Le modèle de composants Fractal

ICAR 2008

Lionel Seinturier
Université Lille 1 – LIFL & INRIA Lille ADAM

Plan

1. Introduction

Contexte : ingénierie du système/intergiciel (*middleware*)

Passé (année 1990) : objet mouvance « à la CORBA »

Besoins

- configuration
- déploiement
- empaquetage (*packaging*)
- assemblage
- dynamicité
- gestion des interactions et des dépendances

Présent : plusieurs tendances

- composant, aspect, MDE, réflexivité

1. Introduction

Composant vs objet

- plus haut niveau abstraction
- meilleure encapsulation, protection, autonomie
  - programmation + systématique + vérifiable
- communications plus explicites
  - port, interface, connecteur
- connectables
  - schéma de connexion (ADL) : « plan » applicatif
- séparation « métier » - technique
- meilleure converture du cycle de vie
  - conception, implémentation, *packaging*, déploiement, exécution

1. Introduction

Définition composant

- 1ère apparition terme [McIlroy 68]
- 30 ans + tard : Sun EJB, OMG CCM, MS .NET/COM+, …
- recensement [Szyperski 02] : 11 définitions +/- ≡

A component is a unit of composition with **contractually specified interfaces** and **context dependencies** only. A software component can be **deployed** independently and is subject to **composition** by third parties. [Szyperski 97]

1. Introduction

Nombreux modèles de composant (20+)

- construits au-dessus Java, C, C++

- EJB, Java Beans, CCM, COM+, JMX, OSGi, SCA, CCA, SF
- Fractal, K-Component, Comet, Kilim, OpenCOM, FuseJ, Jiazzi, SOFA, ArticBeans, PECOS, Draco, Wcomp, Rubus, Koala, PACC-Pin, OLAN, Newton, COSMOS, Java/A, HK2
- Bonobo, Carbon, Plexus, Spring

- au niveau analyse/conception : UML2

1. Introduction

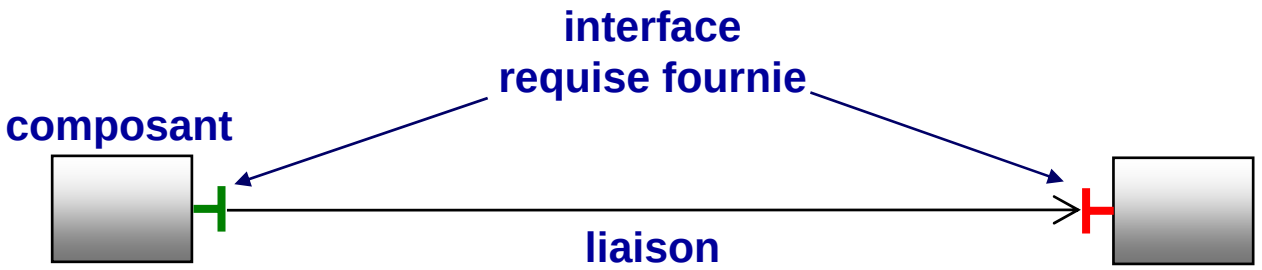
Conséquence de la multiplicité des modèles

- multiplicité du vocabulaire
  - composant, *bean*, *bundle*
  - interface/liaison, port/connecteur, facette, puits, source
  - requis/fourni, client/serveur, export/import, service/référence
  - conteneur, membrane, services techniques, contrôleur
  - *framework*, serveur d'applications
- grande variabilité dans les propriétés attachées aux notions
- exemples
  - Fractal : composant, interface, liaison, client/serveur
  - CCM : composant, facette, port, puits, source
  - UML 2 : composant, fragment, port, interface
  - OSGi : *bundle*, *package* importé/exporté, service/référence
- un même terme peut avoir des acceptations ≠ selon les modèles
- ➢ qualifier les notions (« connecteur au sens … »)
- ➢ pas toujours facile de définir les équivalences

1. Introduction

1ère grande catégorie de modèle de composants

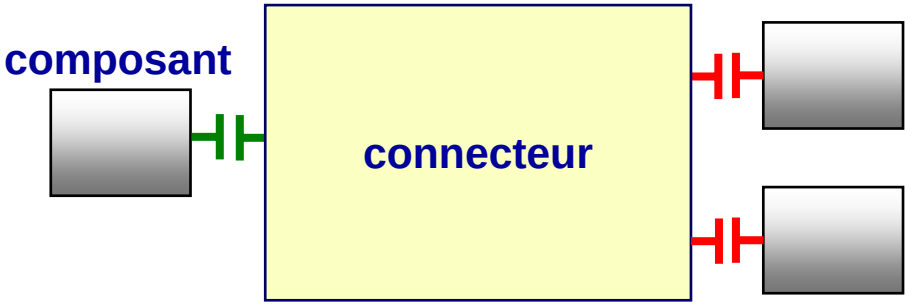
- triptyque : composant, interface, liaison
  - un composant fourni et/ou requiert une ou plusieurs interfaces
  - une liaison est un chemin de communication
    entre une interface requise et une interface fournie



1. Introduction

- 2ème grande catégorie de modèle de composants

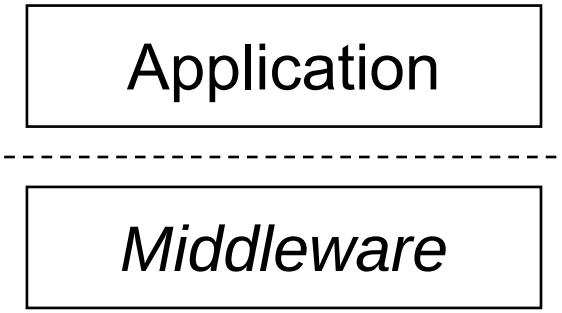
- triptyque : composant, port, connecteur
  - un composant fourni et/ou requiert une ou plusieurs ports
  - un connecteur implémente un schéma de communication entre des composants (client/serveur, diffusion, etc.)
  - un composant est relié à un connecteur via un ou plusieurs ports



- connecteur ≈ liaison avec comportement
- ➢ on peut considérer connecteur = composant (de communication)
- ➢ composant, interface, liaison

1. Introduction

Classification des modèles pour système/intergiciel

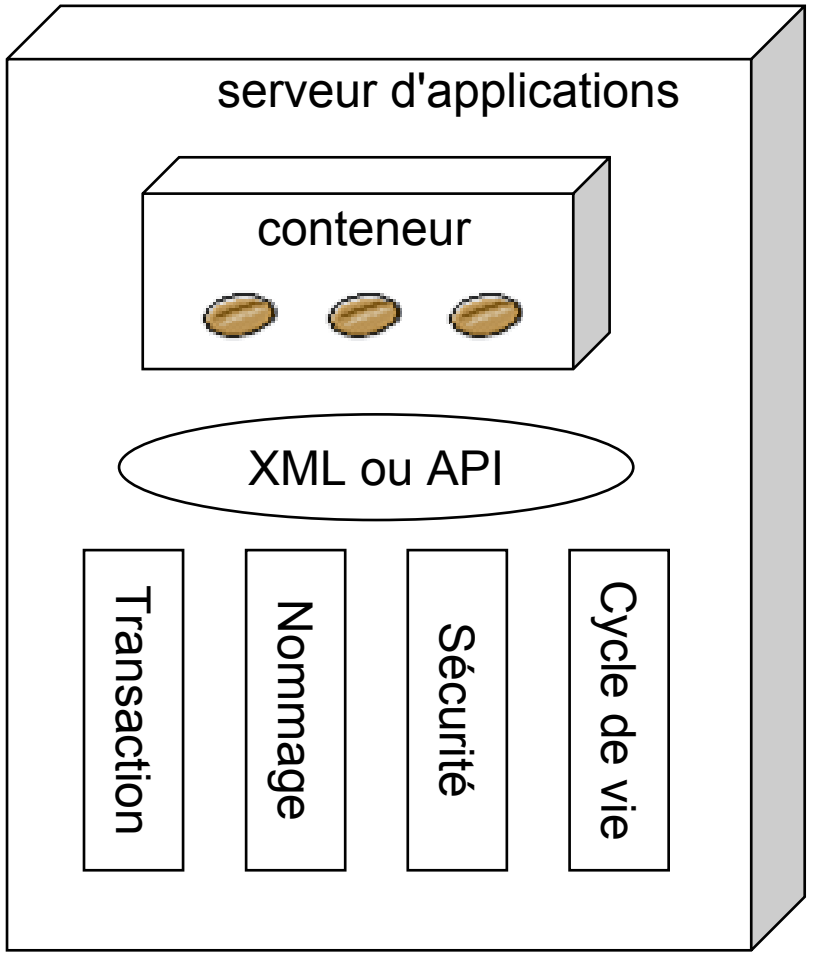


- application : EJB, CCM, .NET/COM+, SCA, Spring
- *middleware* : Fractal, JMX, OpenCOM, OSGi
- *middleware* componentisé pour applications à base de composants
  - JonasALaCarte : Fractal + EJB [Abdellatif 05]
  - OSGi + EJB [Desertot 05]

1. Introduction

Modèles EJB, CCM, .NET/COM+

- focus sur séparation métier/technique
- cibles : applications Internet, système d'information
- *packaging* ++
- déploiement ++
- architecture pauvre
- services figés, pas adaptables



1. Introduction

Quelques « poncifs » à propos des composants

- COTS Commercial Off The Shelf
  - vieux discours (voir procédures, fonctions, objet, …)
  - taille applis ↗ donc besoin : toujours plus de réutilisation
  - mais *quid* de la contractualisation ?

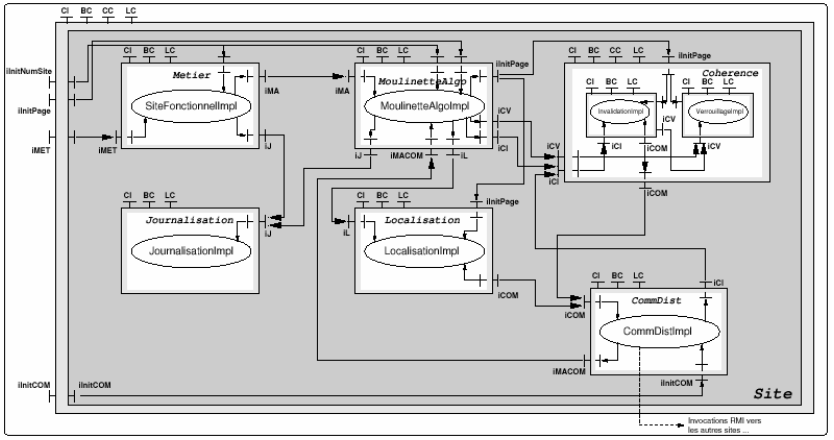
- « *Programming in the large* »
  - vs « *programming in the small* » (objet)
  - vrai d'un certain point de vue
  - mais nouveaux points à traiter (liés au non fonctionnel par ex.)

1. Introduction

Notion d'architecture logicielle

A software architecture of a program or computing system is the structure or **structures** of the system, which comprise **software components**, the externally visible **properties** of those components, and the **relationships** among them. [Bass 98]

- langage : ADL
- souvent en XML
- ➢ *survey* : [Medvidovic 00]



1. Introduction

Complémentarité

- architecture : construite à partir de composants
- composants architecture : assemblés pour construire une

2 visions complémentaires

- architecture : *top-down*
- composants : *bottom-up*

Plan

2. Le modèle Fractal

FT R&D, INRIA

- *open source*
- http://fractal.ow2.org

Historique

- fin 2000 : premières réflexions autour de Fractal
- 06/2002
  - 1ère version stable API
  - implémentation de référence (Julia)
  - 1ère version de l'ADL
- 01/2004
  - définition de l'ADL v2 (ADL extensible)
  - implémentation disponible 03/2004

2. Le modèle Fractal

- ingénierie des systèmes et du *middleware*
- suffisamment général pour être appliqué à tout autre domaine
- grain fin (wrt EJB ou CCM) proche d'un modèle de classe
- léger (surcoût faible par rapport aux objets)
- indépendant des langages de programmation

- vision homogène des couches (OS, *middleware*, services, applications)
  - Fractal *everywhere*
- dans le but de faciliter et d'unifier
  - conception, développement, déploiement, administration

2. Le modèle Fractal

- ouvert et adaptable
  - les services extra-fonctionnels peuvent être personnalisés
  - il n'y a pas une seule "forme" de composants Fractal

- 2 usages possibles avec Fractal
  - *framework* de composants pour construire des applications/systèmes
    - on utilise la forme "standard" des composants Fractal
  - *framework* de *frameworks* de composants
    - on construit d'autres "formes" de composants
      - avec introspection minimale et aggregation simple (à la COM)
      - avec contrôleurs de liaison et de cycle de vie (à la OSGi)
      - avec hiérarchie à 2 niveaux et liaison (à la SCA)
      - avec des liaisons multicast (à la CCA)
      - avec des contrôleurs d'attribut (à la MBean)
      - avec des contrôleurs de persistance et de transaction (à la EJB)
      - …
    - on développe des applications avec ces autres "formes"

2. Le modèle Fractal

- 1 modèle : spécification textuelle + API
- 1 sémantique (kell-calculus) [Stefani 03]

- plusieurs plates-formes
  - 3 en Java
    - Julia implémentation de référence
    - AOKell aspects + componentisation des membranes
    - ProActive composants actifs pour les grilles
  - 2 en C (Think, Cecilia), 1 en C++ (Plasma), 1 en SmallTalk (FracTalk), 1 pour .NET (FractNet)
- ➢ ≠ implémentations pour ≠ besoins

2. Le modèle Fractal

Exemple de *middleware*/applications développées avec Fractal

- comanche : serveur Web
- Speedo : persistance données Sun JDO [Chassande 05]
- GoTM : moniteur transactionnel [Rouvoy 04]
- Joram Dream : serveur JMS Scalagent [Quéma 05]
- JonasALaCarte : serveur Java EE [Abdelatif 05]

- Petals : ESB JBI [EBMWebsourcing / OW2]
- FraSCAti : plate-forme SCA [INRIA / ANR SCOrWare]

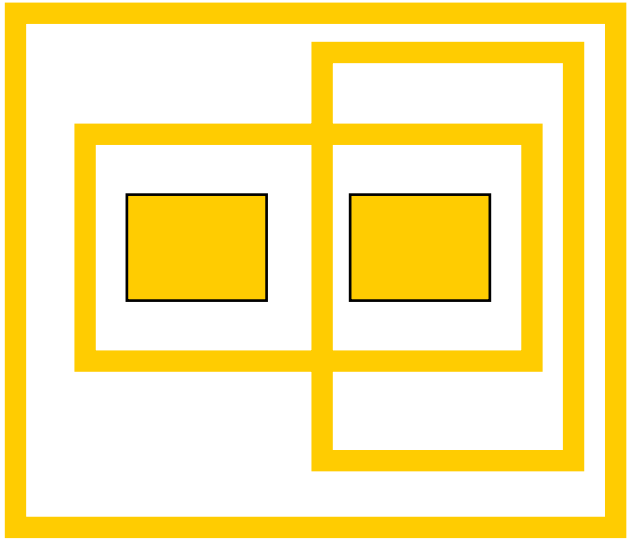
- FractalGUI : conception d'applications Fractal
- FractalExplorer : console d'administration applications Fractal

- serveur données répliquées + cohérence Li & Hudak [Loiret 03]

Plan

2.1.1 Composant

- unité de base d'une application Fractal
- modèles de type et d'instance
- entités *compile-time* et *run-time*



- modèle hiérarchique
  - composant composite
  - composant primitif
- notion de partage
  - modélisation de ressources communes

    par ex.: données, *pools*, caches, activités (*threads*, processus, transactions)

2.1.1 Composant

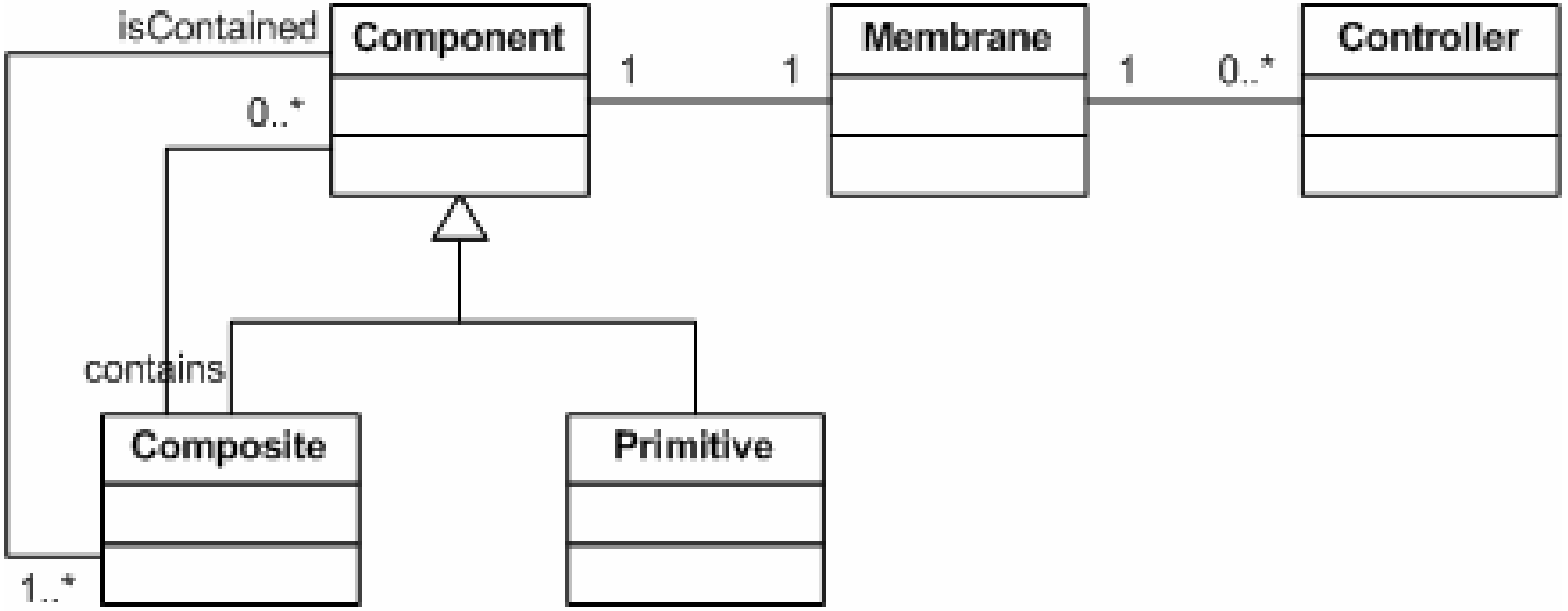
2 dimensions

- métier
- contrôle
  - les propriétés extra-fonctionnelles
  - mise en œuvre par une membrane
  - composée d'un ensemble de contrôleurs
    par ex. : sécurité, transaction, persistance, arrêt/démarrage, nommage
  - contrôleur accessible par une **interface** dite **de contrôle**

  - contrôleurs et membranes : ensembles ouverts
  - rôle d'un *framework* Fractal (Julia, AOKell, …)
    fournir un cadre pour développer
    - des applications Fractal
    - des contrôleurs et des membranes

2.1.1 Composant

En résumé



2.1.2 Interface

- point d'accès à un composant

- émettre / recevoir des invocations d'opérations
- interface typée
- pas de distinction fondamentale entre interfaces métier et de contrôle
  - elles se manipulent de façon identique
- +sieurs interfaces possibles par composant

2.1.2 Interface

Définit des services offerts ou **requis** par un composant

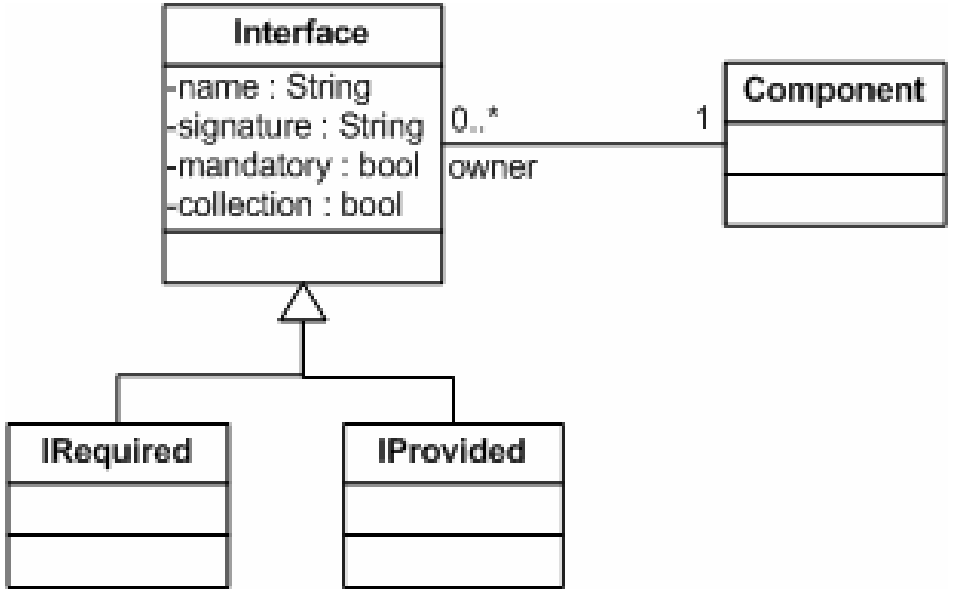
rq : requis n'existe pas en Java

(cf. extension Java : Traits [Schärli 2003])

Interface Fractal

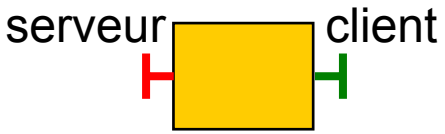
- nom
- signature

- est associée à un composant
- cliente ou serveur
- obligatoire ou facultative
- simple ou multiple (collection)



2.1.2 Interface

Interface : **client** vs **serveur**



- serveur : services fournis
- client : services requis

But interface client

- spécifier les **services nécessaires** au fonctionnement du composant

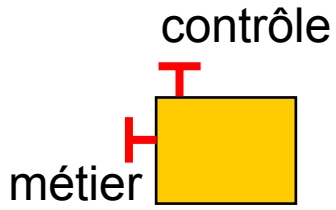
Convention graphique

- client : côté gauche des composants
- serveur : côté droit des composants

2.1.2 Interface

Interface : **métier** vs **contrôle**

- métier : ce pour quoi l'application est faite (sa finalité 1ère)
- contrôle : tout ce qui ne relève pas de sa finalité 1ère



Notion subjective

- dépend du domaine applicatif
- dépend des développeurs
- dépend de l'évolution des applications
- dépend de la granularité (voir les systèmes en couches)

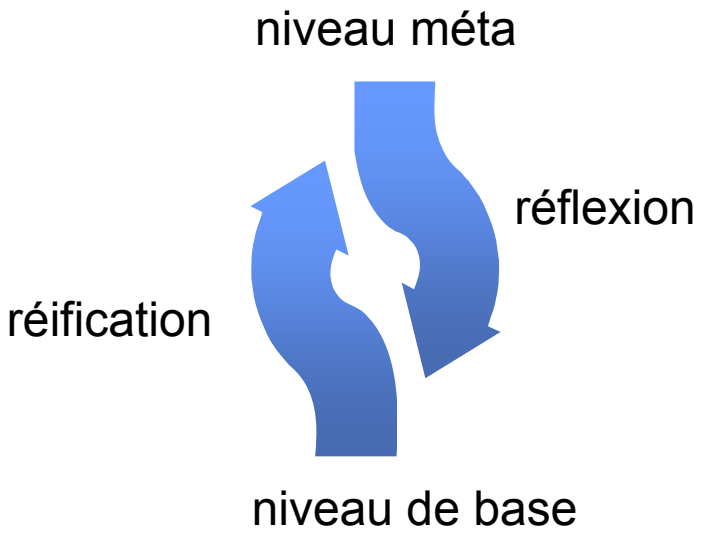
Contrôle

souvent services système (sécurité, persistance, réplication, tolérance aux pannes)
mais pas uniquement : contrats (pre/post), intégrité de données, règles de gestion, tracabilité, gestion de *workflow*, …

2.1.2 Interface

Interface **contrôle** ≈ niveau méta

- niveau de base
  - ⇨ l'application
- niveau méta
  - ⇨ réifie la base
  - ⇨ l'instrospecte
  - ⇨ la modifie
  - ⇨ l' "ouvre"



Interface : **métier/contrôle** vs **client/serveur**

- métier client/serveur : ok
- contrôle
  - contrôle "serveur" : ok
  - contrôle client : ???? (question ouverte)

2.1.2 Interface

Interfaces contrôle **prédéfinies** dans Fractal

Convention de nommage : suffixe -*controller*

Pour les composants primitifs

- binding-controller — gestion d'une liaison (créer/supprimer)
- lifecycle-controller — démarrer/arrêter le composant
- name-controller — nommer du composant
- super-controller — le composite auquel ∈ le composant

\+ pour les composants composites

- content-controller — gérer le contenu d'un composite (ajouter/retirer des sous-composants)

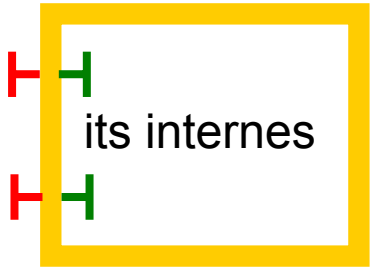
2.1.2 Interface

Interface : **obligatoire** vs **facultative**

- combinaison avec métier/contrôle et client/serveur ok

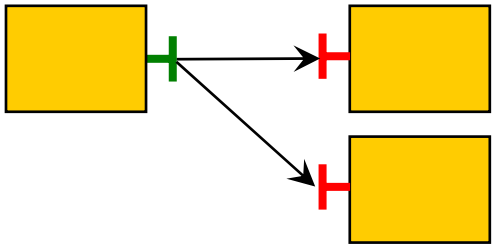
Interface : **interne** vs **externe**

- externe : en façade d'un composant ou d'un composite
- interne : pendant d'une itf externe pour un composite
- combinaison avec métier /contrôle ok
- combinaison avec client/serveur ok



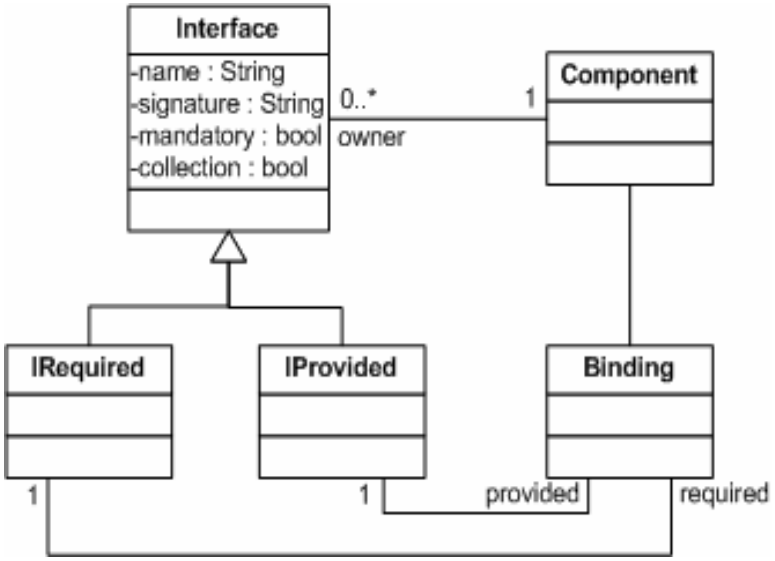
Interface : **simple** vs **multiple** (collection)

- cardinalité de la liaison
- référence ou collection de références
- combinaison avec métier seulement
- combinaison avec client/serveur ok



2.1.3 Liaison

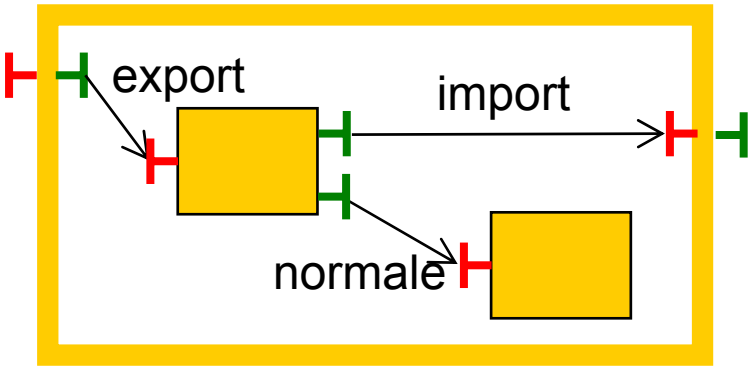
- chemin de communication entre composants
  - + précisemment : entre 1 interface client et 1 interface serveur
- explicite et matérialise les dépendances entre composants
- manipulable à l'exécution
  - reconfiguration dynamique
- sémantique non figée
  - par ex. : invocation méthode locale, distante, diffusion, avec QoS, …
  - dépend du *binding-controller*



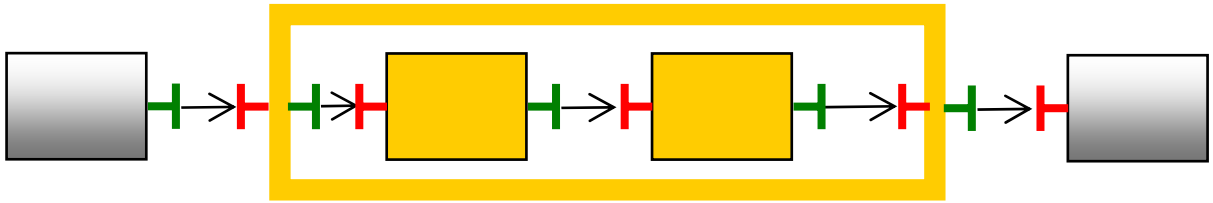
2.1.3 Liaison

2 formes

- primitive
  - au sein d'un même espace d'adressage
  - invocation méthode locale



- composite
  - ≠ sémantiques d'invocation possible
  - par ex. invocation distante



Plan

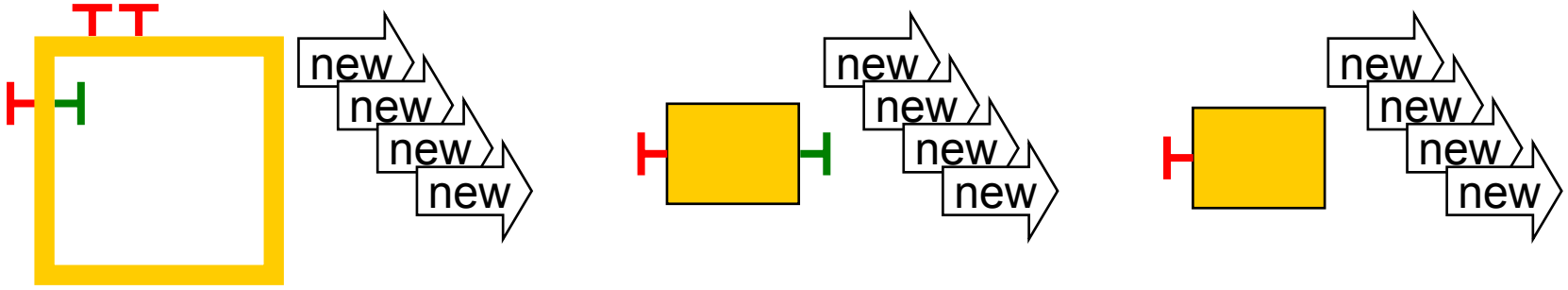
2.2.1 Instanciation

- créer une instance à partir d'une définition de composant
- ≡ *new* en POO
- 2 solutions
  - avec une fabrique (aka *Bootstrap Component*)
  - à partir d'un gabarit (*template*)

2.2.1 Instanciation

Instanciation avec le *Bootstrap Component*

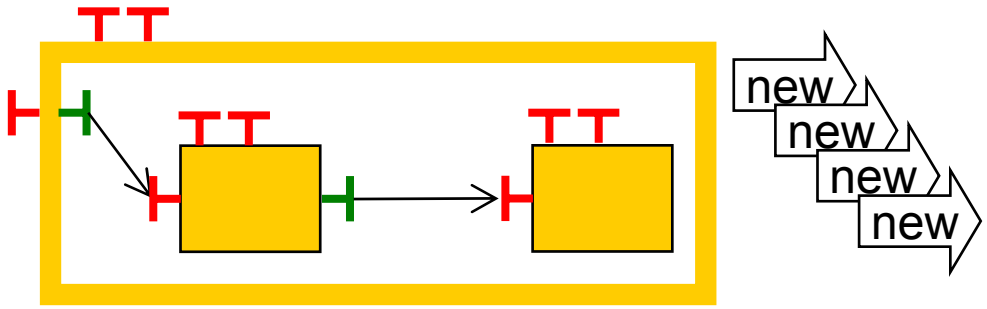
- composant prédéfini par Fractal
- peut instancier n'importe quel composant
- informations nécessaires
  - son type
  - son implémentation
  - une forme de membrane



2.2.1 Instanciation

Instanciation à partir d'un gabarit

- utiliser l'existant (*template*)
  - 1 composant ou
  - 1 assemblage de +sieurs composants
- pour en faire une « copie »



- Avantage
  - moins de manipulation pour instancier de grands assemblages

2.2.2 Typage

Principe de substitution

- relation de sous-typage notée ≤ (T1 ≤ T2 : T1 sous-type de T2)
- composants vus comme des **boîtes noires** (en faisant abstraction implémentation)
- type d'un composant : construit à partir du type de ses interfaces

Ti = < Ci , Si > // Ci : ensemble d'interfaces client – Si : serveur

$T1 \leq T2 \equiv \forall\, c1 \in C1, \exists\, c2 \in C2, c1 \leq c2 \wedge$
$\forall\, s2 \in S2, \exists\, s1 \in S1, s1 \leq s2$

$s1 \leq s2 \equiv s1.name = s2.name \wedge s1.signature \leq s2.signature \wedge$
(s2 obligatoire ⇨ s1 obligatoire) ∧
(s2 collection ⇨ s1 collection)

$c1 \leq c2 \equiv c1.name = c2.name \wedge c2.signature \leq c1.signature \wedge$
(c2 optionnelle ⇨ c1 optionnelle) ∧
(c2 collection ⇨ c1 collection)

2.2.3 Niveaux de conformance

- modèle de composant ouvert
- rien n'est figé
  - système de type
  - sémantique des communications
  - sémantique des composants

- supporte implémentations +/- complète API Fractal
- niveaux de conformance

2.2.3 Niveaux de conformance

	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

Plan

1. Introduction
2. Le modèle Fractal
3. Développer avec Fractal
   - 3.1 Fraclet
   - 3.2 Fractal ADL
   - 3.3 Fractal API
   - 3.4 Autres outils
4. Plates-formes
5. Autre personnalité Fractalienne
6. Conclusion

3. Développer avec Fractal

Développer des applications Fractal en Java

3 outils complémentaires

- Fraclet
  - modèle de programmation à base d'annotations
- Fractal ADL
  - langage de description d'architecture (ADL) basé XML
- Fractal API
  - ensemble d'interfaces Java pour
    - l'introspection
    - la reconfiguration
    - la création/modification dynamique

    de composants et d'assemblage de composants Fractal

3.1 Fraclet

Modèle de programmation Java pour Fractal

- à base d'annotations
  - Java 5 ou
  - Java 1.4 XDoclet
- annotations d'éléments de code (classe, méthode, attribut)
  - apporte aux éléments une signification en lien avec les concepts Fractal
- phase de pré-compilation
  - génération du code source associé aux annotations
- indépendant des plates-formes (Julia, AOKell)

3.1 Fraclet

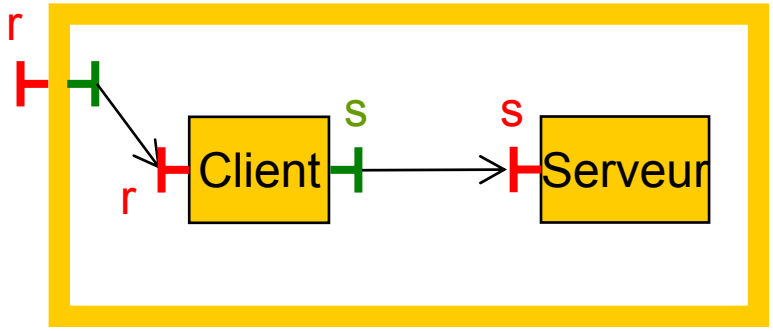
Principales annotations

- @Component
  - s'applique à une classe implémentant un composant
  - 2 attributs optionnels
    - name : le nom du composant
    - provides : les services fournis par le composant
- @Requires
  - s'applique à un attribut (*field*) correspondant à la référence du service requis
    - de type T pour SINGLETON
    - de type Map<String,T> pour COLLECTION
  - indique que l'attribut correspond à une interface cliente
  - 3 attributs optionnels
    - name : le nom de l'interface
    - cardinality : SINGLETON (par défaut) ou COLLECTION
    - contingency : MANDATORY (par défaut) ou OPTIONAL

3.1 Fraclet

Exemple Hello World

- un composant composite racine
- un sous-composant Serveur fournissant une interface
  - de nom s
  - de signature `interface Service { void print(String msg); }`
- un sous-composant Client fournissant une interface
  - de nom r
  - de signature `java.lang.Runnable` (convention *de facto* Fractal)
  - exportée au niveau du composite
- Client requiert le service fournit par l'interface s de Serveur



3.1 Fraclet

Exemple Hello World – Le composant Serveur

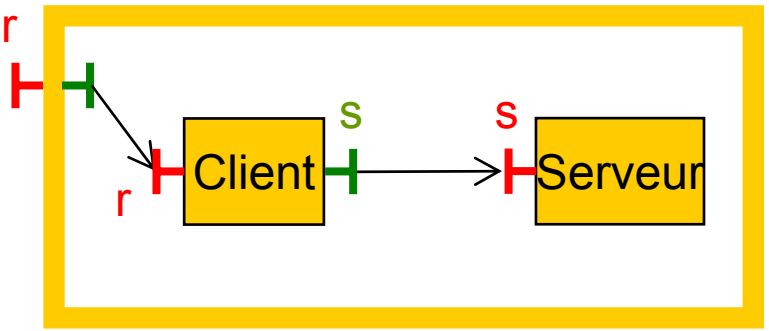
```
@Component(
   provides=
      @Interface(name="s",signature=Service.class) )
public class ServeurImpl implements Service {
   public void print( String msg ) {
      System.out.println(msg);
   }
}
```

3.1 Fraclet

Exemple Hello World – Le composant Client

```
@Component(
   provides=
      @Interface(name="r",signature=Runnable.class) )
public class ClientImpl implements Runnable {

   @Requires(name="s")
   private Service service;

   public void run() {
      service.print("Hello world!");
} }
```

Exemple Hello World – L'assemblage




```
<definition name="HelloWorld">

  <interface name="r" role="server"
             signature="java.lang.Runnable" />

  <component name="client" definition="ClientImpl" />

  <component name="serveur" definition="ServeurImpl" />

  <binding client="this.r" server="client.r" />
  <binding client="client.s" server="serveur.s" />

</definition>
```

3.1 Fraclet

Résumé

- écriture du code d'implémentation
- annotation du code pour ajouter les métainformations Fractal
- écriture des assemblages avec Fractal ADL

- pré-compilation Fraclet
  - génération de code Java et Fractal ADL supplémentaire
- lancement de l'application

3.1 Fraclet

Autres annotations Fraclet

- @Interface : interface serveur Fractal
- @Attribute : attribut d'un composant
- @Lifecycle : gestionnaire d'événements de cycle de vie
- @Controller : injection de référence vers une interface de contrôle

- @Legacy : extension de composants patrimoniaux
- @Membrane : forme de membrane
- @Node : définition d'un nœud virtuel pour les communications distantes

voir http://fractal.ow2.org/fraclet pour plus de détails

Plan

3.2 Fractal ADL

Langage (XML) pour la définition d'**architectures** de composants Fractal

- DTD de base pour la définition
  - interfaces
  - composants
  - liaisons

- langage extensible
  - définition de nouvelles balises
    (ex. : balise pour indiquer site de déploiement, …)

- *front-end* pour l'API
  génération d'appels à l'API pour construire l'architecture décrite en XML
- description de l'**architecture initiale**
  elle peut toujours **évoluer** par manipulation avec l'**API**

3.2 Fractal ADL

Fichier XML avec extension `.fractal`

Balise `<definition>` définit un composite racine contenant

- 0 ou n `<interface>`
- 0 ou n `<component>` (primitif ou composite inclus dans le composite racine)
  composant défini
  - directement dans le fichier (*inline*)
  - dans un fichier `.fractal` externe
- 0 ou n `<binding>`
  entre les interfaces du composite ou des sous-composants

3.2 Fractal ADL

Définition d'interface

<interface

name = "r"	nom de l'interface
role = "server"	server ou client
signature = "java.lang.Runnable"	signature Java de l'interface
cardinality = "singleton"	singleton (défaut) ou collection
contingency = "mandatory"	mandatory (défaut) ou optional

/>

```
<!ELEMENT interface EMPTY >
<!ATTLIST interface
  name CDATA #REQUIRED
  role (client | server) #IMPLIED
  signature CDATA #IMPLIED
  contingency (mandatory | optional) #IMPLIED
  cardinality (singleton | collection) #IMPLIED >
```

3.2 Fractal ADL

Définition de composant

<component name = "MonComp" >
< … interface, component, binding … >
<content class = "ClientImpl" /> classe Java implémentation
<controller desc = "primitive" /> type de membrane
</component> (primitive, composite, …)

<component name = "MonComp" definition = "Ma.Def" />
définition externe dans fichier Ma/Def.fractal

```
<!ELEMENT component (interface*,component*,binding*,content?,
attributes?,controller?,template-controller?) >
<!ATTLIST component
  name CDATA #REQUIRED
  definition CDATA #IMPLIED
>
```

3.2 Fractal ADL

Définition de liaison

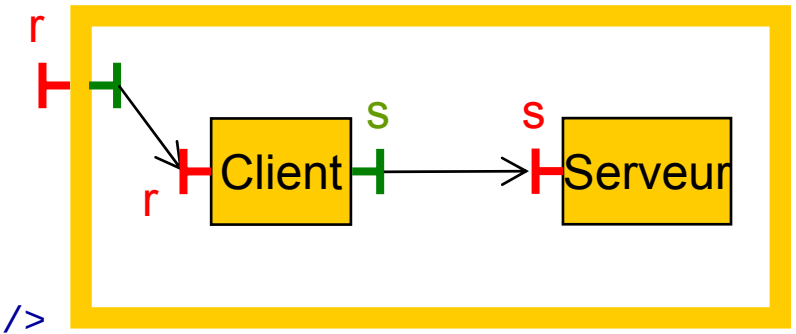
<binding
client = "this.r" interface source
server = "client.r" interface destination
/>

source | destination ::= nom composant . nom interface
this : composant courant

```
<!ELEMENT binding EMPTY >
<!ATTLIST binding
  client CDATA #REQUIRED
  server CDATA #REQUIRED
>
```

3.2 Fractal ADL

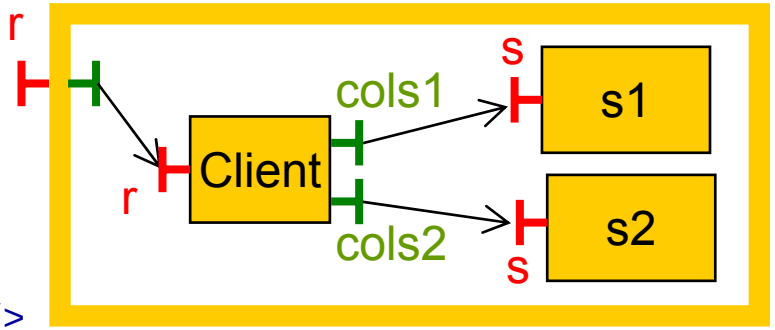
Exemple Hello World




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

  <binding client="this.r" server="client.r" />
  <binding client="client.s" server="serveur.s" />

</definition>
```

3.2 Fractal ADL

Exemple Hello World – Interfaces collection




```
<definition name="HelloWorld">
  <interface name="r" role="server"
             signature="java.lang.Runnable" />
  <component name="client" definition="ClientImpl">
    <interface name="r" role="server" signature="java.lang.Runnable" />
    <interface name="cols" role="client"
               cardinality="collection" signature="Service" />
    <content desc="ClientImpl" />
  </component>
  <component name="s1" definition="ServeurImpl"> ... </component>
  <component name="s2" definition="ServeurImpl"> ... </component>

  <binding client="this.r" server="client.r" />
  <binding client="client.cols1" server="s1.s" />
  <binding client="client.cols2" server="s2.s" />

</definition>
```

3.2 Fractal ADL

Notions additionnelles

- attributs
- héritage de définitions
- partage de composant
- *templates*
- configuration

3.2 Fractal ADL

Notions additionnelles – Attributs

- composants peuvent exporter une interface attribute-controller
  - setter/getter pour des propriétés exportées par le composant
- définition des valeurs initiales de ces propriétés

```
<definition name="Server">
  <attributes signature="ServiceAttributes">
    <attribute name="header" value="->"/>
    <attribute name="count" value="1"/>
  </attributes>
  <content class="ServerImpl"/>
</definition>
```

```
<!ELEMENT attributes (attribute*) >
<!ATTLIST attributes
  signature CDATA #IMPLIED
>
```

```
<!ELEMENT attribute EMPTY >
<!ATTLIST attribute
  name CDATA #REQUIRED
  value CDATA #REQUIRED
>
```

3.2 Fractal ADL

Notions additionnelles - Héritage de définitions

• réutiliser et étendre des définitions précédemment écrites

Bonne pratique d'écriture des architectures Fractal

• séparer la déf. du type de composant, de la déf. de son implémentation

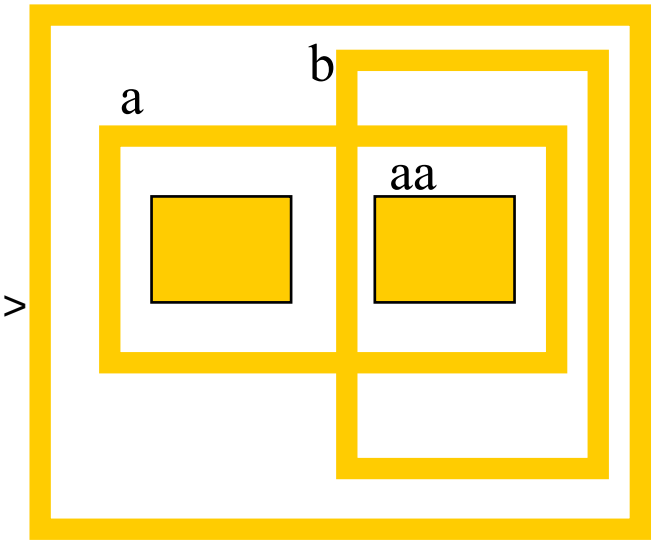
```
<definition name="ClientType">
  <interface name="r" role="server" signature="Main" />
  <interface name="s" role="client" signature="Service" />
</definition>

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

3.2 Fractal ADL

Notions additionnelles - Partage de composants

- nommage de composants avec / selon la hiérarchie d'imbrication `foo/bar/bob`
- 1 définition du composant partagé, les suivantes font référence à la 1ère

```
<definition name="foo">
  <interface name="r" role="server" signature="Main" />
  <component name="a">
    <component ... />
    <component name="aa"> ... </component>
  </component>
  <component name="b">
    <component name="aa" definition="a/aa" />
  </component>
</definition>
```



3.2 Fractal ADL

Notions additionnelles - *Templates*

```
<definition name="Client">
  <interface name="r" role="server" signature="Main" />
  <interface name="s" role="client" signature="Service" />
  <content class="ClientImpl" />
  <template-controller desc="primitiveTemplate" />
</definition>
```

Pour les composites `compositeTemplate`

3.2 Fractal ADL

Notions additionnelles - Configuration

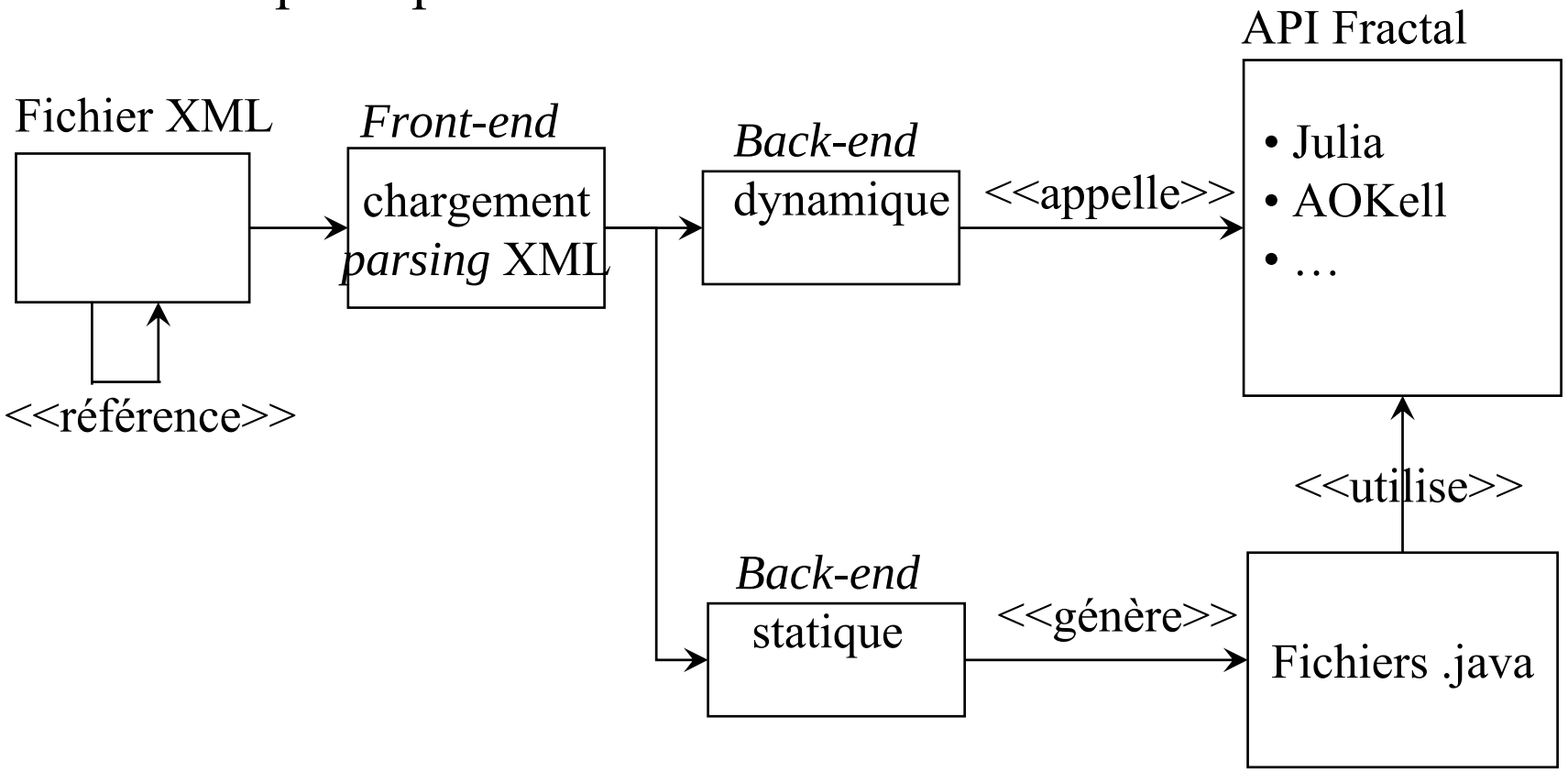
- arguments `${...}` utilisable dans une architecture
- valeurs fournies au moment de la création du composant

```
<definition name="Client" arguments="itfname,impl" >
  <interface name="r" role="server" signature="Main" />
  <interface name="s" role="client" signature="${itfname}" />
  <content class="${impl}" />
</definition>
```

```
Map context = new HashMap();
context.put("itfname","Service");
context.put("impl","ClientImpl");
Object o = f.new Component("HelloWorld",context);
```

3.2 Fractal ADL

Schéma de principe



Par défaut : *back-end* dynamique

Plan

3.3 Fractal API

- modèle dynamique
- les composants et les assemblages sont présents à l'exécution
- ➢ applications dynamiquement adaptables

Introspection et modification

- liaison : contrôleur de liaison (BC)
- composant
  - instropection
    - hiérarchie : contrôleur de contenu (CC) et accès au super (SC)
    - composant : accès aux interfaces et à leur type (*Component*)
  - modification
    - instanciation dynamique (*Bootstrap component* ou *template*)
    - hiérarchie : contrôleur de contenu
    - par défaut : pas de modification des composants existants
      mais : rien ne l'interdit (*Component* idoine à développer)
- ➢ API Fractal

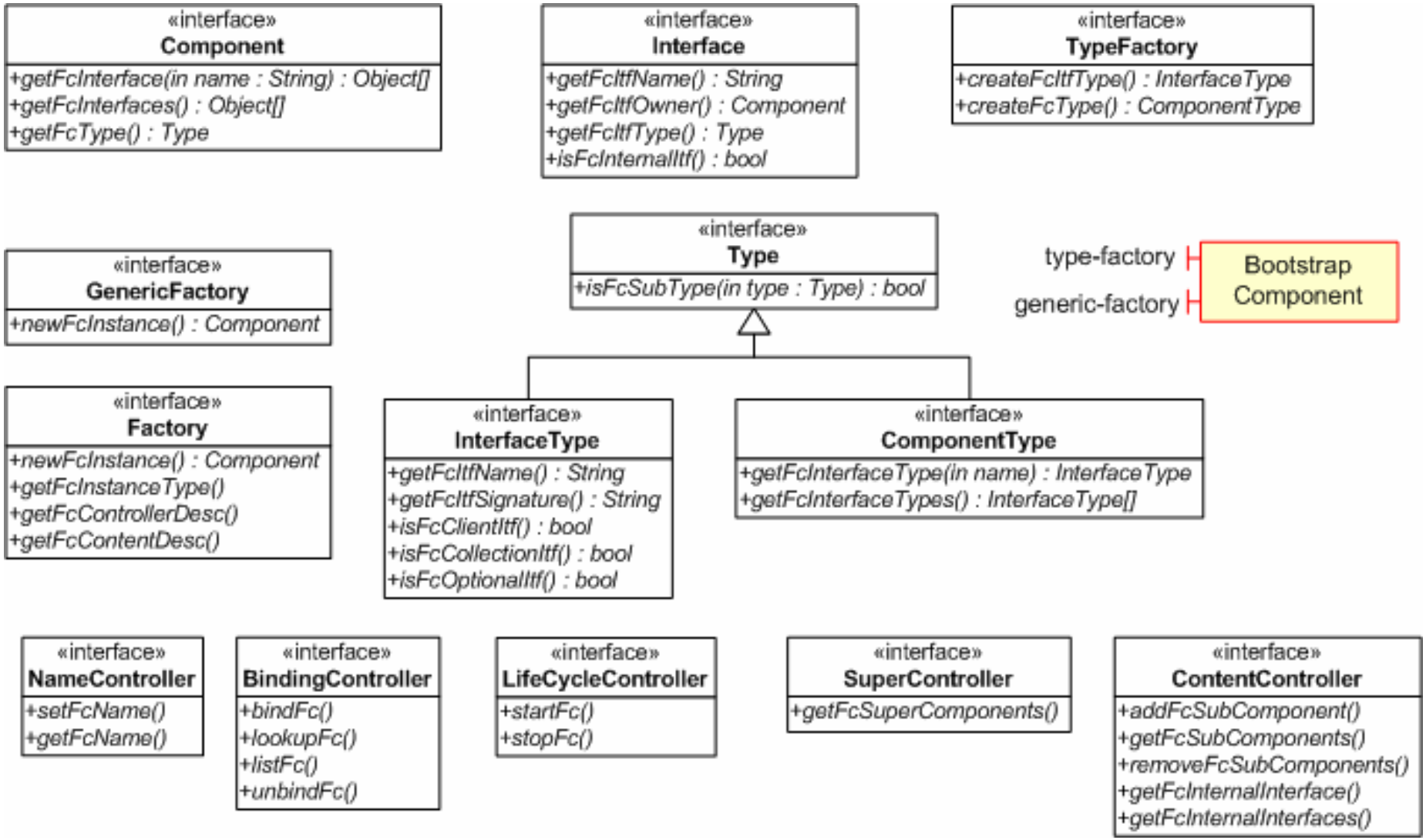
3.3 Fractal API

- légère (16 interfaces, <40 méthodes)
- API Fractal est la base de
  - Fractal ADL : *front-end* pour l'API
  - fraclet : génération de code utilisant l'API

Principe

1. Création des types de composants
2. Création des composants
3. Assemblage des composants
   1. Création des hiérarchies d'imbrication
   2. Création des liaisons
4. Démarrage de l'application

3.3 Fractal API



3.3 Fractal API

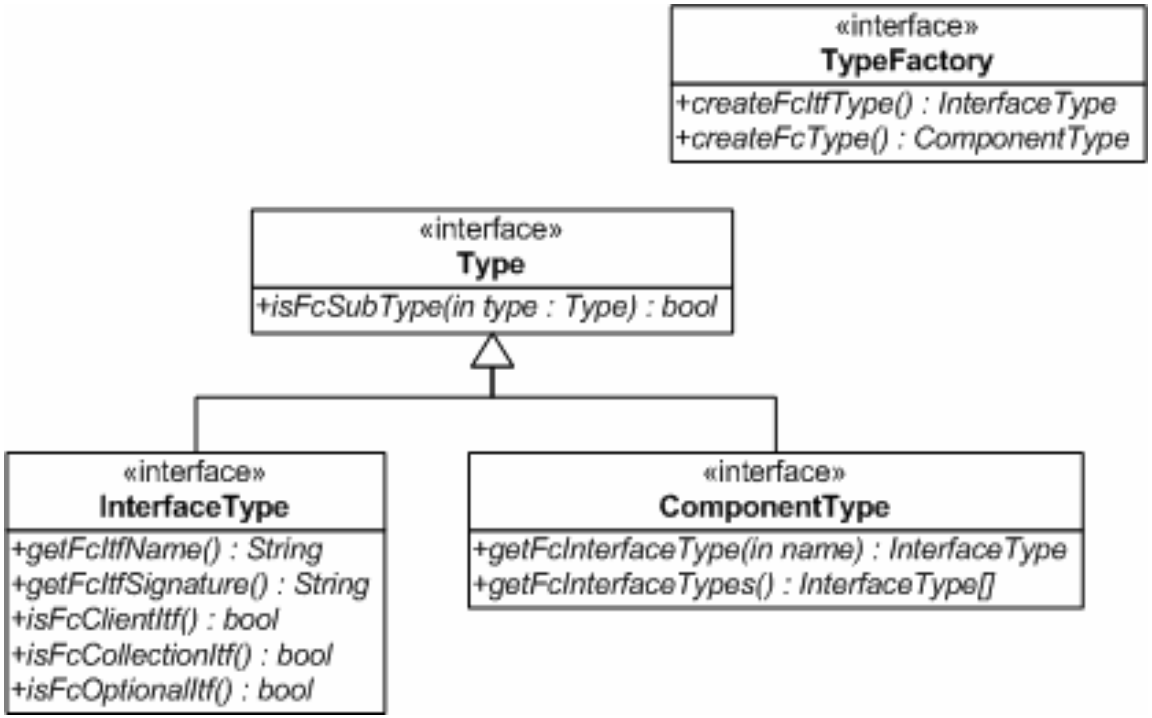
1. Création de types de composants

Type d'interface

- un nom
- une signature
  (interface Java)
- 1 bool : true = client
- 1 bool : true = optionnel
- 1 bool : true = multiple

Type de composant

• 1 ensembre de
  types d'interfaces



3.3 Fractal API

1. Création de types de composants

Récupération d'une instance de TypeFactory

```
Component boot = Fractal.getBootstrapComponent();
TypeFactory tf = Fractal.getTypeFactory(boot);
GenericFactory cf = Fractal.getGenericFactory(boot);
```

Création du type du composite racine

```
ComponentType rType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType(
     "r",                    // nom de l'interface
     "java.lang.Runnable",   // signature Java de l'interface
     false,                  // serveur
     false,                  // obligatoire
     false)                  // singleton
});
```

3.3 Fractal API

1. Création de types de composants

Création du type des types des composants primitifs Client et Serveur

```
ComponentType cType = tf.createFcType(new InterfaceType[] {
        tf.createFcItfType(
            "r", "java.lang.Runnable", false, false, false),
        tf.createFcItfType(
            "s", "Service", true, false, false)
});

ComponentType sType = tf.createFcType(new InterfaceType[] {
        tf.createFcItfType("s", "Service", false, false, false)
});
```

3.3 Fractal API

2. Création des (instances) composants

Instanciation du composite racine

```
Component rComp = cf.newFcInstance(
        rType,          // son type
        "composite",    // son type de membrane
        null);          // son implémentation (ici aucune)
```

Type de membrane

- composite
- primitive
- …

3.3 Fractal API

2. Création des (instances) composants

Instanciation des composants primitifs Client et Serveur

```
Component cComp = cf.newFcInstance(cType, "primitive", "ClientImpl");
Component sComp = cf.newFcInstance(sType, "primitive", "ServerImpl");
```

Classes ClientImpl et ServerImpl

```
public class ServerImpl implements Service {

  public void print(String msg) {
    System.out.println(msg);
  }
}
```

3.3 Fractal API

2. Création des (instances) composants

```
public class ClientImpl implements Runnable, BindingController {

  // Implémentation de l'interface métier Runnable
  public void run() {
    service.print("Hello world!");
  }

  // Implémentation de l'interface de contrôle BindingController
  public String[] listFc() { return new String[]{"s"};}
  public Object lookupFc(String cItf) {
    if (cItf.equals("s"))  return service;
    return null; }
  public void bindFc(String cItf, Object sItf)
    { if (cItf.equals("s"))  service = (Service)sItf; }
  public void unbindFc(String cItf)
    { if (cItf.equals("s"))  service = null; }

  private Service service;
}
```

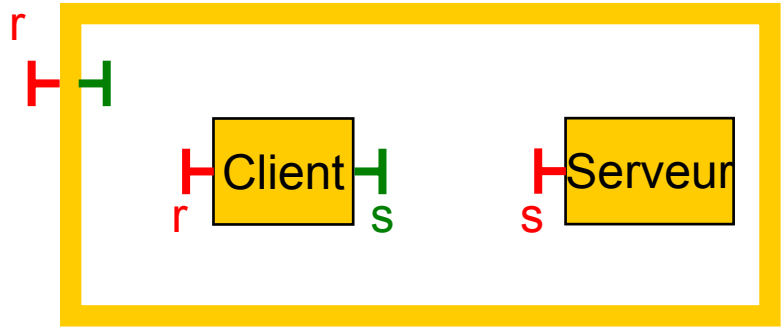
3.3 Fractal API

3. Assemblage des composants

3.1 Création des hiérarchies d'imbrication

Insertion des composants primitifs dans le composite

```
Fractal.getContentController(rComp).addFcSubComponent(cComp);
Fractal.getContentController(rComp).addFcSubComponent(sComp);
```



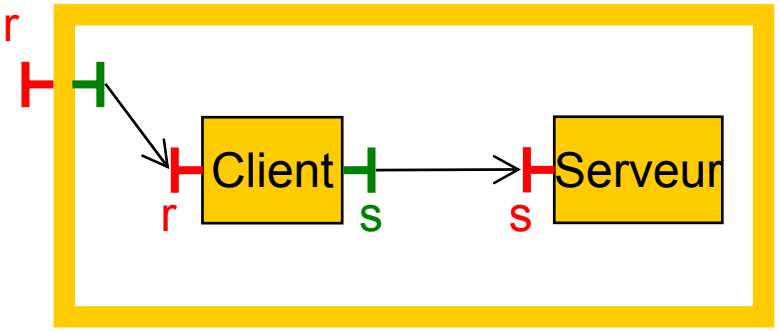
3.3 Fractal API

3. Assemblage des composants

3.2 Création des liaisons

Liaison entre les interfaces

```
Fractal.getBindingController(rComp).bindFc(
        "r", cComp.getFcInterface("r"));

Fractal.getBindingController(cComp).bindFc(
        "s", sComp.getFcInterface("s"));
```



3.3 Fractal API

4. Démarrage de l’application

Démarrage du composant et appel de la méthode run de l'interface r

```
Fractal.getLifeCycleController(rComp).startFc();
((Runnable)rComp.getFcInterface("r")).run();
```

3.3 Fractal API

- verbeux
- mais assez intuitif, pas compliqué
- simplifications importantes : Fractal ADL, fraclet

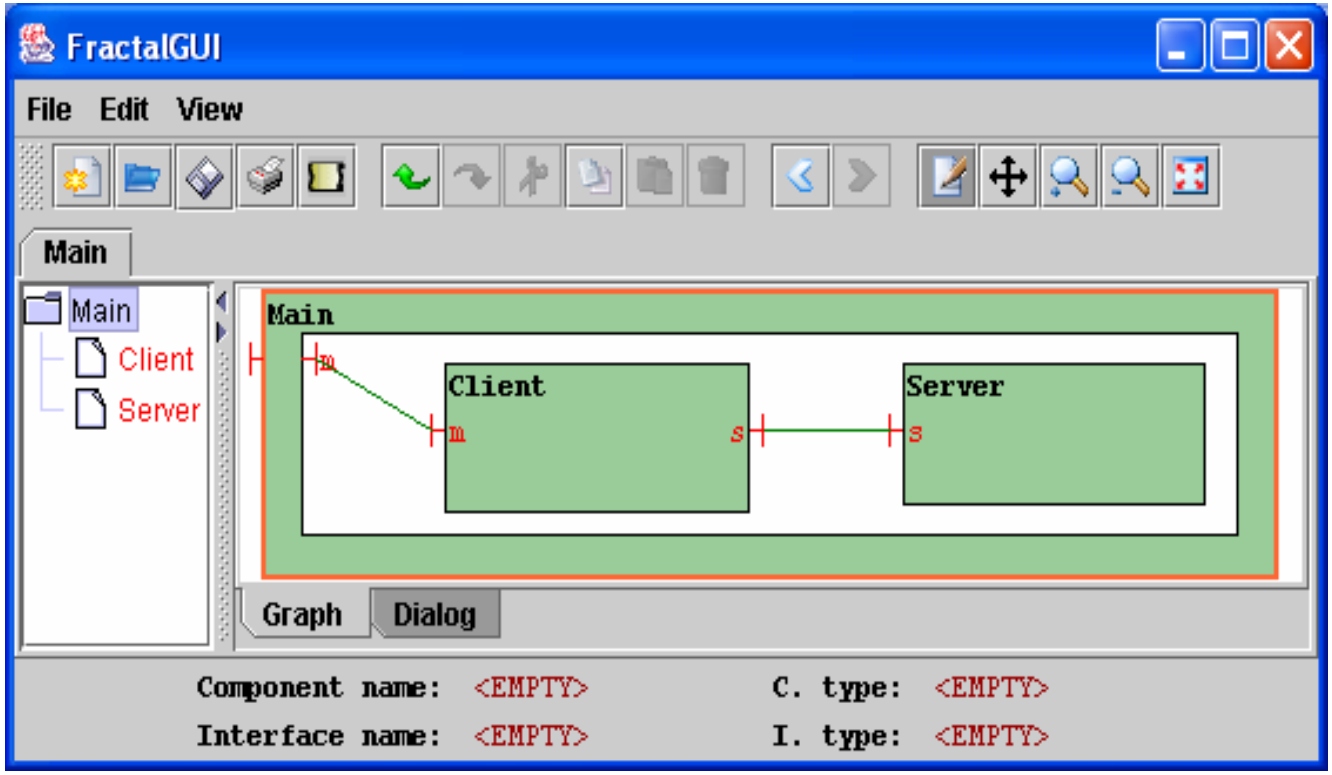
- parfois sous-spécifiée
  - ex. getFcInterface(), lookupFc() retournent Object (pas Interface)
  - la description des membranes et des implémentations sont des String
- ne pas imposer un modèle figé

Plan

3.4 Autres outils

FractalGUI

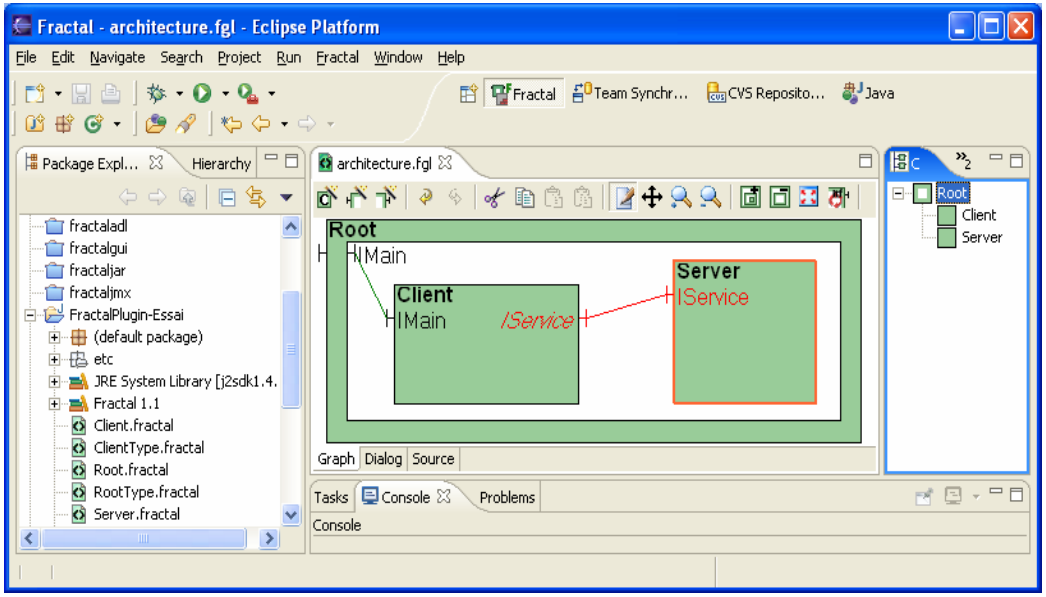
- outil de conception d'architectures Fractal
- génération de squelette de code



3.4 Autres outils

FractalGUI – Plugin Eclipse

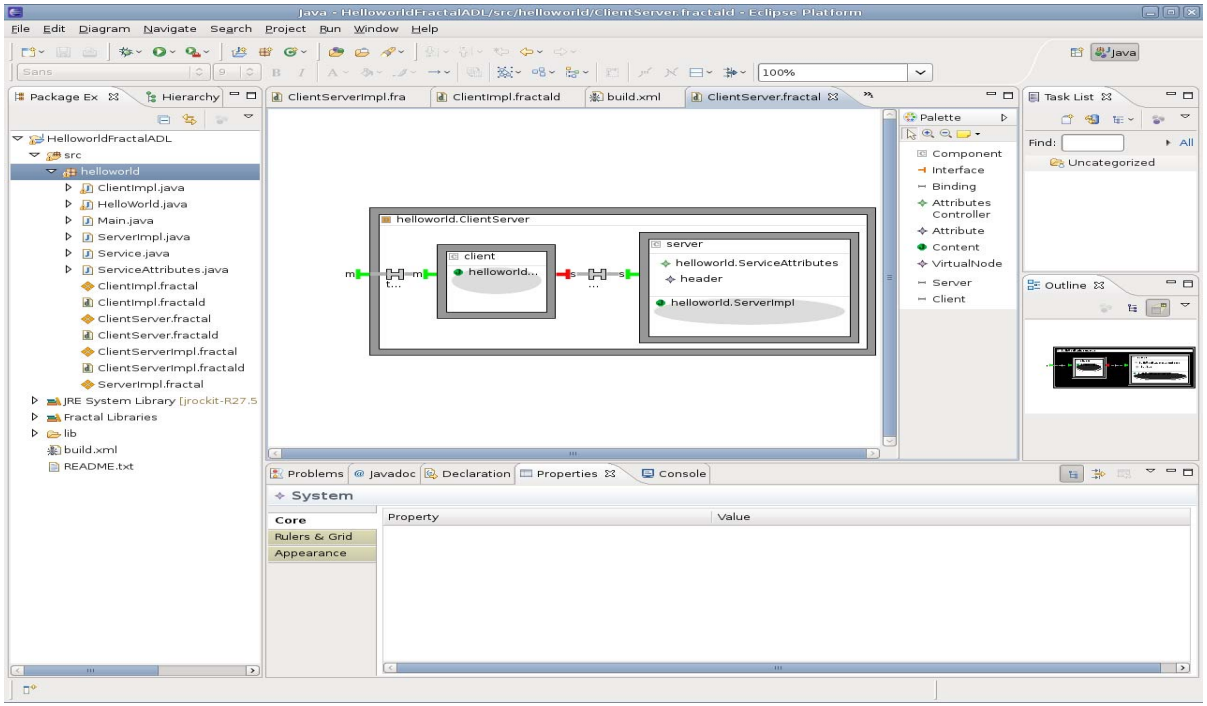
- intégration FractalGUI dans Eclipse
- auteurs
  - C. Boulet (ESSI Nice - 2004), P. Collet
  - O. Ghyselinck (INRIA Lille - 2005) : génération code Fractal 2.0



3.4 Autres outils

F4E – Fractal for Eclipse

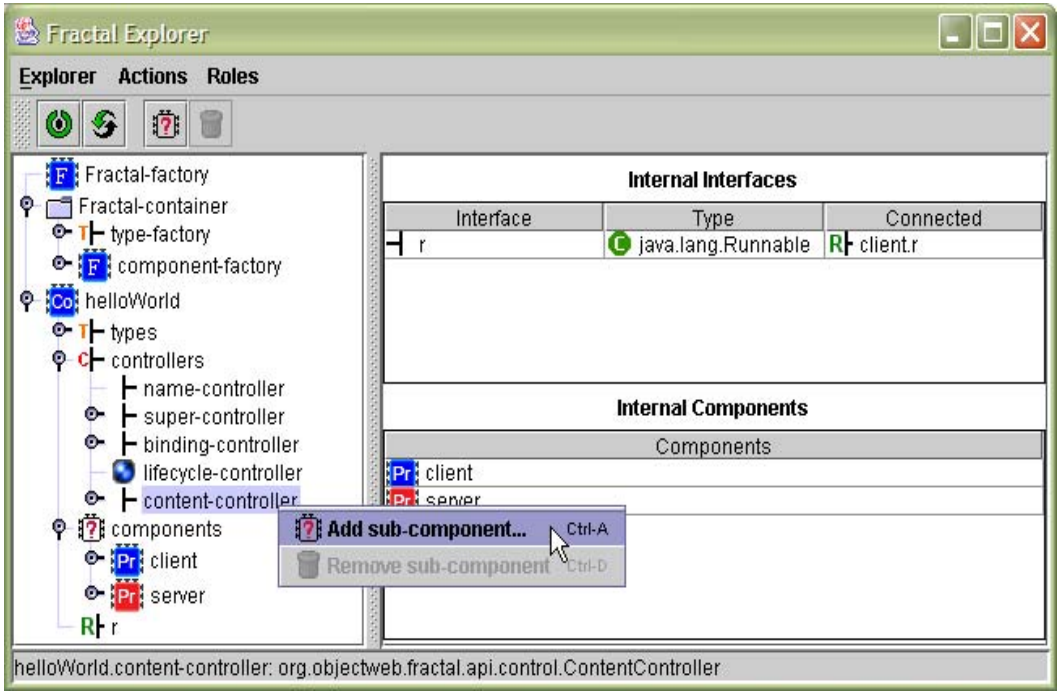
- environnement Eclipse de développement d'applications Fractal



3.4 Autres outils

FractalExplorer

- console d'administration
- pilotage (*run-time*) d'une application Fractal



Plan

1. Introduction
2. Le modèle Fractal
   - 2.1 Notions de base
   - 2.2 Notions avancées
3. Développer avec Fractal
4. Plates-formes
   - 3.1 Julia
   - 3.2 AOKell
5. Autre personnalité Fractalienne
6. Conclusion

4. Plates-formes

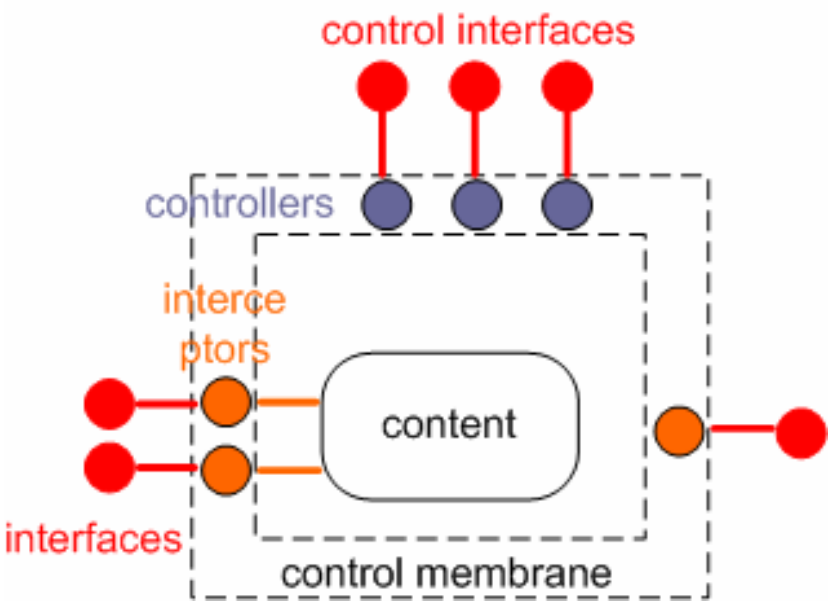
- plusieurs plates-formes
  - 3 en Java
    - Julia — implémentation de référence
    - AOKell — aspects + componentisation des membranes
    - ProActive — composants actifs pour les grilles
  - 2 en C (Think, Cecilia), 1 en C++ (Plasma), 1 en SmallTalk (FracTalk), 1 pour .NET (FractNet)
- ≠ implémentations pour ≠ besoins

4. Plates-formes

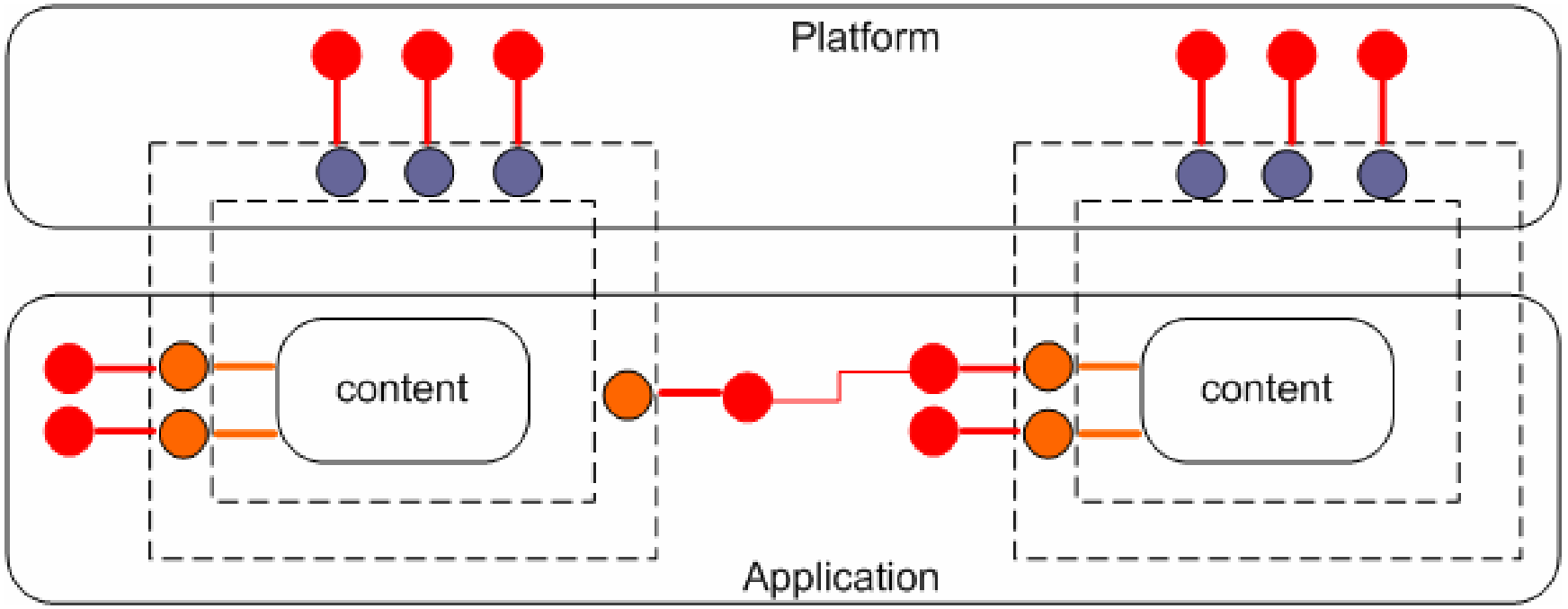
Concepts d'architecture des plates-formes Fractal

- chaque composant est « hébergé » par une membrane
  - fournit services extra-fonctionnels (liaison, nommage, transaction, sécurité, …)
  - ≡ conteneur EJB, CCM
- chaque membrane est composée d'un ensemble de contrôleurs
- un contrôleur
  - fournit un service extra-fonctionnel élémentaire
    - ajoute de nouvelles fonctionnalités
    - et/ou modifie les fonctionnalités existantes (intercepteur)
    - est accessible via une interface
- cœur métier du composant = contenu (content)

Représentation usuelle



Vue plate-forme



Plan

4.1 Julia

- implémentation de référence du modèle Fractal
- fractal.ow2.org/julia

- démontrer pertinence/faisabilité des spécifications
- framework extensible pour programmer des contrôleurs
- implémenter des objets de contrôle de façon à minimiser en priorité
  - le surcoût en temps d'exécution des composants
  - le surcoût en mémoire sur les applications

4.1 Julia

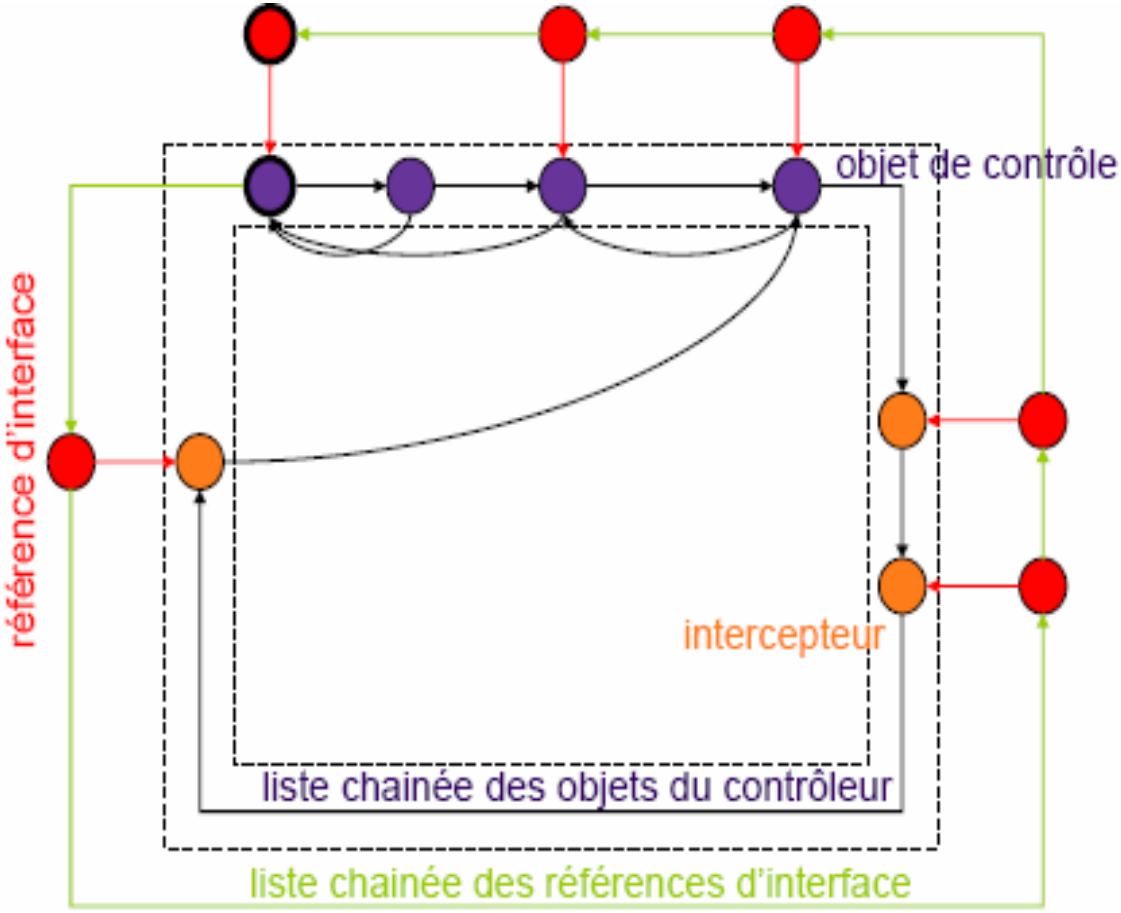
Organisation modulaire de la plate-forme

- fractal-api : API Fractal (partagée avec autres plates-formes)
- julia-runtime : API interne Julia
- julia-asm : *framework* de génération de *bytecode*
  - basé sur la librairie de manipulation de *bytecode* ASM
  - fournit un mécanisme de programmation par *mixins*

- julia-mixins
  - un ensemble de *mixins* mettant en œuvre la personnalité correspondant à la sémantique de référence du modèle Fractal
  - développement d'une nouvelle personnalité
    = développement (réutilisation, extension, …) d'un nouveau module julia-mixins

- (légère) collision autour du nom Julia
  - Julia le *framework* (julia-runtime + julia-asm)
  - Julia la personnalité correspondant à la sémantique de référence (julia-mixins)

Structures de données pour un composant Fractal/Julia



Configuration

Descripteurs de contrôle (type de membrane)

- primitive, composite, parametric…, template…
- 13 par défaut avec Julia

- liste ouverte : on peut en définir de nouveaux
- mécanisme de configuration
    - fichier `julia.cfg` de définition de descripteurs de contrôle
    - modifiable
        - nouveaux descripteurs / contrôleur
        - modifications descripteurs / contrôleur existant
        - optimisations
    - chargé / interprété dynamiquement au lancement

Configuration – Exemple

```
(primitive
  ('interface-class-generator
    ( 'component-itf
      'binding-controller-itf
      'super-controller-itf
      'lifecycle-controller-itf
      'name-controller-itf )
    ( 'component-impl
      'container-binding-controller-impl
      'super-controller-impl
      'lifecycle-controller-impl
      'name-controller-impl )
    ( (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
       org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator ) )
    org.objectweb.fractal.julia.asm.MergeClassGenerator
    'optimizationLevel
```

Interfaces de contrôle

Implémentations

Intercepteurs

Option d'optimisation

4.1 Julia

Génération des membranes de contrôle

Mécanisme de *mixin* [Bracha 90]
inspiré de JAM [`www.disi.unige.it/person/LagorioG/jam`]

- construction d'une classe en fusionnant des méthodes provenant de ≠ classes
- similitudes avec mécanisme de classes partielles dans C# v2.0

- chaque classe définit des méthodes ≠ ⇨ pas de pb pour fusionner
- méthodes avec même signature
    - mécanisme de chaînage pour concaténer le code des ≠ méthodes

Mixin Julia

- classe abstraite
- utilisant méthodes abstraites préfixées par
    - _super_XXX : appel méthode XXX "héritée" d'une autre classe
    - _this_XXX : appel méthode XXX définie dans une autre classe

4.1 Julia

Exemple mise en oeuvre Mixin Julia

```
abstract class M {
  abstract void _super_m();
  abstract void _this_n();
  public int count;
  public void m() {
    ++count;
    _this_n();
    _super_m();
} }
```

```
abstract class N {
  abstract void _super_m();
  public void m() {
    System.out.println("m called");
    _super_m();
  }
}
```

```
abstract class O {
  public void m() {System.out.println("m");}
  public void n() {System.out.println("n");} }
```

4.1 Julia

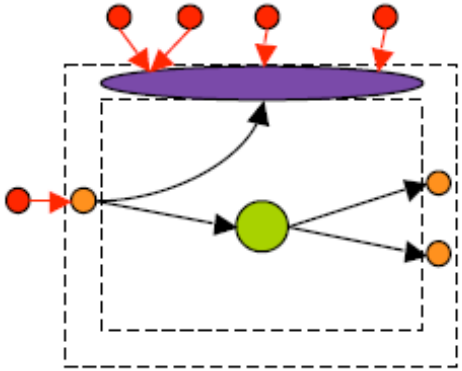
Exemple mise en oeuvre Mixin Julia

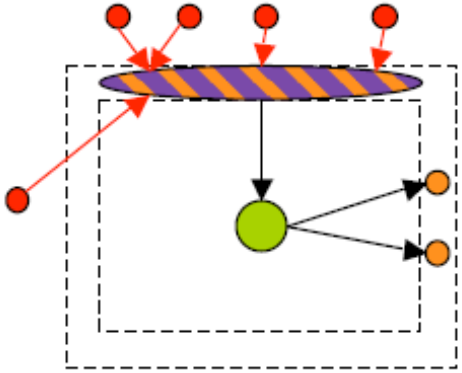
mixin O N M

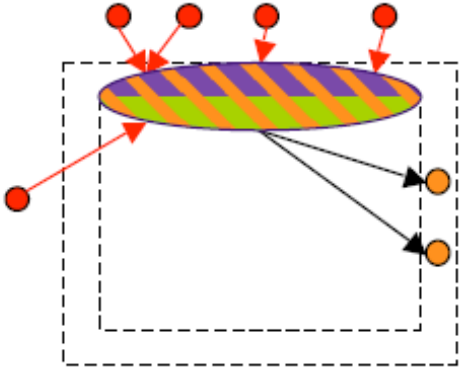
```
public class ONM {

 public void m () {
   ++count;
   n();
   m$0();
 }
 private void m$0 () {
   System.out.println("m called");
   m$1();
 }
 private void m$1 () {
   System.out.println("m");
 }
 public void n() {
   System.out.println("n called");
 }
 public int count;
}
```

Optimisation intra-composant

- fusion
  - des objets de contrôle (gauche)
  - + intercepteurs (milieu)
  - + implémentation composant (droite)
- mise en œuvre
  - manipulation de *bytecode* avec ASM
  - algorithme de *mixin*

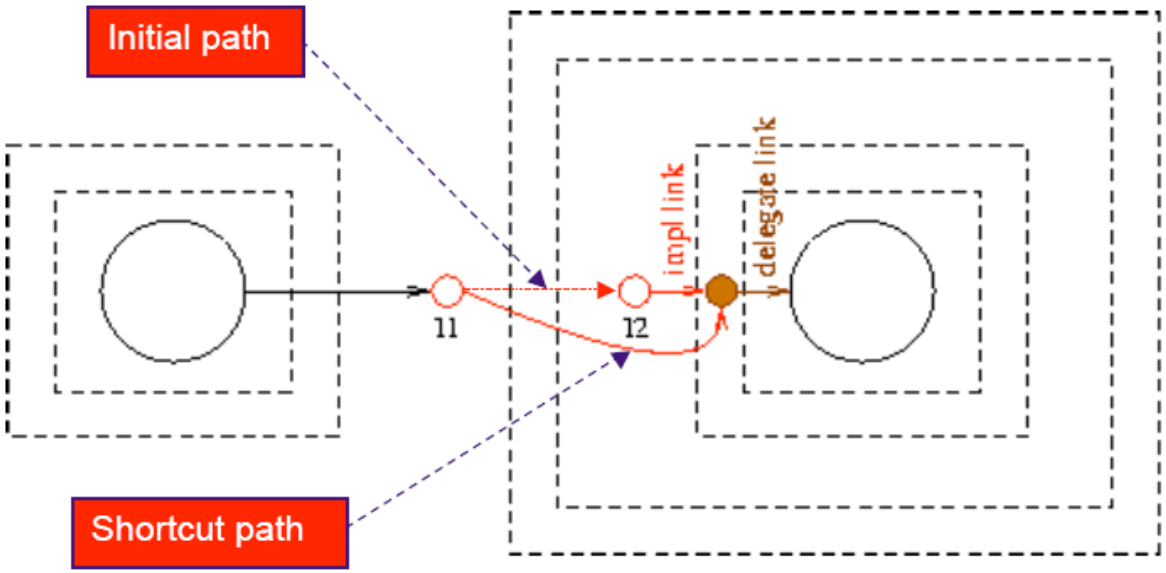






Optimisation inter-composants

- optimisation des liens entre composants
- remarques
  - peu poser pb si les composants échangent des références avec `this`
  - raccourcis doivent être recalculés à chaque changement de liaison



Plan

4.2 AOKell

- implementation of the Fractal Specifications
- joint work with France Telecom R&D (T. Coupaye)
- aspect-based framework for engineering the control level

Expected benefits

- easier to develop, debug, maintain new controllers
- better integration with IDEs
- reducing the development time for writing new controllers
- reducing the learning curve

This work is partially funded by France Telecom
under the external research contract #46131097

4.2 AOKell

2 main contributions

- aspect-oriented glue layer
  - 2 versions
    - AspectJ [Kiczales 01] http://www.eclipse.org/aspectj
    - Spoon [Pawlak 05]
      open-compiler for efficient transformations of Java programs
      http://spoon.gforge.inria.fr
- control level implementation
  - 2 versions: objects or component

4.2 AOKell

Requirements for controllers

- Feature injection (e.g. Binding controller interface)
- Interception (e.g. LifeCycle)

Our proposal

- 1 controller = 1 aspect
  - Feature injection = inter-type declaration (ITD)
  - Interception = code advising (before, after, around)

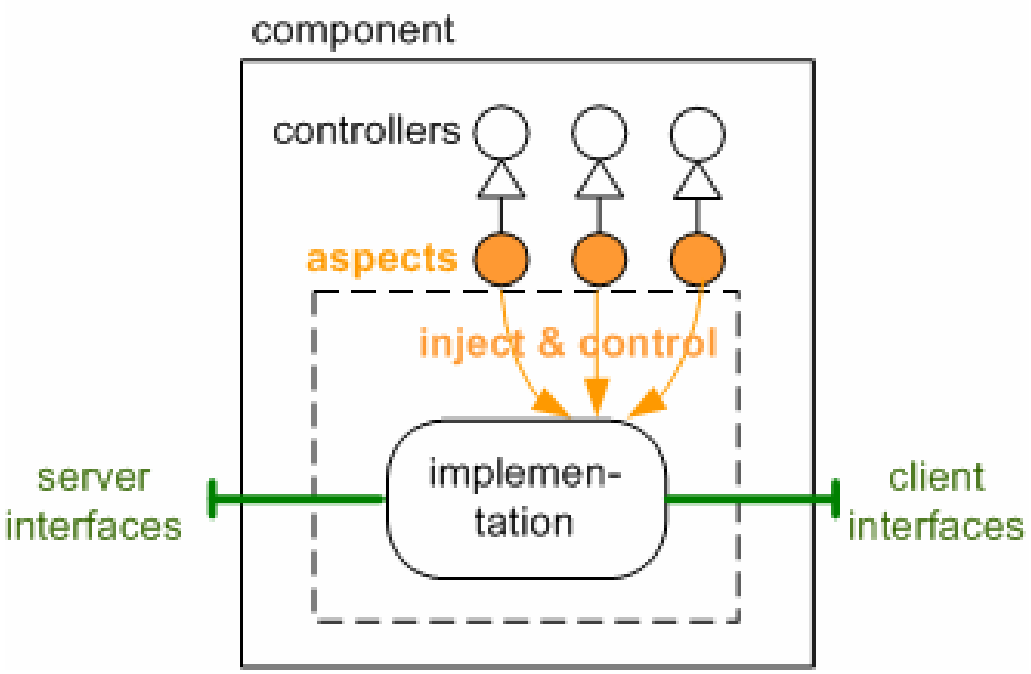
AspectJ

- « reference » aspect weaver
- Compile-time weaving (perf ++)
- Load-time weaving (and run-time in the near future)
- Tooling, IDE & debugger integration

4.2 AOKell

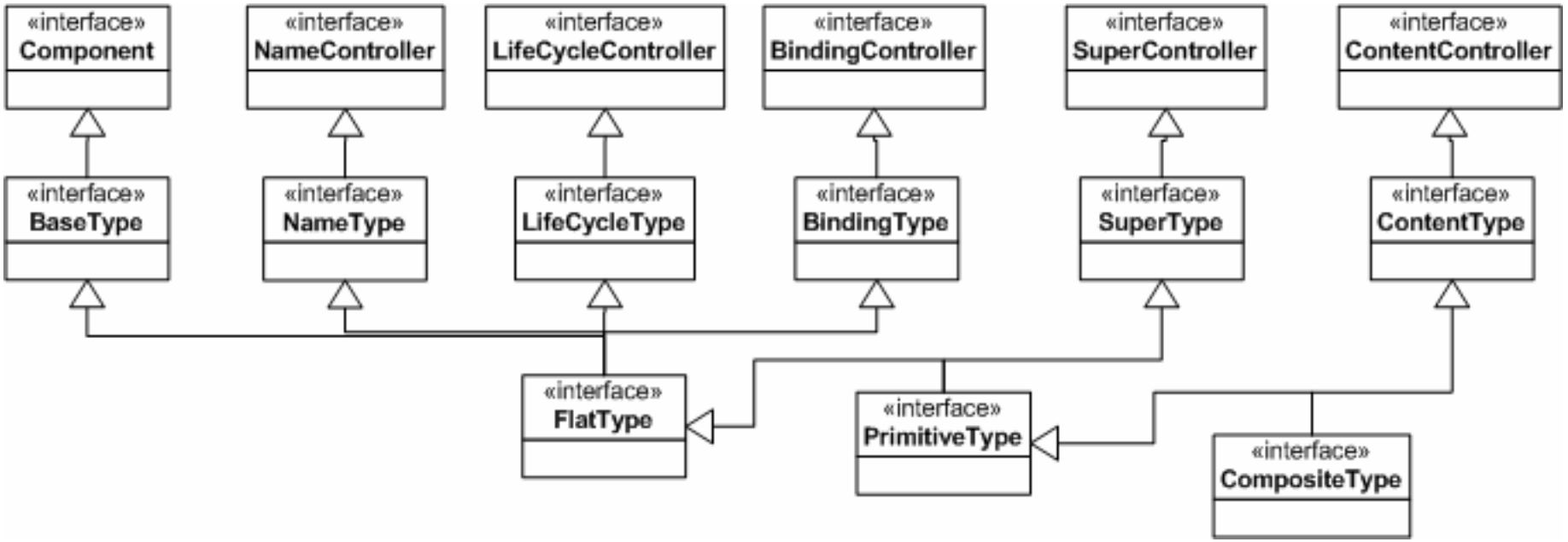
Aspect glue layer

- 1 aspect per controller type
- 7 aspects
  BC, LC, NC, CC, SC, Factory, Component



➢ Each aspect delegates the control logic to a j.l.Object

Controller « type system »



+ parametrics, templates and bootstrap

4.2 AOKell

```
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; }
}
```

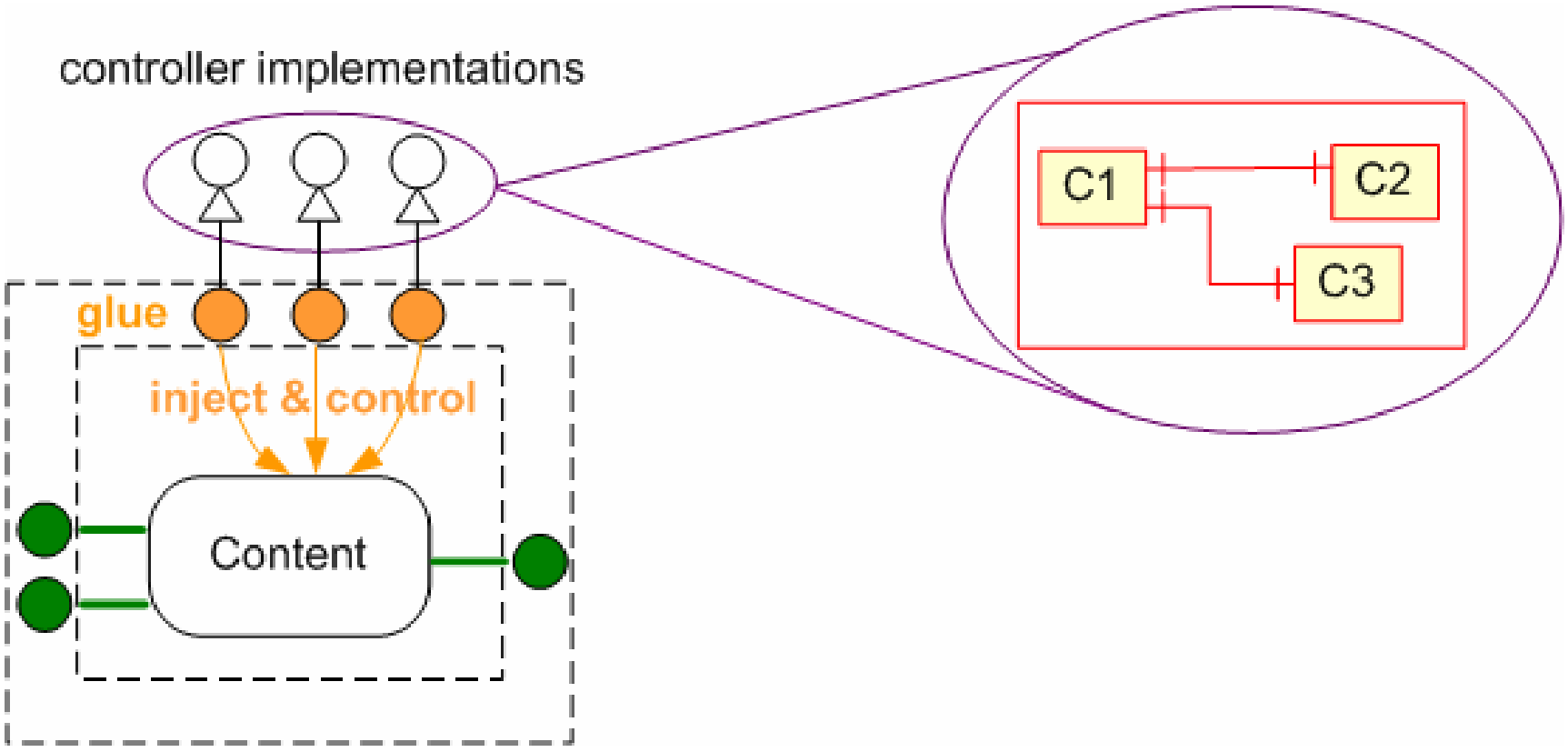
AspectJ
Inter-type declarations

Object implementation of the name controller

4.2 AOKell

```
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");
```

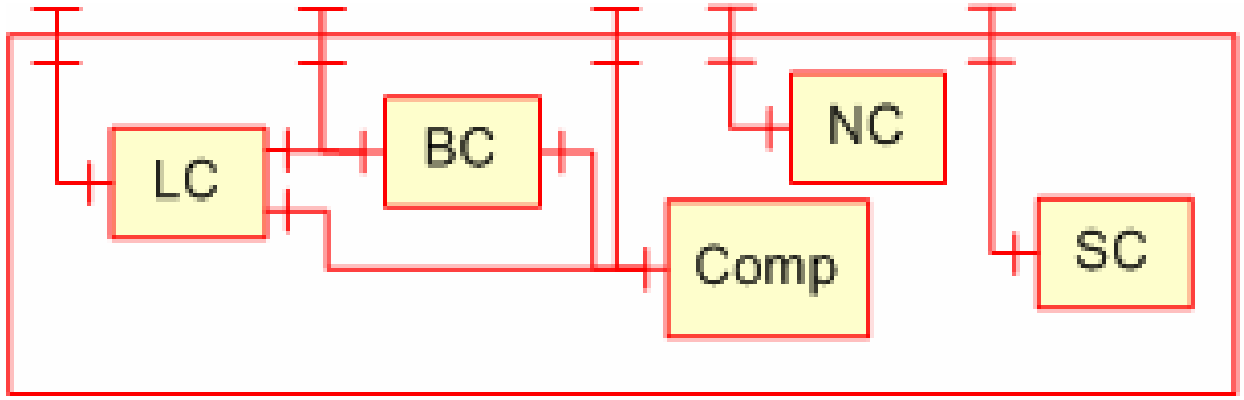
Componentizing the membrane



Componentizing the membrane

- notion of a (primitive) component-controller
  - a component implementing a control interface
    BC, LC, NC, Component, SC,CC, Factory
- membrane
  - an assembly of components-controllers
  - a composite component-controller
  - can benefit from the tools available with Fractal
    e.g. Fractal ADL

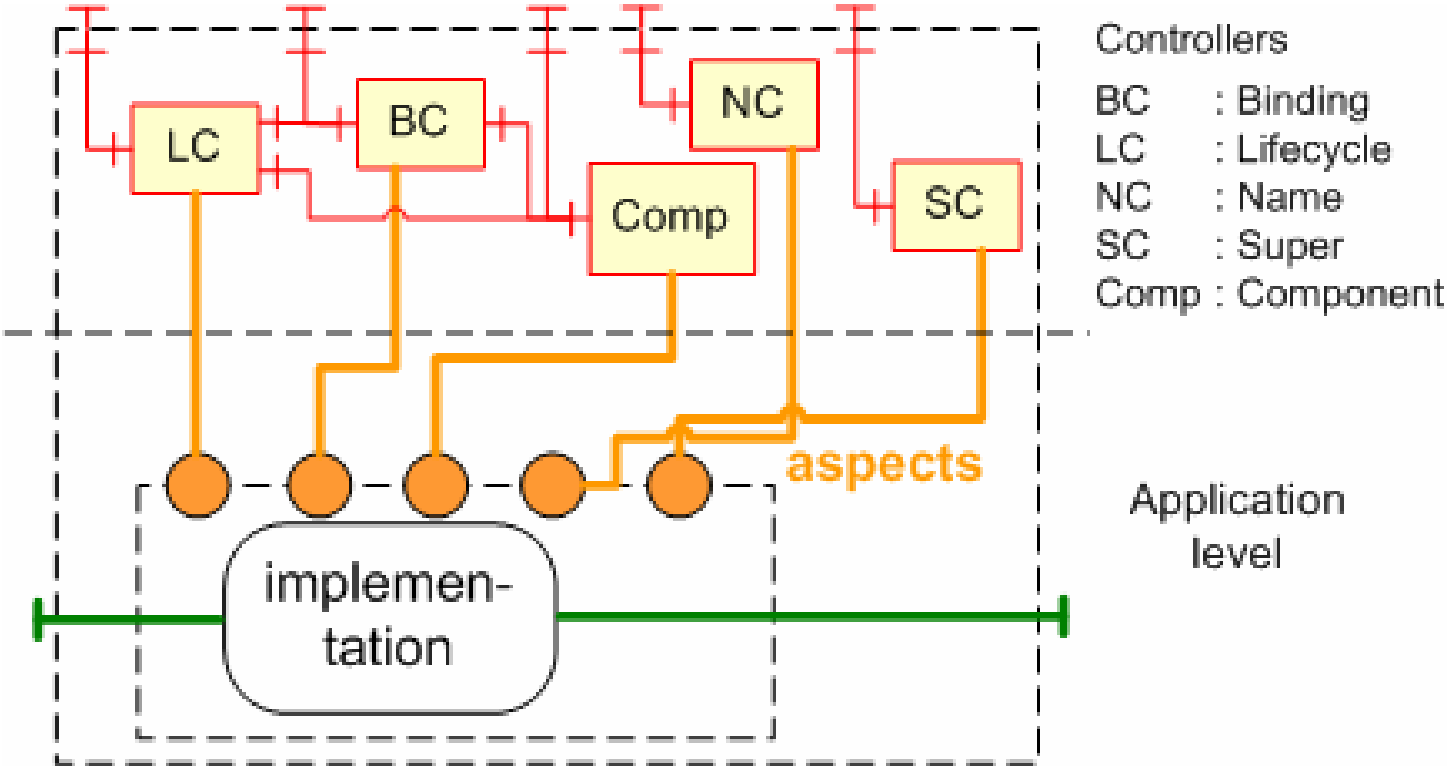
Example: membrane for a primitive component (1/3)



Controllers

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

Example: membrane for a primitive component (2/3)



4.2 AOKell

Primitive membrane definition with Fractal ADL

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

<component name="Comp"     definition="PrimitiveComponentController" />
<component name="NC"       definition="NameController" />
<component name="LC"       definition="NonCompositeLifeCycleController" />
<component name="BC"       definition="PrimitiveBindingController" />
<component name="SC"       definition="SuperController" />
```

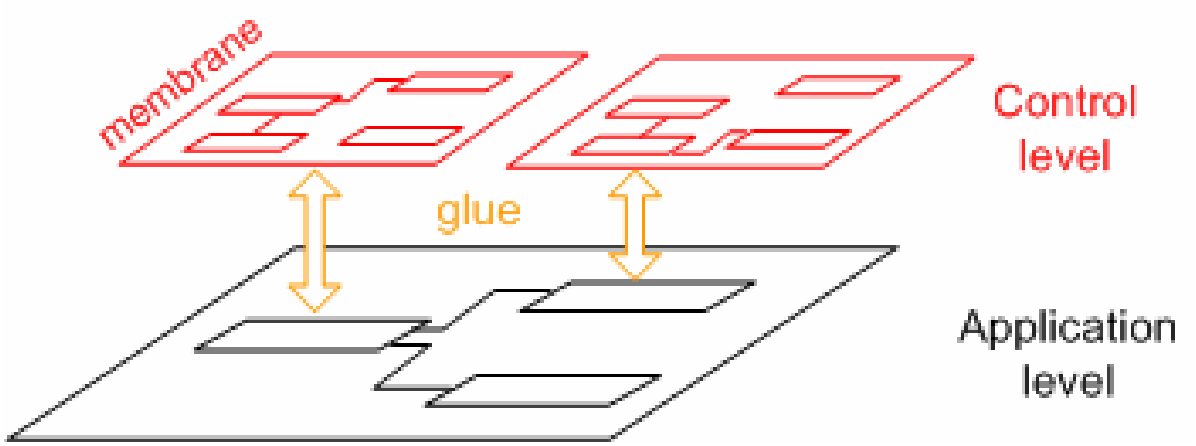
4.2 AOKell

Primitive membrane definition with Fractal ADL (cont'd)

```
    <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>
```

4.2 AOKell

Componentizing the membrane



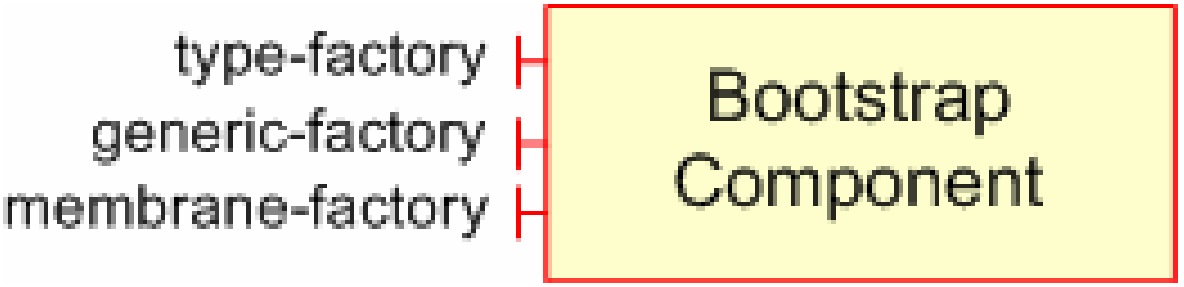
Control level

- no LC
- no templates
- Fractal level ~3.1 components
- 1 aspect per controller
- 1 root composite exporting the control interfaces

Application level

- Fractal level 3.3 components

Fractal Bootstrap component

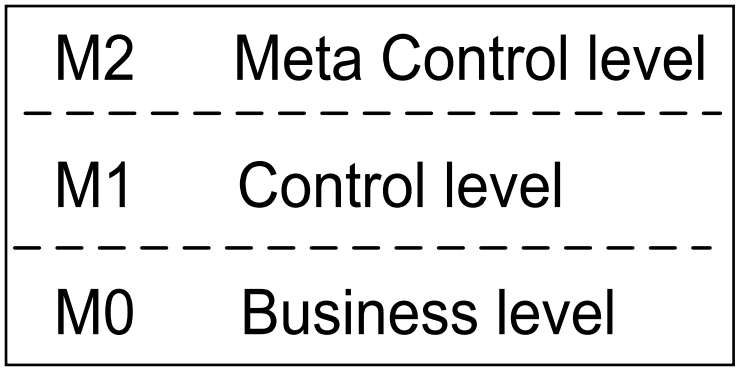


membrane-factory =
generic-factory supporting 2 new controller descriptions
mPrimitive and mComposite

Instantiating a component
- creating the content
- instantiating the membrane (composite component)
- linking the content and the membrane

4.2 AOKell

Issues: controlling the controllers?



Component controllers are regular Fractal components

⇨ they provide control interfaces (BC, NC, …)

How this (meta)-control level must be implemented?

1. « infinite » number of meta-levels (a la 3-Lisp)
2. controllers control themselves (meta-circularity)
3. ad-hoc implementation of the (meta)-control ⇦ chosen solution

4.2 AOKell

- full implementation of the Fractal Specification - **level 3.3 compliant**
- Fractal/Julia junit conformance tests : 138 ok / total: 142
  (4 failed: specific to Julia, or issues in interpreting the specs)
- performances similar to those of Julia
  - no runtime penalty with compentized membranes (but memory footprint ☹)

- existing Fractal applications
  (comanche, FROGi, Fractal Explorer, GoTM, …) run unmodified
- Dream framework [Quéma 05] ported to AOKell
- AOKell has been ported to the .NET platform by Escoffier & Donsez
  - AspectDNG glue: weaving on compiled .NET assemblies ⇨ ∀ .NET languages
- component-based control membrane ported to Julia >= 2.5

- LGPL
- http://fractal.ow2.org/aokell

Plan

5. Autre personnalité Fractalienne

- SCA (Service Component Architecture)
- un modèle de composants pour le SOA
  - Initiative : IBM, Oracle, BEA, SAP, Sun, TIBCO, Siebel, Sybase, Siemens

- But : structurer l'implémentation des SOA
- Mix entre les paradigmes composant et service
- Implémentations actuelles
  - Apache Tuscany, Newton, Fabric3, FraSCAti

- Consortium OpenSOA www.osoa.org
  - 1ères spécifications : 12/2005, v1 03/2007
- En cours de standardisation par OASIS (standards Web Services)

5. Autre personnalité Fractalienne

Tinfi Tinfi is Not a Fractal Implementation

- moteur d'exécution de composants SCA
- développé dans le cadre de la plate-forme FraSCAti (ANR SCOrWare)
- www.scorware.org

OSOA 1.0 Java API

- 5 interfaces (+ 1 for constants)
- 4 exceptions
- 25 annotations
  - 17 general-purpose
  - 2 remote communications
  - 6 policy intent sets

5. Autre personnalité Fractalienne

SCA/Tinfi personality vs Fractal/Julia personality

- dependency injection
  - SCA : @Reference @Constructor
  - Fractal : BindingController
- property injection
  - SCA : @Property
  - Fractal : AttributeController
- component identity
  - SCA : @Context + ComponentContext
  - Fractal : Component
- component name
  - SCA : @ComponentName
  - Fractal : NameController

5. Autre personnalité Fractalienne

SCA/Tinfi personality vs Fractal/Julia personality

- component initialization
  - SCA : ctor + @Init + @EagerInit
  - Fractal : ctor
- component destruction
  - SCA : @Destroy
  - Fractal : *missing*
- stub for accessing components
  - SCA : ServiceReference - CallableReference
  - Fractal : Interface (+ serializable)
- component instanciation policy
  - SCA : @Scope per request, per client, singleton, stateless
  - Fractal : singleton

5. Autre personnalité Fractalienne

SCA/Tinfi personality vs Fractal/Julia personality

- asynchronous method invocation
  - SCA : @OneWay
  - Fractal : see Dream
- callback
  - SCA : @Callback + CallableReference
  - Fractal : *missing*
- conversation management
  - SCA : @Conversation, @ConversationAttributes, @ConversationId, @EndsConversation
  - Fractal : *missing*

5. Autre personnalité Fractalienne

SCA/Tinfi personality vs Fractal/Julia personality

- introspection & reconfiguration API
  - SCA : *missing*
  - Fractal : Component, *Controller
- component hierarchy
  - SCA : tree
  - Fractal : graph (shared component)
- component instanciation
  - SCA : *platform-dependent*
  - Fractal : component factory, template

5. Autre personnalité Fractalienne

Membrane which provides a control semantics to achieve a SCA personality for a component

6 controllers + 1 interceptor

6 controllers

1. SCAComponent : component identity
2. SCAContentController : component instantiation policy
3. SCAIntentController : intent handlers management
4. SCAPropertyController : component properties management

5. SCALifeCycleController : component initialization (@EagerInit)
6. SCABindingController : component bindings

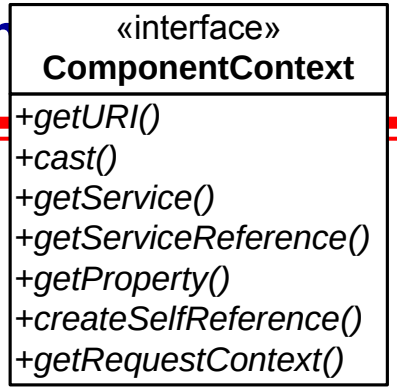
5. Autre personnalité Fractalienn

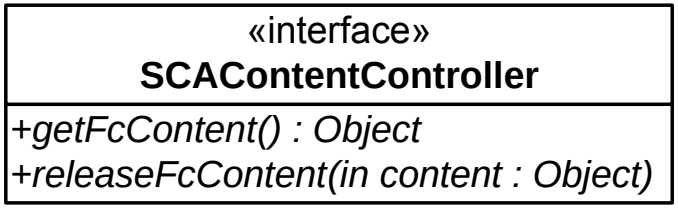
1. SCAComponent
   component identity
   dedicated interface (ComponentContext) and implementation

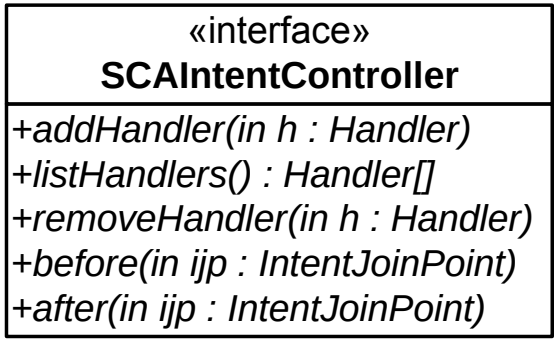
2. SCAContentController
   component instantiation policy
   dedicated implementation,
   private interface (no need to export it)

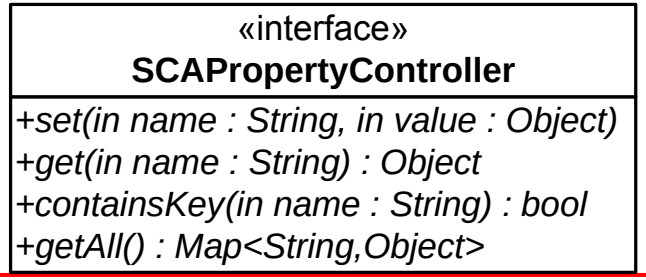
3. SCAIntentController
   intent handlers management

4. SCAPropertyController
   component properties management









5. Autre personnalité Fractalienne

5. SCALifeCycleController : component initialization (@EagerInit)
   same interface as Fractal LC, dedicated implementation
6. SCABindingController : component bindings
   same interface as Fractal BC, dedicated implementation

- interceptor
  1. lifecycle management
  2. component instantiation policy
  3. intent dispatch

- scaPrimitive membrane type
  - 6 SCA-specific controllers
  - + Fractal/Julia controllers
    Component, NC, SC

A scaPrimitive component
- is a SCA component
- is a Fractal component
- can be assembled with other Fractal comp.

5. Autre personnalité Fractalienne

Tinfi scaPrimitive membrane

```
(scaPrimitive
 (
  'interface-class-generator
  (
   'component-itf
   'sca-component-controller-itf
   'binding-controller-itf
   'super-controller-itf
   'lifecycle-controller-itf
   'name-controller-itf
   'sca-content-controller-itf
   'sca-intent-controller-itf
   'sca-property-controller-itf
  )
```

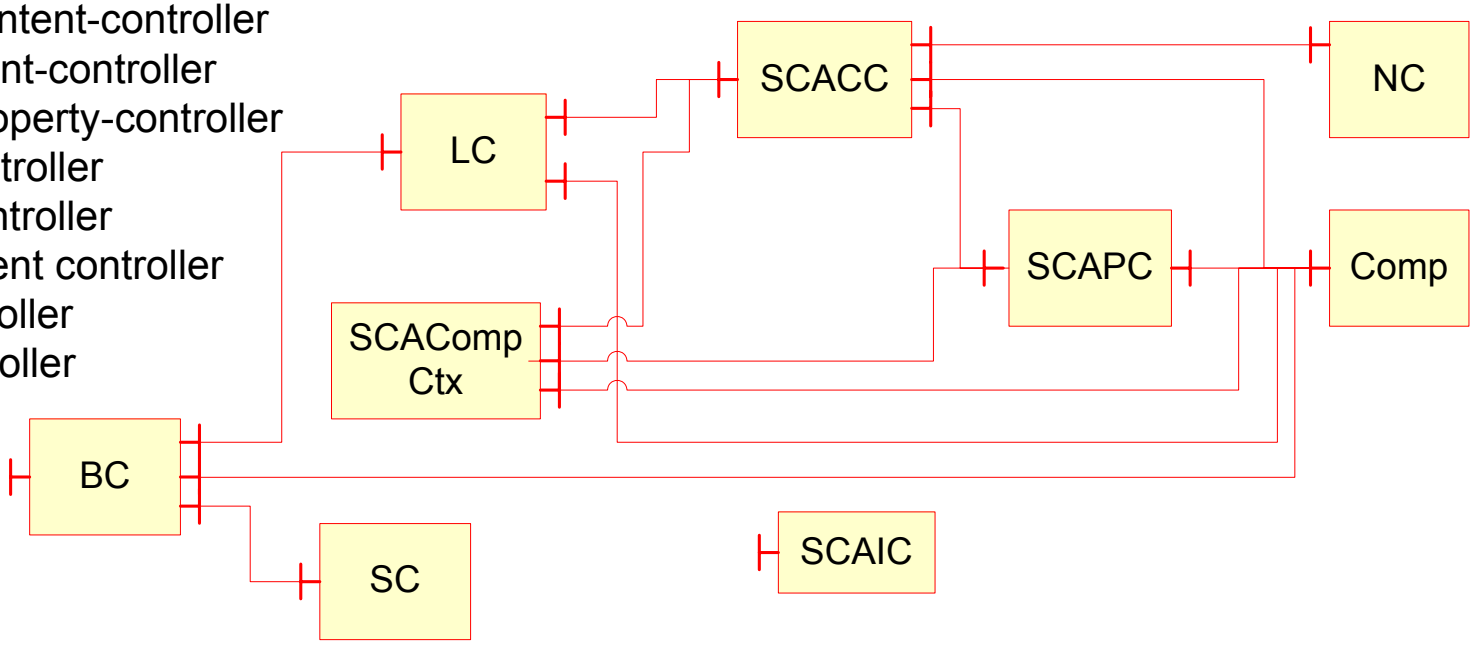
```
 (
  'component-impl
  'sca-component-context-controller-impl
  'sca-primitive-binding-controller-impl
  'super-controller-impl
  'sca-lifecycle-controller-impl
  'name-controller-impl
  'sca-content-controller-impl
  'sca-intent-controller-impl
  'sca-property-controller-impl
 )
 (
  (o.o.f.julia.asm.InterceptorClassGenerator
    o.o.f.julia.asm.LifeCycleCodeGenerator
   …SCAContentInterceptorCodeGenerator
   …SCAIntentInterceptorCodeGenerator
  )
 )
```

5. Autre personnalité Fractalienne

Tinfi scaPrimitive membrane

Controllers

SCACompCtx : sca-component-controller
SCACC: sca-content-controller
SCAIC: sca-intent-controller
SCAPC: sca-property-controller
BC: binding-controller
LC: lifecycle-controller
Comp: component controller
SC: super-controller
NC: name-controller



5. Autre personnalité Fractalienne

Tinfi provides an implementation of the SCA component model

- components created with Tinfi are both
  - SCA compliant
  - Julia compliant
- a way to bring introspection, dynamicity and reconfigurability to SCA
  (features not supported by the SCA specifications)
- proof that the Fractal concepts are suited to implement other component semantics than Julia

Plan

1. Introduction
2. Le modèle Fractal
   - 2.1 Notions de base
   - 2.2 Notions avancées
3. Développer avec Fractal
4. Plates-formes
   - 3.1 Julia
   - 3.2 AOKell
5. Autre personnalité Fractalienne
6. Conclusion

6. Conclusion

Modèle Fractal

- introspectable, dynamique
- composant de grain fin (proche de celui d'une classe)
  - vision hiérarchique favorise le découpage en sous-systèmes
- indépendant des langages de programmation
- vision structurante des applications
  - à la programmation
  - à l'exécution
- un certain nombre d'outils (F4E, FractalExplorer)

mais on peut regretter absence

- profil UML pour Fractal
- lien entre composants UML et composants Fractal

6. Conclusion

Modèle Fractal

Nombreux outils/travaux non abordés dans cet exposé

- communication distante entre composants (Fractal RMI)
- supervision / administration (Fractal JMX)
- Think/Cecilia et l'embarqué
- librairies de composants
  - DREAM, CLIF, Speedo, Perseus, GoTM, …
- usine à liaisons (*Binding Factory*)
- compilation d'applications Fractal (Juliac)

➢ écosystème Fractal vaste

6. Conclusion

- R&D activities and Tools
  - Formal models and calculi (INRIA, Verimag)
  - Configuration (Fractal/Think ADL - FT, INRIA, STM), navigation/query (EMN, FT)
  - Dynamic reconfiguration (FT, INRIA)
  - Management - Fractal JMX (FT)
  - Packaging, deployment (INRIA, LSR, Valoria)
  - Security, isolation (FT)
  - Correctness: structural integrity (FT), behavioural contracts based on assertions (ConFract - I3S, FT), behavior protocols (Charles U., FT), temporal logic (Fractal TLO - FT), automata (INRIA), test (Valoria)
  - QoS management (Plasma - INRIA, Qinna - FT)
  - Self-adaptation, autonomic computing (Jade - INRIA, Safran - EMN, FT)
  - Components & aspects (FAC, Julius, AOKell - INRIA, FT)
  - Components & transactions (Jironde - INRIA)
- Some operational usages
  - Jonathan, Jabyce, Dream, Perseus, Speedo, JOnAS (persistence), GoTM, CLIF…
- Dissemination in industry (FT, STM, Nokia), universities including teaching (Grenoble, Chambéry, Nantes…), conferences (JC, LMO, SC, Euromicro…)

6. Conclusion

Spécifications Fractal
E. Bruneton, T. Coupaye, J.-B. Stefani
The Fractal Component Model
http://fractal.objectweb.org/specification/index.html

Fractal & Julia
E. Bruneton, T. Coupaye, M. Leclerc, V. Quéma, J.-B. Stefani
The Fractal Component Model and its Support in Java
Software Practise and Experience. 36(11-12):1257-1284. 2006.

AOKell
L. Seinturier, N. Pessemier, L. Duchien, T. Coupaye.
A Component Model Engineered with Components and Aspects.
9th Intl. Symp. on Component-Based Software Engineering (CBSE).
LNCS 4063, pp. 139-153. June 2006.

« Référence » sur les composants
C. Szyperski
Component Software – Beyond Object-Oriented Programming
Addison-Wesley, 2nd edition, 2002.

6. Conclusion

- Plus d'informations
  - Site Web : http://fractal.ow2.org
  - Liste de diffusion : fractal@obw2.org

- Remerciements
  - des figures et des transparents empruntés à E. Bruneton, T. Coupaye, J.-B. Stefani