INRIA RHÔNE-ALPES

france telecom

Fractal: A Component Model for Manageable Distributed Systems

Tutorial at ECOOP 2006
Nantes, France, July 4th 2006

T. Coupaye - France Telecom
J.-B. Stefani - INRIA

ObjectWeb
Open Source Middleware

Overview

The Fractal Project

What is the Fractal Project?

- An open source development project
  - Part of the ObjectWeb Consortium code base
  - ObjectWeb is an international consortium that develops open source middleware (http://www.objectweb.org)
- That develops a reflective software component technology for the construction of highly adaptable, and reconfigurable distributed systems
  - A programming language independent component model
  - A set of tools to support programming and software construction using the Fractal component model

Motivations

Components for global computing: a fact of life

- plug-ins, xBeans, packages, COM & .Net, SCA, CCA, etc.

Components: at the crossroad of multiple concerns

- Software architecture
- Software evolution
- Distribution
- Mobility
- Deployment
- Configuration
- Dependability

Motivations

- **Dealing with the complexities of distributed systems construction and management**
  - Physical and logical separations
  - Partial faults and distributed attacks
  - Patches, updates and evolutions
  - Varying loads and distributed resources
  - Multiple overlapping concerns
    - Deployment and configuration
    - Availability and dependability
    - Performance and optimization
    - Protection and security
    - Service level agreements

Motivations

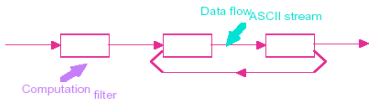
- **Dealing with the complexities of distributed system construction and management requires a component-based approach**
  - components as units for system modularity, reconfiguration, fault isolation
  - explicit software architecture as a basis for system instrumentaiton, deployment, configuration management
  - explicit software architecture for software productivity
    - fighting architectural erosion, facilitating software maintenance and evolution, supporting product families, dealing with system construction tradeoffs, etc

Component-based software engineering (CBSE)

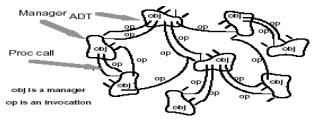
- **A sub-discipline of software engineering dealing with the modular construction of software systems through the explicit composition of software units (components), and the elicitation of software systems' architectures**
  - Semantical foundations
    - Component & composition operators
  - pattern catalogs
    - architectural styles, design patterns
  - design & programming tools
    - specification, analysis, architecture description, and programming

CBSE: Example architectural styles

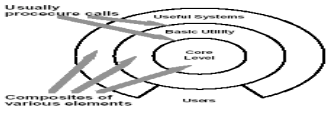
- **Pipes and filters**



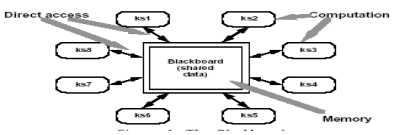
- **Event-based structures**



- **Object-based structures**
- **Layers**

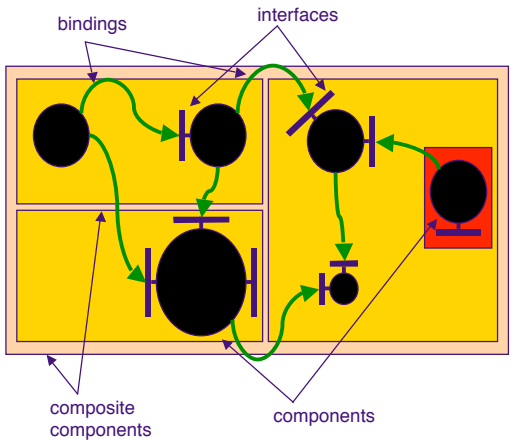


- **Blackboards**



CBSE: Informal foundations

- **Component**
  - piece of data & behavior
  - has well identified access points (interfaces or ports)
    - provides abstraction and information hiding
  - is encapsulated
- **Connector (or Binding)**
  - communication path between components
  - has well identified endpoints (interfaces or ports)
  - encapsulates communication behavior



CBSE: Using software architecture

- **Components as units of modular design**
  - interface/implementation separation
  - decoupled interactions
- **Components as units for management operations**
  - configuration & assemblage
  - changes & evolution
  - deployment & installation
  - monitoring
  - fault isolation
  - replication
  - resource allocation

CBSE: Approaches

- **« Standard » component models**
  - Java Beans, EJBeans, Mbeans, Microsoft COM & .Net, OSGI bundles, UML 2.0 Components, Service Component Architecture (SCA), Common Component Architecture (CCA), etc.
- **Component models and ADLs**
  - Wright, Acme, Rapide, Unicon, C2, Darwin, Room, xArch, ComUnity, OpenCOM, Olan, etc.
- **Programming languages**
  - ArchJava, Jiazzi, ComponentJ, Piccola, Scala, etc

CBSE & Fractal

- **Fractal**
  - a component model
    - programming language independent
      - many different implementations
    - reflective
      - components can provide introspection capabilities
    - open
      - no predefined semantics for component connection, composition and reflection
  - an extensible architecture description language (ADL)
    - core ADL for basic concepts
    - additional ADL modules for different architectural concerns
  - a set of programming and supporting tools

CBSE & Fractal

- **Fractal's original aspects**
  - reflective components
  - open model
  - hierarchical & sharing structures
  - software architecture at run-time
  - lightweight programming support
  - subsumes most existing component models
  - can be understood as a component meta-model
    - with different specializations / personalities possible

Overview

Fractal: Principles

- **Hierarchical components**
  - Components as run-time entities (computational units)
  - Self-similarity : handling systems and subsystems in the same way
- **With sharing**
  - Software architectures with resources
- **Native support for distribution**
  - Naming and binding framework
  - Component connections can have arbitrary semantics and span networks
- **And selective reflection**
  - Components can choose to export arbitrary details of their implementation
  - No pre-defined meta-object protocol for component introspection and intercession

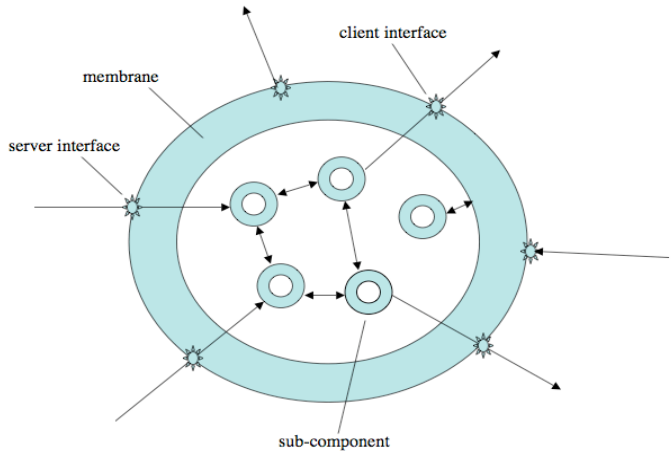
Fractal: classical concepts

- *Components* **are encapsulated data & behavior**
  - runtime entities, not only design time or load time
  - units of encapsulation and behavioral integrity
- *Interfaces* **are the access points to components**
  - aka **ports**
  - interfaces can emit (client) and receive (server) operation invocations
- *Bindings* **mediate interactions between components**
  - aka **connectors**
  - can be primitive (in the same adress space) or composite
  - composite bindings are components + primitive bindings
  - Bindings can span address spaces and networks
  - No fixed semantics for bindings

Fractal: original concepts

- **Component = membrane + content**
  - Content = set of (sub-)components
  - Membrane = composition and reflection behavior
- **Membrane**
  - Can have an internal structure of its own
  - Can provide access to reflection capabilities via controller interfaces
  - No fixed semantics for membranes
  - No fixed set of controllers for component introspection and intercession
- **Sharing**
  - Components can be shared by multiple composites
  - Sharing can model resource sharing and cross-cutting concerns

A Fractal Component



Fractal: Semantics

One can give a formal semantics to the Fractal model using co-algebras

- The formal semantics is very abstract, and respects the generality of the Fractal model
- The original features of the Fractal model are represented simply in a set-theoretic setting

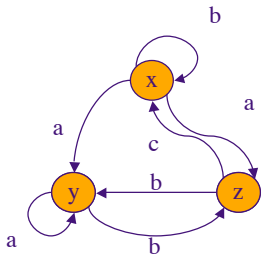
In the next slides:

- A quick informal introduction to the notion of co-algebra
- The basic intuition behind the formalization of Fractal component
- A formal definition of a component in the Fractal model

Fractal: Semantics

Example co-algebras

- A co-algebraic view of streams : X -> A x X
  - x = < a, y > y = < b, x>
  - x = abab... y = baba...
- A co-algebraic view of automata : X -> P(A x X)
  - x = {<a, y>, <a, z>, <b, x>}
  - y = {<a,y>, <b,z>}
  - z = {<c,x>, <b,y>}



Coalgebras

Coalgebra

- An operator G on (hyper)sets is monotone if for all a,b:

  $a \subset b \Rightarrow G(a) \subset G(b)$

- A G-coalgebra is a pair <X,**e**> where X is a set, and **e** is a function

  e : X -> G(X)

- Intuitively, a co-algebra is a system of equations whose form is given by G

Final coalgebra theorem

Let G be a monotone operator on (hyper) sets. Then:

- G has a greatest fixed point G*,
- and every G-coalgebra has a unique solution in G*

Fractal : Semantics

A co-algebraic definition of kells

- characterize kells in a syntax-free manner
- use hypersets to get final models with a straightforward interpretation
- hypersets = non-well-founded sets (cf. Aczel, Barwise & Moss)
  - a system of equations is a tuple (X,A,e), where X and A are 2 disjoint sets and e : X ->P(X U A)
  - AFA (Anti-Foundation Axiom): every system of equations (X,A,e) has a unique solution s

Fractal: Semantics

➡ **Key idea : identify a component with the (hyper) set of its transitions**

<content, inputs, outputs, outcome>

- `content` : a (multi) set of `components`
- `inputs`, `outputs` : (multi) sets of `signals`
- `signal` : a record of `values`
- `value` : a base `value`, a `name` or a `component`
- `outcome` : a (multi) set of `components`
  - accounts for component factories : a component may create other components

Fractal:Semantics

➡ **Operator G (on hypersets)**

- G(X) = **P**($\mathbf{M}_f$(X) x $\mathbf{M}_f$(S) x $\mathbf{M}_f$(S) x $\mathbf{M}_f$(X))
- S = $U_{k \in N}$ (L x D)^k
- D = L + V + X (names + values + kells)
- **P:** powerset $\mathbf{M}_f$**:** finite multisets

➡ **A component c is the unique solution of a pointed G-coalgebra, <X,** e**, c>**

- <X, **e**> is a G-coalgebra
- c is an element of X
- e is a set of (hyperset) equations: **e**: X -> G(X)

Fractal: Forms of components

- **Components without reflection : objects**
- **Components with minimal introspection (Component and Interface controllers) and simple aggregation : COM components**
- **Components with binding controller and lifecycle controller : OSGI bundles**
- **Components with 2-level hierarchies and binding controller : SCA components**
- **Components with binding controller and multicast bindings : CCA components**
- **Components with attribute controller : MBeans**
- **Composites with transaction and persistency interceptors, controlling subcomponents with lifecycle controller : EJB2 containers and EJB2 Beans**

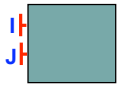
Organisation of the model specification

- **Elements of the model specification**
  - « Levels of control »
    - Foundations
      - Base components with no reflexive capabilities (legacy code)
      - IDL : Fractal is not only for Java
      - Naming and binding API (Name, NamingContext, Binder)
    - Introspection
      - Component and Interface API
      - Introspection of components boundaries
    - Configuration
      - (structural introspection and intercession) Attribute, Content, Binding, Lifecycle control API
      - (Predefined but more generally arbitrary) reflexive control of (white-box) components structure
  - Basic Typing : role, contingency (optional, mandatory), cardinality (singleton, collection)
  - Instantiation
    - Generic factories : create components of a type given as input
    - Standard factories : “ad-hoc” factories that create components of one type each
    - Templates : “facilities” that create components isomorphic to themselves
- **Everything is optional and extensible (« open model »)**
  - Introspection, control interfaces and controllers, factory interfaces, typing
- **Which leads to *conformance levels***

Foundations API

Base components

- Do not provide any control (introspection, configuration) interfaces
- Similar to ODP or plain Java objects : interfaces expressed in Fractal IDL



Fractal Interface Definition Language (IDL)

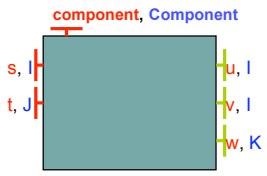
- Interfaces are expressed as Java interfaces with
  - Restrictions : No modifiers, literal fields, no inner interfaces and classes, class types not allowed
  - Extensions : `any` and `string` (`Object` and `String` class types are not allowed)
- Programming languages mapping
  - Java mapping
    - Addition of the `public` modifier
    - `any` maps to `java.lang.Object`
    - `string` maps to `java.lang.String`
  - C mapping
    - used e.g. in Think but not yet expressed in the Fractal specification

Naming and binding framework

- Name management to (remotely) access component interfaces
- Defines API for *names*, *naming contexts* and *binders*

Introspection API

- **This level provides external introspection capabilities**
- **A component at this level owns the `Component` Fractal control interface for interfaces discovery**
  - Component allows discovery of all interfaces owned by a component (server and client, external and internal)
- **Interfaces are named**
  - The name of an interface is valid in the context of the component that owns this interface
- **Interfaces are typed**
  - An interface implements both
    - its functionnal interface signature (expressed in Fractal IDL: e.g. I, J, K)
    - and the Fractal `Interface` interface
- **One can access any interface of a component since it has a reference on any other interface thanks to the getFcItfOwner operation (similar to COM IUnknown)**

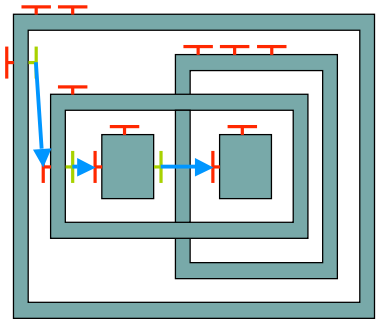



```
public interface Component {
   Type getFcType ();
   Object[] getFcInterfaces ();
   Object getFcInterface (String itfName);
}
```

```
public interface Interface extends Name {
   Component getFcItfOwner ();
   String getFcItfName ();
   Type getFcItfType ();
   boolean isFcInternalItf ();
}
public interface Type {
   boolean isFcSubTypeOf (Type type);
}
```

Configuration API (1/5)

- **This level provides more introspection and intercession capabilities**
- **It allows for exposition and control of components internal structure**
- **It defines 5 « standard controllers » …**
  - `BindingController`
  - `ContentController, SuperController`
  - `AttributeController`
  - `LifeCycleController`

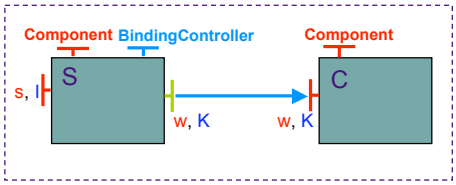


…used for initial configuration and dynamic reconfiguration

- **NB**
  - Bindings and content controls are really central to *architectural configuration*
  - Attributes control is concerned with a restrictive, classical sense of configuration : parametrization
  - Stricly speaking, life cycle control is not concerned with configuration - but it often needed for configuration

Configuration API (2/5)

- **BindingController**
  - Used to manage *local bindings* between components
    - Complex bindings (e.g. remote)
      - must be created using the naming and binding framework
      - may involve binding components
    - Strong semantics of *locality*
      - Binded interfaces must be owned by components in a same direct enclosing component




```
interface BindingController {
  void bindFc (string c, any s);
  void unbindFc (string c);
  string[] listFc();
  any lookupFc (string c);
}
```

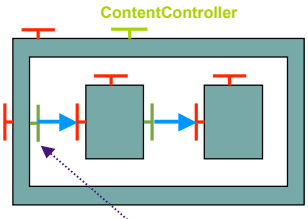
Lists the names of client interfaces of the component

Binds the client interface named c of C to the server interface s of S in the same

Returns the client interface of a given name

Configuration API (3/5)

- **ContentController**
  - Used to manage the hierarchical structure of components
  - Management of bindings between sub and super components (a.k.a. import/export bindings) belongs to content control



Lists the sub components of the component's content

Adds and removes a sub component from a component's content

```
interface ContentController {
  Component[ ] getFcSubComponents();
  void addFcSubComponent (Component c);
  void removeFcSubComponent (Component c);
  any[] getFcInternalInterfaces();
  any getFcInternalInterface (string c);
}

public interface SuperController {
   Component[] getFcSuperComponents ();
}
```

Access to internal interfaces of the component for import/export

Lists the component's enclosing components

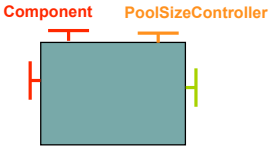
Configuration API (4/5)

- **AttributeController**
  - Attributes are configurable properties of components
  - Attributes control relies on a same design
    - Extension of the empty AttributeController interface
    - Accessors for read, write or read/write attributes

```
interface AttributeController {}
```

Defines a read/write attribute controller

```
interface PoolSizeController
extends AttributeController {
   int getPoolSize ();
   void setPoolSize (int Poolsize);
}
```

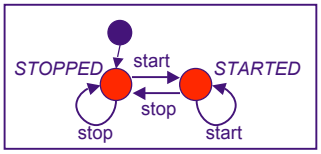


Configuration API (5/5)

LifeCycleController

- Semantics of start and stop are voluntarily as weak as possible
  - May implement usual suspend/resume or start/stop semantics
  - May or may not start/stop sub components
- The central point is the isolation of 2 states
  - STARTED in which components can accept operations only through their functrional interfaces
  - STOPPED in which components can accept operations only through their control interfaces
    - STOPPED aims to be a « safe state » for component reconfiguration but this point is completely left to implementations…
- LifeCycleController will often be extended or completely redefined to suit arbitrary life cycles

```
interface LifeCycleController {
   string getFcState();
   void startFc ();
   void stopFc ();
}
```



Basic Typing API (1/2)

Component type

- *Definition: a set of (functional) interface types*
- Component types are immutable, i.e. they cannot be changed at runtime

```
interface ComponentType {
   InterfaceType[] getFcInterfaceTypes();
   InterfaceType getFcInterfaceType (
         string itfName) ;
}
```

Interface type

- *Name*
- *Signature*
  - language type specified with the IDL
- *Contingency* : *optional* or *mandatory*
  - Guarantee or not that the interface will be available when the component is running
- *Cardinality* : *singleton* or *collection*
  - Number of interfaces of the same type a component may have

```
interface InterfaceType {
   string getFcItfName();
   void getFcItfSignature ();
   boolean isFcClientItf
   boolean isFcOptionalTif ();
   boolean isFcCollectionItf ();
}
```

Type Factory

- Creation of interface types and component types

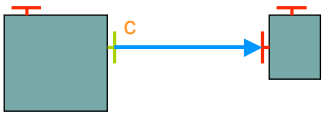
```
Interface TypeFactory {
   InterfaceType createFcType (
         string name,
         string signature,
         boolean isClient,
         boolean isOptional,
         boolean isCollection);
   ComponentType createFcType (
         InterfaceType[] itfTypes);
}
```

Sub-typing relation between types

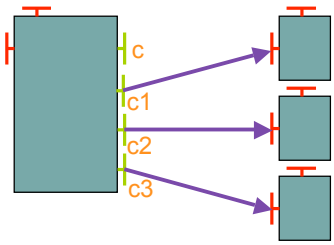
Basic Typing API (2/2)

- **Cardinality**
  - Single and collection
    - Singleton : exactly one interface of this type
    - Collection : a vector of interfaces of this type
  - Uses a lexicographical convention
    - Collection interfaces names start with the name of the type, e.g. c and c1, c2, c3
  - **Collection interfaces are created lazily**
    - by invocations of `bindFc` or `getFcInterface`
    - Complete names (e.g. c1, c2, c3) must be used in `bindFc` or `getFcInterface`
- **Motivations for cardinality**
  - Server interfaces : multiple interfaces with same signature but different implementations (QoS)
  - Client interfaces : listeners…
- **NB: Collection interfaces are "vectors of individual interfaces" - they do not have a multicast-like semantics…**

Singleton interface c



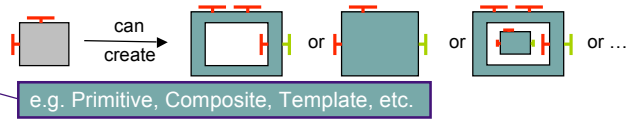
Collection interface c



Instantiation API

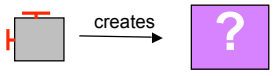
- **Generic (or parametric) factory components**
  - Can create components of arbitrary types given as inputs + description of control and content

```
interface GenericFactory {
   Component newFcInstance (
         Type t,
         any controllerDesc,
         any contentDesc);
}
```

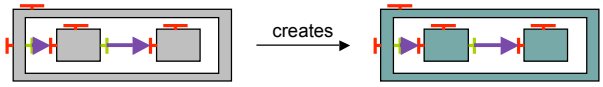


- **Standard factory components**
  - Each create components of one specific type, i.e. **they are explicitely programmed to do so**

```
interface Factory {
   Type getFcType ();
   any getFcControllerDesc ();
   any getFcContentDesc ();
   Component newFcInstance ();
}
```



- **Template components**
  - Components that create components similar (« isomorphic ») to themselves



- ***NB: Need for a boostrap component factory (since factories are themselves components)***

Conformance Levels

	Introspection		(Re)Configuration	Instantiation		Dynamic (Basic) Typing
	C	I	BC, CC, SC, LC, AC	F	T	
0						
0.1			X			
1	X		X			
1.1	X					
2	X	X				
2.1	X	X	X			
3	X	X				X
3.1	X	X	X			X
3.2	X	X	X	X		X
3.3	X	X	X	X	X	X

Legend :
C : Component
I : Interface
BC : BindingController
CC : ContentController
SC : SuperController
LC : LifeCycleController
AC : AttributeController
F : Factory
T : Template

Think

Julia, AOKell

Overview

- I - Fractal and CBSE
  - The Fractal project
  - Motivations
  - Component-Based Software Engineering
  - CBSE & Fractal
- II - Fractal Component Model
  - Principles
  - Concepts
  - Semantics
  - Programming Model (API Overview)
- III - Fractal Tool Chain
  - Fractal ADL
  - Julia
  - AOKell
- IV - Programming Example in Java
  - Programming
  - Configuring
  - Activating
  - Reconfiguring
- V - Ongoing Works around the Fractal project
  - Tools
  - Uses
- VI - Conclusion
  - Summary
  - Perspectives

ADLs Background

- **Software Architecture**
  - *"Sofware architecture is the **fundamental organization of a system,** embodied in its **components,** their **relationships** to each other and the **environment,** and the **principles** governing its **design** and **evolution"*** [IEEE 1471-2000]

- **Architecture Description Languages (ADLs)**
  - Analysis tools such as Wright, Darwin, ACME
  - Code generation tools UniCon, Olan, Darwin… ArchJava

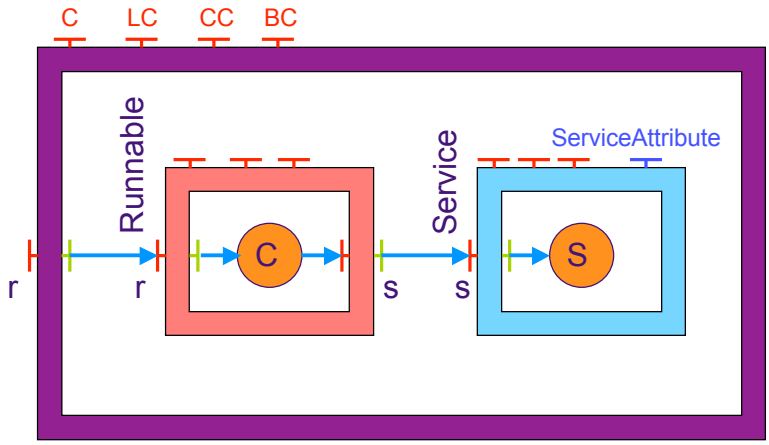
Fractal ADL

- **In brief**
  - A **language** for defining Fractal architectures (configurations description)
  - An associated **parsing tool** a.k.a. Fractal ADL Factory (cf. Section V of this tutorial)
- **More precisely**
  - a **modular** (XML modules defined by DTDs)
  - and **open** (extensible) language to describe
    - Components, interfaces and binding and containment relationships
    - Attributes
    - Typing
    - Implementations including membrane engineering
    - Deployment : packaging, distributed installation (virtual nodes)
    - Behaviour and QoS contracts
    - Logging
    - …
  - Hence Fractal ADL is both a code generation tool and a basis for analysis tools
- **Rational**
  - Open model
  - Several implementations in different programming languages
  - Several usages throughout the software lifecycle

HelloWorld example

Product Specification

- *The server component provides a server interface named "s" of type "`Service`", which provides a "`print`" method. It also has an attribute interface of type "`ServiceAttributes`", which provides four methods to get and set the two attributes of the server component.*
- *The client component provides a server interface named "r" of type "`Main`", which provides a "`main`" method, called when the application is launched. It also has a client interface named "s" of type "`Service`".*

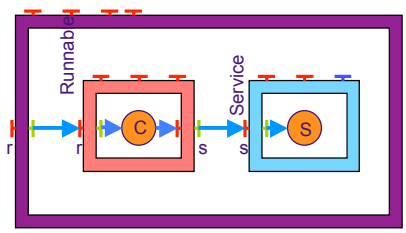


Primitive components

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
 <!DOCTYPE definition PUBLIC
    "-//objectweb.org//DTD Fractal ADL 2.0//EN"

"classpath://org/objectweb/fractal/adl/xml/basic.dtd"
>

 <definition name="ClientImpl">
   <interface name="r" role="server"
signature="java.lang.Runnable"/>
   <interface name="s" role="client"
signature="Service"/>
   <content class="ClientImpl"/>
 </definition>
```

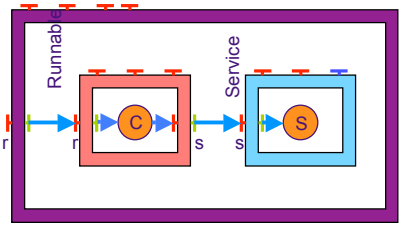
Definition of the 'basic' ADL module:

(interface*,component*,binding* ,content?,attributes?,controller?, template-controller?)



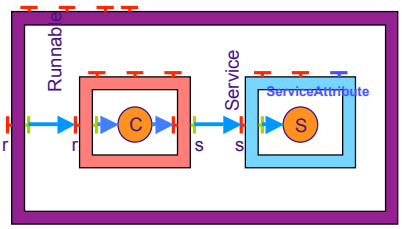
Composite components

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```



Component attributes and controllers

```
<definition name="ServerImpl">
  <interface name="s" role="server" signature="Service"/>
  <content class="ServerImpl"/>
  <attributes signature="ServiceAttributes">
    <attribute name="header" value="->"/>
    <attribute name="count" value="1"/>
  </attributes>
  <controller desc="primitive"/>
</definition>
```



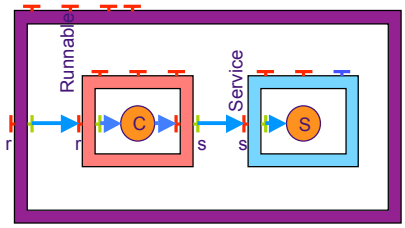
ControllerDesc: identifier of a controllers combination e.g.:
-FlatPrimitive: C, BC, LC, NC
-Primitive: C, BC, LC, NC, SC
-ParametricPrimitive: C, BC, LC, SC, NC, AC
-Composite: C, BC, LC, NC, CC, SC
-…
-ParametricCompositeTemplate: C, BC, CC, SC, LC, AC, NC

Component references (1)

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="client" definition="ClientImpl"/>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

Reuse of an already defined component

Limits the possibility to define alternative implementations for this component



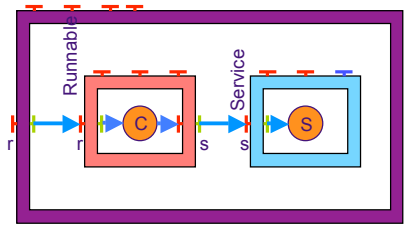
Component references (2)

```
<definition name="ClientType">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <interface name="s" role="client" signature="Service"/>
</definition>
```

Separation of interface and implementation

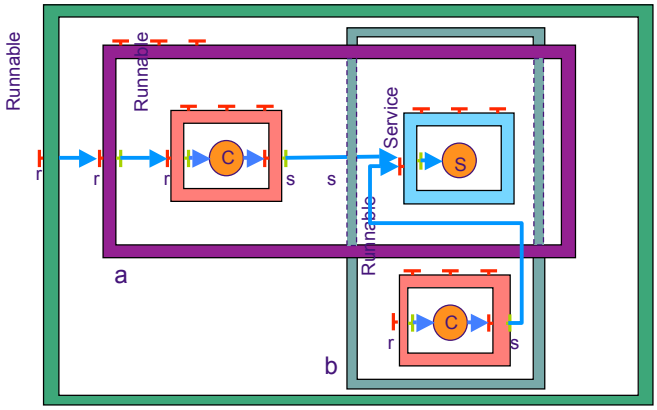
```
<definition name="ClientImpl" extends="ClientType">
  <content class="ClientImpl"/>
</definition>
```

Extension of a previous definition so as to separate interface and implementation



Component references (3)

```
<definition name="SharedHelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="a" definition="HelloWorld"/>
  <component name="b" definition="HelloWorld">
        <component name="server" definition="a/server"/>
  </component>
  <binding
        client="this.r"
        server="a.r"/>
</definition>
```



Arguments

```
<definition name="GenericServerImpl" arguments="impl,header,count">
  <interface name="s" role="server" signature="Service"/>
  <content class= "${impl}" />
  <attributes signature="ServiceAttributes">
    <attribute name="header" value= "${header}" />
    <attribute name="count" value= "${count}" />
  </attributes>
  <controller desc="primitive"/>
</definition>
```

Instantiation

```
Map context = new HashMap();
context.put("bootstrap", …); // optional

// Arguments
context.put("impl", "ServerImpl"); // if definition with arguments
context.put("header", "->");
context.put("count", "1");

// four backends can be used (Fractal/Java, static/dynamic)
Factory f = FactoryFactory.getFactory(FactoryFactory.FRACTAL_BACKEND);

Component c = (Component)f.newComponent(”HelloWorld", context);
```


Summary on Fractal ADL

- **An modular and extensible language for description of the initial architecture (configuration) of a system used for its activation**
  - as in UniCon, OLAN, Darwin
- **Can be used as a pivot format for tools**
  - e.g. Fractal GUI
- **Can be extended for deployment**
  - packaging, installation, etc.
- **Can be extended for verification**
  - as in Aesop, Wright, Rapide, Darwin, C2, ACME
- **Can be extended to be used dynamically**
  - « scripts » to express reconfigurations

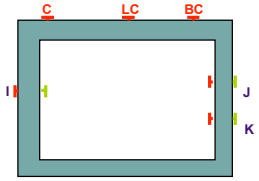
Julia

- **Objectives and motivations**
- **Components Implementation**
- **Membrane programming**
  - Mixin-Based Controllers
  - Interceptors
  - Life Cycle Management
- **Intra and Inter Component Optimization**
- **Bytecode transformation**
- **Julia configuration**
- **Conclusion**

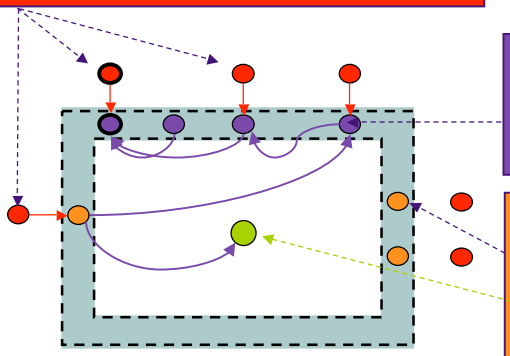
Objectives & Motivations

- **Objectives**
  - Provide a programming and execution support for Fractal components in Java
    - As “containers” JOnAS, WebLogic, WebSphere for EJB, OpenCCM for CCM
  - Support the highest Fractal conformance level (“reference implementation”)
    - Since Fractal is more much open than EJB or CCM, Fractal implementations could very different from one another
      - –Language
      - –Targetted environments
      - –Conformance levels : typing, instantiation, membranes programming
- **Motivations**
  - Extensible framework for membrane programming : interceptors and controllers composition
  - Continuum for static to dynamic composition (optimization trade-offs)
  - Support different java profiles including constrained environments
    - E.g. J2ME: no reflection, no collections, no class loaders

Component Implementation



Component **interfaces are typed**, i.e. implement **both** :
- `Interface` Fractal interface
- Functional (e.g. I, J, K) or control interfaces (e.g. C, LC, BC)



Controllers implement `Controller` Julia interface:

```
public interface Controller {
  void initFcController
    (InitializationContext ic);
}
```

Interceptors implement `Interceptor` Julia interface:

```
public interface Interceptor extends
   Controller {
  Object getFcItfDelegate ();
  void setFcItfDelegate (Object
   delegate);
  Object clone ();
}
```

Julia membrane = controllers + interceptors

Mixin-Based Controllers

- **Motivation: combinational problem due to the openness of Fractal**
  - Open set of controllers, multiple implementations of controllers, conformance level (typing), etc.
  - Fractal controllers - as aspects - are generally not orthogonal
- **Approach**
  - Controllers are built as a composition of control classes and mixins at loadtime
- **Definition**
  - ***A class mixin is a class whose super class is specified in an abstract way with the « minimum » fields and methods its must have. A mixin class can then be applied to any super class that defines at last these fields and methods.***
- **Lexicographic pattern-based Implementation**
  - **`_super_`***`<method_name>`* for required and overloaded methods
  - **`_this_`***`<method_name>`* for required but not overloaded methods
- **Result of of mixins composition depends on the order in which they are composed**
- **Example**

```
class BasicBindingController {
 // …
 String bindFc (…) {
  // … NO lifecycle related code … }}
```

```
abstract class LifeCycleBindingMixin {
abstract String _super_bindFc (…);}
String bindFc (…) {
   if (getFcState() != STOPPED)
     throw new Exception();
   return _super_bindFc(…); }
```

```
class <generated name> extends BasicBindingController {
 String bindFc (…) {
  if (getFcState() != STOPPED)
    throw new Exception();
  return super.bindFc(…);
 }
}
```

controllers = control classes * mixins

Interceptors

- **Interceptors**
  - Most control aspects have two parts
    - A generic part (a.k.a. “advice”)
    - A specific part based on interception of interactions between components (a.k.a. “joint points”, “hooks”)
  - Interceptors have to be inserted in functional (applicative) code
    - Interceptor classes are generated in bytecode form by an generator which relies on ASM
- **Interceptor class generator**
  - `G(class, interface(s), aspect code weaver(s)) -> subclass of class which implement interface(s) and aspect(s)`
  - Transformations are composed (in the class) in the order aspects code weavers are given
- **Aspect code weaver**
  - An object that can manipulate the bytecode of operations arbitrarily
  - Example:
    - Transformation of `void m { return delegate.m }`
    - Into `void m { // pre code… try {delegate.m();} finally {//post code… }}`
- **Configuration**
  - Interceptors associated to a component are specified at component creation time
  - Julia comes with a library of code weavers:
    - life cycle, trace, reification of operation names, reification of operation names and arguments

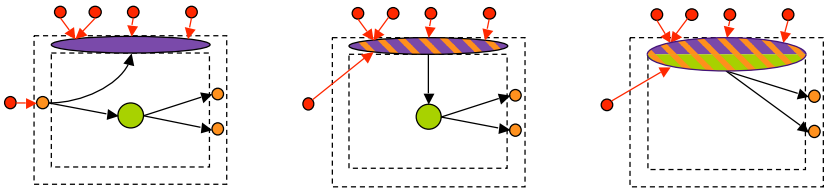
Life Cycle Management

- **Approach based on invocation count**
  - Interceptors behind all interfaces increment and decrement a counter in LifeCycle controller
  - LifeCycle controller
    - waits for counters to be nil to stop the component (STARTED->STOPPED)
    - when then component is in sate STOPPED, all activities (includind new incoming ones) are blocked
    - activities (and counter increment) are unblocked when the component is started again
- **Composite components stop recursively**
  - the primitive components in their content
  - and primitive client components of these components
    - Because of inter-component optimization (detailed later)
  - Same algorithm with n counters
  - NB: needs to wait for n counters to be nil at the same time - with a risk of *livelock*
- **Limitations**
  - Risk of livelock when waiting for n counters to be nil at the same time
  - No state management - hence integrity is not fully guaranteed during reconfigurations

Intra-component Optimization

3 possibilities for memory optimization

- Fusion of controller objects (left)
- Fusion of controller objects and interceptors (middle) if interceptors do all delegate to the same object
- Fusion of controllers and contents (right) for primitive components



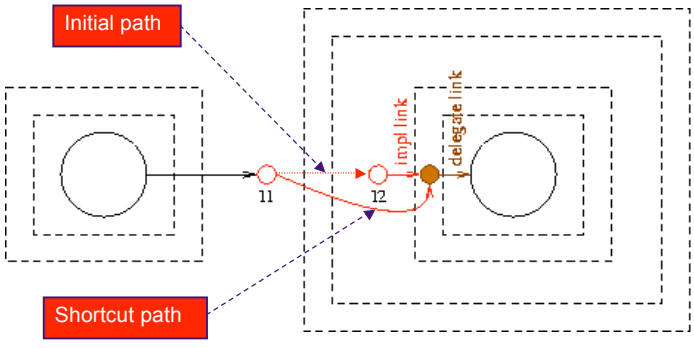
Merging is done in bytecode form by generating a class based on lexicographic patterns in concerned controller classes

- `weavableX` for a required interface of type X in controller is replaced by `this` in the generated class
- `weavableOptY` for a optional required interface of type Y is replaced by `this` or `null` in the generated class

Inter-component Optimization

Shortcut algorithm

- *Optimized* links for performance ("shortcuts") subtituted to *implementation* (⟶) and *delegate* links (⟶) in binding chains



NB:

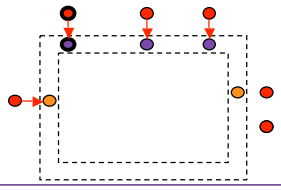
- behaviour is hazardous if components exchange references directly (e.g. **`this`**) instead of always using the Fractal API
- Shorcuts must be recomputed each time a binding is changed

Bytecode Generation

- **Usage**
  - Objects representing component interfaces
  - Mixins composition
  - Interceptors generation
  - Controller classes fusion
  - Component factories generation (« compilation » of controller descriptions)
- **HelloWorld example**
  - 3 components : 1 composite, 2 primitives
  - 28 generated classes (126KB) : 8 application specific, 20 generic
    - 12 for objects representing component interfaces
    - 10 for mixins composition
    - 2 for interceptors generation
    - 4 for controller classes fusion
    - 4 for component factories generation
- **Dynamic or static generation based on ASM**
  - Similar functionnalities as SERP, BCEL
  - Smaller (33KB instead of 350KB for BCEL and 150KB for SERP)
  - Very efficient thanks to interactions-based representation (the overhead of a load time class transformation is of the order of 60% with ASM, 700% or more with BCEL, and 1100% or more with SERP)

Julia Configuration File (julia.cfg)

- **Defines**
  - component controller descriptors : values `primitive`, `parametricPrimitive`, `composite`... of variable `controllerDesc` in the instantiation API
    - Interfaces class generators
    - Control interfaces
    - Controllers (mixins composition)
    - Generators of interceptor classes
  - Merge-optimization options (shortcuts policy is specified by choosing mixins)
- **Uses a LISP-like syntax**
  - Définition = (*name definition*)
  - A définition can reference other definitions :
    - `(x (1 2 3) )`
      Value `of x` is `(1 2 3)`
    - `(y (a b c 'x) )`
      Value of `y` is `(a b c (1 2 3))`



```
...
(primitive
 ( 'interface-class-generator
  ( 'component-itf
   'binding-controller-itf
   'lifecycle-controller-itf )
  ( 'component-impl
   'container-binding-controller-impl
   'lifecycle-controller-impl )
  ( (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
     org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
     org.objectweb.fractal.julia.asm.TraceCodeGenerator) )
  org.objectweb.fractal.julia.asm.MergeClassGenerator
  none
 ) )
...
```

Conclusion on Julia

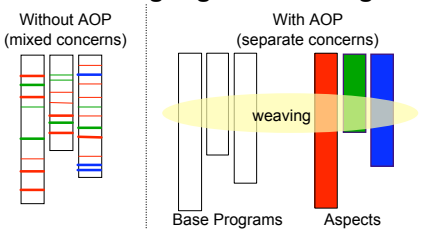
- **Summary**
  - Full fledge implementation of Fractal : support the highest conformance level
  - **Makes use of AOP-like techniques based on interception and mixins**
    - Comes with a library of mixins and interceptors mixed at loadtime
  - **Allows for intra-components and inter-components optimizations**
    - Very acceptable performance has been showed by the Dream use case (~+2% for completely reconfigurable configuration, no overhead in optimized settings)
  - Relies heavily on loadtime bytecode transformation as underlying techniques
  - Support different java profiles including constrained environments
    - JVMs without class loading or reflection
- **Status**
  - v0 march 2002, v1 july 2002, v2 november 2003
  - Released in ObjectWeb as a Fractal module
- **Future work**
  - Restrictions
    - Only one reconfiguration thread
    - No protection against malicious usage
  - Follow the Fractal specification evolutions and more generally support for activities management, component state management, integrity preserving dynamic reconfiguration…
  - NB: Use of actual AOP tool for controlers composition -> AOKell

AOKell

- **Backgound on AOP**
- **Control Aspect Weaving**
  - Injection-only aspect weaving
  - Interception-based aspect weaving
- **Membrane & Controller Components**
  - Rational and Principle
  - Example
  - Membrane Configuration with Fractal ADL
- **Conclusion**

Background on AOP

Code *tangling & scattering*



Concepts & Principle

- ***Aspect***: an entity modularizing a transversal concern (e.g. security, persistence…)
- ***Jointpoint***: a point in execution flow
- ***Advice***: the code inserted for an aspect when reaching a particular jointpoints
- ***Pointcut***: a set of jointpoints for a particular aspect
- ***Weaving:*** the mechanism which ties aspects to base programs by injecting advices at joinpoints

Approach

- Separation of concerns design/programming concept
  - divide-and-conquer
  - middleware: separate business & technical codes

Expected benefits

- Increase modularity
- Decrease development, maintenance, evolution costs by aspects reuse

AOP community

- Typical AOP tools: AspectJ, JAC, PROSE, Spring-AOP, JBOSS-AOP
- AOP Alliance for Java: APIs for reflection, interception, instrumentation, classloading

NB: backgrounf of AOP in reflection interception, program transformation

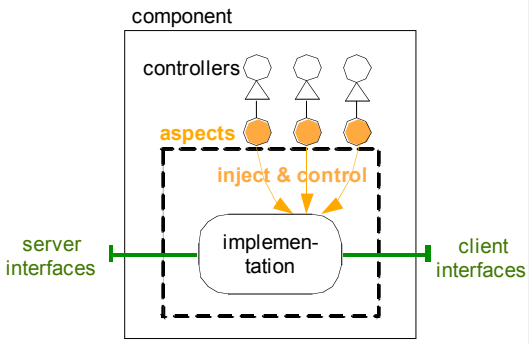
Control Aspects Weaving

"Pure aspect-oriented" component implementation

- Control is directly injected into the content - No need for "callbacks"
- "Type marking" instead e.g.

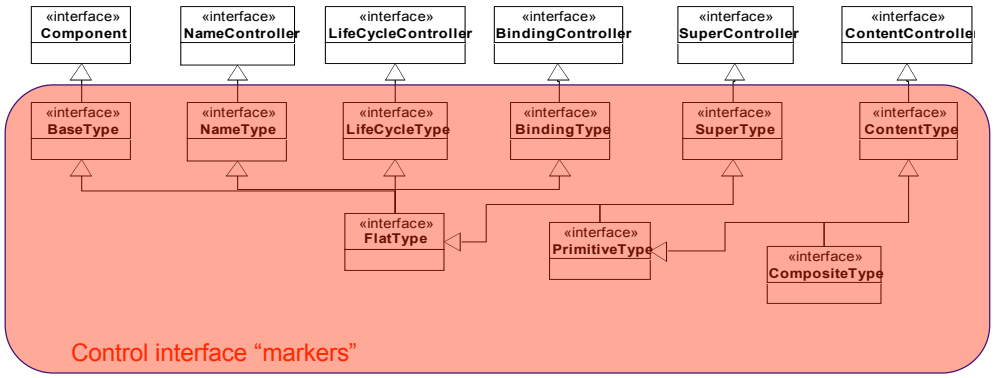
```
public class ClientImpl implements
  PrimitiveType {
  public ClientImpl() {}
  public void run() {
   Service s = (Service) lookupFc("s");
   s.message();
}}
```

- Benefit: methods from control interfaces can be called directly in the implementation (e.g. "lookupFc" above)



There also exists a "Julia-like" component implementation which allows for integration of "legacy" Julia components

Controller « type system »



NB: parametrics, templates and bootstrap are ommitted here
(currently 13 types of membranes)

Injection-only Aspect Weaving

➡ **Example: the Name controller aspect**

```
public aspect ANameController {

   private NameController NameType._nc;

   public String NameType.getFcName() {
      return _nc.getFcName();
   }
   public void NameType.setFcName(String arg0) {
      _nc.setFcName(arg0);
   }

  public NameController NameType.getFcNameController() {
     return _nc;
  }
  public void NameType.setFcNameController(NameController nc) {
     _nc=nc;
  }
}
```

AspectJ
Inter-type declarations
(≡ Feature injection)

Object implementation
of the name controller

Interception-based Aspect Weaving

➡ **Example: the lifecycle controller aspect**

```
public aspect ALifeCycleController {

   private LifeCycleController LCType._lc;

   public String LCType.getFcState() { return _lc.getFcState(); }
   public void LCType.startFc() throws IllegalLifeCycleException { _lc.startFc(); }
   public void LCType.stopFc() throws IllegalLifeCycleException { _lc.stopFc(); }

   pointcut methodsUnderLifecycleControl( LCType adviced ):
      execution( * LCType+.*(..) ) && target(adviced) &&
      ! controllerMethodsExecution() && ! jlObjectMethodsExecution();

   before(LCType adviced) : methodsUnderLifecycleControl(adviced) {
    if( adviced.getFcState().equals(LifeCycleController.STOPPED) ) {
      throw new RuntimeException("Components must be started before
                                          accepting method calls");}
    }
```

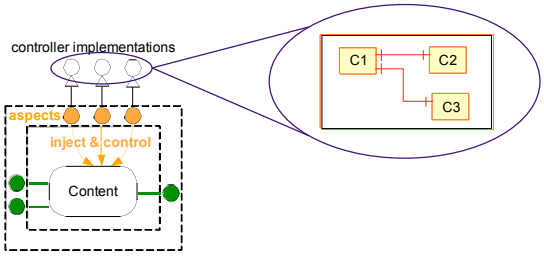

Membrane & Controller Components

➡ **Rational**

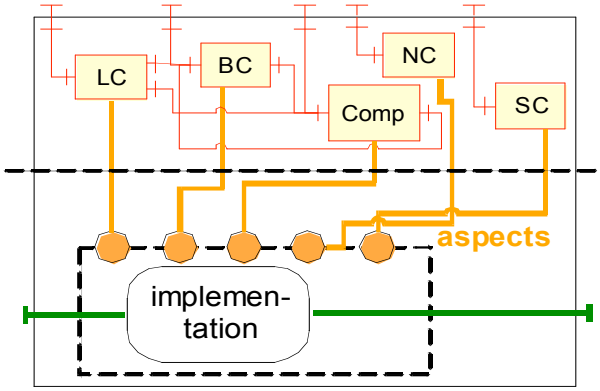
- Membranes can be arbitrarily complex in terms of number of controllers and dependencies between controllers
- Controllers can recursively be arbitrarily complex themselves
- Hence the need for “componentizing” membranes raises naturally so as to benefit from the component-based approach and tools

➡ **Principles**

- « Controllers » and « membranes » which are rather loosely defined concepts in the Fractal model are directly emboddied here into components
- Controller definition
  - A controller is a **component** that implements a control interface
- Membrane definition
  - A membrane is a (composite) component that contains an assembly of **controller components**



A Primitive Component



Controllers

BC : Binding
LC : Lifecycle
NC : Name
SC : Super
Comp : Component

Application level

Membrane Configuration

➡ **Example: the ”primitive” membrane**

```
<definition name="aokell.lib.membrane.primitive.Primitive"
      extends="LifeCycleType, BindingType, ComponentControllerType, NameControllerType, SuperControllerType" >

      <component name="Comp" definition="aokell.lib.control.component.PrimitiveComponentController" />
      <component name="NC"    definition="aokell.lib.control.name.NameController" />
      <component name="LC"    definition="aokell.lib.control.lifecycle.NonCompositeLifeCycleController" />
      <component name="BC"    definition="aokell.lib.control.binding.PrimitiveBindingController" />
      <component name="SC"    definition="aokell.lib.control.superc.SuperController" />

      <binding client="this.//component"          server="Comp.//component" />
      <binding client="this.//name-controller"    server="NC.//name-controller" />
      <binding client="this.//lifecycle-controller" server="LC.//lifecycle-controller" />
      <binding client="this.//binding-controller"  server="BC.//binding-controller" />
      <binding client="this.//super-controller"    server="SC.//super-controller" />

      <binding client="Comp.//binding-controller" server="BC.//binding-controller" />
      <binding client="BC.//component«             server="Comp.//component" />
      <binding client="LC.//binding-controller"    server="BC.//binding-controller" />
      <binding client="LC.//component"             server="Comp.//component" />

      <controller desc="mComposite" />
</definition>
```

Conclusion on AOKell

- **Summary**
  - Full fledge implementation of Fractal : support the highest conformance level
    - Tested on Comanche, Fractal RMI, Fractal Explorer, GoTM, DREAM
  - **Makes use of AOP tools AspectJ, Spoon** (also ported to the .NET platform: FractNet)
    - Currently static and load-time weaving
  - Allows for object-based membranes and **component-based membranes**
    - Control interfaces are implemented by controller components
    - Membranes are composite that encapsulate controller components
  - Allows for reuse of components implementation coded for Julia
    - **Otherwise no need for callbacks**
  - Same performance as Julia (no penalty from componentized membranes)
  - Support different java profiles including constrained environments
    - JVMs without class loading or reflection
- **Status**
  - v0 march 2005, v1 february 2006
  - Released in ObjectWeb as a Fractal module
- **Future work**
  - Julia-AOKell interoperability
    - Mixed use of Julia and AOKell components in a single system
  - More generally definition of a Service Provider Interface (SPI opposed to API) for Fractal implementations
  - Dynamic membranes reconfiguration (needs runtime aspect weaving)

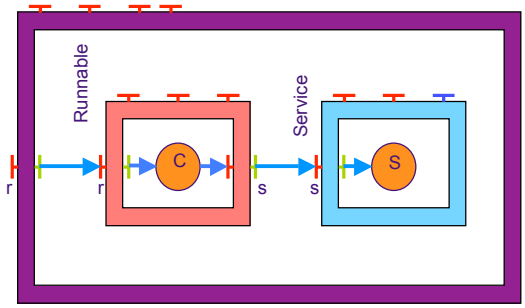
Conclusion on Fractal Tool Chain

- **Julia**
  - First effort towards control engineering based on well-known techniques: bytecode injection and mixins
  - First study of optimization possibilities (performance, memory) in CBSE
    - First studies such as Dream demonstrates the viability of CBSE ("*components are not necessarily expensive*")
    - …not to mention industrial uses through JORM, Speedo, JOnAS, CLIF since then
- **AOKell**
  - Innovative aspect-based membrane engineering (***control composition***)
  - First effort towards component-based membrane engineering (***control architecture***)
  - Part of a wider study of components & aspects (INRIA Jacquard, France Telecom)
- **Fractal ADL**
  - Favorite entry point to Fractal components programming
    - The ADL embeds concepts of the Fractal component model
  - Not (yet) a complete component-oriented language (in the Turing sense) - hence still need support for host programming languages a.k.a. "implementations" e.g. Julia and AOKell in Java, Think in C, etc.
  - Ongoing efforts (e.g. scripting facilities) towards a dynamic ADL (reconfigurations) - possibly leading (mid to long term) to a complete component-oriented language covering software lifecycle

Overview

- I - Fractal and CBSE
  - The Fractal project
  - Motivations
  - Component-Based Software Engineering
  - CBSE & Fractal
- II - Fractal Component Model
  - Principles
  - Concepts
  - Semantics
  - Programming Model (API Overview)
- III - Fractal Tool Chain
  - Fractal ADL
  - Julia
  - AOKell
- IV - Programming Example in Java
  - Programming
  - Configuring
  - Activating
  - Reconfiguring
- V - Ongoing Works around the Fractal project
  - Tools
  - Uses
- VI - Conclusion
  - Summary
  - Perspectives

Programming Example



- Focus
  - Development activities & deployment activities - both covered by Fractal APIs
  - Constraints on components implementations due to Julia, AOKell (“programming styles”)
  - Configuration by different means: API only, ADL, GUI
  - Reconfiguration by APIs or tools e.g. Fractal Explorer

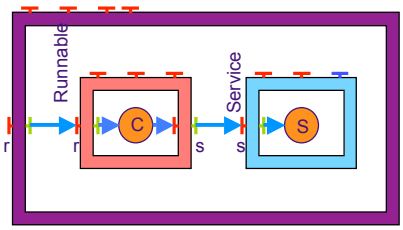
Software Life Cycle

- **Development activities**
  - Specify requirements
  - Specify product
  - Design
  - Code
  - Test
  - Validate
- **Deployment activities**
  - Install
  - Configure
  - Instantiate
  - Activate
  - Deactivate
  - (Re)configure (update)
  - Uninstall

- **A few comments**
  - Development and deployment processes can be arbitrarily complex
    - Source vs executable
    - Versions, variants management
  - Deployment is not so well known/studied - especially for component-based systems
    - Components coming from differents providers can only be tested/validated individually by providers
  - Interactions between development and deployment is not so well known/studied
    - Release/Unrelease
    - Pack/Unpack
    - Transfer

Scenario

- **Development**
  - Specify requirements
  - Specify product
  - Design
  - **Code components (Step 1)**

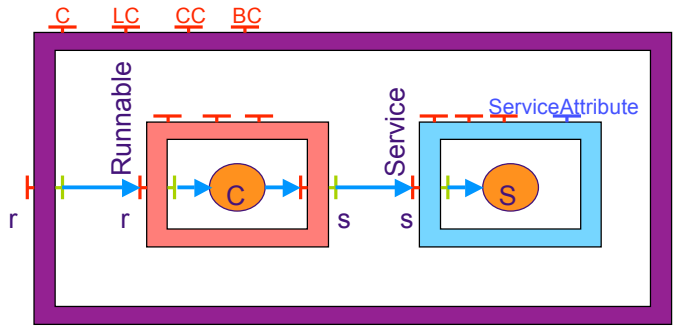


- **Deployment**
  - **Configure system (Step 2)**
  - **Instantiate system (Step 3)**
  - **Activate system (Step 4)**
  - ….
  - **Deactivate system (Step 5)**
  - **Reconfigure system (Step 6)**
    - **Replace the implementation of the Server component**
  - **(Re)activate system (Step 7)**

Collectively referred to as "management" activities

Developing, Deploying and Reconfiguring HelloWorld

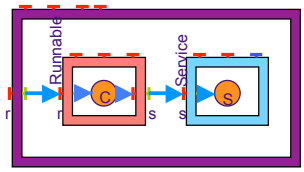
- **Requirements Specification**
  - *A Client/Server system which provides an interface to print messages on the console that can be parameterized by two attributes : a "header" attribute to configure the header printed in front of each message, and a "count" attribute to configure the number of times each message should be printed*
- **Product Specification**
  - *The server component provides a server interface named "s" of type "`Service`", which provides a "`print`" method. It also has an attribute interface of type "`ServiceAttributes`", which provides four methods to get and set the two attributes of the server component.*
  - *The client component provides a server interface named "r" of type "`Main`", which provides a "`main`" method, called when the application is launched. It also has a client interface named "s" of type "`Service`".*
- **Design**



Step 1 - Code components

- **Step 1.1 - Code the interfaces**
- **Step 1.2 - Code the server implementation**
- **Step 1.3 - Code the client implementation**
  a. for Julia (or AOKell-JL)
  b. for AOKell

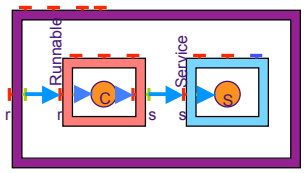
Step 1.1 - Coding component interfaces




```
public interface Runnable {
  void main (String[] args);
}

public interface Service {
  void print (String msg);
}

public interface ServiceAttributes
  extends AttributeController
{
  String getHeader ();
  void setHeader (String header);
  int getCount ();
  void setCount (int count);
}
```

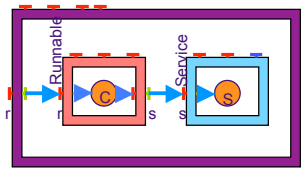
- **Attributes are configurable properties of components**
- **Attributes control relies on a design pattern**
  - Extension of the empty AttributeController interface
  - Accessors for read, write or read/write attributes

Step 1.2 -Coding the serveur component implementation




```
public class S implements Service, ServiceAttributes {

  private String h = "header";
  private int c = 1;

  public void print (String msg) {
    for (int i = 0; i < c; ++i) {
      System.out.println(h + msg);
    }
  }
  public String getHeader () { return header; }
  public void setHeader (String header) { h = header; }
  public int getCount () { return count; }
  public void setCount (int count) { c = count; }
}
```

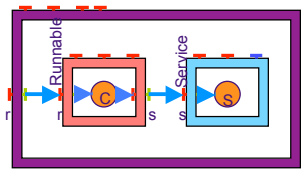
Step 1.3a - Coding the client component implementation for Julia (or AOKell-JL)




```
public class C implements Runnable,
BindingController {
  private Service service;
  public void run () {
    service.print("hello world");
  }
  public String[] listFc () {return new
String[]{"s'}}
  public Object lookupFc (String name) {
    if (name.equals("s")) return service;
    return null;
  }
  public Object bindFc (String name, Object
itf){
    if (name.equals("s")) service = (S)itf;
  }
  public Object unbindFc (String name){
    if (name.equals("s")) service = null;
  }
```

```
interface BindingController {
  void bindFc (string c, any s);
  void unbindFc (string c);
  string[] listFc();
  any lookupFc (string c);
}
```


Step 1.3b - Coding the client component implementation for AOKell

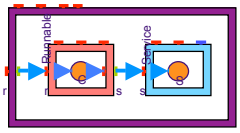



```
public class C implements Runnable, PrimitiveType {
  private Service service;
  public void run () {
    try {
          service = (Service) lookupFc("s");
      }
      catch( NoSuchInterfaceException nsie ) {
          throw new RuntimeException(nsie.getMessage());
      }
      service.print("hello world");
  }
}
```

Step 2-4 - Configure, Instantiate, Activate with API only

- **Step 2 - Configure system**
  - Step 2.1 - Coding Fractal types
  - Step 2.2 - Create Fractal templates
  - Step 2.3 - Configure Templates
- **Step 3 - Instantiate system from templates**
  - Step 3.1 - Create templates
  - Step 3.3 - Configure templates
- **Step 4 - Activate system**

Step 2.1 - Coding Fractal Types



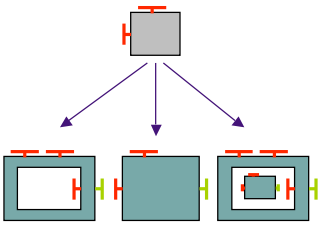

```
ComponentIdentity boot =
   Fractal.getBootstrapComponent();
TypeFactory tf =
   (TypeFactory)boot.getFcInterface("type-
   factory");
ComponentType rType = tf.createFcType(new
   InterfaceType[] {
   tf.createFcItfType(”r", ”Runnable", false,
   false, false)});
ComponentType cType = tf.createFcType(new
   InterfaceType[] {
   tf.createFcItfType(”r", ”Runnable", false,
   false, false),
   tf.createFcItfType("s", "Service", true, false,
   false)});
ComponentType sType = tf.createFcType(new
   InterfaceType[] {
   tf.createFcItfType("s", "Service", false, false,
   false),
   tf.createFcItfType("attribute-controller",
   "ServiceAttributes",…)});
```

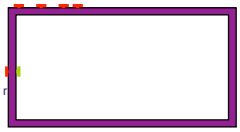
```
Interface TypeFactory {
   InterfaceType createFcItfType (
          string name,
          string signature,
          boolean isClient,
          boolean isOptional,
          boolean isCollection);
   ComponentType createFcType (
          InterfaceType[]
   itfTypes);
}
```

Step 2.2 - Create Templates

```
GenericFactory gf =
   (GenericFactory)boot.getFcInterface("generic-
   factory");


Component rTmpl =
   gf.newInstance(rType,"compositeTemplate", new
   Object[]{"composite",null)});


Component cTmpl =
   gf.newInstance(cType,"template", new
   Object[]{"primitive","C")});


Component sTmpl =
   gf.newInstance(sType,"template", new
   Object[]{"primitive","S")});
```

```
interface GenericFactory {
   Component newFcInstance (
         Type t,
         any controllerDesc,
         any contentDesc);
}
```



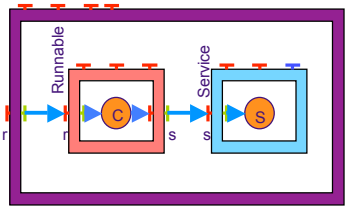


Step 2.3 - Configure Templates

```
ContentController cc = (ContentController)
   rTmpl.getFcInterface("content-controller);

cc.addFcSubComponent(cTmpl);

cc.addFcSubComponent(sTmpl);


((BindingController)rTempl.getFcInterfac
   e("binding-controller")).bindFc("r",
   cTmpl.getFcInterface("r"));


((BindingController)cTempl.getFcInterfac
   e("binding-controller")).bindFc("r",
   sTmpl.getFcInterface("s"));
```

```
interface ContentController {
  Component[ ] getFcSubComponents();
  void addFcSubComponent (Component c);
  void removeFcSubComponent (Component c);
  any[] getFcInternalInterfaces();
  any getFcInternalInterface (string c);
}

public interface SuperController {
   Component[] getFcSuperComponents ();
}
```

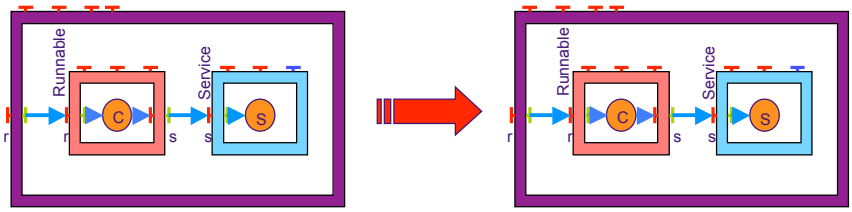
```
interface BindingController {
  void bindFc (string c, any s);
  void unbindFc (string c);
  string[] listFc();
  any lookupFc (string c);
}
```



Step 3 - Instantiate components from Templates

```
interface Factory {
   Type getFcType ();
   any getFcControllerDesc ();
   any getFcContentDesc ();
   Component newFcInstance ();
}
```

```
Component hw =
  ((Factory)rTmpl.getFcInterface("factory")).newFcInstance();
```

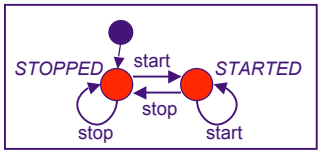


Step 4 - Activate (start)

```
((LifeCycleController)hw.getFcInterface("lifecyle-controller")).startFc();
```

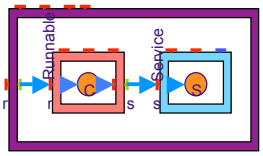
```
interface LifeCycleController {
   string getFcState();
   void startFc ();
   void stopFc ();
}
```

- **Semantics of start and stop are voluntarily as weak as possible**
  - May implement usual suspend/resume or start/stop semantics
  - May or may not start/stop sub components
- **The central point is the isolation of 2 states**
  - STARTED in which components can accept operations only through their functrional interfaces
  - STOPPED in which components can accept operations only through their control interfaces
    - STOPPED aims to be a « safe state » for component reconfiguration but this point is completely left to implementations…



Step 2-4 - Configure, Instantiate, Activate with ADL

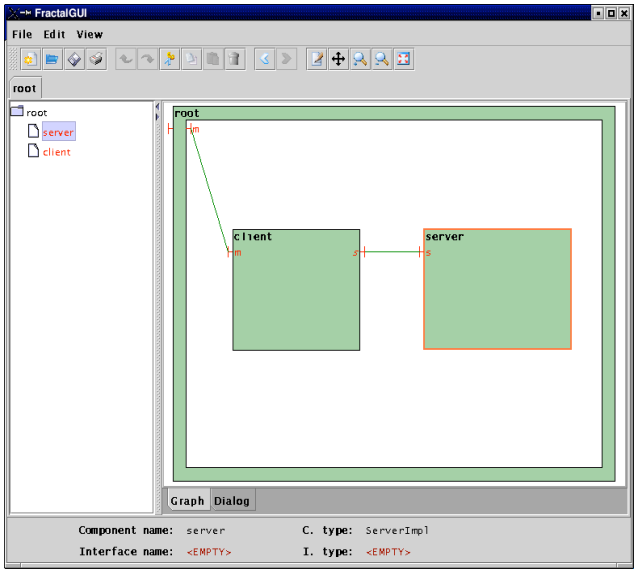
```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```



+ Instantiation by program

```
Factory f = FactoryFactory.getFactory(FactoryFactory.FRACTAL_BACKEND);
Component c = (Component)f.newComponent("HelloWorld", null);
```


Step 2-4 - Configure, Instantiate, Activate with GUI



Overview

- I - Fractal and CBSE
  - The Fractal project
  - Motivations
  - Component-Based Software Engineering
  - CBSE & Fractal
- II - Fractal Component Model
  - Principles
  - Concepts
  - Semantics
  - Programming Model (API Overview)
- III - Fractal Tool Chain
  - Fractal ADL
  - Julia
  - AOKell
- IV - Programming Example in Java
  - Programming
  - Configuring
  - Activating
  - Reconfiguring
- V - Ongoing Works around the Fractal project
  - Tools
  - Uses
- VI - Conclusion
  - Summary
  - Perspectives

Ongoing work around Fractal

- **Extending & refining Fractal technology**
  - formal basis: INRIA Sardes
  - dynamic reconfiguration: France Telecom, INRIA Sardes
  - management (admin): France Telecom
  - component behavior, formal verification, test: U. Prague, I3S/U. Nice, France Telecom, INRIA (INRIA Sardes, Oasis, Vasy), Valoria
  - security, isolation: France Telecom
  - Qos management (INRIA Sardes, France Telecom)
  - combining components & aspects: INRIA (Jacquard, Sardes), France Telecom, U. Prague
  - ADL support & programming tools
    - INRIA (Jacquard, Obasco, Sardes), France Telecom, STMicroelectronics
  - Packaging, deployment: INRIA (Jacquard, Sardes, Oasis), LSR, ENSTB
  - …

Ongoing work around Fractal

- **Using Fractal**
  - building infrastructure software (OS, middleware)
    - INRIA (Sardes, Oasis, Jacquard)
      - operating systems, asynchronous middleware, transaction management, Grid middleware, system management
    - IMAG-LSR
      - database management systems, persistency middleware
    - France Telecom
      - operating systems, persistency middleware, system monitoring and benchmarking, system management, M2M platform
    - STMicroelectronics
      - operating systems, multimedia programming
  - building frameworks for adaptive applications (multimedia, mobile, context-aware)
    - INRIA (Sardes, Obasco)
    - France Telecom
    - ENM Douai
    - Nokia

Sample software projects using Fractal

- **Operating system kernels**
  - Think (France Telecom, STMicroelectronics & INRIA Sardes)
- **Asynchronous middleware & communication subsystems**
  - DREAM (INRIA Sardes)
- **Transaction management**
  - GOTM, Jironde (LIFL-INRIA Jacquard, INRIA Sardes)
- **Persistency services**
  - Perseus (France Telecom, IMAG-LSR), Speedo (France Telecom), JOnAS persistency (France Telecom)
- **Middleware for Grid applications**
  - Proactive (INRIA Oasis)
- **Self-adaptive structures**
  - (EMN-INRIA Obasco, Nokia)
- **Distributed systems management**
  - Jade (INRIA Sardes), Jasmine (Bull)
- **Performance evaluation**
  - CLIF (France Telecom)
- **Middleware for Enterprise Application Integration**
  - Petals (EBM Websourcing)

Overview

- I - Fractal and CBSE
  - The Fractal project
  - Motivations
  - Component-Based Software Engineering
  - CBSE & Fractal
- II - Fractal Component Model
  - Principles
  - Concepts
  - Semantics
  - Programming Model (API Overview)
- III - Fractal Tool Chain
  - Fractal ADL
  - Julia
  - AOKell
- IV - Programming Example in Java
  - Programming
  - Configuring
  - Activating
  - Reconfiguring
- V - Ongoing Works around the Fractal project
  - Tools
  - Uses
- VI - Conclusion
  - Summary
  - Perspectives

Fractal: Summary

- **FRACTAL : from objects to reflective components to build manageable systems**
  - Interfaces (objects)
  - Explicit connections (components)
  - Membranes (reflective components)
- **FRACTAL : computational model for open distributed systems**
  - open binding semantics
  - open reflection semantics
  - extensible ADL

Fractal: perspectives

Fractal v3

- refined specifications (sharing, controller library, language mappings)
- multiple mature implementations (Julia v3, AOKell v1, Think v3)

Developing Fractal technology

- extensible & retargettable ADL compiler
- formal semantics for full Fractal
- strongly typed, dynamic ADL
- verification and validation tools
- additional language mappings and implementations
- model refinement: failures, transactions, aspects

Fractal perspectives

Using Fractal in ObjectWeb projects

- Software deployment & configuration management
- Autonomic system management
- Asynchronous middleware & Enterprise Service Bus (JBI implementation)
- JOnAS v5
  - at least service deployment and configuration
- JOnAS v6
  - fully Fractal-based implementation and EJB container

Thank you!

Getting more information

- Web site: http:fractal.objectweb.org
- Mailing-list: fractal@objectweb.org

Acknowledgments

- Some slides are based on material from E. Bruneton and L. Seinturier