

Master Informatique IFPRU
UE FMIN 304

Réutilisation et Composant - Introduction

Notes de cours - 2005-2010
Christophe Dony
Université Montpellier-II

1 Contenu du cours

- Rappel sur les schémas de réutilisation objet et sur l'architecture des lignes de produits (frameworks).
- Limites de l'approche objet. Présentation de l'idée de composant.
- Composants assemblables. L'exemple des Java-Beans. Etude des schémas de connexion par événements (schema Observer, schéma Adapter, étude dédiée du schéma MVC). Etude des différents types de propriétés typiques (propriétés liées).
- Composants distribués. L'exemple des composants EJB dans la technologie JEE :
 - *session bean* pour traiter les requêtes métier des clients,
 - *message-driven bean* pour gérer l'asynchronisme des communications,
 - Abstraction de la persistance et des transactions
 - Architecture globale d'une application JEE de base avec JSP et Servlet.
- Composants avancés (Witgets) pour les applications WEB :

Comparaison de deux approches :

- composants JSF-SEAM dans le framework JEE,
- composants dans le framework Seaside,
- Aspects : Découplage entre préoccupations fonctionnelles et non fonctionnelles et assemblage selon les besoins.
- Programmation par composants avec la technologie OSGI (Open Service Gateway Initiative),
- Web services : composants et protocoles de communication textuels.
L'exemple de JEE.

2 Entités réutilisables du génie logiciel : des fonctions aux objets puis aux composants

Mots clés : Paramétrage, Découplage, Non anticipation.

2.1 Paramétrage d'une fonction par une autre

Exemple des itérateurs.

Itérateur : fonction appliquant une fonction successivement à tous les éléments d'une collection et retournant (ou fabriquant) la collection des résultats obtenus.

Une version en *Scheme* : la collection est fournie via une liste.

```
(define (carre x) (* x x))
(define (cube x) (* x x x))

(define (map f l)
  (if (null? l)
      l
      (cons (f (car l)) (map f (cdr l)))))

> (map carre '(1 2 3))
= (1 4 9)
> (map cube '(1 2 3))
= (1 8 27)
```

Une version en C.

```
#include <stdio.h>

int square(int a){ return a * a;}
int cube(int a){ return a * a * a;}

map(int (*f1)(int), int t1[], int r[], int taille){
    int i;
    for (i = 0; i < taille; i++){
        r[i] = f1(t1[i]);
    }
}
```

2.2 Paramétrage d'un algorithme par une fonction

```
(define (tri-insertion l inf?)
  (define (inserer x l)
    (cond ((null? l) (list x))
          ((inf? x (car l)) (cons x l))
          (#t (cons (car l) (inserer x (cdr l))))))
  (if (null? l)
      ()
      (inserer (car l) (tri-insertion (cdr l) inf?))))

(tri-insertion '(7 3 5 2 6 1) (lambda (x y) (< x y)))
; = (1 2 3 5 6 7)
(tri-insertion '(#\d #\a #\c #\b) char<?)
; = (#\a #\b #\c #\d)
(tri-insertion '("bonjour" "tout" "le" "monde") string-ci<?)
; = ("bonjour" "le" "monde" "tout")
```

2.3 Paramétrage par un objet

La programmation par objets introduit de nouveaux schémas :

- d'extension (description différentielle, héritage)
- de paramétrage par fonctions d'ordre supérieur : encapsulation, passage d'objets en argument et liaison dynamique.

Objet : encapsulation de données dotées d'un ensemble de fonctions.

Passer un objet en argument, c'est passer un ensemble de fonctions.

2.4 Paramétrage par Spécialisation

```
abstract class Produit{
    protected int TVA;
    int prixTTC() {
        return this.prixHT() * (1 + this.getTVA());
    }
    abstract int prixHT();
    int getTVA() {return TVA;}}

class Voiture extends Produit {
    int prixHT() {return (prixCaisse()+prixAccessoires()+ ...)} ... }

class Livre extends Produit {
    protected boolean tauxSpecial = true;
    int prixHT() {...}
    int getTVA() {if (tauxSpecial) return (0,055) else return (0,196);}
```

2.5 Paramétrage par composition

Code générique (méthode *service*) adapté par un plugin défini sur un composite.

```
class Compiler{
  Parser p;
  CodeGenerator c:
  Compiler (Parser p, CodeGenerator c){
    this.p = p;
    this.c = c;}

  public void compile (SourceText st)
    AST a = p.parse(st);
    generatedText gt = c.generate(a);}
```

2.6 Applications

- **API, Bibliothèques de classes**

Ensemble de classes spécifiques d'un domaine.

Ex l'API *java.lang.io* ou les bibliothèque de *Collection*.

- **Hiérarchie de classe**

Une hiérarchie de classe est une bibliothèque thématique et extensible.

Article historique : *R.E.Johnson, B.Foote : Designing reusable classes. Journal of Object-oriented Programming, Vol 1, No 2, pp 22-35, July 1988.*

- **Frameworks - Lignes de Produits**

Application pré-packagée spécialisable et Adaptable selon divers schémas (spécialisation, composition, fonctions d'ordre supérieur, descripteurs) pour en faire des applications spécifiques.

Invention du pattern **Inversion of Control** ou **Hollywood Principle**.

Articles :

M.E. Fayad, D.C. Schmidt, R.E. Johnson, "Building Application Frameworks", Addison-Wesley, 1999. Special Issue of CACM, October 1997.

G. Booch, Designing an Application Framework, Dr. Dobb's Journal 19, no 2, February 1994.

- **Plug-ins**

Composant destiné à paramétrer un framework.

Exemple courant : l'environnement de développement *ECLIPSE* dispose par exemple de plugins pour le développement en Java, en C++ ou en J2EE.

2.7 Plus de découplage - Paramétrage par des Aspects

La programmation par aspects propose une possibilité de séparer une fonctionnalité de ses diverses fonctionnalités accessoires.

Elle permet d'adapter au cas par cas et selon les besoins la fonctionnalité essentielle à différents contextes d'utilisation, sans toucher à son code.

Chaque fonctionnalité accessoire peut également s'adapter à diverses fonctionnalités essentielles.

Exemple : aspect *Synchronized* d'une méthode Java.

2.8 Limites de l'approche objet

2.8.1 Pb No1 Couplage explicite

```
class A{
    B b = new B();
    public ma(){ b.mb(); }

class B{
    public mb(){ ... }}
```

Problèmes :

- Impossible de voir que A est dépendant de B sans regarder le code de A.
- Impossible pour une application de changer B par autre chose sans modifier A.
- Difficulté à gérer l'instantiation et la mise en relation d'objets en dehors du code. Ceci entraîne de graves difficultés pour des application devant mettre en relation des dizaines ou des centaines d'objets.

2.8.2 Solutions purement objet au couplage

Patterns **Injection de dépendances** et **Fabrique**.

L'injection de dépendance désigne une solution permettant de réaliser un couplage entre objets a posteriori.

L'injection de dépendance est utilisé dans les frameworks pour réaliser l'**inversion de contrôle** (Hollywood Principle).

L'injection de dépendance préfigure la connexion externe des langages à composants dont elle constituera une des solutions.

Une solution manuelle pour l'injection de dépendance en paramétrage par composition.

```
class A{
    IB b;
    public void setB(IB b){this.b = b;}
    public int ma(){ return (1 + b.mb()); }
}

interface IB {public mb();}

class B implements IB {public int mb(){ return(2); }}

class Application{
    public main () {
        A a = new A();
        Ib b = new B();
        a.setB(b);
        a.ma();
    }
}
```

Solutions basées sur le pattern **Fabrique**.

Factory methods are common in toolkits and frameworks where framework code needs to create objects of types which may be subclassed by applications using the framework. *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides : Design Patterns : Elements of Reusable Object-Oriented Software Addison Wesley, 1994.*

Subclasses of *View* in Smalltalk-80 specify `defaultControllerClass` as the factory method that subclasses should override and which returns a class used by the View class, `defaultController`, to create instances.

2.8.3 Pb No 2 : Anticipation

Dans l'exemple de la programmation par évènements, comment décider à l'avance quels évènements doit émettre un objet ? Comment deviner quel évènement va intéresser un client ?

La non anticipation consisterait à laisser un utilisateur d'un objet décider a posteriori et sans modifier le code de l'objet, quels évènements, relatifs à la vie de l'objet, l'intéressent.

La programmation par aspects propose une solution pour la non anticipation.

2.8.4 Pb No 3 : Programmation de grandes applications distribuées

Difficulté à gérer

- une application constituée de centaines d'objets en interaction lorsque les dépendances sont masquées dans le code.
- l'interconnexion des instances et des modules
- la concurrence
- la distribution, le déploiement compte tenu de l'hétérogénéité des plateformes
- les changements éventuels des protocoles de communication
- l'ajout de fonctionnalités
- la chaîne de recompilation et de redéploiement en cas de modifications.

Les composants offrent une solution générale à ces problèmes

2.9 Composant

- Une définition :
 - entité logicielle pré-fabriquée, qui peut
 - être paramétrée de diverses façon pour fonctionner de différentes manières et dans différents contextes,
 - être assemblé de diverses manières avec d'autres composants compatibles.
- Une autre définition :
 - entité logicielle pré-fabriquée, définie par une ou plusieurs interfaces, qui peut être “composée” de diverses façons à d'autres, dans différents contextes, pour réaliser de nouveaux composites. *Clemens Szyperski. Component Software. Acm Press 2002*

2.10 Evolution de l'idée de composant - D'après D Hoarau (Valoria)

1960s Components would be a good idea 1970s Modules and objects 198x OLE 1.0 1991 CORBA 1.1 1993 OLE 2.0 / COM 1995 CORBA 2.0, DCOM? 1997 JavaBeans, COM+, ActiveX, Koala 1998 Sofa 1999 Enter prise JavaBeans (EJB)

2002 Fractal Component Model, Corba Component Model (CCM) 2002 ADL :
Langages de description d'architectures à base de composants. 2005 Widgets
pour la programmation WEB 2008 Langages de programmation par composants.

2.11 Les familles de langages

- L'approche Industrielle : (D)COM, .NET, J2EE-EJB
- L'approche Académique (recherche)
 - Langages de Description d'architecture (ADLs) : Unicon, C2, Wright, ACME, Fractal
 - Les langages de programmation par composants : ArchJava, ComponentJ, SCL, Julia(Fractal)
- Les modèles et normes : CCM (Corba Component Model), UML2.0

2.12 composants assemblables

Encapsulations de services, décrites par des interface, réglable, connectables via des ports ou via des puits d'évènements, composables en utilisant des langages de programmation par composants ou des langages de description d'architectures.

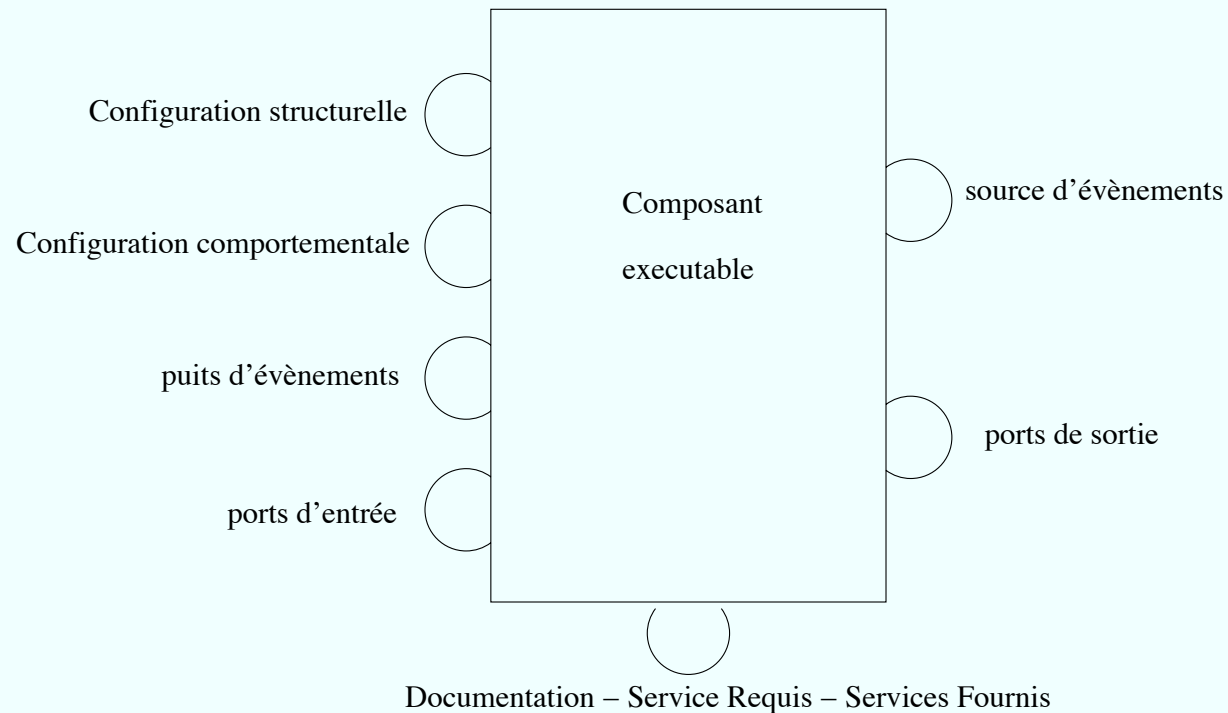


FIG. 1 – Encapsulation de services, réglable et composable

Modèles et langages de programmation par composants assemblables : COM, Java-beans, Fractal (Julia), ArchJava, ComponentJ, UniCon, Darwin, Acme, composants UML2.0, ...

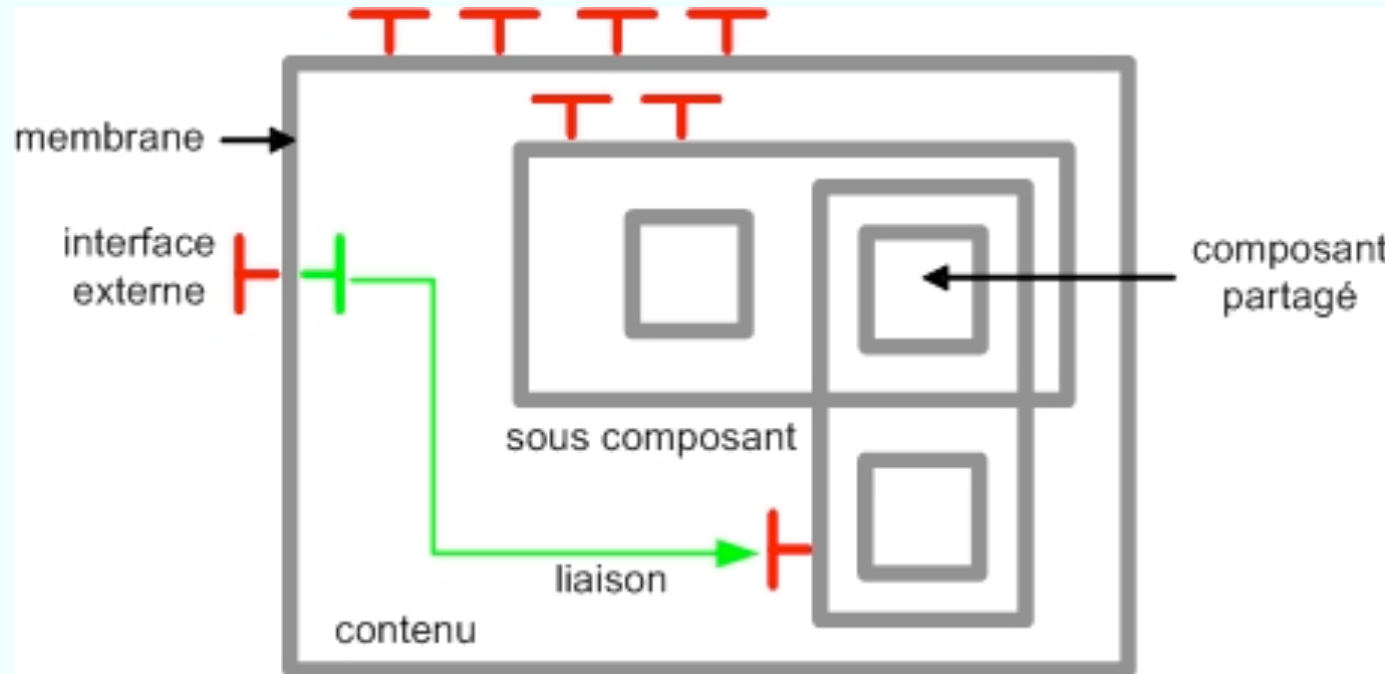


FIG. 2 – Composant Fractal (<http://wordpress.pragmaconsult.lu/wp-content/uploads/2008/07/fractal-component.jpg>). Voir aussi <http://fractal.ow2.org/>.

Concepts clé : composants, ports, connecteurs, interfaces, composites, protocoles d'interaction (requête-réponse, diffusion-abonnement (*publish/subscribe*)).

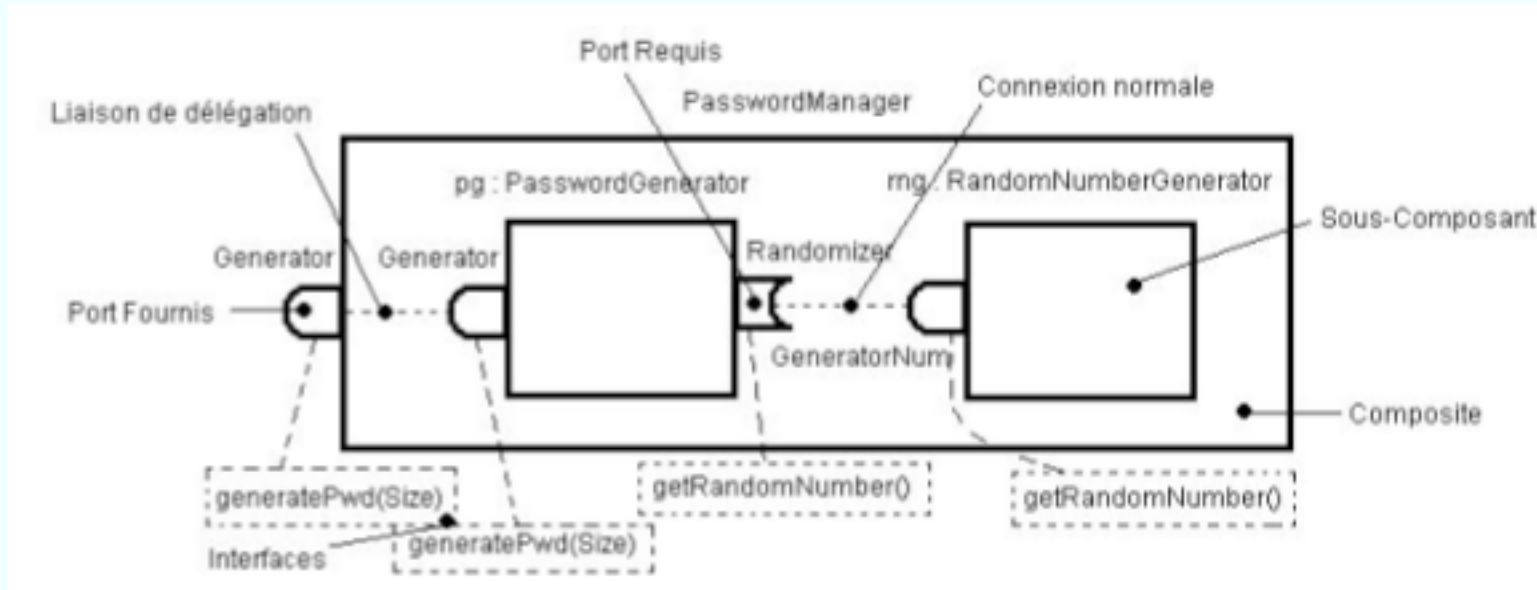


FIG. 3 – Concepts clés - Luc Fabresse - thèse de doctorat Université Montpellier-II 2008

2.13 composant distribué

Composants configurables, adaptables et utilisables à distance via des intergiciels offrant différents services.

Exemple : Les EJB (Enterprise Java Beans) sont des composants distribués utilisables dans l'environnement J2EE pour développer des applications distribuées N/Tier à base de composants en bénéficiant d'un ensemble de services prédéfinis.

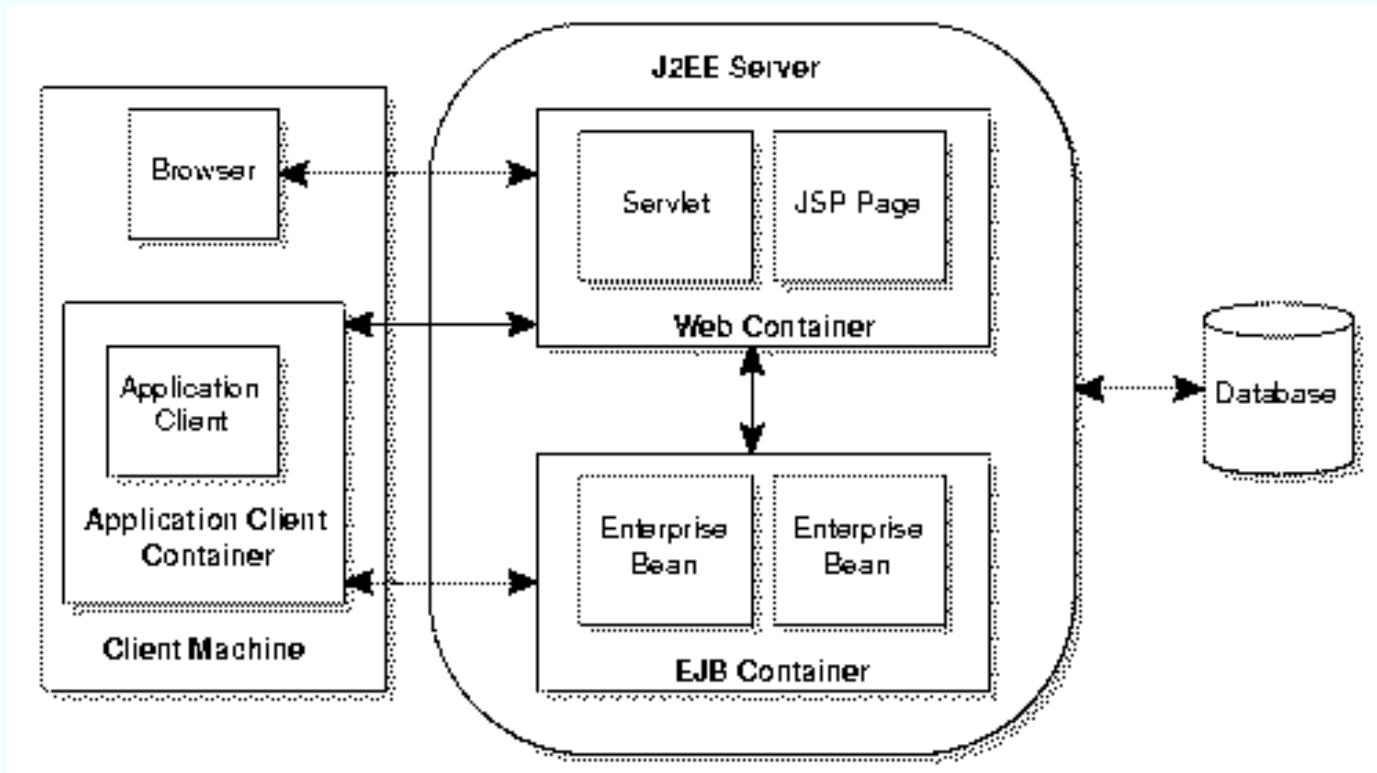


FIG. 4 – Conteneurs et Composants j2ee ©Sun

2.14 Web services

Un service Web est un ensemble de protocoles et de normes informatiques utilisés pour échanger des données entre les applications i.e. pour réutiliser à distance et de façon interopérable (norme SOA - Service Oriented Architecture ou Architecture orientée services) des composants.