30. Architecture Systems

Lecturer: Dr. Sebastian Götz

Prof. Dr. Uwe Aßmann

Technische Universität Dresden

Institut für Software- und Multimediatechnik

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

8. Mai 2017

1. Separation of Concerns
2. Concepts of an ADL
3. Examples of ADL
4. Appendix:
    1. Architecture Specification in UML
    2. Refinement of Connectors in MDSD

Obligatory Literature

- E. W. Dijkstra. EWD 447: On the role of scientific thought. Selected Writings on Computing: A Personal Perspective, pages 60–66, 1982. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html
- D. Garlan and M. Shaw. An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company, 1993, Ed. V. Ambriola and G. Tortora, S. 1-40. Nice introductory article. http://www-2.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html
- Shaw, M. and Clements, P.C. A Field Guide to Boxology. Preliminary Classification of Architectural Styles for Software Systems. CMU April 1996. http://www.cs.cmu.edu/~Vit/paper_abstracts/Boxology.html
- C. Hofmeister, R. L. Nord, D. Soni. Describing Software Architecture with UML. In P. Donohoe, editor, Proceedings of Working IFIP Conference on Software Architecture, pages 145--160. Kluwer Academic Publishers, February 1999. http://link.springer.com/chapter/10.1007/978-0-387-35563-4_9

Literature

- Shaw, M., Garlan, D. Software Architecture – Perspectives for an Emerging Discipline. Prentice-Hall,1996. Nice Introduction.
- Clements, Paul C. A Survey of Architecture Description Languages. Int. Workshop on Software Specification and Design, 1996.
- C. Hofmeister, R. Nord, D. Soni. Applied Software Architecture. Addision-Wesley, 2000. Very nice book on architectural elements in UML.
- Martin Alt. On Parallel Compilation. PhD Dissertation, Universität Saarbrücken, Feb. 1997. (CoSy prototype)
- ACE b.V. Amsterdam. CoSy Manuals.
  - http://www.ace.nl/compiler/cosy
- Overview of EAST-ADL
  - http://www.maenad.eu/public_pw/conceptpresentations/EAST-ADL_WhitePaper_M2.1.10.pdf

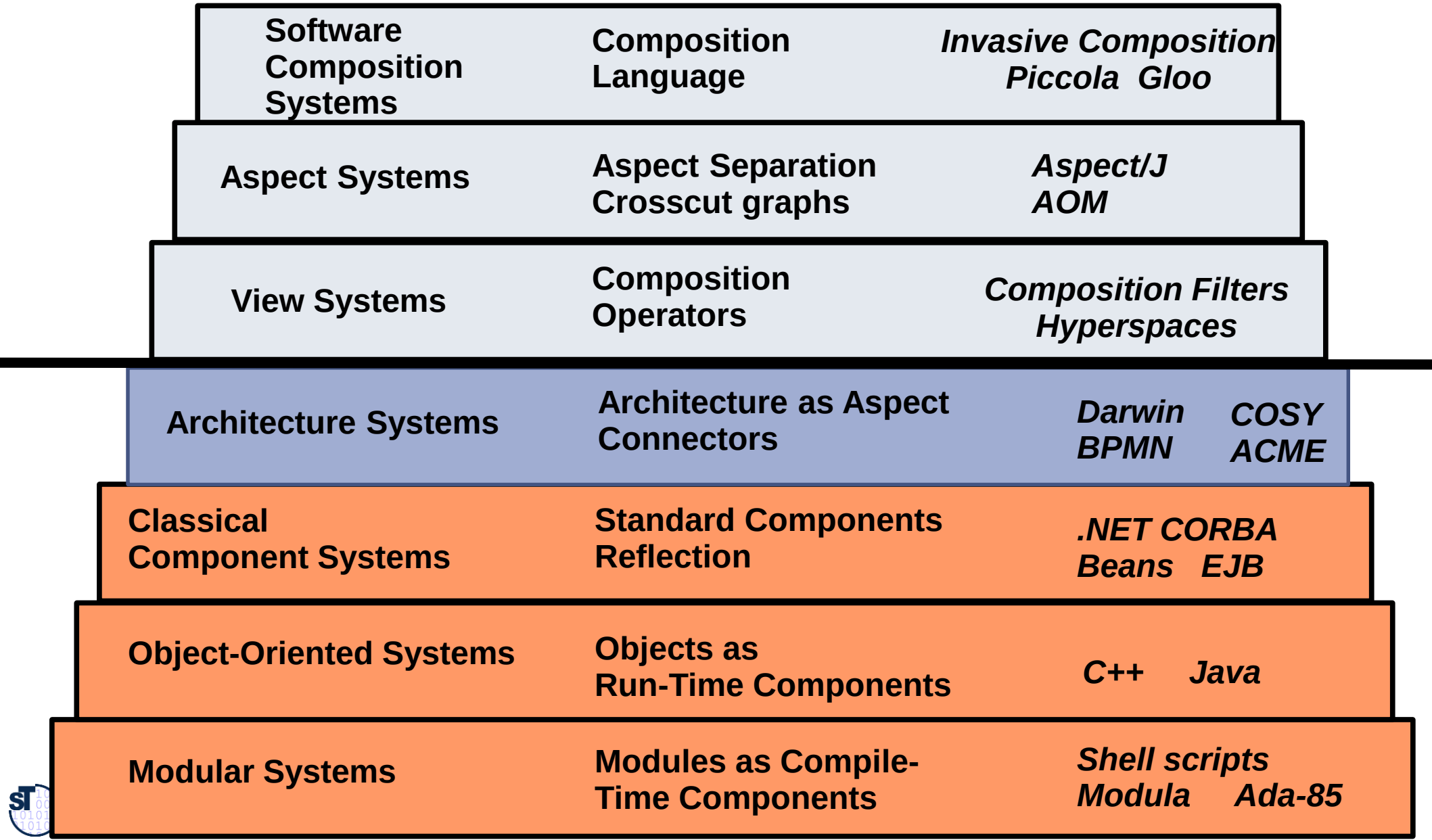
Examples of Architecture Systems

- Shaw, M, DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G, Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, April 1995, S. 314-335. (UNICON) http://ieeexplore.ieee.org/abstract/document/385970/
- D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering, S. 717--734, Sept. 1995. (RAPIDE)
- Gregory Zelesnik. The UniCon Language User Manual. School of Computer Science, Carnegie Mellon University Pittsburgh, Pennsylvania
- 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.

Other References

- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004.
- McMenamin, S., Palmer, J.: Essential Systems Analysis. Yourdon Press, 1984
  - In German: Strukturierte Systemanalyse; Hanser Verlag 1988

The Ladder of Composition Systems



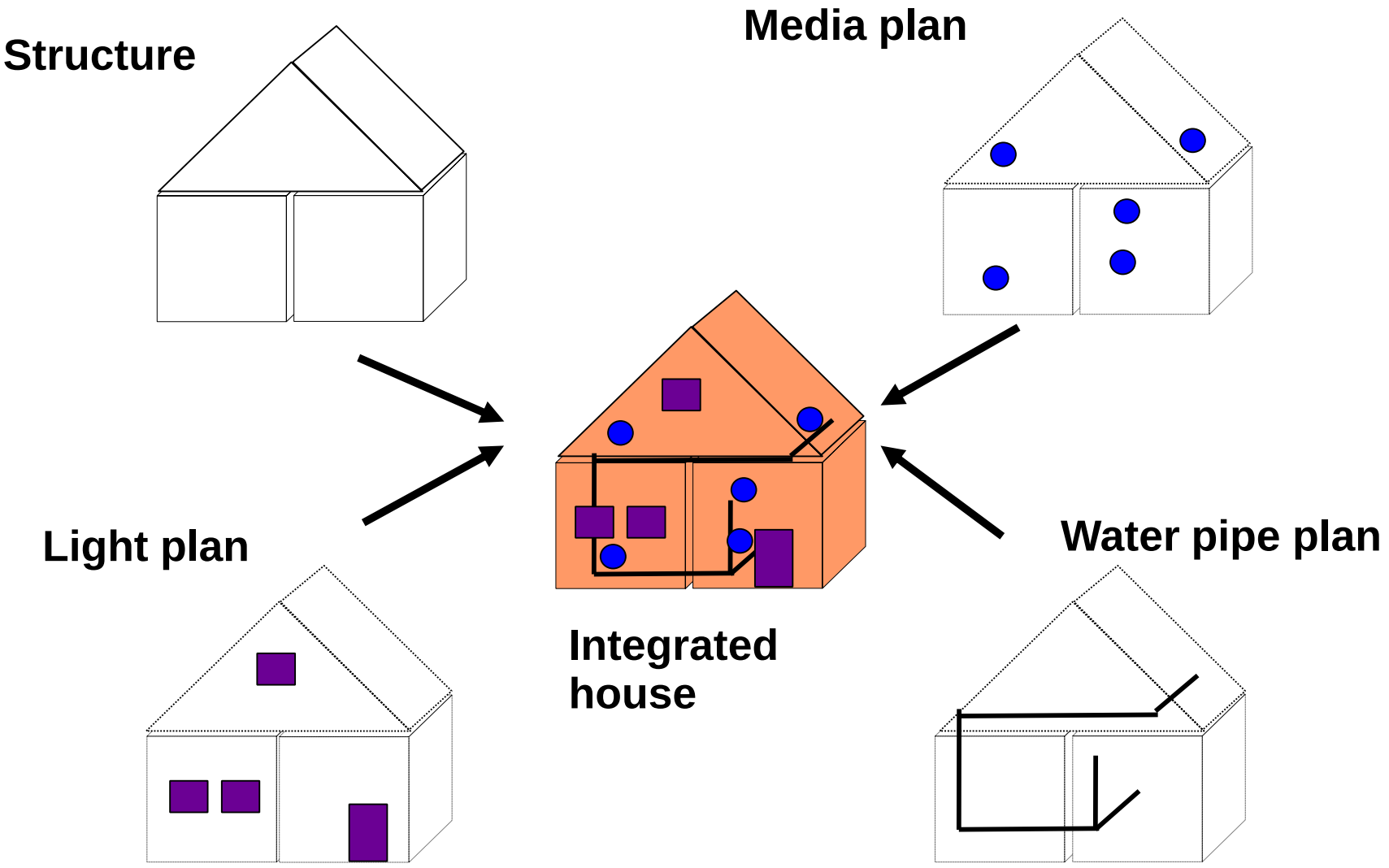
30.1. Separation of Concerns

- Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. ... **It is what I sometimes have called "the separation of concerns"**, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of.
- Edsgar W. Dijkstra, "On the role of scientific thought", [Dij82]

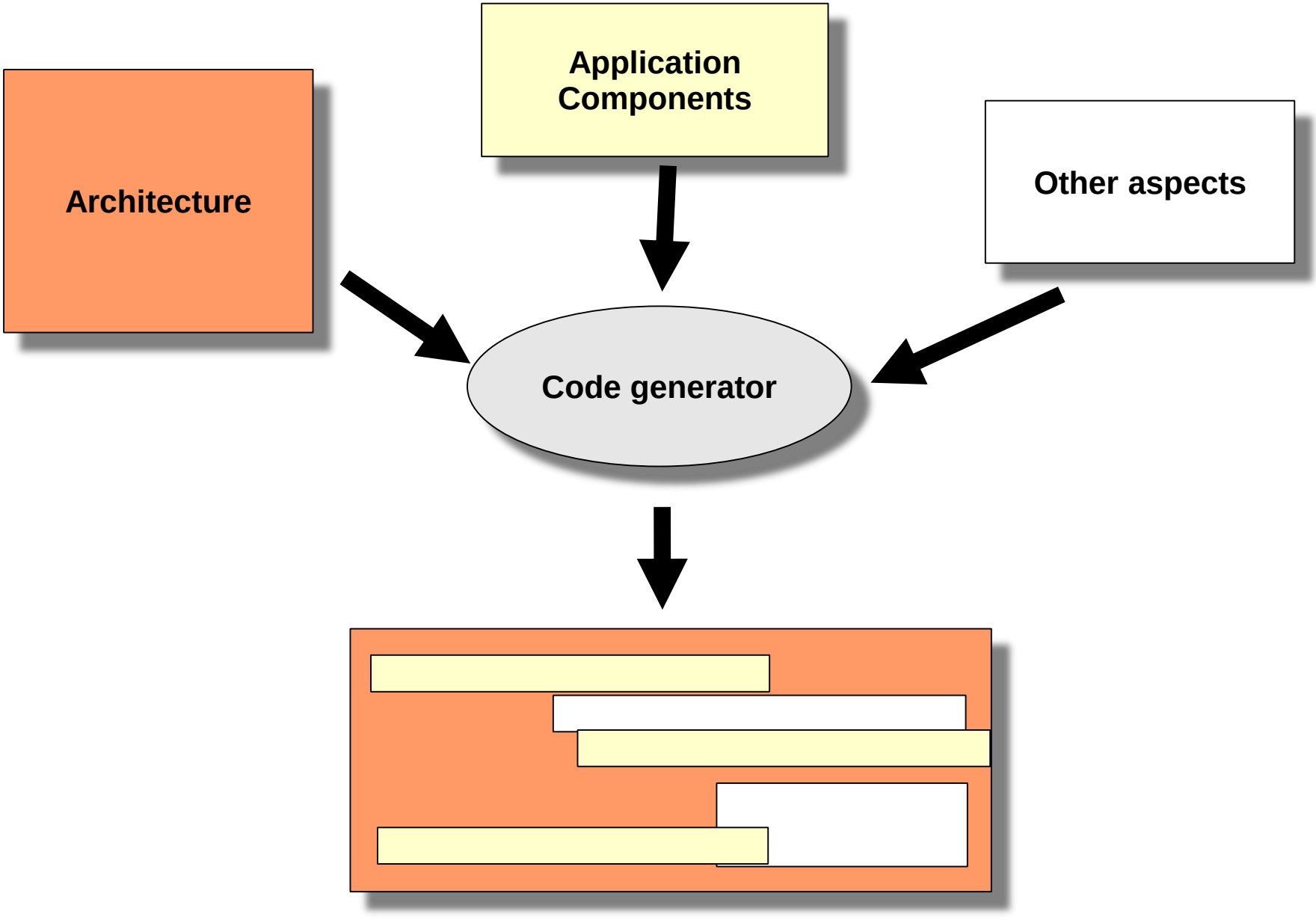
A Basic Rule for Design

- ... is to focus at one problem at a time and to forget about others:
- *Abstraction* is *neglection of unnecessary detail*
  - Display and consider only essential information
- Heuristic: *Separation of Concerns (SoC)*
  - Different concepts should be separated so that they can be specified independently
  - Every separated concept neglects unnecessary details
  - Dimensional specification: Specify a system from different viewpoints and abstract for every viewpoint from unnecessary details
- An Example of SoC: Separate Policy and Mechanism
  - Mechanism:
    - The way how to technically realize a solution
  - Policy:
    - The way how to parameterize the realization of a solution
  - Objective: vary policy independently from mechanism

Aspects in Architecture as an Example of SoC



Another Example of SoC: Architectural Aspect in Software



Architecture Systems as Automated Architectural Views

Architecture Systems advance in all three criteria groups for composition systems:

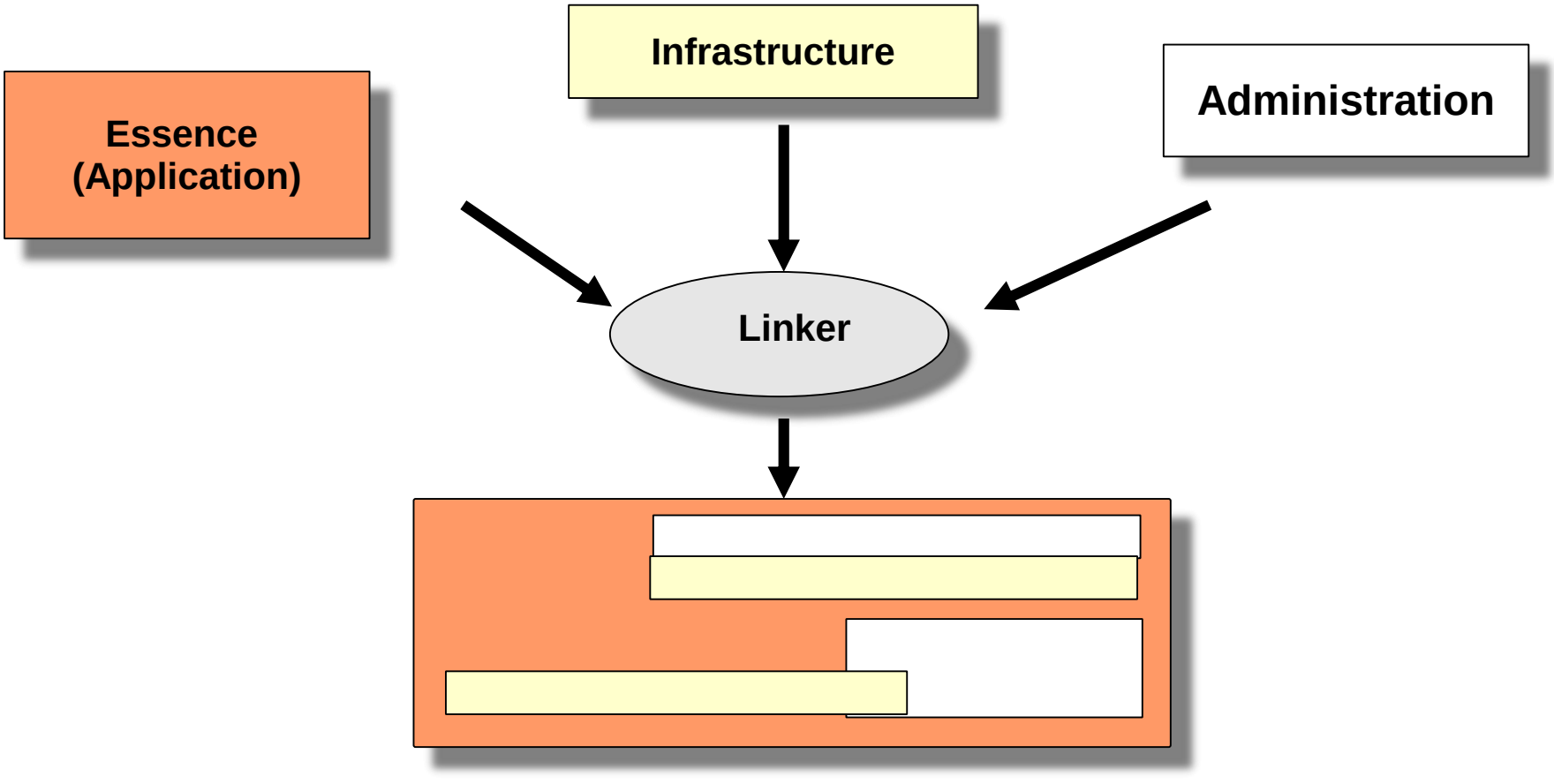
- **Component model**
  - Binding points: Ports
  - Transfer (carrier) of the communication is transparent
  - Hierarchical components by *encapsulation*
- **Composition technique**
  - Adaptation and glue code by *connectors*
  - Aspect separation: application and communication are separated
    1. Topology (with whom?): Skeletons, coordinators
    2. Carrier (how?)
  - Scalability (distribution, binding time with dynamic architectures)
  - Architectural skeletons as composition operators
- **Composition language:** Architecture Description Language (ADL)

What are the Concerns of Architecture we are Interested in?

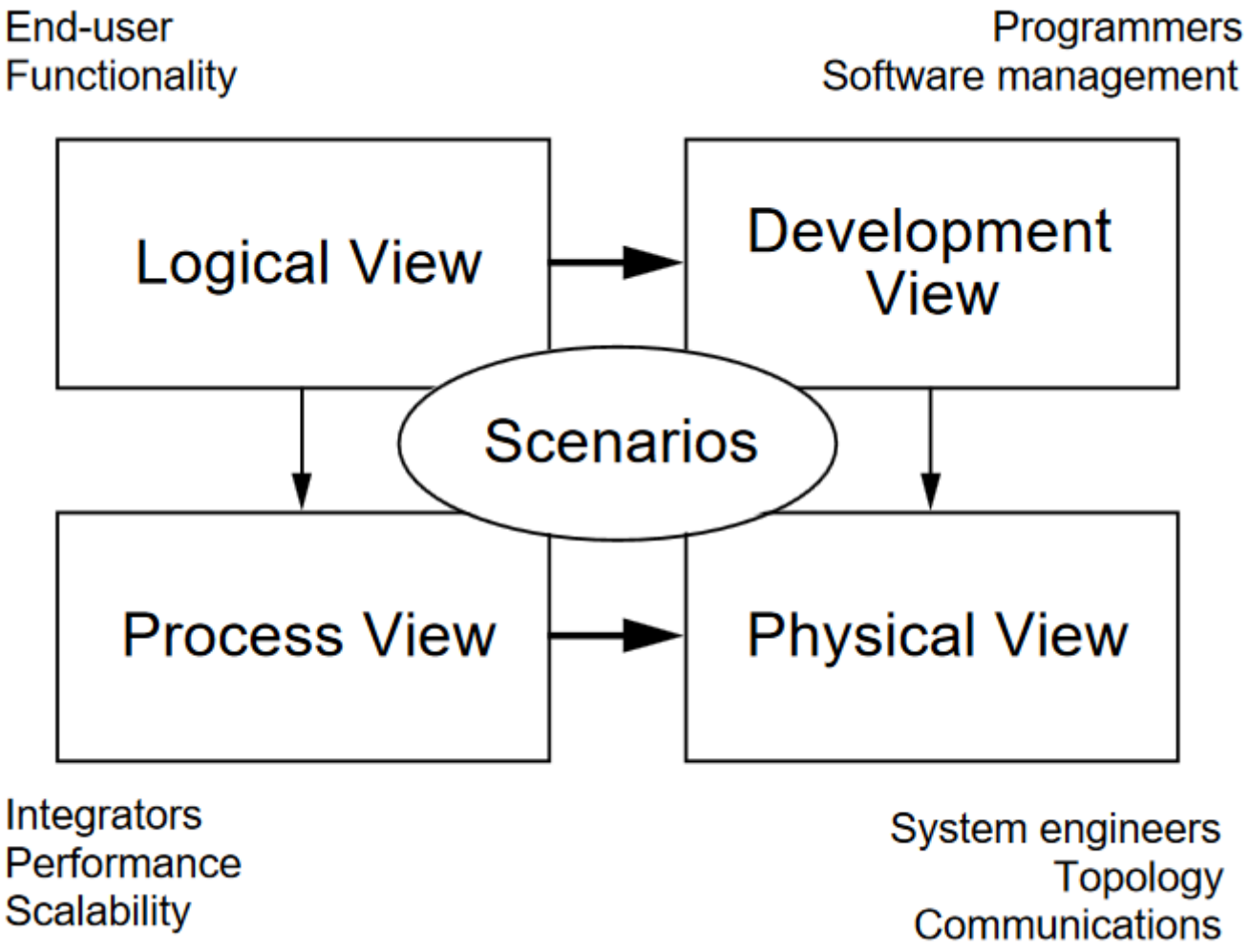
30.1.2 VIEW MODELS OF ARCHITECTURE

Essence – Administration – Infrastructure (EAI)

- [McMenamen/Palmer] EAI separates a software system into three types of components
  - Functional essence (application components)
  - Administration (contracts, quality assurance)
  - Infrastructure components (communication, architecture)



Kruchten‘s 4+1 View Model of Software

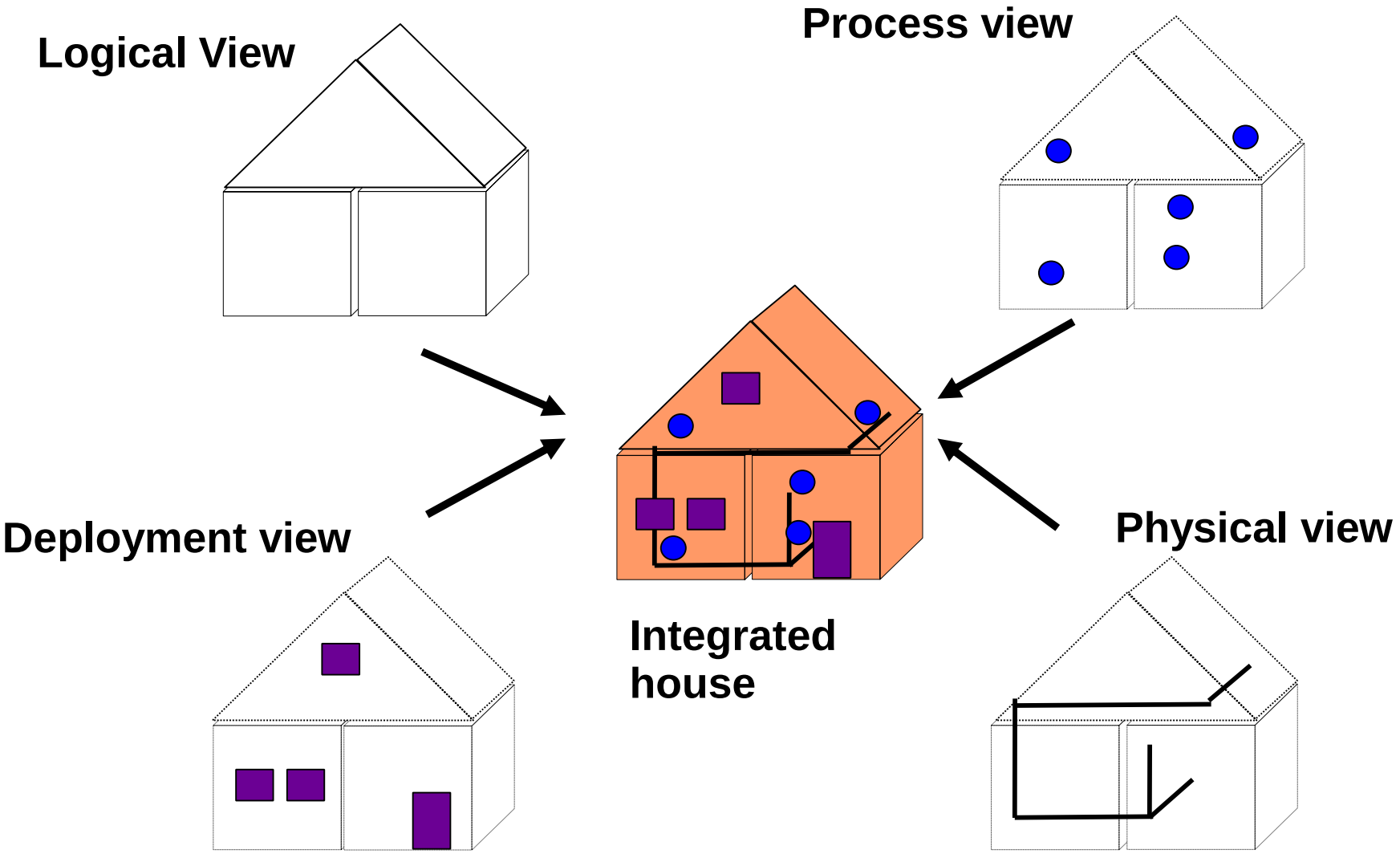


Philippe Kruchten: Architectural Blueprints—The “4+1” View Model of Software Architecture. In IEEE Software 12 (6). November 1995, pp. 42-50

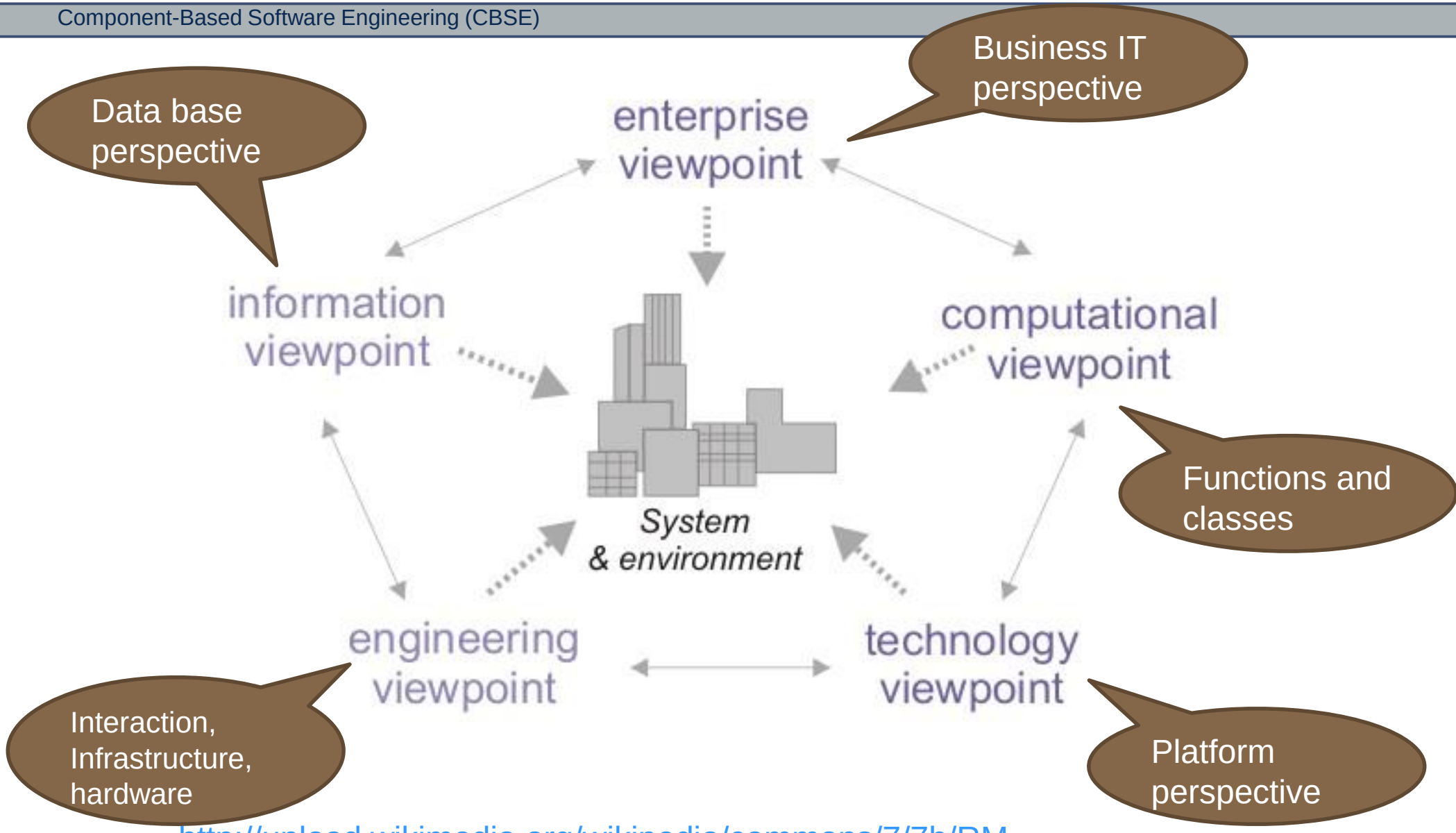
The 4-View to Software Architecture

- [Hofmeister/Sony/Nord. Applied Software Architecture] fills the Kruchten Model with more content
- Software architecture consists of 4 views
  - *logical view (conceptual view,* component-connector architectures)
    - specifies the functional requirements and structure, in a component-based UML model
    - This is the focus in this chapter
  - *process view (dynamic view)*
    - specifies non-functional features as performance, reliability, fault tolerance, parallelism, division in processes.
  - *development view (project tree structure)*
    - specifies the file organisation the modules, libraries, subsystems, the static structure the software in the development environment
  - *physical view (run-time view)*
    - specifies the mapping of the software to the hardware, distribution, processes, etc., and the run-time execution structure
- For all these views, architecture diagrams can be made in different modelling languages
  - Here, we treat only the *logical view*

Hofmeister Aspects (Profiles in UML)



Reference Model for Open Distributed Processing (RM-ODP)

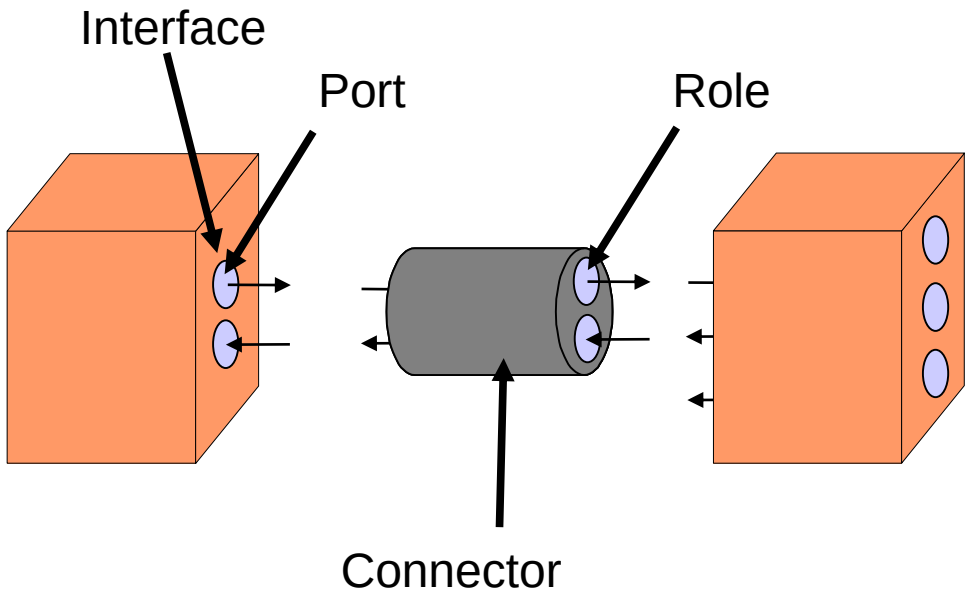


http://upload.wikimedia.org/wikipedia/commons/7/7b/RM-ODP_viewpoints.jpg

30.2 ELEMENTS OF ARCHITECTURE DESCRIPTION LANGUAGES

Component Model in Architecture Systems

- *Ports* abstract interface points (event channels, methods)
  - Ports specify the data-flow into and out of a component
  - In the simplest case, ports are methods, such as in(data), out(data)
- *Connectors* are special communication components
  - Connectors are attached to ports
  - Connectors abstract from the concrete carrier
  - Can be binary or n-ary
  - Connector end is called a *role*
  - A role fits only to certain types of ports (typing)
- *Encapsulation* is a composition operator to build hierarchies



30.2.1 PORTS

Abstract Binding Points: Data Ports

- A **data port** denotes a spot in the component, where it reads data from (or writes data to) the outer world
- A **control-flow port (service port)** denotes a spot in the component which can be triggered for an activity
- Ports are **provided** or **required**
  - A port is **provided**, if the component offers its implementation for external use
  - A port is **required**, if the component needs an implementation for it from another component in the external world
- Ports are
  - **Synchronous** or **asynchronous** (partner has to wait or not)
  - **Singular** or **continuous** (communication can take place once or many times)
  - **Atomic** or **composite**

Different Data Ports

Synchronicity

- **Input data ports** are synchronous or asynchronous: in(data)
  - get(data) aka receive(data): Synchronous in-port, taking in one data
  - testAndGet(data): Asynchronous in-port, taking in one data, if it is available
- **Output data ports** are synchronous or asynchronous: out(data)
  - set(data): Synchronous out-port, putting out one data, waiting until acknowledge
  - put(data) aka send(data): Asynchronous out port, putting out one data, not waiting until acknowledge

Continuity

- **Stream ports** (**channels, pipes**): continuous data port
  - Can be realized by Design Pattern Iterator
- **Event port**: asynchronous continuous data port

Composite Ports (Services)

Ports can be **atomic** or **composite (structured)**

- A **service** is a structured port (groups of ports)
- A **data service** is a tuple of atomic ports:

```
[in(data), ..., in(data), out(data), ..., out(data)]
```

- A **call port** is a synchronous input/output composite, singular port with one out-port, the return

```
[in(data), ..., in(data), out(data)]
```

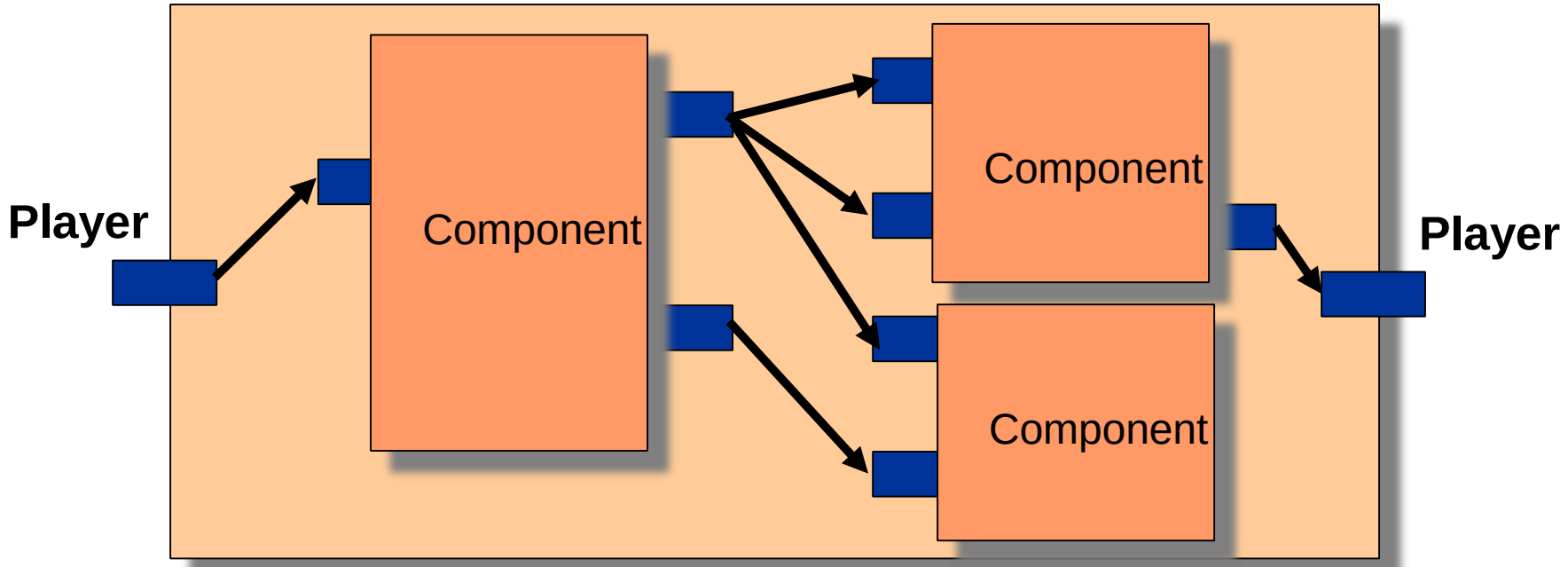
- A **property service** is a synchronous singular data service to access component attributes, i.e., a simple tuple of in and out ports

```
[in(data), out(data)]
```

30.2.2 ENCAPSULATION

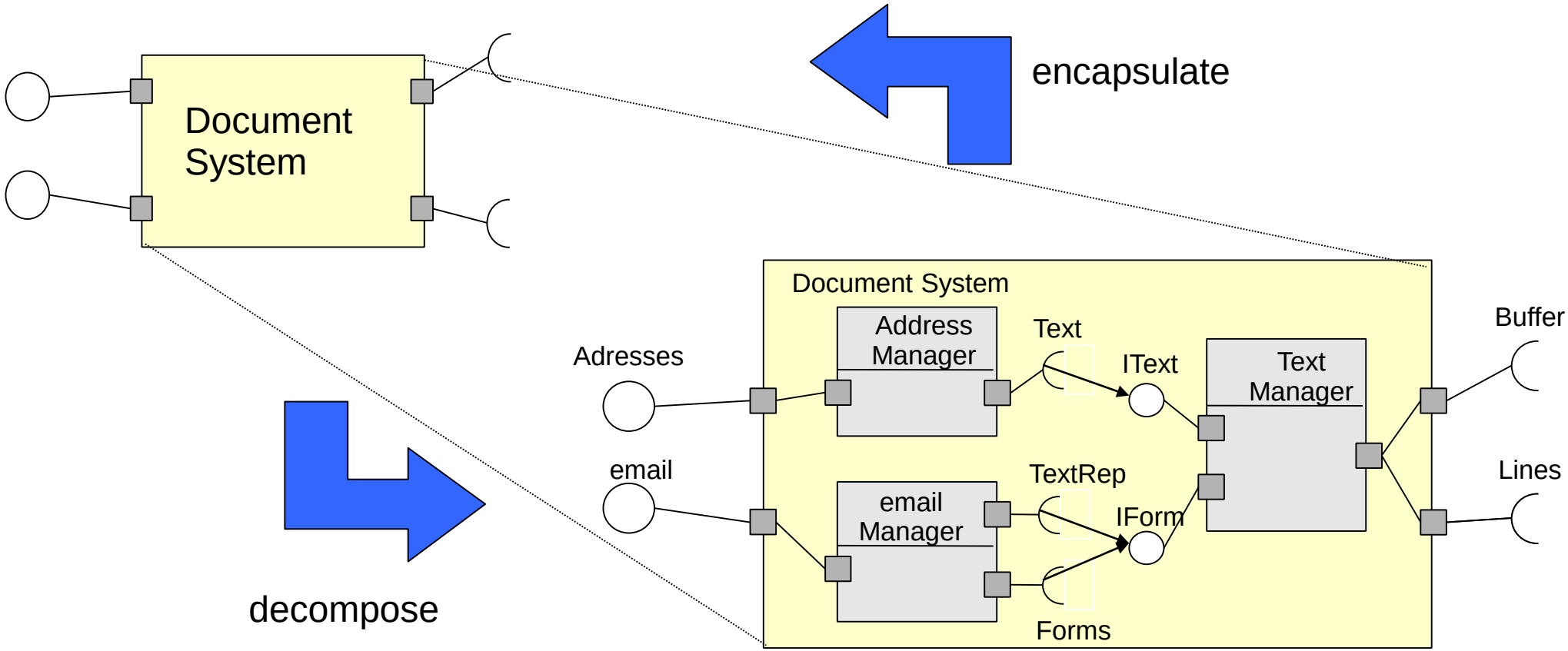
Hierarchic Architectures with Encapsulation

- Components can be connected by **connectors**
  - Then, the tuple space is split into communication channels, avoiding bottleneck
  - Protocols are hidden in the connector
- Components can be nested by an **encapsulation operator**
  - The operation "encapsulate" hides encapsulated components in an outer component
  - Architectures become hierarchical, reducible structures (with fractal-like zoom-in/out)
  - Ports of outer components are called *players*
  - Connectors from players to ports of inner components are called *delegation connectors*
  - A *topology* is the network of connectors and ports within a component



Nesting of Components with the Encapsulation Operator

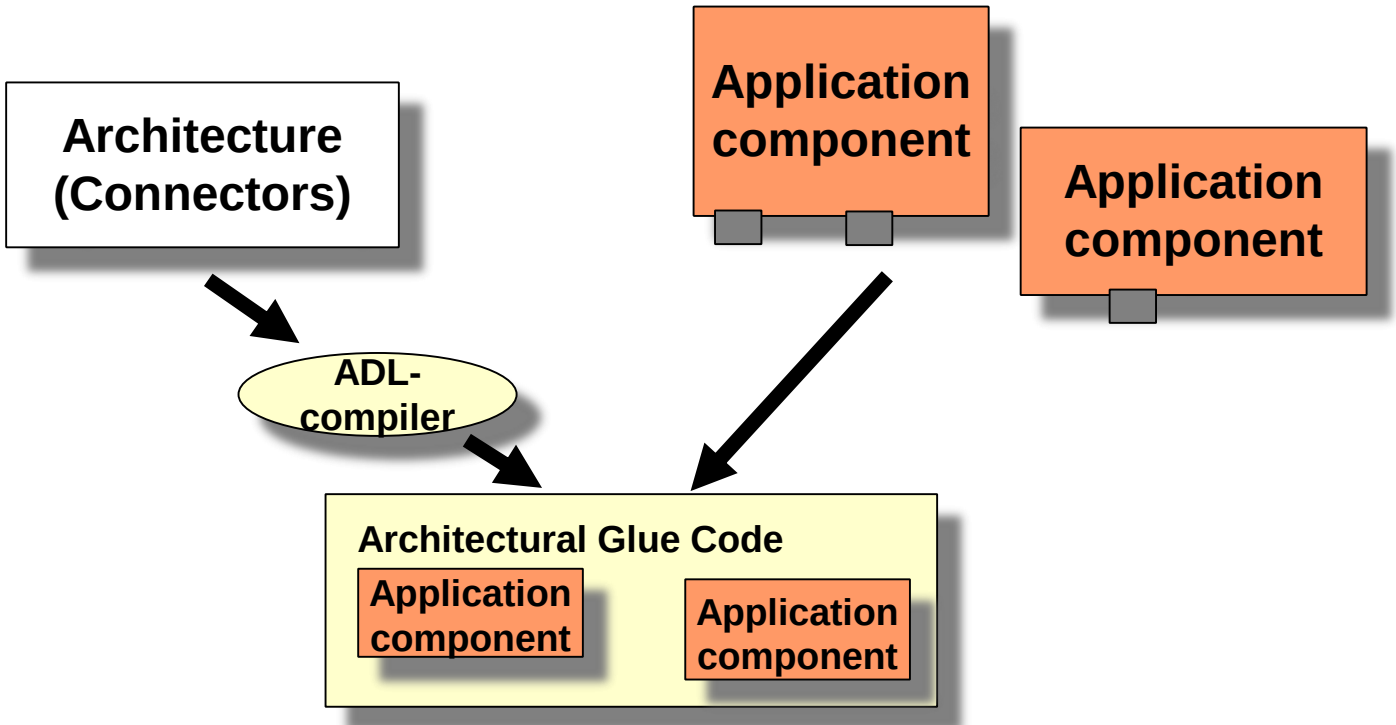
- In most component models, components are nested.
- Nesting is indicated by *aggregation* and *part-of relationship*.
- Nesting is introduced by an encapsulation operator encapsulate.



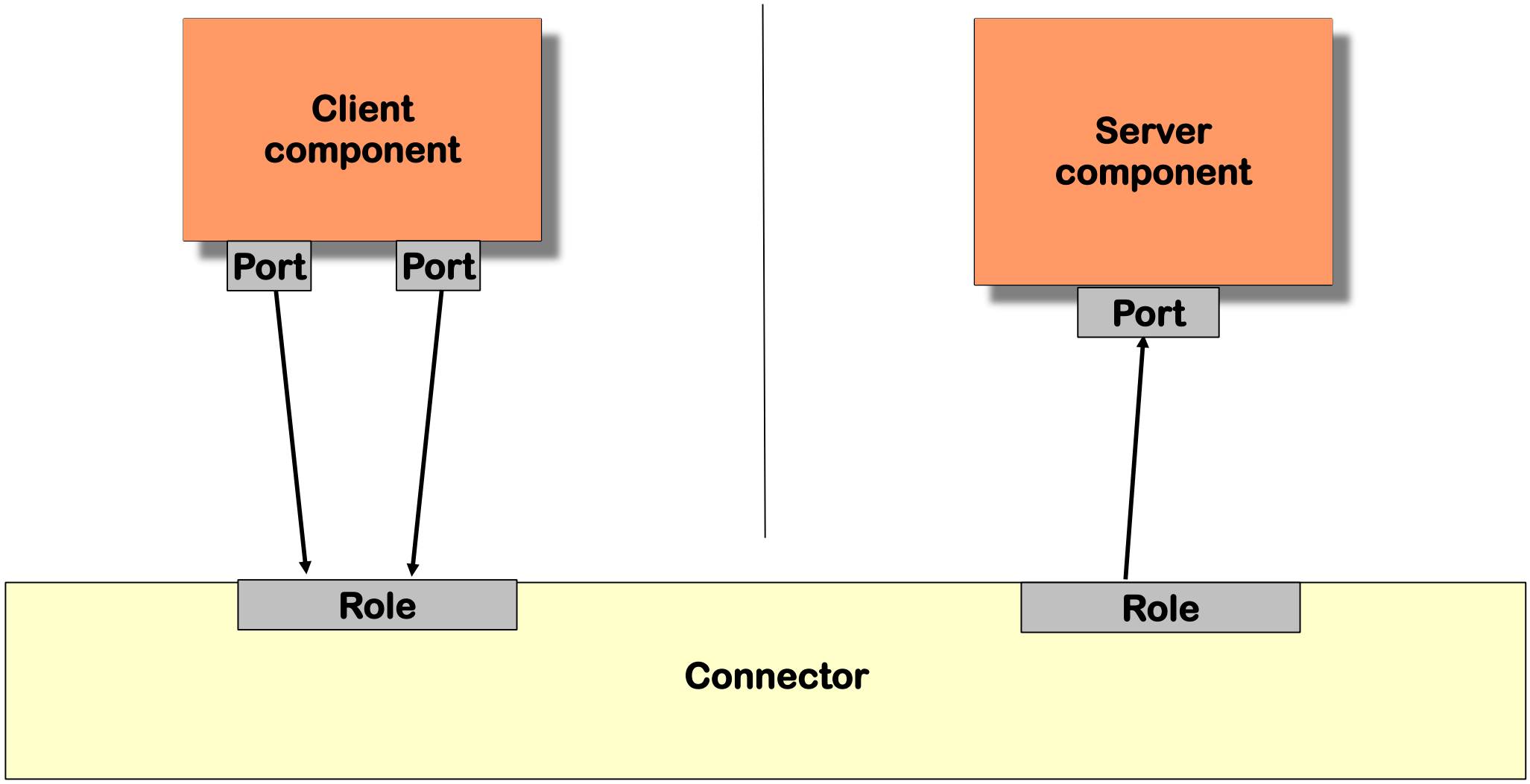
30.2.3 CONNECTORS

Connectors Generate Architectural Code

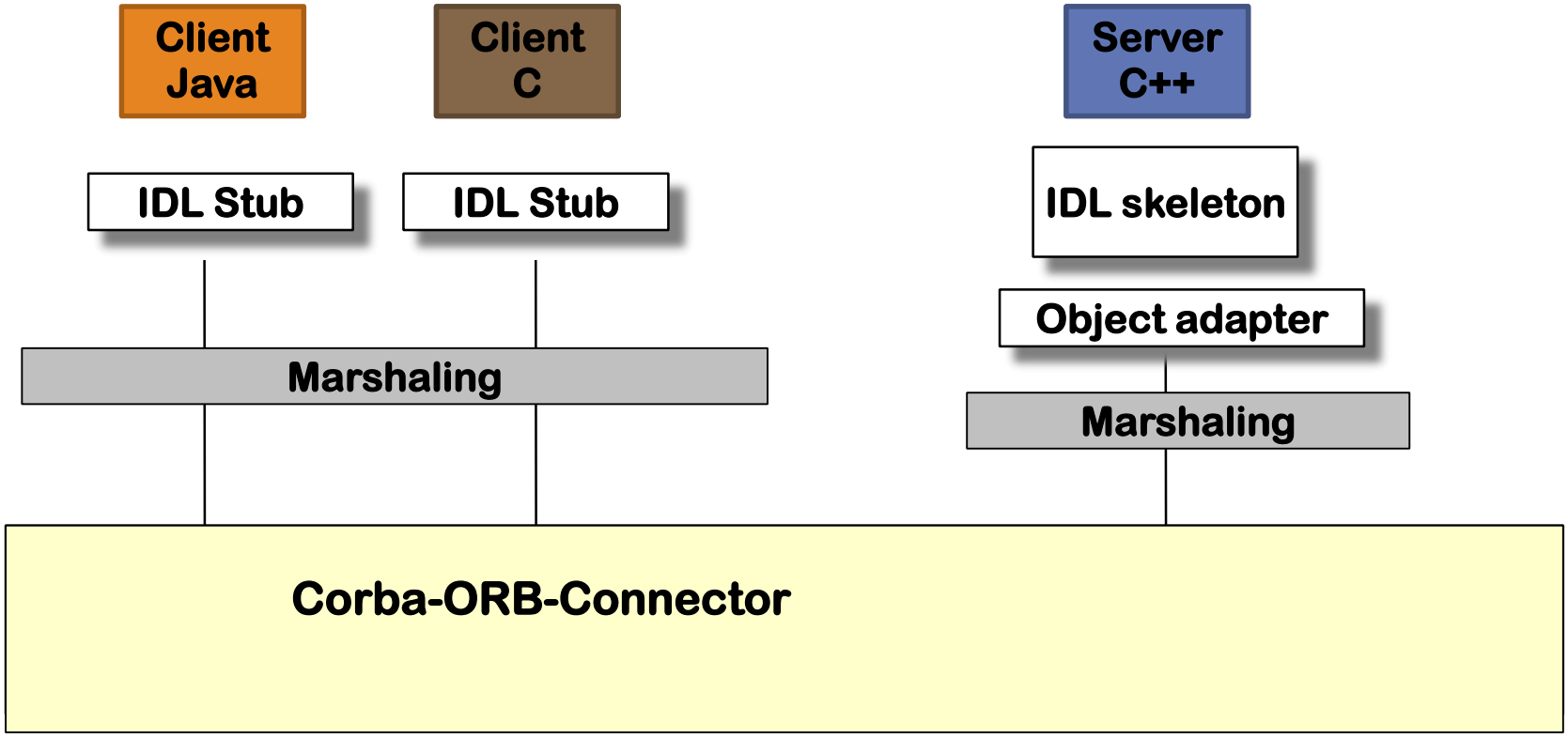
- Glue- and adapter code from connectors, skeletons, and ADL-specifications
  - Mapping of the protocols of the components to each other
- Simulations of architectures:
  - Test dummies and mocks (dummies with protocol machines)
  - The architecture can be created first and tested standalone
  - Analysis of run-time possible (if those of components are known)
- Test cases for architectures



Connectors are Abstract Communication Buses

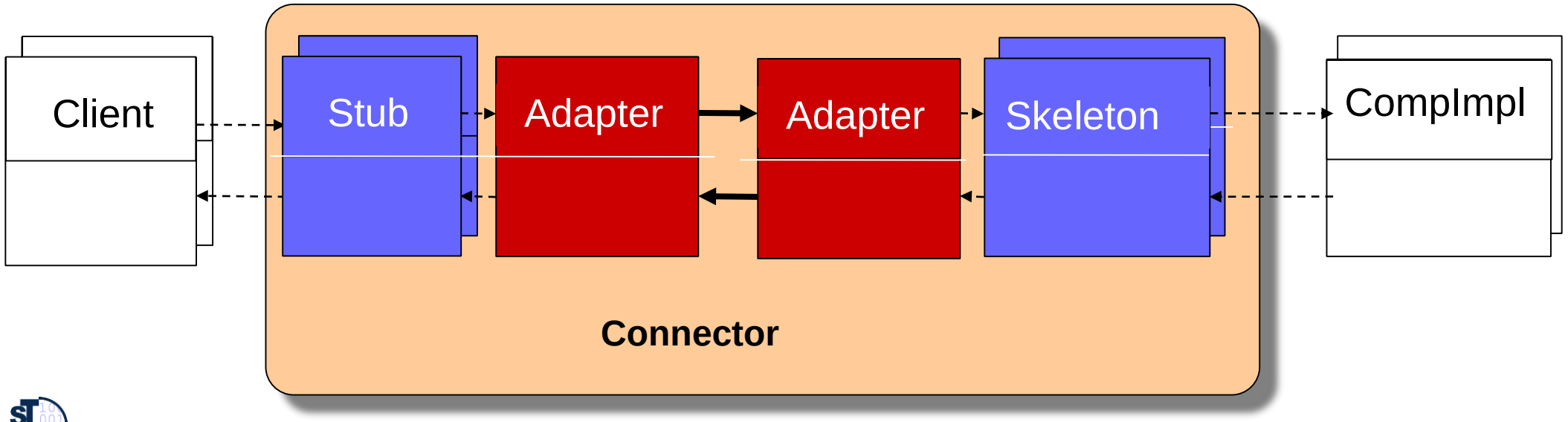


Corba has a Simple Connector, but it is not explicit



Most Commercial Component Systems Provide Restricted Forms of Connectors

- It turns out that most commercial component systems do not offer connectors as explicit modelling concepts, but
  - offer communication mechanisms that can be encapsulated into a connector component
  - For instance, CORBA remote connections can be packed into connectors



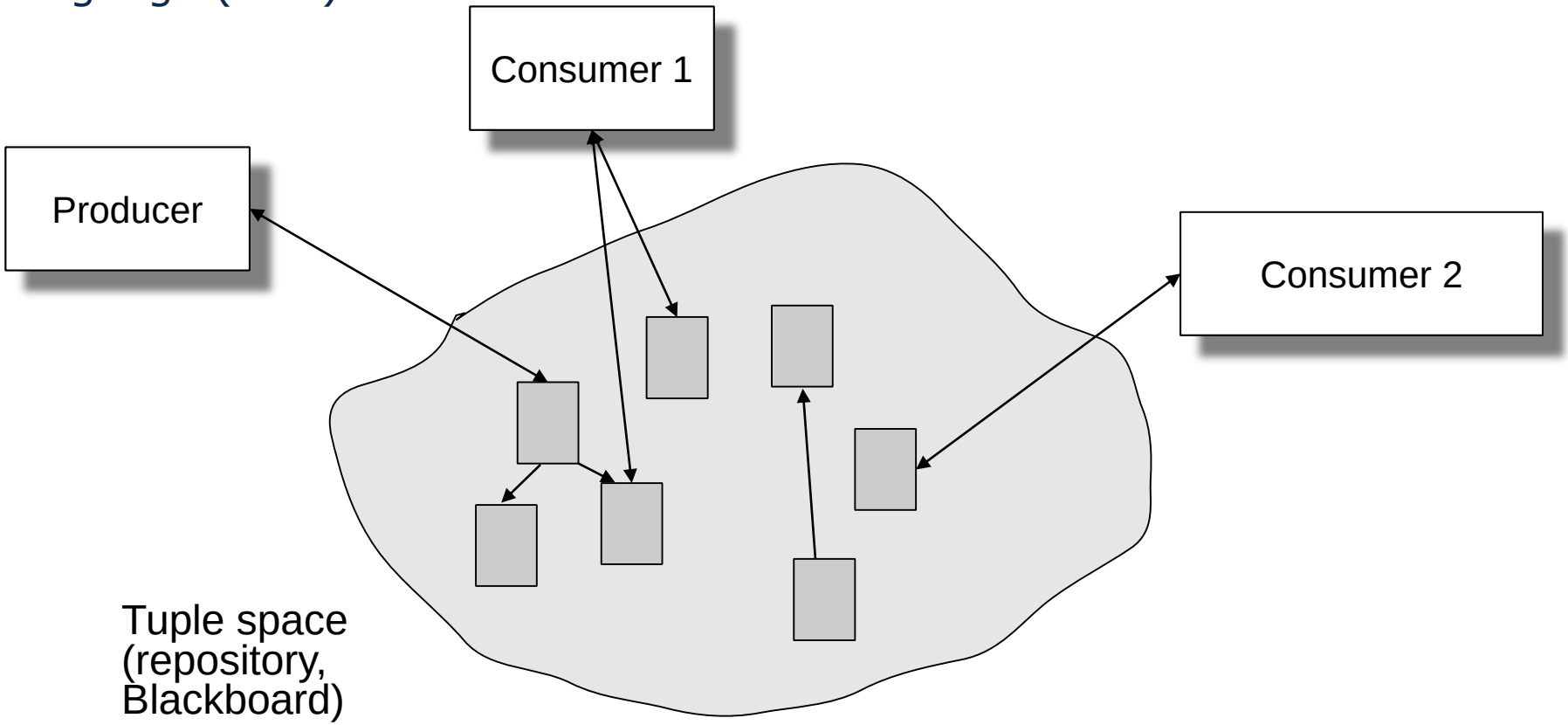
Insight: CORBA is a Simple Architecture System with Restricted Connectors

- CORBA:
  - Client and service components
  - ORB client side, server side
  - Marshalling, proxy, Stub, Skeleton, Object Adapter
  - Interfaces in IDL (not abstracted to ports)
  - static call
  - dynamic call
  - connectors always binary
  - Events, callbacks, persistence as services (cannot be exchanged to other communications)

- Architecture System:
  - Components
  - Connectors
  - Roles
  - Ports
  - procedure call connector (also distributed)
  - dynamic reconfigurable connectors (e.g., in Darwin)
  - connectors n-ary
  - All these as connectors (can be exchanged to other communications)

A Complex Connector: Repository Connector (Tuple Space Connector)

- A specific, large connector is the **repository (tuple space)**
- Based on data ports, components can communicate via tuples of data, emitting and receiving from a tuple space
  - Repository offers data objects (material) with data ports
  - Active components work (tools) on the material
- Data in tuple spaces can be untyped, or typed by a data definition language (DDL)



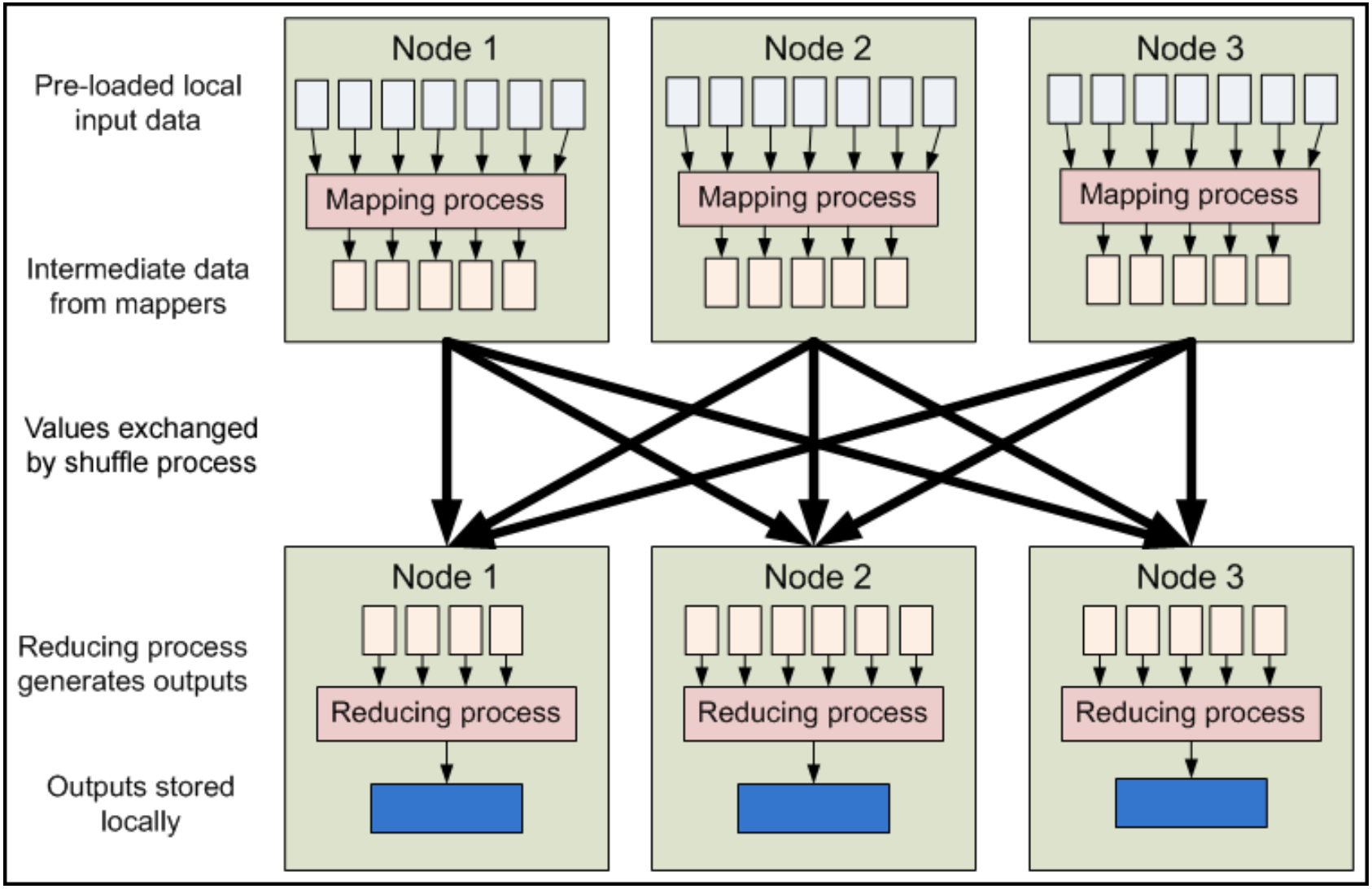
http://homepages.inf.ed.ac.uk/mic/Skeletons/

30.2.4 COORDINATORS (SKELETONS)

Complex Composition Operator: Coordination Skeletons

- An **architectural skeleton** is a coordination scheme for a set of components superimposing a topology of connectors (connector nets)
  - their encapsulation to a new component
- Example: the **Map-Reduce Skeleton** (Google) for searching
  - Divide-and-conquer, partition, zip, serialize, ...

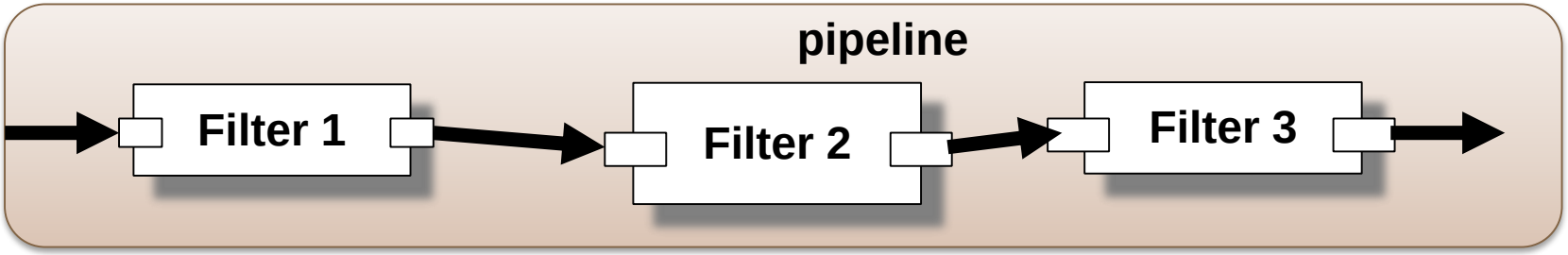
Map-Reduce in Hadoop



https://developer.yahoo.com/hadoop/tutorial/module4.html

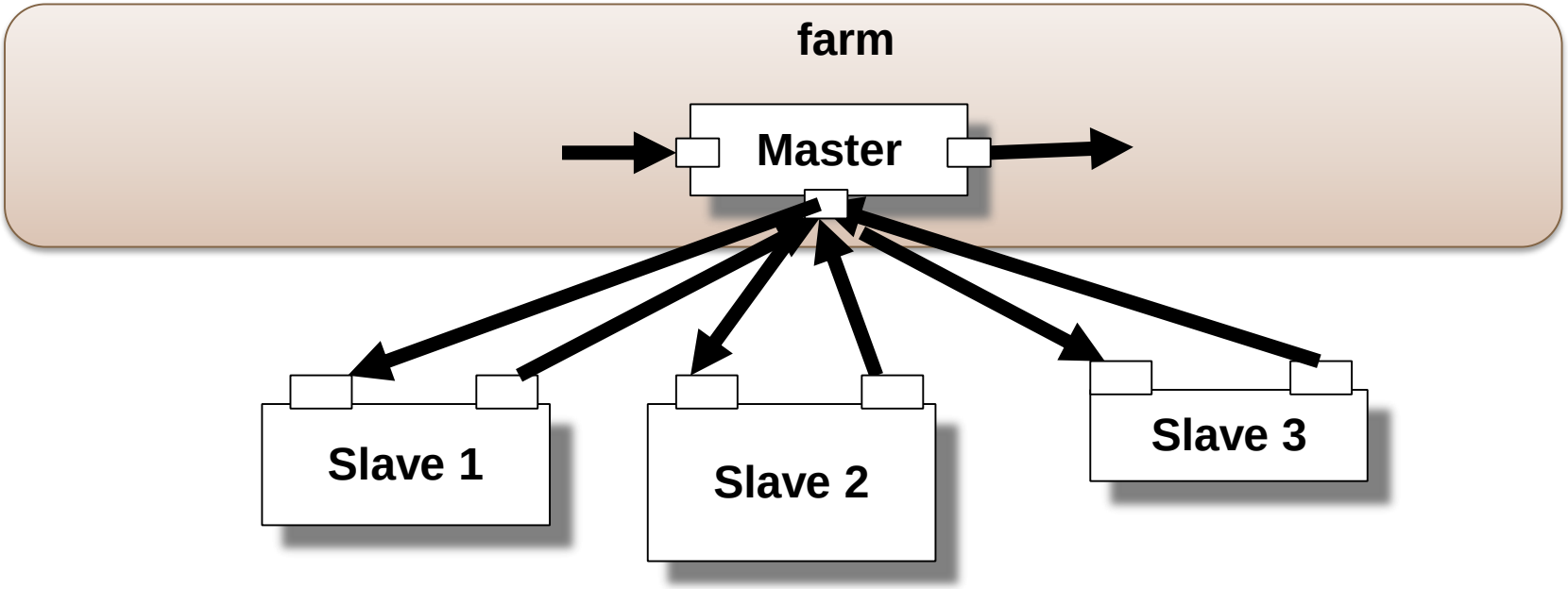
Pipeline Skeleton

- Pipes use continuous ports (data flow)



Farm Skeleton (Coordinator)

- Farms use masters distributing tasks to workers (call ports)



30.2.5 Architecture Styles

Architectural Styles

- An **architecture style** employs for a system or a layer only particular architectural concepts and defines constraints on their employment [Garlan/Shaw: Software Architecture]:
  - particular composition operators (specific connectors, coordinators, skeletons, …)
  - particular communication carriers or topologies
  - Particular form of control- and data-flow
- Architectural sytems can be expressed **by architectural rules** (architectural constraints), often specified in logic
  - Wellformedness constraints, which architectural concepts may be used in the style and which are forbidden
  - Ex.: Pipe-and-filter style, repository style, call-based style, event-driven architecture, 3-tier architecture, and many more

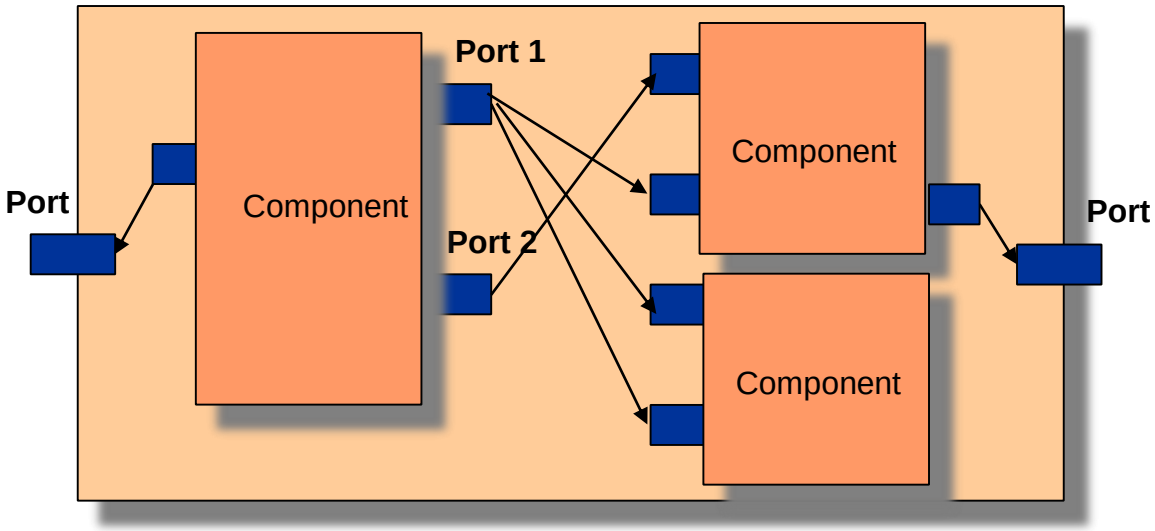
Which Types of Control- and Data-flow Specifications Exist for Architectures?

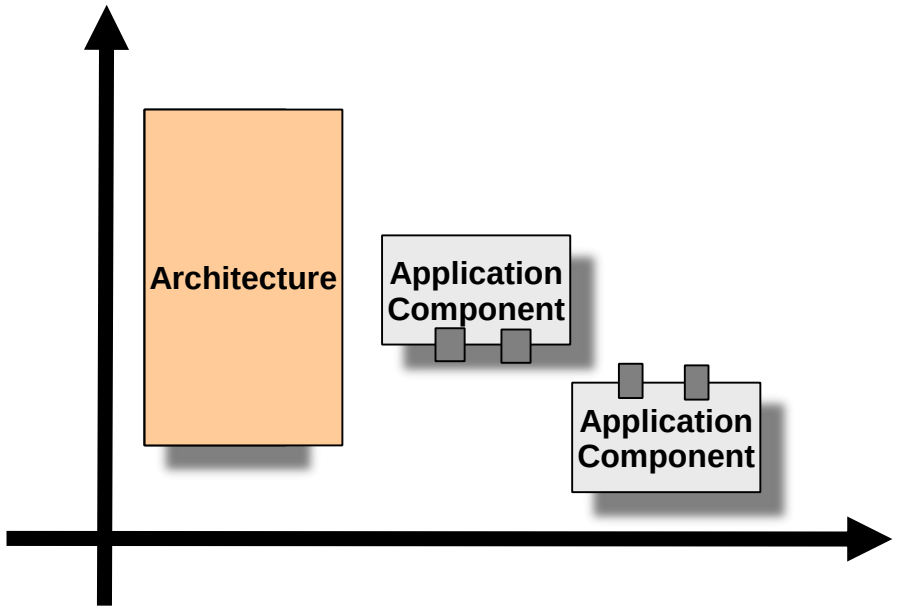
- **Data-flow graphs** (data flow diagrams, DFD) focus on data flowing through operations
  - Activity diagrams: data flows through actions
  - See courses Softwaretechnologie II
- **Control-flow graphs** (CFG) focus on control dependencies
  - Nodes are control-flow operations that start other operations on a state
  - The standard representation for imperative programs
- **State systems** focus on transitions between states
  - Finite State Machines (FSM): events trigger state transitions
  - Statecharts: Hierarchical FSM
- Mixed approaches
  - **Colored Petri nets:** tokens mark control and data-flow, see course Softwaretechnologie II
  - **Cyclic data-flow graphs** (also called static-single assignment graphs, SSA)
    - Cycles are marked by phi-nodes that contain control-flow guards
  - **Workflow languages** mix control and data-flow
    - Provide specific split and join operators for control and data flow

30.2.6 ARCHITECTURE DESCRIPTION LANGUAGES (ADL)

Architecture can be Exchanged Independently of Components

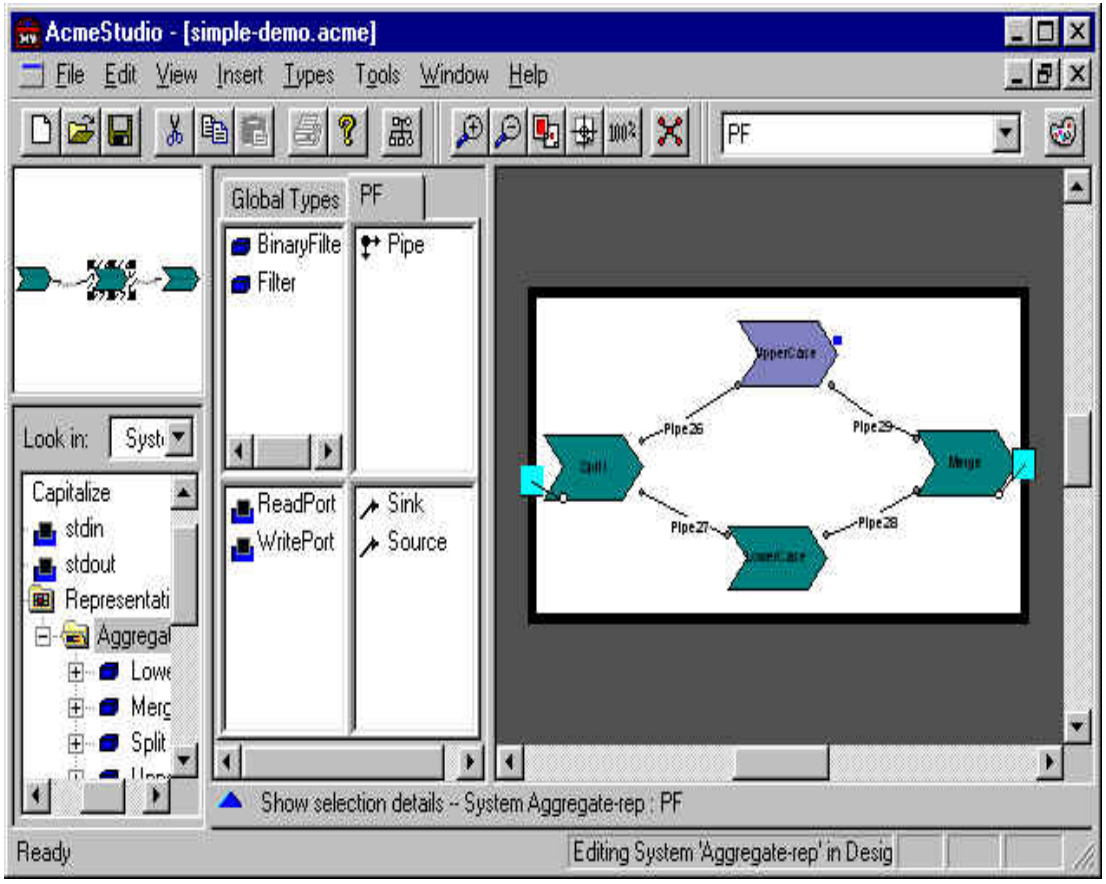
- Reuse of components and architectures is fundamentally improved
- Two dimensions of reuse
  - Architecture and components can be reused independently of each other





Architecture Systems

- Unicon [Shaw 95]
- Darwin [Kramer 92]
- Rapide [Luckham95]
- C2 [Medvedovic]
- Wright [Garlan/Allen]
- CoSy [Aßmann/Alt/vanSomeren 94]
- Modelica http://www.modelica.org, equation-based connectors
- EAST-ADL for architectures of embedded systems
  - http://www.east-adl.info/Specification.html
- ARTOP for architectures on AutoSAR for cars
  - https://www.artop.org/
- Aesop [Garlan95]
- ACME [Garlan97]:



The Composition Language: ADL and its Tools

- For an **architecture language** (architectural description language, ADL), there are several tools
  - **ADL-compiler** generating code for encapsulators, connectors and skeletons
  - **ADL-editor:** (graphic and textual) simple specification of architectures
    - The architecture is a reducible graph
    - The reducibility of the architecture allows for simple overview, evolution, and documentation
  - **ADL-style editor:** Specification of architectural styles
    - Architectural constraints, such as control/data-flow style
  - **ADL-checker:** export of the architecture to a model checker or other semantic analysis tool
  - **ADL-exchange:** XML-Readers/Writers for ADL
  - **ADL-reference-architecture editor:** Specification of reference architectures

Reference Architectures

- A **reference architecture** is a template or framework of an architecture, most often for a particular application domain.
  - It uses a predominant architectural style
  - Strong emphasis on architectural design rules
  - Can be instantiated or derived to a concrete architecture
  - Often used in product families

- Later, we will see how generic programming and view-based programming can be used to specify reference architectures

What ADL Offer for the Software Process

- Support for requirements specification
  - Client can understand the architecture graphics well
  - *Architectural styles* classify the nature of a system (similar to design patterns)
- Design support
  - Visual and textual views to the software resp. the design
  - Refinement of architectures (stepwise design, design to several levels)
  - Design of product families
    - A *reference architecture* fixes the commonalities of the product line
    - The components express the variability
- Support for validation
  - Consistency checking tools for consistency of architectures
  - Type checking: are all ports bound? Do all protocols fit?
  - Does the architecture corresponds to a certain style ?
  - Does the architecture fit to a reference architecture?
  - Checking, analysing deadlock, liveness, fairness checking

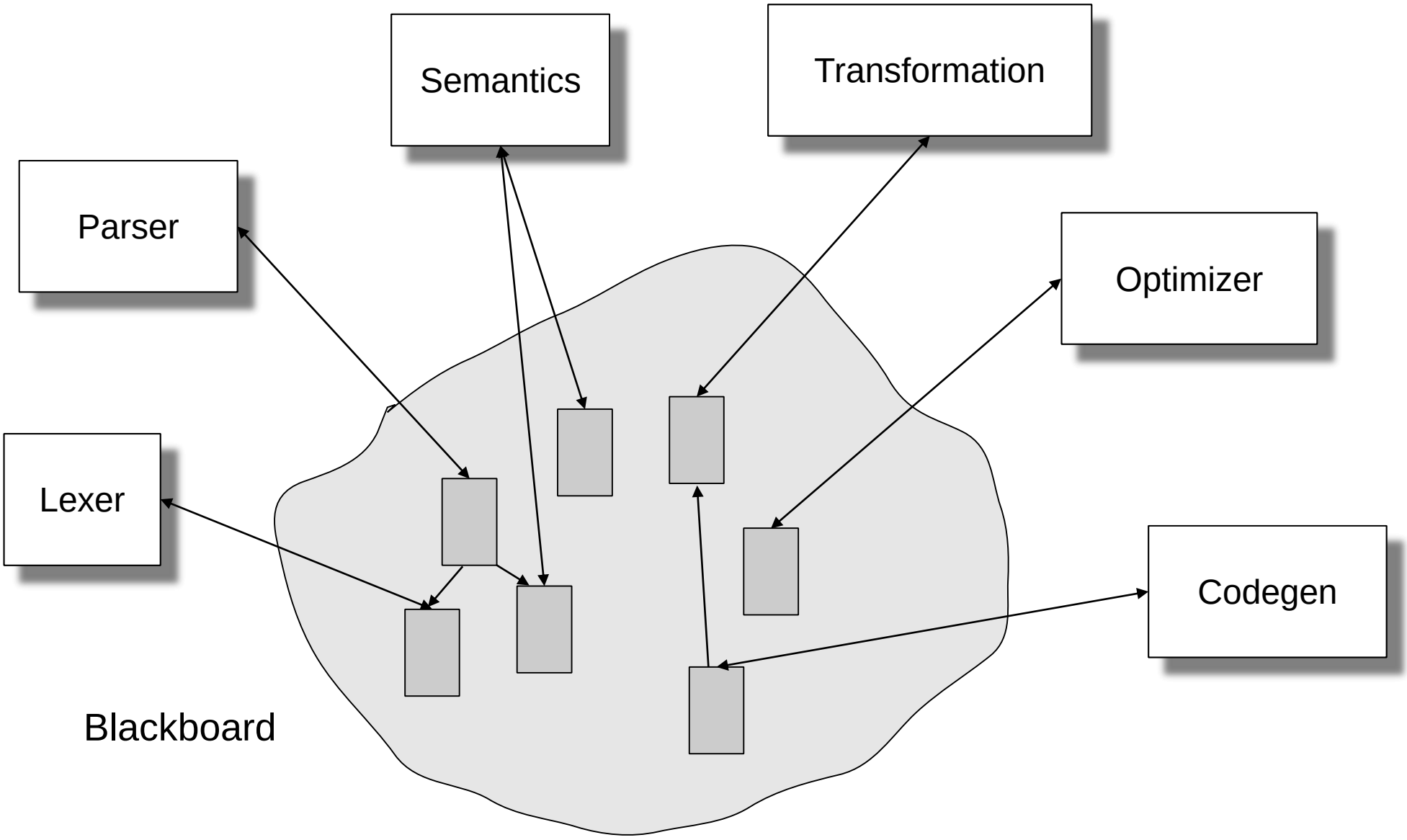
Support for Testing

- Dummy-in-the-loop vs software-in-the-loop:
  - Instead of the components, dummies can be used to test the architecture
- Software-in-the-loop vs Hardware-in-the-loop
- Test suites for architecture and components can be split
  - And reused independently

30.3 Examples for Architecture Systems

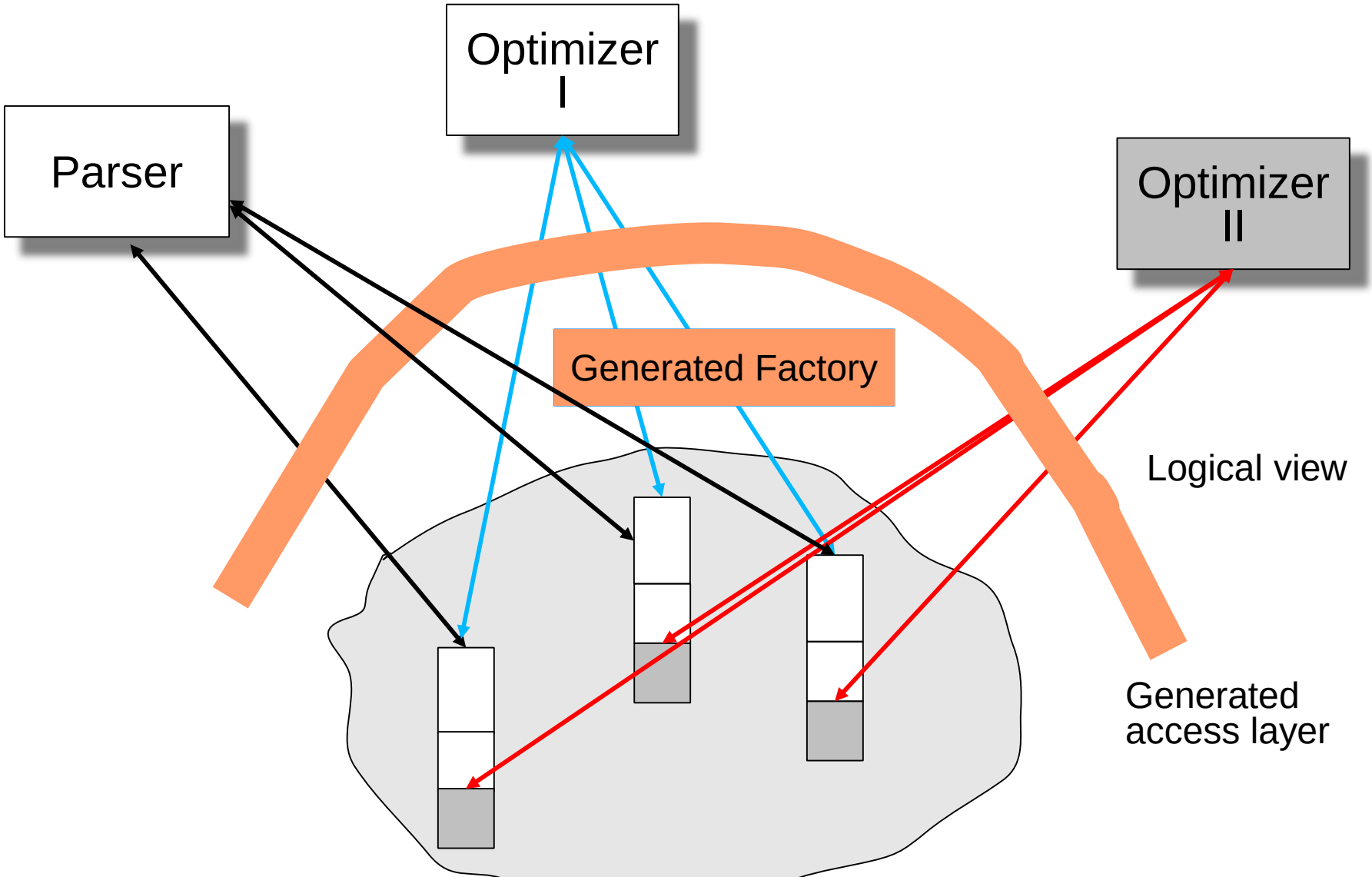
- 30.3.1 CoSy - A commercial architecture system for compilers and repository-based systems

A CoSy Compiler with Repository-Style Architecture (Typed Tuple Space)

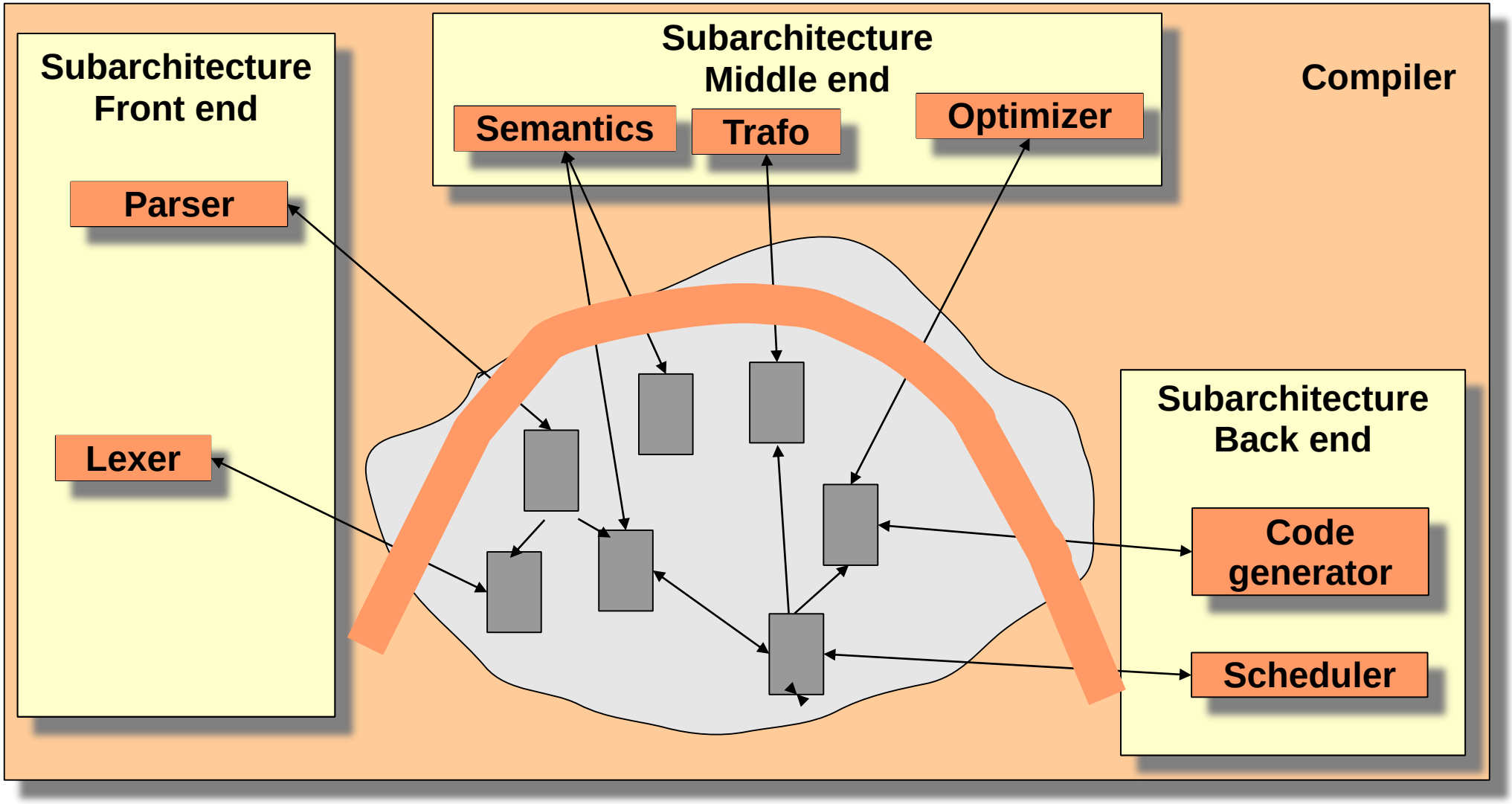


A CoSy Compiler has a Repository Connector Layer (Access Layer)

- Access to data objects (material) in repository is via *memory (tuple space) connector*

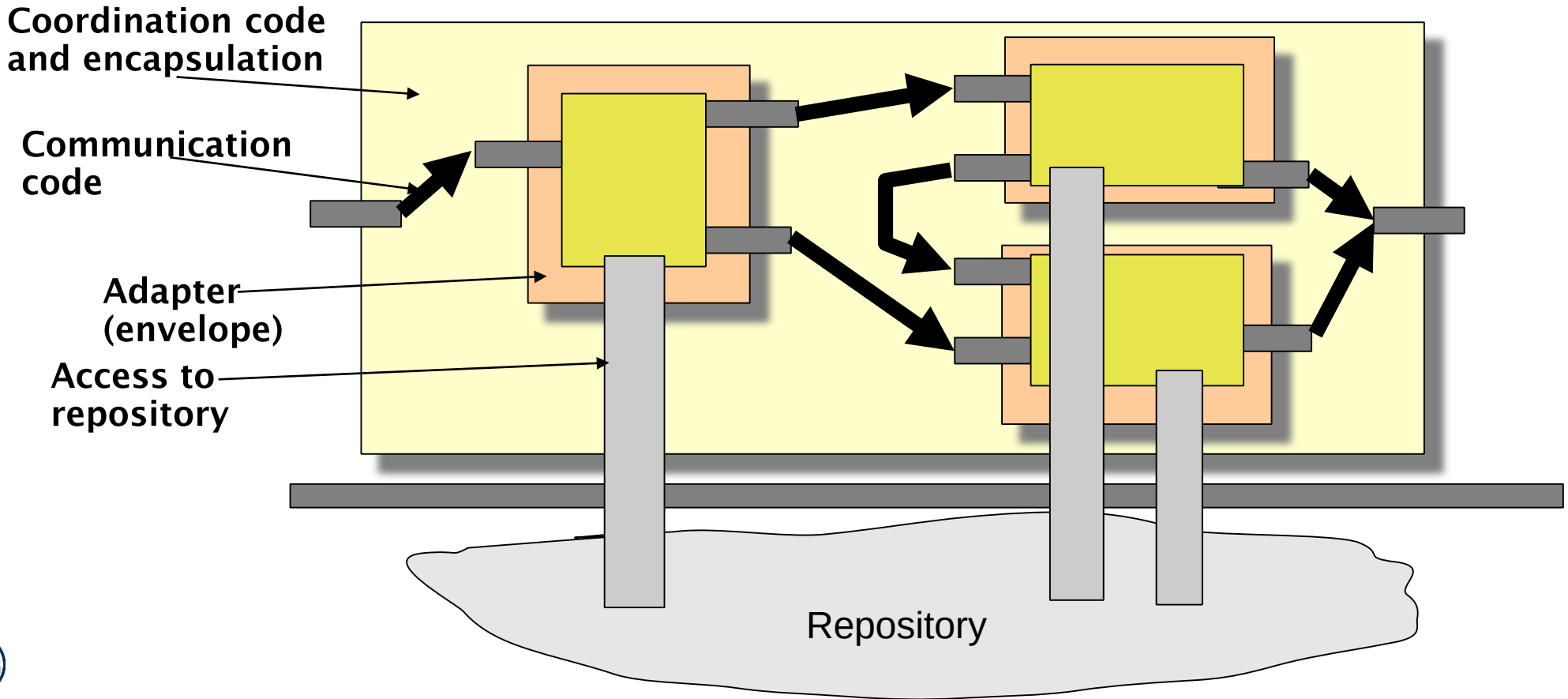


Hierarchical Components in the Repository Style (CoSy)



Hierarchical Repository Style

- CoSy generates for every component an adapter (envelope, container),
  - that maps the protocol of the component to that of the environment
  - Coordination, communication, encapsulation and access to the repository (memory connectors) are generated



Evaluation of CoSy

- CoSy is one of the few commercial architecture systems with professional support
  - CoSy realizes hierarchical repositories with memory connectors
  - The outer call layers of the compiler are generated from the ADL
  - Sequential and parallel implementation can be exchanged
  - There is also a non-commercial prototype [Martin Alt: On Parallel Compilation. PhD Dissertation Universität Saarbrücken])
  - Access layer to the repository is efficient (solved by generation of macros)
- Because of views a CoSy-compiler is very simply extensible
  - That's why it is expensive
  - Reconfiguration of a compiler within an hour

- UNICON supports
  - Components in C
  - Simple and user-defined connectors
- Design Goals
  - Uniform access to a large set of connections
  - Check of architectures (connections) with analysis tools should be possible
  - Both Graphics and Text
  - Reuse of existing legacy components

30.3.2 UNICON

Description of Components and Connectors in UNICON

- Name
- Interface (component) resp. protocol (connector)
- Type
  - component: modules, computation, SeqFile, Filter, process, general
  - connectors: Pipe, FileIO, procedureCall, DataAccess, RPC, RTScheduler
- Global assertions in form of a feature list (property list)
- Collection of
  - Players for components (for ports and port mappings for components of different nesting layers)
  - Roles for connectors
- The UNICON-compiler generates
  - Odin-Files from components and connectors. Odin is an extended Makefile
  - Connection code

Supported Player Types per Component Type

- Modules:
  - RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, ReadFile, WriteFile
- Computation:
  - RoutineDef, RoutineCall, GlobalDataUse,
- SharedData:
  - GlobalDataDef, GlobalDataUse,
- SeqFile:
  - ReadNext, WriteNext
- Filter:
  - StreamIn, StreamOut
- Process:
  - RPCDef, RPCCall
- Schedprocess:
  - RPCDef, RPCCall, RTLoad
- General:
  - All

Supported Role Types For Connector Types

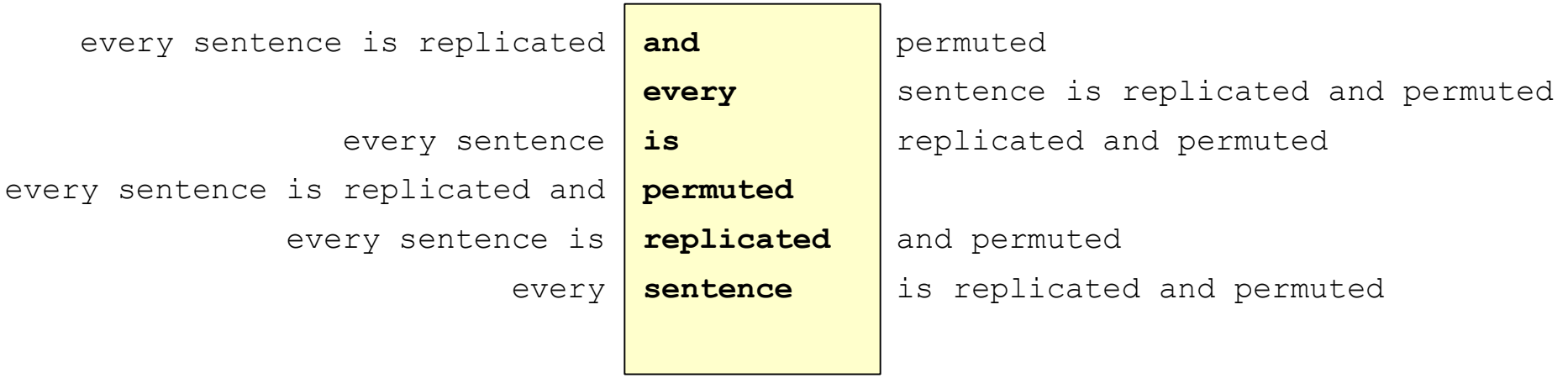
- Pipe:
  - Source fits to Filter.StreamOut, SeqFile.ReadNext
  - Sink fits to Filter.StreamIn, SeqFile.WriteNext
- FileIO:
  - Reader fits to modules.ReadFile
  - Readee fits to SeqFile.ReadNext
  - Writer fits to Modules.WriteFile
  - Writee fits to SeqFile.WriteNext
- ProcedureCall:
  - Definer fits to (Computation|Modules).RoutineDef
  - User fits to (SharedData|Computation|Modules).GlobalDataUse
- RPC
  - Definer fits to (Process|Schedprocess).RPCDef
  - User fits to (Process|Schedprocess).RPCCall
- RTScheduler
  - Load fits to Schedprocess.RTLoad

A Filter in UNICON


```
COMPONENT Reverser INTERFACE IS
TYPE Filter
PLAYER input IS StreamIn SIGNATURE ("line") PORTBINDING (stdin) END input
PLAYER output IS StreamOut SIGNATURE ("line") PORTBINDING (stdout) END output PLAYER
error IS StreamOut SIGNATURE ("line") PORTBINDING (stderr) END error
END INTERFACE

IMPLEMENTATION IS
/* Component instantiations are declared below. */
USES reverse INTERFACE Reverse
USES stack INTERFACE Stack
USES libc INTERFACE Libc
USES datause protocol C-shared-data

/* We will use <establish> statements for the procedure call connections (next page) */

/* Now for the configuration of connectors to players */
/* CONNECTs bind ports to roles */
CONNECT reverse._iob TO datause.user
CONNECT libc._iob TO datause.definer
END IMPLEMENTATION END Reverser
```

The KWIC Problem in UNICON

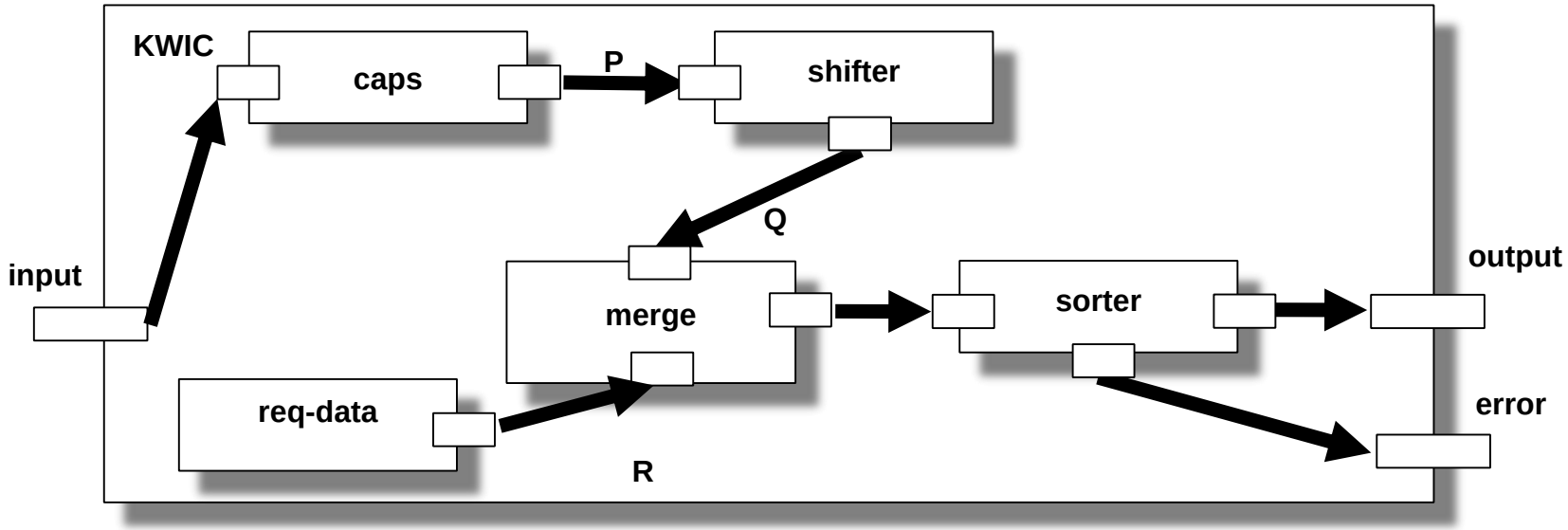
- Example from UniCon distribution:
- "Keyword in Context" problem (KWIC)
  - The KWIC problem is one of the 10 model problems of architecture systems
  - Proposed by Parnas to illustrate advantages of different designs [Parnas72]
  - For a text, a KWIC algorithm produces a permuted index
    - every sentence is replicated and permuted in its words, i.e., the words are shifted from left to right
    - every first word of a permutation is entered into an alphabetical index, the permuted index



The KWIC Problem in Unicon

- KWIC is a compound component KWIC
  - Works in a pipe-and-filter style
  - PLAYER definitions define ports of the outer component
    - stream input port input
    - and two output ports output and error
  - BIND statements connect ports from outer components to ports of inner components (delegation connectors)
  - USES definitions create instances of components and connectors
  - CONNECT statements connect connectors to ports at their roles

- Components
  - **caps**: replicates the sentences as necessary
  - **shifter**: permutes the generated sentences
  - **req-data**: provides some data to the merge component
  - **merge**: join, piping the generated data to the component
  - **sorter**: sorts the shifted sentences



KWIC in Text

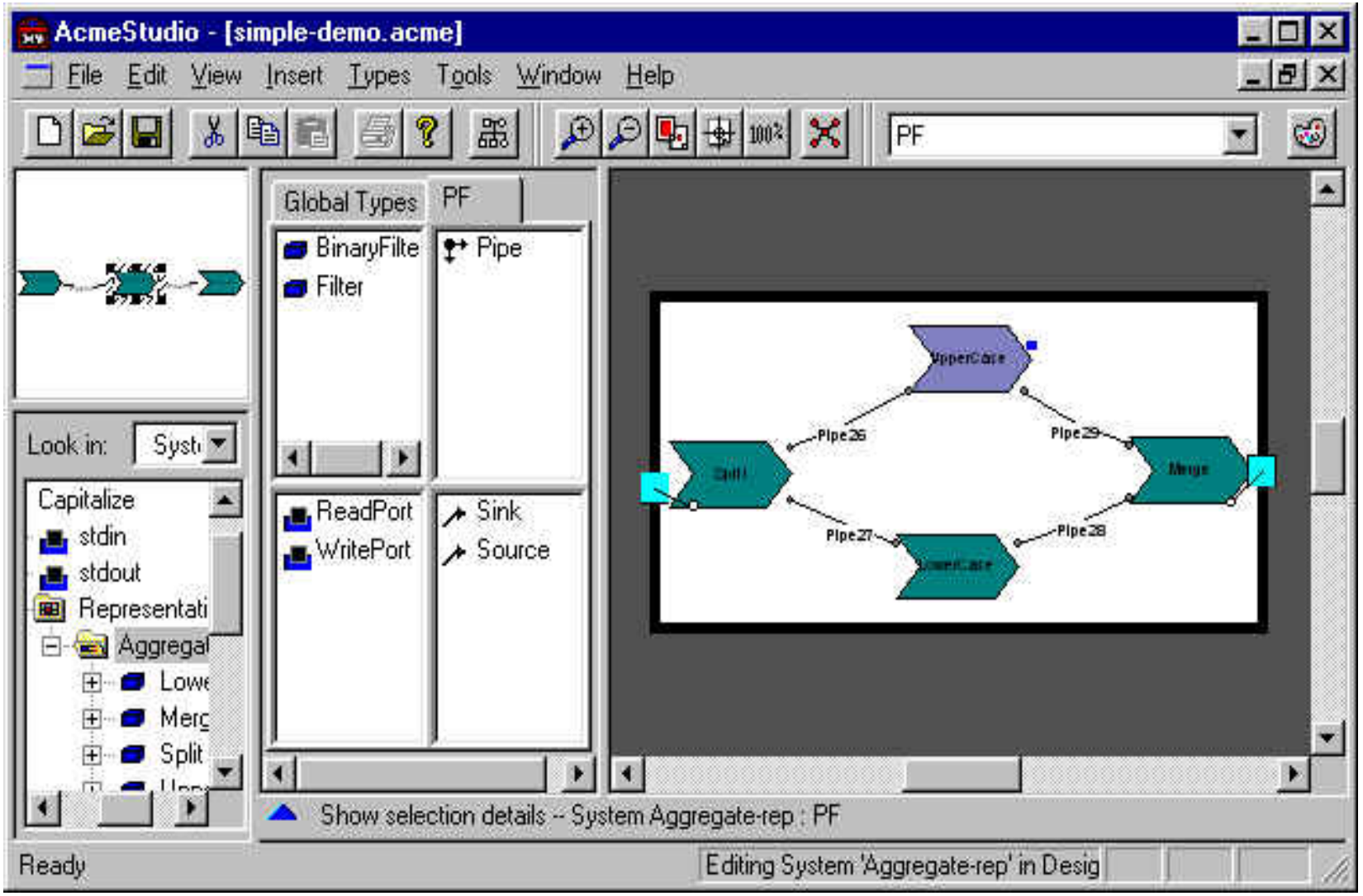

```
COMPONENT KWIC
   /* This is the interface of KWIC with in- and output ports */
  INTERFACE IS TYPE Filter
     PLAYER input    IS StreamIn    SIGNATURE ("line")
          PORTBINDING (stdin)    END input
     PLAYER output IS StreamOut SIGNATURE ("line")
          PORTBINDING (stdout) END output
  END INTERFACE
  IMPLEMENTATION IS
     /* Here come the component definitions */
     USES caps          INTERFACE upcase         END caps
     USES shifter       INTERFACE cshift         END shifter
     USES req-data INTERFACE const-data END req-data
     USES merge       INTERFACE converge      END merge
     USES sorter      INTERFACE sort           END sorter
     /* Here come the connector definitions */
     USES P PROTOCOL Unix-pipe END P
     USES Q PROTOCOL Unix-pipe END Q
     USES R PROTOCOL Unix-pipe  END R
```

```
/* Here come the connections */
      BIND         input              TO caps.input
      CONNECT caps.output      TO P.source
      CONNECT shifter.input     TO P.sink
      CONNECT shifter.output  TO Q.source
      CONNECT req-data.read TO R.source
      CONNECT merge.in1         TO R.sink
      CONNECT merge.in2         TO Q.sink
      /* Syntactic sugar is provided for complete connections */
      ESTABLISH Unix-pipe WITH
         merge.output AS source
         sorter.input    AS sink
      END Unix-pipe
      BIND output TO sorter.output
   END IMPLEMENTATION
END KWIC
```

30.3.3 Architectural Style Rules with Aesop and ACME

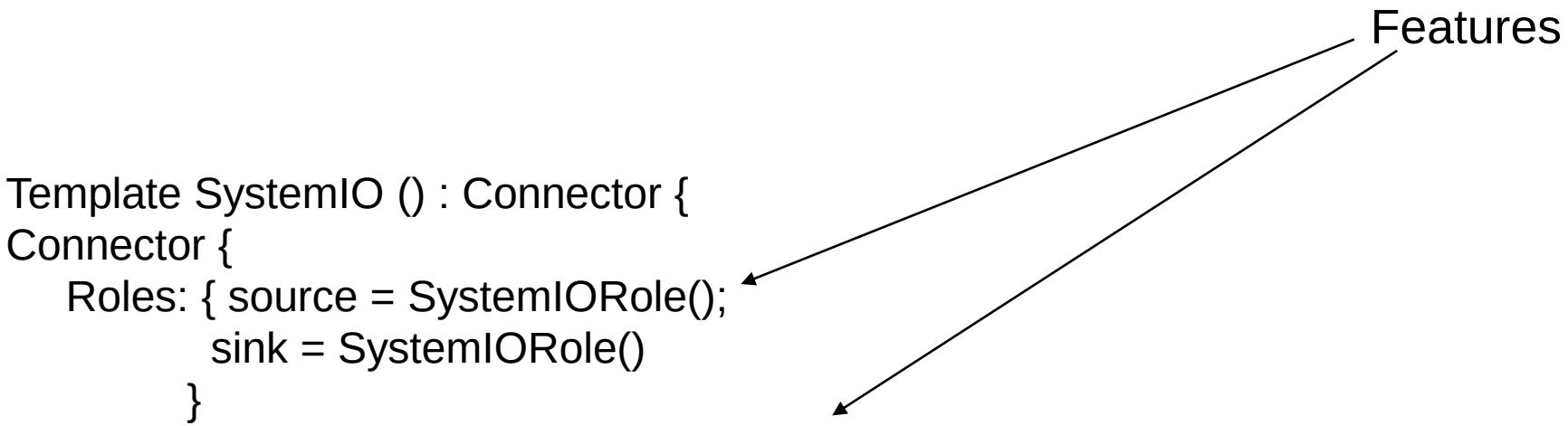
- Connectors are first class language elements, i.e., can be defined by users
  - Connectors are classes which can be refined by inheritance from system connectors
- Aesop supports the definition of architectural styles with *fables*
  - Architectural styles obey rules (logic constraints)
  - Editor for architectural styles edits *design rules*
    1. A design rule is a code fragment by which a class extends a method of a super class. Has:
       - A pre-check that helps control whether the method should be run or not.
       - A post-action
- Design Environments
  - A design environment tailored to a particular *architectural style*.
    1. It includes a set of policies about the style
    2. A set of tools that work in harmony with the style, visualization information for tools
    3. If something is part of the formal meaning, it should be part of a style

ACME Studio as Graphic Environment



ACME (CMU)

- ACME offers an exchange language (exchange format), to which different ADL can be mapped (UNICON, Aesop,..).
- It consists of abstract syntax specification
  - Similar to feature terms (terms with attributes).
  - With inheritance



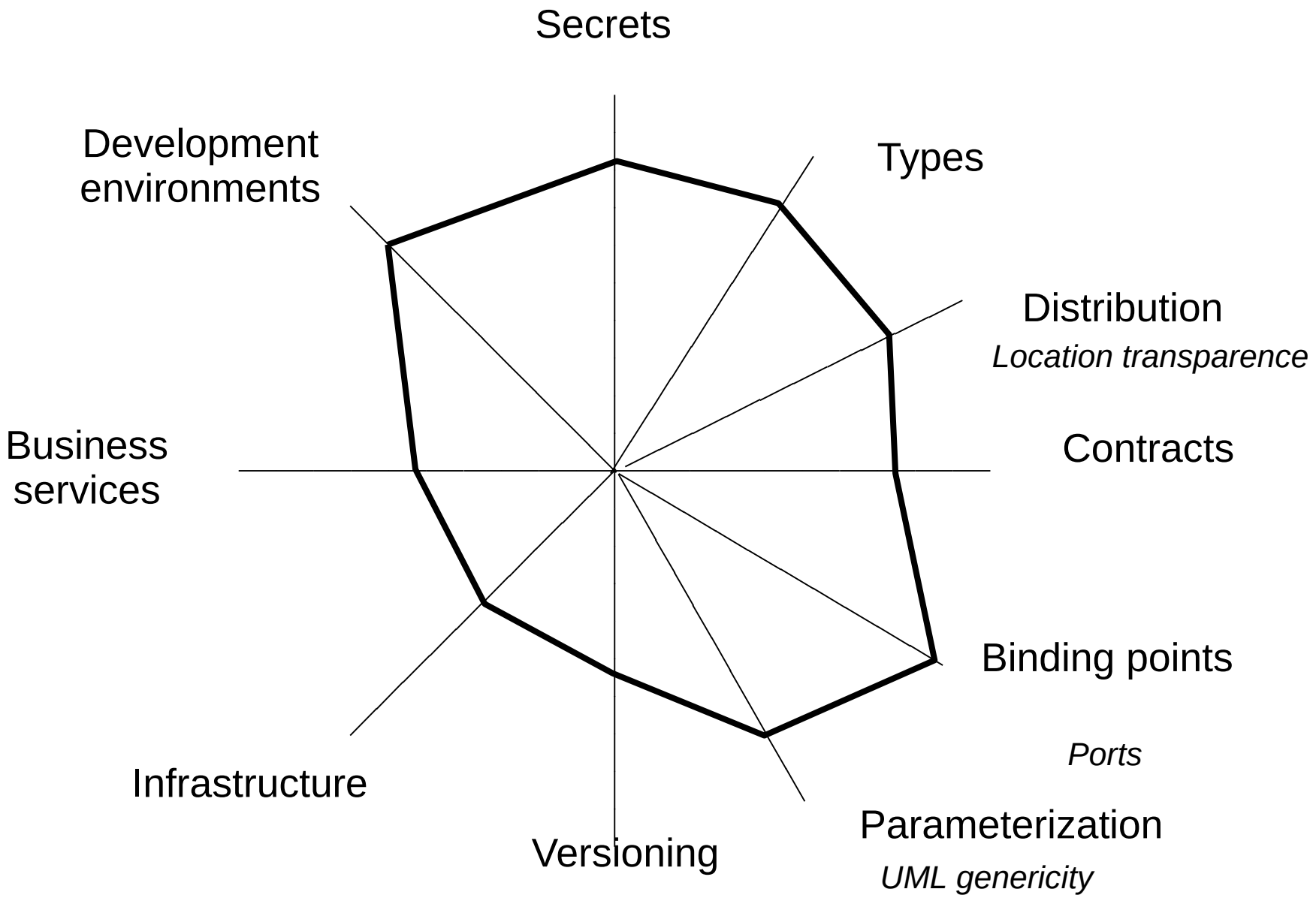
30.4 Architecture Systems: Evaluation

- How to evaluate architecture systems as composition systems?
  - Component model
  - Composition technique
  - Composition language

ADL: Mechanisms for Modularization

- Component concepts
  - Clean language-, interfaces and component concepts
  - New type of component: connectors
  - Clean documentation
  - Secrets: Connectors hide
    - Communication transfer
    - Partner of the communication
    - Distribution
- Parameterisation: depends on language
- Standardization: still pending

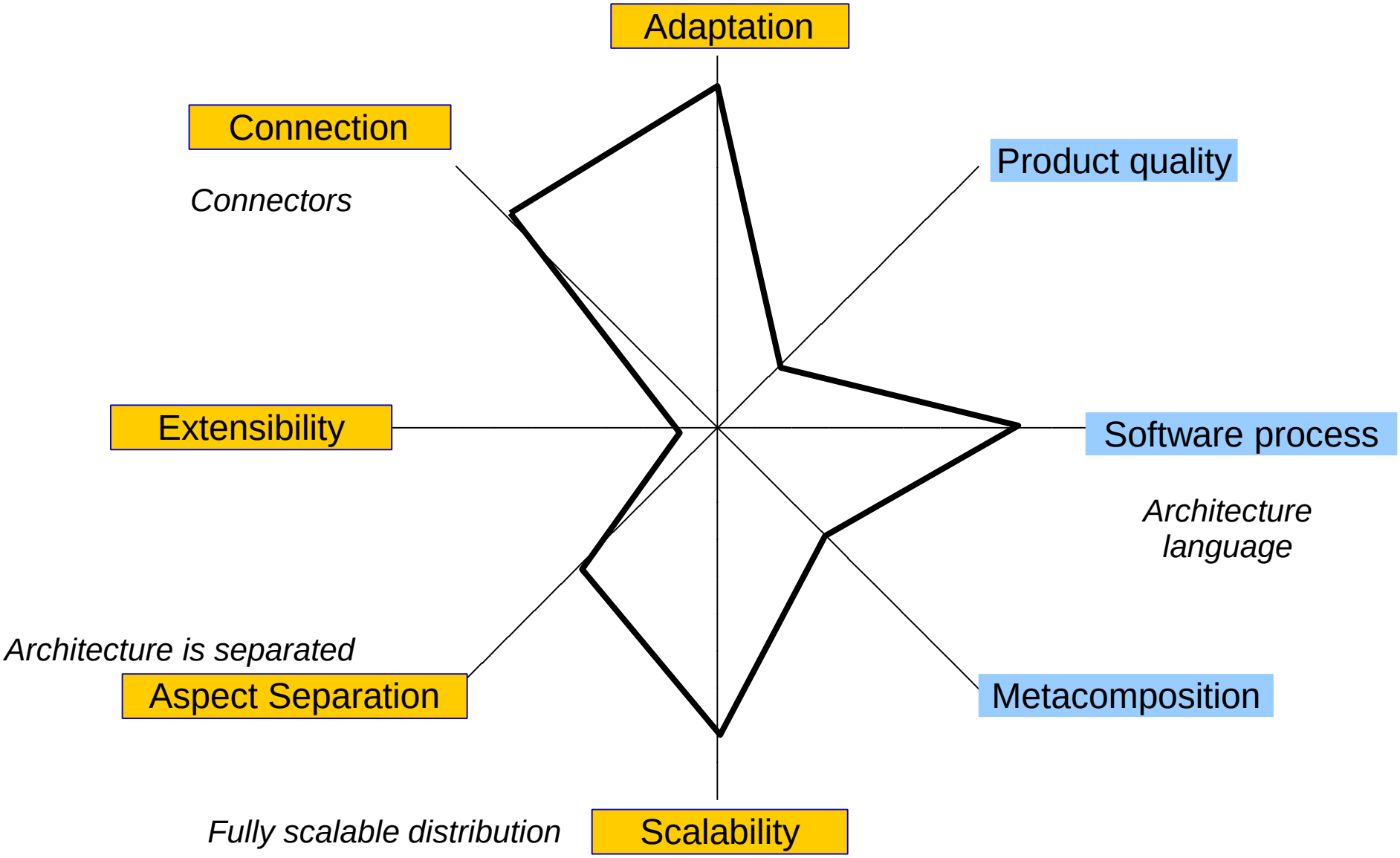
Architecture Systems - Component Model



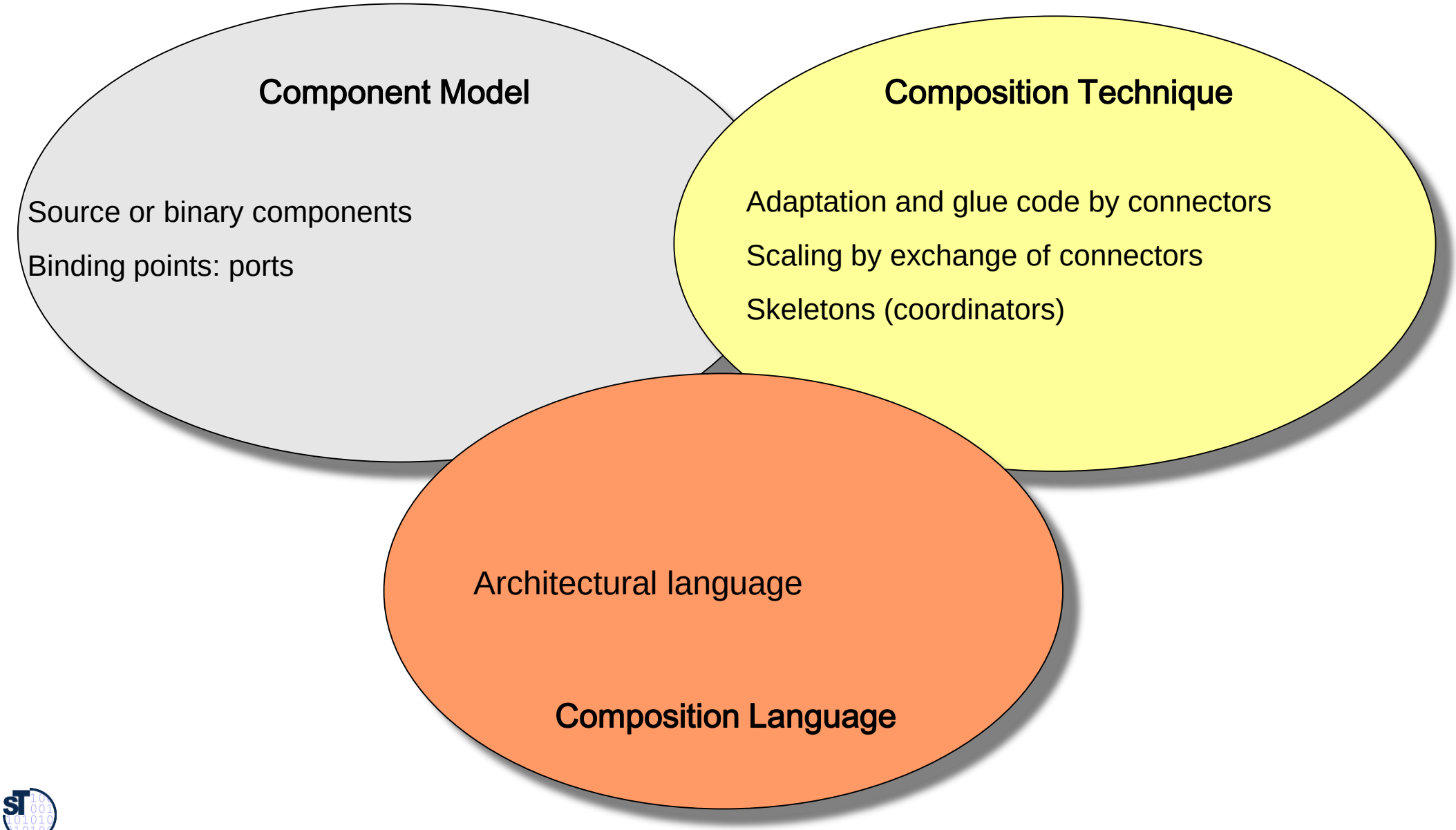
ADL: Mechanisms for Adaptation

- Connectors generate glue code: very good!
  - Many types of glue code possible
  - User definable connectors allow for specific glue
  - Tools analyze the interfaces and find about the necessary adaptation code automatically
- Mechanisms for aspect separation. At least 3 aspects are distinguished:
  - Architecture (topology and hierarchy)
  - Communication carrier
  - Application
- No weaving
  - The aspects are not weaved, but encapsulated in glue
- An ADL-compiler is only a rudimentary weaver

Architecture Systems – Composition Technique and Language



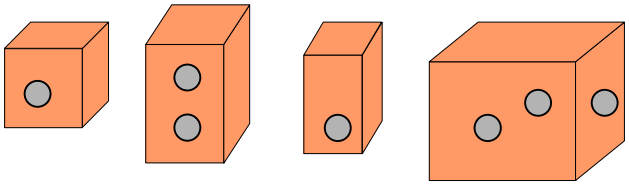
Architecture Systems as Composition Systems



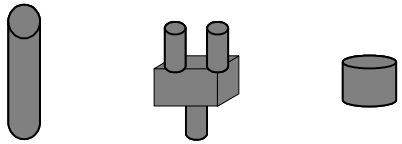
What Have We Learned?

- Architecture systems provide an important step forward in software engineering
  - For the first time, *architecture* becomes visible
- Concepts can be applied in UML already today
- Architectural languages are the most advanced form of blackbox composition technology so far

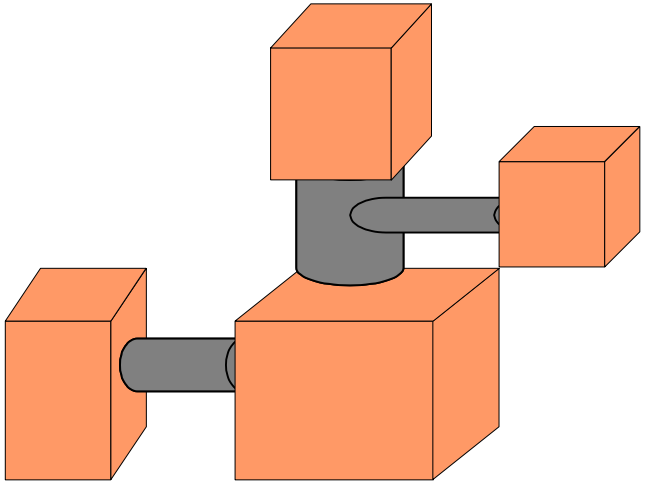
Blackbox Composition in an Architecture System



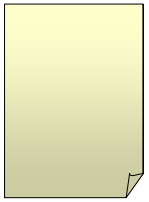
Components



Connectors - Coordinators



Component-based applications



Composition program

How the Future Will Look Like

- Metamodels of architecture concepts (with MOF in UML) will replace architecture languages
  - The attempts are promising which describe architecture concepts with UML
  - Example: EAST-ADL, an ADL for the automotive domain:
  - http://en.wikipedia.org/wiki/EAST-ADL
- Web service languages have taken over the role of ADL in practice
- More aspects can be distinguished (see later)
  - Leading to more MOF-based extensions of UML
  - We should think more about general software composition mechanisms
  - Adaptation by glue is only a simple way of composing components (... see invasive composition)

The End