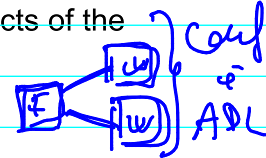
Common concepts among various components models

- Components
  - runtime entities with contractually specified interfaces
  - written in programming languages such C, C++, Java, ...
- Interfaces (aka ports)
  - client (required) / server (provided) interfaces
  - usually defined using an Interface Definition Language (IDL)
- Bindings (aka connectors)
  - connectors among interfaces of the components
  - various types: synchronous/asynchronous, local/remote, ...
- Architecture Description Language (ADL)
  - to define relationships among components and their properties

CORBA
RPC
JAVA = Interface
1 COMP.
ASM

Concepts depending on the component model and implementation language #1/2

- ADL
  - dedicated language (ie. XML based) or using constructs of the programming language (ie. Java 5 Annotations)
- Hierachical components
  - only at design time (in the ADL)
  - also at runtime time (**primitive** and **composite** components)
- Introspection and intercessions capabilities
  - possibility to discover components interfaces and relationships among them
  - possibility to reconfigure the architecture at runtime
  - fixed (by the model) / open reflection capabilities
  - in general, how much they can be controlled, introspected, instatiated, destroyed

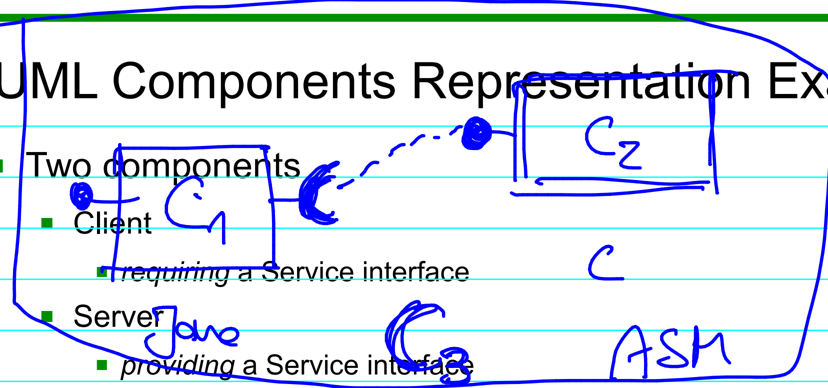


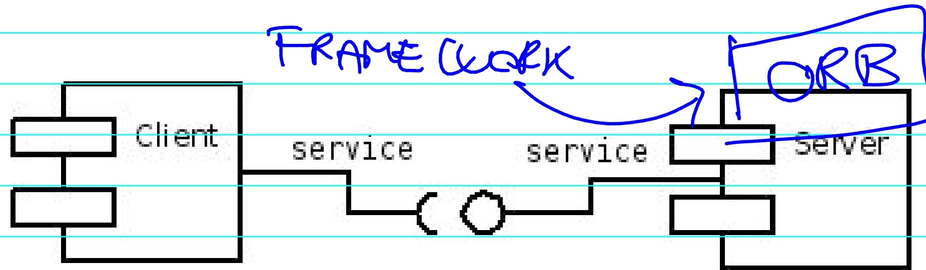
Concepts depending on the component model and implementation language #2/2

- Multi-language support
  - implementations for different programming languages (ie. C++, Java)
  - interoperability among components written in different languages
    - ie. through an ORB
- Programming language invasiveness
  - mandatory interfaces to implement / classes to extend
- Interface Definition Language (IDL)
  - dedicate language (ie. Corba IDL) or using constructs of the language used to write components code (ie. Java Interfaces)
- Container
  - do components need to be deployed on a container?

UML Components Representation Example

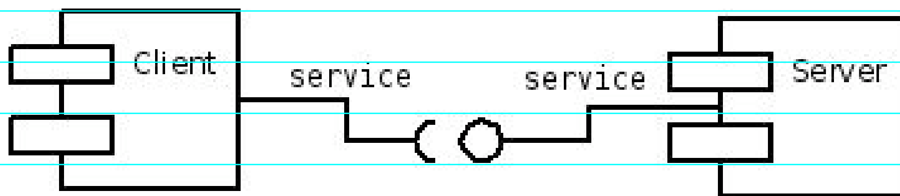
- Two components
  - Client
    - *requiring* a Service interface
  - Server
    - *providing* a Service interface





UML Components Representation Example

- Two components
  - Client
    - *requiring* a Service interface
  - Server
    - *providing* a Service interface



Component models used nowadays

- EJB
  - Java EE standard component model
- Spring Framework
  - Java implementation very widespread
  - .NET counterpart exists also
- Corba Component Model (CCM)
  - niche market
  - mainly C++ and Java implementations
- Microsoft Component Model (COM)
  - Microsoft platforms
- ...

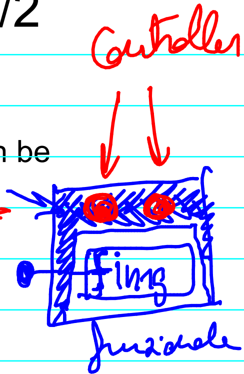
The Fractal Component Model

Reasons for the Fractal Component Model

- Limitations in other component models and ADLs:
  - limited support for extension and adaptation
  - fixed forms of composition
  - fixed forms of introspection & intercession
- « Develop a powerful (**reflective**) but **flexible** / **extensible** / **customisable language independent** component model for any kind of software (from middlewares to operative systems) throughout the complete software lifecycle, with an enphasis on **runtime reconfiguration** and **management** which is in general the least well handled parts in existing component models. »

The Fractal Component Model #1/2

- **Open**
  - extra-functional services associated to a component can be customized through the notion of a **control membrane**
- **Recursive**
  - components can be nested in **composite components**
  - uniform view at any level of the system architecture
- **Execution Model Independant**
  - **no execution model is imposed**. Components can be run within other execution models than the classical thread-based model such as event-based models and so on
- **Language agnostic**
  - implementations for **various programming languages** (Java, C, ...)



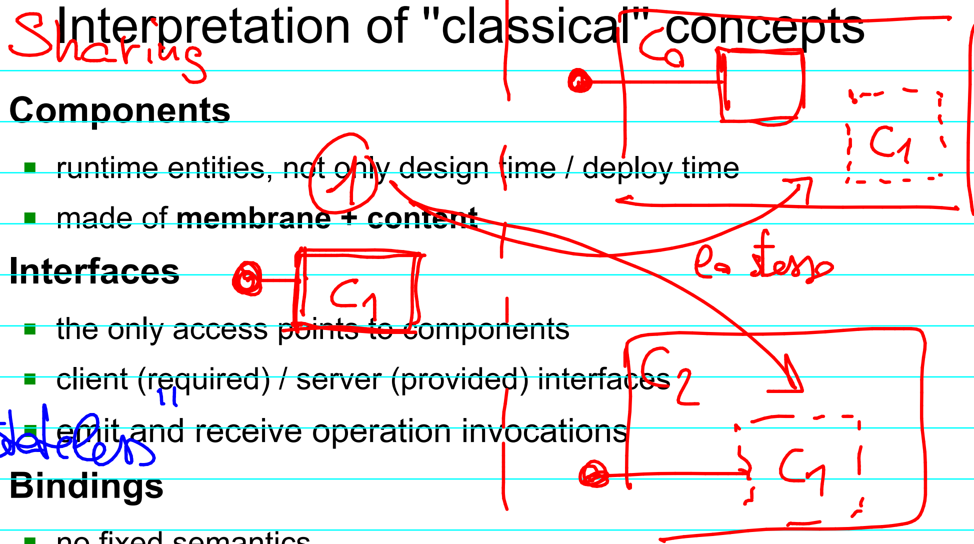
The Fractal Component Model #2/2

- **Component Sharing**
  - a given **component instance can be included (or shared) by more than one component.** This is useful to model shared resources such as memory manager or device drivers for instance
- **Binding Components**
  - a single abstraction for components connections that is called bindings. **Bindings can embed any communication semantics** from synchronous method calls to remote procedure calls
- **Selective reflection**
  - components *can* have **full introspection and intercession capabilities**
  - different components in the same architecture may have **different level of introspection and intercession**

⊖ introsp. ⇔ legacy components

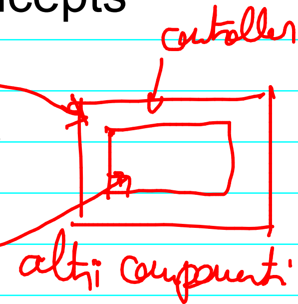
Interpretation of "classical" concepts

- **Components**
  - runtime entities, not only design time / deploy time
  - made of **membrane + content**
- **Interfaces**
  - the only access points to components
  - client (required) / server (provided) interfaces
  - emit and receive operation invocations
- **Bindings**
  - no fixed semantics
  - **primitive** bindings: in the same address space (ie. an object reference)
  - **composite** bindings: for distributed (ie. RMI) or heterogeneus (ie. JNI) communication



FRACTAL:

Interpretation of "classical" concepts

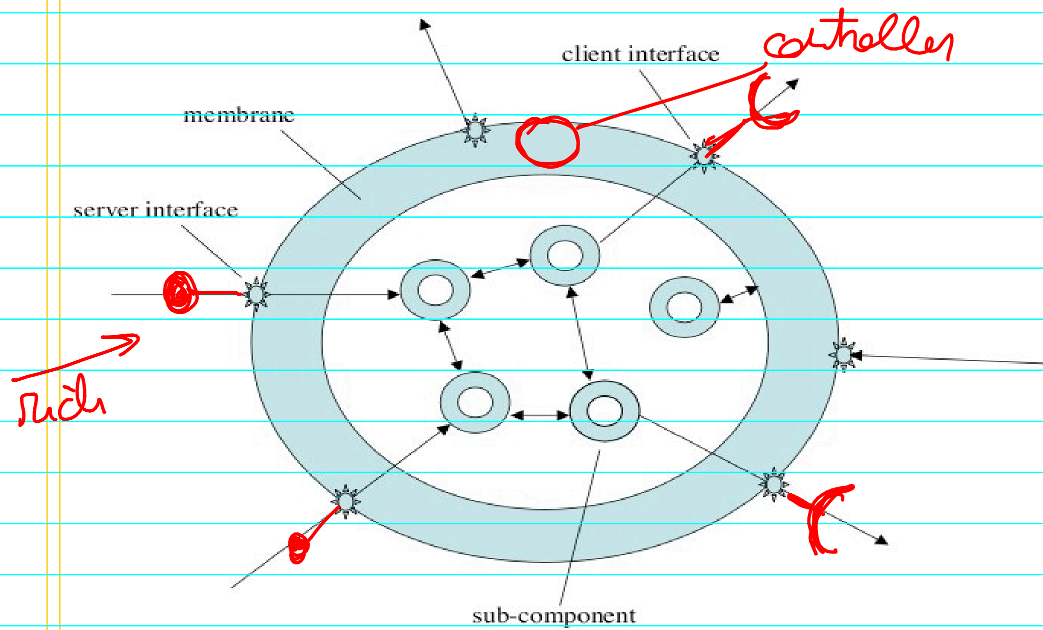


- **Components**
  - runtime entities, not only design time / deploy time
  - made of **membrane + content**
- **Interfaces**
  - the only access points to components
  - client (required) / server (provided) interfaces
  - emit and receive operation invocations
- **Bindings**
  - no fixed semantics
  - **primitive** bindings: in the same address space (ie. an object reference)
  - **composite** bindings: for distributed (ie. RMI) or heterogeneus (ie. JNI) communication

The membrane

- **Composition** and **reflection** behaviour
- Can provide access to reflection capabilities via **controller interfaces**
- **No fixed set of controllers** for component introspection and intercession
- *Can* have an internal structure of its own
- **No fixed semantics**
- Can have interceptors
- Components in the same architecture can have different membrane structure

Components: membrane + content

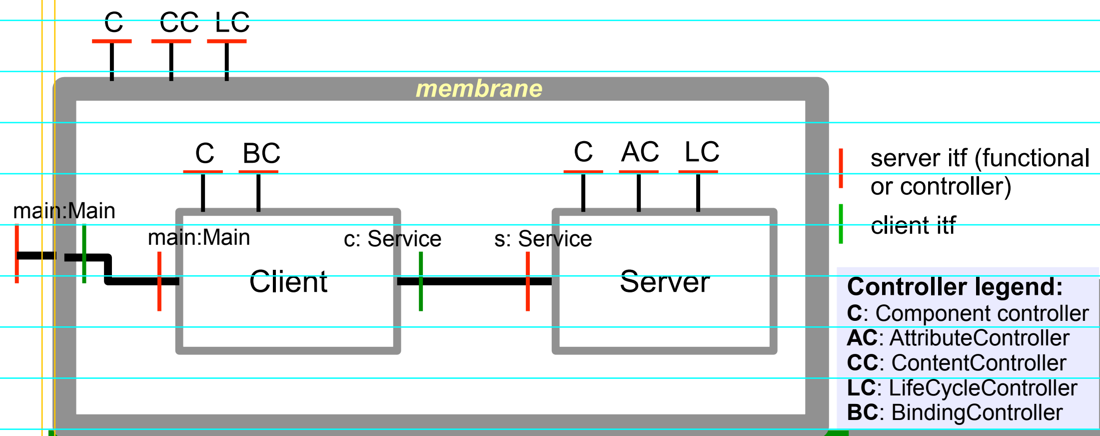


Standard Controllers

- Reflection : minimal
  - **Component controller** (discovering component interfaces)
  - **Binding controller** (binding an external component interface)
- Reflection : structural
  - **Content controller** (adding, removing subcomponents)
  - **Attribute controller** (setting, getting component attributes)
- Reflection : behavioral
  - **Lifecycle controller** (starting, stopping the component)

A Fractal example: HelloWorld

- HelloWorld composite component with
  - *external* interfaces: exposed to the exterior
  - *internal* interfaces: to 'export' sub components' interfaces
- Client and Server primitive components



Programming with Fractal

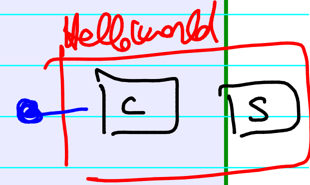
- Extensible Architecture Description Language (**ADL**)
- Extensible and retargettable **ADL toolchain**
- Host **programming environments**
  - Java: Julia, AOKell
  - C: Think, Cecilia
  - C++: Plasm
  - SmallTalk: FracTalk
  - .NET: FracNet

Fractal ADL

- The Fractal XML-based extensible Architecture Definition Language
  - different modules to cover different aspects: components definition, their interfaces, bindings among them, attributes (properties), component containment, component content, component remote deployment, component definition extension from another definition, ...
  - new modules can be added to cover other aspects
    - ie. BindingFactory module to allow bindings (client/server interfaces) over arbitrary communication protocols
- The language grammar is defined by means of an XML DTD (Document Type Definition)

A Fractal ADL example: HelloWorld

```
<definition name='HelloWorld'>

  <interface name='main' role='server'  signature='boot.api.Main' />

  <component name='client'>
    <interface name='m' role='server' signature='boot.api.Main' />
    <interface name='c' role='client' signature='hw.Service' />
    <content class='hw.client' />
  </component>

  <component name='server'>
    <interface name='s' role='server' signature='hw.Service' />
    <content class='hw.server' />
  </component>

  <binding client='this.main' server='client.m' />
  <binding client='client.c' server='server.s' />

</definition>
```



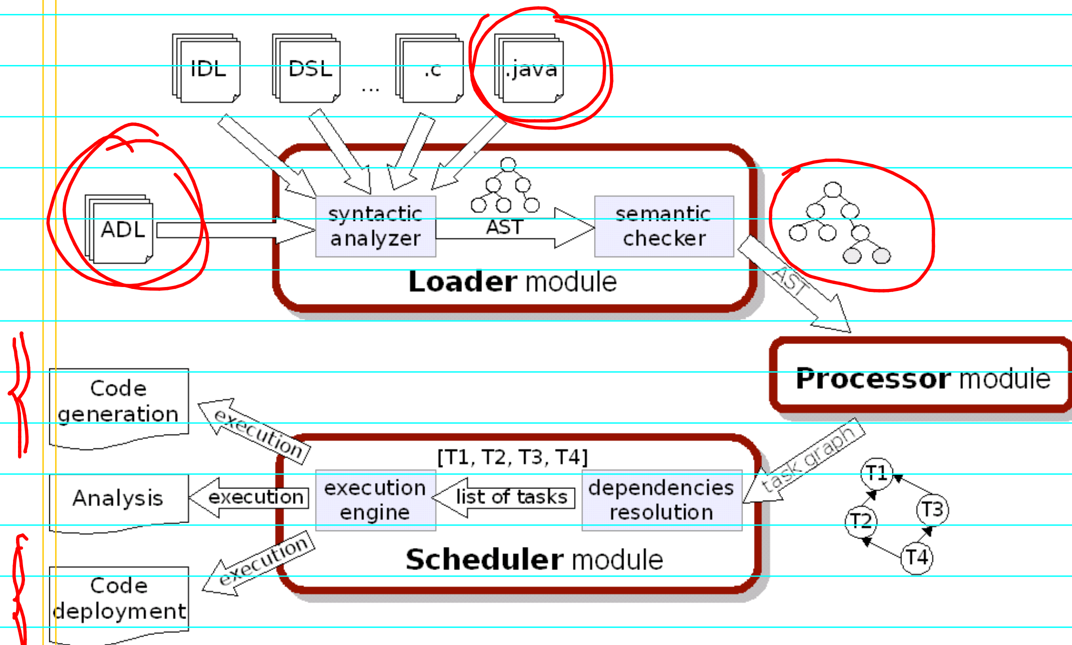
Composition and binding rules

- Top level component in an ADL file as a <**definition>** element
- Sub components as **<component>** sub elements
- Top level or sub components declare their **<interface>**s
  - name, role ("client" | "server"), signature
- Primitive components declare their implementation artifact
  - **<content class="path.to.implFile" />**
  - type of the artifact depends on the Fractal implemetation (Java class, C file, ...)
- A composite component declare **<binding>**s for its direct sub components and its internal interfaces

The Fractal ADL toolchain

- It is itself a Fractal application
- Extensible and very modular
  - mantains a uniform representation (AST) for possibly heterogeneus languages to process:
    - ADLs, IDLs, DSLs, ...
  - may cover different architectural concerns:
    - analysys, code generation, code compilation, deployment, ...
  - handling components written in different languages:
    - Java, C
  - plugin based

Fractal ADL toolchain architecture



Samples of the Fractal versatility

- **Operating systems**: written with Think or Cecilia
- **Middlewares**: Dream
- **Transaction management**: GOTM, Jironde
- **Persistency Services**: Speedo, Perseus, JORM
- **Computational Grids**: Proactive
- **Middleware for enterprise application integration** (EAI): Petals
- **Auto-adaptive EJB servers**: ReflectAll
- **Distributed systems management**: Jade, Jasmine
- ...

Fractal: conclusions

- From objects to reflective components to build manageable systems
  - Interfaces
  - Explicit connections
  - Membranes (reflective components)
- Computational model for open systems
  - open binding semantics
  - open reflection semantics
- Extensible ADL & associated toolchain
- More info on the website
  - http://fractal.objectweb.org/
  - http://fractal.objectweb.org/fractal-distribution (*experimental*)

Cap. 25 del manuale di ProActive
(4.0.2)

~~Cecilia~~