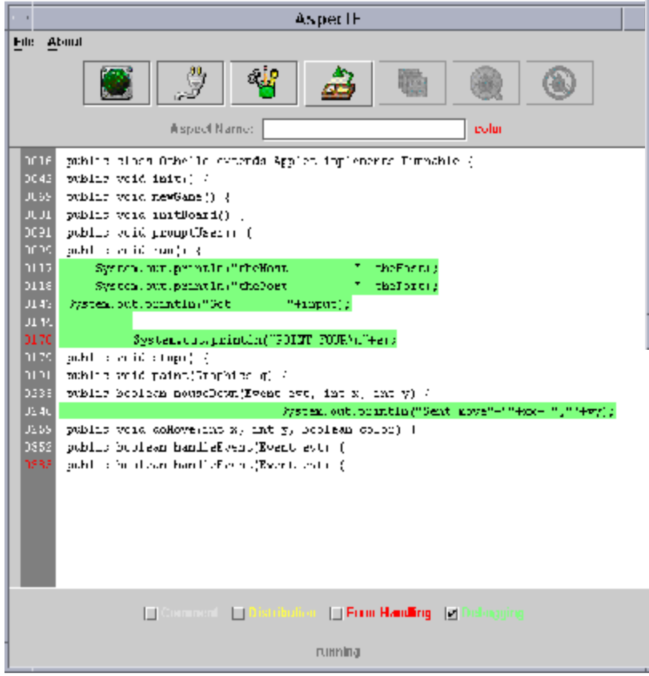
Evaluation of SOP as Composition System

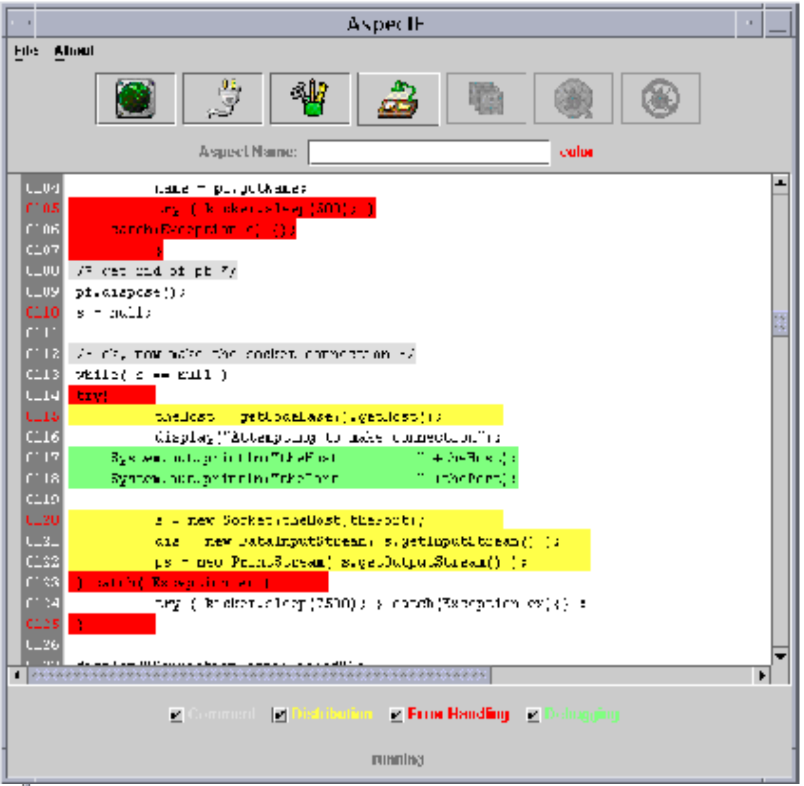
- Advantage
  - C++ applications become simply extensible with new views that can be merged into existing ones by the extension operators
  - Stakeholder-specific views
  - Design view
  - Implementation view
  - Model-Driven Architecture (MDA) is easily possible:
    - Platform-independent view
    - Platform-specific views
- Disadvantage:
  - No real composition language: the set of composition operators is fixed!
  - No control flow on compositions

43.4 Hyperspaces

Color Coding of Concerns







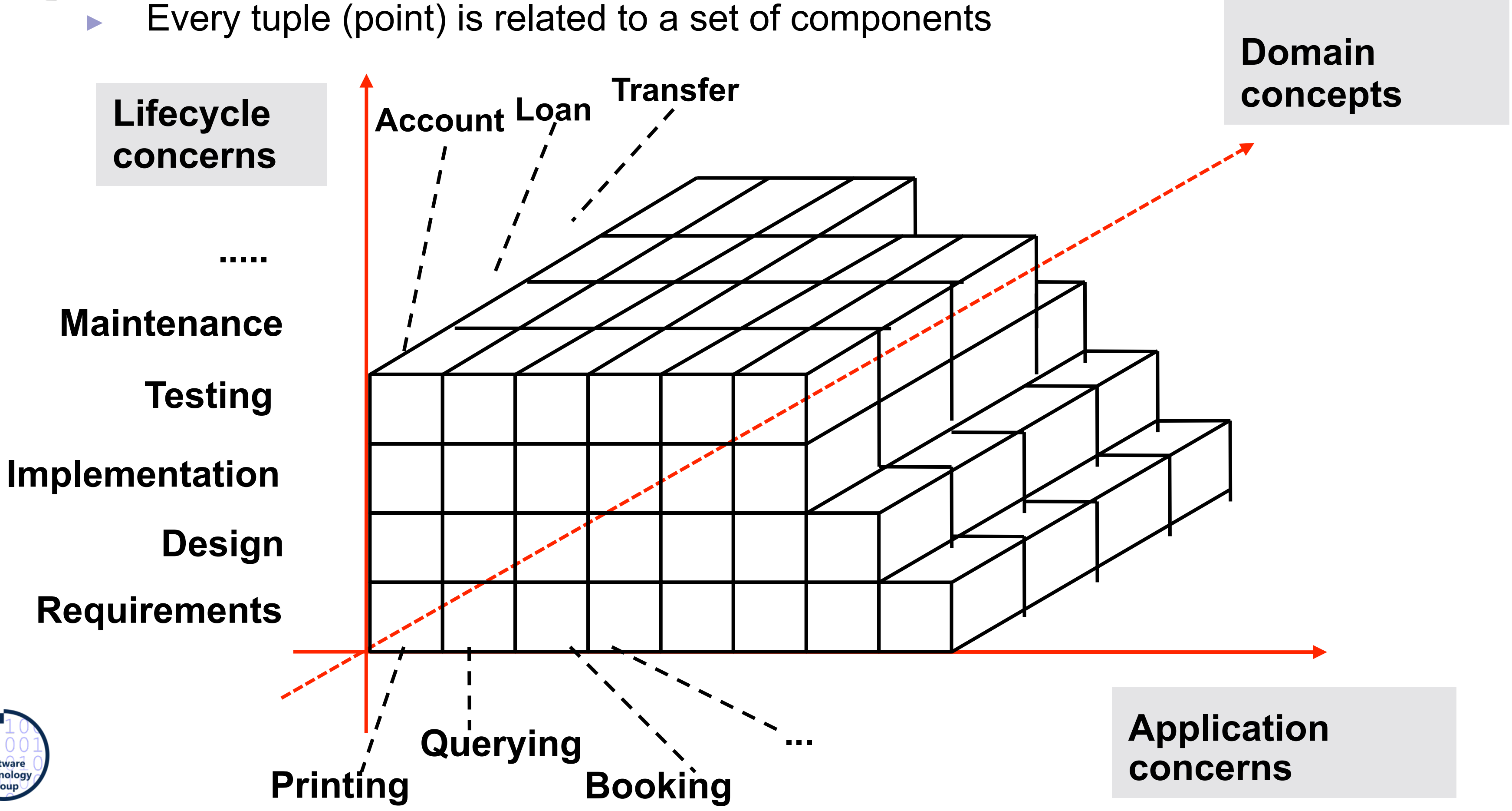
Fragments can be colored with regard to a certain concern (concern mapping) [Panas, Andersson, Aßmann: The Editing Aspect of Aspects SEA 2002]

Hyperspaces

- Hyperspaces generalize SOP. Instead of classes, hyperspaces work on sets of *fragments* (aka *units)*, i.e, fragment groups
  - Open definitions for classes, methods, and all kinds of other definitions
  - A hyperspace represents an environment for *dimensional development*, a specific form of view-based development
- A **hyperspace** is a multi-dimensional space over concerns related to components
  - Each axis (dimension) is a dimension of software concerns
    - Color dimension
    - Texture dimension
    - Striping dimension, etc
- Each point on the axis is a ***concern,*** *expressed by* ***tags***
  - A *concern* groups (tags) semantically related fragments to fragment groups
  - Each concern can be seen as a
    - Color in the color dimension (blue, green, yellow…)
    - Texture in the texture dimension (sanded, squared,..)
    - Striping in the striping dimension (vertical stripes, horizontal stripes,..)

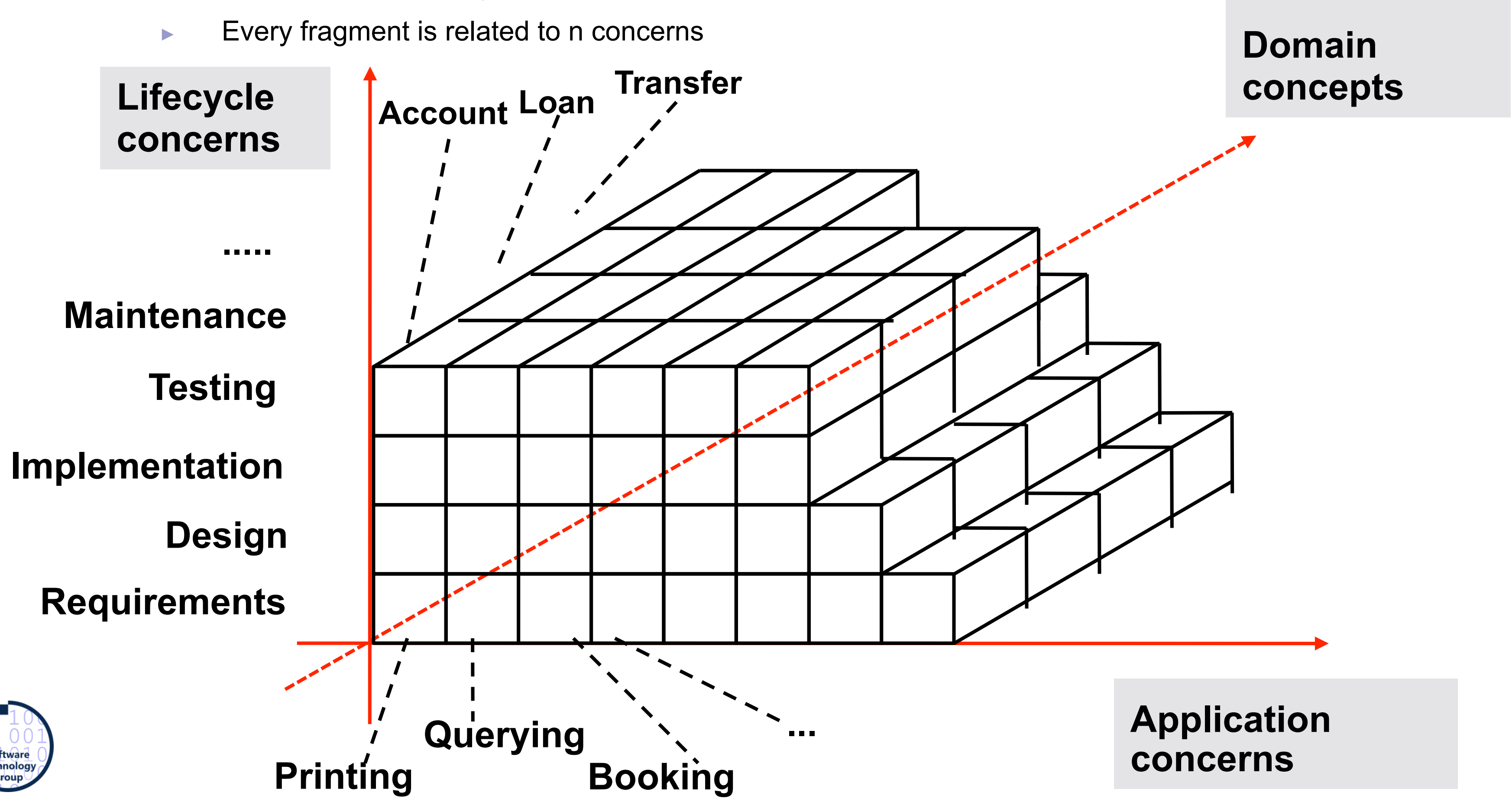
The Concern Matrix of the Hyperspace Describes the Concern Space

- Concerns are grouped into an ***n-dimensional space***, arranged in ***concern dimensions*** (ex.: @Lifecycle.design, @Application.querying, @Domain.Transfer)
- A point of the space forms a ***concern tuple*** (@c_1, ..,@ c_n)
- Every component is related to a tuple of n concerns
- Every tuple (point) is related to a set of components



Fragment Hyperspaces

- In a **fragment hyperspace,** the components are program, model, documentation, test data fragments
- These fragments are grouped into an *n-dimensional space of concerns*, arranged in *concern dimensions*, with points
  - related to a set of fragments
  - Every fragment is related to n concerns



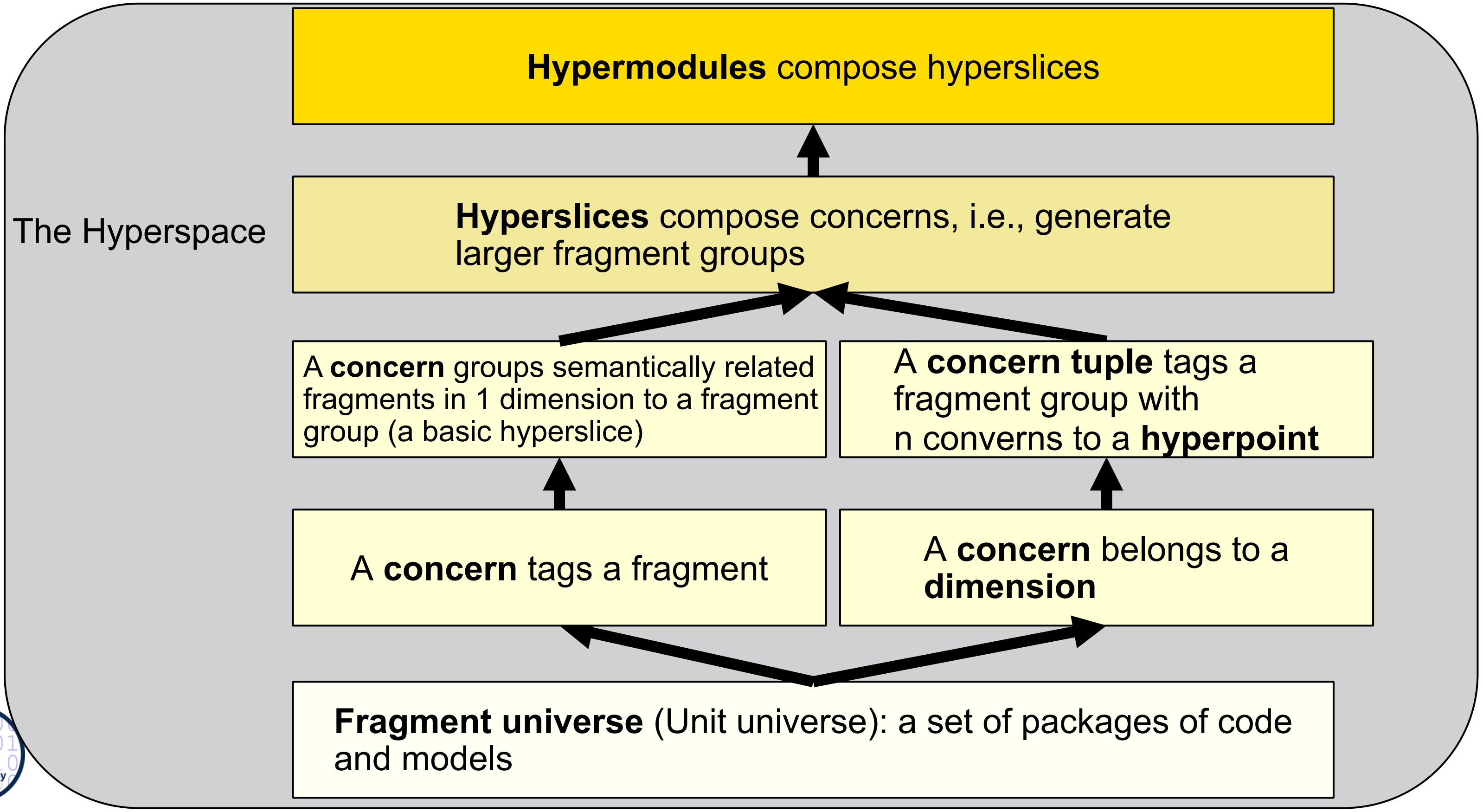
Lifecycle concerns

Domain concepts

Application concerns

The Hyperspace, a Fragment Space

- A **hyperslice** is a view (slice) of a system, based on a selection of concerns
- A **hyperpoint** is the view (set of fragments) related to a n-tuple of concerns
- A **basic hyperslice** is a view based on *one* concern of some dimension
- Composition operation: *unify (merge-by-name) of fragment groups* by merging of concerns and hyperslices



The Hyperspace

Hypermodules compose hyperslices

Hyperslices compose concerns, i.e., generate larger fragment groups

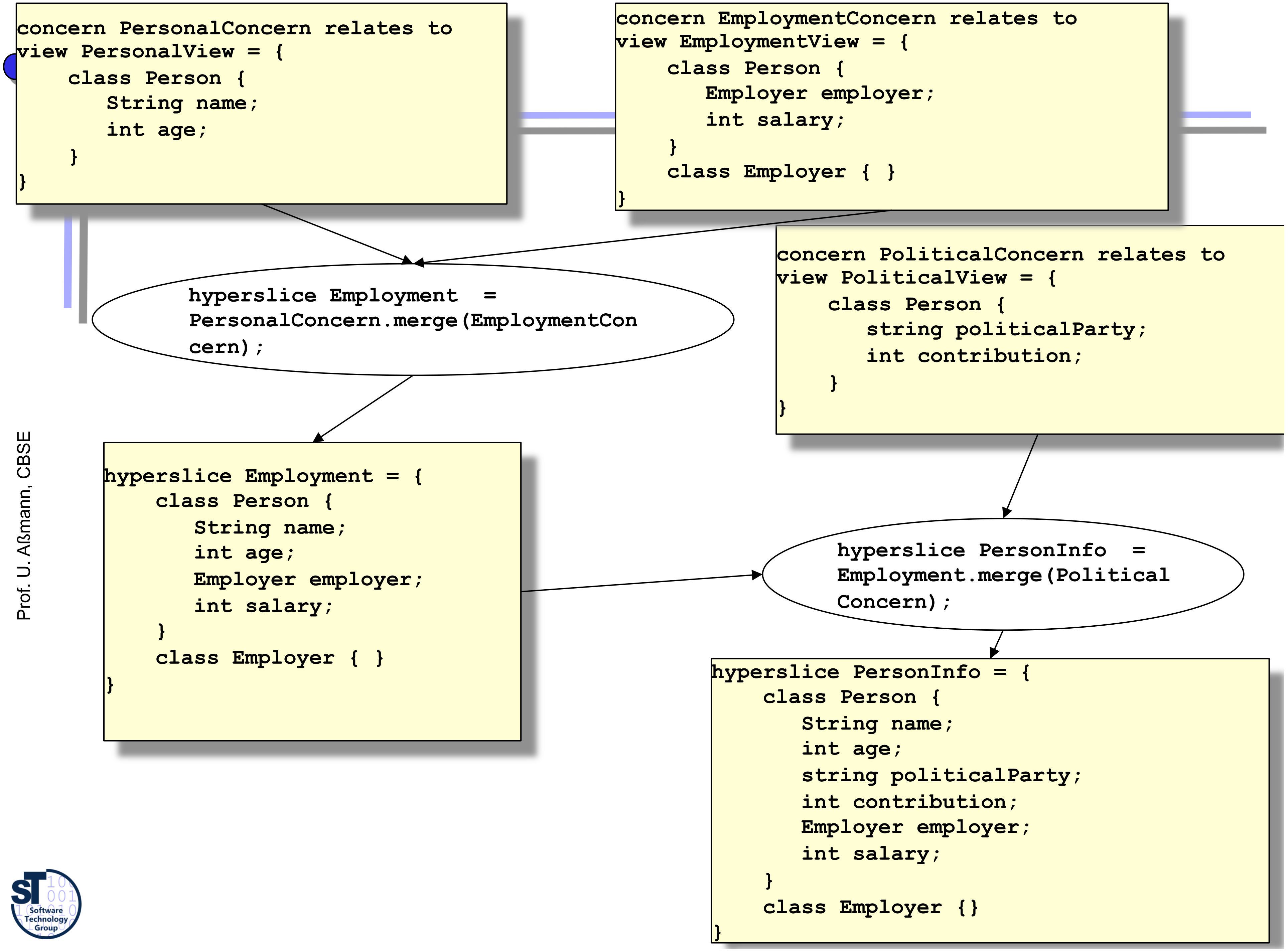
A **concern** groups semantically related fragments in 1 dimension to a fragment group (a basic hyperslice)

A **concern tuple** tags a fragment group with n converns to a **hyperpoint**

A **concern** tags a fragment

A **concern** belongs to a **dimension**

Fragment universe (Unit universe): a set of packages of code and models



```
concern PersonalConcern relates to
view PersonalView = {
    class Person {
        String name;
        int age;
    }
}
```

```
concern EmploymentConcern relates to
view EmploymentView = {
    class Person {
        Employer employer;
        int salary;
    }
    class Employer { }
}
```

```
hyperslice Employment  =
PersonalConcern.merge(EmploymentCon
cern);
```

```
concern PoliticalConcern relates to
view PoliticalView = {
    class Person {
        string politicalParty;
        int contribution;
    }
}
```

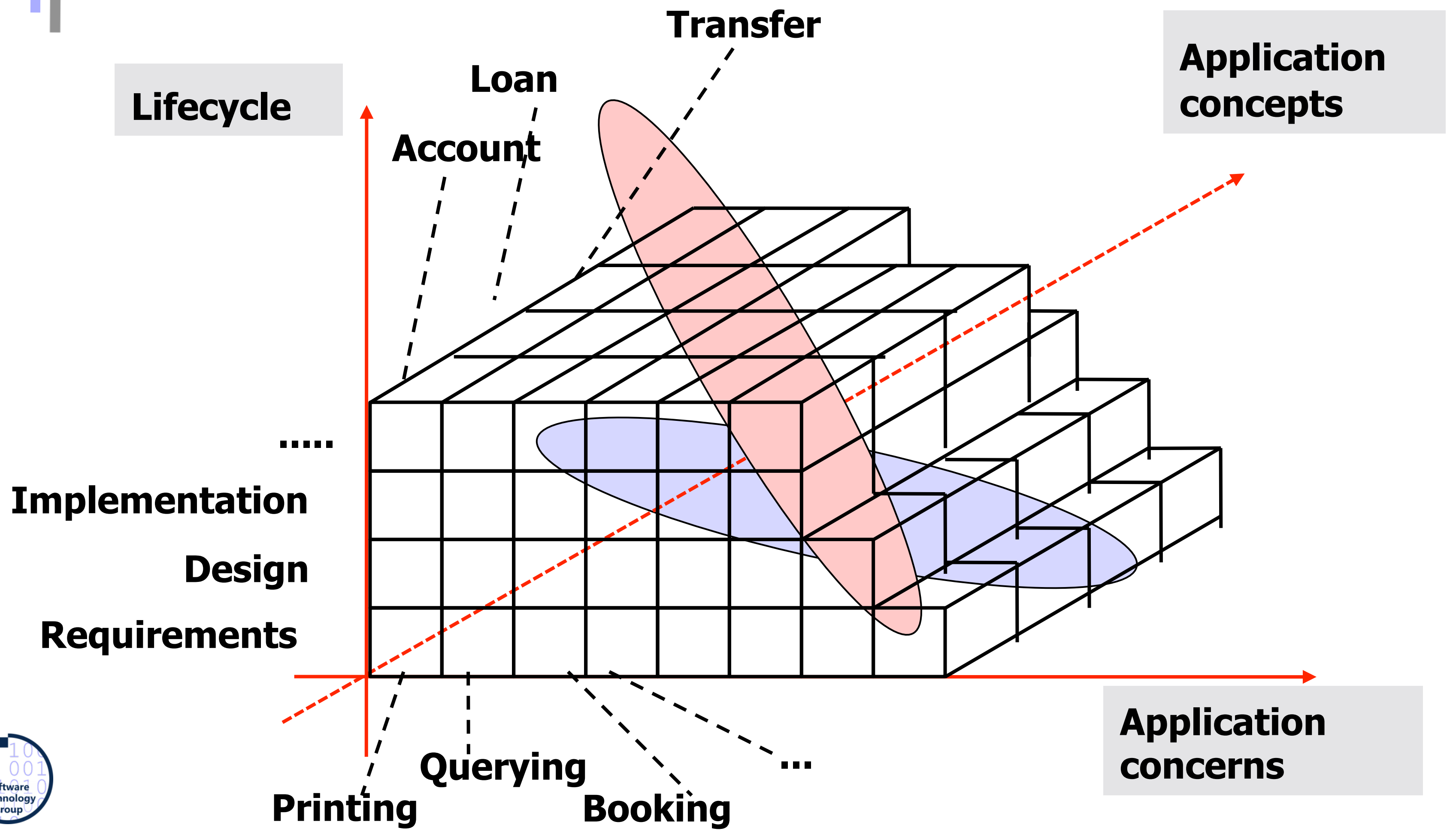
```
hyperslice Employment = {
    class Person {
        String name;
        int age;
        Employer employer;
        int salary;
    }
    class Employer { }
}
```

```
hyperslice PersonInfo  =
Employment.merge(Political
Concern);
```

```
hyperslice PersonInfo = {
    class Person {
        String name;
        int age;
        string politicalParty;
        int contribution;
        Employer employer;
        int salary;
    }
    class Employer {}
}
```

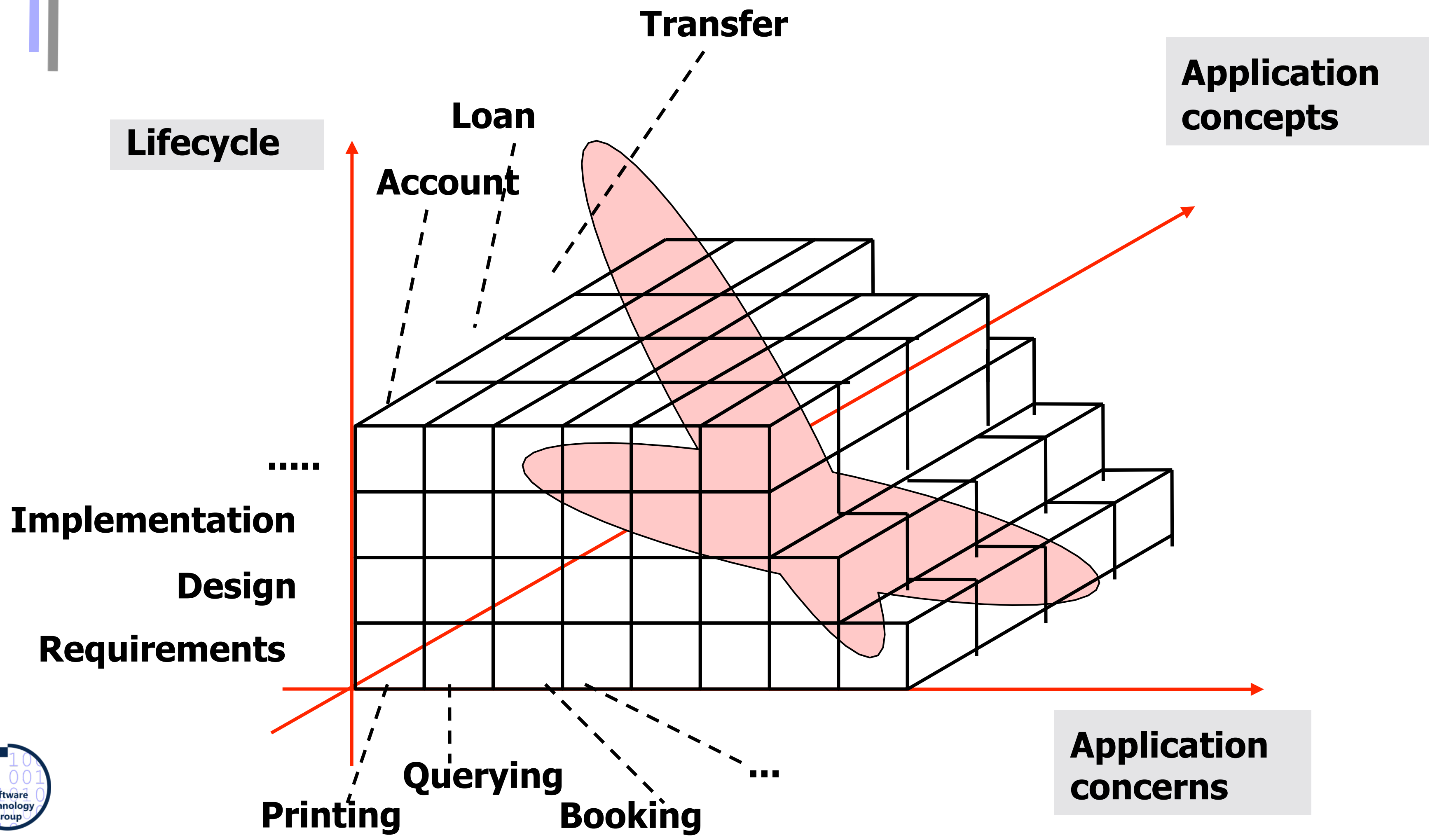
Hyperslices are Composed out of Concerns

- 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



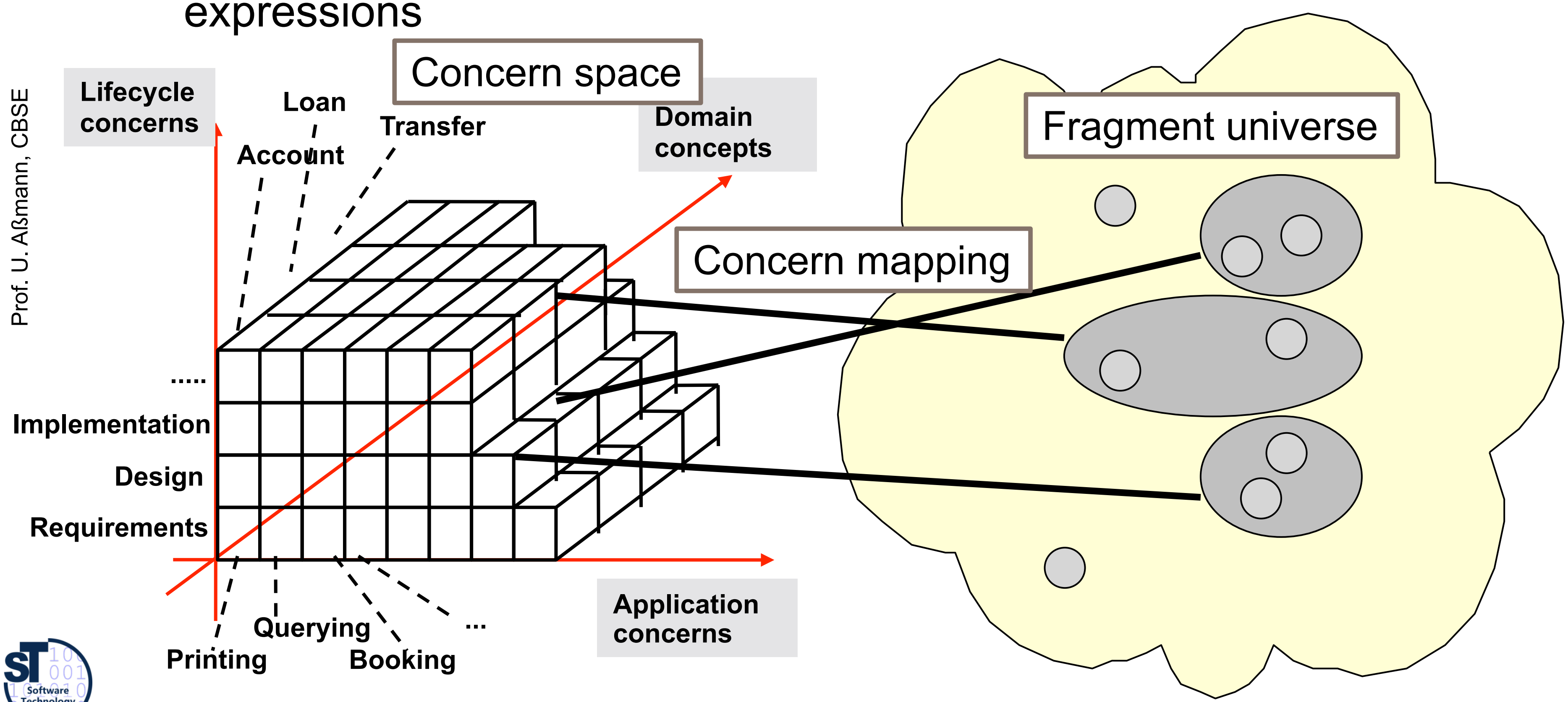
Hypermodules are Named Compositions of Hyperslices

- Hypermodules are deployable products



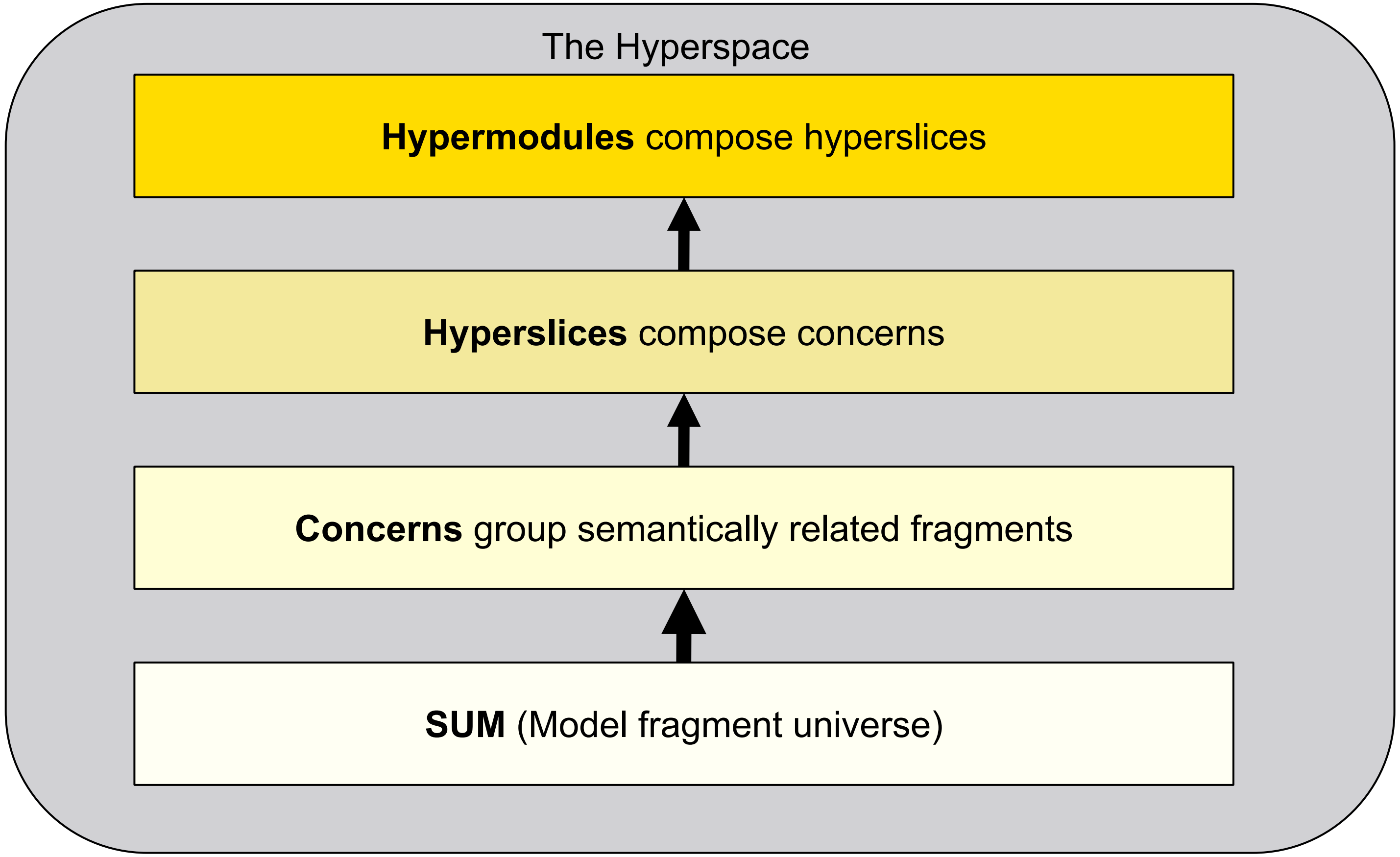
The Concern Matrix maps Concerns to the Sets of Fragments

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



OSM as Specific Hyperspaces: The Single Underlying Model (SUM)

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



The Hyperspace

Hypermodules compose hyperslices

Hyperslices compose concerns

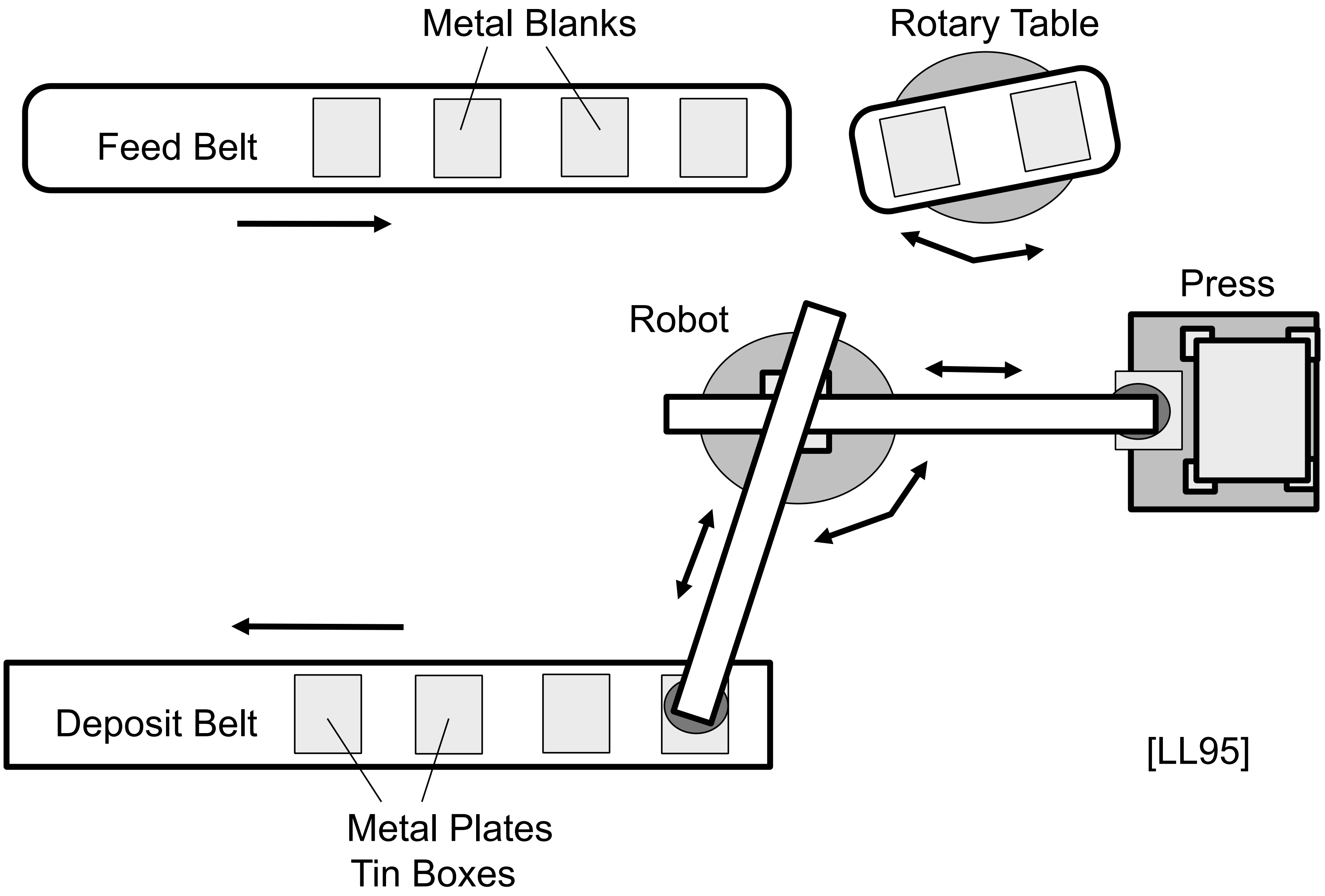
Concerns group semantically related fragments

SUM (Model fragment universe)

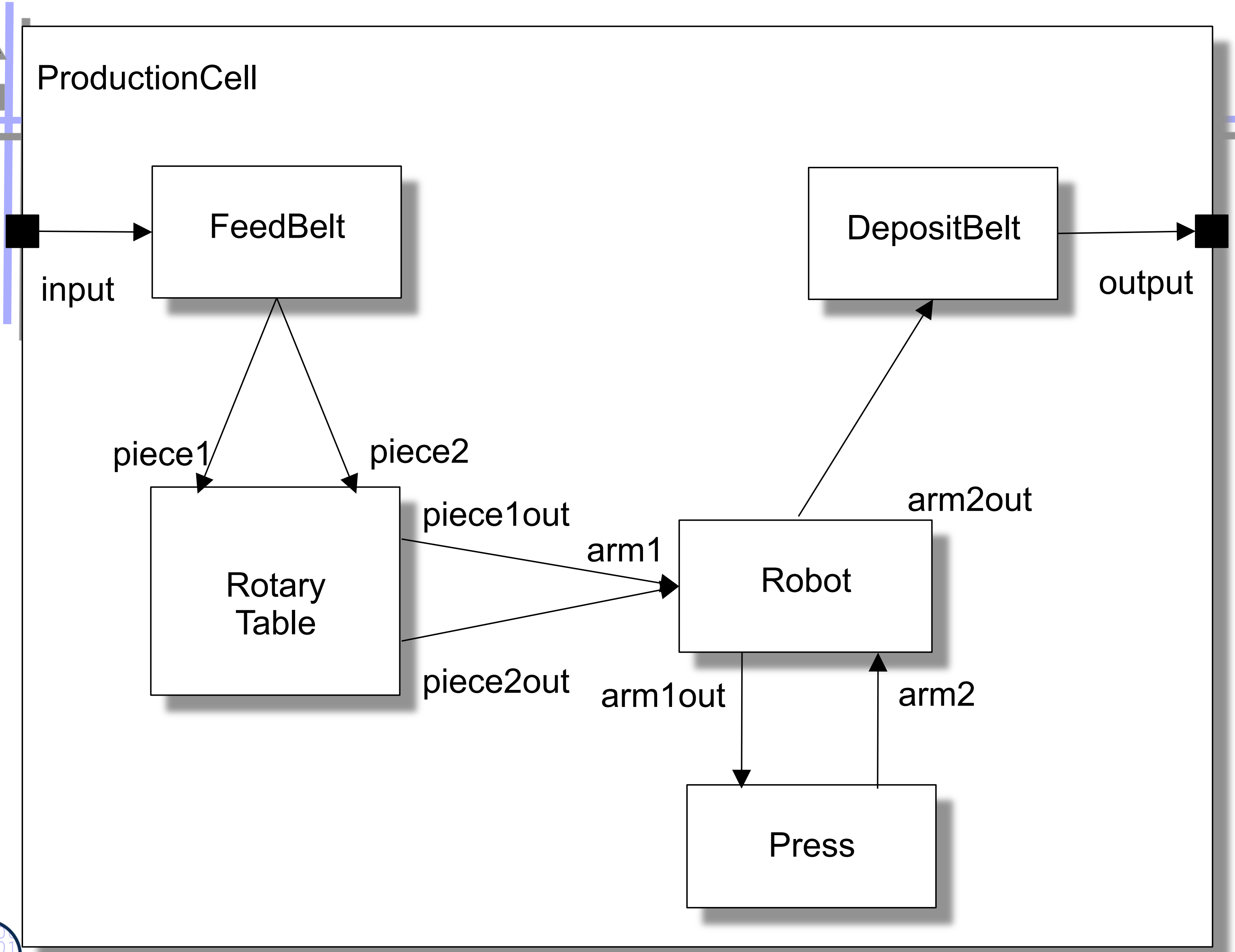
43.4.1 Hyperspace Programming

Example

The Production Cell Case Study



[LL95]



ProductionCell

FeedBelt

DepositBelt

input

output

piece1

piece2

piece1out

arm1

arm2out

Robot

Rotary Table

piece2out

arm1out

arm2

Press

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" all
// classes, methods, fragments of some Java packages
hyperspace ProductionCell = {
   composable class passiveDevices.*
   composable class activeDevices.*
   composable class tracing.*
   composable class visualization.*
   composable class contracts.*
}
```

Composition Technology – Concern Mapping

- For package **`passiveDevices`**, we define the following *concern mapping* between concerns and Java fragments
  - Tagging (embedded or offline): a name is related to a tag
  - 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>`**

Default mapping for the entire package

```
// 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
```

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`**

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

Default mapping for the entire package

Specific mappings

Mapping

Composition Technology – Concern Mapping

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

Default mapping for the entire package

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

Specific mappings

```
package visualization:  @Visualization.Graphics
        class Vectorgraphics: @Visualization.VectorGraphics
        class BaseGraphics: @Visualization.VectorGraphics,
                            @Visualization.PixelGraphics
```

Specific mappings

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 = {
used hyperslices: @Feature.Transfer, @Feature.WorkPieces,
@Logging.Tracing
  composition 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.

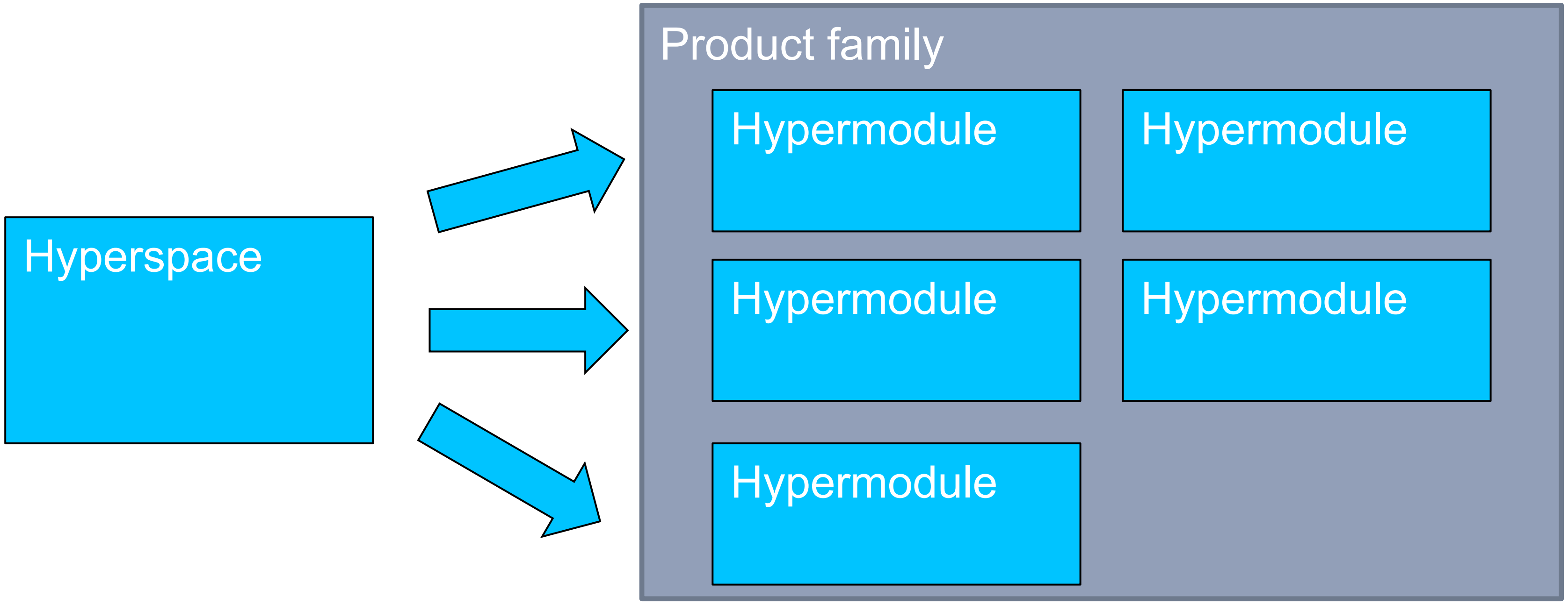
```
hypermodule ProductionCell = {
   hyperslices: @Feature.Transfer, @Feature.WorkPieces,
        @Feature.ActiveDeviceBehavior
   composition relationships: overrideByName
}
```

- and this is a hypermodule with visualization:

```
hypermodule VisualizingProductionCell = {
   hyperslices: @Feature.Transfer, @Feature.WorkPieces,
     @Feature.ActiveDeviceBehavior, @Visualization.VectorGraphics
   composition 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

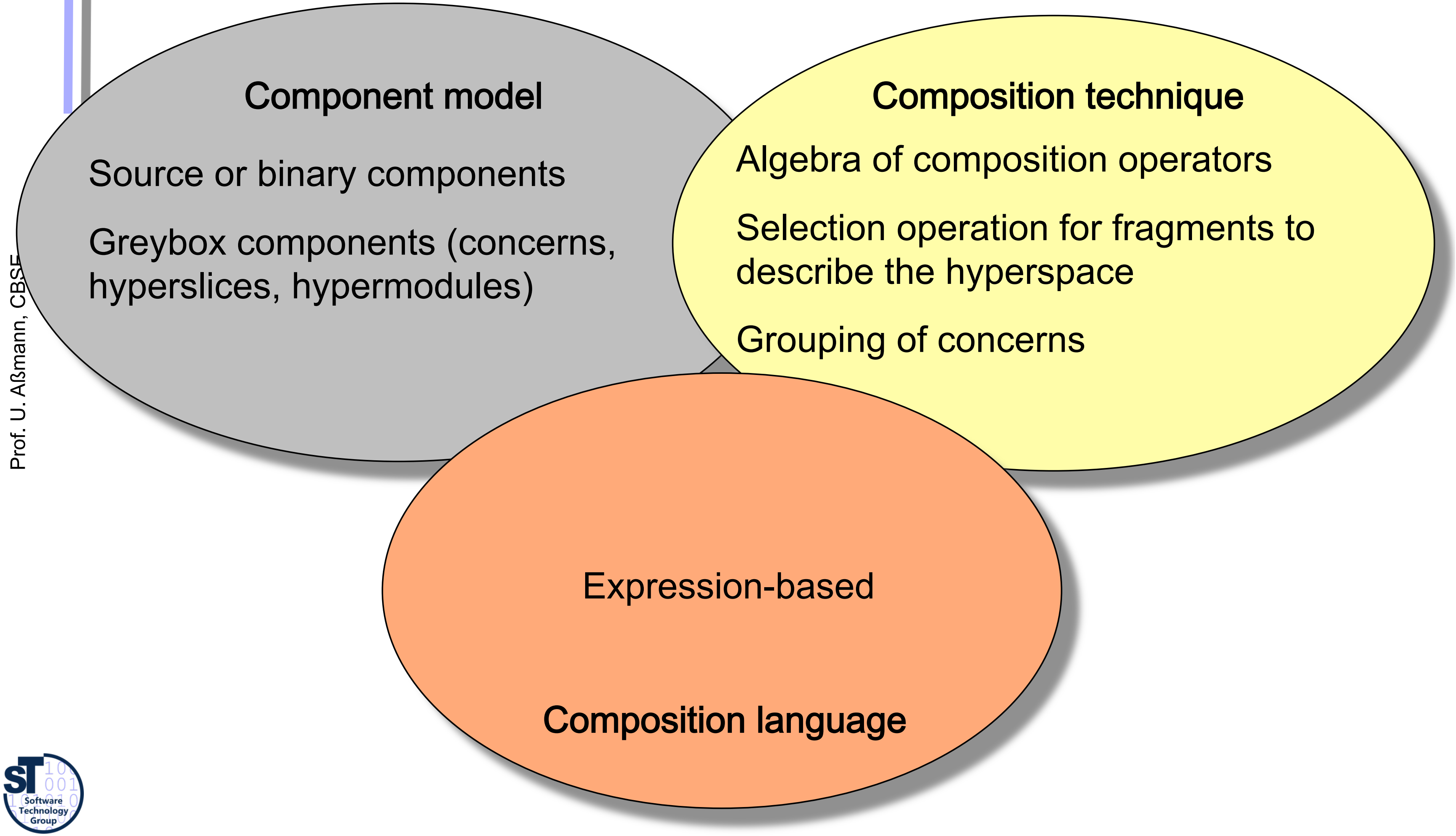
Universal extensibility: A language is called *universally extensible*, if it provides extensibility for every collection-like language construct.

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.

43.5 Evaluation: Hyperspaces as Composition System



Component model

Source or binary components

Greybox components (concerns, hyperslices, hypermodules)

Composition technique

Algebra of composition operators

Selection operation for fragments to describe the hyperspace

Grouping of concerns

Expression-based

Composition language

The End - Appendix

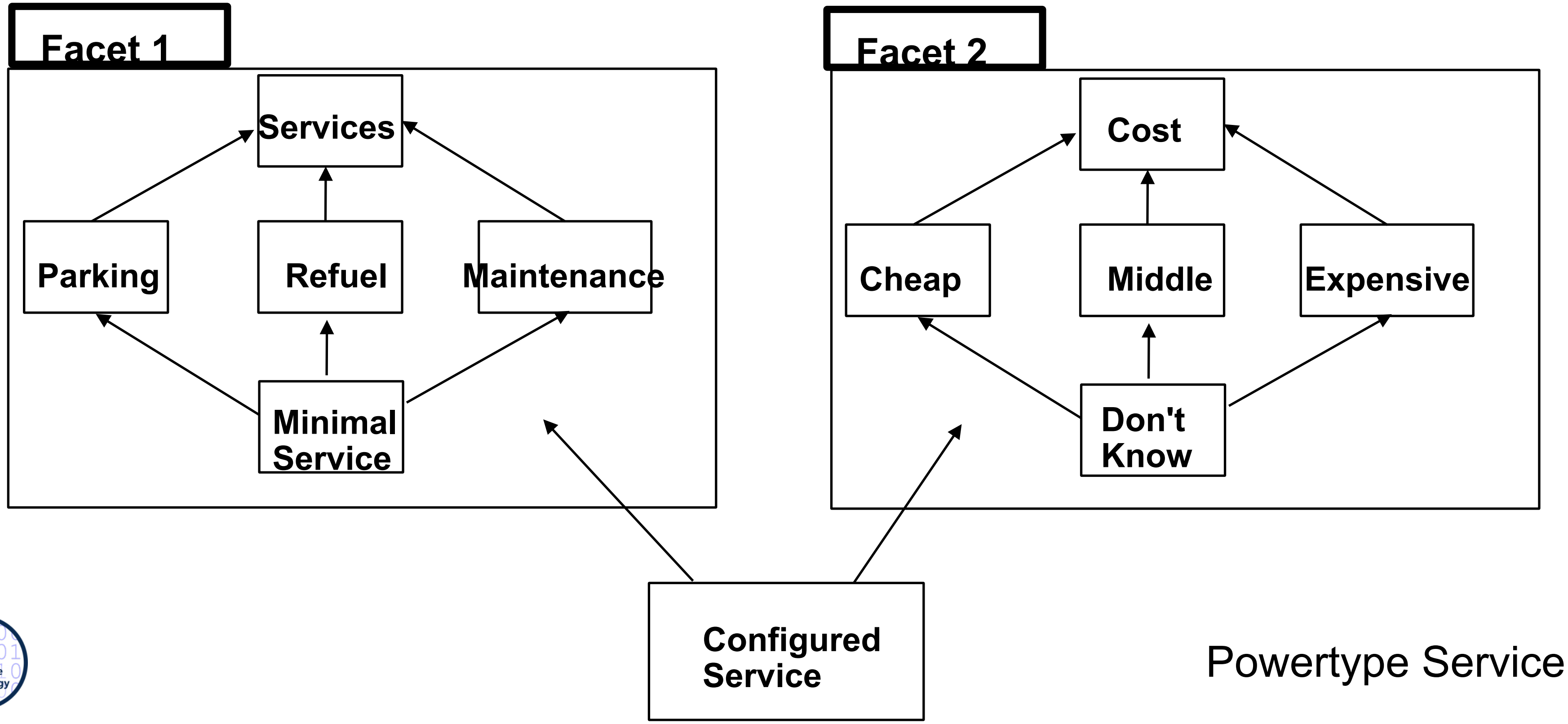
- How do constructive and projective views differ?
- Explain the difference of the merge operator and the extend operator.
  - In LambdaN calculus, is there any difference of merge and extend?
- What happens, if the base language is not functional, i.e., not free of side effects?
- How do you realize views with mixin-based inheritance (GenVoca pattern or Mixin Layer pattern)?

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



Powertype Service

Side Remark: The Facet Matrix Describes Objects Dimensionally