Software Architecture Systems

<table>
<thead>
<tr>
<th>[Sørensen 21.1–24.1], and references on course home page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. Motivation: Separate architecture aspect from application</td>
</tr>
<tr>
<td>2. Case studies: Unix, CompoDy</td>
</tr>
<tr>
<td>3. Other architecture systems (some material for self-studies)</td>
</tr>
<tr>
<td>4. Modeling Software Architecture with UML and UML 2.0</td>
</tr>
<tr>
<td>5. Summary</td>
</tr>
</tbody>
</table>

Additional Literature (cont.)


Examples of Architecture Systems


The Ladder of Component and Composition Systems

- Architecture Systems: Architecture as Aspect
  - Design: ADM1
  - XML-based Wrappers for Standard Components
  -疥

- Web Services: SOAP Components
  - Standard Components
  - SOAP Complex

- Object-Oriented Systems: Object-Oriented Components
  - Non-Nothing Components
  - 0+1 (Java)

- Modular Systems: Modular as Compile-Time Components
  - Modula, Ada-93

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 1998]

  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 neglect 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

Structure
- Media plan
- Light plan
- Water pipe plan
- Integrated house

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
  - Adapter 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.

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 ;
  }
}

Ports In More Detail

- Input ports are synchronous or asynchronous:
  - in(data) (aka. received data): Synchronous in port, taking in one data
  - testAndGet(data): Synchronous in port, taking in one data if it is available

- Output ports are synchronous or asynchronous:
  - out(data): Synchronous out port, putting out one data, waiting until acknowledge
  - put(data): Asynchronous out port, putting out one data, not waiting until acknowledge

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)]

Architectural Styles

- Frequently occurring connection topology patterns (Architectural Design Patterns)
  - Pipe-and-Filter
  - Client-Server Architecture
  - CORBA RPC, Java RMI, ...
  - Layered Architecture (aka. Onion Architecture)
  - Layered operating systems (UNIX, Windows)
  - Multi-tier architecture (e.g. 3-tier: clients / server objects / DB)
  - Blackboard Architecture (aka. Repository Architecture)
  - Linda (Computer Graphics)
  - Service discovery repositories, e.g., Jini, CORBA repositories
  - CoSy, CCMIR, and more, and combinations of these

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

Most Commercial Component Systems Provide Restricted Forms of Connectors
### Architecture Systems

#### Examples
- Unicon [Shaw 95]
- Aesop [Garlan 95]
- Darwin [Kramer 92]
- Rapide [Luckham 95], C2 [Medvedovic]
- Wright [Garlan/Allen]
- ACME [Garlan 2000]
- CoSy [Assmann/Alt/vanSomeren'94] [www.ace.nl]

### Example: The KWIC Problem in Unicon

#### [ISC pp. 74-76]

**Example from UniCon distribution**

The **“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.

#### 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

#### The KWIC Problem in Unicon

- **KWIC has ports**
  - stream input port, input
  - and two output ports, output and error

- **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.**

- **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.
  - The 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 introduced by only changing the type of connectors in a USES declaration.
  - Names, 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!
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
CONNECT sorter.input TO R.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

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
- Validation: Tools for consistency of architectures
  - Are all ports bound?
  - Do all protocols in the connectors fit?
  - Does the architecture respect a certain style?
  - Do the architecture fit to a reference architecture?
  - Parallelism features as deadlocks, fairness, liveness,
  - Deadparts 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)
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

A CoSy Compiler with Repository-Architecture

- 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!

Engine

- 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

A CoSy Compiler
Hierarchical Components in the Repository Style (CoSy)

- Compliance with the Repository Style (CoSy)
  - Subarchitecture: Front end, Middle end, Back end
  - Coordinator: Emitter, Receiver, Adapter, Communication, Access to the repository

Adapter (Envelope, Container)

- 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.

Example for CoSy EDL (Engine Description Language)

```
Example CoSy EDL:

END_PROCEDURE
END_KLASS
END_KLASS
END_DEV
```

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)

Appendix: Survey of Other Architecture Systems

- For self-studies:
  - UniCon
  - RAPIDE
  - Acop
  - Acme
  - Dawn

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 architecture (connections) should be possible
  - Analysis tools
  - Graphs 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 reading layers)
  - Roles for connectors
- The UNICON-compiler generates
  - Collection of Global assertions in form of a feature list (property list)

Supported Role Types For Connector Types

- Pipe:
  - Source fits to Filter.StreamOut, ReadFile, Writer
  - Sink fits to Filter.StreamIn, WriteFile, Reader
- FileIO:
  - Reader fits to modules, ReadFile, ReadNext
  - Writer fits to Modules.WriteFile, WriteNext
- ProcedureCall:
  - User fits to (Computation|Module): ProcedureDef
  - User fits to (SharedData|Computation|Module): GlobalDataDef, GlobalDataUse
  - Callr fits to modules, computation, SeqFile, Filter, process, general
  - Caller fits to modules, computation, PLBandler, RPC, RTScheduler

A Filter

COMPONENT Reverser INTERFACE IS
  /* Component interface. */
  TYPE error
  SIGNATURE (error; void)
  /* Error interface. */
  /* Global interface. */
  /* Interface for file handling. */
  /* Interface for code generation. */
END INTERFACE

A Modules Component

LIBRARY modules
  PLAYER timeshow IS
    Interface:
      SIGNATURE (new_type; void)
  /* TimeShow interface. */
  PLAYER timeget IS
    Interface:
      SIGNATURE (void)
  /* TimeGet interface. */
END INTERFACE

Supported Player Types per Component Type

- Modules:
  - RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBandler
  - FunctionDef, FunctionCall, GlobalDataDef, GlobalDataUse, PLBandler
  - GlobalDataDef, GlobalDataUse, PLBandler
  - SharedData:
    - GlobalDataDef, GlobalDataUse, PLBandler
  - SeqFile:
    - ReadNext, WriteNext

Filter:
- StreamIn, StreamOut

Process:
- RoutineDef, RPCDef, RPCCall

Supported Role Types For Connector Types

- Pipe:
  - Source fits to Filter.StreamOut, ReadFile
  - Sink fits to Filter.StreamIn, WriteFile
- FileIO:
  - Reader fits to modules, ReadFile, ReadNext
  - Writer fits to Modules.WriteFile
- ProcedureCall:
  - User fits to (Computation|Module): ProcedureDef
  - User fits to (SharedData|Computation|Module): GlobalDataDef, GlobalDataUse
  - Callr fits to modules, computation, SeqFile, Filter, process, general
  - Caller fits to modules, computation, PLBandler, RPC, RTScheduler

Connection code
- ProcedureCall: ESTABLISH C-proc-call WITH reverse.strlen AS callr libc.strlen AS definer END C-proc-call
  ESTABLISH C-proc-call WITH reverse.strcpy AS callr libc.strcpy AS definer END C-proc-call
  ESTABLISH C-proc-call WITH reverse.malloc AS callr libc.malloc 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.exit AS callr libc.exit 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.push AS callr stack.push 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.stack_init AS caller stack.stack_init AS definer END C-proc-call

- ESTABLISHs bind connectors to ports
  ESTABLISH PIPE with reverse.echo TO modules.echo END PIPE
  ESTABLISH PIPE with reverse.tos TO modules.tos END PIPE
  ESTABLISH PIPE with modules.tos TO reverse.tos END PIPE
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.

```plaintext
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
```

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.

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
    _iwhether IS GlobalDataDef
      SIGNATURE ("struct _iobuf *") END _iwhether
  END INTERFACE
  IMPLEMENTATION IS
    VARIANT libc IN "-lc"
    IMPLTYPE (ObjectLibrary)
  END libc
  END IMPLEMENTATION
END Libc
```

RAPIDE

- 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 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.
  - Generalize ports to calls.
  - Fundamentally more flexible concept for modules!
  - Rapide was marketed by a start-up company.

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 128 128")")
    END caps (remaining definition owithted)
  END IMPLEMENTATION
END KWIC
```

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.

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 the environment.
  - 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

Example ACME Pipe/Filter-Family

- Describe a simple pipe-filter family: This family contains components that are designed for the PipeFilterFam family.
- Extend the basic filter type with a subclass (inheritance).
- Describe component types.
- Describe component variables.
- Describe component connections.
- Describe component implementation.
- Describe component roles.
- Describe component properties.
- Describe component ports.

Instance of an ACME System

ACME Studio as Graphic Environment

London Ambulance System in ACME
**London Ambulance System in ACME**

- 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 explicitly and in sets
  - Several versions available (C++, Java)
- Graphic or textual edits

**Simple Producer/Consumer in Text**

```plaintext
Component Flowcontrol {
    consumer: Consumer;
    producer: Producer;
    send: Sender;
    receive: Receiver;
    timeout: Timer;
}
```

**Architectural Languages in UML**

Hofmeister, Nord, Son: Describing Software Architecture with UML, 1999
Architecture Languages versus UML

- So far, architecture systems and languages were research toys (except Coll)  
- "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.  
  - Student <<person>>  
  - someMethod() <<call>>  
  - SomeMethod() <<call>>  
- 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 frame(s), 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

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:

- RequestDataPacket
- incoming packet(pd)
- outgoing subscribe
- desubscribe
- requestPacket
- packet(pd)

Incoming messages

Outgoing messages

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

Component Diagrams in UML 2.0 Components can be nested

Robot

Arm

Engine

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 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**

Producer

Consumer

Sender

Network

Receiver

Timer

**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**

- Component Model
  - Source or library components
  - Modeling parts
- Composition Technique
  - Adaptation and glue code by connectors
  - Scaling by exchange of connectors

**ADL: Mechanisms for Modularization**

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

- Development environments
- Business services
- Infrastructure
- Distribution
- Contracts
- Binding points
- Parameterization
- UML genericity

**Parameterization**

- Types
- Distribution
- Location transparence

**Binding points**

- Ports
- Versatility

**Contracts**

- Secrets
- Business services

**Development environments**

- Components
- Secrets
- Business services

**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

**Architecture Systems – Composition Technique and Language**

- Connectors
- Infrastructure
- Distribution
- Contracts
- Binding points
- Parameterization
- UML genericity

**Adaptation**

- Product quality
- Software process
- Architecture language
- Adaptation
- Architecture is separated
- Meta-composition
- Only scalable distribution

**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

**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)