

# Components and Architecture Description Languages

*Jean-Marie FAVRE*

*Equipe ADELE*  
Laboratoire LSR-IMAG  
Université Grenoble I, FRANCE

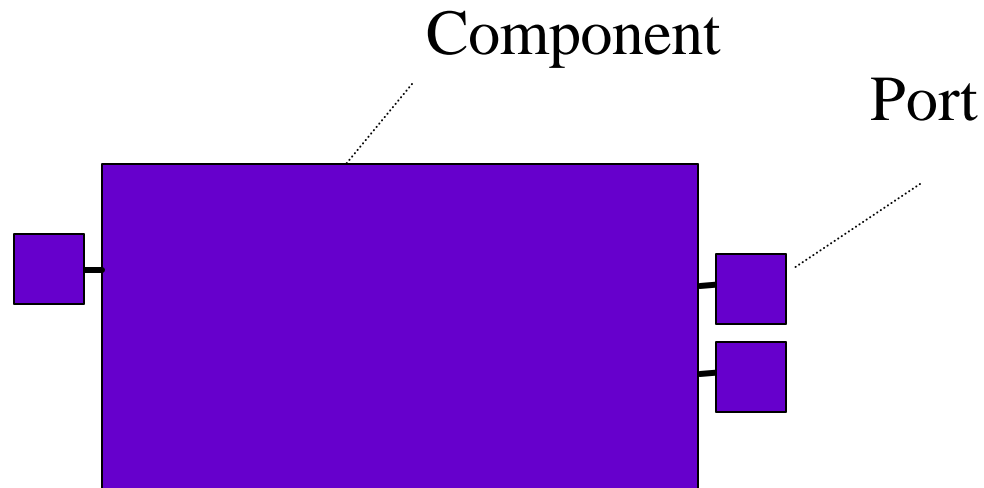
# Objectives

- ◆ Architectural description language (ADL)
- ◆ « *An important problem for component-based systems engineering is finding appropriate notations to describe those systems. Good notations make it possible to document component-based design clearly, reason about their properties, and automate their analysis and system generation.*
- ◆ Focuses on design rather than implementation

# Main features

- Entities with an arbitrary set of *properties* and *constraints*
- Everything is typed : entities, connectors, constraints ...
- Hierarchical description of architecture
- Description of families of architecture, styles, ...
- Focuses on structural aspects
- No link with code

# Components and Ports

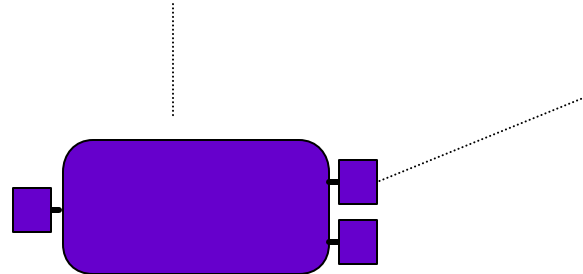


**Interface of a component**

# Connectors and Roles

Connector

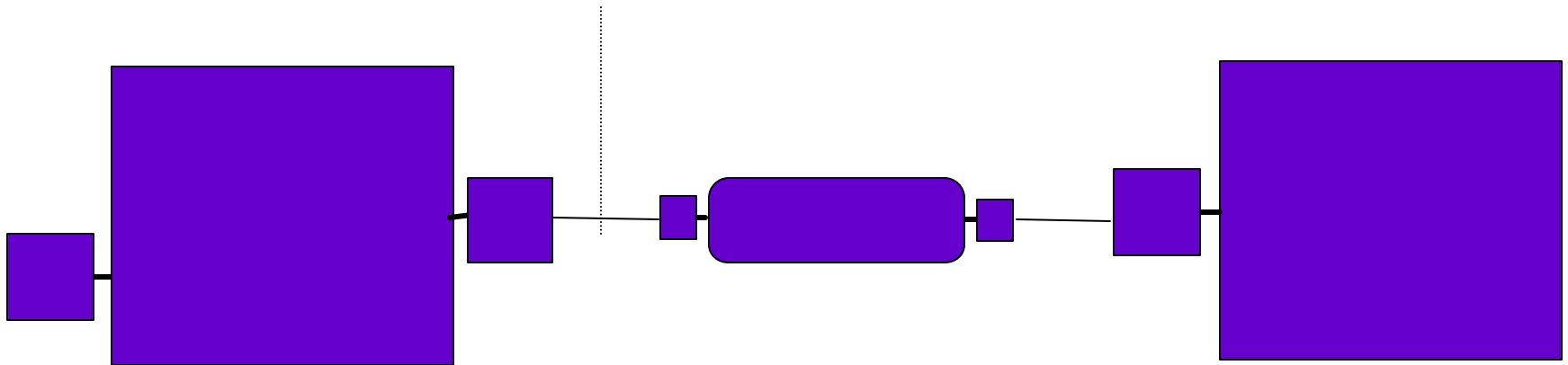
Role



**Interface of a connector**

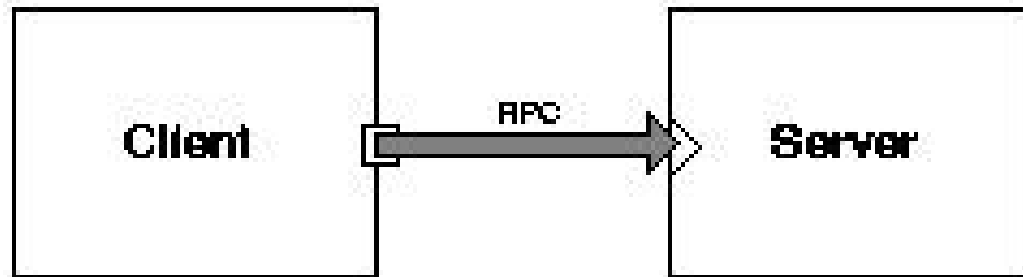
# Systems and attachements

Attachement



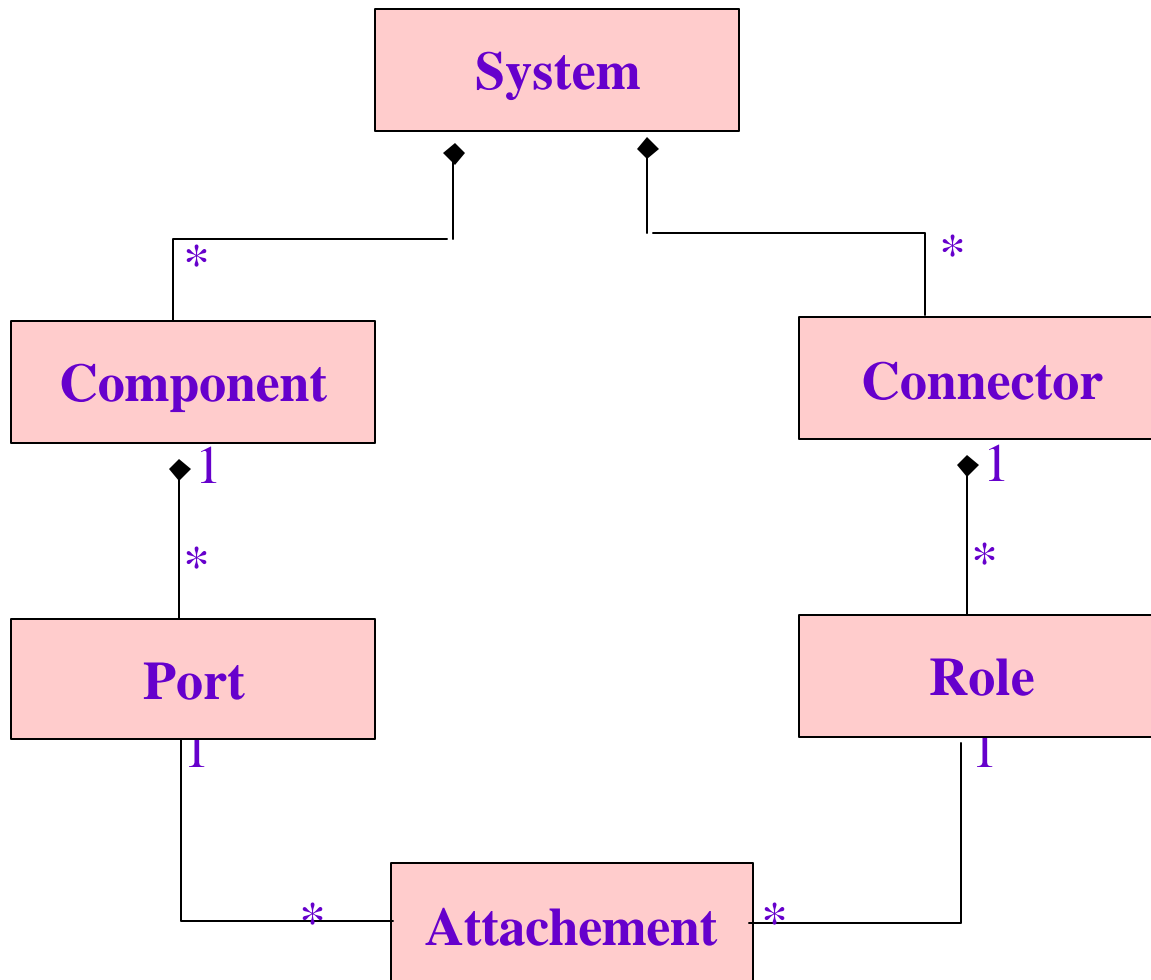
**A simple system**

# Example of a System

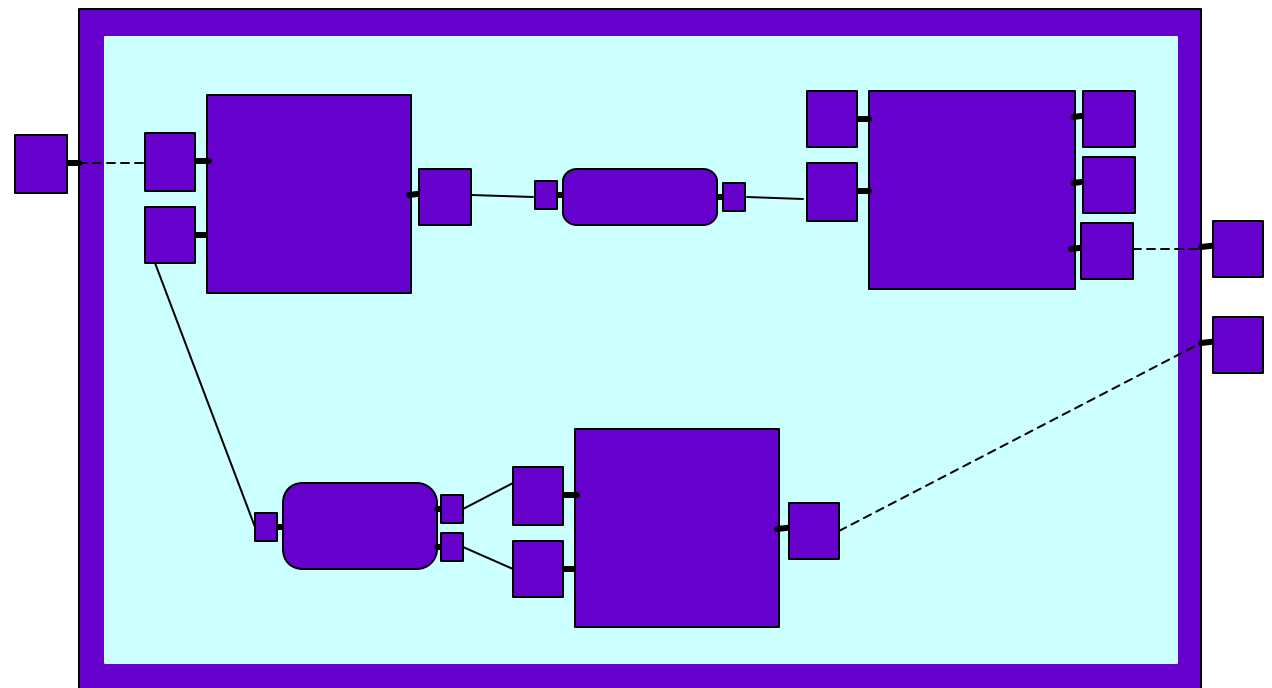
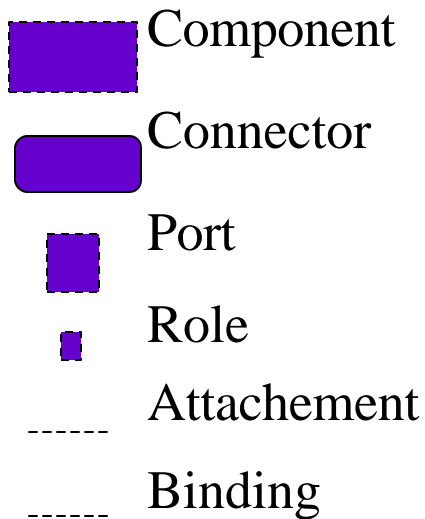


```
System simple_cs = {  
  Component client = { Port send-request; };  
  Component server = { Port receive-request; };  
  Connector rpc = { Roles { caller, callee} };  
  Attachments {  
    client.send-request to rpc.caller;  
    server.receive-request to rpc.callee;  
  }  
}
```

# Meta model

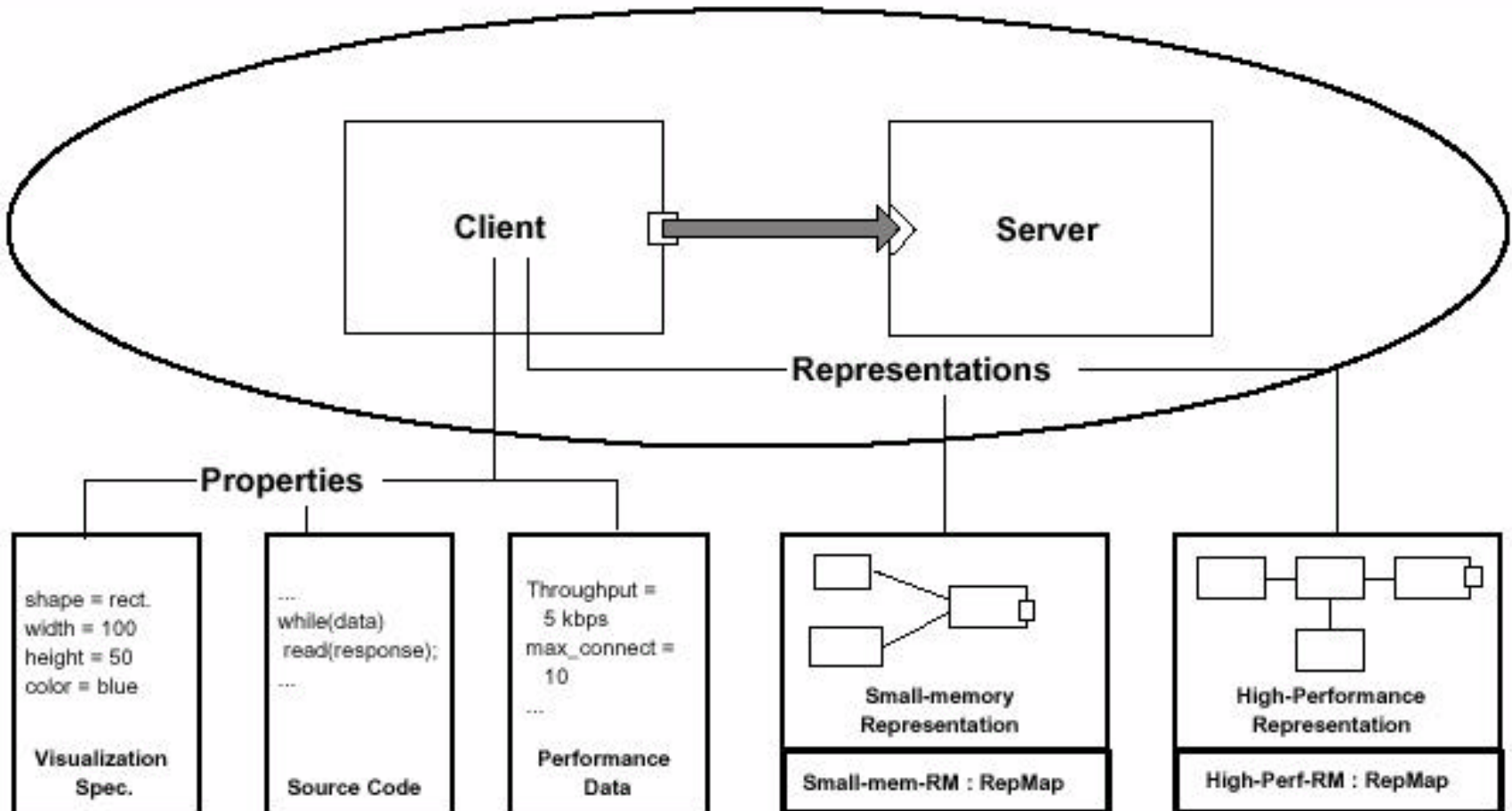


# Representations and Bindings



Representation of a component

# Representations and Properties





```
System simple_cs = {
  Component client = {
    Port send-request;
    Properties { Aesop-style : style-id = client-server;
                 UniCon-style : style-id = cs;
                 source-code : external = "CODE-LIB/client.c" }}
  Component server = {
    Port receive-request;
    Properties { idempotence : boolean = true;
                 max-concurrent-clients : integer = 1;
                 source-code : external = "CODE-LIB/server.c" }}
  Connector rpc = {
    Roles {caller, callee}
    Properties { synchronous : boolean = true;
                 max-roles : integer = 2
                 protocol : Wright = "..." }}
  Attachments {
    client.send-request to rpc.caller ;
    server.receive-request to rpc.callee }
```

# Constraints

$\text{exists client, server, rpc |}$   
 $\text{component(client) } \wedge \text{ component(server) } \wedge \text{ connector(rpc) } \wedge$   
 $\text{attached(client.send-request, rpc.caller) } \wedge$   
 $\text{attached(server.receive-request, rpc.callee) } \wedge$   
 $\text{client } \neq \text{ server } \wedge \text{ server } \neq \text{ rpc } \wedge \text{ client } \neq \text{ rpc } \wedge$   
 $(\text{for all } y:\text{component } (y) \Rightarrow y = \text{client} \mid y = \text{server}) \wedge$   
 $(\text{for all } y:\text{connector}(y) \Rightarrow y = \text{rpc}) \wedge$   
 $(\text{for all } p, q: \text{attached}(p, q) \Rightarrow$   
 $\quad (p = \text{client.send-request } \wedge q = \text{rpc.caller})$   
 $\quad \mid (p = \text{server.receive-request } \wedge q = \text{rpc.callee}))$

# Families

```
Family PipeFilterFam = {
  Component Type FilterT = {
    // All filters define at least two ports
    Ports { stdin; stdout; };
    Property throughput : int; };
  Component Type UnixFilterT extends FilterT with {
    Port stderr;
    Property implementationFile : String; };
  Connector Type PipeT = {
    Roles { source; sink; };
  Connector Type PipeT = {
    Roles { source; sink; };
  Property Type StringMsgFormatT =
    Record [ size:int; msg:String; ];
  Property Type TasksT =
    enum {sort, transform, split, merge};
};
```

# ACME

- Interface
  - ◆ Type of components, Arbitrary types of ports
- Composition
  - ◆ Connectors, Assembly, ...
- Implementation
  - ◆ Refinement but no link to the implementation
- Life cycle.
  - ◆ Design
- Infrastructure
  - ◆ No infrastructure