

Intergiciels, composants et architectures logicielles

Université des Sciences et Technologies de Lille
Master Recherche Informatique

L. Duchien, C. Gransart, A.-F. Le Meur, P. Merle, L. Seinturier (resp.)

LIFL – Équipe GOAL & Équipe-projet INRIA ADAM

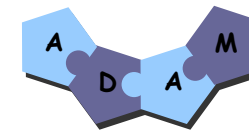
Planning des cours

- Composant (L. Seinturier)
 - ◆ mercredi 19/9, 3/10 – 10h30-12h
 - ◆ jeudi 4/10 – 14h-15h30
- Architecture logicielle (L. Duchien, A.-F. Le Meur)
 - ◆ mercredi 10, 17 & 24/10 – 10h30-12h
- Intergiciel (P. Merle)
 - ◆ mercredi 7, 14 & 21/11 – 10h30-12h
- Application mobile (C. Gransart)
 - ◆ mercredi 28/11 – 10h30-12h

Contenu et modalités

■ Modalités d'évaluation

- ◆ fiche de lecture **individuelle** 40%
 - ❖ présentation du sujet mercredi 3/10
 - ❖ à lire : document sur les *middlewares* reconfigurables
 - ❖ à écrire : 5-6 pages
 - ◆ résumer le document
 - ◆ recherche bibliographique sur les travaux connexes
 - ❖ à rendre pour le 30/11/2007
- ◆ examen final en décembre 60%



Composants pour les intergiciels et les applications réparties

Lionel.Seinturier@lifl.fr

LIFL – Équipe GOAL & Projet INRIA Jacquard

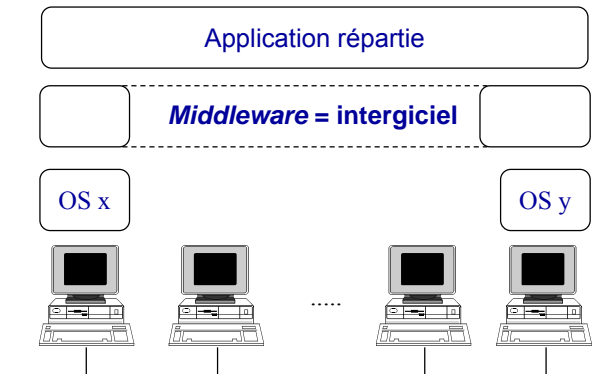
Plan

1. Introduction
2. Le modèle Fractal
 - 2.1 Notions de base
 - 2.2 Notions avancées
3. Développer avec Fractal
4. Plate-forme de référence
5. Plate-formes avancées
 - 5.1 Composant et aspect
 - 5.2 Composant et service
6. Conclusion

1. Introduction

« *Middleware is everywhere* »

© IBM



1. Introduction

Dualité application / intergiciel (*middleware*)

Middleware

- s'appuie sur les API de « haut niveau » fourni par l'OS
- fournit des API de « haut niveau » aux applications réparties
- masque l'hétérogénéité des OS sous-jacents
- fournit des protocoles de communication distants
- fournit des services aux applications réparties
- fournit un interfaçage avec 1 ou plusieurs langages de progr.

1. Introduction

Dualité application / intergiciel (*middleware*)

Concepts du génie logiciel

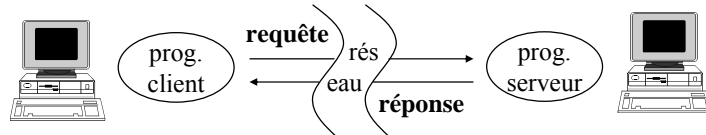
- procédure, objet, composant, ...
- appliqués aux 2 niveaux
- parfois de manière uniforme
 - ◆ ex. : appli objet sur *middleware* objet (Java RMI, CORBA/C++)
- parfois en décalage
 - ◆ ex. : appli composants sur *middleware* objet (EJB, CCM)
 - ◆ ex. : appli C sur *middleware* objet (CORBA)

Introduction

≠ styles de *middleware*

Nombreux paradigmes de communication associés

⇒ le principal : interaction **requête/réponse** ou **client/serveur**



Interaction client/serveur

= 1 requête + 1 réponse

= demande d'exécution d'un traitement à distance et réponse

≈ appel procédural étendu au cas où appelant et appelé ne sont pas situés sur la même machine

Introduction

≠ styles de *middleware*

Deuxième paradigme

⇒ interaction **par messagerie** (MOM : *Message-Oriented Middleware*)



Attention ≠ envoi message sur socket

Protocoles de niveau applicatif (pas transport) + propriétés (ex transactionnelles)

MOM : comm. **asynchrone** (fonctionnement client et serveur découplés)

Interaction client/serveur comm. **synchrone**

1. Introduction

≠ styles de *middleware*

Autres styles de *middleware*

- de diffusion de flux de données
- pair-à-pair (P2P *peer-to-peer*)
- espace de données partagées
- à base de code mobile
- à base d'agents « intelligents »

- chaque style impose une « forme » d'appli répartie
- applications +/- riches

1. Introduction

≠ cibles pour le *middleware*

- applications « Internet » de commerce en ligne
 - ◆ EJB, .NET
- grilles et clusters de calcul
 - ◆ MPI, CORBA
- interconnexion d'applications sur Internet
 - ◆ Web Services, MOM
- telecom, contrôle commande
 - ◆ CORBA, protocole de messageries industrielles

1. Introduction

La recherche en *middleware*

- trouver les bons (nouveaux) concepts pour définir, concevoir, implémenter le *middleware*
- propriétés recherchées
 - ◆ **adaptable** au contexte, aux besoins des utilisateurs/développeurs
 - ◆ **pouvoir d'expression élevé** pour exprimer simplement la complexité des mécanismes à mettre en œuvre
 - ◆ ayant une bonne **assise formelle** pour autoriser la preuve, vérification
 - ◆ suffisamment **ouvert** pour autoriser un degré de variabilité important
 - ◆ performant pour envisager une mise en œuvre à large échelle
- des challenges (parmi d'autres) pour la recherche en *middleware*
 - ◆ *middleware* universel **multi-échelles** (de l'embarqué à la grille)
 - ◆ *middleware* *autonomic*
 - ◆ démarche intégrée sur tout le cycle de conception (des modèles au déploiement)

1. Introduction

Dans le cadre de ce cours

- composants logiciels (L. Seinturier)
- ADL (*Architecture Description Languages*) (L. Duchien)
- DSL (*Domain Specific Languages*) (A.-F. Le Meur)

- architecture des plates-formes *middleware* (P. Merle)
- *middleware* pour les applications mobiles (C. Gransart)

1. Introduction

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

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

Besoins

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

Présent : plusieurs tendances

- composant, aspect, MDE, réflexivité

1. Introduction

Ce que l'on espère des composants (vs objets)

- 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 couverture 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
- Bonobo, Carbon, Plexus, Spring
- au niveau analyse/conception : UML2

1. Introduction

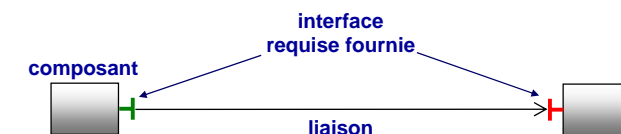
Conséquence de la multiplicité des modèles

- 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

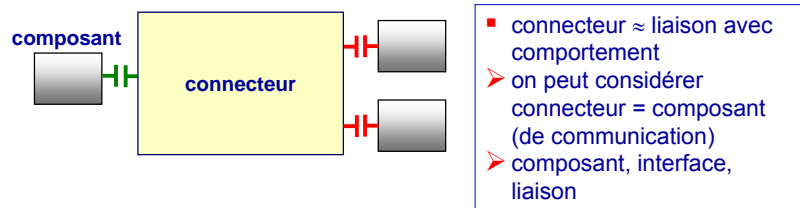


1. Introduction

2ème grande catégorie de modèle de composants

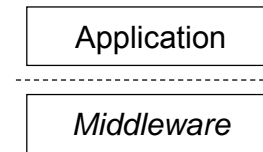
■ triptyque : composant, port, connecteur

- ◆ 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



1. Introduction

Classification des modèles pour système/intergiciel

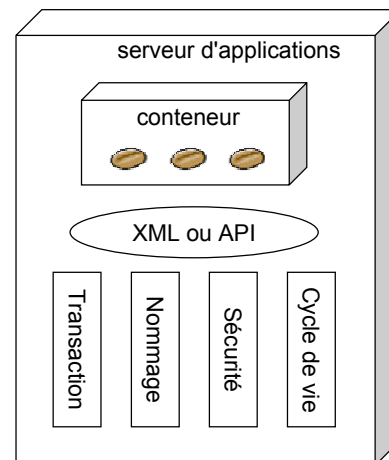


- application : EJB, CCM, .NET/COM+, SCA, Spring
- *middleware* : Fractal, JMX, OpenCOM, OSGi
- *middleware* componentisé pour applications à base de composants
 - ◆ JonasALaCarte : Fractal + EJB [Abdellatif 05]
 - ◆ OSGi + EJB [Desertot 05]

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

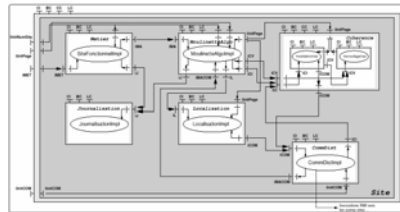
- COTS Commercial Off The Shelf
 - ◆ vieux discours (voir procédures, fonctions, objet, ...)
 - ◆ taille applis ↗ donc besoin : toujours plus de réutilisation
 - ◆ mais *quid* de la contractualisation ?
- « *Programming in the large* »
 - ◆ vs « *programming in the small* » (objet)
 - ◆ vrai d'un certain 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 : assemblés pour construire une architecture

2 visions complémentaires

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

Plan

1. Introduction
2. Le modèle Fractal
 - 2.1 Notions de base
 - 2.2 Notions avancées
3. Développer avec Fractal
4. Plate-forme de référence
5. Plates-formes avancées
 - 5.1 Composant et aspect
 - 5.2 Composant et service
6. Conclusion

2. Le modèle Fractal

FT R&D, INRIA

- *open source*
- <http://fractal.objectweb.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
 - ◆ 1 en C (Think), 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 J2EE [Abdelatif 05]
- Petals : serveur JBI

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

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

2. Le modèle Fractal

2.1 Notions de base

2.1.1 Composant

2.1.2 Interface

2.1.3 Liaison

2.2 Notions avancées

2.2.1 Instanciation

2.2.2 Typage

2.2.3 Niveaux de conformance

■ unité de base d'une application Fractal

■ modèles de type et d'instance

■ entités *compile-time* et *run-time*

■ modèle hiérarchique

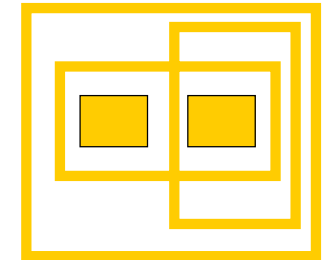
◆ composant composite

◆ 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

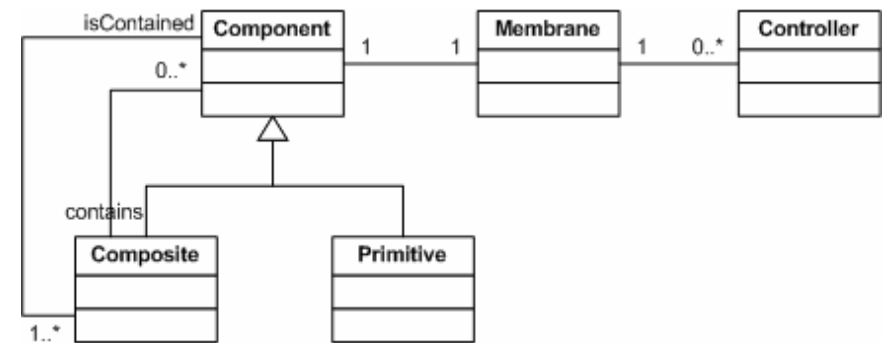
- ◆ rôle d'un *framework* Fractal (Julia, AOKell, ...)

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

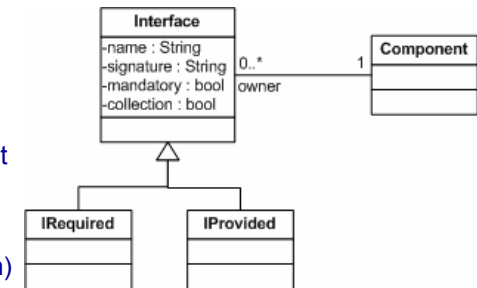
Définit des services offerts ou **requis** par un composant

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

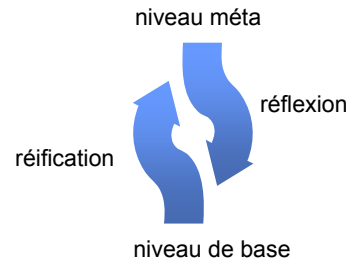
souvent services système (sécurité, persistance, réplication, tolérance aux pannes)
mais pas uniquement : contrats (pre/post), intégrité de données, règles de gestion,
tracabilité, gestion de *workflow*, ...



2.1.2 Interface

Interface **contrôle** \approx 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 \in le composant

+ pour les composants composites

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

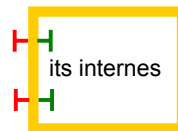
2.1.2 Interface

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

- combinaison avec métier/contrôle et client/serveur ok

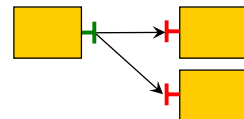
Interface : **interne** vs **externe**

- 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

- ◆ + précisément : entre 1 interface client et 1 interface serveur

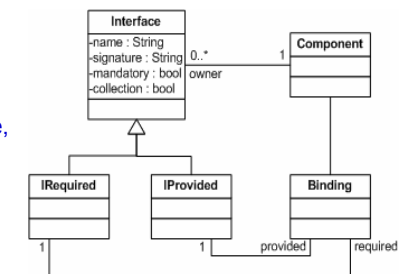
■ explicite et matérialise les dépendances entre composants

■ manipulable à l'exécution

- reconfiguration dynamique

■ sémantique non figée

- ◆ par ex. : invocation méthode locale, distante, diffusion, avec QoS, ...
- ◆ dépend du *binding-controller*

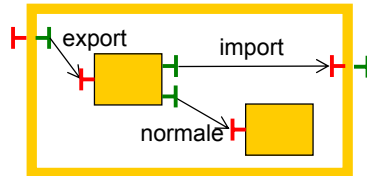


2.1.3 Liaison

2 formes

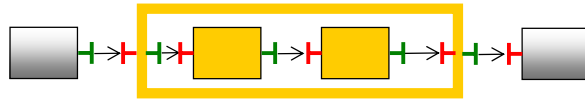
■ primitive

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



■ composite

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



Plan

2. Le modèle Fractal

2.1 Notions de base

2.1.1 Composant

2.1.2 Interface

2.1.3 Liaison

2.2 Notions avancées

2.2.1 Instanciation

2.2.2 Typage

2.2.3 Niveaux de conformance

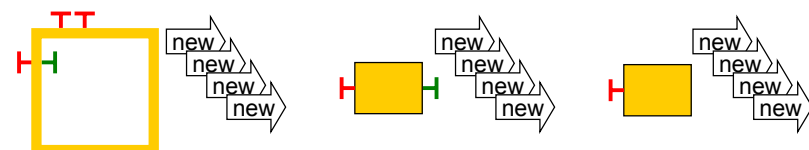
2.2.1 Instanciation

- 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*

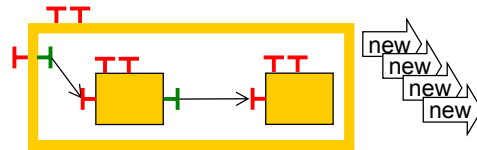
- composant prédéfini par Fractal
- peut instancier n'importe quel composant
- informations nécessaires
 - ◆ son type
 - ◆ son implémentation
 - ◆ une forme de membrane



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 \leq ($T1 \leq T2$: $T1$ 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

$T_i = \langle C_i, S_i \rangle$ // C_i : ensemble d'interfaces client – S_i : serveur

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

$$s1 \leq s2 \equiv s1.name = s2.name \wedge s1.signature \leq s2.signature \wedge (s2 \text{ obligatoire} \Rightarrow s1 \text{ obligatoire}) \wedge (s2 \text{ collection} \Rightarrow s1 \text{ collection})$$

$$c1 \leq c2 \equiv c1.name = c2.name \wedge c2.signature \leq c1.signature \wedge (c2 \text{ optionnelle} \Rightarrow c1 \text{ optionnelle}) \wedge (c2 \text{ collection} \Rightarrow c1 \text{ collection})$$

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
 - 2.1 Notions de base
 - 2.2 Notions avancées
- 3. Développer avec Fractal**
4. Plate-forme de référence
5. Plates-formes avancées
 - 5.1 Composant et aspect
 - 5.2 Composant et service
6. Conclusion

3. Développer avec Fractal

Développer des applications Fractal en Java

- 3 outils complémentaires
 - ◆ API ensemble d'interface Java
 - ◆ fraclet annotations Java 5 ou Java 4 XDoclet
 - ◆ Fractal ADL ADL XML
- 2 étapes
 - ◆ définition des composants : avec l'API et/ou avec fraclet
 - ◆ définition de l'assemblage : avec l'API et/ou avec Fractal ADL
- scénarii d'utilisation possibles
 - ◆ API pure (composants + assemblage)
 - ◆ API (composants) + Fractal ADL (assemblage)
 - ◆ fraclet (composants) + API (assemblage)
 - ◆ fraclet (composants) + ADL (assemblage)
 - ◆ API + fraclet (composants) + Fractal ADL (assemblage)

3. Développer avec Fractal

Fraclet

- définit un modèle de programmation Java pour Fractal
- à base d'annotations
 - ◆ Java 5 ou
 - ◆ Java 1.4 XDoclet
- réduit le volume de code à écrire (voir + loin API)
- phase de pré-compilation
 - ◆ génération du code source associé aux annotations
- indépendant des plates-formes (Julia, AOKell)
- développé par R. Rouvoy et N. Pessemier
- voir <http://fractal.objectweb.org> module fraclet

3. Développer avec Fractal

Fraclet – Principales annotations

- package org.objectweb.fractal.fraclet.annotation
- @FractalComponent(controllerDesc=« ... »)
 - ◆ s'applique à une classe implémentant un composant
 - ◆ définit la membrane associée à un composant
 - ◆ principale membrane : primitive
- @Provides(...)
 - ◆ s'applique à une classe implémentant un composant
 - ◆ indique que le composant fournit un ensemble d'interfaces serveur
- @Requires(name=« ... »)
 - ◆ s'applique à un attribut (*field*)
 - ◆ indique que l'attribut correspond à une interface cliente
 - ◆ autres paramètres possibles pour l'annotation
 - ❖ cardinality : SINGLETON (par défaut) ou COLLECTION
 - ❖ contingency : MANDATORY (par défaut) ou OPTIONAL

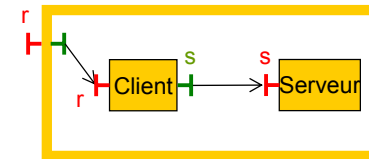
Plan

3. Développer avec Fractal

- 3.1 Exemple « Hello World »
- 3.2 API Fractal
- 3.3 Fractal ADL
- 3.4 Autres outils

3.1 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 Exemple « Hello World »

Le composant Serveur

```
@FractalComponent(controllerDesc="primitive")
@Provides(
    interfaces=
        @Interface(name="s",signature=Service.class) )
public class ServeurImpl implements Service {
    public void print( String msg ) {
        System.out.println(msg);
    }
}
```

3.1 Exemple « Hello World »

Le composant Client

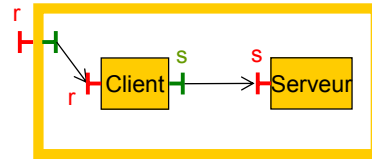
```
@FractalComponent(controllerDesc="primitive")
@Provides(
    interfaces=
        @Interface(name="r",signature=Runnable.class) )
public class ClientImpl implements Runnable {

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

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

3.1 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 Exemple « Hello World »

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

Plan

3. Développer avec Fractal

- 3.1 Exemple « Hello World »
- 3.2 API Fractal
- 3.3 Fractal ADL
- 3.4 Autres outils

3.2 API Fractal

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

Introspection et modification

- liaison : contrôleur de liaison (BC)
- composant
 - ◆ introspection
 - ❖ 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

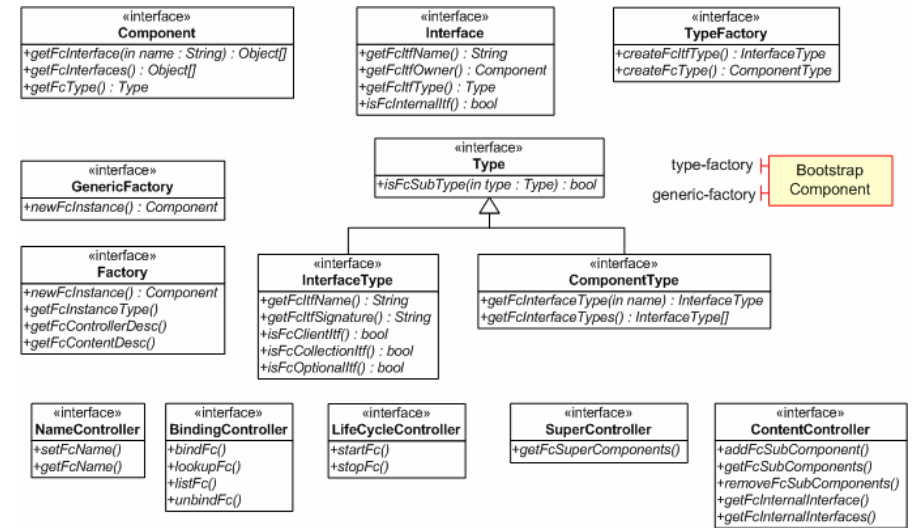
3.2 API Fractal

- légère (16 interfaces, <40 méthodes)
- API Fractal est la base de
 - ◆ Fractal ADL : front-end pour l'API
 - ◆ fractlet : 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

3.2 API Fractal



3.2 API Fractal

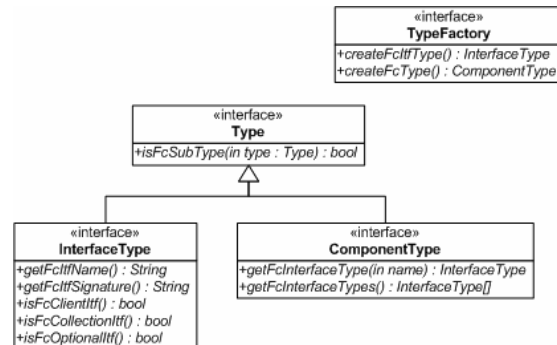
1. Création de types de composants

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 ensemble de types d'interfaces



3.2 API Fractal

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.createFcIType(
        "r", // nom de l'interface
        "java.lang.Runnable", // signature Java de l'interface
        false, // serveur
        false, // obligatoire
        false) // singleton
});
    
```

3.2 API Fractal

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)
});
```

3.2 API Fractal

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
- ...



3.2 API Fractal

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);
    }
}
```

3.2 API Fractal

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;
}
```

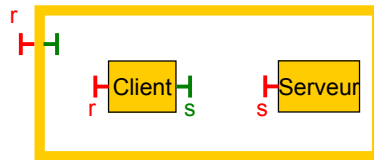
3.2 API Fractal

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);
```



3.2 API Fractal

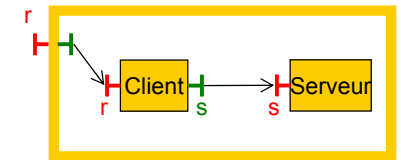
3. Assemblage des composants

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"));
```



3.2 API Fractal

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();
```

3.2 API Fractal

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

- 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

3. Développer avec Fractal

- 3.1 Exemple « Hello World »
- 3.2 API Fractal
- 3.3 Fractal ADL**
- 3.4 Autres outils

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

3.3 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

3.3 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 >
```

3.3 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
>
```

3.3 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
>
```

3.3 Fractal ADL

HelloWorld.fractal

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="Runnable" />
  <component name="Client" definition="Client" />
  <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>
```

Client.fractal

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

3.3 Fractal ADL

Notions additionnelles

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

3.3 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
>
```

3.3 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>
```

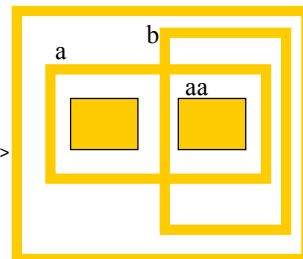
```
<definition name="Client" extends="ClientType">
  <content class="ClientImpl" />
</definition>
```

3.3 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 lère

```
<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>
```



3.3 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

3.3 Fractal ADL

Notions additionnelles - Configuration

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

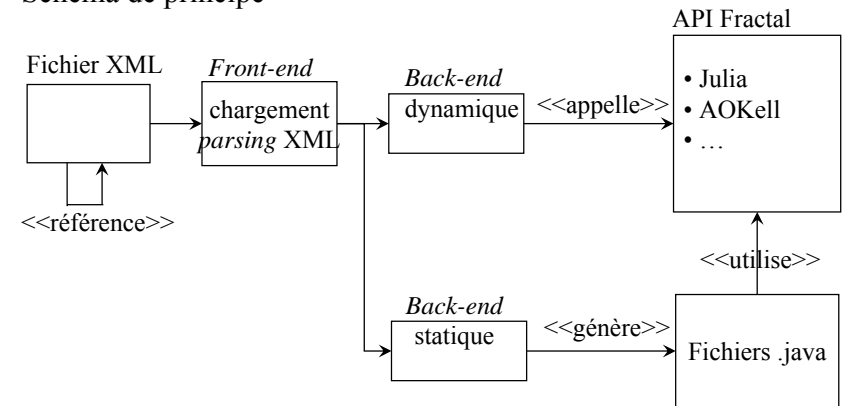
```
<definition name="Client" arguments="itfname,impl" >
  <interface name="r" role="server" signature="Main" />
  <interface name="s" role="client" signature="\${itfname}" />
  <content class="\${impl}" />
</definition>
```

```
Map context = new HashMap();
context.put("itfname", "Service");
context.put("impl", "ClientImpl");

Object o = f.new Component("HelloWorld", context);
```

3.3 Fractal ADL

Schéma de principe



Par défaut : *back-end* dynamique

Plan

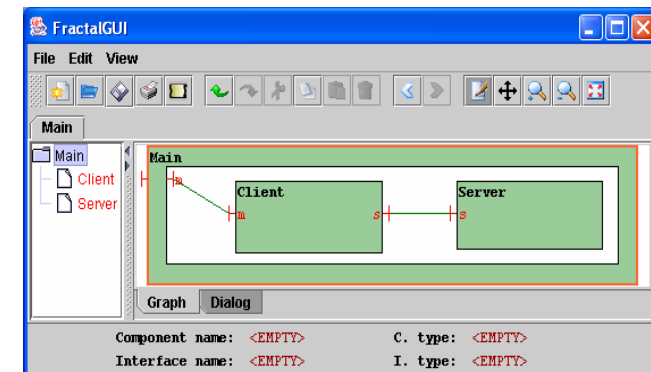
3. Développer avec Fractal

- 3.1 Exemple « Hello World »
- 3.2 API Fractal
- 3.3 Fractal ADL
- 3.4 Autres outils

3.4 Autres outils

FractalGUI

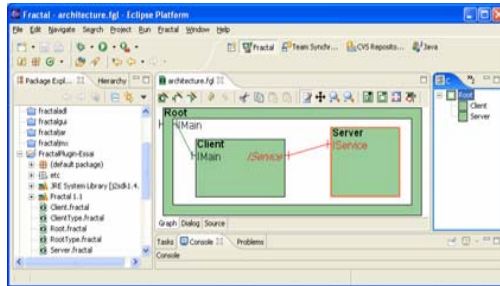
- outil de conception d'architectures Fractal
- génération de squelette de code



3.4 Autres outils

FractalGUI - Plugin Eclipse

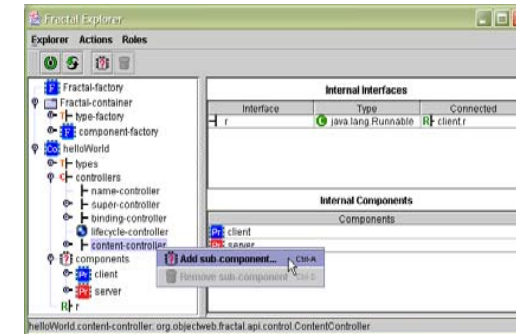
- intégration FractalGUI dans plug-in Eclipse
- auteurs
 - ◆ C. Boulet (ESSI Nice - 2004), P. Collet
 - ◆ O. Ghyselinx (INRIA Lille - 2005) : génération code Fractal 2.0



3.4 Autres outils

FractalExplorer

- console d'administration
- pilotage (run-time) d'une application Fractal
<http://fractal.objectweb.org/tutorials/explorer/index.html>



Plan

1. Introduction
2. Le modèle Fractal
 - 2.1 Notions de base
 - 2.2 Notions avancées
3. Développer avec Fractal
- 4. Plate-forme de référence**
5. Plates-formes avancées
 - 5.1 Composant et aspect
 - 5.2 Composant et service
6. Conclusion

4. Plate-forme de référence

Spécifications (v2.0.3) : fractal.objectweb.org/specification/index.html

≠ implémentations

pour ≠ langages, besoins, domaines ciblés (grilles, embarqué, ...), performances

- ◆ 3 en Java : Julia, ProActive, AOKell
- ◆ 1 en SmallTalk : FracTalk
- ◆ 1 en C : Think/Cecilia
- ◆ 1 en C++ : Plasma
- ◆ 1 pour .NET : FractNet (basée sur AOKell)

Julia (plate-forme de référence) : fractal.objectweb.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

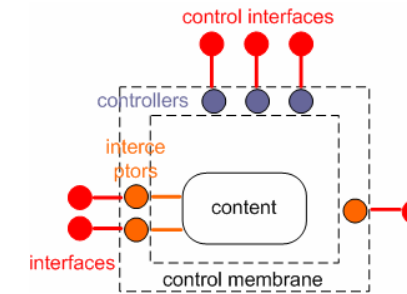
4. Plate-forme de référence

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)

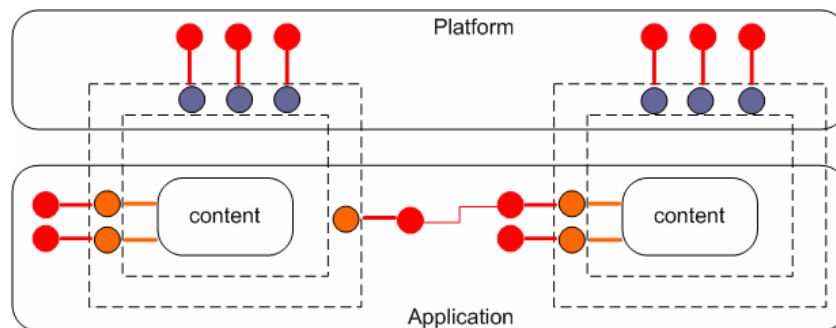
4. Plate-forme de référence

■ représentation usuelle



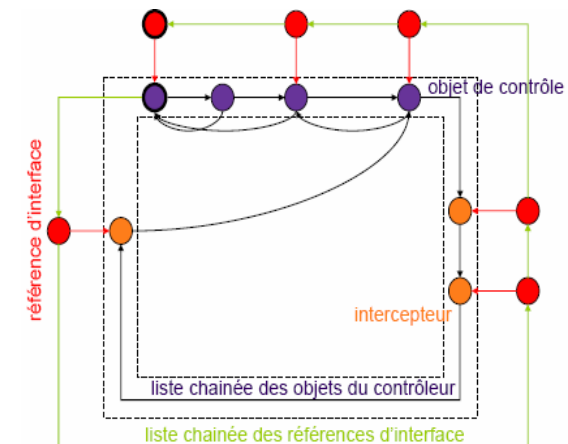
4. Plate-forme de référence

■ vue plate-forme



4. Plate-forme de référence

Structures de données pour un composant Fractal/Julia



4. Plate-forme de référence

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

4. Plate-forme de référence

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. Plate-forme de référence

Génération des membranes de contrôle

Mécanisme de *mixin* [Bracka 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. Plate-forme de référence

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. Plate-forme de référence

Exemple mise en oeuvre Mixin Julia

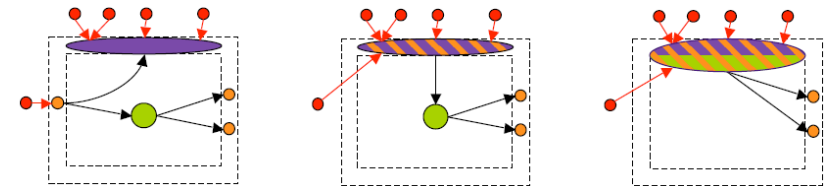
mixin O N M

```
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");
        n$0();
    }
    public int count;
}
```

4. Plate-forme de référence

Optimisation intra-composant

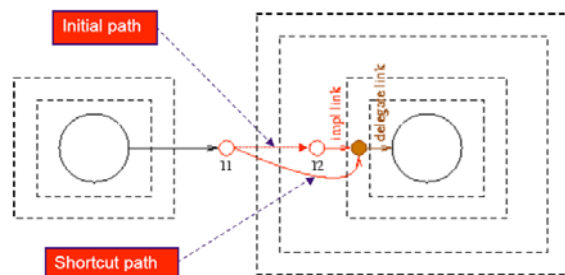
- fusion
 - ◆ des objets de contrôle (gauche)
 - ◆ + intercepteurs (milieu)
 - ◆ + implémentation composant (droite)
- mise en oeuvre
 - ◆ manipulation de *bytecode* avec ASM
 - ◆ algorithme de *mixin*



4. Plate-forme de référence

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

1. Introduction
2. Le modèle Fractal
 - 2.1 Notions de base
 - 2.2 Notions avancées
3. Développer avec Fractal
4. Plate-forme de référence
5. Plates-formes avancées
 - 5.1 Composant et aspect
 - 5.2 Composant et service
6. Conclusion

5. Plates-formes avancées

Retour sur la problématique

- *middleware* : fournit des services
- application : utilise les services *middleware*
- application : construite comme assemblage de composants

2 problématiques

- couplage vertical (application-services *middleware*) fort
- couplage horizontal (application-application) fort

Inconvénient couplage fort

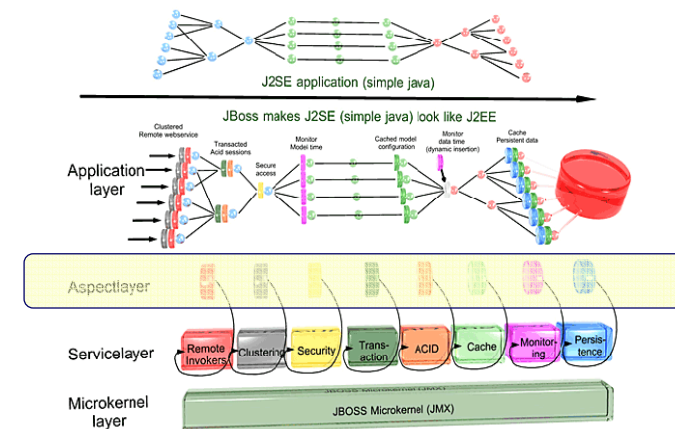
- nuit évolutivité, réutilisabilité

2 directions de recherche pour y remédier

- composant et aspect
- composant et service

5.1 Composant et aspect

Exemple : Serveur Java EE JBoss et JBoss AOP



5.1 Composant et aspect

Les aspects en tant que concept pour réaliser le couplage entre application et services middleware

Vocabulaire

- AOP (Aspect-Oriented Programmation)
- par extension : AOSD (AO Software Development)

5.1 Composant et aspect

5. Plates-formes avancées

5.1 Composant et aspect

5.1.1 AOP

5.1.2 Plate-forme AOKell

5.1.3 Fractal Aspect Component

5.2 Composant et service

5.1.1 AOP

Aux limites de la programmation objet

- La programmation orientée objet
 - 1968 : Simula
 - 70s : SmallTalk
 - 80s : C++
 - 90s : Java puis C#
- indéniablement un succès
- programmation plus claire, modulaire, efficace (par rapport procédural, ...)

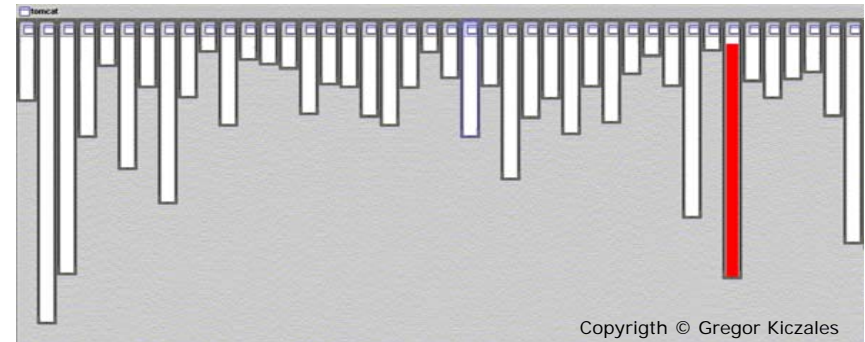
Mais plusieurs exemples de "résistance" à l'*objectisation*

- entrelacement et dispersion de code
- pas de bonne structuration naturelle

5.1.1 AOP

Tomcat

- XML parsing in org.apache.tomcat
 - ◆ red shows relevant lines of code
 - ◆ nicely fits in one box

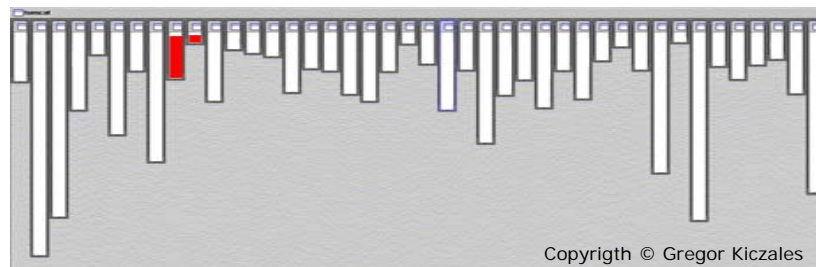


Copyright © Gregor Kiczales

5.1.1 AOP

Tomcat

- URL pattern matching in org.apache.tomcat
 - ◆ nicely fits in two boxes (using inheritance)



Copyright © Gregor Kiczales

5.1.1 AOP

Session expiration



Copyright © Gregor Kiczales

5.1.1 AOP

Analyse

Code

- ⇒ **dispersé** à plusieurs endroits (*scattered*)
- ⇒ **entrelacé** avec le reste de l'application (*tangled*)
- ⇒ parfois redondant (mêmes fragments à plusieurs endroits)

- ⇒ difficile à maîtriser (pas de structure claire)
- ⇒ difficile à changer
 - ◆ retrouver toutes les localisations (sans en oublier)
 - ◆ faire attention aux liaisons avec le reste du code

5.1.1 AOP

Autre exemple de code dispersé et entremêlé

```
class BlockingBuffer {                                /* Politique buffer vide */
                                                       /* Politique buffer plein */
                                                       /* Politique FIFO */

    final int size = 1000;
    int first, last;
    int[] buf = new int[size];

    public synchronized
    int read() {
        while (first==last) {
            wait();
        }
        int element = buf[first];
        first = (first+1)%size;
        notify();
        return element;
    }

    public synchronized
    void write(int e) {
        int nextposition = (last+1)%size;
        while( nextposition == first ) {
            wait();
            nextposition = (last+1)%size;
        }
        last = nextposition;
        buf[last] = e;
        notify();
    }
}
```

5.1.1 AOP

Autre exemple de code entremêlé

Programme JDBC, Java EE, ...

```
public void setX(int value)
    throws RemoteException {
    if (User.getAuthentication()...
        Tx.beginTransaction() ;
        // code JDBC : stockage de X
        ...
        Tx.commitTransaction() ;
    }
    << com dist
    << sécurité
    << transaction
    << persistance
    << métier
    << transaction
```

5.1.1 AOP

Contraintes d'intégrité référentielles

- classes : Facture et Client
- contrainte : ne pas supprimer un client qui a encore des factures non réglées
- où coder cette contrainte ?
 - classe Client
 - on introduit une dépendance entre Client et Facture
 - on ne peut pas réutiliser Client indépendamment de Facture
 - classe Facture
 - pas logique : aucune raison de faire cela
- solution la moins pire : classe Client
- la contrainte naît de la relation entre Client et Facture
 - transverse à ces deux classes

5.1.1 AOP

AOP : Aspect-Oriented Programming

[Kiczales 97]

Aspect

- Fonctionnalité **transverse** (*crosscutting*) à plusieurs entités logicielles
- ex : sécurité, persistance, réplication, tolérance aux fautes
- ex : contrats, *design patterns*, intégrité de données, règles de gestion

■ Buts

- ⇒ pouvoir concevoir chacun de ces aspects séparément
- ⇒ pour les intégrer à une application

5.1.1 AOP

Tissage

Notion de "tisseur d'aspects" (*aspect weaver*)

- ⇒ entrée : aspects, application, directives de tissage
- ⇒ sortie : l'application augmentée des aspects

- tissage statique (AspectJ, ...) aspect = entité *compile time*
- tissage dynamique (JAC, ...) aspect = entité *run time*
- performance vs flexibilité

5.1.1 AOP

Langages / frameworks AOP

■ Java

- AspectJ, CaesarJ, ComposeJ, DJ, D, abc, LogicAJ, LoopsAJ
- JAC, JBoss AOP, EAOP, PROSE, Nanning, Wool, AspectWerkz, JMangler, DAOP, DWF, dynaop, Jeet, josh, JoyAop, Scope, Spring, FAC

⇒ Smalltalk : AOP/ST, Apostole, AspectS, FracTalk-AOP

⇒ C/C++ : AspectC, AspectC++, Arachne

⇒ C#

- AspectDNG, AOP.NET, AspectC#, Loom.NET, Weave.NET, AopDotNetAop, AOPEngine.NET, Aspect.Net, Aspect#, Eos, PostSharp, SetPoint!, NAspect

⇒ PHP : AOPHP, phpaspect

⇒ Python : Pythius

⇒ COBOL : AspectCobol, cobble

5.1.1 AOP

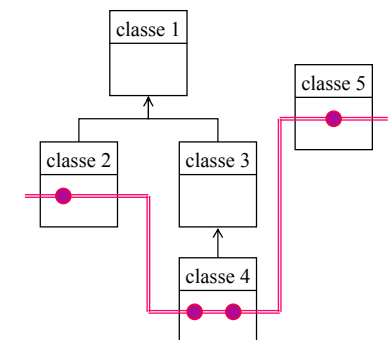
4 notions de base

- **point de jonction** (*join point*)
- ⇒ point dans le flot d'exec. du prog.

- **coupe** (*pointcut*)
- ⇒ ensemble de points de jonction
- ⇒ langage de *pattern* (* .. +)
- ⇒ quantification sur le programme

- **advice**
- ⇒ le comportement d'un aspect (≈ méthode)

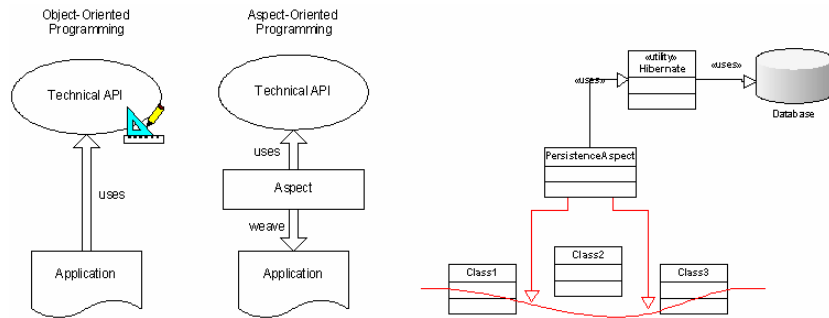
- **aspect**
- ⇒ coupe(s) associée à advice
- ⇒ *advice(s)* exécutés avant/après points de jonction de la coupe



5.1.1 AOP

2 constats sur l'AOP

- ⇒ conduit à une inversion du contrôle (IoC: *Inversion of Control*)
- ⇒ technique d'intégration



5.1.2 AOKell

Plate-forme Fractal

- conforme aux spécifications
- travail de recherche commun INRIA / France Telecom R&D

2 contributions principales

- aspect : réalise glu composant-contrôleurs
 - ◆ 2 versions : AspectJ et Spoon
 - ❖ AspectJ [Kiczales 01] www.eclipse.org/aspectj
 - ❖ Spoon [Pawlak 05] spoon.gforge.inria.fr
Open-compiler for efficient transformations of Java programs
- réflexivité : notion de "composant contrôleur"

5.1.2 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)

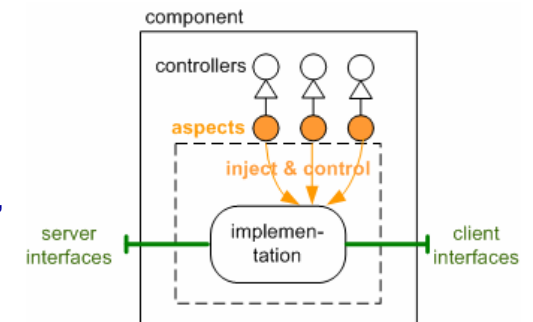
AspectJ

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

5.1.2 AOKell

Aspect glue layer

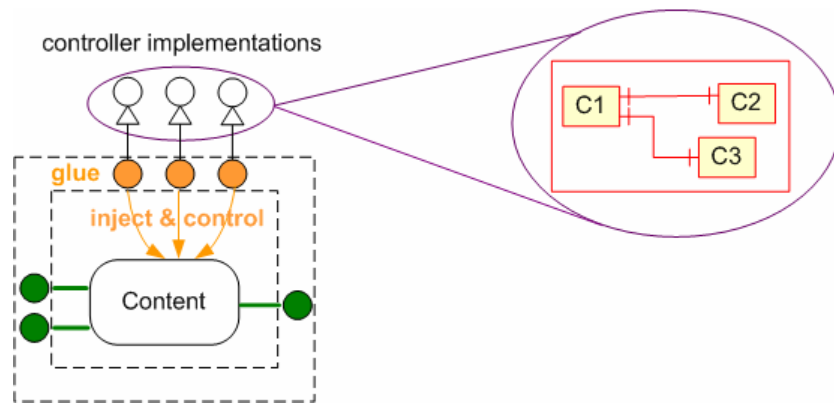
- 1 aspect per controller type
- 7 aspects
BC, LC, NC, CC, SC,
Factory, Component



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

5.1.2 AOKell

Componentizing the membrane



5.1.2 AOKell

Componentizing the membrane

■ notion of a (primitive) component-controller

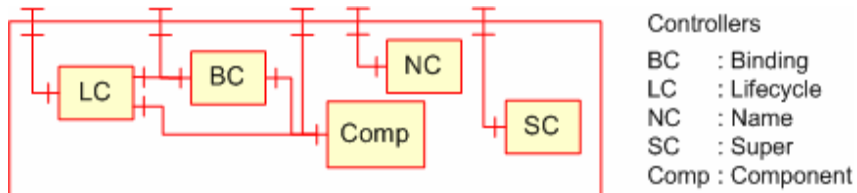
- ◆ a component implementing a control interface
BC, LC, NC, Component, SC, CC, Factory

■ membrane

- ◆ an assembly of components-controllers
- ◆ a composite component-controller
- ◆ can benefit from the tools available with Fractal
e.g. Fractal ADL

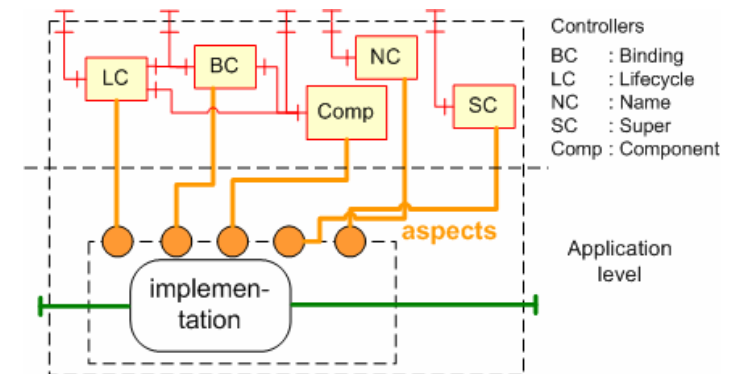
5.1.2 AOKell

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



5.1.2 AOKell

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



5.1.2 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" />
```

5.1.2 AOKell

Primitive membrane definition with Fractal ADL (cont'd)

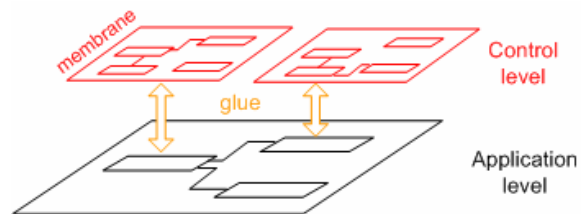
```
<binding client="this//component"    server="Comp//component" />
<binding client="this//name-controller" server="NC//name-controller" />
<binding client="this//lifecycle-controller" server="LC//lifecycle-controller" />
<binding client="this//binding-controller" server="BC//binding-controller" />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

5.1.2 AOKell

Componentizing the membrane



5.1.2 AOKell

Issues: controlling the controllers?

M2	Meta Control level
M1	Control level
M0	Business level

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

5.1.2 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
- LGPL
- <http://fractal.objectweb.org/aokell>

5.1.3 Fractal Aspect Component

N. Pessemier PhD thesis [2004-07] (advisors: L. Seinturier, L. Duchien) + FT R&D

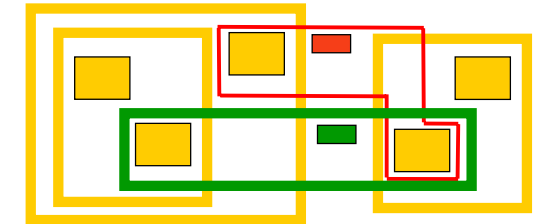
Goal

- an aspect-based programming model for components
- interpreting aspect-oriented concepts in term of component notions
- extending component notions

Results: 3 notions

- aspect component
- aspect domain
- aspect binding

⇒ FAC
<http://fac.gforge.inria.fr>



5.2 Composant et service

- 5. Plates-formes avancées
 - 5.1 Composant et aspect
 - 5.2 Composant et service

5.2 Composant et service

Constat : applications réparties (CORBA, .NET, Java EE, ...) actuelles

- souvent complexes, rigides, peu évolutives

Tendance SOA (Service Oriented Architecture)

- besoins identifiés
 - ◆ interfaces bien définies avec une sémantique lié au métier
 - ◆ protocoles de communication standardisés
 - ◆ recombinaison flexible de services pour améliorer la flexibilité du logiciel
- "vision" [Consortium Open SOA – www.osoa.org]
 - ◆ *A service is an abstraction that encapsulates a software function*
 - ◆ *Developers build services, use services and develop solutions that aggregate services*
 - ◆ *Composition of services into integrated solutions is a key activity*

5.2 Composant et service

SOA

- essentiellement des principes architecturaux (logiciels)
 - ◆ couplage faible
 - ❖ *Components integrate with other components without needing to know how other components are implemented*
 - ◆ flexibilité
 - ❖ *Components can easily be replaced by other components*
 - ◆ services
 - ❖ *Services can be easily invoked either synchronously or asynchronously*
 - ◆ composition
 - ❖ *Composition of solutions clearly described*
 - ◆ productivité
 - ❖ *Easier to integrate components to form composite application*
- souvent organisés autour des Web Services (mais pas nécessairement)
 - ◆ SOAP (HTTP, XML), WS-*, orchestration BPEL

5.2 Composant et service

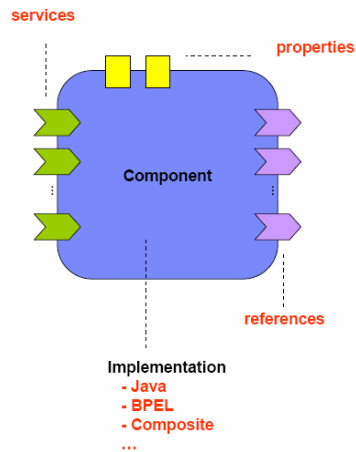
SCA : un modèle de composants pour le SOA

- Initiative : IBM, Oracle, BEA, SAP, Sun, TIBCO, ...
- But : structurer l'implémentation des SOA
- 1ères spécifications : 12/2005, v1 03/2007
- Implémentations : Apache Tuscany, Newton, fabric3
- Assembly model
 - ◆ how to define structure of composite applications
- Component implementations specifications
 - ◆ how to write business services in particular languages
 - ◆ Java, C++, PHP, Spring, BPEL, EJB SLSB, ...
- Binding specifications
 - ◆ how to access services
 - ◆ Web services, JMS, RMI-IIOP, REST, ...
- Policy framework
 - ◆ how to add infrastructure services
 - ◆ security, transactions, reliable messaging, ...

5.2 Composant et service

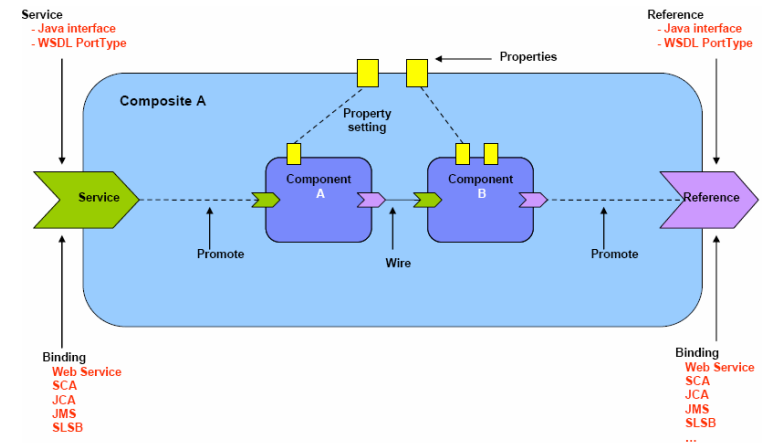
Composant SCA

- service/référence
- propriété
- implémentation



5.2 Composant et service

Assemblage SCA



5.2 Composant et service

Assemblage SCA

■ service/référence

- ◆ type
 - ❖ langage de définition d'interface (IDL) : Java ou WSDL
- ◆ *binding* : association avec une technologie de communication
 - ❖ Web Service, RMI-IIOP, JMS, appel local intra-JVM

■ liens

- ◆ référence vers service : *wire*
- ◆ service/service ou référence/référence : lien de promotion (*promote*)

5.2 Composant et service

Modèle de programmation

- Assembly model
 - ◆ ADL basé XML
- Component implementations
 - ◆ 1 ensemble de règles de programmation par langage
 - ❖ Java : utilise intensément annotations Java 5

```
@Service(AccountService.class)
```

```
public class AccountServiceImpl implements AccountService {  
    @Reference public AccountDataService accountDataService;  
    @Reference public StockQuoteService stockQuoteService;  
    @Property public String currency;  
    public AccountReport getAccountReport(String s) { ... }  
}
```

5.2 Composant et service

SCA

■ *yet another component model*

- ◆ une façon de revisiter les problématiques
 - ❖ de la répartition
 - ❖ des services pour les plates-formes *middleware*
 - ❖ des concepts des *frameworks* de composants
- ◆ renforce l'idée de l'indépendance services/implémentations

■ point original : indépendance interfaces/protocoles communications

■ après CORBA, .NET

- ◆ nouvelle tentative autour du "*middleware* universel" pour faire communiquer l'existant

■ comme CORBA, .NET

- ◆ met plus l'accent sur le modèle de programmation (*software engineering*) des applications que sur l'architecture de la plate-forme *middleware* (*product dependant*)
- ◆ comment trouver la bonne abstraction pour l'ingénierie de la répartition

Plan

1. Introduction
2. Le modèle Fractal
 - 2.1 Notions de base
 - 2.2 Notions avancées
3. Développer avec Fractal
4. Plate-forme de référence
5. Plates-formes avancées
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 (FractalGUI, FractalExplorer)

mais on peut regretter absence

- vrai plug-in IDE (Eclipse)
- profil UML pour Fractal
- lien entre composants UML et composants Fractal

6. Conclusion

Modèle Fractal

Nombreux autres outils/travaux non abordés dans cet exposé

- communication distante entre composants (Fractal RMI)
- supervision / administration (Fractal JMX)
- Think et l'embarqué
- bibliothèques de composants
 - ◆ DREAM, CLIF, Speedo, Perseus, GoTM, ...
- ...
- 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.objectweb.org>
- ◆ Liste de diffusion : fractal@objectweb.org

■ Remerciements

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