13. Architecture Systems

Prof. Dr. Uwe Aßmann

Florian Heidenreich

Technische Universität Dresden

Institut für Software- und Multimediatechnik

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

Version 12-0.3, April 10, 2012

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://citeseer.ist.psu.edu/hofmeister99describing.html

Literature

- Shaw, M., Garlan, D. Software Architecture – Perspectives for an Emerging Discipline. Prentice-Hall,1996. Nice Introduction.
- http://www.cs.cmu.edu (Shaw, Garlan)
- 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/cosy

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://citeseer.ist.psu.edu/shaw95abstractions.html
- D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering, S. 717--734, Sept. 1995. (RAPIDE)
- http://www.doc.ac.ic.uk (Darwin)
- 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.

- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004.

The Ladder of Composition Systems

Software Composition Systems	Composition Language	*Invasive Composition Piccola Gloo*
Aspect Systems	Aspect Separation Crosscut graphs	*Aspect/J AOM*
View Systems	Composition Operators	*Composition Filters Hyperspaces*
Architecture Systems	Architecture as Aspect Connectors	*Darwin COSY BPMN ACME*
Classical Component Systems	Standard Components Reflection	*.NET CORBA Beans EJB*
Object-Oriented Systems	Objects as Run-Time Components	*C++ Java*
Modular Systems	Modules as Compile-Time Components	*Shell scripts Modula Ada-85*

13.1. Separation of Concerns

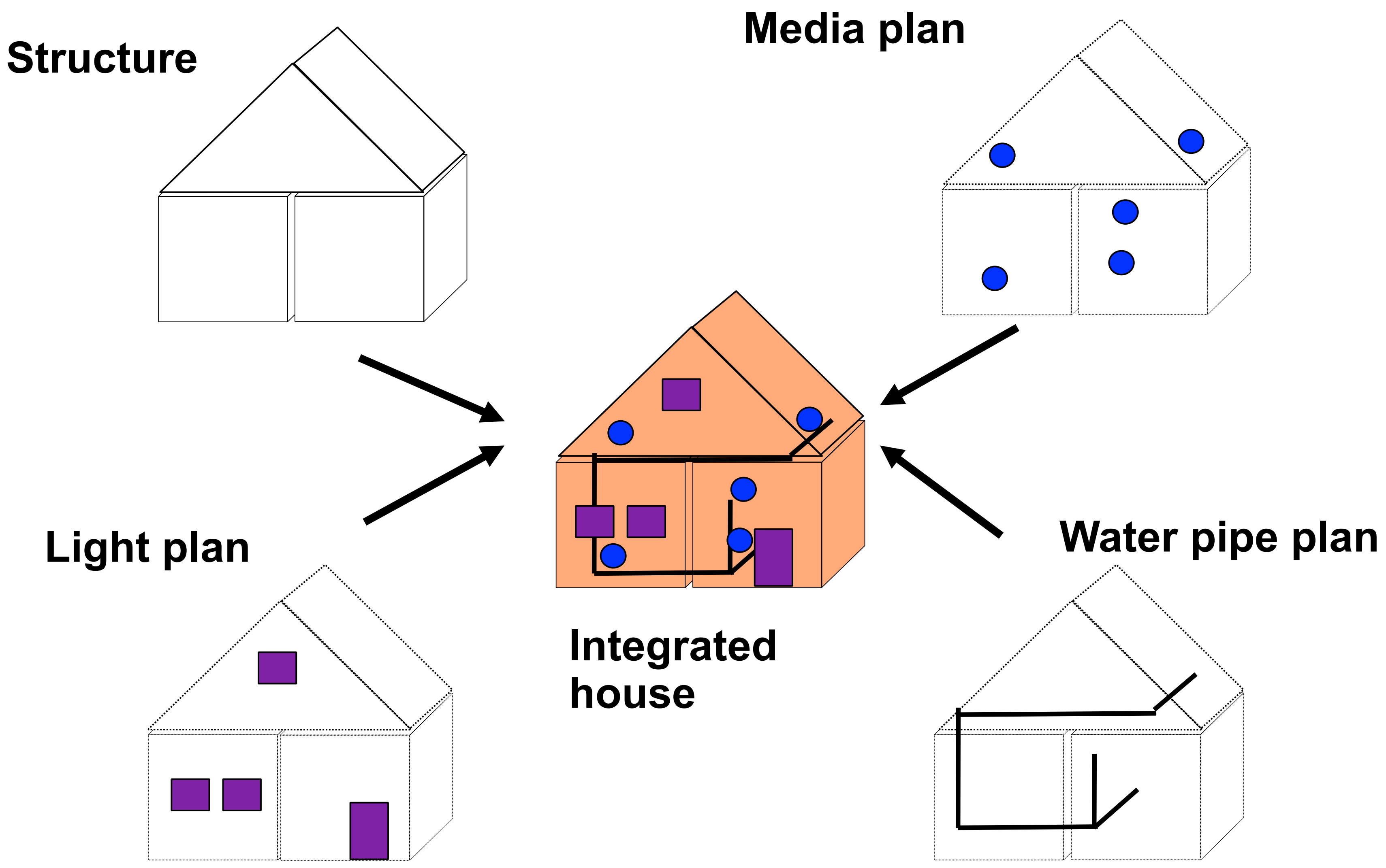
Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. ... 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]

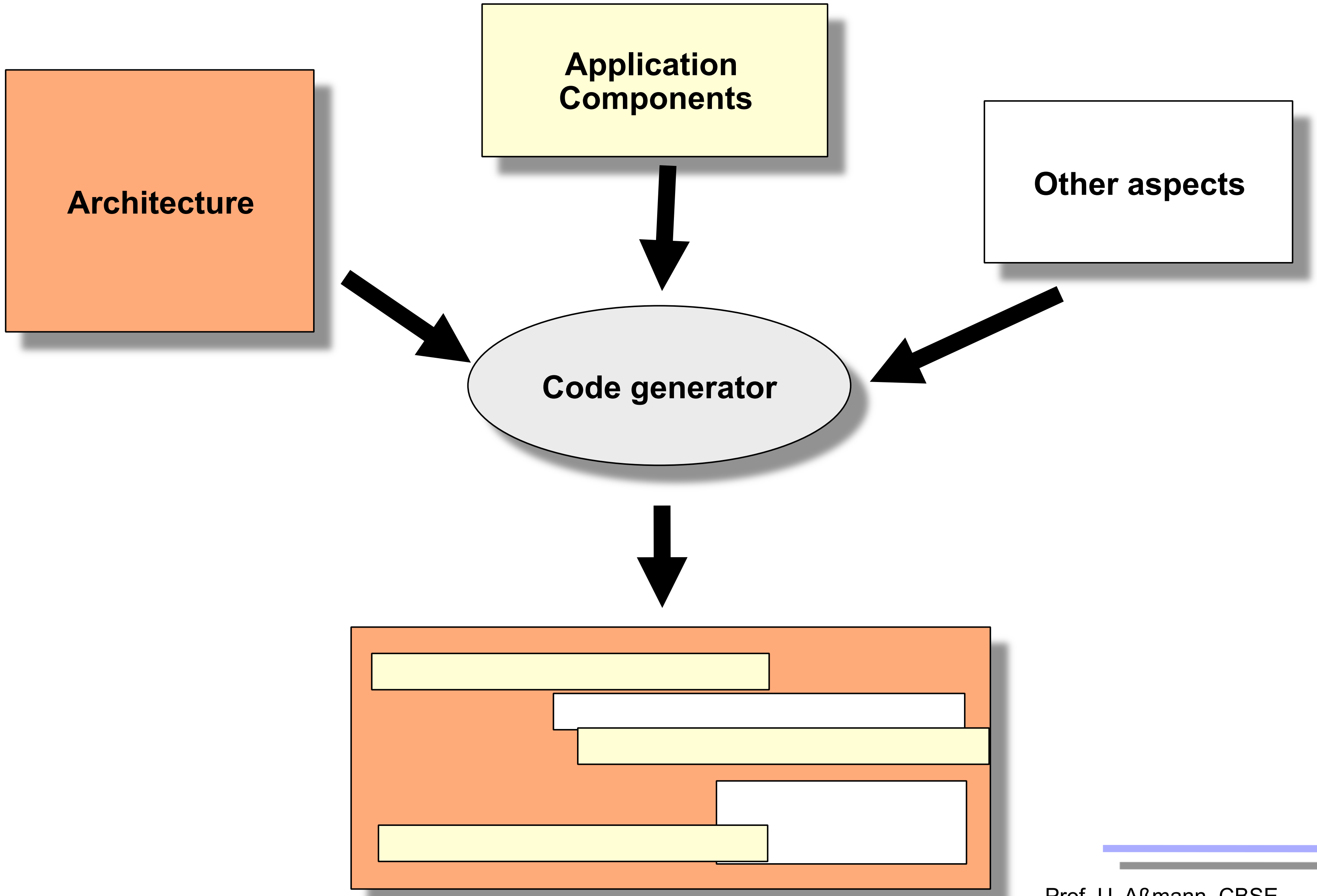
A Basic Rule for Design

- ... 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



The 4-View to Software Architecture

- [Hofmeister/Sony/Nord. Applied Software Architecture]
- 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*
    - specifies non-functional features as performance, reliability, fault tolerance, parallelism, division in processes.
  - *development view*
    - 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
  - But until now, no architecture system supports these 4 levels; most of them tackle the logical view

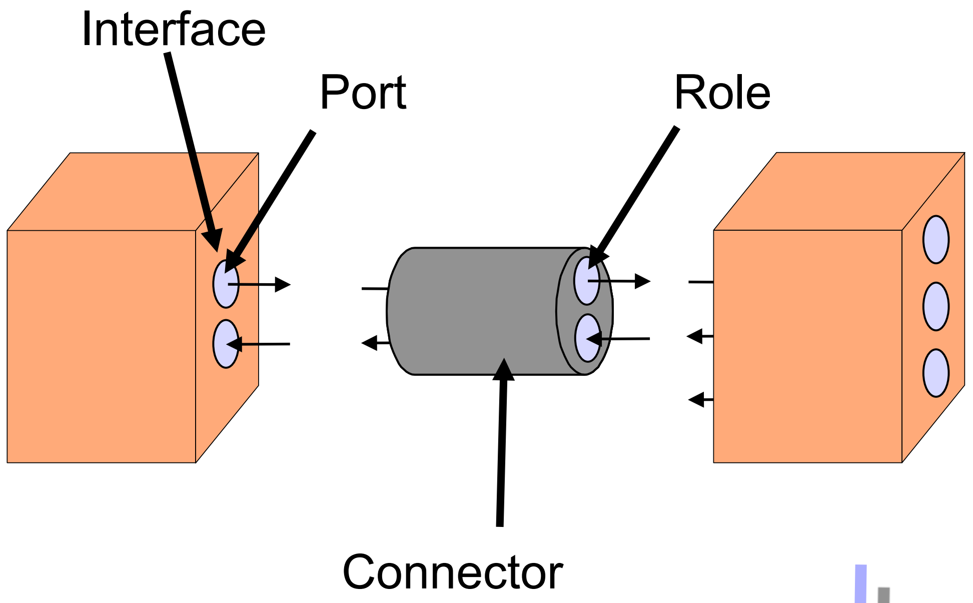
Architecture Systems as Automated Architectural Views

- **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
    - Topology (with whom?)
    - Carrier (how?)
  - Scalability (distribution, binding time with dynamic architectures)
  - Architectural skeletons as composition operators
- Composition language: Architecture Description Language (ADL)

13.2 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
    - .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)



13.2.1 Ports

Abstract Binding Points: Data Ports

- Ports are
  - **Provided** or **required**
  - **Synchronous** or **asynchronous** (partner has to wait or not)
  - **Singular** or **continuous** (communication can take place once or many times)
  - **Atomic** or **composite**

- 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

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)]
```

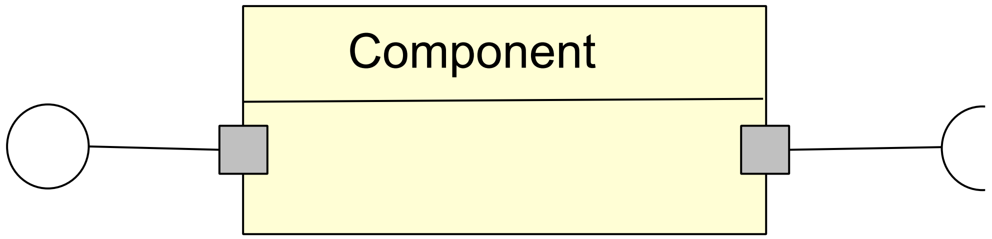
- A **property service** is a synchronous singular data service to access component attributes, i.e., a simple tuple of in and out ports

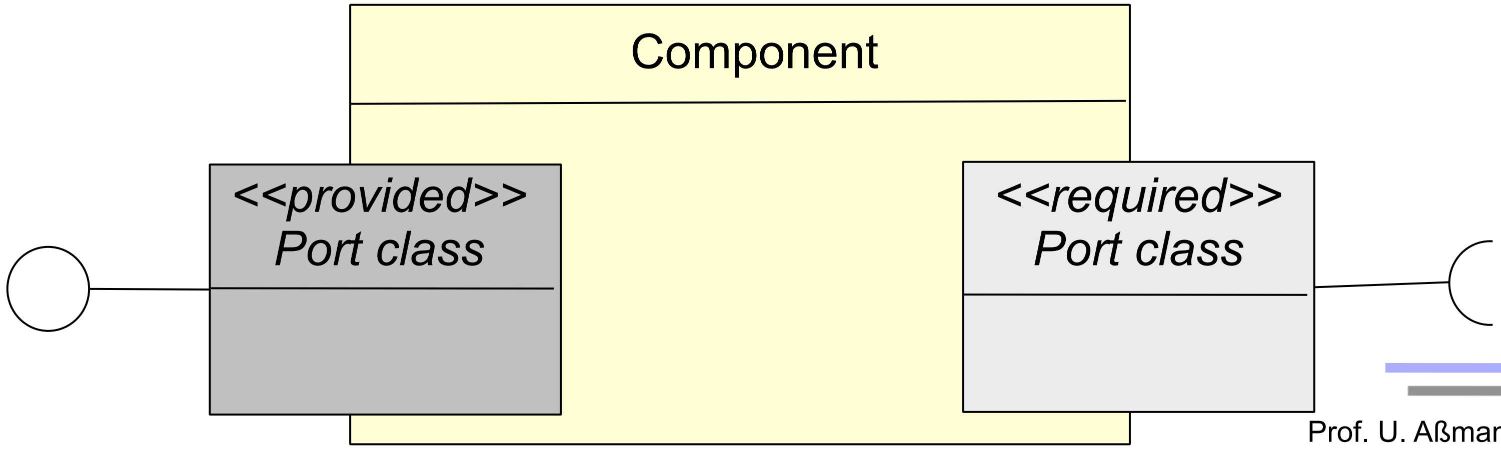
```
[in(data), out(data)]
```

Gates (Port Objects)

- So far, provided ports were
  - Data ports (simple in, out, set, get procedures)
  - Service ports (structured ports and procedures)

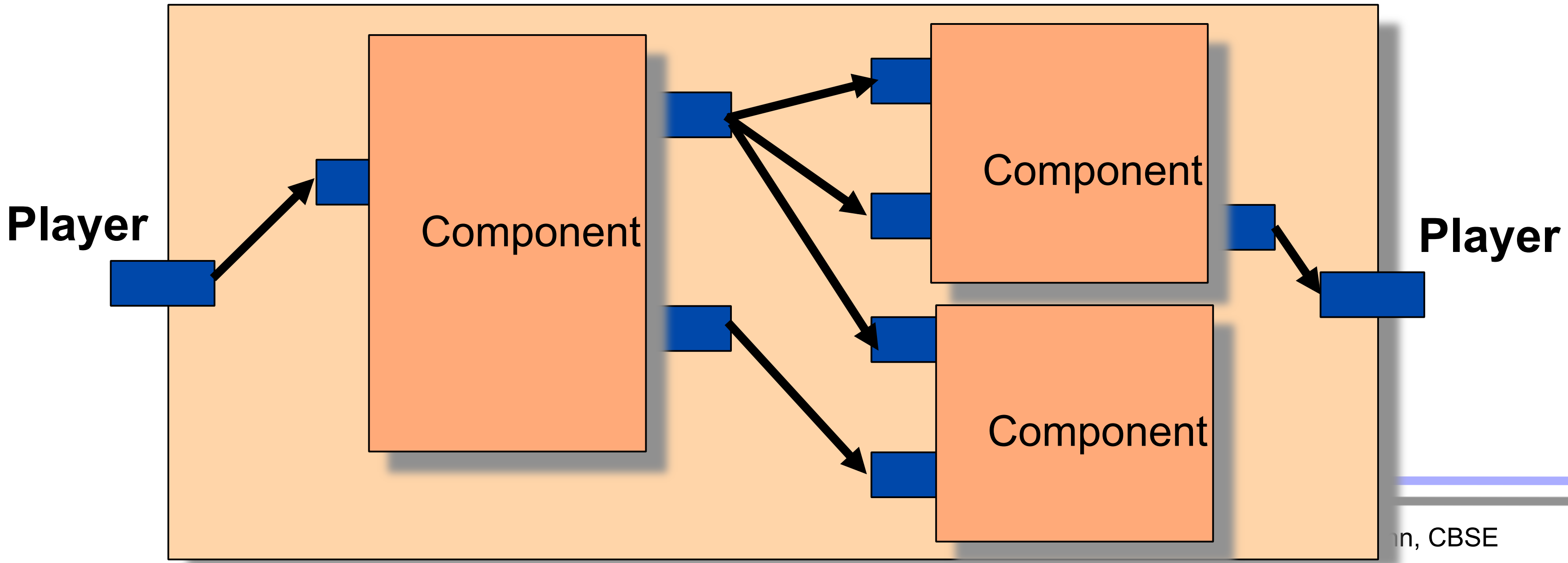
- Ports can also be **gates**, port objects, with interfaces (roles)





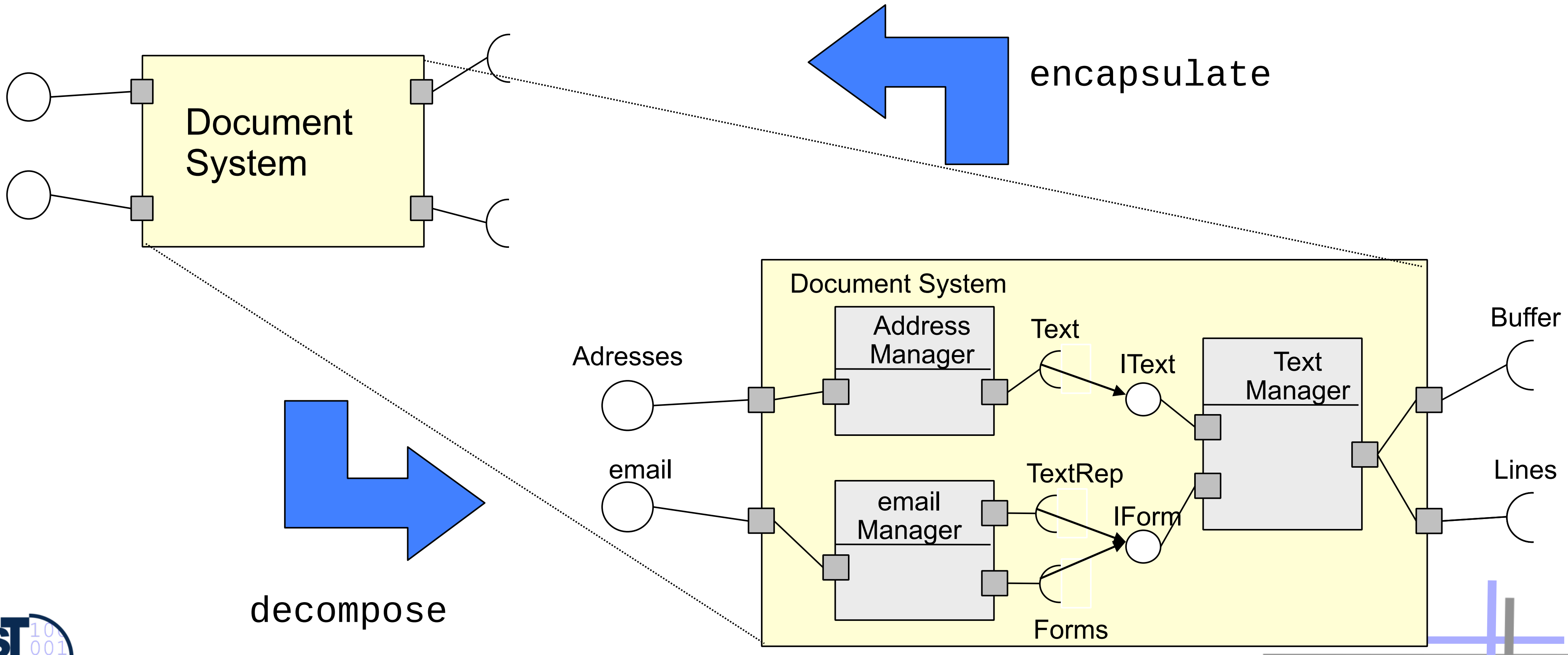
Hierarchic Architectures with Encapsulation

- 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** so that architectures become reducible structures (fractal-like zoom-in/out)
  - The operation "encapsulate" hides encapsulated components in an outer component
  - 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

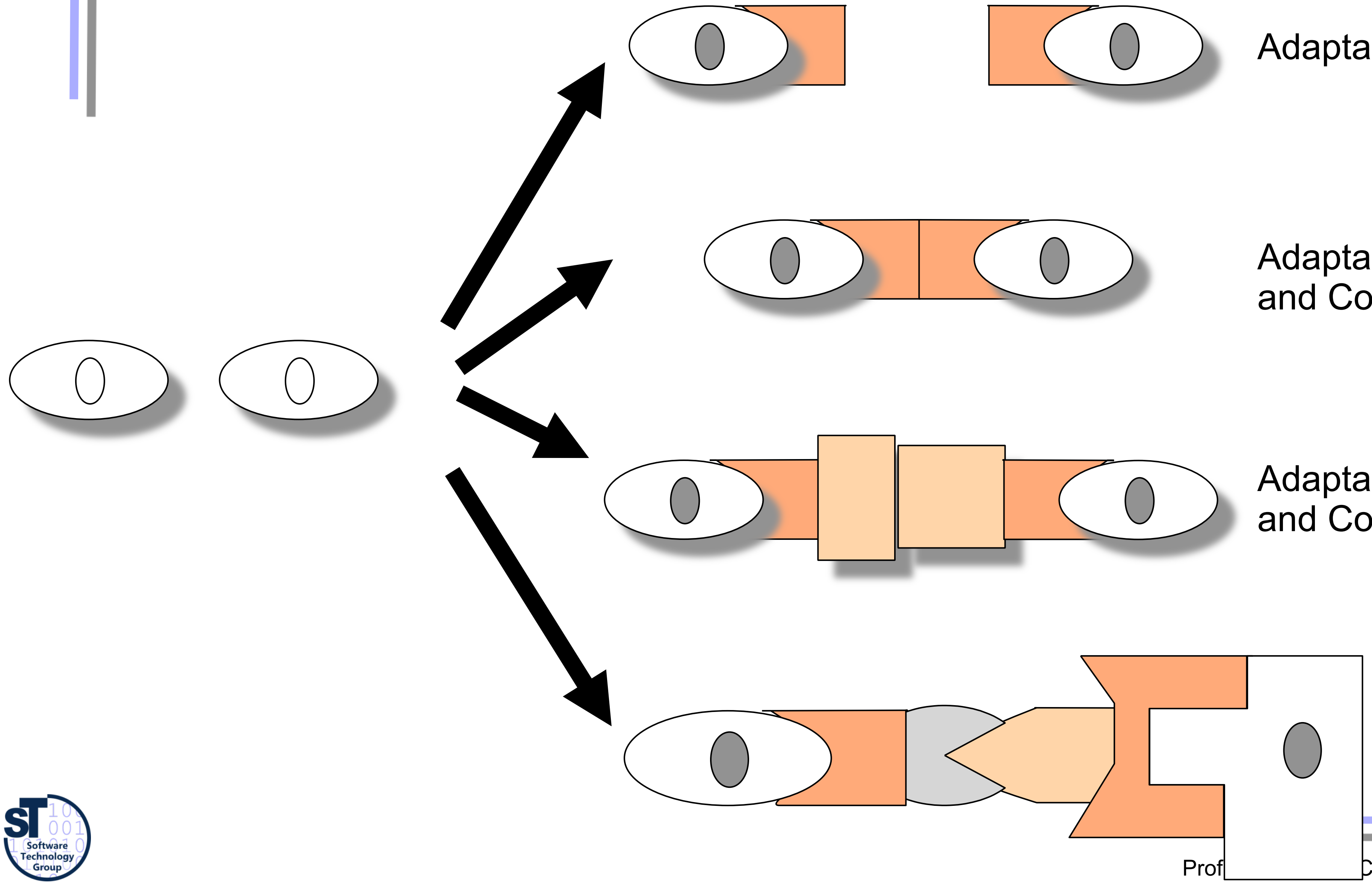
- In most component models, components are nested, i.e., are bobs.
- Nesting is indicated by *aggregation* and *part-of relationship*.
- Nesting is introduced by an encapsulation operator `encapsulate`.



13.2.2 Connectors

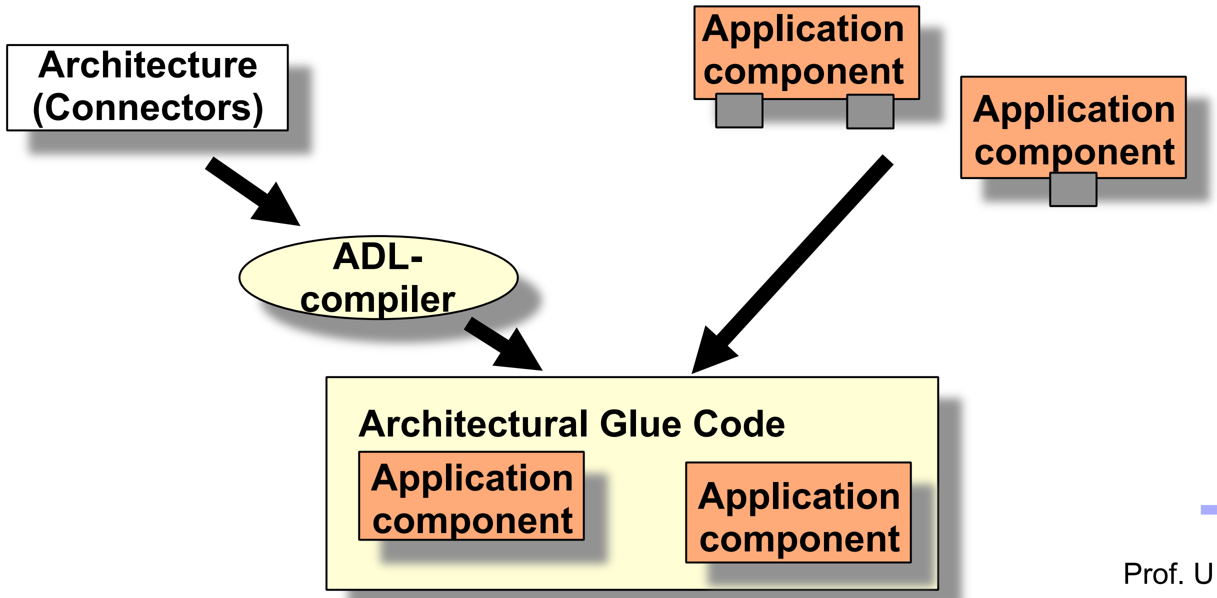
Connectors Provide Adaptation, Glueing and Connection

- Connectors can stack adapters onto each other

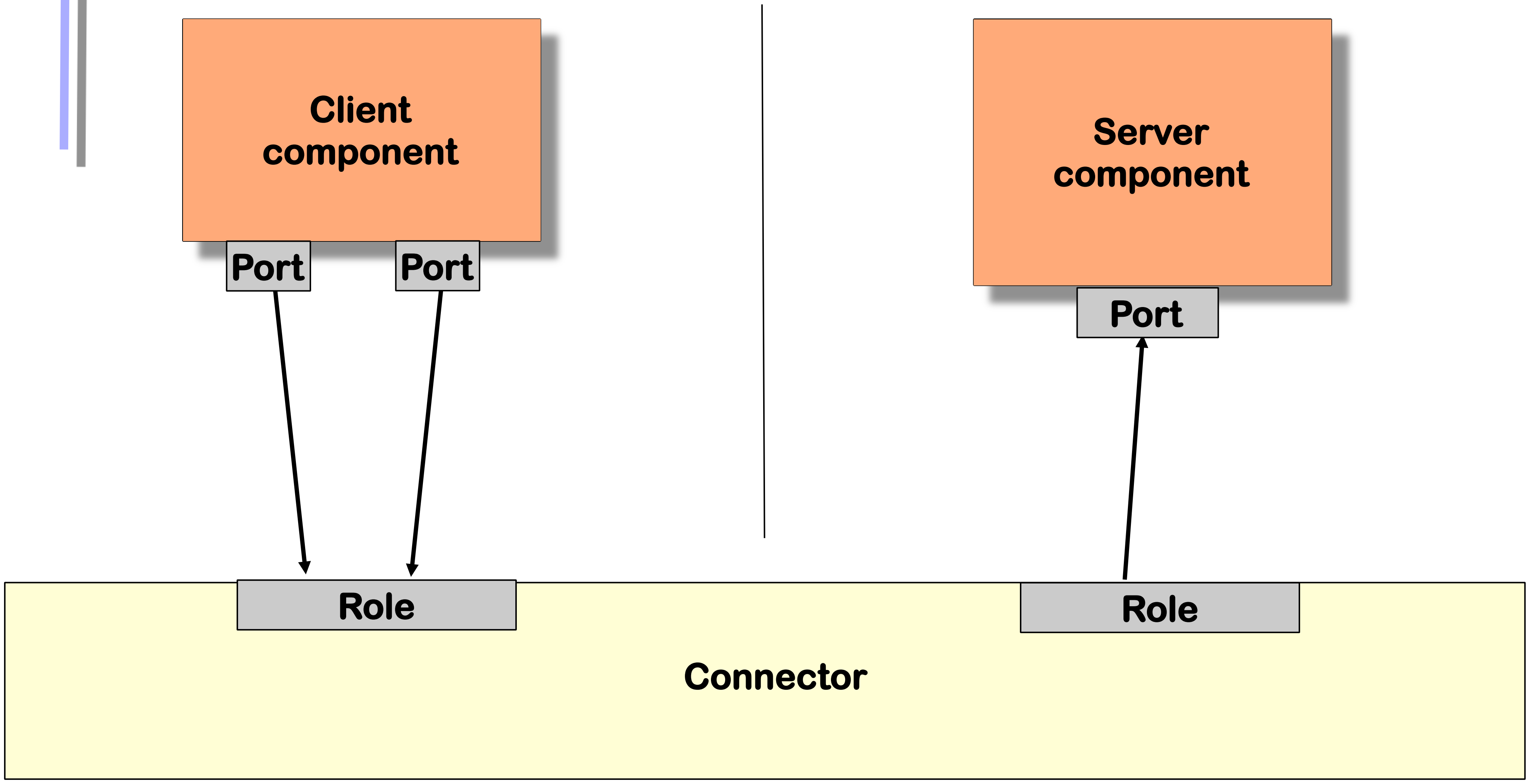


Connectors Generate Architectural Code

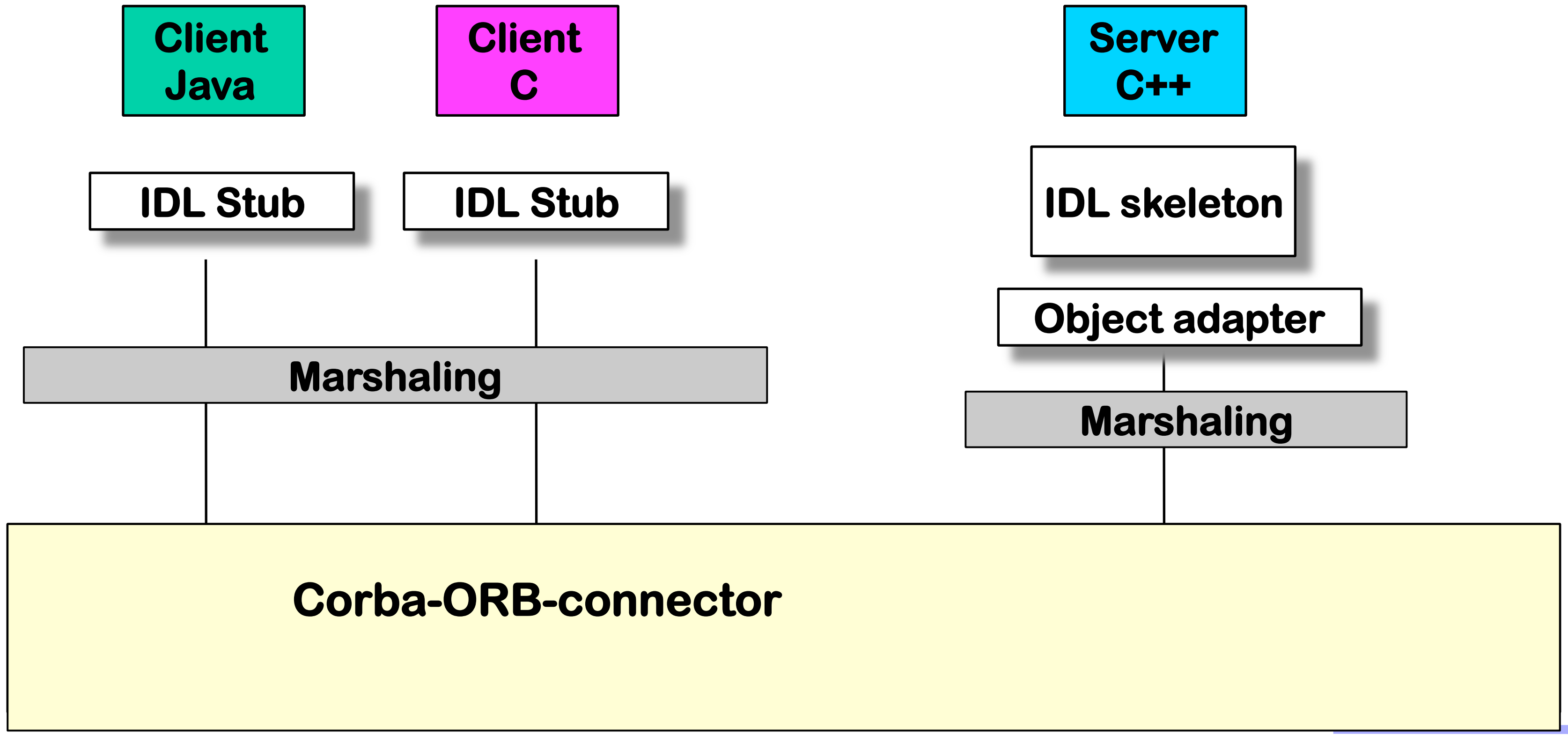
- Glue- and adapter code from connectors, skeletons, and ADL-specifications
  - Mapping of the protocols of the components to each other
- Simulations of architectures:
  - Test dummies and mocks (dummies with protocol machines)
  - The architecture can be created first and tested standalone
  - Analysis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



Connectors are Abstract Communication Buses

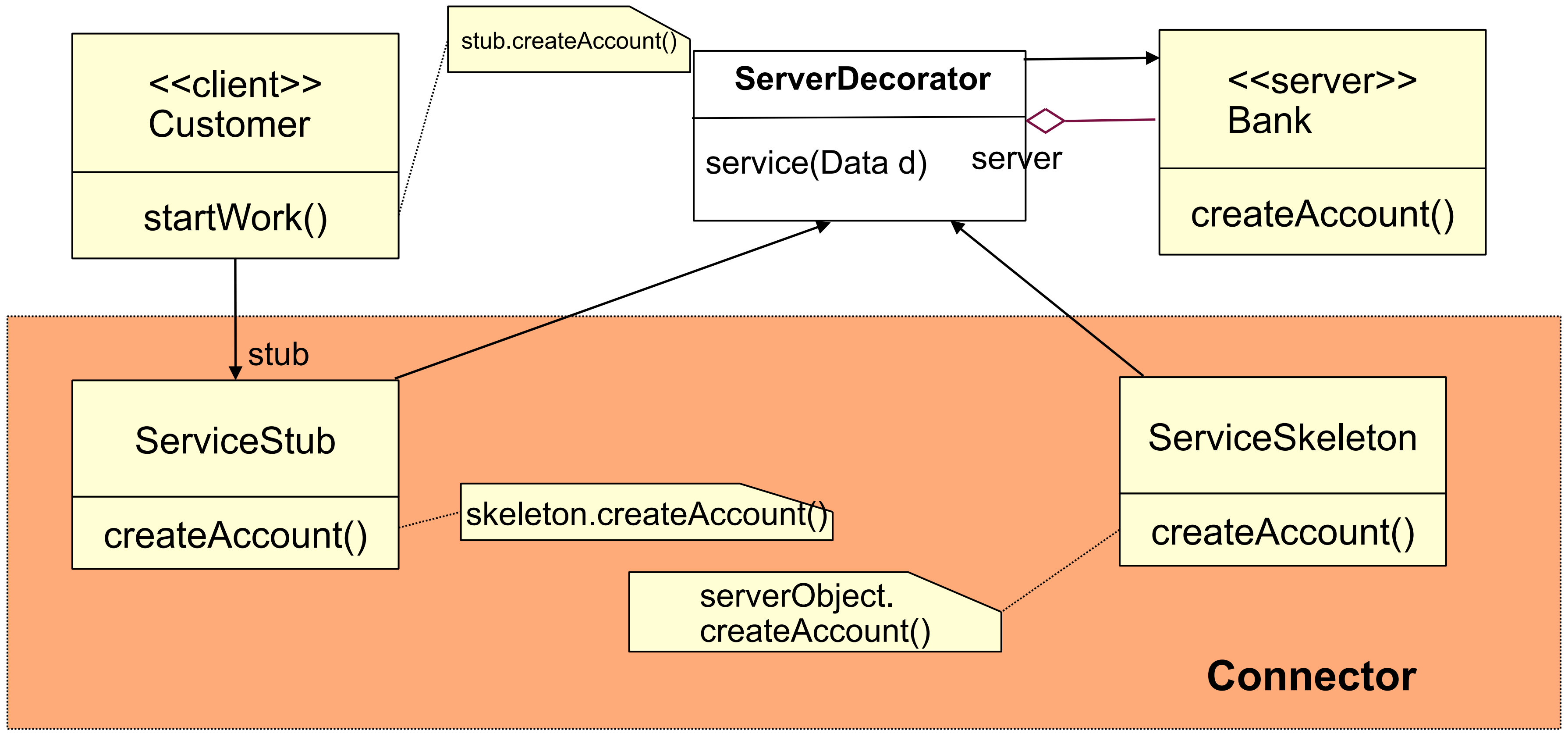


But we know that already from Corba



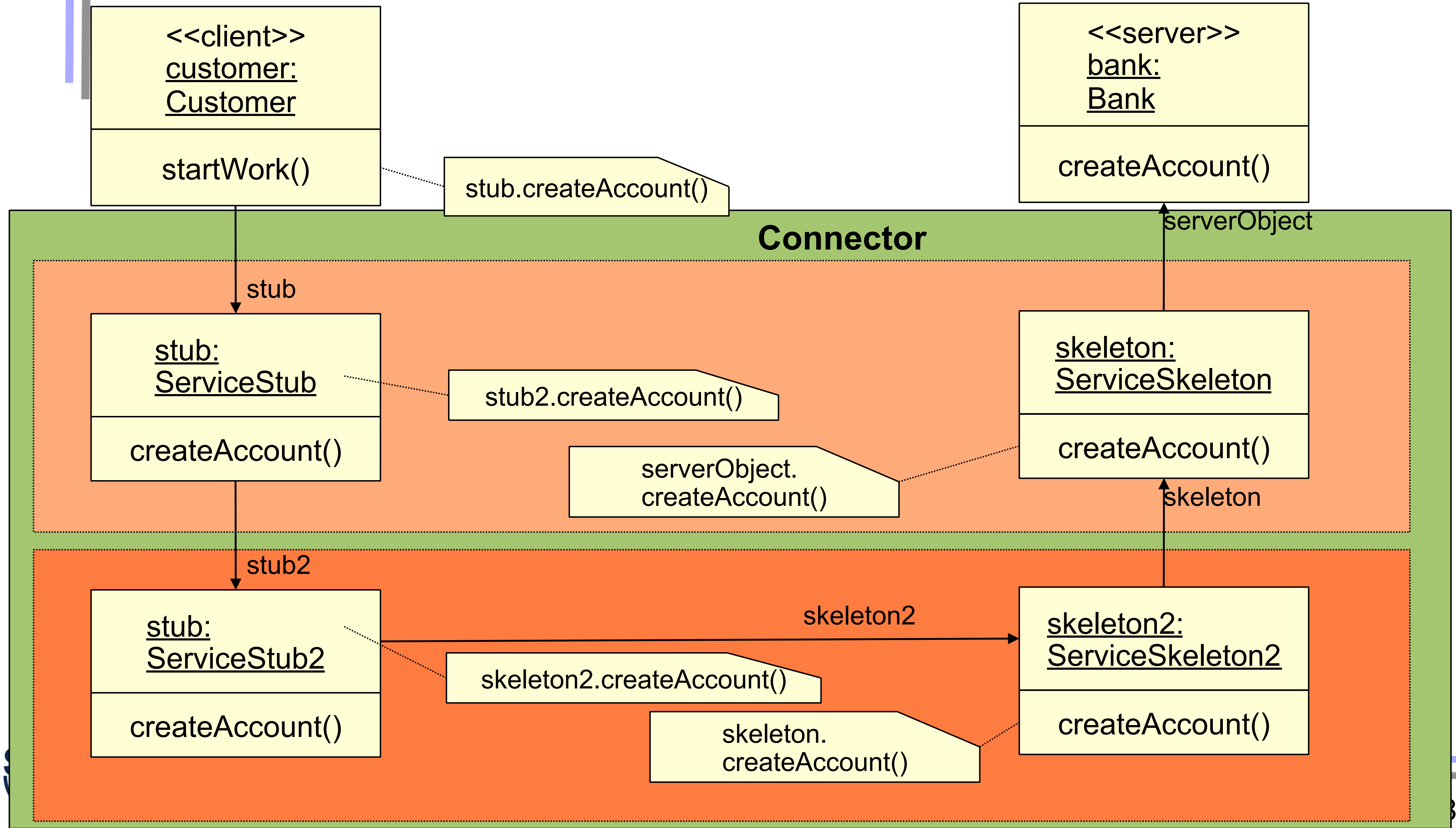
The Connector Pattern (Rept.)

- Client and server are connected via a layer of stubs and skeletons (the *connector*)
- The connector consists of two decorators of the server
- Decorator chain is inherited



More Layers are Possible in a Decorator-Connector

- More decorators can be stuffed into the connector in additional layers:

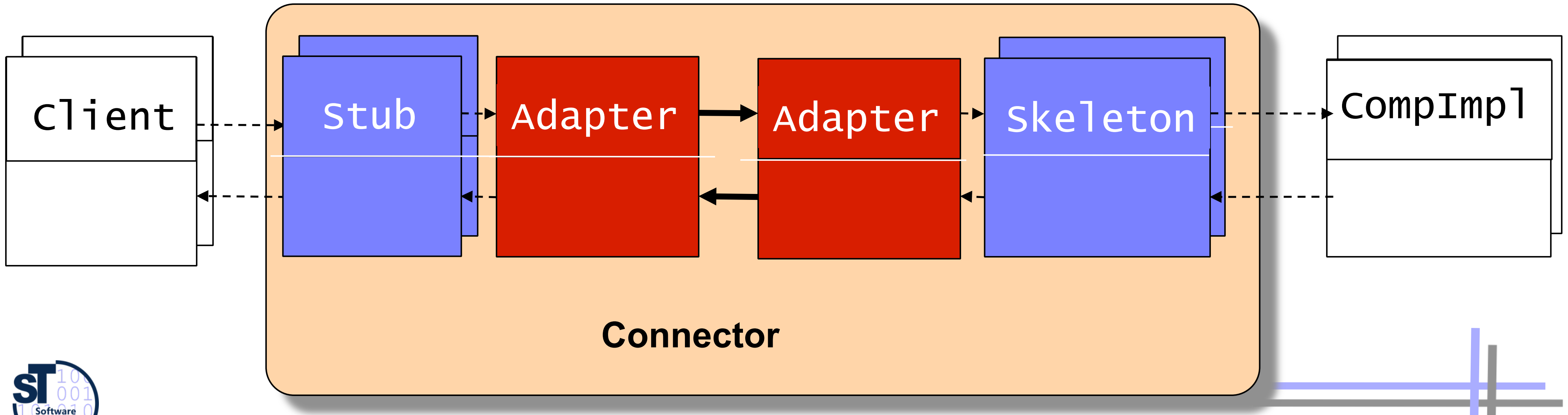


CORBA is a Simple Architecture System with Restricted Connectors

➢ Corba:	➢ Architecture System:
▸ Client and service components	▸ Components
▸ ORB client side, server side	▸ Connectors
▸ Marshalling, proxy, Stub, Skeleton, Object Adapter	▸ Roles
▸ Interfaces in IDL (not abstracted to ports)	▸ Ports
▸ static call	▸ procedure call connector (also distributed)
▸ dynamic call	▸ dynamic reconfigurable connectors (e.g., in Darwin)
▸ connectors always binary	▸ connectors n-ary
▸ Events, callbacks, persistence as services (cannot be exchanged to other communications)	▸ All these as connectors (can be exchanged to other communications)

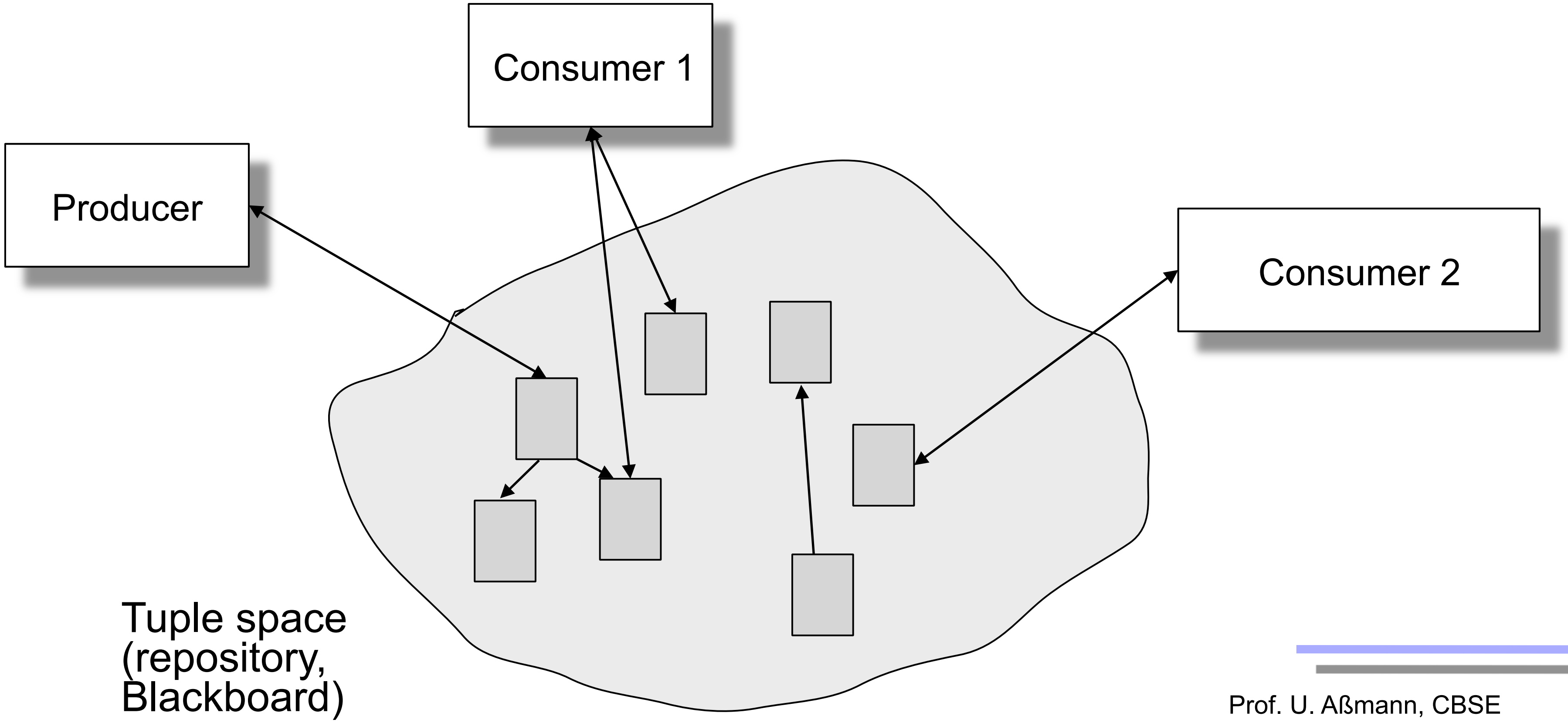
Most Commercial Component Systems Provide Restricted Forms of Connectors

- 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



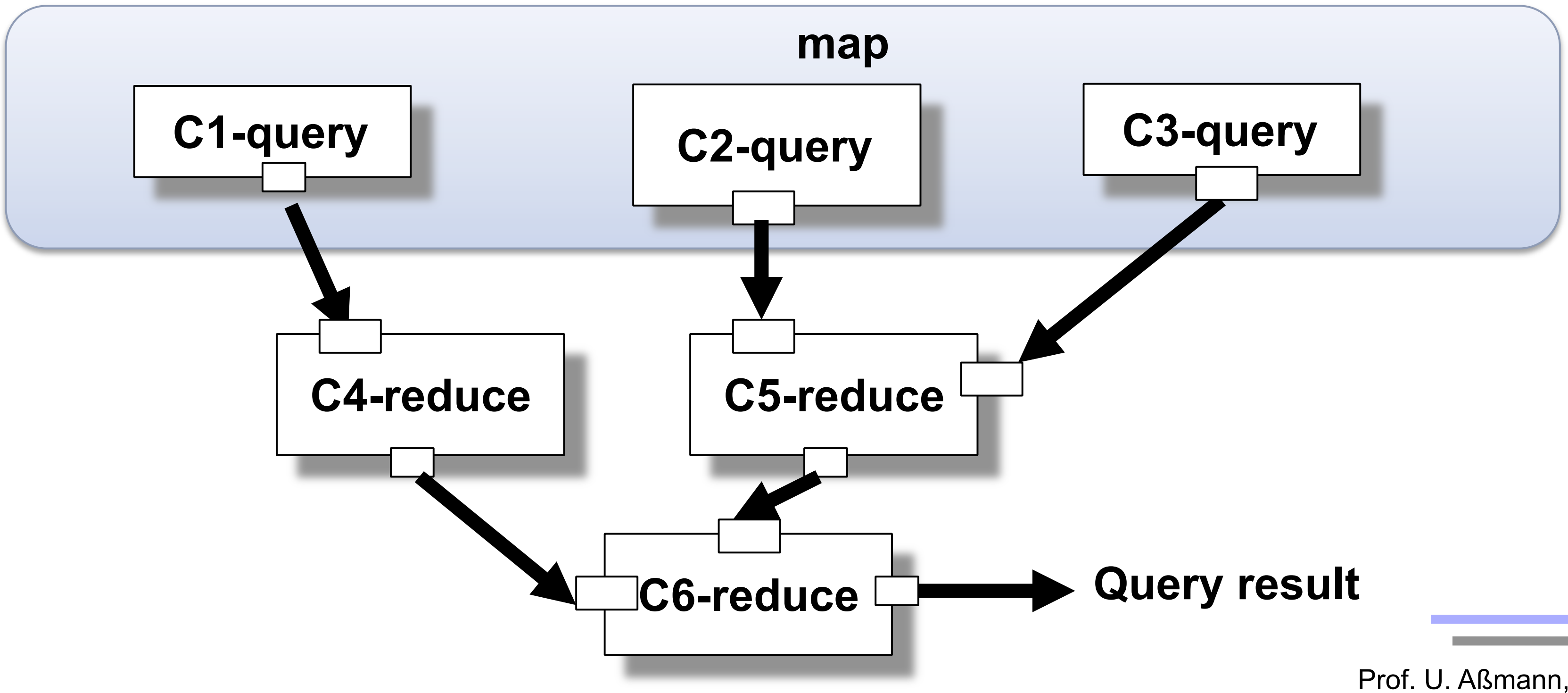
Repository Style with Tuple Space Connector

- 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) (see course „Softwarewerkzeuge“)



A Complex Composition Operator: Skeleton

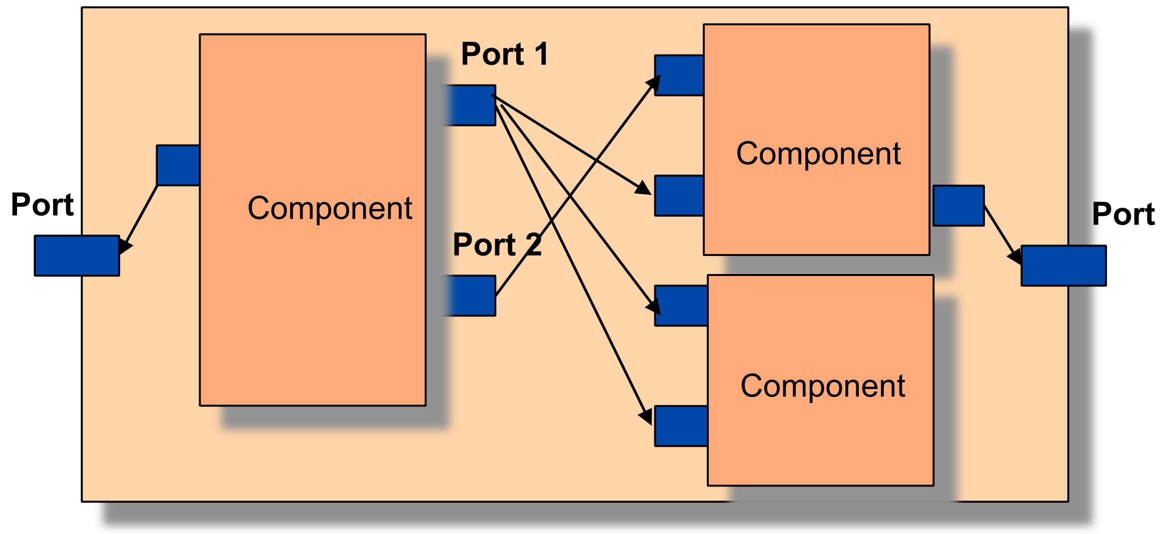
- An **architectural skeleton** is a coordination scheme for a set of components superimposing a topology of connectors (connector nets)
  - their encapsulation to a new component
- Example: the Map-Reduce Skeleton (Google) for searching
  - Divide-and-conquer, partition, zip, serialize, ...

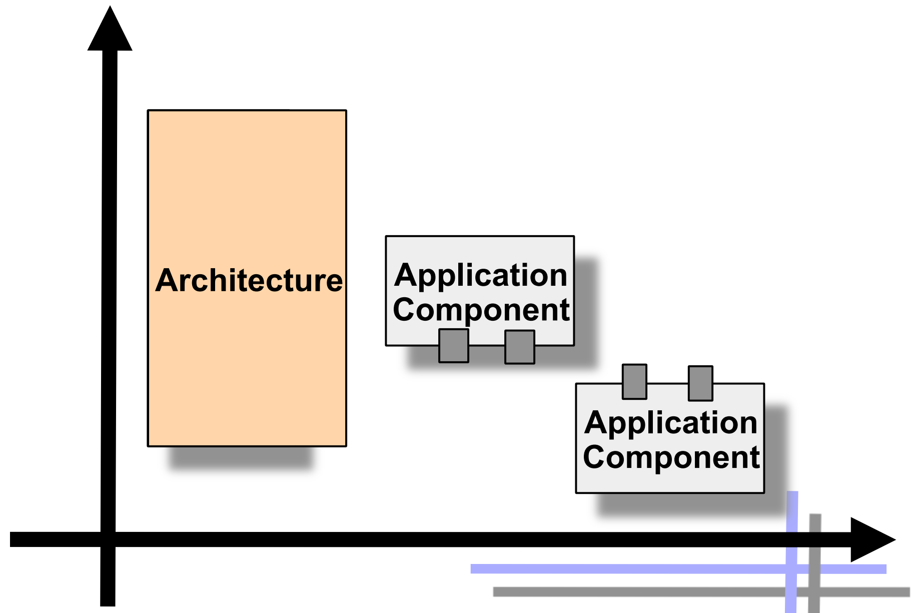


13.2.3 ADL

Architecture can be Exchanged Independently of Components

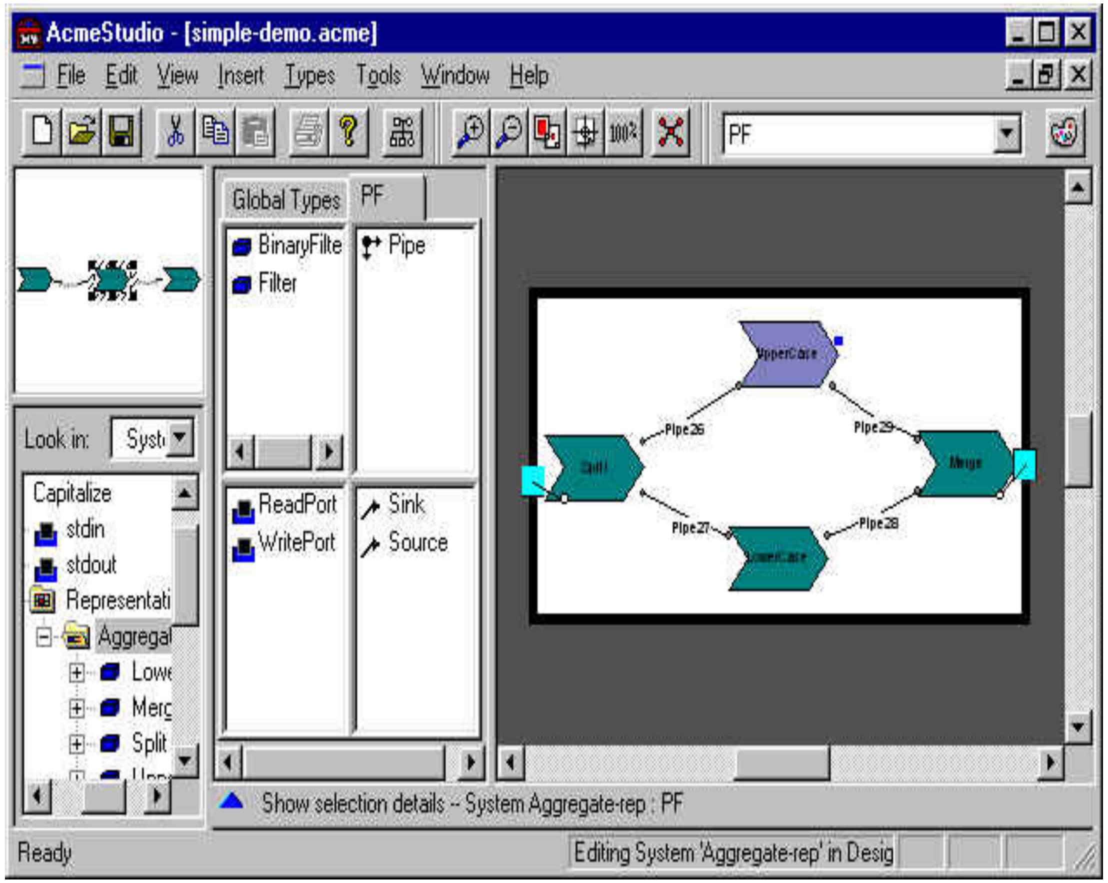
- Reuse of components and architectures is fundamentally improved
- Two dimensions of reuse
  - Architecture and components can be reused independently of each other





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
- Aesop [Garlan95]
- ACME [Garlan97]:



Show selection details -- System Aggregate-rep : PF

Editing System 'Aggregate-rep' in Desig

The Composition Language: ADL

- Architecture language (architectural description language, ADL)
  - ADL-compiler generating code for connectors and skeletons
  - ADL graphic and textual editors: simple specification
    - The architecture is a reducible graph with all its advantages
    - The reducibility of the architecture allows for simple overview, evolution, and documentation
  - XML-Readers/Writers for ADL
- An **architecture style** employs for a system or a layer only particular architectural concepts [Garlan/Shaw: Software Architecture] :
  - particular composition operators (connectors, skeletons, …)
  - particular communication carriers or topologies
  - Obeys specific architectural rules, often specified in logic
  - Ex.: Pipe-and-filter style, repository style, call-based style, event-driven architecture, 3-tier architecture, and many more

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

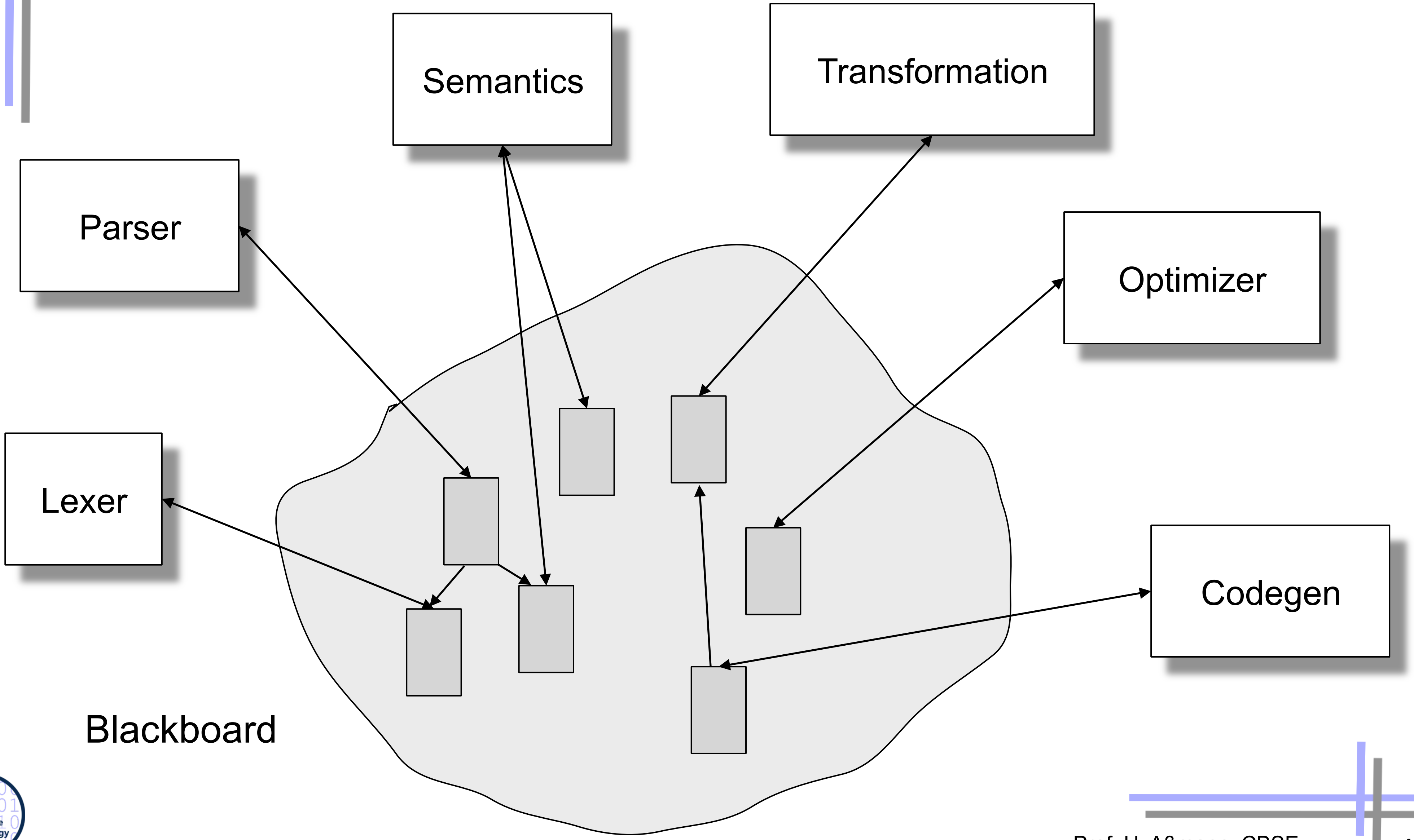
What ADL Offer for the Software Process

- 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
- 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

13.3 Examples

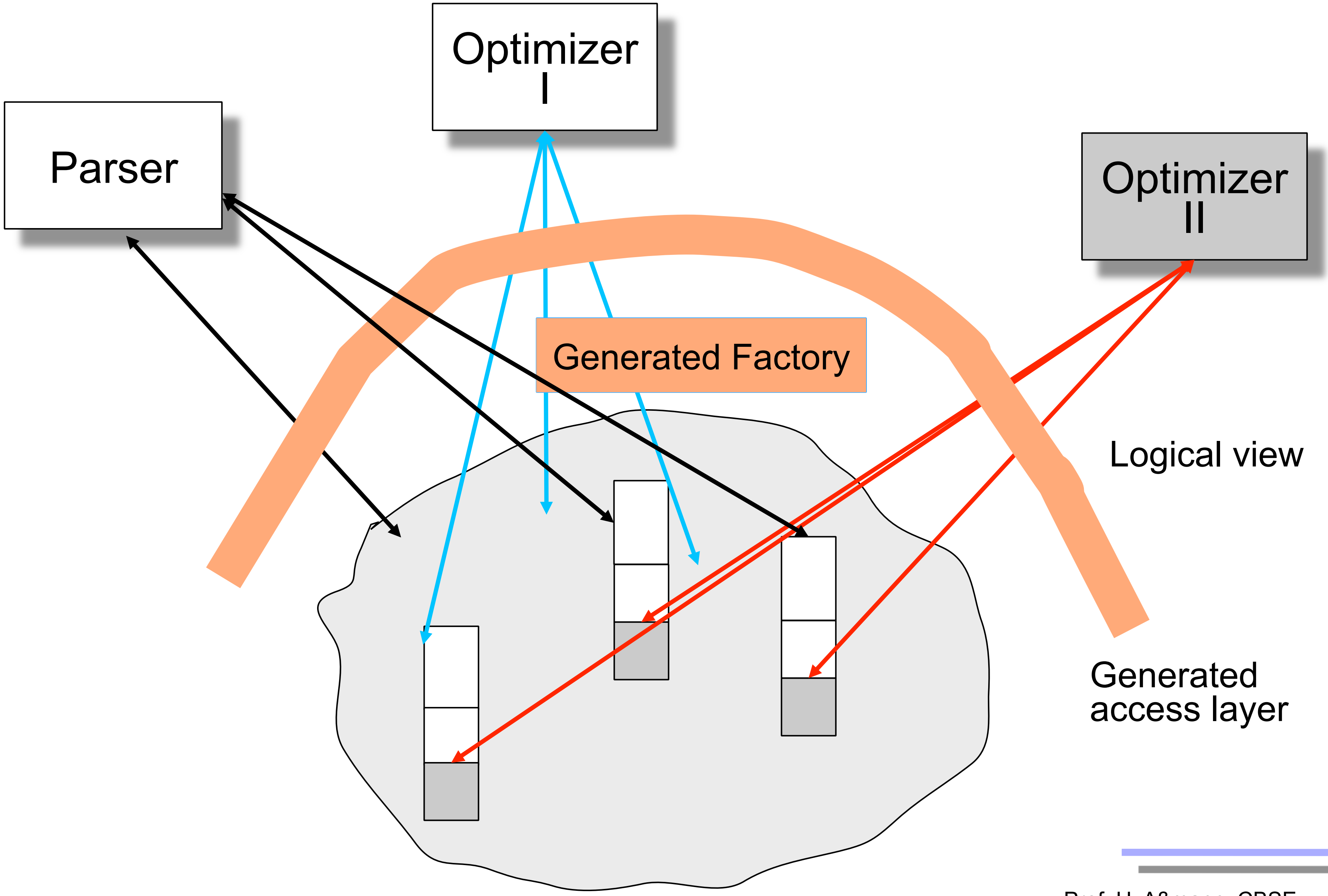
13.3.1 CoSy - A commercial architecture system for compilers

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

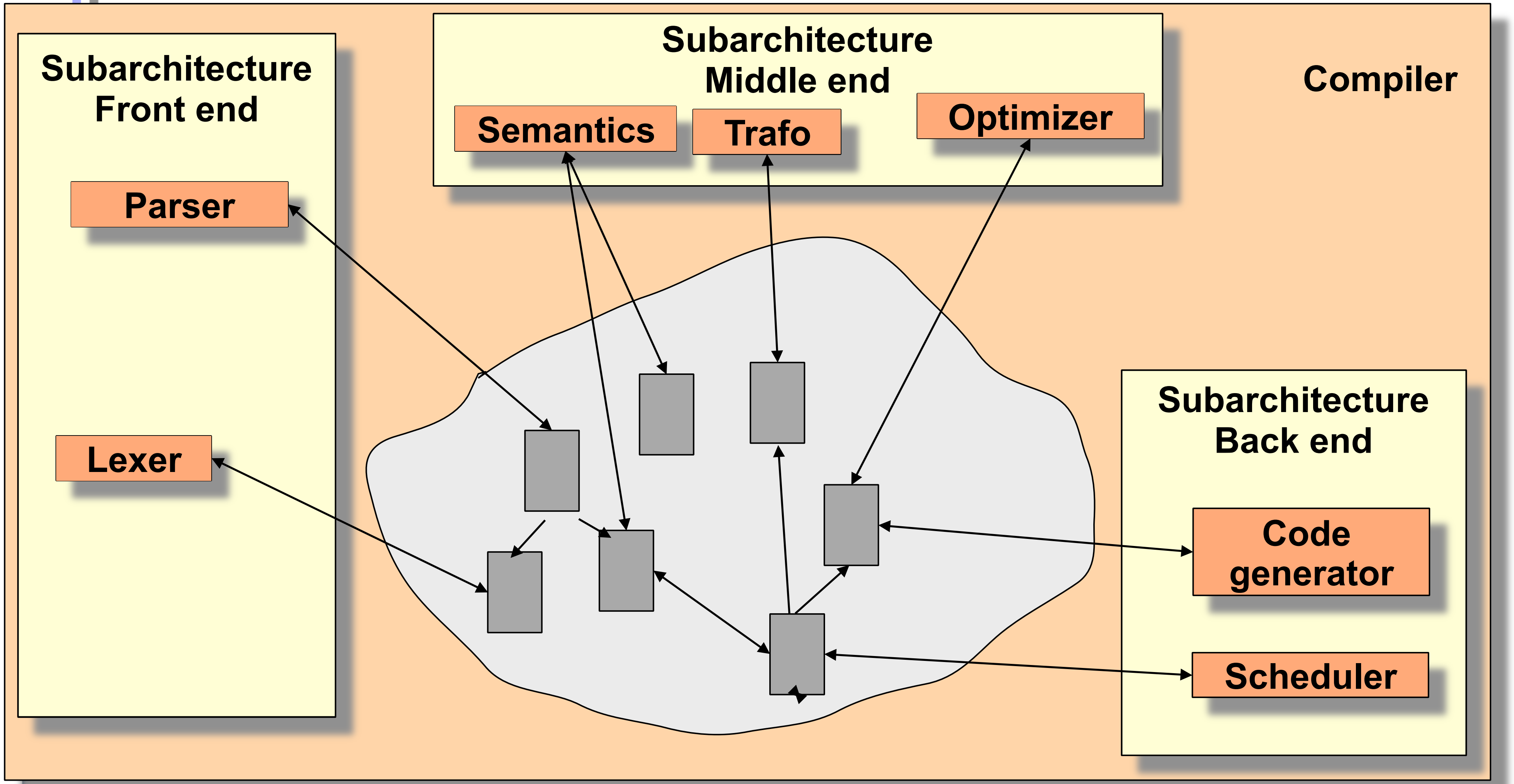


40

A CoSy Compiler



Hierarchical Components in the Repository Style (CoSy)



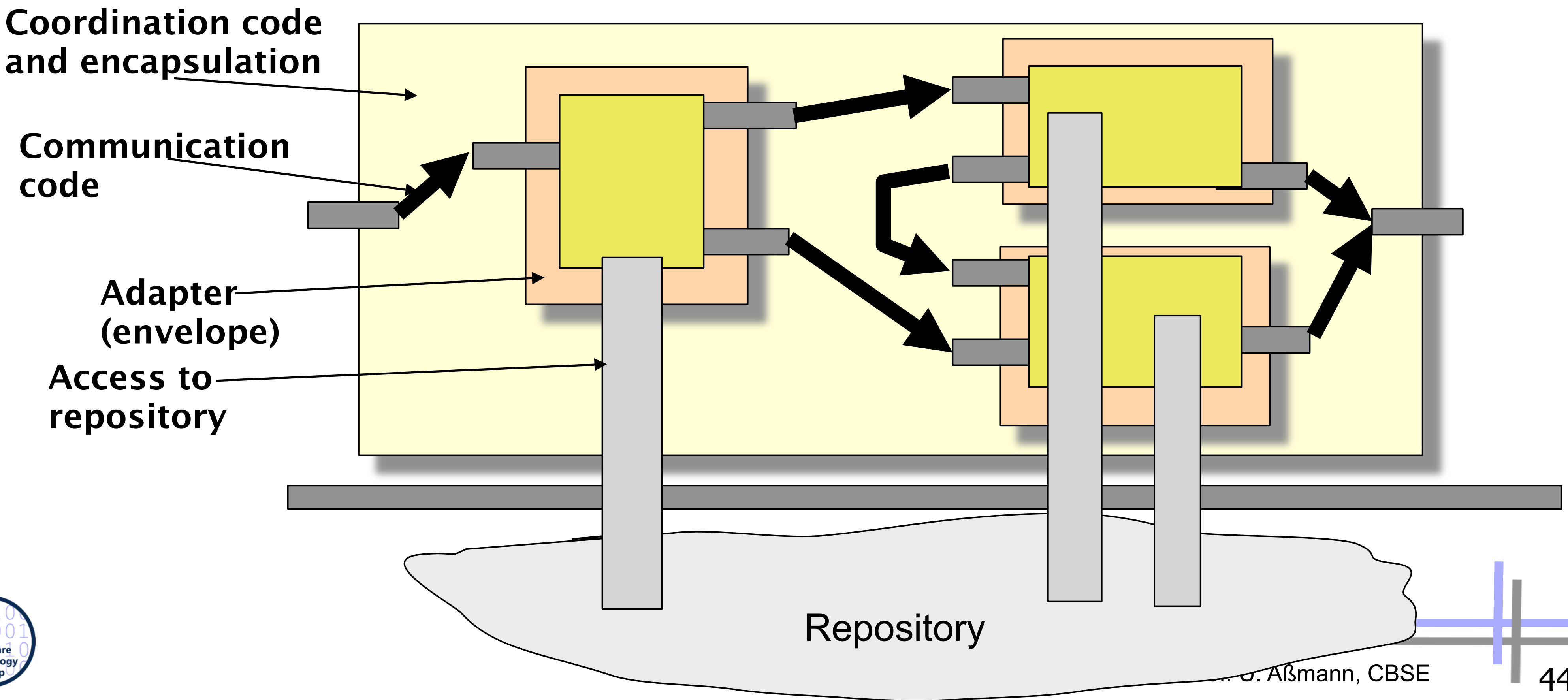
Example from EDL (Engine Description Language)

- Component classes (engine class)
- Component instances (engines)
- Basic components are implemented in C
- Interaction schemes form complex connectors
  - SEQUENTIAL
  - PIPELINE
  - DATAPARALLEL
  - SPECULATIVE
- EDL can embed automatically
  - Single-call-components into pipes
  - p<> means a pipe p
  - EDL can map their protocols to each other (p vs p<>)

```
ENGINE CLASS optimzer (procedure p) {
      controlflowAnalyser cfa;
      commonSubExprEliminator cse;
      loopVariableSimplifier lvs;

      PIPELINE cfa(p); cse(p); lvs(p);
}

ENGINE CLASS  compiler (file f) {
     .... Token token;
     Modules m;
PIPELINE
      // lexer takes file, delivers token pipe
      lexer(IN f, OUT token<>);
      // Parser delivers a module
      parser(IN token<>, OUT m);
      sema(m);
      decompose(m,p<>);
      // here comes a Pipe of procedures
      // from the module
      optimizer(p<>);
      backend(p<>);
}
```

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 are generated



Evaluation of CoSy

- CoSy is one of the few commercial architecture systems with professional support
  - CoSy realizes hierarchical repositories
  - 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

13.3.2 UNICON

- 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

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, PLBandler, 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, PLBandle, ReadFile, WriteFile
- Computation:
  - RoutineDef, RoutineCall, GlobalDataUse, PLBandle
- SharedData:
  - GlobalDataDef, GlobalDataUse, PLBandle
- 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
- PLBandler:
  - Participant fits to PLBandle, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef
- RPC
  - Definer fits to (Process| Schedprocess).RPCDef
  - User fits to (Process| Schedprocess).RPCCall
- RTScheduler
  - Load fits to Schedprocess.RTLoad

A "Modules" Component

```
INTERFACE IS

TYPE modules

LIBRARY
PLAYER timeget IS RoutineDef
    SIGNATURE ("new_type"; "void")
END timeget
PLAYER timeshow IS RoutineDef
    SIGNATURE (; "void")
END timeshow

END INTERFACE
```

A Filter

```
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
```

```
/* Establish connections  ESTABLISHs bind connectors to ports */
ESTABLISH C-proc-call WITH reverse.stack_init AS caller stack.stack_init AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.stack_is_empty AS caller stack.stack_is_empty AS definer
    END C-proc-call
ESTABLISH C-proc-call WITH reverse.push AS callr stack.push AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.pop AS callr stack.pop AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.exit AS callr libc.exit AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.fgets AS callr libc.fgets AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.fprintf AS callr libc.fprintf AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.malloc AS callr libc.malloc AS definer END C-proc-cal
ESTABLISH C-proc-call WITH reverse.strcpy AS callr libc.strcpy AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.strlen AS callr libc.strlen AS definer END C-proc-call

/* Lastly, we bind the players in the interface
to players in the implementation. Remember, it is okay to omit the bind of player "error." */
BIND input TO ABSTRACTION MAPSTO (reverse.fgets) END input
BIND output TO ABSTRACTION MAPSTO (reverse.fprintf) END output
END IMPLEMENTATION
END Reverser
```

Definition of Connectors

- In Version 4.0, connectors can be defined by users
- However, the extension of the compilers is complex:
  - a delegation class has to be developed,
  - the semantic analysis,
  - and the architecture analysis must be supported.

```
CONNECTOR C-proc-call
    protocol IS
    TYPE procedureCall
    ROLE definer IS Definer
    ROLE callr IS Callr
  END protocol
  IMPLEMENTATION IS BUILTIN
  END IMPLEMENTATION
END C-proc-call

CONNECTOR C-shared-data
  protocol IS
    TYPE DataAccess
    ROLE definer IS Definer
    ROLE user IS User
  END protocol
  IMPLEMENTATION IS  BUILTIN
  END IMPLEMENTATION
END C-shared-data
```

Attachment of External Libraries

```
COMPONENT Libc
INTERFACE IS
TYPE modules
LIBRARY PLAYER exit IS RoutineDef
SIGNATURE ("int"; "void") END exit PLAYER fgets IS RoutineDef
SIGNATURE ("char *", "int", "struct _iobuf *"; "char *") END fgets PLAYER fprintf IS RoutineDef
SIGNATURE ("struct _iobuf *", "char *", "char *"; "int") END fprintf PLAYER malloc IS RoutineDef
SIGNATURE ("unsigned"; "char *") END malloc PLAYER strcpy IS RoutineDef
SIGNATURE ("char *", "char *"; "char *") END strcpy PLAYER strlen IS RoutineDef
SIGNATURE ("char *"; "int") END strlen PLAYER _iwhether IS GlobalDataDef
SIGNATURE ("struct _iobuf *") END _iwhether END INTERFACE

IMPLEMENTATION IS
   VARIANT libc IN "-lc"
   IMPLTYPE (ObjectLibrary)
   END libc
END IMPLEMENTATION
END Libc
```

A Component with GUI-Annotations

```
COMPONENT KWIC
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
GUI-SCREEN-SIZE ("(lis :real-width 800 :width-unit "" :real-height 350 :height-unit "")")
DIRECTORY ("(lis "/usr/examples/ upcase.uni" "/usr/examples/cshift.uni"
                    "/usr/examples/ data.uni" "/usr/examples/converge.uni"
                    "/usr/examples/ sort.uni" "/usr/examples/unix-pipe.uni"
                    "/usr/examples/ reverse-f.uni")")
USES caps INTERFACE upcase
GUI-SCREEN-POSITION ("(lis :position (@pos 68 123) :player-positions (lis
                                    (cons "input" (cons `left 0.5)) (cons "error" (cons `right 0.6625))
                                    (cons "output" (cons `right 0.3375))))")
END caps (remaining definition owithted)
END IMPLEMENTATION
END KWIC
```

The KWIC Problem in UNICON

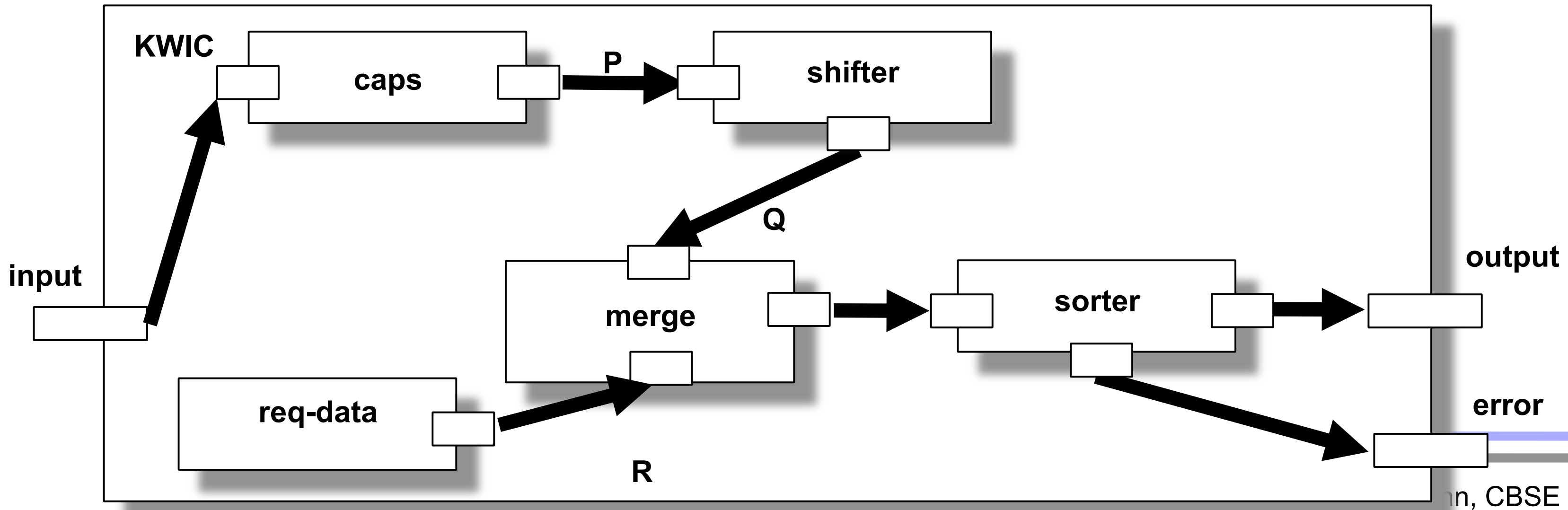
- Example from UniCon distribution:
- "Keyword in Context" problem (KWIC)
  - The KWIC problem is one of the 10 model problems of architecture systems
  - Proposed by Parnas to illustrate advantages of different designs [Parnas72]
  - For a text, a KWIC algorithm produces a permuted index
    - every 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

```
    every sentence is replicated  and          permuted
                                  every        sentence is replicated and permuted
                  every sentence  is           replicated and permuted
every sentence is replicated and  permuted
               every sentence is  replicated   and permuted
                           every  sentence     is replicated and permuted
```

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
```

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*
    - 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*.
    - It includes a set of policies about the style
    - A set of tools that work in harmony with the style, visualization information for tools
    - If something is part of the formal meaning, it should be part of a style

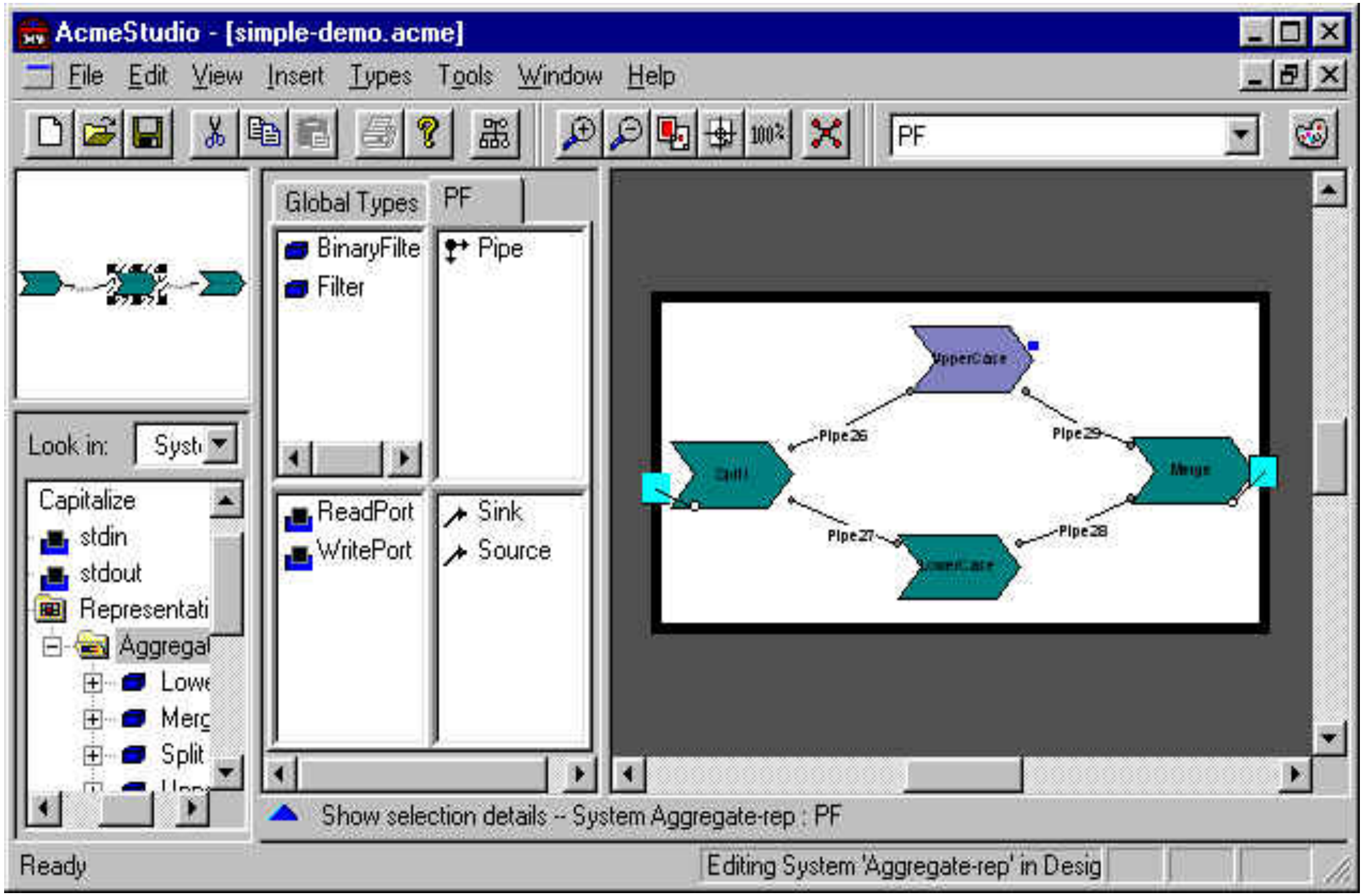
ACME (CMU)

- ACME is 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

Features

```
Template SystemIO () : Connector {
Connector {
    Roles: { source = SystemIORole();
             sink = SystemIORole()
           }
    properties: { blockingtype = non-blocking;
                  Aesop-style = subroutine-call
                }
  }
}
```

ACME Studio as Graphic Environment



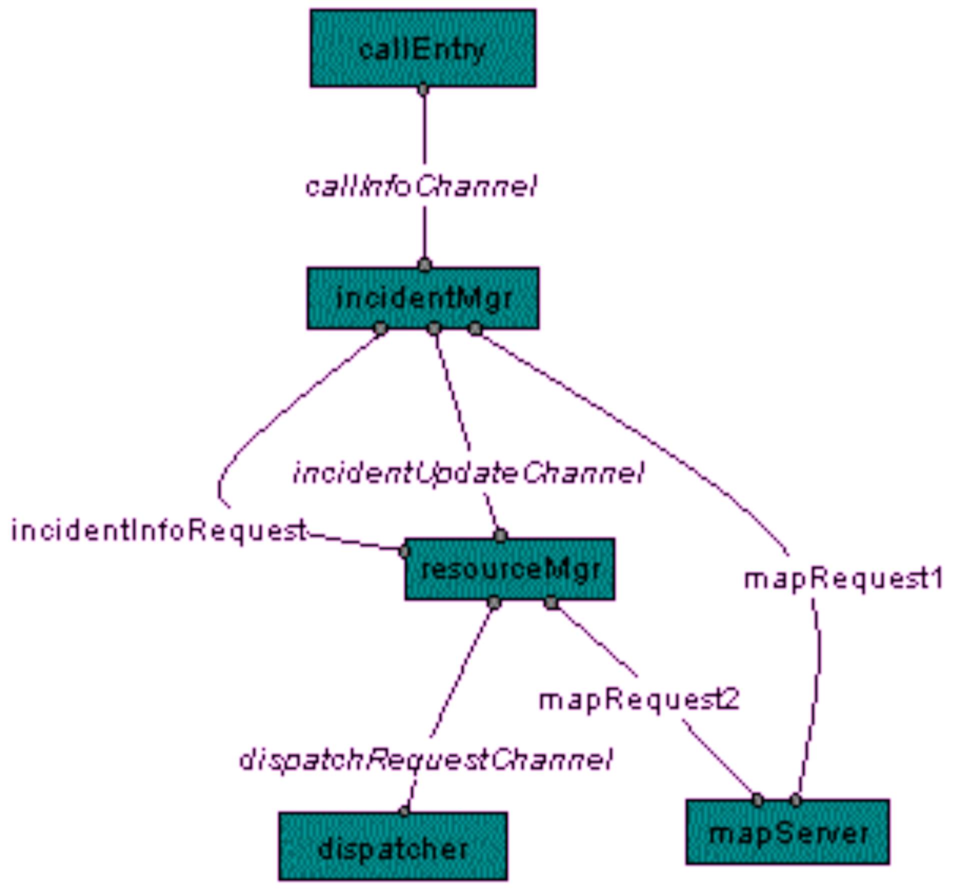
Example ACME Pipe/Filter-Family

```
// Describe a simple pipe-filter family.  This family
// definition demonstrates Acme's ability to specify
// a family of architectures as well as individual
// architectural instances.

// An Acme family includes the a set of component,
// connector, port and
// role types that define the design vocabulary
// provided by the family.

Family PipeFilterFam = {
  // Declare component types
  // A component type definition in Acme allows you to
  // to define the structure required by the type.
  // This structure
  // is defined using the same syntax as an instance
  // of a component.
  Component Type FilterT = {
        // All filters define at least two ports
        Ports { stdin; stdout; };
        property throughput : int;
  };
// Extend the basic filter type with a subclass (inheritance)
// Instances of UnixFilterT will have all of the properties and
// ports of instances of FilterT, plus a port and an
// implementationFile property
  Component Type UnixFilterT extends FilterT with {
        Port stther;
        property implementationFile : String;
  };

 // Declare the pipe connector type.  Like component types,
  // a connector type aso describes required structure.
  Connector Type PipeT = {
        Roles { source; sink; };
        property bufferSize : int;
  };
   // Declare some property types that can be used by systems
   // designed for the PipeFilterFam family
   property Type StringMsgFormatT = Record [ size:int; msg:String; ];
   property Type TasksT = enin order to {sort, transform, split, merge};
};
```

Instance of an ACME System

```
// Declare non-family property types thas will be used by this system
// instance.
property Type ShapeT = enum order to { rect, oval, roand-rect, line, arrow };
property Type ColorT = enum order to { black, blue, green, yellow, red, white };
property Type VisualizationT = Record [ x, y, width, height : int;
                                        shape : ShapeT;  color : ColorT; ];

// Describe an instance of a system using the PipeFilterFam family.
System simplePF : PipeFilterFam = {
  // Declare the components to be used in this design.
    // the component smooth has a visualization added
    Component smooth : FilterT = new FilterT extended with {
        property viz : VisualizationT = [ x = 20; y = 30; width = 100;
                                          height = 75; shape = rect; color = black ];
    };
    // detectErrors has a visualization added, as well as a
    // representation thas refers by name to a system that is
    // defined elsewhere.
    Component detectErrors : FilterT = new FilterT extended with {
        property viz : VisualizationT = [ x = 200; y = 30; width = 100;
                                   height = 75; shape = rect; color = black ];
        Representation r = {
          System showTracksSubsystem = {
               port stdout; port stdin;
               // ... the rest of the system description is ellided...
          };
          Bindings {
              stdout to showTracksSubsystem.stdout;
              stdin  to showTracksSubsystem.stdin;
          }
        }
    };

// Associate a value with the implementationFile property
// that comes with the UnixFilterT type.
    Component showTracks : UnixFilterT =
            new UnixFilterT extended with {
        property viz : VisualizationT = [x = 400; y = 30; width = 100;
                        height = 75; shape = rect; color = black ];
        property implementationFile : String
                  = "IMPL_HOME/showTracks.c";
    };

    // Declare the system's connectors.
    Connector firstPipe : PipeT;
    Connector secondPipe : PipeT;

    // Declare the system's attachments/topology.
    Attachment smooth.stdout to firstPipe.source;
    Attachment detectErrors.stdin to firstPipe.sink;
    Attachment detectErrors.stdout to secondPipe.source;
    Attachment showTracks.stdin to secondPipe.sink;
}
```

London Ambulance System in ACME



London Ambulance System in ACME

```
 property Type FlowDirectionT = enin order to { from2to,
to2from };
 Connector Type MessagePassChannelT = {
      Roles { fromRole; toRole; };
      property msgFlow : FlowDirectionT;
 };
 Connector Type RPC_T = { Roles { clientEnd; serverEnd; } };

 // Instance based example - simple LAS architecture:
// declare system components (none of which are typed)

System LAS = {
  Component callntry = { Port sendCallMsg; };
  Component incidentMgr = {
      Ports { mapRequest; incidentInfoRequests;
             sendIncidentInfo; receiveCallMsg; }
  };
  Component resourceMgr = {
      Ports { mapRequest; incidentInfoRequest;
             receiveIncidentInfo; sendDispatchRequest; }
  };
  // RPC connnectors
  Connector incidentInfoRequest : RPC_T = {
      Roles { clientEnd; serverEnd; }
  };
  Connector mapRequest1 : RPC_T = {
      Roles { clientEnd; serverEnd; }
  };
  Connector mapRequest2 : RPC_T = {
      Roles { clientEnd; serverEnd; }
  };
 Component dispatcher = { Port
receiveDispatchRequest; };
  Component mapServer = {
      Ports { requestPort1; requestPort2; }
  };
  // declare system connectors
  // message passing connectors
  Connector callInfoChannel : MessagePassChannelT = {
      Roles { fromRole; toRole; }
      property msgFlow : FlowDirectionT = from2to;
  };
  Connector incidentUpdateChannel :
MessagePassChannelT = {
      Roles { fromRole; toRole; }
      property msgFlow : FlowDirectionT = from2to;
  };
  Connector dispatchRequestChannel :
MessagePassChannelT = {
      Roles { fromRole; toRole; }
      property msgFlow : FlowDirectionT = from2to;
  };
```

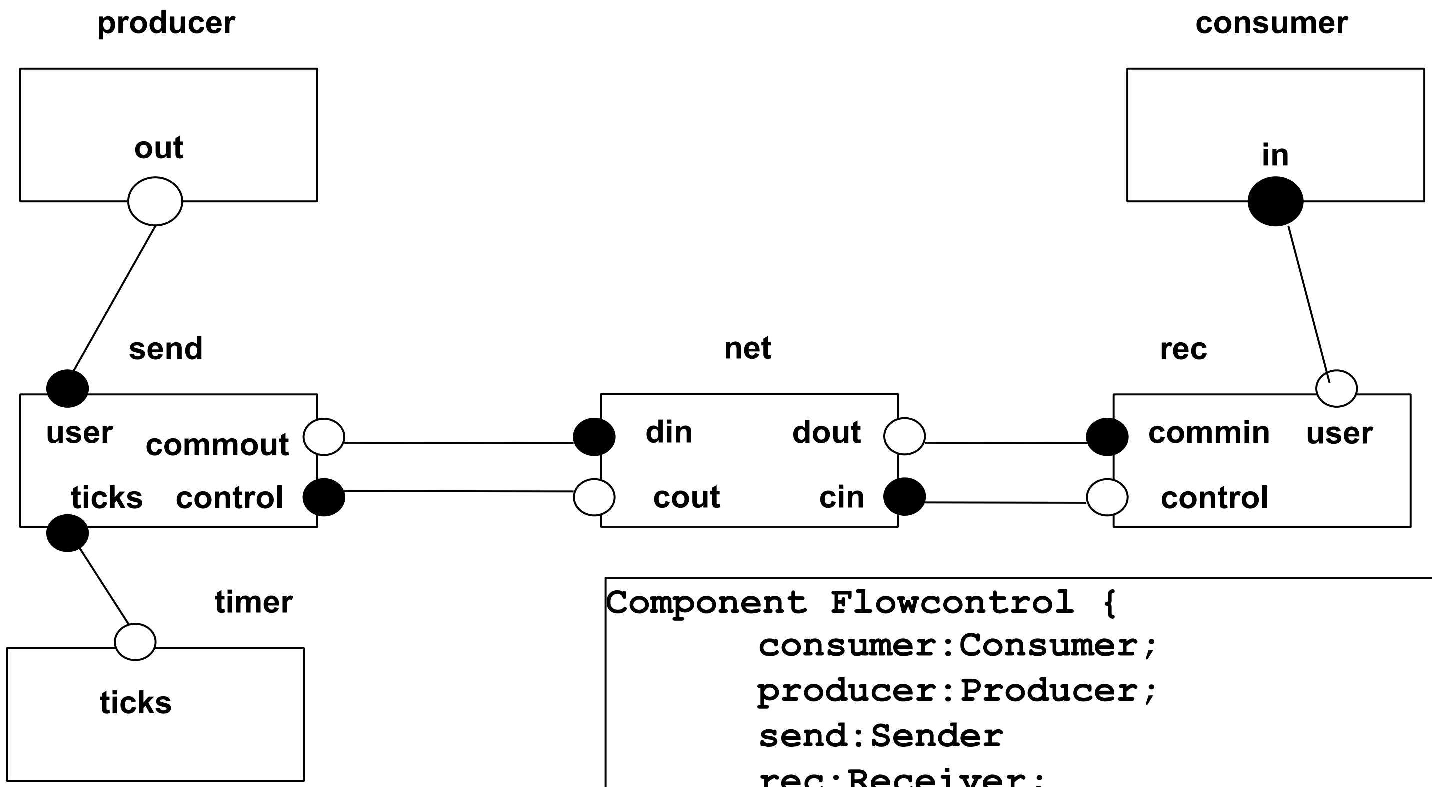
```
// incidentInfoPath attachments
  Attachments {
      // calls to incident_manager
      callntry.sendCallMsg to callInfoChannel.fromRole;
      incidentMgr.receiveCallMsg to callInfoChannel.toRole;
      // incident updates to resource manager
      incidentMgr.sendIncidentInfo
           to incidentUpdateChannel.fromRole;
      resourceMgr.receiveIncidentInfo
           to incidentUpdateChannel.toRole;
      // dispatch requests to dispatcher
      resourceMgr.sendDispatchRequest
           to dispatchRequestChannel.fromRole;
      dispatcher.receiveDispatchRequest
           to dispatchRequestChannel.toRole;
  };
```

```
// rpcRequests attachments
  Attachments {
      // calls to map server
      incidentMgr.mapRequest to mapRequest1.clientEnd;
      mapServer.requestPort1 to mapRequest1.serverEnd;
      resourceMgr.mapRequest to mapRequest2.clientEnd;
      mapServer.requestPort2 to mapRequest2.serverEnd;
      // incident info from incident_mgr
      resourceMgr.incidentInfoRequest to
             incidentInfoRequest.clientEnd;
      incidentMgr.incidentInfoRequests to
             incidentInfoRequest.serverEnd;
  };
}
```

13.3.4 Darwin (Imperial College)

- Components
  - Primitive and composed
  - Components can be recursively specified or iterated by index range
  - Components can be parameterized
- Ports
  - In, out (required, provided)
  - Ports can be bound implicitly and in sets
- Several versions available (C++, Java)
- Graphic or textual edits

Simple Producer/Consumer



```
Component Flowcontrol {
        consumer:Consumer;
        producer:Producer;
        send:Sender
        rec:Receiver;
        net:Net;
        timer:Timer;
Bind
        producer.out            -- send.user;
        timer.ticks     -- send.ticks;
        net.cout                -- send.control;
        send.commout    -- net.din;
        net.dout                -- rec.commin;
        rec.control     -- net.cin;
        rec.user                -- consumer.in;
}
```

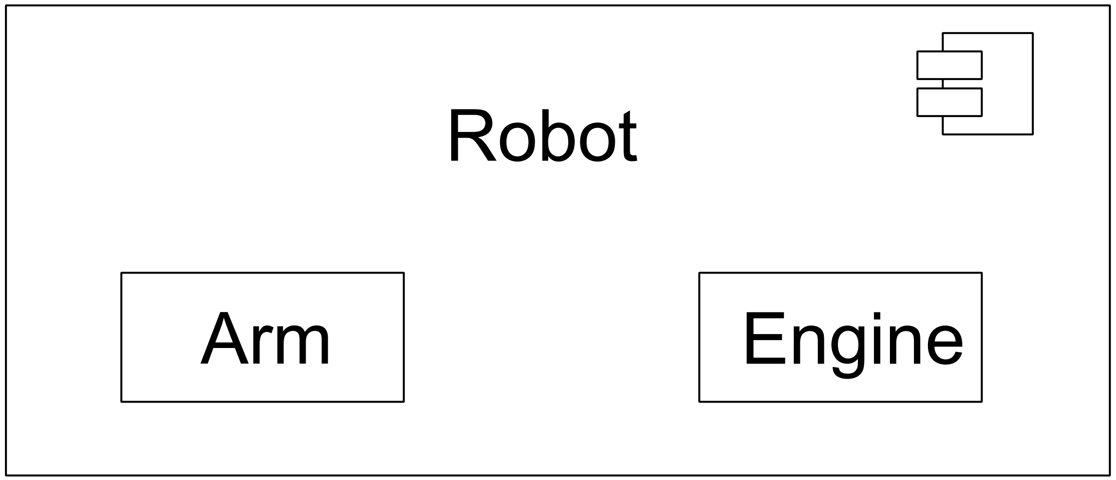
13.4 Architectural Languages in UML

Architecture Languages in UML

- “I have to learn UML, how should I also learn an ADL??”
  - Learning curve for the standard developer
  - Standard tools? Development environments?
- The Hofmeister Model of Architecture Description
  - [Soni/Nord/Hofmeister] is the first article that has propagated the idea of specifying and architecture language with UML
  - Conceptual level: Conceptual architecture (components, connectors)
  - Modules interconnection architecture (modules and their connections)
  - Execution architecture: runtime architecture
  - Code architecture level: division of systems onto files
- Describe every views in UML with profiles
  - UML allows the definition of *stereotypes*
    - Model connectors and ports, modules, runtime components with stereotypes
    - Map them to icons, so that the UML specification looks similar to a specification in a architecture system

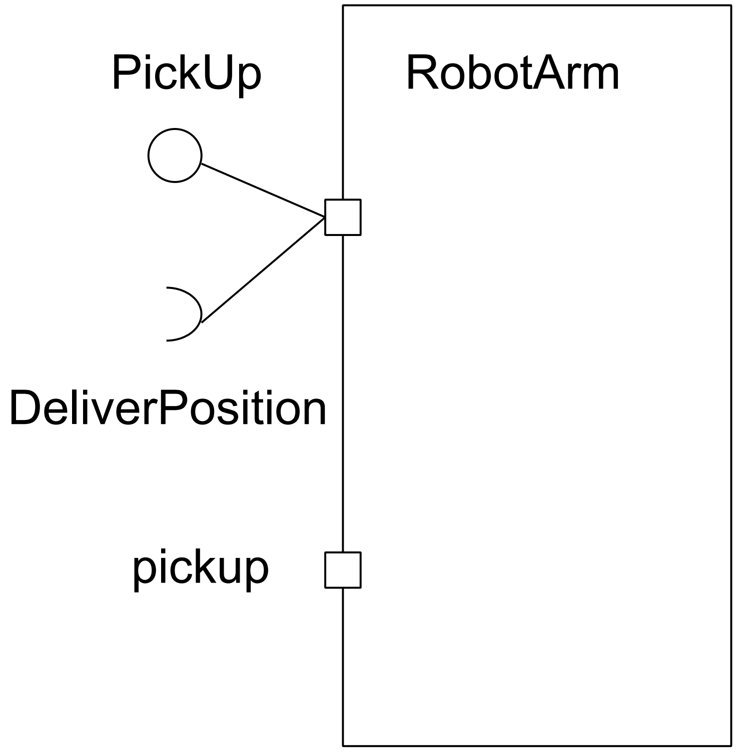
Components in UML 2.0

- Idea has been taken over by UML 2.0:
  - ”a component is a self-contained unit that encapsulates the state and behavior of a number of classifiers.
  - .. A component specifies a formal contract of services
  - .. Has *provided* and *required* interfaces...”
  - Components can be nested
  - A *delegation connector* maps external interfaces to components
- Difference to UML classes:
  - The features of a component are provided and required interfaces
  - UML components can be nested



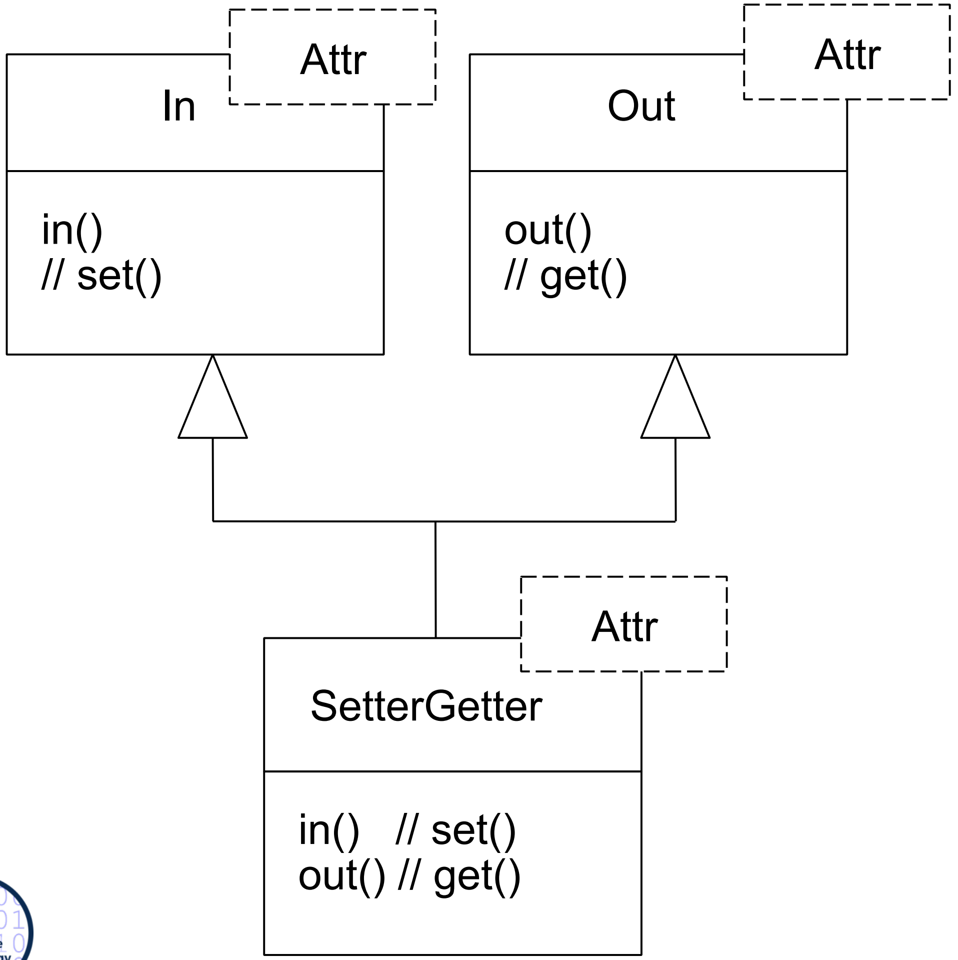
Ports in UML 2.0

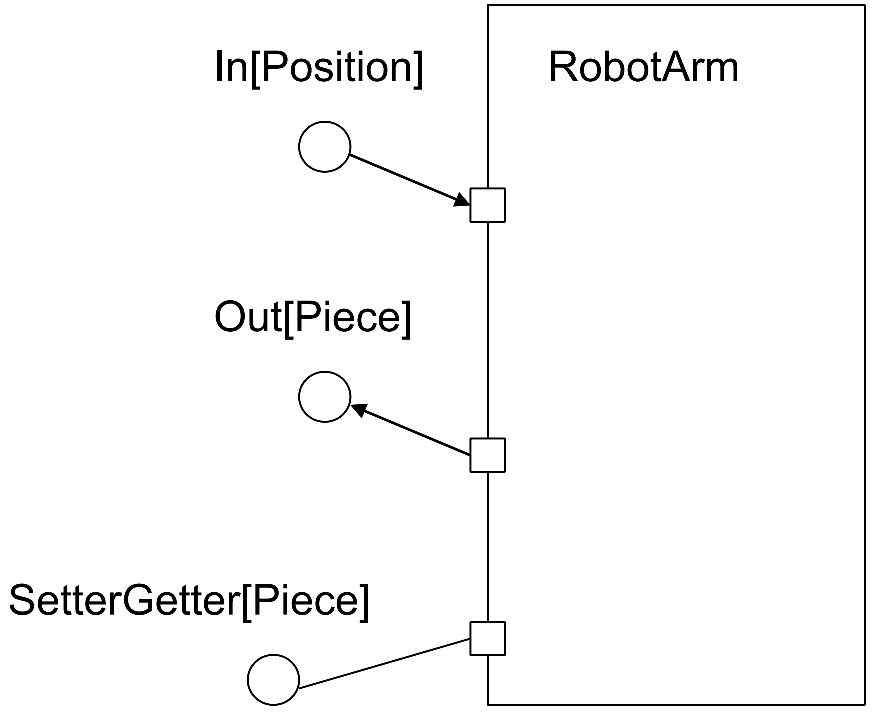
- Ports in UML 2.0 are port objects (gates, interaction points) that govern the communication of a component
- Ports may be simple (only data-flow, data service)
  - in or out
- Ports may be complex services
  - Then, they implement a provided or required interface



Ports in UML 2.0

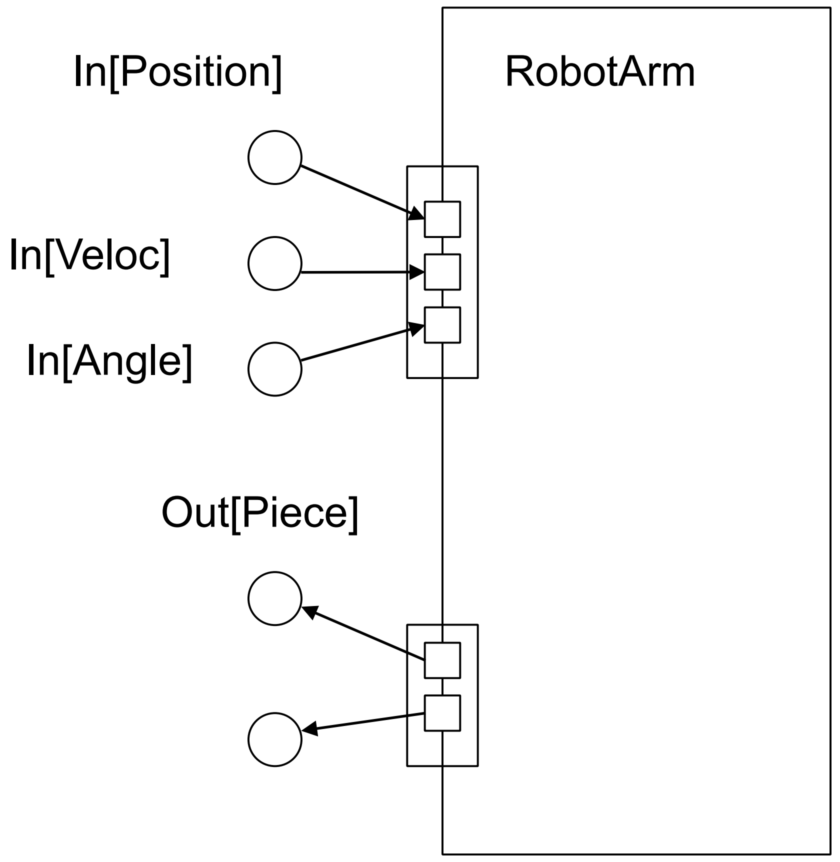
- We use the following conventions





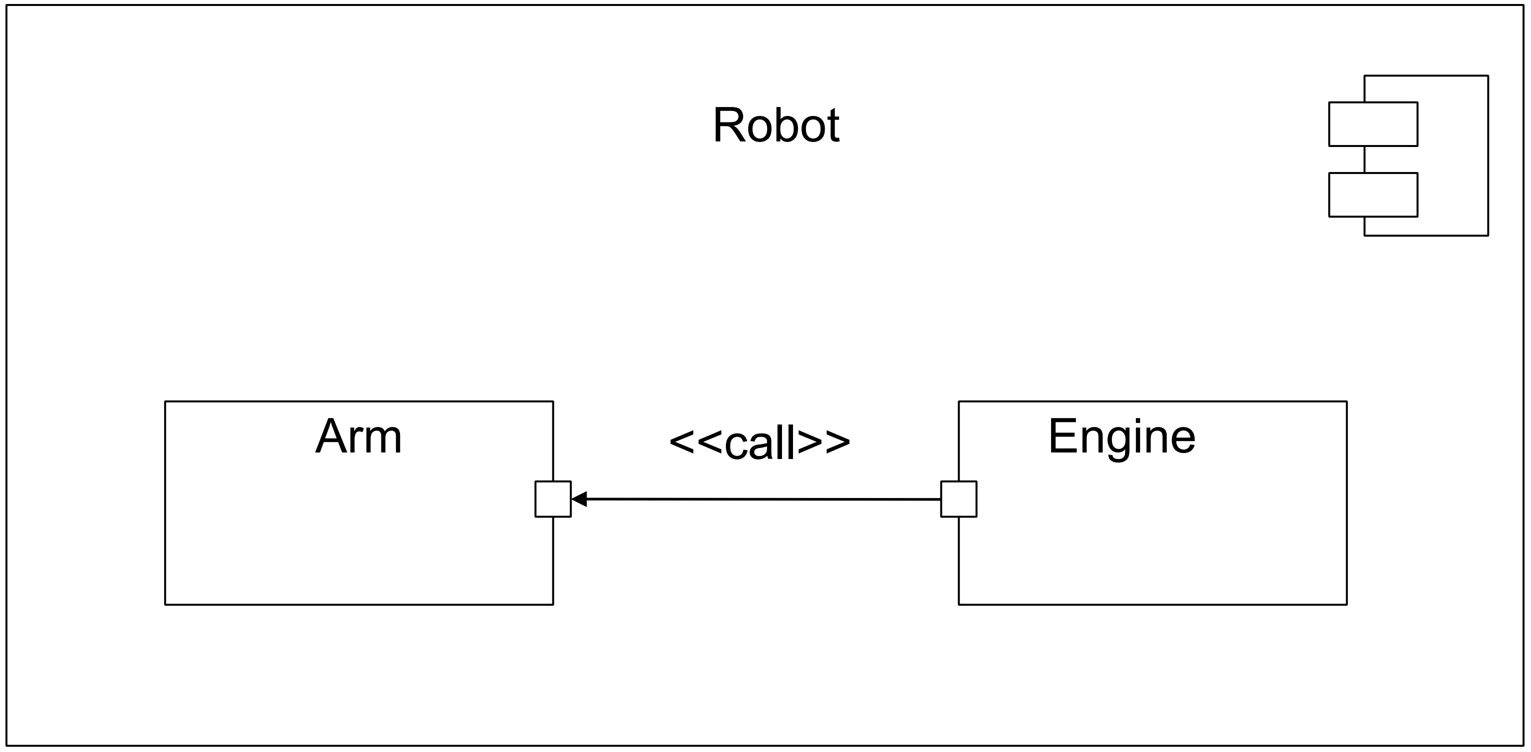
Services

- Ports can be grouped to Services

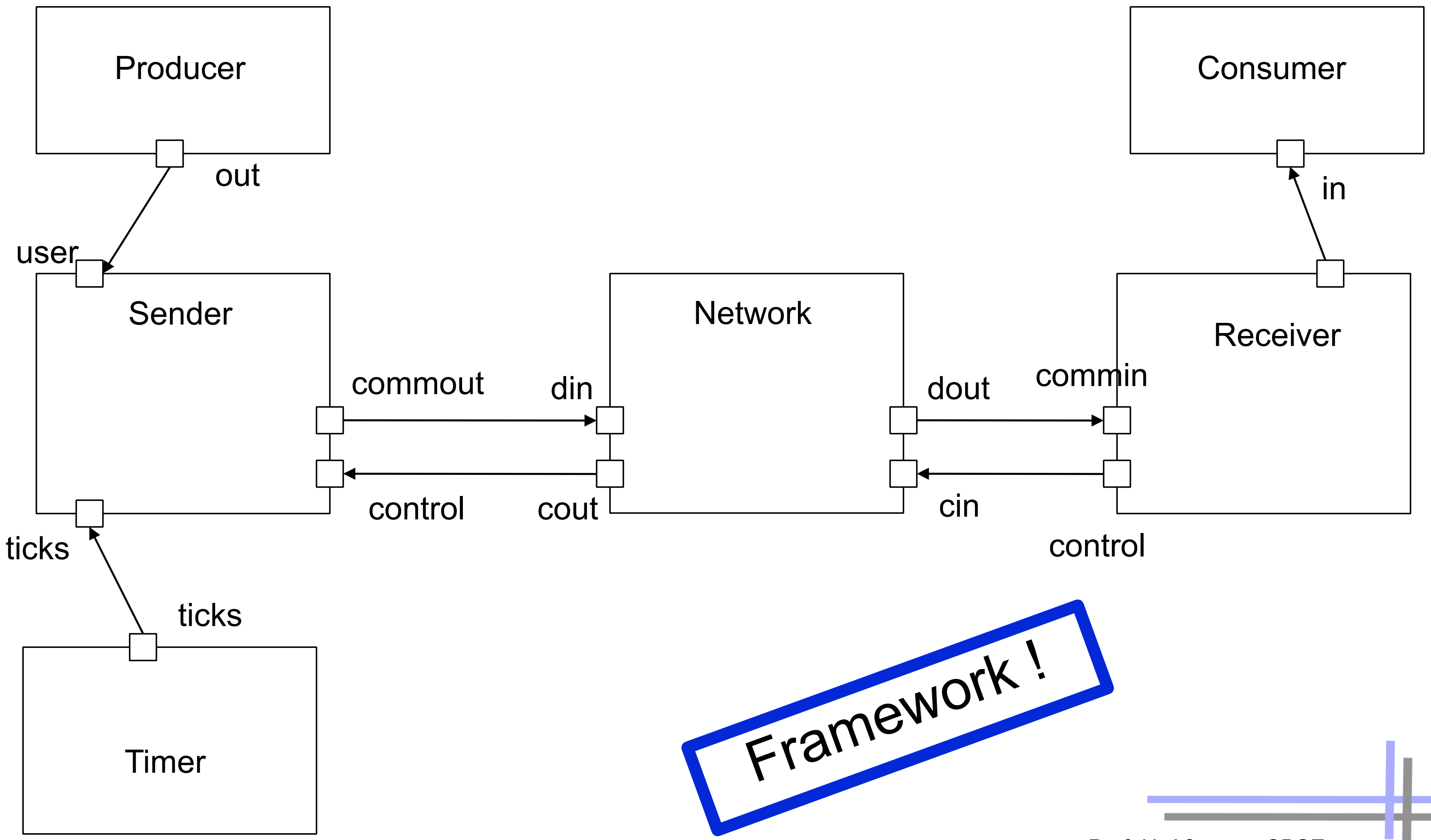


Connectors in UML 2.0

- Connectors become special associations, marked up by stereotypes, that link ports



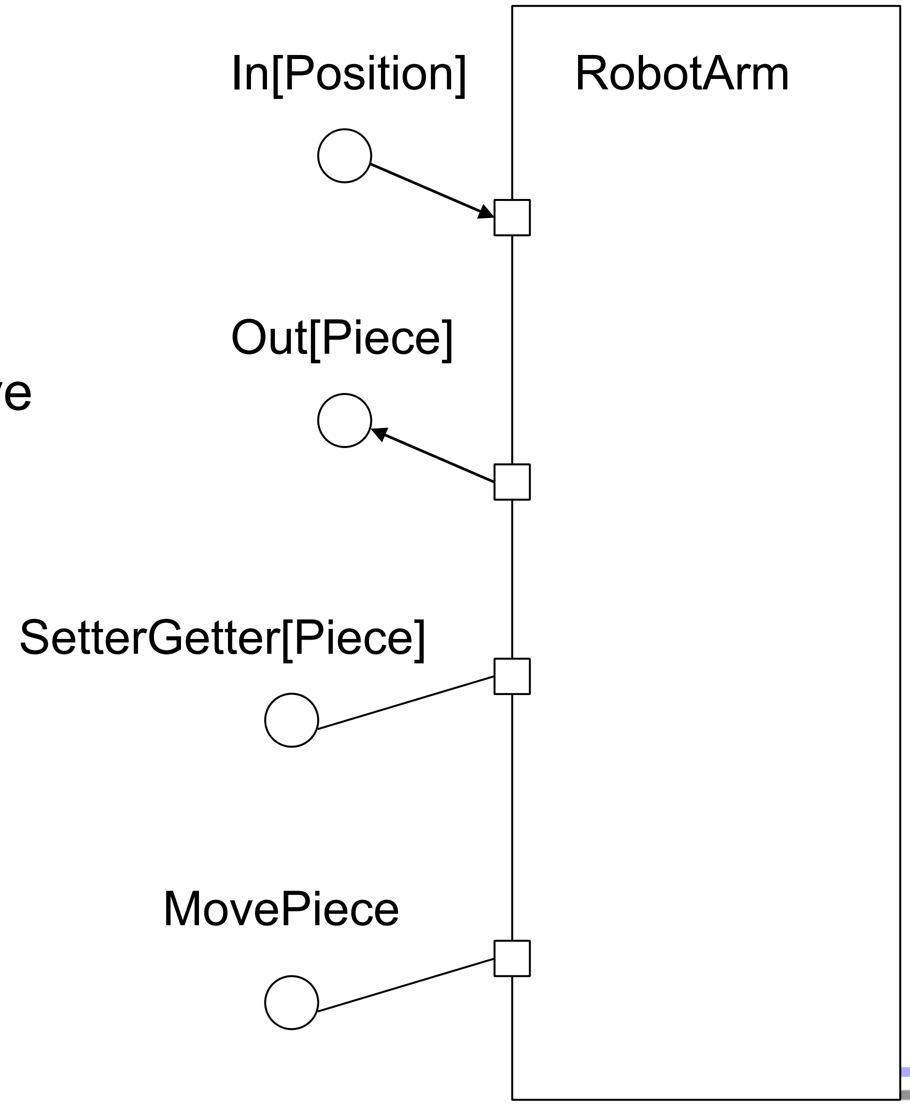
Simple Producer/Consumer in UML 2.0



Framework !

Exchange of Connectors

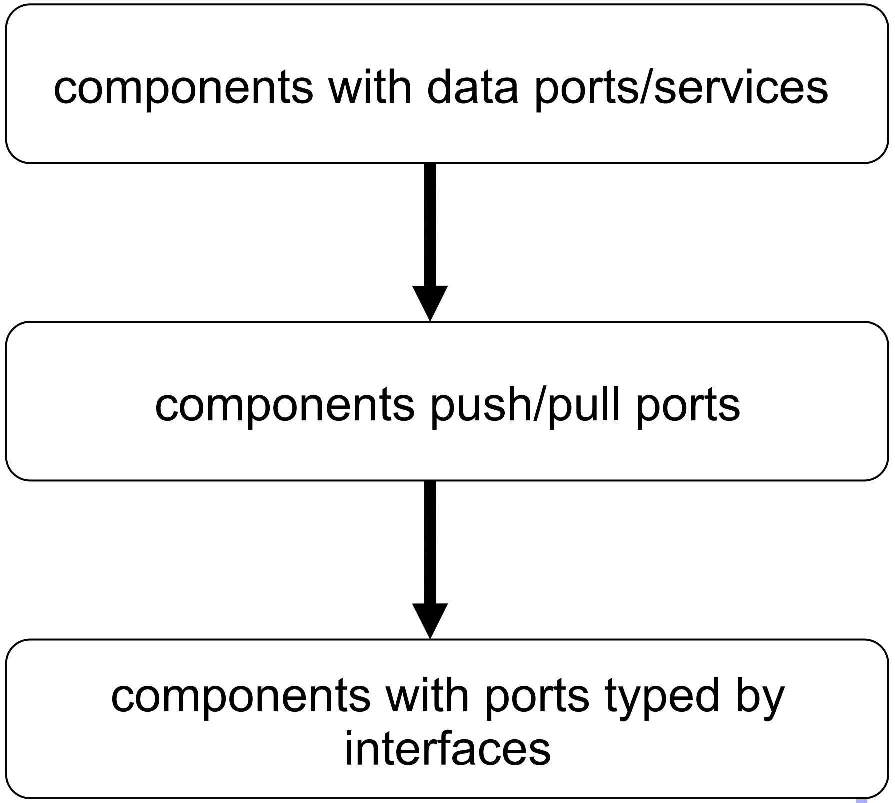
- The more complex the interface of the port, the more difficult it is to exchange the connectors
- Data-flow ports and data services abstract from many details
- Complex ports fix more details
- Only with data services and property services, connectors have best exchangeability



Rule of Thumbs for Architectural Design with UML 2.0

- Start the design with data ports and services
- Develop connectors
- In a second step, fix control flow
  - push-pull
  - Refine connectors
- In a third step, introduce synchronization
  - Parallel/sequential
  - Refine connectors

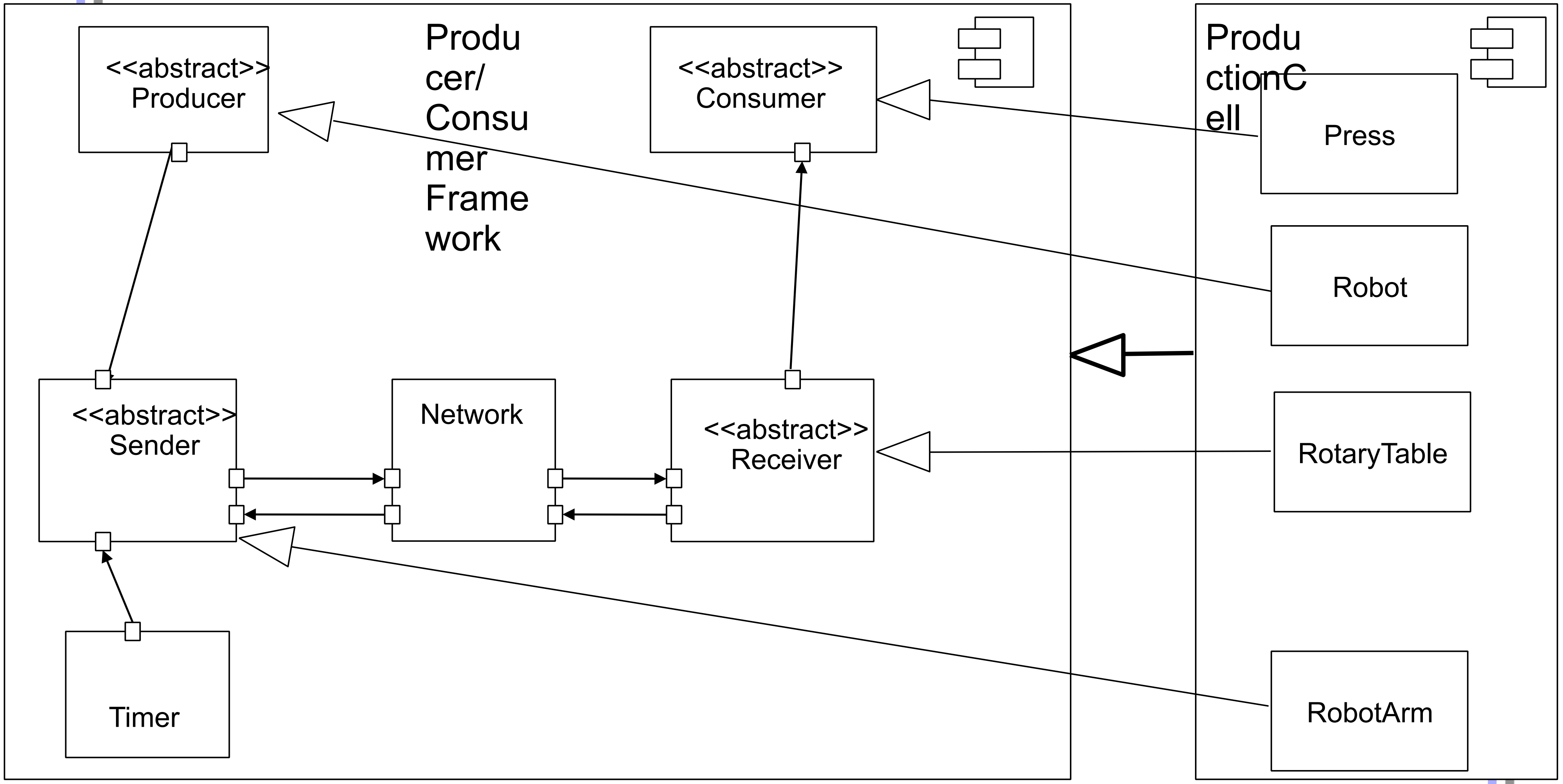
- In MDA levels:


components with data ports/services
components push/pull ports
components with ports typed by interfaces

Components and Frameworks

- In UML 2.0 frameworks can be defined by components and connectors
  - The classes in a specification can be held abstract, by abstract classes or genericity
- *Whitebox framework*:
  - Construct an application by subclassing
- *Blackbox framework*:
  - Construct an application by delegation
- *Generic framework*:
  - Construct an application by parameterization

Whitebox Framework in UML 2.0 with Components and Connectors

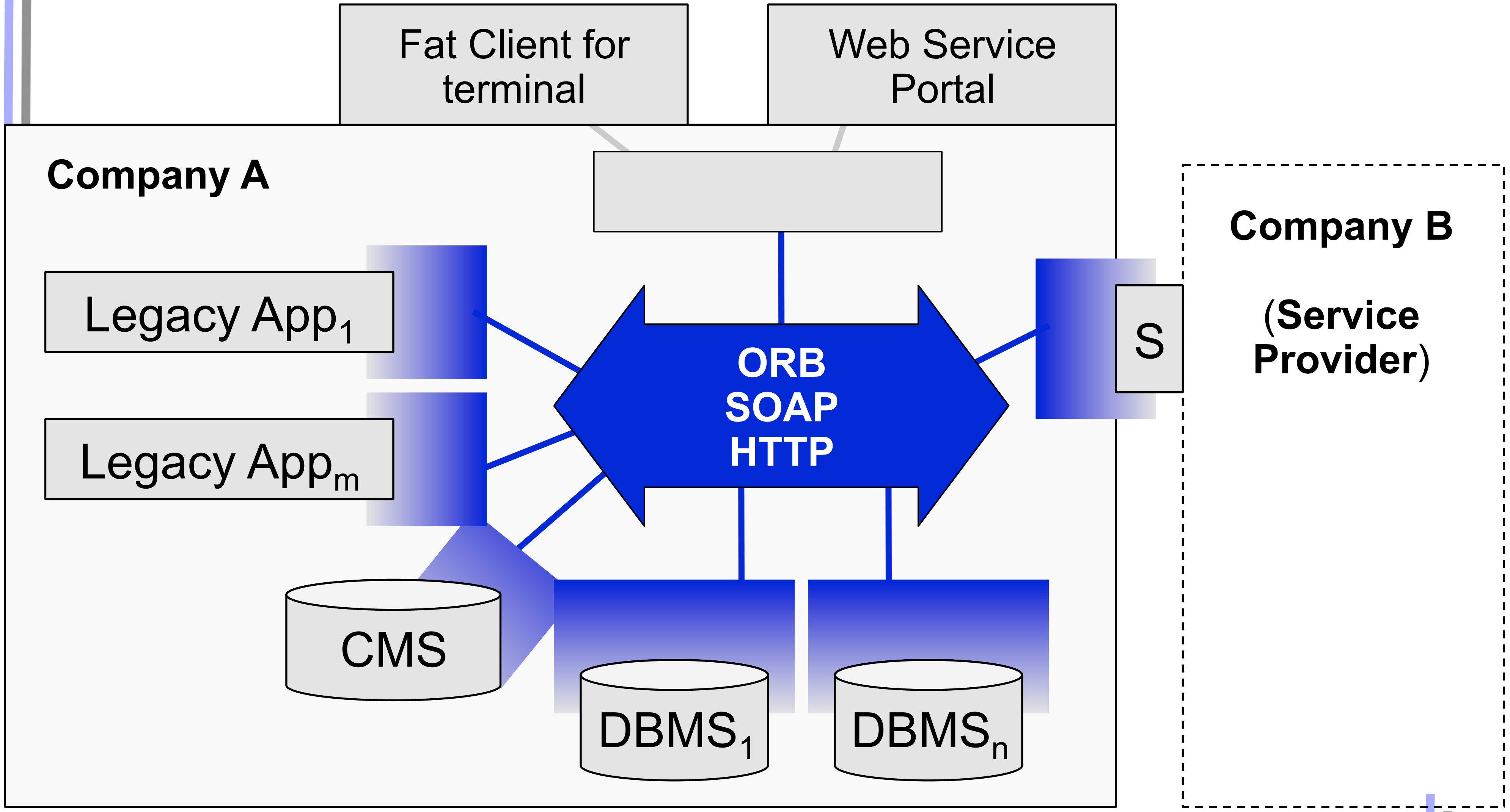


13.5 Refinement of UML Connectors in Model-Driven Development

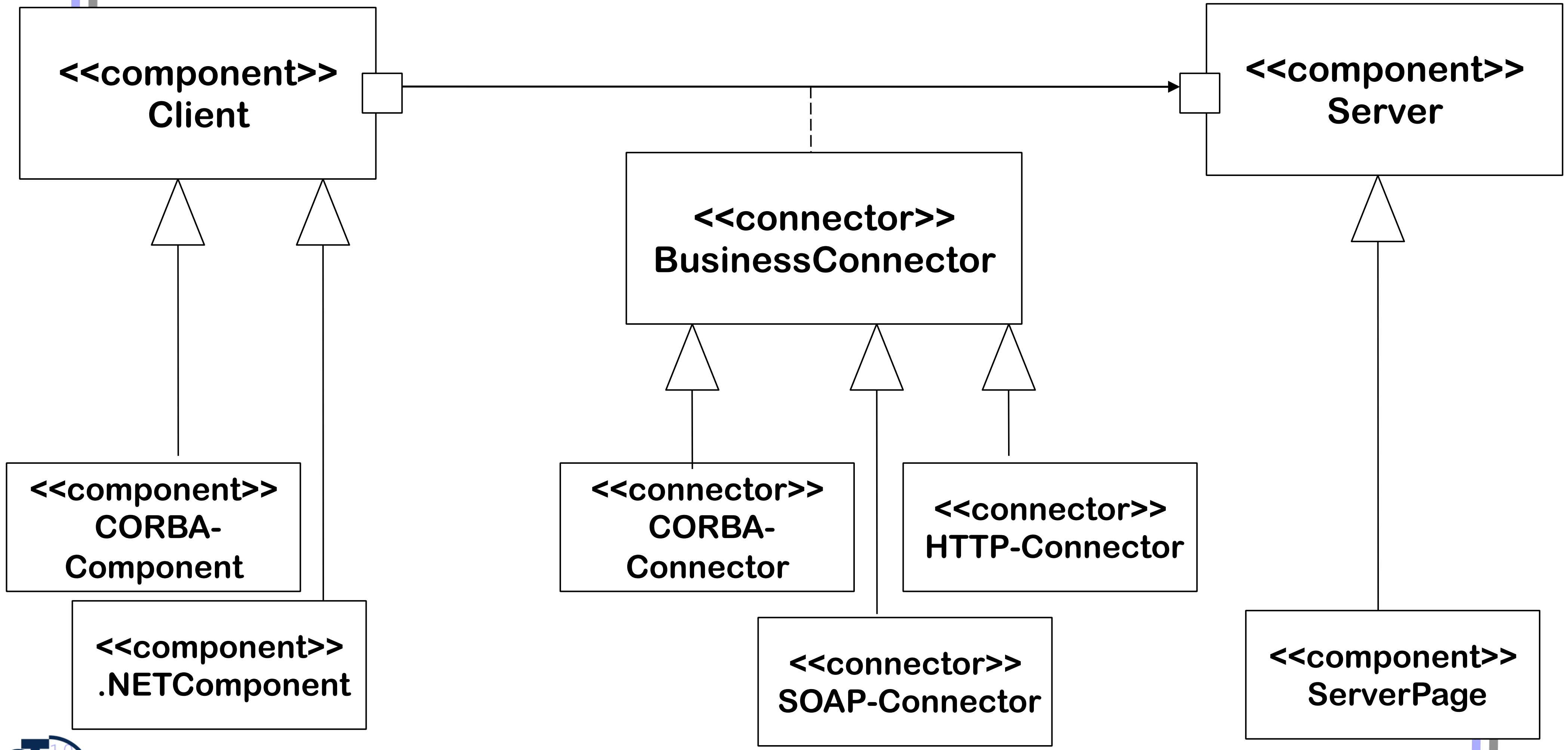
Idea: Use UML to Create Connectors for all Classical Component Systems

- Since classical component systems do not provide connectors, introduce them via stereotypes in UML
- The connection mechanisms are available
  - however, the encapsulation to connectors is missing
- Use the connectors in design
- Implementation
  - Generate the implementation
  - Refine to languages, such as ArchJava or ISC (see later)
  - or implement by hand, as usual.

Example Complex Connectors CORBA/SOAP/HTTP



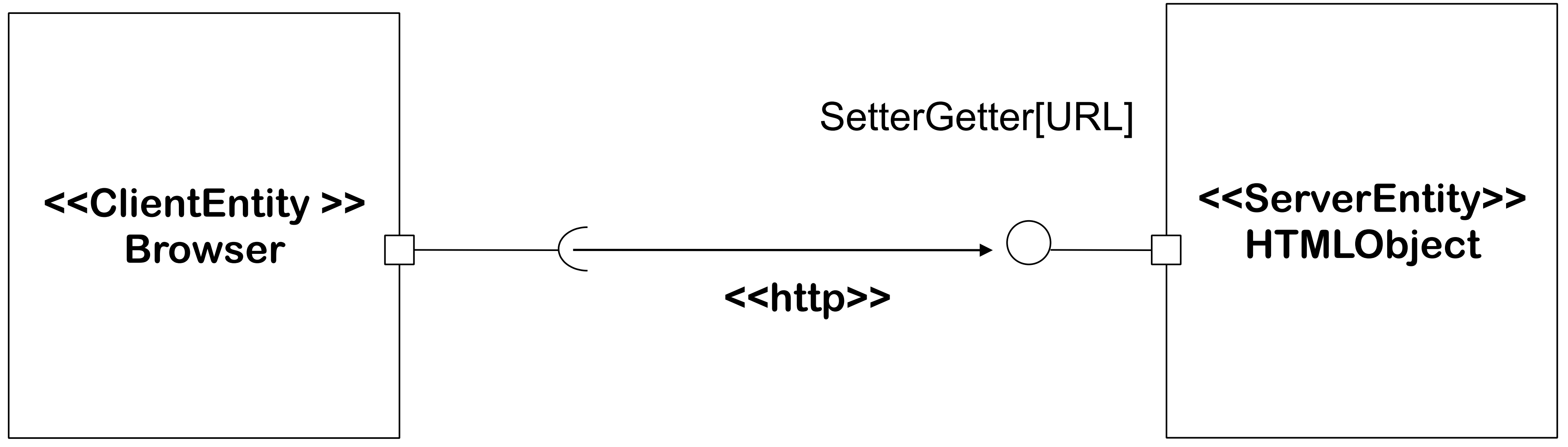
Connectors as Association Classes



<<component>> Client
<<component>> Server
<<connector>> BusinessConnector
<<component>> CORBA-Component
<<component>> .NETComponent
<<connector>> CORBA-Connector
<<connector>> HTTP-Connector
<<connector>> SOAP-Connector
<<component>> ServerPage

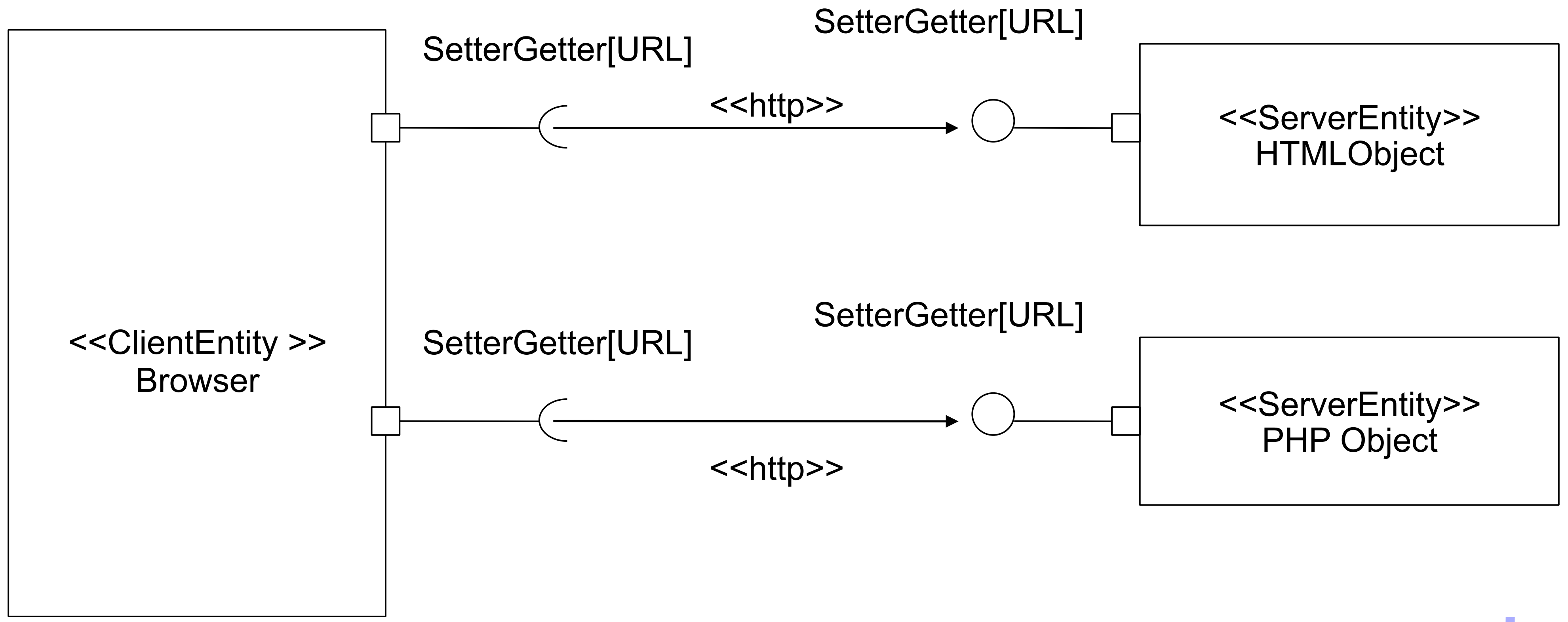
Example: Web Design with Connectors

- We can use a connector to express the relationship between web server and web client



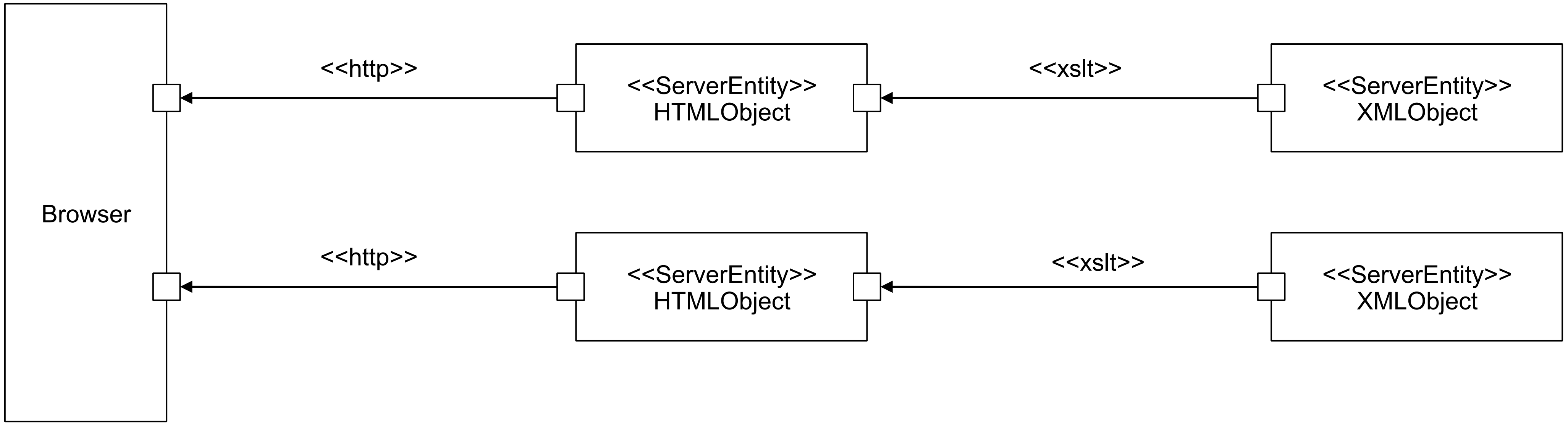
What are WebSites in UML?

- Nets of connections; every link, every cgi-call a connector



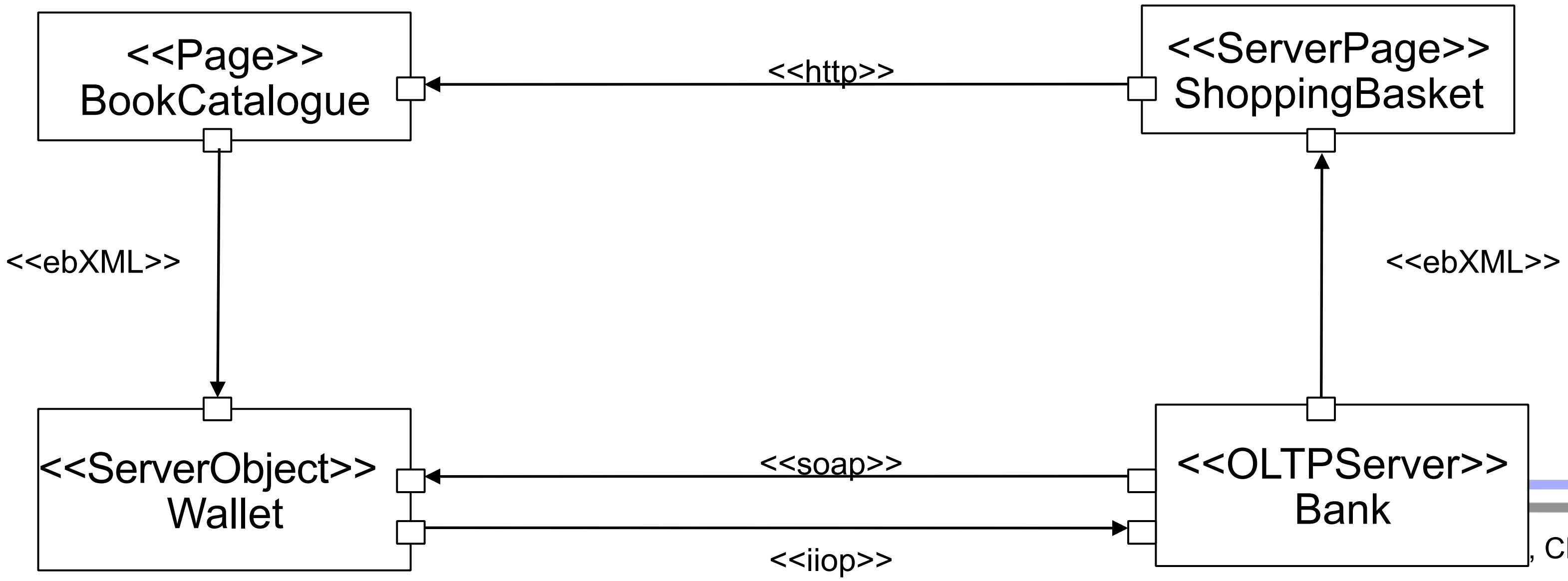
Consequences for Web Development

- With UML 2.0 and the connector concept, it is possible to describe the *architecture of web sites*
  - Frameworks for web sites become possible
  - A Content Management System (CMS) is a net of connections
  - Every transformation script a pipe connector


Browser
<<http>>
<<ServerEntity>> HTMLObject
<<xslt>>
<<ServerEntity>> XMLObject
<<http>>
<<ServerEntity>> HTMLObject
<<xslt>>
<<ServerEntity>> XMLObject

Architecture of Web Sites

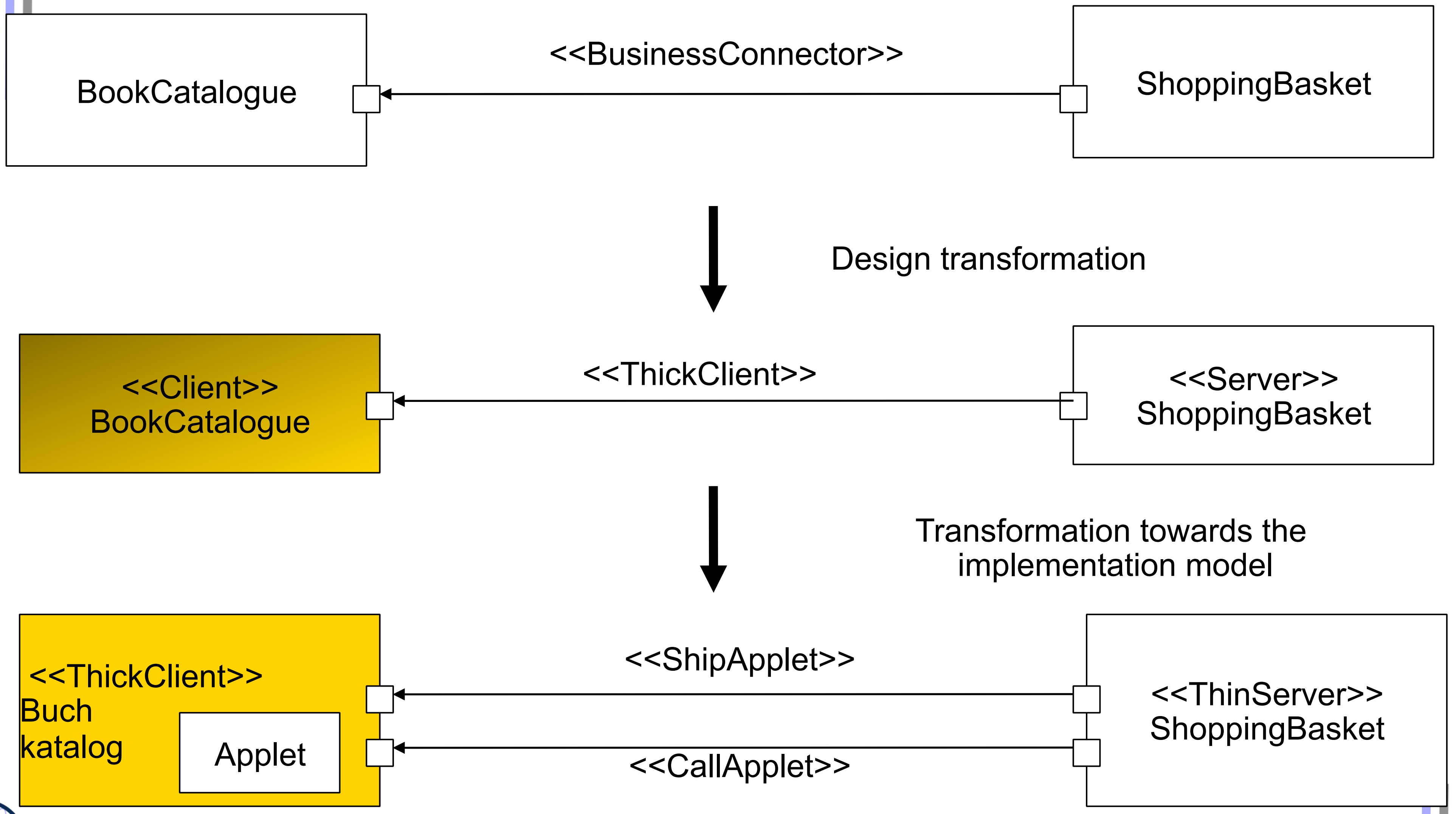
- Using connectors, web sites get an architecture
- Connectors abstract from the transfer mechanism
  - http, CORBA, IIOP, DCOM, SOAP, ebXML, XSLT-scripts via pipes and Sockets, ….
- With connectors, everything can be represented as connection nets
  - Uniform representation of links, scripts, protocols
  - Servers can no longer be distinguished from client browsers
  - Transfer mechanisms can be exchanged
- Clients and servers become scalable



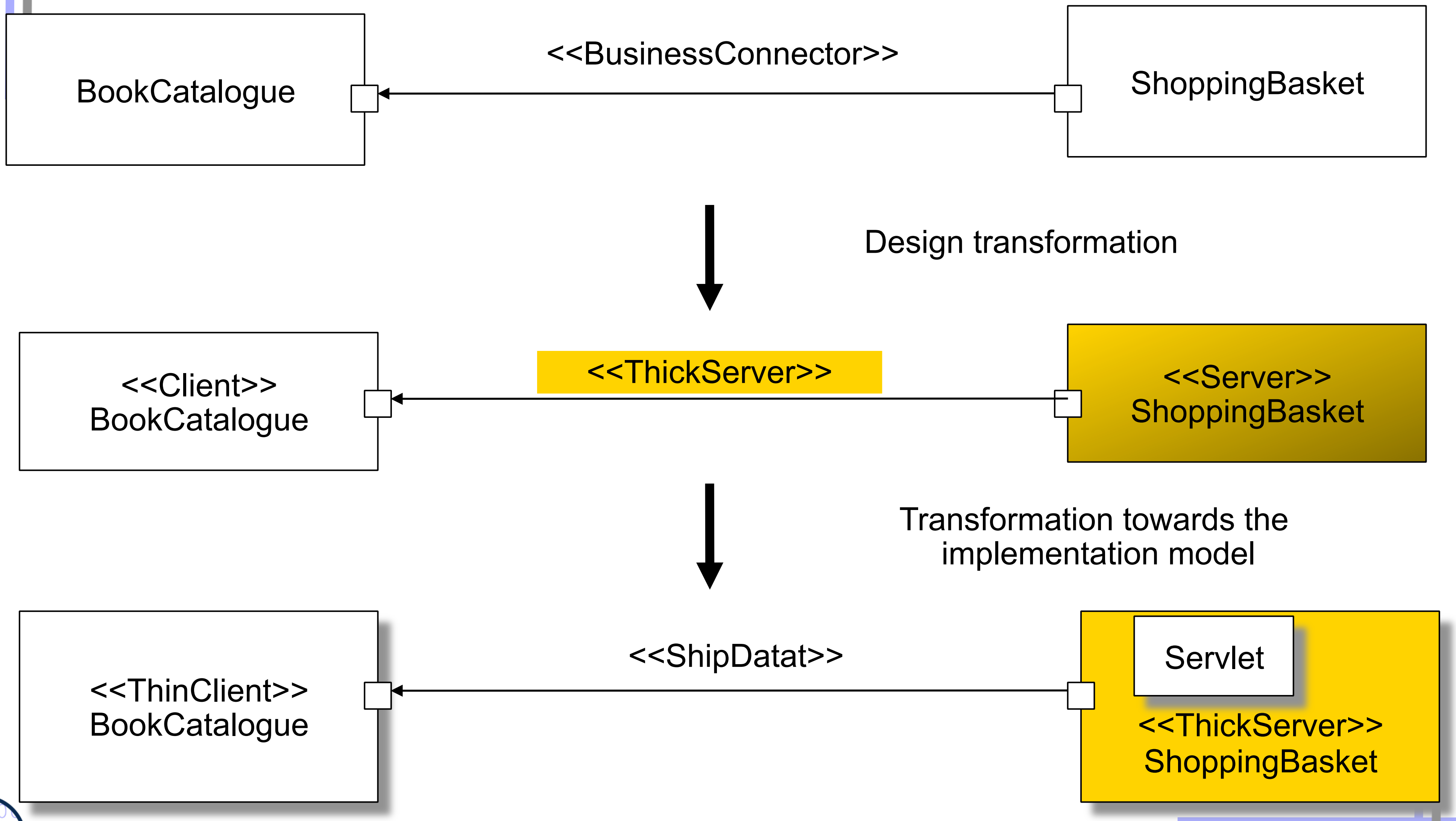
13.5.2 Example for Connector Refinement: The Thin/Thick Client – Problem in MDA

- Clients and servers become scalable
- Where should the computation go on?
  - Server?
    - Costly with large data sets
  - Client?
    - Costly with weak client
- Should be scalable
  - Thin Client / thick Server
  - Thick Client / thin Server
  - Dynamically exchangeable
- Solution: connectors on different abstraction levels in the MDA

Solution: Connectors in the MDA Example Shopping Basket



Solution 2: Thick-Server Connector



Solution Implementation

- Transform to a language with ports and connectors
  - ArchJava
- Transform to a connector library
  - Use Invasive software composition (ISC) e.g., with Reuseware or the COMPOST library
  - Write new connectors as metaprograms
- Tools for UML 2.0 will offer template-based code generation for connectors
  - Connectors are just special stereotypes
- UML-Profile editors will enable the construction of UML connector libraries

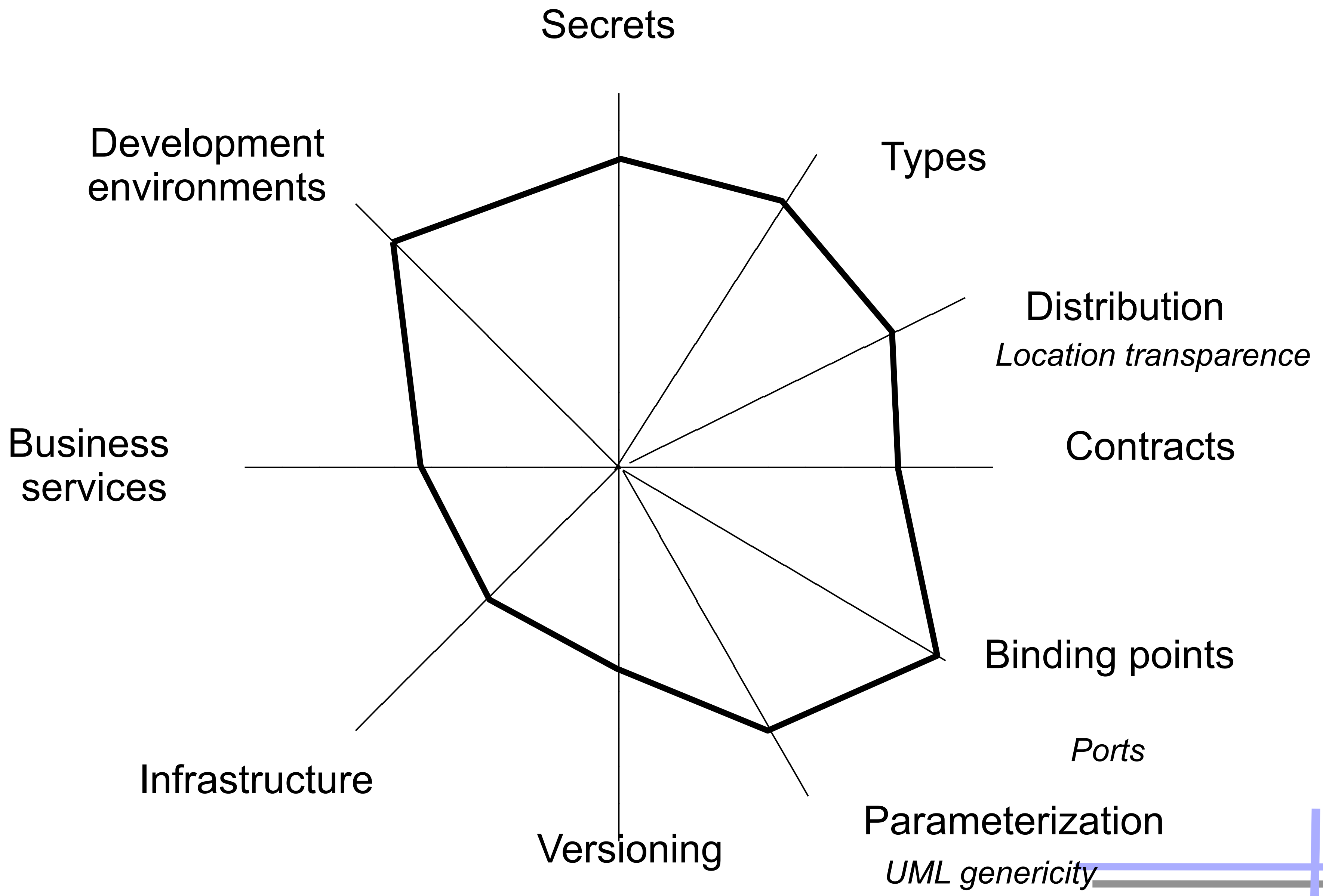
13.6 Architecture Systems: Evaluation

- 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

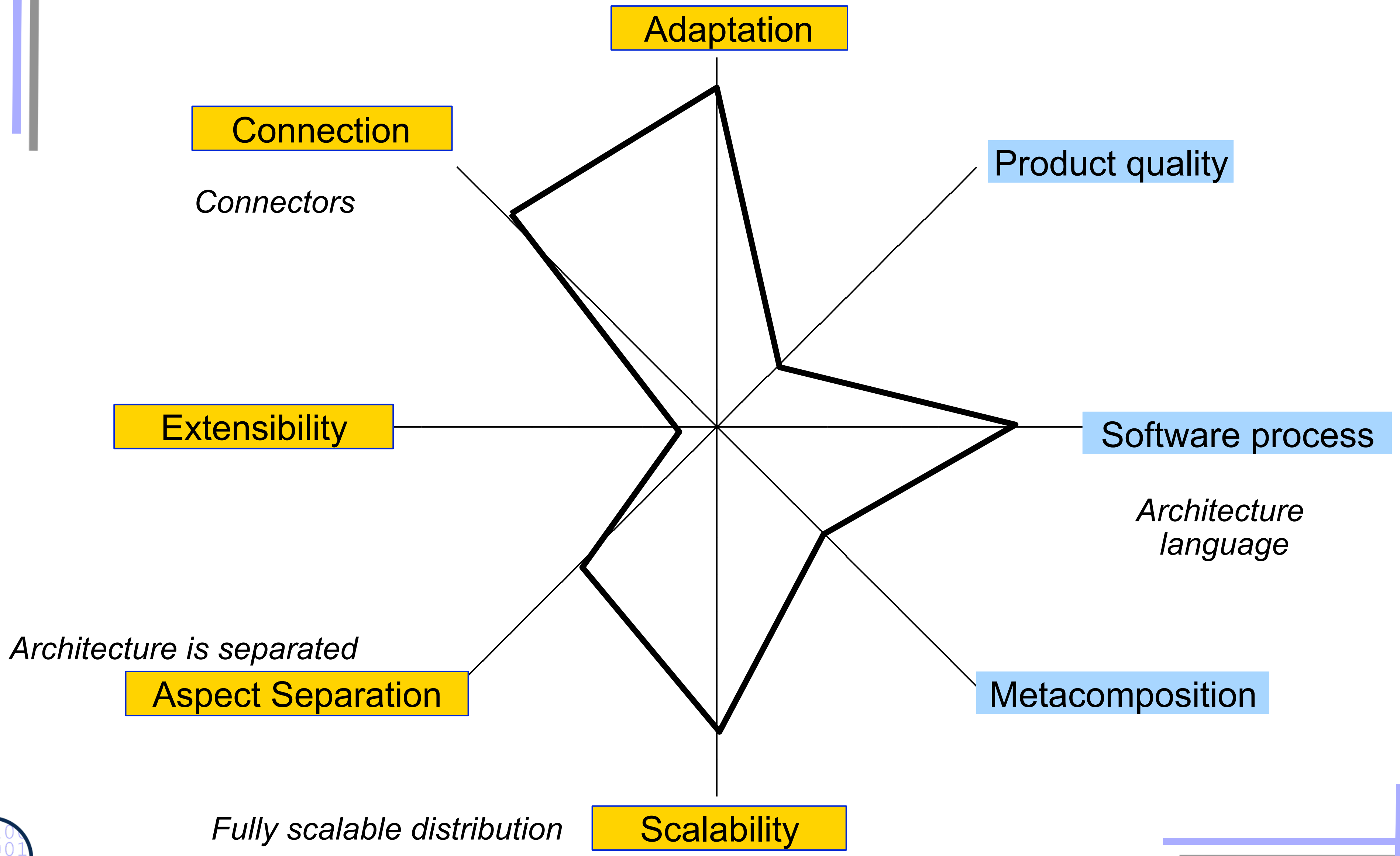
Architecture Systems - Component Model



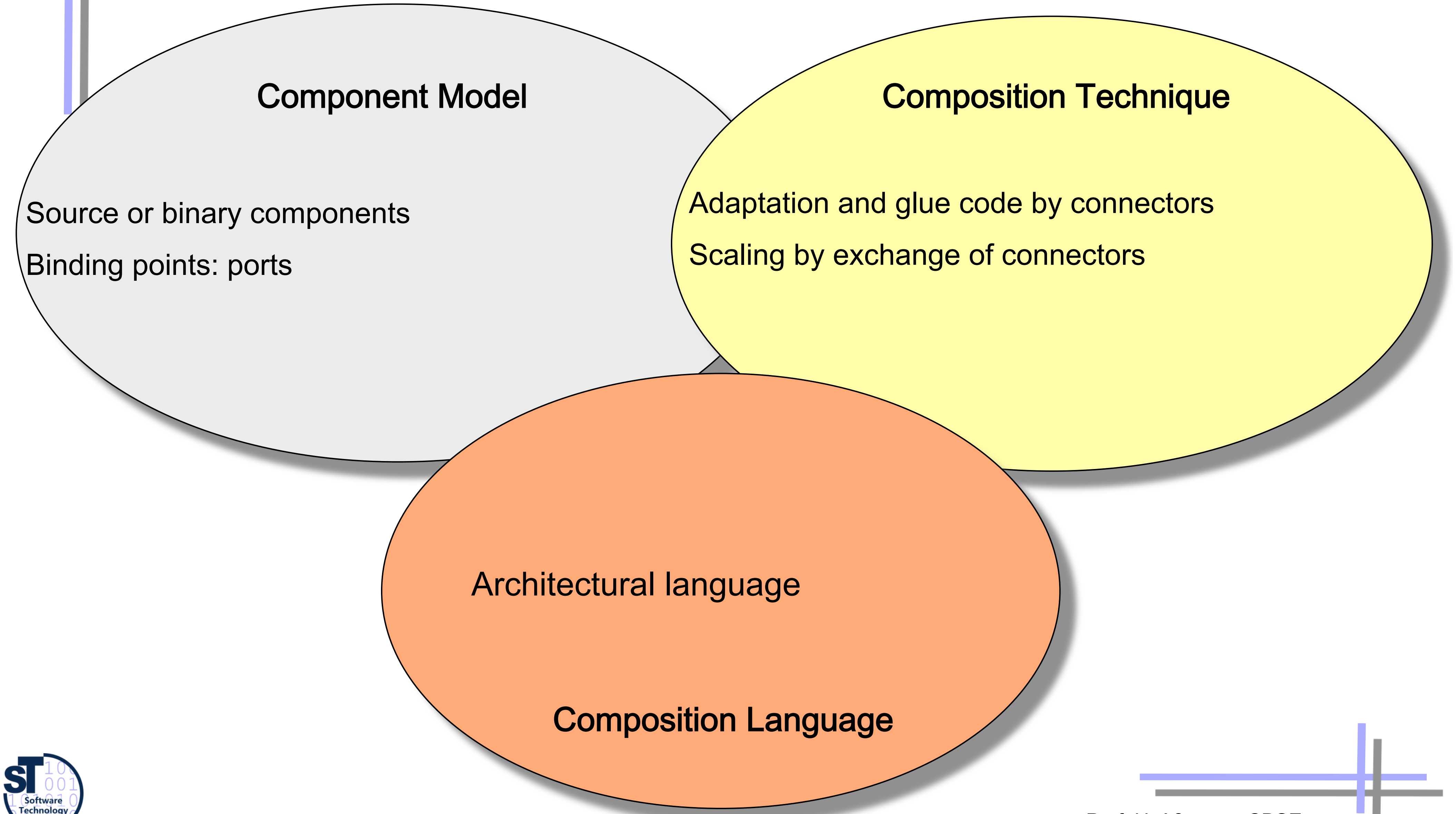
ADL: Mechanisms for Adaptation

- 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



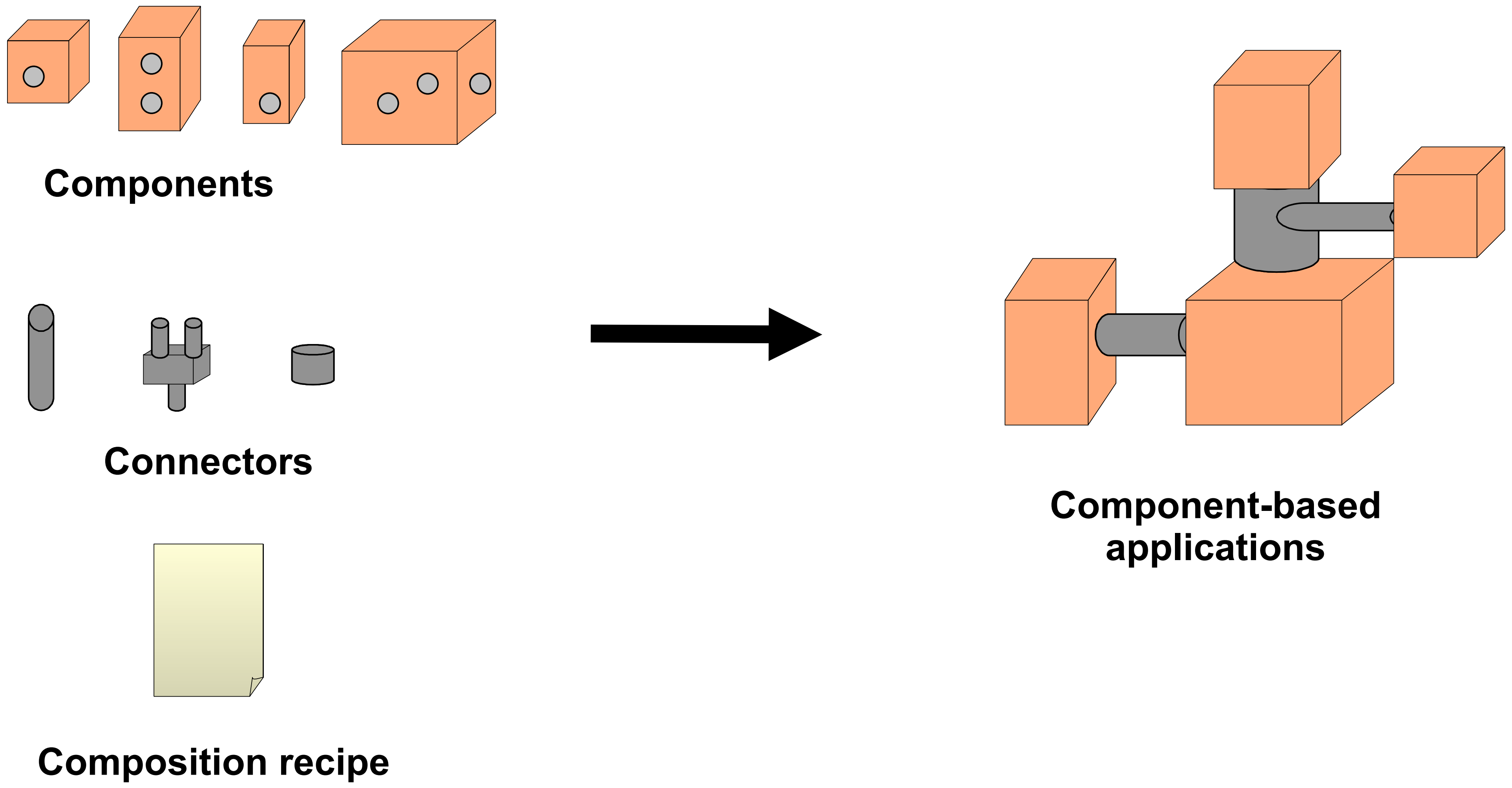
Architecture Systems as Composition Systems



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



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