Le système de composant Fractal

M. Belguidoum

Université Mentouri de Constantine

Master2 Académique

Plan

1. Concepts de base
   - Composant
   - Interface
   - Liaison
2. Développer avec Fractal
   - Fraclet
   - FractalADL
   - FractalAPI
   - Autres outils
3. Plates-formes
   - Julia
   - AOKell
4. Comparaison
5. Conclusion

Quelques défis

- faire face à la complexité sans cesse croissante des logiciels
- répondre aux grands défis de l'ingénierie des systèmes : passage à grande échelle, administration, autonomie.
- avoir des entités logicielles composables aux interfaces spécifiées contractuellement, déployables et configurables
- avoir des plates-formes à composants suffisamment performantes et légères pour ne pas pénaliser les performances du système

Le modèle de composant Fractal : historique

- défini par France Telecom R&D et l'INRIA.
- un projet du consortium ObjectWeb[1] pour le middleware open source.
- Fractal est un modèle général dédié à la construction, au déploiement et à l'administration (e.g. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation.
- Plusieurs spécification et implémentations en Java, C, C++, SmallTalk, etc.

1. `http://fractal.ow2.org`

Le modèle de composant Fractal : historique

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

Le modèle de composant Fractal : principe

- **composants composites** : pour avoir une vue uniforme des applications à différents niveaux d'abstraction.
- **composants partagés** : pour modéliser les ressources et leur partage, tout en préservant l'encapsulation des composants.
- **capacités d'introspection** : pour observer l'exécution d'un système.
- **capacités de (re)configuration** : pour déployer et configurer dynamiquement un système

Le modèle de composant Fractal : développement

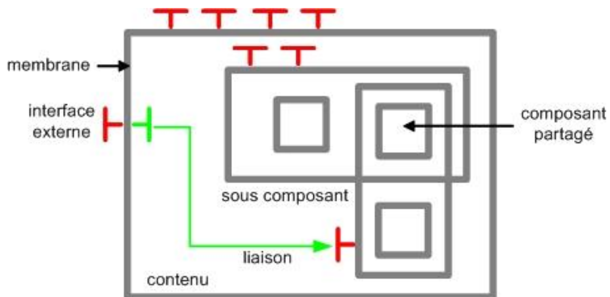
Il existe différents rôles dans les activités de développement autour du modèle Fractal :

- **les développeurs de composants applicatifs** : développent des composants et des assemblages de composants à l'aide de l'API Fractal et du langage de description d'architecture Fractal ADL. Ces composants utilisent des contrôleurs et des plates-formes existants.
- **les développeurs de contrôleurs** : personnalisation du contrôle offerts aux composants. Ils développent de nouvelles politiques de contrôle (extra-fonctionnelles, etc.) et permettant d'adapter les applications à différents contextes d'exécution. Par exemple, mixin pour Julia ou aspect pour AOKell.
- **les développeurs de plates-formes** : fournir une implémentation des spécifications Fractal dans un langage de programmation et des mécanismes pour personnaliser le contrôle. Par exemple : en Java ou C

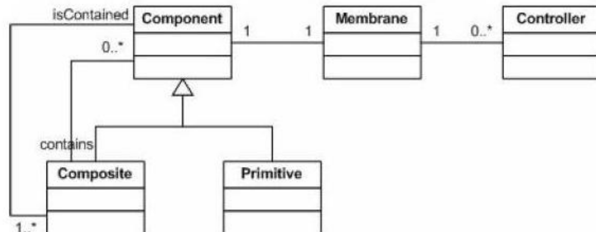
Le modèle de composant Fractal : développement

- un **composant** possède une ou plusieurs **interfaces**.
- une interface est un **point d'accès** au composant, elle implante un type d'interface qui spécifie les opérations supportées par l'interface.
- Il existe deux catégories d'interfaces :
  - les interfaces **serveurs** : services **fournis** par le composant
  - les interfaces **clients** : services **requis** par le composant.
- un composant Fractal est composé de deux parties :
  - une **membrane** qui possède des interfaces fonctionnelles, d'introspection et de configuration (dynamique) du composant
    - Les interfaces d'une membrane sont soit *externes* (accessibles de l'extérieur), soit *internes* (accessibles par les sous-composants).
    - la **membrane** d'un composant est constituée d'un *ensemble* de **contrôleurs**
    - chaque **contrôleur** a un **rôle** particulier
    - **exemples** : contrôler le comportement d'un composant et/ou de ses sous-composants, suspendre/reprendre l'exécution d'un composant, etc.
  - un **contenu** qui est constitué d'un ensemble fini de sous-composants.

Le modèle de composant

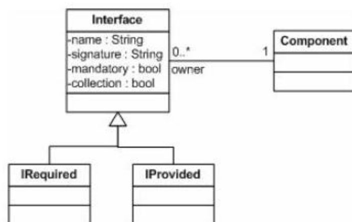


Le modèle de composant Fractal



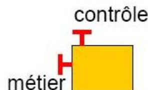
Le modèle de composant Fractal : interfaces

- Définit des services offerts ou requis par un composant
- interface Fractal :
  - nom
  - signature
  - est associée à un composant
  - cliente ou serveur
  - obligatoire ou facultative
  - simple ou multiple (collection)



Le modèle de composant Fractal : interfaces

- Interface : **client** vs **serveur**
  - client : côté gauche des composants
  - serveur : côté droit des composants
- Interface : **métier** vs **contrôle**
  - métier : ce pour quoi l'application est faite (sa finalité 1ère)
  - 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,



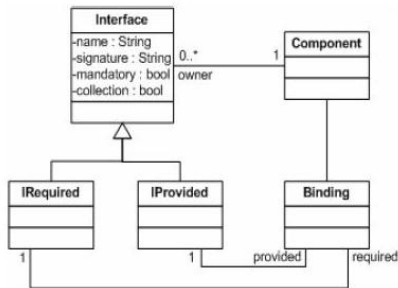
Les contrôleurs

- la spécification Fractal définit différents contrôleurs :
  - **d'attributs** pour configurer les attributs d'un composant.
  - **de liaisons** pour créer/rompre une liaison primitive entre deux interfaces de composants.
  - **de contenu** pour ajouter/retrancher des sous-composants au contenu d'un composant composite.
  - **de cycle de vie** pour contrôler les principales phases comportementales d'un composant. Par exemple, démarrer et stopper l'exécution du composant.
- les développeurs peuvent implémenter leurs propres contrôleurs pour étendre ou spécialiser les capacités réflexives de leurs composants.

Le modèle de composant Fractal : Liaison

- chemin de communication entre composants : 1 interface client et 1 interface serveur
- explicite et matérialise les dépendances entre composants
- Deux types de liaisons :
  - **primitive** : établie entre une interface client et une interface serveur de deux composants résidant dans le même espace d'adressage. Par exemple, une liaison primitive dans le langage C (resp. Java) est implantée à l'aide d'un pointeur (resp. référence).
  - **composite** : le chemin de communication entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (e.g. stub, skeleton) reliés par des liaisons primitives.
- manipulable à l'exécution : reconfiguration dynamique
- sémantique non figée
  - par ex. : invocation méthode locale, distante, diffusion, avec QoS, etc
  - dépend du *binding-controller*

Le modèle de composant Fractal : Liaison



Typage

Principe de substitution

- relation de sous-typage notée $\leq$. (T1 $\leq$ T2 : T1 sous-type de T2)
- composants vus comme des boites noires (en faisant abstraction implémentation)
- type d'un composant : construit a partir du type de ses interfaces
  - $Ti =< Ci, Si >$ // Ci : ensemble d'interfaces client . Si : serveur
  - $T1 \leq T2 \equiv \forall c1 \in C1, \exists c2 \in C2, c1 \leq c2 \wedge$
    $\forall s2 \in S2, \exists s1 \in S1, s1 \leq s2$
  - $s1 \leq s2 \equiv s1.name = s2.name \wedge s1.signature \leq s2.signature \wedge$
    $(s2\ obligatoire \Rightarrow s1 obligatoire) \wedge$
    $(s2\ collection \Rightarrow s1 collection)$
  - $c1 \leq c2 \equiv c1.name = c2.name \wedge c2.signature \leq c1.signature \wedge$
    $(c2\ optionnelle \Rightarrow c1\ optionnelle) \wedge$
    $(c2\ collection \Rightarrow c1\ collection)$

Développer avec Fractal

Développer des applications Fractal en Java

- 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
- 2 étapes
  - définition des composants : avec l'API et/ou avec fraclet
  - définition de l'assemblage : avec l'API et/ou avec Fractal ADL
- Scenarii 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)

Fraclet

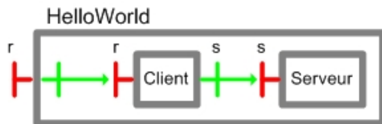
- modèle de programmation Java pour Fractal
- à base d'annotations Java 5 ou Java 1.4 XDoclet
- annotations d'éléments de code (classe, méthode, attribut) : apporte aux éléments une signification en lien avec les concepts Fractal
- phase de pré-compilation : génération du code source associé aux annotations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

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

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`
  - exportée au niveau du composite
- client requiert le service fournit par l'interface s de Serveur



Fraclet : Exemple Hello World

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) ;
  }
}
```

Fraclet : Exemple Hello World

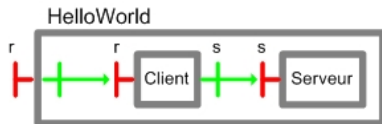
Exemple Hello World : **Le composant Client** :

```
@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!");
} }
```

Fraclet : Exemple 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>
```



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

Fraclet : Autres annotations

- `@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 noeud virtuel pour les communications distantes
- voir `http://fractal.ow2.org/fraclet` pour plus de détails

FractalADL

- 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 (exemple : balise pour indiquer site de déploiement, etc)
- *front-end* pour l'API : génération d'appels à l'API pour construire l'architecture décrite en XML
- description de l'architecture initiale : qui peut toujours évoluer par manipulation avec l'API

FractalADL

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

FractalADL : 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 >
```

FractalADL : 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
>
```

FractalADL : définition de la liaison

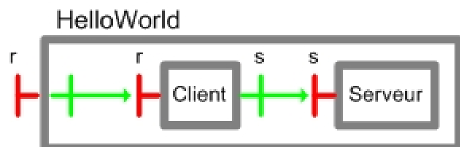
<binding
client = "this.r" interface source
server = "client.r" interface destination
/>

source | destination : := nom composant.nom interface

this : composant courant

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

FractalADL : exemple HelloWorld




```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="Client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="org.objectweb.julia.example.ClientImpl"/>
    <controller desc="primitive"/>
  </component>
  <component name="Server">
    <interface name="s" role="server" signature="Service"/>
    <content class="org.objectweb.julia.example.ServerImpl"/>
    <controller desc="primitive"/>
  </component>
  <binding client="this.r" server="Client.r"/>
  <binding client="Client.s" server="Server.s"/>
  <controller desc="composite"/>
</definition>
```

Exemple HelloWorld

```
import org.objectweb.fractal.api.control.BindingController;

public class Client implements Runnable, BindingController {

  // Implementation de Runnable
  public void run() {
    service.print("Hello world!");
  }

  // Implementation de 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;
}

public class ServerImpl implements Service {
  public void print( String msg) {
    System.err.println(msg);
  }
}
```

HelloWorld : implémentation des composants Client et Server

- La classe `ClientImpl` correspond au composant Client et fournit une implémentation pour l'interface `r` de type `java.lang.Runnable`.
- le composant Client manipule sa liaison avec le composant Server : il implémente l'interface `BindingController` définie dans l'API Fractal.
- la classe `ServerImpl` implémente le composant Server.

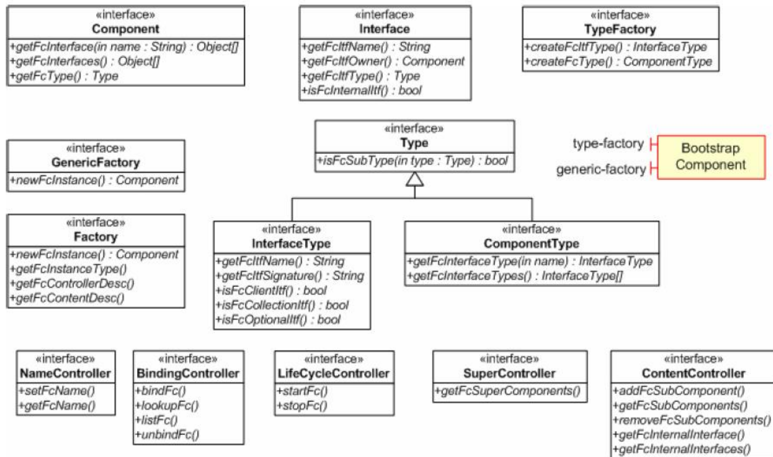
FractalAPI

- modèle dynamique
- les composants et les assemblages sont présents à l'exécution : applications dynamiquement adaptable
- 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)

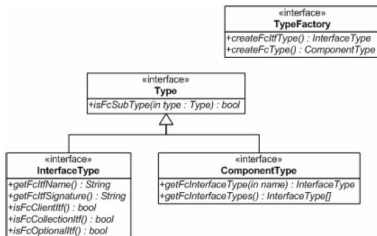
FractalAPI

- 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

FractalAPI



FractalAPI : 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** : un ensembre de types d'interfaces

FractalAPI : 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
});
```

FractalAPI : création de types de composants

Création du type 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)
});
```

FractalAPI : 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;
}
```

FractalAPI : assemblage des composants

- Création des hiérarchies d'imbrication : insertion des composants primitifs dans le composite

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

- Création des liaisons entre les interfaces :

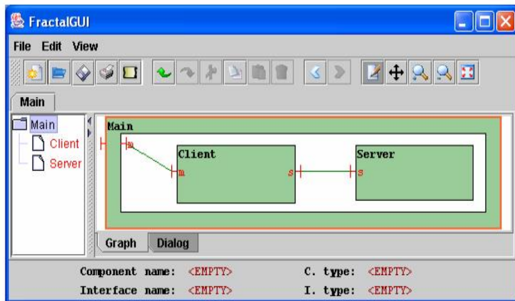
```
Fractal.getBindingController(rComp).bindFc(
        "r", cComp.getFcInterface("r")) ;
Fractal.getBindingController(cComp).bindFc(
        "s", sComp.getFcInterface("s")) ;
```

- 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() ;
```

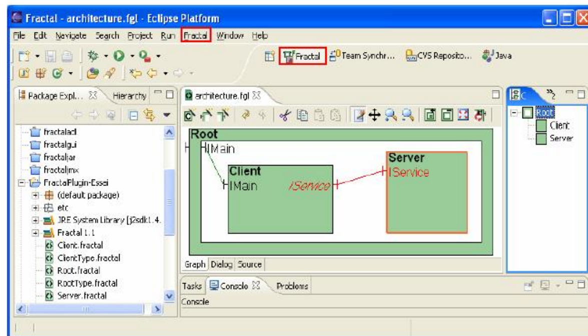
Autres outils : FractalGUI

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



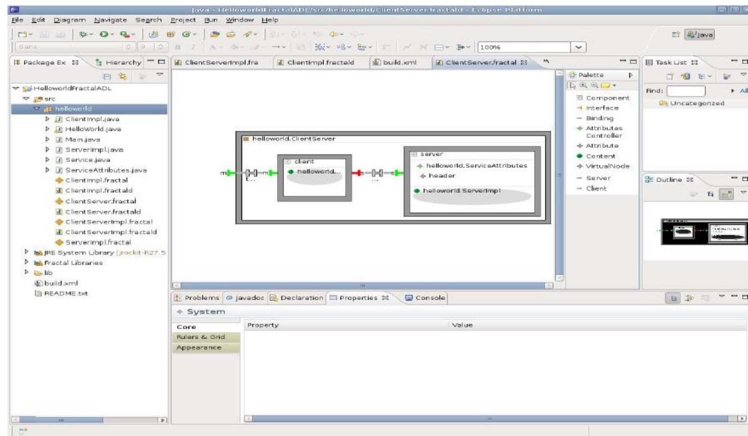
Autres outils : FractalGUI-Plugin Eclipse

FractalGUI-Plugin Eclipse : intégration FractalGUI dans Eclipse



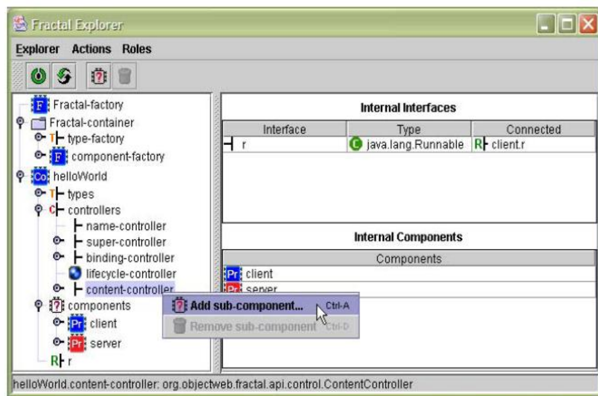
Autres outils : F4E Fractal for Eclipse

F4E Fractal for Eclipse : environnement Eclipse de développement d'applications Fractal



Autres outils : FractalExplorer

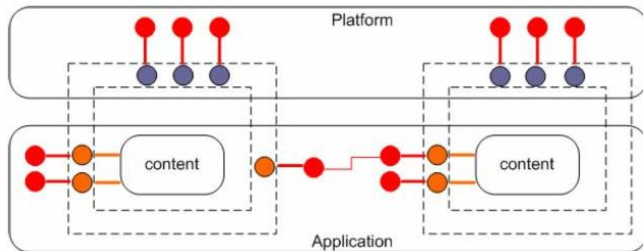
- console d'administration
- pilotage (run-time) d'une application Fractal



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)
- différentes implémentations pour differents besoins

Plates-formes



Plates-formes : Julia

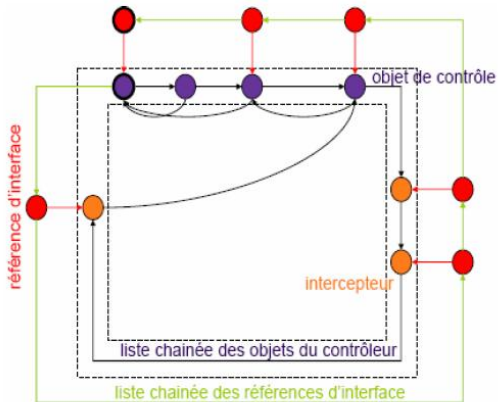
- implémentation de référence du modèle Fractal [2]
- canevas logiciel écrit en Java qui permet de programmer les membranes des composants.
- framework extensible pour programmer des contrôleurs que l'utilisateur peut assembler.
- implémenter des objets de contrôle de façon à minimiser en priorité
  - le surcoût en temps d'exécution des composants
  - le surcoût en mémoire sur les applications

2. fractal.ow2.org/julia

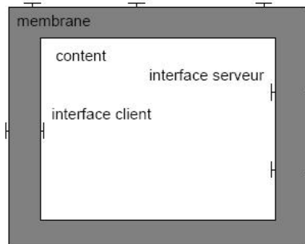
Julia : principales structures de données

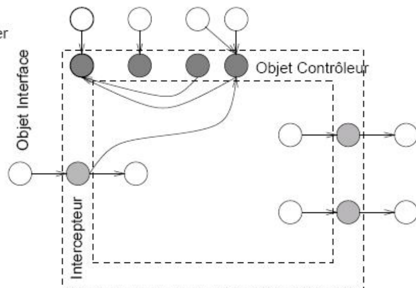
- Un composant Fractal est formé de plusieurs objets Java
  - Les objets qui implémentent le contenu du composant. Ils peuvent être des sous-composants (dans le cas de composants composites) ou des objets Java (pour les composants primitifs).
  - Les objets qui implémentent la partie de contrôle du composant (représentés en gris) :
    - les objets implémentant les interfaces de contrôle
    - les intercepteurs optionnels qui interceptent les appels de méthodes entrants et sortants.
    - les contrôleurs et les intercepteurs ont références les uns vers les autres.
  - Les objets qui référencent les interfaces du composant (en blanc). Ces objets sont le seul moyen pour un composant de posséder des références vers un autre composant.
- La mise en place de ces différents objets est effectuée par des fabriques de composants. Celles-ci fournissent une méthode de création qui prend en paramètres la description des parties fonctionnelle et de contrôle du composant.

Julia



Un composant Fractal et son implantation Julia





Julia : configuration

Descripteurs de contrôle (type de membrane)

- primitive, composite, parametric, template, etc.
- 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

Julia : exemple de configuration

```
(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

Julia : développement des contrôleurs

- possibilité de construire diverses formes de contrôleurs et diverses sémantiques de contrôle. différentes implantations de l'interface BindingController selon les vérifications qu'elles font lors de la création/destruction d'une liaison :
  - interaction avec le contrôleur de cycle de vie pour :
    - vérifier qu'un composant est stoppé
    - vérification que les types d'interface sont compatibles quand un système de types est utilisé,
    - vérification que les composants liés sont parents d'un même composite quand les contrôleurs de contenu sont utilisés, etc.
- pas d'héritage de classes car cela conduit à une explosion combinatoire
  - il faut $2^3 = 8$ classes pour vérifier le système de types, le cycle de vie et le contrôleur de contenu
  - **problème** : beaucoup de duplications de code.
  - **solution** : utilisation de classes *mixin* [Bracha and Cook 1990]

Julia : développement des contrôleurs

Une classe *mixin*

est une classe dont la super-classe est spécifiée de manière abstraite en indiquant les champs et méthodes que cette super-classe doit posséder. Elle peut s'appliquer (c'est-à-dire surcharger et ajouter des méthodes) à toute classe qui possède les caractéristiques de cette super-classe. Ces classes *mixin* sont appliquées au chargement à l'aide de l'outil ASM [ASM 2002].

- Dans Julia, les classes mixin sont des classes abstraites développées avec certaines conventions.
- Elles ne nécessitent pas l'utilisation d'un compilateur Java modifié ou d'un pré-processeur comme c'est le cas des classes mixins développées à l'aide d'extensions du langage Java.

Julia : mécanisme de mixin

- Mécanisme de mixin (Bracha 90) inspiré de JAM [3]
  - construction d'une classe en fusionnant des méthodes provenant de différentes classes
  - similitudes avec le mécanisme de classes partielles dans C♯ v2.0
  - chaque classe definit des methodes différentes donc pas de pb pour fusionner
  - méthodes avec meme signature : mecanisme de chaînage pour concaténer le code des différentes 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

3. www.disi.unige.it/person/LagorioG/jam

Julia : exemple de mixin

- la classe mixin (gauche) s'écrit en Julia en pur Java (droite).
- le mot clé `inherited` en JAM est équivalent au préfixe `super` utilisé dans Julia.
- `super` spécifie les méthodes qui sont surchargées par le mixin.
- les méthodes qui sont requises mais pas surchargées sont spécifiées à l'aide du préfixe `this` .

```
mixin Compteur {
    inherited public void m ();
    public int count;
    public void m () {
        ++count;
        super.m();
    }
}
```

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

Julia : développement des intercepteurs

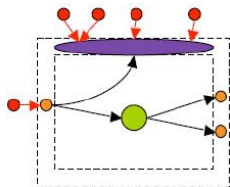
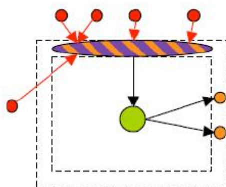
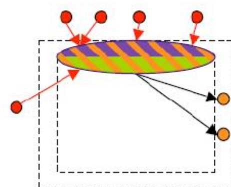
- possibilité de développer des intercepteurs dont le rôle est d'intercepter les appels de méthode entrant et/ou sortant des interfaces d'un composant.
- Les intercepteurs doivent implémenter les interfaces interceptées.
- il est inconcevable de développer, pour un aspect de contrôle donné, autant d'intercepteurs qu'il y a d'interfaces à intercepter dans l'application.
- En conséquence, Julia fournit un outil, appelé générateur d'intercepteurs, qui permet de générer dynamiquement le code de ces intercepteurs.
- Par exemple : des blocs de code a exécuter avant et aprés l'interception.

Julia : optimisation

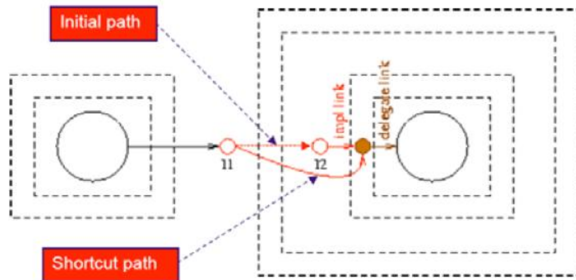
- Deux mécanismes d'optimisation : intra et inter composants
  - intra-composant permet de réduire l'empreinte mémoire d'un composant en fusionnant une partie de ses objets de contrôle. Pour ce faire, Julia fournit un outil utilisant ASM (ASM 2002) et imposant certaines contraintes sur les objets de contrôle fusionnés : par exemple, deux objets fusionnés ne peuvent pas implémenter la même interface.
  - inter-composant permet d'optimiser les chaînes de liaison entre composants : il permet de court-circuiter les parties contrôle des composites qui n'ont pas d'intercepteurs. Chaque interface serveur de composant est représentée par un objet qui contient une référence vers un objet implantant réellement l'interface.

Julia : optimisation intra-composants

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

Julia : optimisation inter-composants



Julia : organisation modulaire de la plate-forme

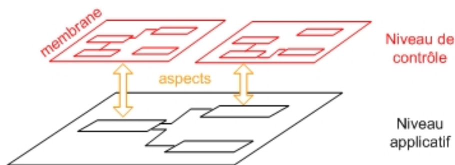
- **fractal-api** : API Fractal (partagée avec autres plates-formes)
- **julia-runtime** : API interne Julia
- **julia-asm** : framework de génération de bytecode
  - basé sur la librairie de manipulation de bytecode ASM
  - fournit un mécanisme de programmation par mixins
- **julia-mixins**
  - un ensemble de mixins mettant en oeuvre la personnalité correspondant à la sémantique de référence du modèle Fractal
  - développement d'une nouvelle personnalité : développement (réutilisation, extension, etc.) d'un nouveau module julia-mixins

AOKell

- Comme Julia, le canevas logiciel AOKell [Seinturier et al. 2006] est une implémentation complète des spécifications Fractal.
- AOKell diffère de Julia sur deux points :
  - l'intégration des fonctions de contrôle dans les composants est réalisée à l'aide **d'aspects**
  - les contrôleurs sont implémentés eux-mêmes sous forme de composants.
- l'objectif d'AOKell est de simplifier et de réduire le temps de développement de nouveaux contrôleurs et de nouvelles membranes par rapport à Julia qui utilise un mécanisme de mixin et de la génération de bytecode à la volée avec ASM

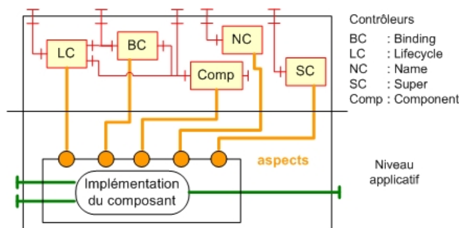
AOKell : membranes componentisées

- les concepts de composant, d'interface et de liaison sont utilisées pour concevoir le niveau applicatif et le niveau de contrôle.
- Une membrane AOKell est un assemblage exportant des interfaces de contrôle et contenant des sous-composants. Chacun d'eux implémente une fonctionnalité de contrôle particulière.
- chaque composant de contrôle est aussi associé à un aspect qui intègre cette logique de contrôle dans les composants de niveau applicatif.



AOKell : membrane de contrôle

- membrane des composants primitifs fournit cinq contrôleurs pour gérer le cycle de vie (LC), les liaisons (BC), le nommage (NC), les références vers les composants parents (SC) et les caractéristiques communes à tout composant Fractal (Comp).
- la fonction de contrôle des composants primitifs est le résultat de la coopération de ces cinq contrôleurs



AOKell : Intégration des contrôleurs à l'aide d'aspects

- chaque contrôleur est associé à un aspect chargé de l'intégration de la logique du contrôleur dans le composant.
- La logique d'intégration repose sur deux mécanismes : l'injection de code et la modification de comportement
  - l'injection de code : (connu dans AspectJ par déclaration inter-type (ITD [4]) les méthodes des interfaces de contrôle sont injectées dans les classes implémentant les composants. Le code injecté est constitué d'une souche qui délègue le traitement à l'objet implémentant le contrôleur.
  - la modification de comportement (AspectJ à *advice*) la définition d'un aspect est constituée de coupes et de code advice. Les coupes sélectionnent un ensemble de points de jonction qui sont des points dans le flot d'exécution du programme autour desquels l'aspect doit être appliquée. Le code advice est utilisé dans AOKell pour intercepter les appels et les exécutions des opérations des composants.
- les aspects intégrent de nouvelles fonctionnalités dans les composants et contrôlent leur exécution.

4. Inter-Type Declaration

Comparaison avec les initiatives industrielles

- **Initiatives industrielles**
  - **Exemple** : EJB, COM+/.NET, CCM, OSGi
  - **Caractéristiques**
    - COM+ composants avec des propriétés d'introspection
    - OSGi et CCM composants avec liaisons et cycle de vie
    - EJB, CCM et COM+/.NET possédent un modèle figé de services techniques offerts par des conteneurs aux composants
  - **Comparaison avec Fractal**
    - Fractal est *reflexif* et *introspectable*, autorise des liaisons selon différentes sémantiques de communication
    - Fractal fournit un modèle hiérarchique autorisant le partage
    - Fractal est basé sur un modèle ouvert, dans lequel les services techniques sont entièrement programmables via la notion de contrôleur

Comparaison avec les initiatives industrielles

- **SCA** [5] [SCA 2005] a défini un modèle de composants pour des architectures orientées services.
- Le projet Tuscany [Tuscany 2006] fournit une implémentation en Java et en C++ de ces spécifications.
- SCA propose la notion de liaison pour l'assemblage et de module pour la création de hiérarchies de composants.
- SCA n'impose pas une forme prédéterminée de liaison, mais autorise l'utilisation de différentes technologies, comme SOAP, JMS ou IIOP pour mettre en œuvre ces liaisons.
- De même, Fractal autorise différents types de liaisons et n'impose pas de technologie particulière.
- FraSCAti est un framework basé sur les composants pour l'exécution de SCA (OW2FraSCAti 1.1.1 disponible le 24 Nov 2009)

5. Service Component Architecture

Comparaison avec les logiciels libres

- **Logiciels libre**
  - **Exemple** : Avalon, Kilim, Pico et Hivemind qui ciblent la configuration de logiciel, Spring, Carbon et Plexus qui ciblent les conteneurs de composants de type EJB.
  - **Comparaison avec Fractal**
    - De manière générale, ces modèles sont moins ouverts et extensibles que ne l'est Fractal.

Comparaison avec les initiatives académiques

- **Initiatives académiques**
  - **Exemple** : ArchJava, FuseJ, KComponent, OpenCOM v1 et v2.
  - **Caractéristiques**
    - OpenCOM cible les systèmes devant être reconfigurés dynamiquement et en particulier les systèmes d'exploitation, les intergiciels, les PDA et les systèmes embarqués.
    - Au niveau applicatif, les composants OpenCOM fournissent des interfaces et requièrent des réceptacles.
    - L'architecture d'une application OpenCOM est introspectable et peut être modifiée dynamiquement.
  - **Comparaison avec Fractal**
    - OpenCOM est le modèle le plus proche de Fractal.
    - Les binders et les loaders dans OpenCOM sont comparables aux contrôleurs de Fractal.
    - Les contrôleurs Fractal ne sont pas limités à ces deux types de propriétés extrafonctionnelles et peuvent englober d'autres services techniques.
    - Fractal/AOKell permet de réifier l'architecture de contrôle sous la forme d'un assemblage de composants.

Conclusion

- Fractal est un modèle hiérarchique (primitifs ou composites)
- Deux parties dans un composant Fractal : le contenu et la membrane.
- La membrane fournit un niveau méta de contrôle et de supervision du contenu. Elle est composée de contrôleurs, qui implémentent des interfaces de contrôle.
- Fractal est ouvert car de nouveaux contrôleurs peuvent être ajoutés par les développeurs en fonction des besoins.
- Un composant Fractal possède des interfaces fournies (dite serveur) et/ou des interfaces requises (dites client).
- Le concept de liaison permet de définir des chemins de communication entre interfaces clientes et serveurs.
- Le modèle Fractal est indépendant des langages de programmation.
- Fractal ADL est un langage ouvert basé sur XML, sa DTD peut être étendue avec de nouvelles balises. Un mécanisme d'extension permet de traiter ces nouvelles balises.

Conclusion

- Julia et AOKell sont la mise en oeuvre de Fractal en Java.
- D'autres plates-formes existent pour les langages Smalltalk, C, C++ et les langages de la plate-forme .NET.
- Julia est la plate-forme de référence du modèle Fractal.
- Le développement des contrôleurs se fait à l'aide d'un système de classes *mixin*.
- AOKell utilise des techniques issues de la programmation orientée aspect pour l'intégration des niveaux de base et de contrôle.
- Plusieurs bibliothèques pour les développeurs Fractal :
  - Dream, qui permet de développer des intergiciels.
  - Fractal RMI qui permet de construire des assemblages de composants distribués communicants via un mécanisme d'invocation de méthodes à distance
  - Fractal JMX pour l'administration de composants à l'aide de la technologie JMX.

Quelques Outils pour Fractal

- 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)

Quelques Outils pour Fractal

- Some operational usages : Jonathan, Jabyce, Dream, Perseus, Speedo, JOnAS (persistence), GoTM, CLIF, etc
- Dissemination in industry (FT, STM, Nokia), universities including teaching (Grenoble, Chambéry, Nantes, etc), conferences (JC, LMO, SC, Euromicro, etc.)

Références

- T. Coupaye, V.Quema, L. Seinturier, J.B. Stefani : Le système de composant Fractal, chapitre 3 Intergiciel et Construction d'applications réparties
- Site Web : http://fractal.ow2.org
- des figures et des transparents empruntés de L. Seinturier