# Software Architecture Systems

[Szyperski 21.1+24.1], and references on course home page

0. Motivation: Separate architecture aspect from application
1. Software Architecture Systems: Foundations
2. Case studies: Unicon, CoSy
3. Other architecture systems (some material for self-studies)
4. Modeling Software Architecture with UML and UML 2.0
5. Summary

TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Additional Literature


- **D. Garlan and M. Shaw, *An Introduction to Software Architecture.*** In V. Ambriola and G. Tortora (eds.), *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 1993, pp. 1-40. Nice introductory article. http://www-2.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html
- **M. Shaw, P.C. Clements: *A Field Guide to Boxology. Preliminary Classification of Architectural Styles for Software Systems.*** CMU, April 1996. http://citeseer.ist.psu.edu/shaw96field.html
- **C. Hofmeister, R. L. Nord, D. Soni. *Describing Software Architecture with UML.*** In P. Donohoe, editor, Proc. IFIP Working Conference on Software Architecture, pp. 145-160. Kluwer Academic Publishers, Feb. 1999.




TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Additional Literature (cont.)


- **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.*** Addison-Wesley, 2000. *Very nice book on architectural elements in UML.*
- **Rikard Land: *A Brief Survey of Software Architecture.*** MRTC report ISSN 1404-3041 ISRN MDH-MRTC-57/2002-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2002
- **Martin Alt. *On Parallel Compilation.*** PhD Dissertation, Universität des Saarlandes, Saarbrücken, Feb. 1997. *(CoSy prototype)*
- **ACE b.V. Amsterdam. *CoSy Compilers.*** System documentation, Apr. 2003. http://www.ace.nl




TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

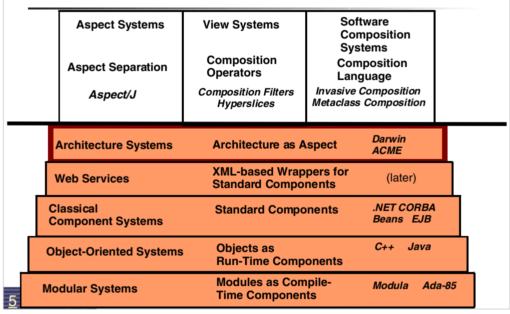
## Examples of Architecture Systems


- **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, pp. 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, pp. 717--734, Sept. 1995. (RAPIDE)
- **(Darwin)** http://www-dse.doc.ic.ac.uk/Software/Darwin/
- **Gregory Zelesnik. *The UniCon Language User Manual.*** School of Computer Science, Carnegie Mellon University Pittsburgh, Pennsylvania
- **Gregory Zelesnik. *The UniCon Language Reference Manual.*** School of Computer Science, Carnegie Mellon University Pittsburgh, Pennsylvania




TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## The Ladder of Component and Composition Systems






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Software Architecture

- **Software architecture**
  - Structural organization of an application's implementation (code) into software components and their interconnection
  - The first step in producing a software design [Garlan, Shaw 1996]
- **Basic ingredients**
  - Components (modules with interfaces)
  - Connectors (abstraction of communication)
  - Operators that create systems from subsystems
- **Software architecture systems**
  - Architecture description language (ADL)
    - For writing construction plans

## A Basic Rule for Design ...

- **... is to focus on one problem at a time and to forget about others.**
- ***Abstraction* is neglection of unnecessary detail**
  - Display and consider only essential information





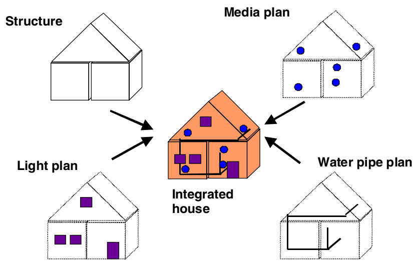
## Separation of Concerns

- **Different concerns should be separated**
  - so that they can be specified independently
- **Dimensional specifications**
- **Specify from different viewpoints**
- **But: different concerns are not always independent of each other**
  - Interferences
  - Consistency issues
  - Ordering constraints on application





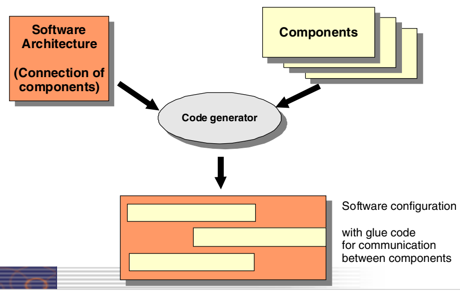
## Aspects in Architecture








## An Example of Separation of Concerns: Architectural Aspect in Software








## Software Architecture Systems as Composition Systems

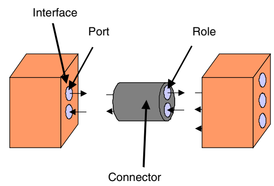
- **Component model**
  - Binding points: *Ports*
  - Communication between component instances is split off in *connectors*: Transfer (carrier) of the communication is transparent
- **Composition technique**
  - Adaptation and glue code generated from *connectors*
  - Aspect separation: application and communication are separated
    - Topology (who communicates with whom?)
    - Carrier (how?)
    - When?
  - Scalability (distribution, binding time with dynamic architectures)
- **Composition language:** An *Architecture Description Language* (ADL) is a simple composition language!





## Component Model in Architecture Systems

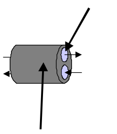
- ***Ports*** = abstract interface points (events, methods)
- Ports specify the data-flow into and out of a component
  - in(data)
  - out(data)
- ***Connectors*** as special communication components
  - Connectors are attached to ports
  - Connectors are explicitly applied per communication




Components and connectors are bound together to form a ***configuration.***

## Abstract Binding Points: Ports

- **Ports abstract from the concrete carrier, but indicate where data has to flow in and out of the component**
  - To fit to connectors, a legacy system must convert all procedure calls to ports, *i.e.*, to abstract calls
  - Ports have protocols
- **Connectors can be binary or *n*-ary**
  - Every end is called a *role.*
  - Roles fit only to certain types of ports = Typing of roles and ports.
- **The interfaces remain at run time**



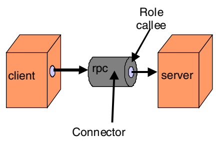



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## A Simple Example

- **A description of a small example architecture in the ADL *Acme*** [Garlan et al., CMU, 2000]

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc = { Roles { caller, callee } }
  Attachments : {
    client.sendRequest to rpc.caller ;
    server.receiveRequest to rpc.callee ;
  }
}
```

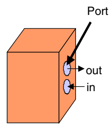





TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Ports In More Detail

- **Input 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 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






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

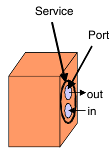
## Ports and Services

- ***Services*** are groups of ports.
- A ***data service*** is a tuple

  [in(data), ..., in(data), out(data), ..., out(data)]
- A special case is a ***call service*** with one return port:

  [in(data), ..., in(data), out(data)]
- A ***property service*** is a service to access component attributes, i.e., a simple tuple

  [in(data), out(data)]






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Architectural Styles

e.g. [Garlan/Shaw: Software Architecture, Prentice-Hall 1996]

- **Frequently occurring connection topology patterns (Architectural Design Patterns)**
  - Pipe-and-Filter
    - UNIX shells
    - Stream-parallel programming languages
  - Client-Server Architecture
    - CORBA RPC, Java RMI, ...
  - Layered Architecture (aka. Onion Architecture)
    - Layered operating systems (UNIX, Windows)
    - Multi-tier architectures (e.g. 3-tier: clients / server objects / DB)
  - Blackboard Architecture (aka. Repository Architecture)
    - Linda [Carriero/Gelernter'96]
    - Service discovery repositories, e.g. Jini, CORBA repositories
    - CoSy CCMIR

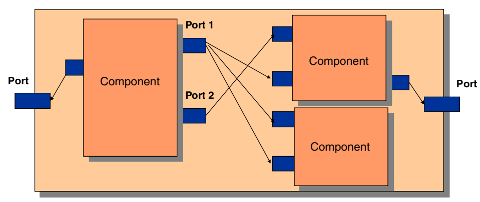
.... and more, and combinations of these



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

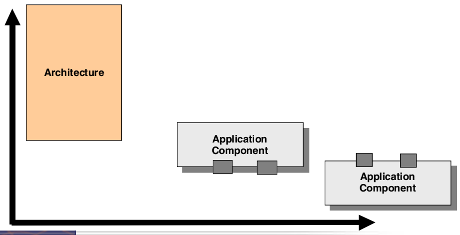
## Architecture can be Exchanged Independently of Components

- "Rewiring"
- Reuse of components and architectures is fundamentally improved



## Two Dimensions of Reuse

- Architecture and components can be reused independently of each other

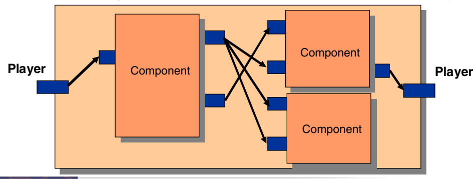







## Architecture Descriptions are Reducible

- Components are nested (fractal-like behavior)
- Ports of outer components are called ***players.***
- This type of diagram is now supported in UML 2.0 as ***component diagram***

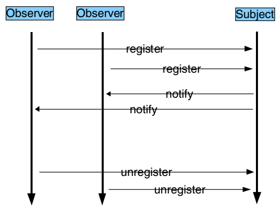







## Additionally, Connectors have Protocols

- **A connector, since it is a precise concept to specify communication of components, must have a *protocol***

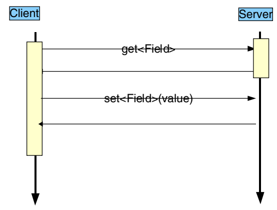







## Set/Get Connector Protocol

- **on data services**

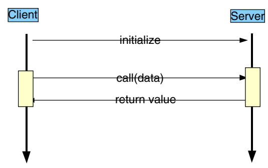







## Call Connector Protocol

- **on call services**

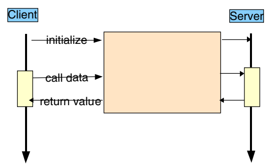




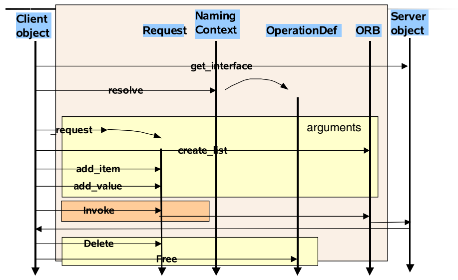



## RPC Connector

- **on call services**



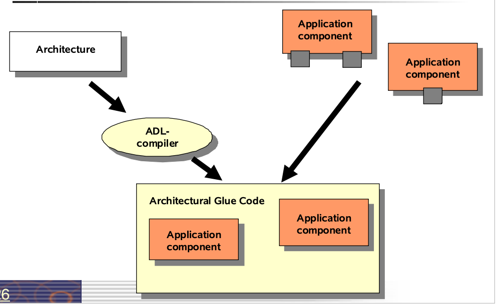
## Dynamic Call via CORBA DII - Protocol







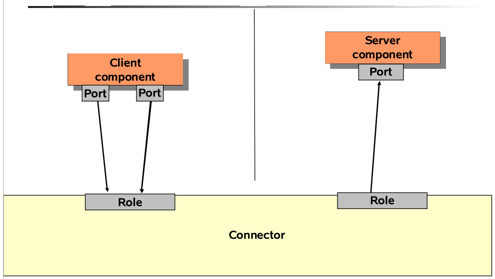

## From Connectors in ADL Specification Generate Architectural Glue Code








## Connectors are Abstract Communication Buses

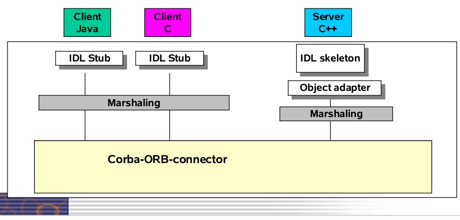







## But we know that already from CORBA:

- CORBA is a simple architecture system with restricted connectors:








## CORBA is a Simple Architecture System with Restricted Connectors

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

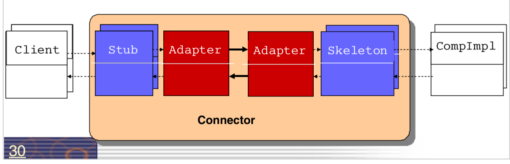
**Architecture Systems:**
- Components
- Connectors
- Roles
- Ports
- procedure call connector (also distributed)
- dynamically reconfigurable connectors (*e.g.*, in Darwin)
- connectors *n*-ary
- Events, callbacks, persistence as connectors (can be exchanged to other communications)





## 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



## Architecture Systems

**Examples**

- **Unicon [Shaw 95]**
- **Aesop [Garlan95]**
- **Darwin [Kramer 92]**
- **Rapide [Luckham95], C2 [Medvedovic]**
- **Wright [Garlan/Allen]**
- **ACME [Garlan 2000]**

- **CoSy [Aßmann/Alt/vanSomeren'94] www.ace.nl**



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Example: The KWIC Problem in UNICON
[ISC pp. 74-76]

- **Example from UniCon distribution**
- **"Keyword in Context" problem (KWIC)**
  - The KWIC problem is one of the 10 model problems of architecture systems
  - Originally proposed by Parnas to illustrate advantages of different designs [Parnas'72]
  - 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.



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## A KWIC 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 |



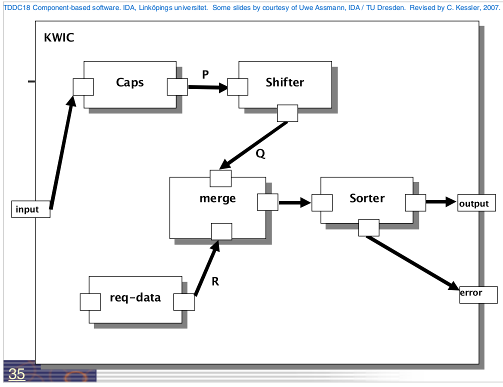
TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## The KWIC Problem in Unicon

- **The components of KWIC work in a pipe-and-filter style**
- **KWIC has ports**
  - stream input port *input*,
  - and two output ports *output* and *error*. They read text and spit out the permuted index
- **KWIC is a compound component KWIC** (Components in Unicon can be nested)
  - PLAYER definitions define ports of outer components.
  - BIND statements connect ports from outer components to ports of inner components.
- **USES definitions** create instances of components and connectors.
- **CONNECT statements** connect connectors to ports at their roles.



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## The KWIC Problem in Unicon

- **Components**
  - The component *caps* converts the sentence to uppercase as necessary.
  - The *shifter* creates permutations of the sentence.
  - The *req-data* provides some data to the *merge* component which pipes the generated data to the component *sorter*.
  - *sorter* sorts the shifted sentences so that they form a keyword-in-context index.
- **Only connectors in the style of UNIX pipes are used**
  - Other connection kinds can be introduces by only changing the type of connectors in a USES declaration.
  - Hence, communication kinds can be exchanged easily, e.g. for Shared memory, Abstract data types, Message passing [Garlan/Shaw'94]
- **Architecture systems allow for scalable communication: binding procedures can be exchanged easily!**

## KWIC in Unicon

```
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 for anonymous 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
```





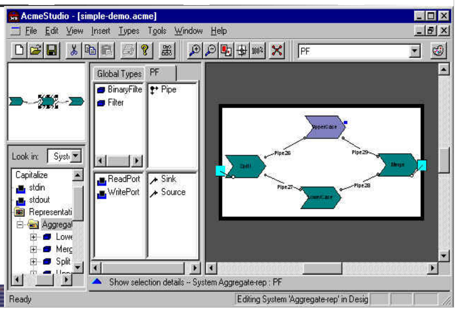
## The Composition Language: ADL

- **Architecture language (architectural description language, ADL)**
  - ADL-compiler
  - XML-Readers/Writers for ADL.
- **The reducibility of the architecture allows for simple overview, evolution, and documentation**
  - The architecture is a reducible graph, with all its advantages
- **Graphic editing of systems**





## ACME Studio








## What ADL Offer for the Software Process

- **Support when doing the requirements specification**
  - Visualization for the customer: architecture graphics better to understand
  - Architecture styles classify the nature of a system in simple terms
- **Design support**
  - Simple specification by graphic editors
  - Stepwise design and refinement of architectures
  - Visual and textual views
- **Design of product families is easy**
  - A *reference architecture* fixes the commonalities of the product line
  - The components express the variability





## Checking and Validating

- **Checking, analysing**
  - Test of (part of) an architecture with dummy components
  - Deadlock checking
  - Liveness checking
- **Validation: Tools for consistency of architectures**
  - Are all ports bound?
  - Do all protocols in the connectors fit?
  - Does the architecture correspond to a certain style ?
  - Does the architecture fit to a reference architecture?
  - Parallelism features as deadlocks, fairness, liveness,
  - Dead parts of the systems: Is everything reachable at run time?

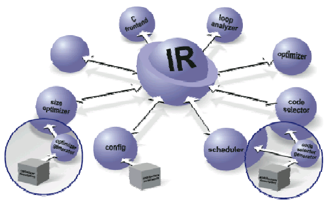




## What can be generated?

- **Glue- and adapter code from connectors and ADL-specifications**
  - Mapping of the protocols of the components to each other
  - Generation of glue code from the connectors
- **Simulations of architectures (with dummy components):**
  - The architecture can be created first
  - And tested stand-alone
  - Run time estimates are possible (if run times of components are known)
- **Test cases for architectures**
- **Documentation** (graphic structure diagrams)

# CoSy



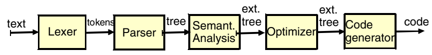

A commercial architecture system for compilers

www.ace.nl

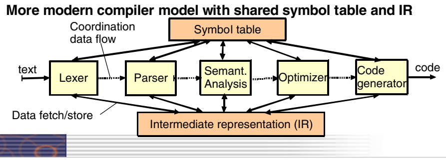
TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Traditional Compiler Structure

- **Traditional compiler model: sequential process**



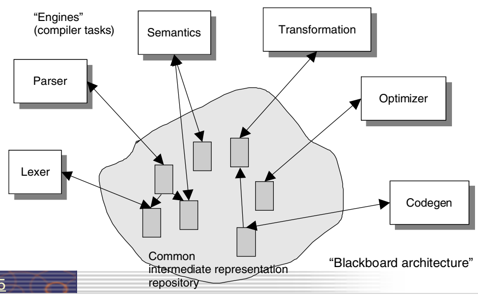

- **Improvement: Pipelining (by files/modules, classes, functions)**
- **More modern compiler model with shared symbol table and IR**






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

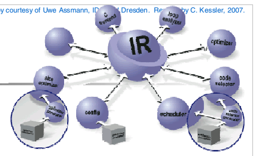
## A CoSy Compiler with Repository-Architecture






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Engine




- Modular compiler building block
- Performs a well-defined task
- Focus on algorithms, not compiler configuration
- Parameters are handles on the underlying common IR repository
- Execution may be in a separate process or as subroutine call - *the engine writer does not know!*
- *View* of an engine class: the part of the common IR repository that it can access (scope set by access rights: read, write, create)

Examples: Analyzers, Lowerers, Optimizers, Translators, Support



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Composite Engines in CoSy

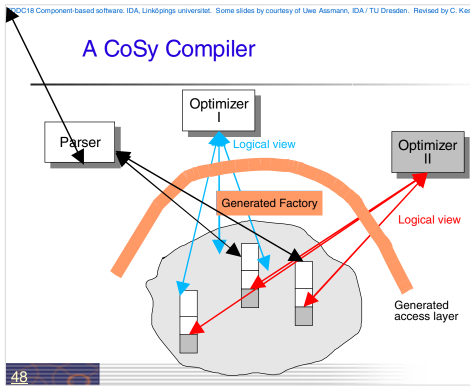
- **Built from simple engines or from other composite engines** by combining engines in interaction schemes **(Loop, Pipeline, Fork, Parallel, Speculative, ...)**
- **Described in EDL (Engine Description Language)**
- **View defined by the joint effect of constituent engines**
- **A compiler is nothing more than a large composite engine**

```
ENGINE CLASS compile (IN u: mirUNIT) {
  PIPELINE
    frontend (u)
    optimizer (u)
    backend (u)
}
```

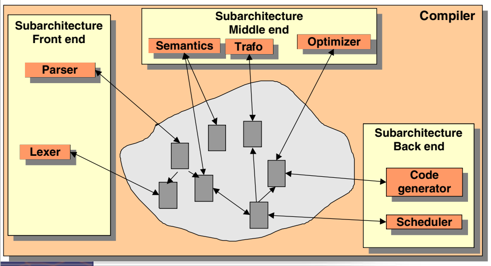


TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## A CoSy Compiler



## Hierarchical Components in the Repository Style (CoSy)








## Example for CoSy EDL (Engine Description Language)

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

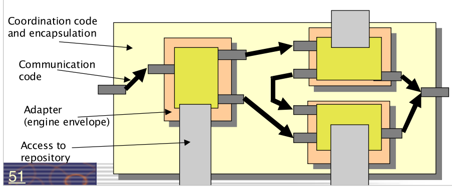
```
ENGINE CLASS optimizer ( procedure p ) {
    controlflowAnalyser cfa;
    commonSubExprEliminator cse;
    loopVariableSimplifier lvs;
    PIPELINE cfa(p); cse(p); lvs(p);
}
ENGINE CLASS compiler ( file f ) {
  .... Token token;
  Module m;
  PIPELINE  // lexer takes file, delivers token stream:
        lexer( IN f, OUT token<> );
        // Parser delivers a module
        parser( IN token<>, OUT m );
        sema( m );
        decompose( m, p<> );
        // here comes a stream of procedures
        // from the module
        optimizer( p<> );
        backend( p<> );
}
```





## Adapter (Envelope, Container)

- **CoSy generates for every component an adapter (envelope, container)**
  - that maps the protocol of the component to that of the environment (all combinations of interaction schemes are possible)
  - Coordination, communication, encapsulation and access to the repository are generated.








## Evaluation of CoSy

- CoSy is one of the single commercial architecture systems with professional support
- The outer call layers of the compiler are generated from the ADL
  - Adapter, coordination, communication, encapsulation
  - Sequential and parallel implementation can be exchanged (cf. skeletons)
  - There is also a non-commercial prototype [Martin Alt: *On Parallel Compilation.* PhD thesis, 1997, Univ. Saarbrücken]
- Access layer to the repository must be 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





# Appendix: Survey of Other Architecture Systems

- **For self-studies...**
  - UniCon
  - RAPIDE
  - Aesop
  - Acme
  - Darwin



## An Example System: UNICON

- **UNICON supports**
  - Components in C
  - Simple and user-defined connectors
- **Design Goals**
  - Practical tool for real problems
  - Uniform access to a large set of connections
  - Check of architectures (connections) should be possible
  - Analysis tools
  - Graphics and Text
  - Reuse of existing legacy components
  - Reduce additional run time costs

## Description of Components and Connectors

- **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
```



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## 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
```



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## 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
```



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## RAPIDE

- Luckham/Vera/Meldal. *Three Concepts of System Architecture.* Stanford University 1995.
- **Central idea:**
  Rapide leaves the ***object connection architecture***, in which the objects are attached to each other directly, for an ***interface connection architecture***, in which *required and provided* interfaces are related to each other
  - Specify in a interface not only the required methods, but also the offered ones (provided and required ports)
  - Connect the ports in a architecture description (separate)
  - Advantage: calls can be bound to other ports with different names
  - Generalizes ports to calls
- **Fundamentally more flexible concept for modules!**
- Rapide was marketed by a start-up company



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

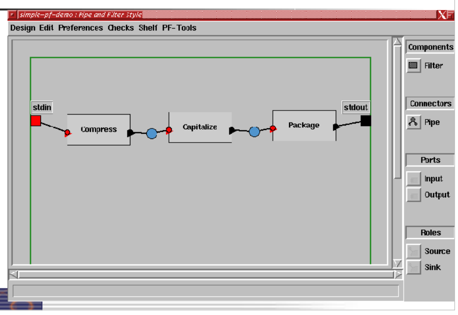
## Aesop

- **Connectors are first class language elements**
  i.e., can be defined by users
  - Connectors are classes which can be refined by inheritance
- **Users can derive their own connectors from system connectors**
- **Aesop supports the definition of architectural styles with *fables***
  - Architectural styles obey rules



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Pipe-Filter Visual in Aesop



## Aesop Supports Architectural Styles (Fables)

- **Design Rule**
  - A *design rule* is an element of code with which a class extends a method of a super class. A design rule consists of the following:
    - A *pre-check* that helps control whether the method should be run or not.
    - A *post-action*
- **Environment**
  - A design environment tailored to a particular *architectural style*.
    - It includes a set of policies about the style, and 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
  - If it is part of the presentation to the user, it should be part of the environment.





## 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

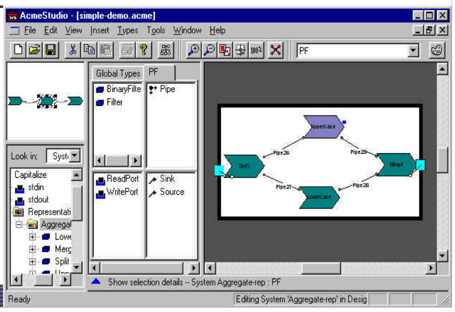
Features

```
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 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 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};
};
```





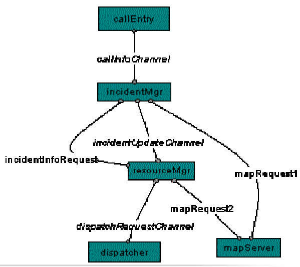
## Instance of an ACME System

```
// 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;
  };
```



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## London Ambulance System in ACME (cont.)

```
// 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;
  };
 }
```



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

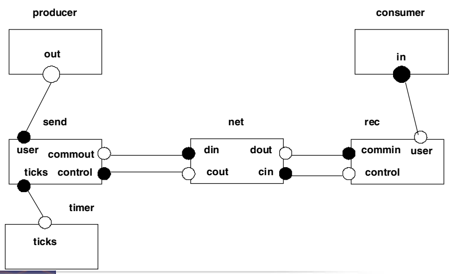
## Darwin (Imperial College)

- **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**



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Simple Producer/Consumer






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Simple Producer/Consumer in Text

```
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;

}
```



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

# Architectural Languages in UML

Hofmeister, Nord, Soni:
Describing Software Architecture with UML.
1999

## Architecture Languages versus UML

- **So far, architecture systems and languages were research toys** (except CoSy)
- **"I have to learn UML anyway, should I also learn an ADL??"**
  - Learning curve for the standard developer
  - Standard?
  - Development environments?
- **This changes with UML 2.0**





## The Hofmeister Model of Architecture

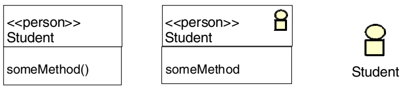
- [Hofmeister/Nord/Soni'99] **is the first article that has propagated the idea of specifying an architecture language with UML**
  - Conceptual view: Functionality + interaction (components, ports, connectors)
  - Module view: Layering, modules and their interconnections
  - Execution view: runtime architecture (mapping modules to time and resources)
  - Code view: division of systems into files
- **Describe these single views in UML**
  - 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



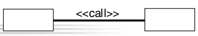


## Background: Stereotypes in UML

- **A *stereotype* is a UML modeling element introduced at modeling time. It represents a subclass of an existing modeling element (->metalevel) with the same form (attributes and relationships) but with a different intent, maybe special constraints.**




- **To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype.**
- **A mechanism for extending/customizing UML without changing it.**
- [UML Notation Guide, 1997]








## Modeling software architectures in UML

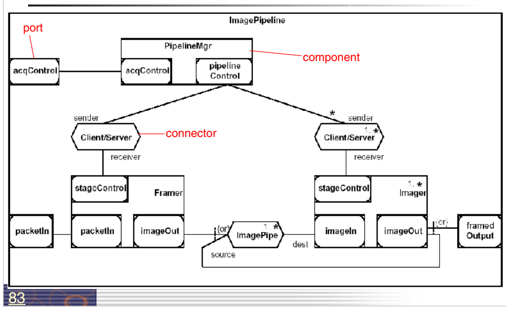
**Example scenario:** [Hofmeister/Nord/Soni'99]

- **Digital camera**
  produces sequence of image frames,
  flattened into a stream of pixel data
- **Image acquisition system**
  selects, starts, adjusts an image acquisition procedure
- **Image processing pipeline**
  - Framer: Restore complete image frames from pixel stream
  - Imager: One or more image transformation(s)
- **Display images**





## UML model for image processing example







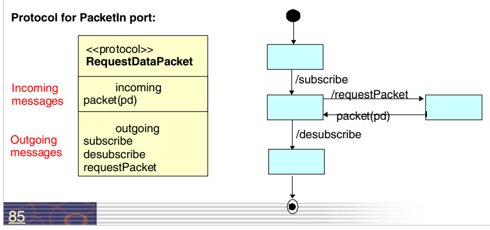

## Modeling software architecture in UML with the Hofmeister/Nord/Soni approach

- **For conceptual view: Class diagram**
- **Components, ports, connectors are a stereotype of Class:**
  <<component>>, <<port>>, <<connector>>
- **Use special symbols for ports and connectors**
- **Omit the stereotype for components and show their associations with their ports by nesting**
- **Roles are a stereotype of Association:**
  <<role>>
  - shown as labels on port-connector associations
  - Default multiplicity is 1

# Modeling software architecture in UML

- **For modeling protocols, use UML Sequence diagram or State diagram**

**Protocol for PacketIn port:**

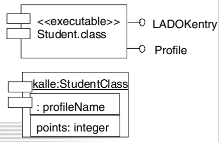







# Component Diagrams in UML 2.0

- **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 ..."
  - Provided and required interfaces
  - Substitutable
  - Run-time representation of one or several classes
  - Source or binary code
- **Difference to UML classes:**
  - No inheritance
- **New symbols**
  - Components, component instances
  - New UML element, not a stereotype

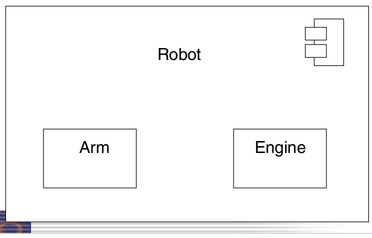







# Component Diagrams in UML 2.0

- **Components can be nested**

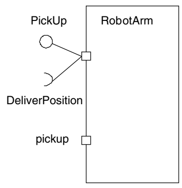







# Ports in UML 2.0 Component Diagrams

- **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

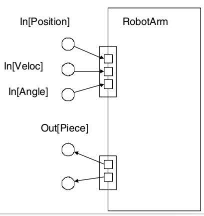







# Services

- **Ports can be grouped to Services**

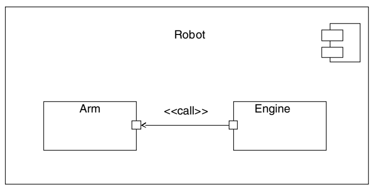




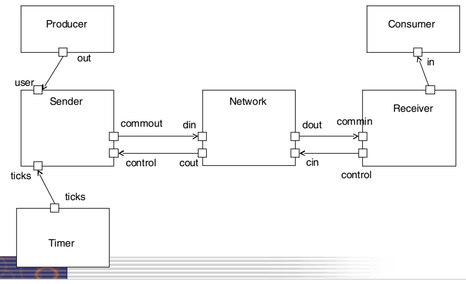



# Connectors in UML 2.0

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



## Simple Producer/Consumer in UML 2.0

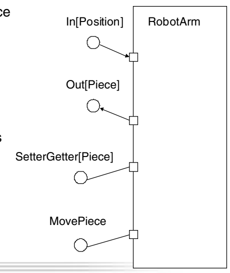







## Exchangeability of Connectors

- 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 Thumb 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



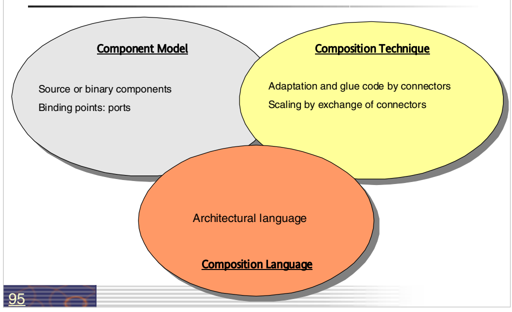


# Architecture Systems: Summary

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



## Architecture Systems as Composition Systems



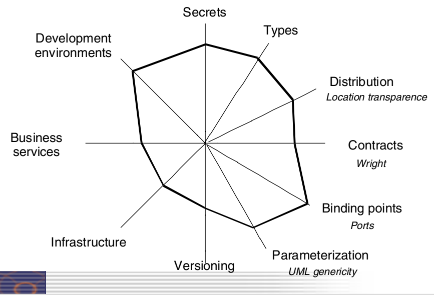





## ADL: Mechanisms for Modularization

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

## Architecture Systems - Component Model






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

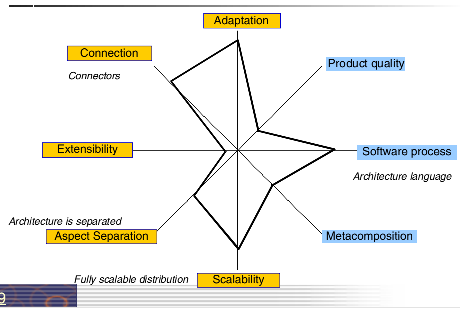
## ADL: Mechanisms for Adaptation

- **Connectors generate glue code: very good!**
  - Many types of glue code possible
  - User definable connectors allow for specific glue
  - Tools analyze the interfaces and derive the necessary adaptation code automatically
- **Mechanisms for aspect separation. 2 major aspects are distinguished:**
  - Architecture (sub-aspects: topology, hierarchy, communication carrier)
  - Application functionality
- **An ADL-compiler is only a rudimentary weaver**
  - Aspects are not weaved together but encapsulated in glue code



TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## Architecture Systems – Composition Technique and Language

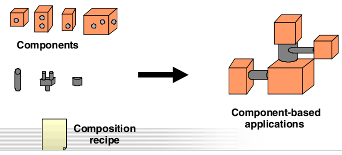





TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## What Have We Learned?

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






TDDC18 Component-based software. IDA, Linköpings universitet. Some slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2007.

## How the Future Will Look Like

- **Metamodels of architecture concepts (with MOF in UML) will replace architecture languages**
  - The attempts to describe architecture concepts with UML are promising
- **Model-driven architecture**
  - Increasingly popular, also in embedded / realtime domain
- **We should think more about general software composition mechanisms**
  - Adaptation by glue is only a simple way of composing components (... see invasive composition)