Software Architecture Systems

[Separali 27.1-24.1], and references on course home page

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

Additional Literature


Examples of Architecture Systems

- (Darwin) http://www.doe.doc.ic.ac.uk/Software/Darwin/

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

<table>
<thead>
<tr>
<th>Structure</th>
<th>Media plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light plan</td>
<td>Water pipe plan</td>
</tr>
<tr>
<td>Integrated house</td>
<td></td>
</tr>
</tbody>
</table>

An Example of Separation of Concerns: Architectural Aspect in Software

- Software Architecture (Connection of components)

Component Model in Architecture Systems

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

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 connections, 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]

Port 2
Port 1
Port
Component
Application
Component
Application
Component

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 )
- get( data ) (aka: receive( data ))
- synchronous in port, taking in one data
- waitAndGet( data )
- asynchronous in port, taking in one data if it is available

Output ports are synchronous or asynchronous:
- out( data )
- put( data ) (aka: send( data ))
- synchronous out port, putting out one data, waiting until acknowledge
- 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

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 [Gelernter/Carriero'96]
- Service discovery repositories, e.g. Jini, CORBA repositories
- Coddy CCMR
- and more, and combinations of these

Architecture can be Exchanged Independently of Components

"Rewiring"

Reuse of components and architectures is fundamentally improved

```
<table>
<thead>
<tr>
<th>Component</th>
<th>Port 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Port 2</td>
</tr>
<tr>
<td>Component</td>
<td></td>
</tr>
</tbody>
</table>
```

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
Connectors Provide Glueing in Connections

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

**CORBA**
- Client and service provider
- ORB client side, server side
- Marshaling, 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 may
- Events, callbacks, persistence as connectors (can be exchanged to other communications)

**Most Commercial Component Systems**
- 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 [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

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.

A KWIC Index

<table>
<thead>
<tr>
<th>every sentence is replicated</th>
<th>and permuted</th>
</tr>
</thead>
<tbody>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
</tr>
</tbody>
</table>

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 split 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.

The KWIC Problem in Unicon

Components

- The component cap 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 introduced by only changing the type of connectors in a USES declaration.
- Parnas, 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

Modelica [Fritzson 2004]

- Equation-based language for modeling and simulation of systems in physics and engineering
- Component-based language
- Simple example: Resistor component

```
model Resistor "Ideal resistor"
extends Oskar von Hassel parameter Resistance R equation
V = R . . .
end Resistor
```

Modelica (cont.)

- Components are connected by `connect` statements
- Composition by equality of port variables connects equations (realizes e.g. Kirchoff’s Node Law, Newton’s First Law,...)
- Compiler builds a system of differential equations (ODE’s, DAE’s)
- Solving this equation system (numerically) = Simulation of the system.

Example: DC motor model

Modelica (cont.)

- Compound components
  - Code generation from connectors: (Modelica compiler)
  - Connect statements result in equations between port variables
  - Component libraries
  - Commercial implementations (Dynasim AB, MathCore AB) and open source (OpenModelica, PELAB/MathCore)
  - Thesis projects available at PELAB! www.ida.liu.se/~pelab

Modelica (cont.)

- Graphical composition (attaching ports by drag-and-drop) creates connect statements in the model description code

- Back to classical software architecture systems: 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
    - 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
  - 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

[CoSy 1.3]  
www.ace.nl
A CoSy Compiler with Repository-Architecture

- **Parser**
- **Semantics**
- **Transformation**
- **Optimizer**
- **Compiler**

**Subarchitecture**
- **Front end**
- **Middle end**
- **Back end**

**Common Repository**

"Blackboard architecture"

**Composite Engines in CoSy**

- Built from simple engines or from other composite engines by composing 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**

- 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: Analysers, Lowerers, Optimisers, Translators, Support

**Hierarchical Components in the Repository Style (CoSy)**

**Example for CoSy EDL (Engine Description Language)**

- Component class (engine class)
- Component instance (engines)
- Basic components are implemented in C
- Interaction schemes (of skeletons) from complex combinations
- **Generated**
- **Optimiser**
- **Compiler**

EDL can embed automatically viewed and instantiate into a class

**Generated**

- Core routines for compilation:
- **Program**
- **Code**
- **Repository**
- **Access rights**
- **Examples**: Analyzers, Lowerers, Optimizers, Translators, Support

**Generated Factory**

- Created access layer

**Logical view**

- Optimizer
- **Parser**
- **Generated Factory**
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.

Survey of Other Architecture Systems

- For self-studies...
  - UniCon
  - RAPIDE
  - 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, GlobalDataExchange, PLBandler, ReadFile, WriteFile
- Computation:
  - RoutineDef, RoutineCall, GlobalDataExchange, PLBandler
- SharedData:
  - GlobalDataExchange, PLBandler
- 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
  - Sink fits to Filter.StreamIn

- **FileIO**:
  - Reader fits to modules.ReadFile
  - Writer fits to Modules.WhiteFile

- **ProcedureCall**:
  - Define fits to (SharedData|Computation)
  - User fits to SharedData.Computation
  - Modules.GlobalDate

**PLBander**:
- Participant fits to PLBander.
- RoutineDef.
- RoutineCall.
- GlobalDate

**RPC**:
- Define fits to (ProcSet/Sharedprocess).
- User fits to (ProcSet/Sharedprocess).
- RPCDef

**RTScheduler**:
- Load fits to Sharedprocess.RTLoad

---

**ProcedureCall**

CONNECT libc._iob TO datause.definer
CONNECT reverse._iob TO datause.user

/* CONNECTs bind ports to roles */

/* Now for the configuration of connectors to players */

**USES libc INTERFACE Libc**

**USES stack INTERFACE Stack**

/* Component instantiations are declared below. */

**IMPLEMENTATION IS**

**END INTERFACE**

error IS StreamOut SIGNATURE ("line") PORTBINDING (stderr) END error

PLAYER input IS StreamIn SIGNATURE ("line") PORTBINDING (stdin) END input

**TYPE Filter**

**COMPONENT Reverser INTERFACE IS**

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.

**Definition of Connectors**

**Attachment of External Libraries**

**A Filter**

COMPONENT Reverser INTERFACE IS

PLAYER input IS StreamIn SIGNATURE ("line") PORTBINDING (stdin) END input

USES INTERFACE Stack

USES GlobalDate protocol:character-data

* We also use verification analogous to the procedure call connections (next page) *

* Now for the configuration of connections to players *

CONNECT input, user TO datause.use

CONNECT error, user TO datause.errorORTER

END IMPLEMENTATION END Reverser

**A Modules Component**

**INTERFACE IS**

**TYPE modules**

**LIBRARY**

PLAYER image IS RoutineDef

SIGNATURE ("new_type", "void") END image

PLAYER image IS RoutineDef

SIGNATURE ("(struct _iobuf *)", "int") END _iwhether END INTERFACE

**END IMPLEMENTATION**

IMPLEMENTATION IS

**END INTERFACE**
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

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 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
  
  ```
  Template SystemG:: Connector {
    Connector {
    Roles {
      source = SystemRoles()
      sink = SystemRoles()
    }
    properties {
      blocking = nonblocking
      Aesop-style = subcontract-ball
    }
  }
  ```
  
  Features
Instance of an ACME System

```plaintext
// The component smooth has a visualization added
// Describe an instance of a system using the PipeFilterFam family.

shape : ShapeT; color : ColorT; ];

property

property

// Declare non-family property types that will be used by this system instance.
// defined elsewhere.
// representation refers by name to a system that is
detectErrors has a visualization added, as well as a
};

stdout to showTracksSubsystem.stdout;
}

Roles { clientEnd; serverEnd; }
Connector mapRequest2 : RPC_T = {

Roles { clientEnd; serverEnd; }
Connector incidentInfoRequest : RPC_T = {

sendIncident; receiveCallMsg; }
Ports { mapRequest; incidentInfoRequests;
Component callntry = { Port sendCallMsg; }

System LAS = {
// declare system components (none of which are typed)
// Instance based example - simple LAS architecture:
Connector Type RPC_T = { Roles { clientEnd; serverEnd; } }

property msgFlow : FlowDirectionT;
Roles { fromRole; toRole; }
Connector Type MessagePassChannelT = {

property

// That comes with the UnixFilterT type.
// Associate a value with the implementationFile property
// Desired properties that can be used by systems
// Property types defined for the PipeFilterFam family
// property Type Task = ...
```

London Ambulance System in ACME

```plaintext
Component dispatch is... (cont'd from previous page)
Attachment filter = ...
Attachments
tables is an array...
```
Darwin (Imperial College)

- Components
  - Primitives 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

```java
Component.ProducerConsumer()
    consumer, Consumer;
    producer.Producer;
    send(Sender);
    rec(Receiver);
    out();
    timer(Timer);
    bind
        producer.out --> send.user;
        timer.tick --> send.tick;
        rec.out --> rec.receiver;
        send.commout = rec.in;
        rec.in --> send.receiver;
```

Architectural Languages in UML

- **Hofmeister Model of Architecture**
  - (Hofmeister/Nord/Sand'99) is the first article that has propagated the idea of specifying an architecture language with UML.
  - Conceptual view: Functionality + interaction (components, ports, connections)
  - 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

- **Architecture Languages versus UML**
  - So far, architecture systems and languages were research toys (except COFF)
  - “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

- **Simple Producer/Consumer**
  - Producer
  - Consumer
  - Communication
  - Timing
  - Control
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.

  ```plaintext
  <<person>>                      <<person>>
  Student                             Student
  someMethod()                     someMethod
  ```

- 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.

  ```plaintext
  A mechanism for extending/customizing UML without changing it.  
  ```

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

Modeling software architecture in UML

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:
  - 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 Pedaits port:

- For modeling protocols, use UML Sequence diagram or State diagram
- Provided and required interfaces
- Substitutable
- Run-time representation of one or several classes
- Source or binary code

Components in UML 2.0

- Components in 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
Components in UML 2.0

- 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

Services

- Ports can be grouped to Services

Connectors in UML 2.0

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

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 binary components
- Binding points, ports

Composition Technique
- Adaptation and glue code by connectors
- Scaling by exchange of connectors

Architectural language

Composition language

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
  - Parameterization: depends on language
  - Standardization: still pending

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
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)