Le modèle de composants Fractal

Eric Bruneton

Plan

- **Introduction**
- **Premier aperçu**
- **Définition du modèle**
- **Déploiement des composants**
- **Langage de description d'architecture**
- **Reconfiguration dynamique**
- **Implémentation de référence**
- **Conclusion**

Introduction : enjeux liés aux plate-formes

- **Maîtrise**
  - du déploiement
  - de l'administration
  - de l'évolution
  - de l'intégration
- **d'infrastructures logicielles**
  - réparties
  - hétérogènes
  - et en parties patrimoniales (legacy)
- **en prenant en compte les besoins applicatifs (QoS)**

Introduction : objectifs

- **Principes architecturaux pour l'ingénierie des systèmes**
  - Construction par composition de briques logicielles : les composants
  - Vision homogène de la topologie des systèmes logiciels
    - services & applications, intergiciel (middleware), systèmes d'exploitation
  - Gestion uniforme des ressources, des activités et des domaines de contrôle
  - Couverture du cycle de vie complet :
    - développement, déploiement, supervision
- **Réalisation d'un support d'exécution pour composants**
  - Support d'exécution pouvant être
    - spécialisé, outillé, étendu
    - composé avec d'autres canevas : persistance, réplication,sécurité, …

Introduction : objectifs

- **Application aux plate-formes middleware d'ObjectWeb**
  - Configuration, intégration des composants ObjectWeb
  - Composants communs, partage de ressources entre composants :
    - Transaction Manager, Pool, Cache, Logger, …
  - Outils de développement communs : configuration, supervision, …
- **Recherche et développement**
  - Techniques de composition sûres et efficaces :
    - Assertions, contrats, méthodes formelles, …
  - Intégrité des reconfigurations dynamiques
  - Disponibilité et tolérance aux fautes: persistance, duplication, mobilité, …

Plan (rappel)

- **Introduction**
- **Premier aperçu**
- **Définition du modèle**
- **Déploiement des composants**
- **Langage de description d'architecture**
- **Reconfiguration dynamique**
- **Implémentation de référence**
- **Conclusion**

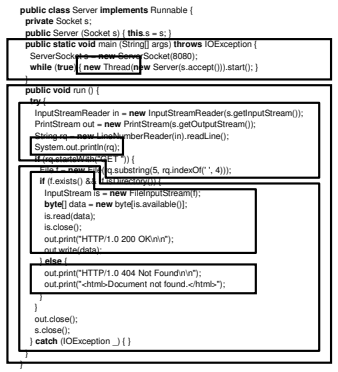
Aperçu : l'exemple Comanche

- **Serveur HTTP minimal**
  - écoute sur une *server socket*
  - pour chaque connexion:
    - lance un nouveau *thread*
    - lit l'URL
    - envoie le fichier demandé
    - ou un message d'erreur

```
public class Server implements Runnable {
  private Socket s;
  public Server (Socket s) { this.s = s; }
  public static void main (String[] args) throws IOException {
    ServerSocket s = new ServerSocket(8080);
    while (true) { new Thread(new Server(s.accept())).start(); }
  }
  public void run () {
    try {
      InputStreamReader in = new InputStreamReader(s.getInputStream());
      PrintStream out = new PrintStream(s.getOutputStream());
      String rq = new LineNumberReader(in).readLine();
      System.out.println(rq);
      if (rq.startsWith("GET ")) {
        File f = new File(rq.substring(5, rq.indexOf(' ', 4)));
        if (f.exists() && !f.isDirectory()) {
          InputStream is = new FileInputStream(f);
          byte[] data = new byte[is.available()];
          is.read(data);
          is.close();
          out.print("HTTP/1.0 200 OK\r\n\r\n");
          out.write(data);
        } else {
          out.print("HTTP/1.0 404 Not Found\r\n\r\n");
          out.print("<html>Document not found.</html>");
        }
      }
      out.close();
      s.close();
    } catch (IOException _) { }
  }
}
```


Conception : trouver les composants

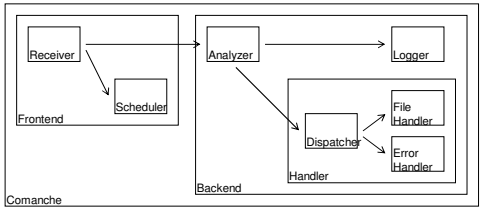
- **Composants statiques**
  - durée de vie
    - celle de l'application
  - correspondent aux « services »
- **Composants dynamiques**
  - durée de vie plus courte
  - correspondent aux « données »
- **Définition des composants**
  - trouver
    - les services
    - [les structures de données]
  - un service = un composant



Conception : définir l'architecture

➔**Dépendances :**
- utiliser des scénarios et des cas d'utilisation

➔**Structure hiérarchique :**
- correspond au niveau d'abstraction



Conception : définir les contrats

➔**Contrats entre composant :**
- doivent être définis avec attention
  - pour être le plus stable possible
- ne doivent concerner que les aspects fonctionnels

➔**Contrats pour Comanche**
- logger : une opération `log(String)`
- scheduler : une opération `schedule(Runnable)`
- handlers : une opération `handleRequest`
  - analyzer : lit l'URL
  - dispatcher : essaie les différents handlers, jusqu'à un succès
  - file handler : essaie de lire et de renvoyer le fichier à l'URL indiquée
  - error handler : retourne un message d'erreur

Implémentation : interfaces

➔**Choix de la granularité des composants**
- plusieurs composants identifiés à la conception peuvent être implémentés sous la forme d'un seul composant

➔**Implémentation des interfaces de composant**
- Fractal impose une séparation stricte entres les interfaces et leurs implémentations

➔**Interfaces de Comanche**
- public interface Logger { void log (String msg); }
- public interface Scheduler { void schedule (Runnable task); }
- public interface RequestHandler { void handleRequest (Request r) throws IOException; }
- public class Request { Socket s; Reader r; PrintStream out; String url; }

Implémentation : composants

➔**Composants sans dépendances :**
- implémentés comme en Java standard

```
public class BasicLogger implements Logger {
  public void log (String msg) { System.out.println(msg); }
}
public class SequentialScheduler implements Scheduler {
  public synchronized void schedule (Runnable task) { task.run(); }
}
public class MultiThreadScheduler implements Scheduler {
  public void schedule (Runnable task) { new Thread(task).start(); }
}
```

Implémentation : composants

➔**Composants avec dépendances :**

➢première solution : ne permet pas la configuration statique

```
public class RequestReceiver implements Runnable {
  private Scheduler s = new MultiThreadScheduler();
  private RequestHandler rh = new RequestAnalyzer();
  // rest of the code not shown
}
```

➢deuxième solution: ne permet pas la reconfiguration dynamique

```
public class RequestReceiver implements Runnable {
  private Scheduler s;
  private RequestHandler rh;
  public RequestReceiver (Scheduler s, RequestHandler rh) { this.s = s; this.rh = rh; }
  // rest of the code not shown
}
```


Implémentation : composants

➔**Composants avec dépendances :**

```
public class RequestReceiver implements Runnable, BindingController {
  private Scheduler s;
  private RequestHandler rh;
  public String[] listFc () { return new String[] { "s", "rh" }; }
  public Object lookupFc (String n) {
    if (n.equals("s")) { return s; } else if (n.equals("rh")) { return rh; } else return null;
  }
  public void bindFc (String n, Object v) {
    if (n.equals("s")) { s = (Scheduler)v; } else if (n.equals("rh")) { rh = (RequestHandler)v; }
  }
  public void unbindFc (String n) {
    if (n.equals("s")) { s = null; } else if (n.equals("rh")) { rh = null; }
  }
  // …
```


Configuration : par programme

➔**Avantages :**

➢méthode la plus directe

➢aucun outils requis

➔**Inconvénients :**

➢propices aux erreurs

➢ne montre pas l'architecture

➢*mélange les aspects architecture et déploiement*

```
public class Server {
  public static void main (String[] args) {
    RequestReceiver rr = new RequestReceiver();
    RequestAnalyzer ra = new RequestAnalyzer();
    RequestDispatcher rd = new RequestDispatcher();
    FileRequestHandler frh = new FileRequestHandler();
    ErrorRequestHandler erh = new ErrorRequestHandler();
    Scheduler s = new MultiThreadScheduler();
    Logger l = new BasicLogger();
    rr.bindFc("rh", ra);
    rr.bindFc("s", s);
    ra.bindFc("rh", rd);
    ra.bindFc("l", l);
    rd.bindFc("h0", frh);
    rd.bindFc("h1", erh);
    rr.run();
  }
}
```


Configuration : avec un ADL

➔**Avantages :**

➢bonne séparation des aspects architecture et déploiement

➢vérifications statiques possibles

- liaisons invalides,
- liaisons manquantes,
- …

➔**Inconvénients :**

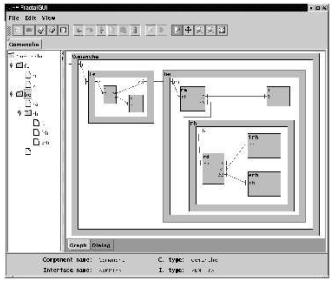
➢l'architecture n'est pas très visible

```
<component-type name="HandlerType">
 <provides>
  <interface-type name="rh" signature="comanche.RequestHandler"/>
 </provides>
</component-type>

<primitive-template name="FileHandler" implements="HandlerType">
 <primitive-content class="comanche.FileRequestHandler"/>
</primitive-template>

<composite-template name="Comanche" implements="RunnableType">
 <composite-content>
  <components>
   <component name="fe" type="FrontendType" implementation="Frontend"/>
   <component name="be" type="HandlerType" implementation="Backend"/>
  </components>
  <bindings>
   <binding client="this.r" server="fe.r"/>
   <binding client="fe.rh" server="be.rh"/>
  </bindings>
 </composite-content>
</composite-template>
```

Configuration : avec un outil graphique

➔**Outils graphique pour éditer des fichiers ADL**

➢la représentation en graphe montre clairement l'architecture



Reconfiguration : cycle de vie

➔**Reconfiguration :**

➢statique : stopper l'application, changer l'ADL, redémarrer

●pas toujours possible, par exemple pour raisons de disponibilité

➢dynamique : reconfigurer l'application pendant son exécution

●pose des problèmes de cohérence

–outil de base pour aider à les résoudre : gestion du cycle de vie

➔**Exemple :**

➢remplacer le composant `FileHandler` dynamiquement :

```
-RequestHandler rh = new FileAndDirRequestHandler();
-rd.unbindFc("h0");
-rd.bindFc("h0", rh);
```

●le `RequestDispatcher` doit être suspendu pendant l'opération

Reconfiguration : cycle de vie

➔**Interface `LifeCycleController` : 1ère solution**

```
public class RequestDispatcher implements RequestHandler, BindingController, LifeCycleController {
  private boolean started;
  private int counter;
  public String getFcState () { return started ? STARTED : STOPPED; }
  public synchronized void startFc () { started = true; notifyAll(); }
  public synchronized void stopFc () { while (counter > 0) { wait(); } started = false; }
  public void handleRequest (Request r) throws IOException {
    synchronized (this) { while (counter == 0 && !started) { wait(); } ++counter; }
    try {
      // original code
    } finally { synchronized (this) { --counter; if (counter == 0) { notifyAll(); } } }
  }
  // rest of the class unchanged
```


Reconfiguration: cycle de vie

➔**Interface `LifeCycleController` : autres solutions**

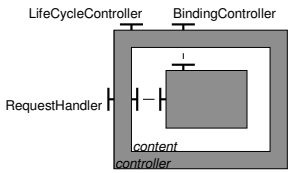
➢implémenter l'interface dans une (sous) classe séparée

●meilleure séparation des aspects

–composants déployables avec ou sans gestion du cycle de vie

➢générer cette classe ou cette sous classe automatiquement

●génération de code statique ou dynamique



Reconfiguration : autres APIs

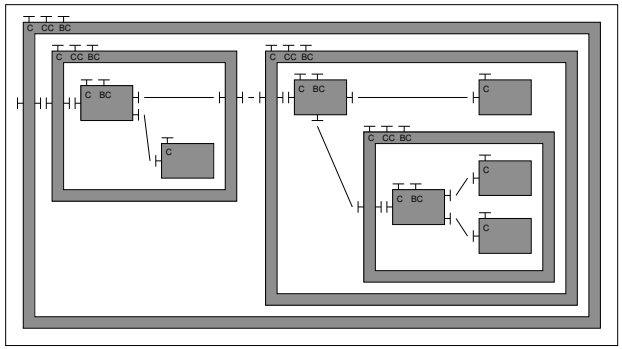
➔**Avant de stopper ou de reconfigurer un composant, il faut obtenir sa référence**

- ➢d'où les interfaces d'introspection (et de reconfiguration) suivantes :

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

```
public interface ContentController {
  Object[] getFcInternalInterfaces ();
  Object getFcInterfaceInterface(String itfName);
  Component[] getFcSubComponents ();
  void addFcSubComponent (Component c);
  void removeFcSubComponent(Component c);
}
```


Quelques choix de déploiement



Aperçu: conclusion

➔**Fractal**

- ➢utilise des patrons de conception bien connus et les organise en un modèle à composants uniforme, extensible et indépendant des langages

➔**Avantages**

- ➢impose la séparation entre interfaces et implémentations
  - ●assure un niveau de flexibilité minimum
- ➢impose la séparation des aspects fonctionnels, d'architecture et de déploiement
  - ●permet d'instancier une application de plusieurs façons
    - –depuis une configuration très optimisée mais non reconfigurable jusqu'à une configuration moins performante mais dynamiquement reconfigurable

Plan (rappel)

Modèle : fondations

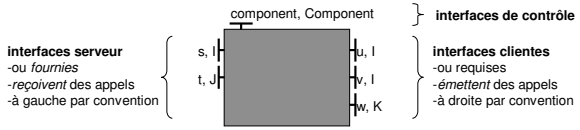
➔**Composant Fractal :**

- entité *à l'exécution*
  - par exemple, en Java, des *objets* et non pas des classes ou des .jar
- de n'importe quel type
  - service, ressource, donnée, activité, processus, protocole, liaison, …
- sans cible ni granularité particulière
  - application, intergiciel (*middleware*), système d'exploitation

Modèle : structure externe

➔**De l'extérieur, un composant Fractal :**

- est une boite noire,
- avec des points d'accès appelés *interfaces (externes)*
  - chaque interface à un *nom* (unique dans le composant)
  - une *signature* (ensemble d'*opérations* – i.e. méthodes en Java)
  - un *rôle* (client, serveur ou contrôle)



Modèle : structure externe

➔**Composant de base :**

- aucune interface de contrôle => pas d'introspection
- équivalent à un objet au sens des langages orienté objets

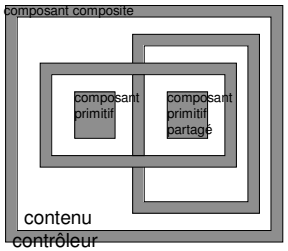
➔**Premiers niveaux de contrôle :**

- introspection des composants :
  - interface `Component`
  - donne accès à la liste des interfaces d'un composant
- introspection des interfaces :
  - interface `Interface`
  - donne accès au nom d'une interface
  - donne accès à l'interface `Component` du composant

Modèle : structure interne

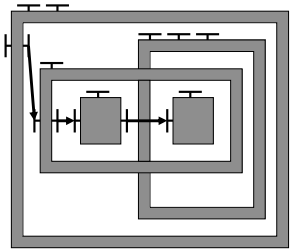
➔**A l'intérieur, un composant :**

- à un *contrôleur* et un *contenu*
- contenu = d'autres composants
  - notion de *sous composant*,
  - composants primitifs et composites,
  - composants partagés



- le contrôleur peut (entre autres) :
  - fournir des fonctions d'introspection et de reconfiguration
  - intercepter les appels d'opération entrants et sortants
    - –pour les aspects non fonctionnels (cycle de vie, transactions, sécurité, …)
  - modifier le comportement de ses sous contrôleurs

Modèle : structure interne



➔**liaisons : interactions**

- ➢liaison primitive
  - ●lie
    - –une interface cliente
    - –à une interface serveur
  - ●dans le même espace d'adressage
- ➢liaison composite
  - ●lie un nombre arbitraire d'interfaces
  - ●est un ensemble de liaisons primitives et de composants de liaisons
- ➢liaisons normales, d'import et d'export

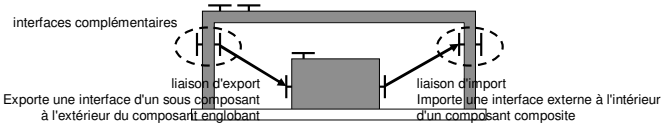
Modèle : structure interne

➔**Interfaces internes**

- ➢interfaces du contrôleur
- ➢visibles seulement des sous composants

➔**Interfaces complémentaires**

- ➢deux interfaces internes et externes
- ➢de même nom et même signature, mais de rôle opposés



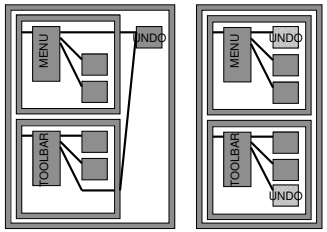
Modèle : structure interne

➔**Récursion :**

- ➢gestion à grain arbitraire :
  - ●passage à l'échelle,
  - ●homogénéité

➔**Partage :**

- ➢préserve l'encapsulation
- ➢permet de modéliser des
  - ●ressources : données, pools, caches, connections, …
  - ●activités : *threads*, transactions, procédés / tâches, …
  - ●domaines de contrôle : défaillance, sécurité, persistance, mobilité, …

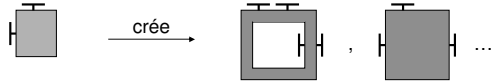


sans partage avec partage

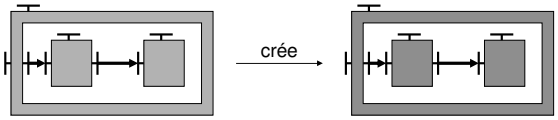
Modèle : création

➔**Trois méthodes possibles**

- ➢`GenericFactory` : crée n'importe quel type de composant

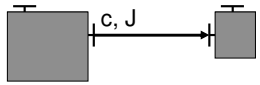


- ➢`Factory` : crée des composants tous de même type
- ➢template : crée des composants similaires à lui même



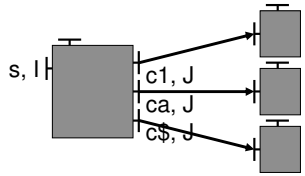
Modèle : typage

➔**Type de composant**
- ➢ensemble de types d'interface
- ➢*non modifiable* à l'exécution



c : interface de type singleton

➔**Type d'interface**
- ➢nom
- ➢signature
- ➢contingence
  - ●obligatoire, optionnelle
- ➢cardinalité
  - ●singleton, collection



c : interface de type collection

Modèle: options

➔***Tout* est optionnel**
- ➢introspection, interfaces de contrôle, de fabriques, typage, …

➔**et extensible**
- ➢sous composants dans les contrôleurs, typage alternatif, …

➔**Niveaux de conformité :**

	C	I	Typage	(Re)configuration	Fabriques	Templates
0						
0.1				x		
1	x					
1.1	x			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

Plan (rappel)

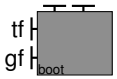
➔**Introduction**
➔**Premier aperçu**
➔**Définition du modèle**
➔**Déploiement des composants**
➔**Langage de description d'architecture**
➔**Reconfiguration dynamique**
➔**Implémentation de référence**
➔**Conclusion**

Déploiement : avec l'API

```
Component boot = Fractal.getBootstrapComponent();
```

```
class Fractal
  static getBootstrapComponent
```

```
interface Component
  Interface[] getFcInterfaces
  Interface getFcInterface
  getFcType
```

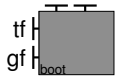



```
interface Interface
  Component getFcItfOwner
  getFcItfName
  getFcItfType
  isFcInternalItf
```

Déploiement : avec l'API

```
ComponentIdentity boot = Fractal.getBootstrapComponent();
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
```

TypeFactory
- createFcInterfaceType
- createFcType

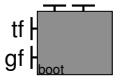


Déploiement : avec l'API

```
Component boot = Fractal.getBootstrapComponent();
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
ComponentType rType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("m", "Main", false, false, false)});
```

ComponentType extends Type
- getFcInterfaceTypes
- getFcInterfaceType

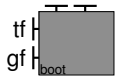
InterfaceType extends Type
- getFcItfName
- getFcItfSignature
- isFcClientItf
- isFcOptionalItf
- isFcCollectionItf



Déploiement : avec l'API

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

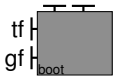
```
ComponentType sType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("s", "Service", false, false, false),
  tf.createFcItfType("attribute-controller", "ServiceAttributes",…)});
```



Déploiement : avec l'API

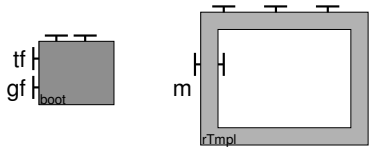
```
GenericFactory gf = Fractal.getGenericFactory(boot);
```

GenericFactory
- newFcInstance



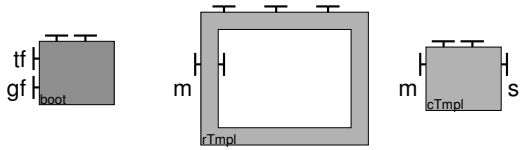
Déploiement : avec l'API

```
GenericFactory gf = Fractal.getGenericFactory(boot);
Component rTmpl =
  gf.newFcInstance(
    rType, "compositeTemplate", new Object[] { "composite", null});
```



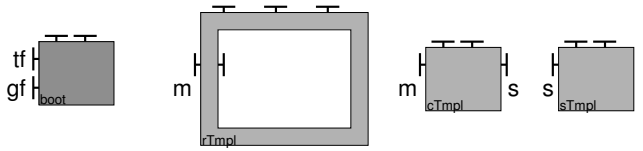
Déploiement : avec l'API

```
Component cTmpl =
  gf.newFcInstance(
    cType, "template", new Object[] {"primitive", "ClientImpl"});
```



Déploiement : avec l'API

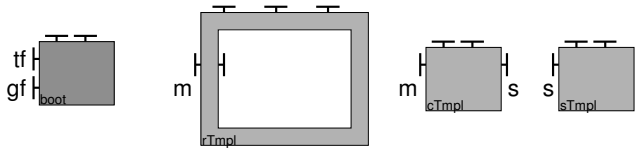
```
Component cTmpl =
  gf.newFcInstance(
    cType, "template", new Object[] {"primitive", "ClientImpl"});
Component sTmpl =
  gf.newFcInstance(sType,
    "parametricTemplate", new Object[] { "primitive", "ServerImpl"});
```



Déploiement : avec l'API

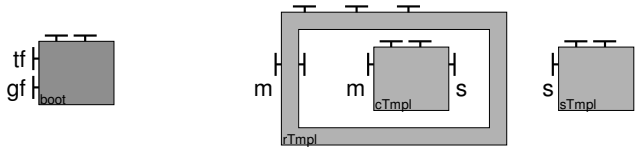
```
Fractal.getContentController(rTmpl).addFcSubComponent(cTmpl);
```

ContentController
- getFcInternalInterfaces
- getFcInternalInterface
- getFcSubComponents
- addFcSubComponent
- removeFcSubComponent



Déploiement : avec l'API

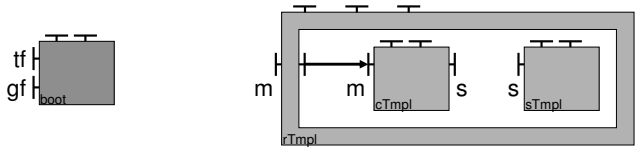
```
Fractal.getContentController(rTmpl).addFcSubComponent(cTmpl);
Fractal.getContentController(rTmpl).addFcSubComponent(sTmpl);
```



Déploiement : avec l'API

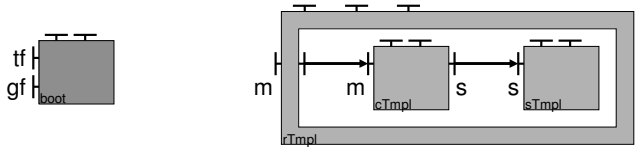
```
Fractal.getBindingController(rTmpl).bindFc(
  "m", cTmpl.getFcInterface("m"));
```

BindingController
- listFc
- lookupFc
- bindFc
- unbindFc



Déploiement : avec l'API

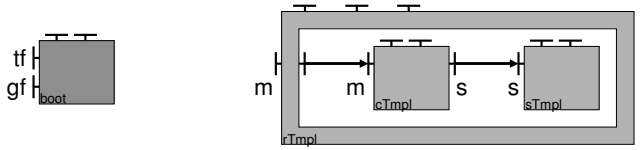
```
Fractal.getBindingController(rTmpl).bindFc(
  "m", cTmpl.getFcInterface("m"));
Fractal.getBindingController(cTmpl).bindFc(
  "s", sTmpl.getFcInterface("s"));
```



Déploiement : avec l'API

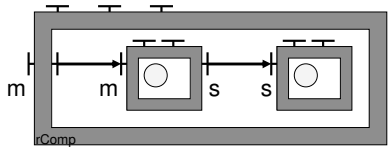
```
Component rComp =
 Fractal.getFactory(rTmpl).newFcInstance();
```

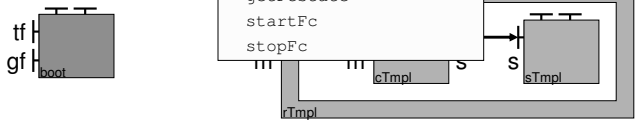
Factory
- getFcControllerDesc
- getFcContentDesc
- newFcInstance



Déploiement : avec l'API

```
Fractal.
  getLifeCycleController(
    rComp).startFc();
((Main)rComp.
  getFcInterface("m")).
    main(null);
```





Plan (rappel)

- Introduction
- Premier aperçu
- Définition du modèle
- Déploiement des composants
- Langage de description d'architecture
- Reconfiguration dynamique
- Implémentation de référence
- Conclusion

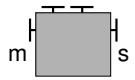
Déploiement : Fractal ADL

Définition des types de composants

```
<component-type name="RootType">
 <provides>
  <interface-type name="m" signature="pkg.api.Main" contingency="mandatory" cardinality="singleton"/>
 </provides>
</component-type>
```

optionnel

```
<component-type name="ClientType">
 <provides>
  <interface-type name="m" signature="pkg.api.Main"/>
 </provides>
 <requires>
  <interface-type name="s" signature="pkg.api.Service"/>
 </requires>
</component-type>
```



Déploiement : Fractal ADL

Héritage entre définitions

```
<component-type name="ClientType" extends="RootType">
 <requires>
  <interface-type name="s" signature="pkg.api.Service"/>
 </requires>
</component-type>
```

Est équivalent à :

```
<component-type name="ClientType">
 <provides>
  <interface-type name="m" signature="pkg.api.Main" contingency="mandatory" cardinality="singleton"/>
 </provides>
 <requires>
  <interface-type name="s" signature="pkg.api.Service"/>
 </requires>
</component-type>
```

Déploiement : Fractal ADL

➔Héritage entre définitions (suite)

```
<component-type name="ExtendedClientType" extends="ClientType">
  <requires>
    <interface-type name="s" signature="pkg.api.ExtendedService"/>
  </requires>
</component-type>
```

Est équivalent à :

```
<component-type name="ExtendedClientType">
 <provides>
   <interface-type name="m" signature="pkg.api.Main" contingency="mandatory" cardinality="singleton"/>
 </provides>
 <requires>
   <interface-type name="s" signature="pkg.api.ExtendedService"/>
 </requires>
</component-type>
```


Déploiement : Fractal ADL

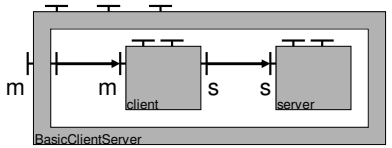
➔Définition des composants primitifs

```
<primitive-template name="BasicClient" implements="ClientType">
  <primitive-content class="pkg.lib.ClientImpl"/>
</primitive-template>

<primitive-template name="BasicServer" implements="ServerType">
  <primitive-content class="pkg.lib.ServerImpl"/>
  <controller>
    <attributes signature="pkg.api.ServiceAttributes">
      <attribute name="Header" value="-> "/>
      <attribute name="Count" value="1"/>
    </attributes>
  </controller>
</primitive-template>
```


Déploiement : Fractal ADL

➔Définition des composants composites

```
<composite-template name="BasicClientServer" implements="RootType">
  <composite-content>
    <components>
      <component name="client" type="ClientType" implementation="BasicClient"/>
      <component name="server" type="ServerType" implementation="BasicServer"/>
    </components>
    <bindings>
      <binding client="this.m"   server="client.m"/>
      <binding client="client.s" server="server.s"/>
    </bindings>
  </composite-content>
</composite-template>
```



Déploiement : Fractal ADL

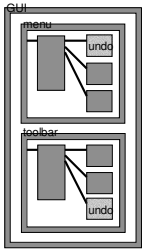
➔Définition des composants composites (suite)

```
<composite-template name="AbstractClientServer" implements="RootType">
  <composite-content>
    <components>
      <component name="client" type="ClientType"/> <!-- pas d'implémentation -->
      <component name="server" type="ServerType"/> <!-- pas d'implémentation -->
    </components>
    <bindings>
      <binding client="this.m"   server="client.m"/>
      <binding client="client.s" server="server.s"/>
    </bindings>
  </composite-content>
</composite-template>

<composite-template name="BasicClientServer" extends="AbstractClientServer">
  <composite-content>
    <components>
      <component name="client" implementation="BasicClient"/>
      <component name="server" implementation="BasicServer"/>
    </components>
  </composite-content>
</composite-template>
```

Déploiement : Fractal ADL

➔**Définition des composants partagés**

```
<composite-template name="GUI" implements="…">
 <composite-content>
  <components>
   …
  </components>
  <bindings>
   …
  </bindings>
  <sharing>
   <shared-component path="menu/undo" ref="toolbar/undo"/>
   …
  </sharing>
 </composite-content>
</composite-template>
```



Déploiement : Fractal ADL

➔**Création d'un composant avec l'ADL (en Java)**

- en ligne de commande

```
java … org.objectweb.fractal.adl.Launcher template
```

- par programme

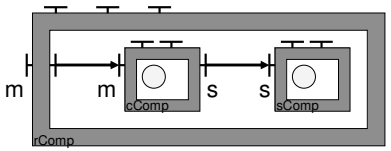
```
Parser parser = Launcher.getBootstrapParser();
Component comp = parser.loadTemplate("template", false);
// ou :
Component tmpl = parser.loadTemplate("template", true);
Component comp = Fractal.getFactory(tmpl).newFcInstance();
```

- les définitions doivent être dans le CLASSPATH

Plan (rappel)

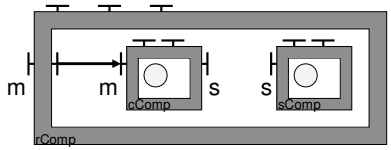
Reconfiguration dynamique

```
ContentControlller rCompCC = Fractal.getContentController(rComp);
Component cComp = rCompCC.getFcSubComponents()[0];
Component sComp = rCompCC.getFcSubComponents()[1];
Fractal.getLifeCycleController(rComp).stopFc();
Fractal.getBindingController(cComp).unbindFc("s");
```



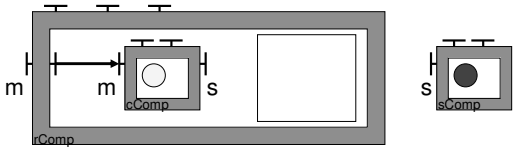
Reconfiguration dynamique

```
rCompCC.removeFcSubComponent(sComp);
```



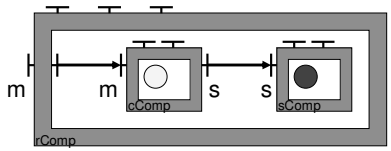
Reconfiguration dynamique

```
rCompCC.removeFcSubComponent(sComp);
sComp = gf.newFcInstance(sType, "parametricPrimitive", "NewServerImpl");
rCompCC.addFcSubComponent(sComp);
```



Reconfiguration dynamique

```
rCompCC.removeFcSubComponent(sComp);
sComp = gf.newFcInstance(sType, "parametricPrimitive", "NewServerImpl");
rCompCC.addFcSubComponent(sComp);
Fractal.getBindingController(cComp).
  bindFc("s", sComp.getFcInterface("s"));
Fractal.getLifeCycleController(rComp).startFc();
```



Plan (rappel)

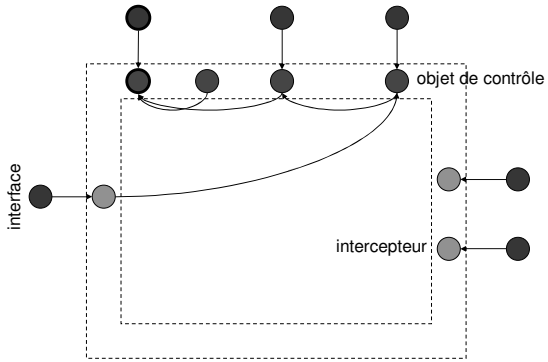
- ➔Introduction
- ➔Premier aperçu
- ➔Définition du modèle
- ➔Déploiement des composants
- ➔Langage de description d'architecture
- ➔Reconfiguration dynamique
- ➔Implémentation de référence
- ➔Conclusion

Implémentation : aperçu

➔**Julia**
- ➢implémentation de référence du modèle Fractal (niveau 3.3)

➔**Objectifs**
- ➢réaliser un *framework* extensible pour programmer des contrôleurs
- ➢fournir des objets de contrôle variés :
  - ●offrant un *continuum* entre configuration statique et reconfiguration dynamique
  - ● pouvant être optimisés et déoptimisés dynamiquement
- ➢implanter les objets de contrôle de façon à minimiser, en priorité :
  - ●le surcoût en temps sur les applications,
  - ●le coût en temps des méthodes de l'API Fractal,
  - ●le surcoût en mémoire sur les applications,
  - ●Le coût en mémoire des méthodes de l'API Fractal

Implémentation : structures de données



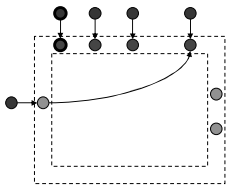
Implémentation : fichier de configuration

➔**Définit les descripteurs de contrôleur Fractal**
- ➢Tels que : primitive, parametricPrimitive, composite…
- ➢Utilisés dans `GenericFactory.newFcInstance(…)`

➔**Contient une liste de définitions**
- ➢Définition = (*name definition*)
- ➢Une définition peut référencer d'autres définitions :
  - ●`(x (1 2 3) )  # x est égal à (1 2 3)`
  - ●`(y (a b c 'x) )  # y est égal à (a b c (1 2 3))`

Implémentation : fichier de configuration

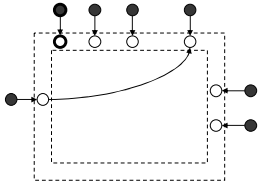
```
(primitive
 ( 'interface-class-generator
  ( 'component-itf
   'binding-controller-itf
   'lifecycle-controller-itf )
  ( 'component-impl
   'container-binding-controller-impl
   'lifecycle-controller-impl )
  ( (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
     org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator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



Implémentation : fichier de configuration

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

générateur de classes pour les interfaces

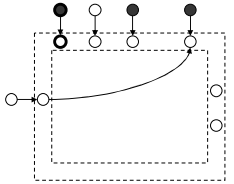


```
(interface-class-generator
  (org.objectweb.fractal.julia.asm.InterfaceClassGenerator
    org.objectweb.fractal.julia.BasicComponentInterface)
)
```


Implémentation : fichier de configuration

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

types des interfaces de contrôle publiques

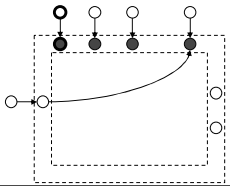


```
(component-itf
  (component org.objectweb.fractal.api.Component)
)
```


Implémentation : fichier de configuration

```
(primitive
 ( 'interface-class-generator
  ( 'component-itf
   'binding-controller-itf
   'lifecycle-controller-itf )
  ( 'component-impl
   'container-binding-controller-impl
   'lifecycle-controller-impl )
  ( (org.objectweb.fractal.julia.asm.InterceptorClassG
     org.objectweb.fractal.julia.asm.LifeCycleCodeGe
     org.objectweb.fractal.julia.asm.TraceCodeGenera
   org.objectweb.fractal.julia.asm.MergeClassGenerat
   none
 ) )
```

objets de contrôle

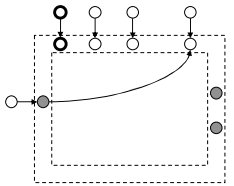


```
(component-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    ComponentImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.julia.BasicComponentMixin
    org.objectweb.fractal.julia.TypeComponentMixin
  ))
)
```


Implémentation : fichier de configuration

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

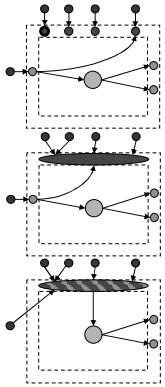
générateur de classes pour les intercepteurs



Implémentation : fichier de configuration

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

options d'optimisation



Plan (rappel)

- Introduction
- Premier aperçu
- Définition du modèle
- Déploiement des composants
- Langage de description d'architecture
- Reconfiguration dynamique
- Implémentation de référence
- Conclusion

Conclusion : apports d'usage attendus

- **Facilité de développement et d'intégration des plate-formes**
  - En développement
  - Configuration au déploiement
  - Administration et reconfiguration en production
- **Productivité des développeurs & intégrateurs**
  - Simplicité
  - Méthodes standards
  - Outillage standard
- **Pérennité des plate-formes**
  - Adaptabilité & réutilisabilité
  - Maintenabilité & évolutivité
  - Support des standards par déclinaison

Conclusion : état actuel

- **Principes architecturaux pour l'ingénierie des systèmes**
  - ✓ Construction par composition de briques logicielles : les composants
  - ✓ Vision homogène de la topologie des systèmes logiciels
    - applications, intergiciel, systèmes d'exploitation
      - –exemples : FractalGUI, Speedo, Jabyce, THINK
  - ✗ Gestion uniforme des ressources, des activités et des domaines de contrôle
  - ✓ Couverture du cycle de vie complet :
    - développement, déploiement (uniquement centralisé), supervision (API)
- **Réalisation d'un support d'exécution pour composants**
  - Support d'exécution pouvant être
    - ✓ spécialisé, outillé, étendu
    - ✗ composé avec d'autres canevas : persistance, réplication,sécurité, …

Conclusion : perspectives

- **Administration : JMX**
- **Déploiement distribué**
- **Outils multi-cibles (Java ou C)**
- **Evaluation : application à ObjectWeb**
- **Recherches en cours**
  - FTR&D : comportements (spécification/observation TLA, composition d'automates), événements/réactions (composition, auto-adaptation…), domaines de sécurité, personnalité EJB
  - INRIA : optimisations, transactions, formalisation
  - EMN : auto-adaptation, ADL
  - I3S : contrats, assertions
  - Université Valenciennes : propriétés non fonctionnelles

Conclusion : information

- **Spécification du modèle**
  - Démarré en Janvier 2002
  - Spécification ObjectWeb (1.0 : juillet 2002, 2.0 : août 2003)
  - Auteurs
    - E. Bruneton - France Télécom R&D
    - T. Coupaye - France Télécom R&D
    - J.B. Stefani - INRIA
- **Liens**
  - Site web : http://fractal.objectweb.org
  - Mailing list : fractal@objectweb.org