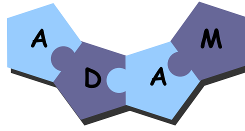


USTL – Master Mention Informatique
M2 spécialité recherche

Intergiciel, Composants et Architectures Logicielles (ICAL)

Composants pour les intergiciels et les applications réparties

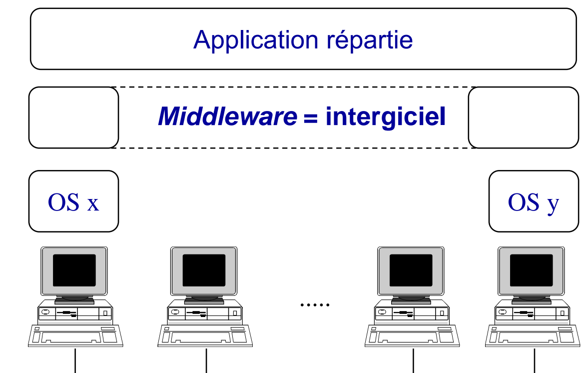
Lionel.Seinturier@univ-lille.fr
LIFL – Équipe GOAL & Équipe-projet INRIA ADAM

Plan

1. Introduction
2. Deux exemples de modèles de composants
   2.1 SCA
   2.2 Fractal
   2.3 Synthèse
3. Problématique de recherche
   3.1 Adaptation niveau plate-forme
   3.2 Adaptation niveau application
4. Conclusion

1. Introduction

« *Middleware is everywhere* » © IBM



1. Introduction

Dualité application / intergiciel (*middleware*)

Middleware

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

1. Introduction

Dualité application / intergiciel (*middleware*)

Concepts du génie logiciel

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

1. Introduction

≠ styles de *middleware*

- client/serveur (requête/réponse)
- orienté message (MOM *Message-Oriented Middleware*)
- de diffusion de flux de données
- pair-à-pair (P2P *peer-to-peer*)
- espace de données partagées
- à base de code mobile
- à base d'agents « intelligents »

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

1. Introduction

≠ cibles pour le *middleware*

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

1. Introduction

La recherche en *middleware*

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

1. Introduction

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

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

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

Présent : plusieurs tendances
- composant, aspect, MDE, réflexivité

1. Introduction

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

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

1. Introduction

Définition composant

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

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

1. Introduction

Nombreux modèles de composant (20+)

- construits au-dessus Java, C, C++
- EJB, Java Beans, CCM, COM+, JMX, OSGi, SCA, CCA, SF
- Fractal, K-Component, Comet, Kilim, OpenCOM, FuseJ, Jiazzi, SOFA, ArticBeans, PECOS, Draco, Wcomp, Rubus, Koala, PACC-Pin, Java/A, HK2
- Bonobo, Carbon, Plexus, Spring
- au niveau analyse/conception : UML2

1. Introduction

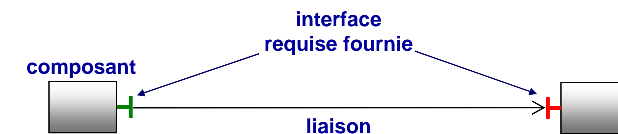
Conséquence de la multiplicité des modèles

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

1. Introduction

1ère grande catégorie de modèle de composants

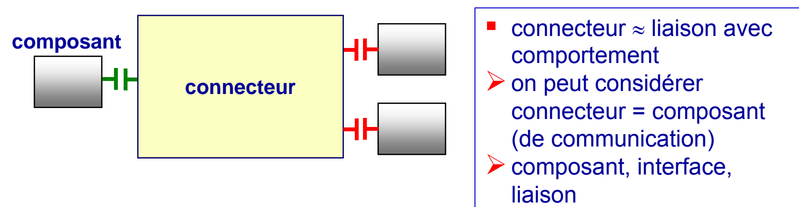
- triptyque : composant, interface, liaison
  - un composant fourni et/ou requiert une ou plusieurs interfaces
  - une liaison est un chemin de communication
    entre une interface requise et une interface fournie



1. Introduction

2ème grande catégorie de modèle de composants

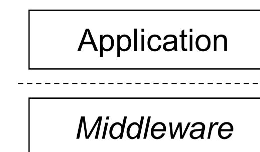
- triptyque : composant, port, connecteur
  - un composant fourni et/ou requiert une ou plusieurs ports
  - un connecteur implémente un schéma de communication entre des composants (client/serveur, diffusion, etc.)
  - un composant est relié à un connecteur via un ou plusieurs ports



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

1. Introduction

Classification des modèles pour système/intergiciel



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

1. Introduction

Quelques « poncifs » à propos des composants

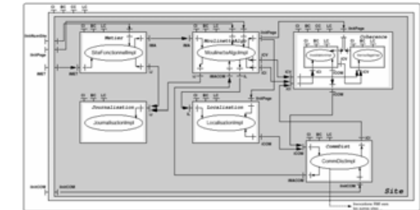
- COTS Commercial Off The Shelf
  - vieux discours (voir procédures, fonctions, objet, …)
  - taille applis ↗ donc besoin : toujours plus de réutilisation
  - mais *quid* de la contractualisation ?
- « *Programming in the large* »
  - vs « *programming in the small* » (objet)
  - vrai d'un certain point de vue
  - mais nouveaux points à traiter (liés au non fonctionnel par ex.)

1. Introduction

Notion d'architecture logicielle

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

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



1. Introduction

Complémentarité

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

2 visions complémentaires

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

Plan

1. Introduction
2. Deux exemples de modèles de composants
   2.1 SCA
   2.2 Fractal
   2.3 Synthèse
3. Problématique de recherche
   3.1 Adaptation niveau plate-forme
   3.2 Adaptation niveau application
4. Conclusion

2.1 SCA

- Constat : applications réparties (CORBA, .NET, Java EE, …) actuelles
- souvent complexes, rigides, peu évolutives

- Tendance SOA (*Service Oriented Architecture*)
- besoins identifiés
  - interfaces bien définies avec une sémantique liée au métier
    - focalisation sur l'échange des données métier
  - protocoles de communication standardisés
  - recombinaison flexible de services pour améliorer la fléxibilité du logiciel
- vision
  - *A service is an abstraction that encapsulates a software function*
  - *Developers build services, use services and develop solutions that aggregate services*
  - *Composition of services into integrated solutions is a key activity*

2.1 SCA

Service

- SOA : *Software Oriented Architecture*
- SaaS : *Software As A Service*

Modèle "économique" tout est service

Principes des SOA

• *Encapsulation*	existant encapsulé pour pouvoir être réutilisé avec SOA
• *Loose coupling*	minimiser les dépenses
• *Contract*	communication ne se font que via un accord entre services
• *Abstraction*	masquer la logique du service au monde extérieur
• *Reusability*	découpage des services pour promouvoir la réutilisabilité
• *Composability*	services peuvent être composés et coordonnés pour former des services composites
• *Autonomy*	services contrôlent la logique qu'ils encapsulent
• *Optimization*	services peuvent être optimisés individuellement
• *Discoverability*	services sont fait pour être découvert

2.1 SCA

Service

1ère vague Épuration

Constat d'une trop grande hétérogénéité du middleware
taille, OS, langage, cible matérielle, modèle de programmation
Focalisation sur un PPCM (plus pétit commun dénominateur)

d'où W3C Web Services = HTTP + XML + SOAP

adoption par Sun (Java EE), Microsoft (.NET) comme solution
pour l'intéropérabilité (interne et externe)

2.1 SCA

Service

2ème vague … en fait

réintroduction des notions de composants et d'architecture logicielle
- Enterprise Software Bus (par ex. Sun JBI)
- OSGi
  - domotique (*box*, …)
  - embarqué (secteur automobile, …)
  - modularité (voir Java 7), système à *plugins* (Eclipse, serveur d'applications)
- SCA (*Software Component Architecture*)
  - donner une structuration aux applications orientée services
  - supporter différents
    langages de programmation, protocoles de communication,
    langages de définition d'interfaces, services non fonctionnels

2.1 SCA

SCA (Service Component Architecture)
- a component model for SOA
- 11/2005

Hosted by the Open SOA consortium
- http://www.osoa.org

Community extended to OASIS
- http://www.oasis-opencsa.org

Platform providers
- Open Source: Apache Tuscany, Newton, Fabric3, FraSCAti
- Vendors: IBM WebSphere FP for SOA, TIBCO ActiveMatrix, Covansys SCA Framework, Paremus, Newton, Rogue Wave HydraSCA, Oracle Fusion Middleware

2.1 SCA

A set of specifications (15) (09/2008)

Assembly model
- how to define structure of composite applications

Component implementation specifications
- how to write business services in particular languages
- Java, C++, PHP, Spring, BPEL, EJB SLSB, COBOL, C, …

Binding specifications
- how to access services
- Web services, JMS, JCA, RMI-IIOP

Policy framework
- how to add infrastructure services
- security, transaction, reliable messaging, …

Integration
- SCA Java EE Integration
- SCA OSGi/Spring (draft)

+ SDO for accessing data sources

2.1 SCA

Vision SCA

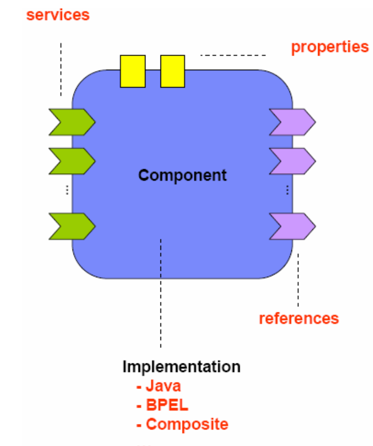
- composants de service à gros grain (interfaces avec méthodes orientées métier) assemblés à partir de composants à grain plus fin
- s'abstraire des détails du middleware
- conformance avec les standards existants
- spécifications ouvertes, multi-vendeurs
- couplage faible pour permettre l'agilité
- capitaliser sur les pratiques du SOA

2.1 SCA

Composant SCA

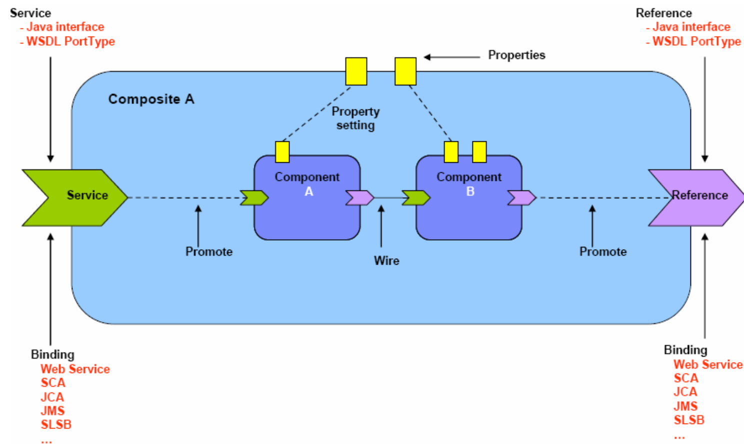
- service/référence
- propriété
- implémentation

- propriété non fonctionnelle (*intent*)



2.1 SCA

Assemblage SCA



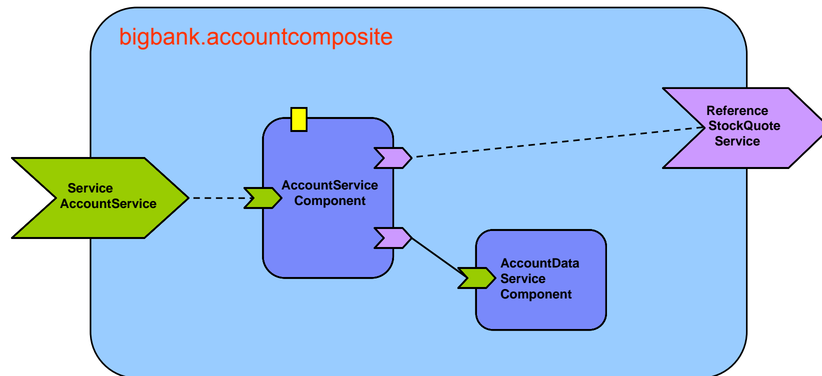
2.1 SCA

Indépendances

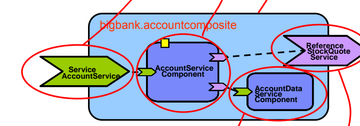
- langage de programmation
- langage de définition d'interfaces
- protocoles de communication
- propriétés non fonctionnelles

Simple Example

[Mike Edwards]
IBM Hursley Lab, England




```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           name="bigbank.accountcomposite" >

   <service name="AccountService" promote="AccountServiceComponent">
      <interface.java interface="services.account.AccountService"/>
      <binding.ws port="http://www.example.org/AccountService#
        wsdl.endpoint(AccountService/AccountServiceSOAP)"/>
   </service>

   <component name="AccountServiceComponent">
      <implementation.java class="services.account.AccountServiceImpl"/>
      <reference name="StockQuoteService"/>
      <reference name="AccountDataService"
                 target="AccountDataServiceComponent/AccountDataService"/>
      <property name="currency">EURO</property>
   </component>

   <component name="AccountDataServiceComponent">
      <implementation.bpel process="QName"/>
      <service name="AccountDataService">
         <interface.java interface="services.accountdata.AccountDataService"/>
      </service>
   </component>

   <reference name="StockQuoteService" promote="AccountServiceComponent/StockQuoteService">
      <interface.java interface="services.stockquote.StockQuoteService"/>
      <binding.ws port="http://example.org/StockQuoteService#
        wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
   </reference>
</composite>
```



OASIS

Java Implementation Example: Service Interface

```
package org.example.services.account;

@Remotable
public interface AccountService {

   public AccountReport getAccountReport(String customerID);
}
```

Interface is callable remotely eg. as a Web service

OASIS

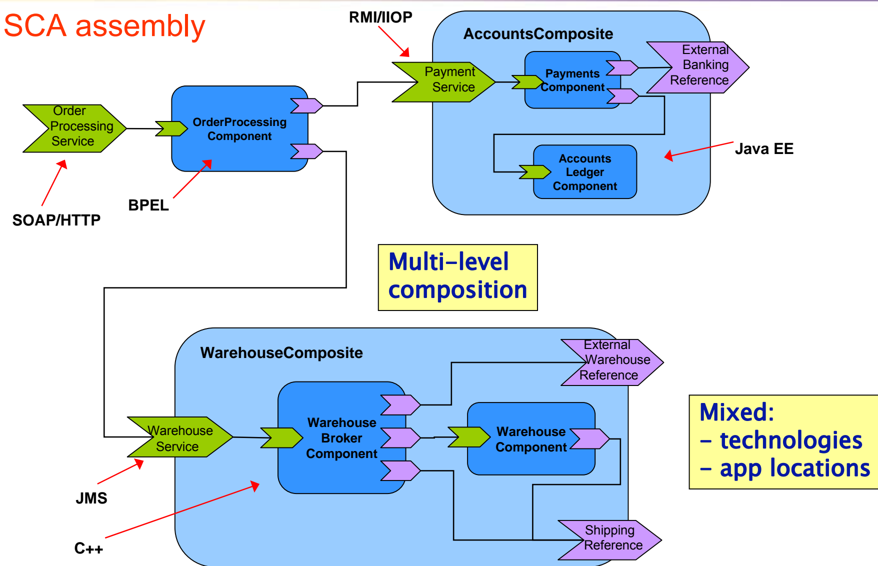
Java Implementation Example (1)

```
package org.example.services.account;

import org.osoa.sca.annotations.*;

@Service(interfaces = AccountService.class)
public class AccountServiceImpl implements AccountService {

    private String currency = "USD";
    private AccountDataService accountDataService;
    private StockQuoteService stockQuoteService;

    public AccountServiceImpl(
            @Property("currency") String currency,
            @Reference("accountDataService") AccountDataService dataService
            @Reference("stockQuoteService") StockQuoteService stockService) {
       this.currency           = currency;
       this.accountDataService = dataService;
       this.stockQuoteService  = stockService;
    }
```

Annotation for the service offered by this class

Constructor with annotations for injected property and references

OASIS



2.1 SCA

- SCA
- *yet another component model*
  - une façon de revisiter les problématiques
    - de la répartition
    - des services pour les plates-formes *middleware*
    - des concepts des *frameworks* de composants
  - renforce l'idée de l'indépendance services/implémentations
- point original : indépendance IDL/protocole/langage/*intent*

2.1 SCA

- Spécifications
  - OpenSOA http://www.osoa.org
  - OASIS OpenSCA http://www.oasis-opencsa.org
- Implémentations
  - Tuscany http://incubator.apache.org/projects/tuscany.html
  - Newton http://newton.codecauldron.org/site/index.html
  - Fabric3 http://xircles.codehaus.org/projects/fabric3
  - FraSCAti http://www.scorware.org
- Ressources
  - http://www.osoa.org/display/Main/SCA+Resources
  - http://www-128.ibm.com/developerworks/library/specification/ws-sca/
  - http://www.davidchappell.com/articles/Introducing_SCA.pdf
  - http://www-128.ibm.com /developerworks/websphere/techjournal/0510_brent/0509_brent.html
  - [Mike Edwards] http://events.oasis-open.org/home/sites/events.oasis-open.org.home/files/Flexible_Agile_Composition_01.ppt
  - http://www.osoa.org/download/attachments/250/Power_Combination_SCA_Spring_OSGi.pdf?version=3

Plan

1. Introduction
2. Deux exemples de modèles de composants
   2.1 SCA
   2.2 Fractal
   2.3 Synthèse
3. Problématique de recherche
   3.1 Adaptation niveau plate-forme
   3.2 Adaptation niveau application
4. Conclusion

2.2 Fractal

FT R&D (Orange Labs), INRIA

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

Historique

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

2.2 Fractal

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

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

2.2 Fractal

Exemple de *middleware*/applications développées avec Fractal

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

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

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

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

2.2 Fractal

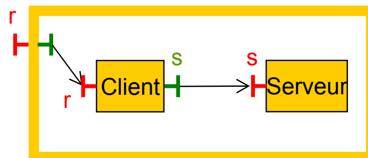
Principales annotations

- @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*) : la référence du service requis (de type T)
    - attribut de type T pour SINGLETON
    - attribut 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 : Cardinality.SINGLETON (par défaut) ou COLLECTION
    - contingency : Contingency.MANDATORY (par défaut) ou OPTIONAL

2.2 Fractal

Exemple Hello World

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



2.2 Fractal

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);
  }
}
```

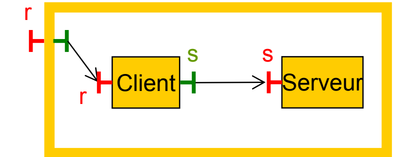
2.2 Fractal

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!");
} }
```


2.2 Fractal

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>
```


Plan

1. Introduction
2. Deux exemples de modèles de composants
   2.1 SCA
   2.2 Fractal
   2.3 Synthèse
3. Problématique de recherche
   3.1 Adaptation niveau plate-forme
   3.2 Adaptation niveau application
4. Conclusion

2.3 Synthèse

- SCA — un modèle de composants pour le SOA
- Fractal — un modèle de composant pour la construction de briques logicielles pour les plates-formes *middleware*

- objet -> composant -> architecture logicielle (ADL)
- objet -> composant
  - ajout de métainformation sur les classes
  - utilisation @ Java 5
- composant -> ADL
  - grammaire **XML** pour décrire les assemblages

- valable pour SCA, Fractal
- mais aussi pour OSGi (iPOJO + *Declarative Services*), …

Plan

1. Introduction
2. Deux exemples de modèles de composants
   2.1 SCA
   2.2 Fractal
   2.3 Synthèse
3. Problématique de recherche
   3.1 Adaptation niveau plate-forme
   3.2 Adaptation niveau application
4. Conclusion

3. Problématique de recherche

Une problématique de recherche (parmi d'autres)

- Adaptation
  - répondre à de nouveaux besoins fonctionnels
  - s'adapter à des conditions d'exécution nouvelles
  - mettre à jour des fonctionnalités, corriger des bugs
  - s'adapter aux habitudes des utilisateurs
  - prendre en compte de nouveaux périphériques, matériels

  - supporter différentes granularités de composants
    - de l'embarqué aux grilles en passant par les serveurs d'application
  - supporter différents modèles de composants (voir les 20+ existants)
    - un modèle ne répond pas à tous les besoins

- 2 niveaux de solution en terme de *middleware* orienté composant
  - adaptation de la plate-forme d'exécution
  - adaptation de l'application

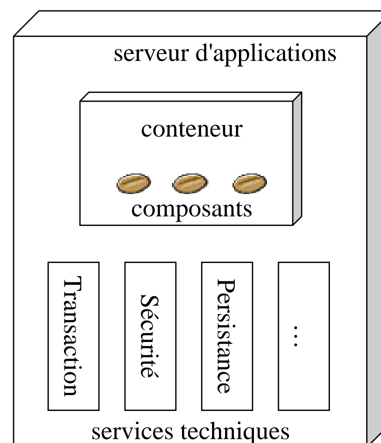
3.1 Adaptation niveau plate-forme

Structuration des plates-formes composant

- métier vs technique
- notion de conteneur
- principe d'inversion du contrôle

- ex. : EJB

- pb: figé, pas adaptable

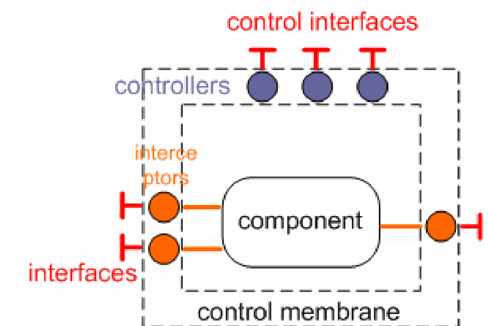


3.1 Adaptation niveau plate-forme

Solution pour l'adaptabilité des plates-formes composant

Notions

- contrôleur
- membrane de contrôle
- intercepteur
- interface de contrôle



Rendre ces éléments programmables

3.1 Adaptation niveau plate-forme

Métier vs contrôle

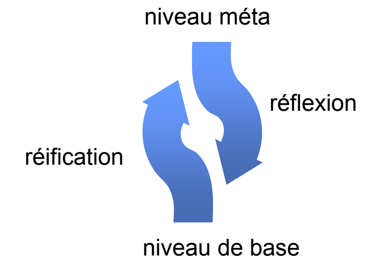
- Notion subjective
  - dépend du domaine applicatif
  - dépend des développeurs
  - dépend de l'évolution des applications
  - dépend de la granularité (voir les systèmes en couches)
- Contrôle
  - souvent services système
    - sécurité, persistance, réplication, tolérance aux pannes, …
  - mais pas uniquement
    - contrats (pre/post), intégrité de données, règles de gestion métier, traçabilité, gestion de *workflow*, …
  - par défaut dans le cas de Fractal
    - liaison, propriétés, hiérarchie de composants, démarrage/arrêt

3.1 Adaptation niveau plate-forme

Métier vs contrôle

- similitude avec métaprogrammation
- niveau de base
  - l'application
- niveau méta
  - réifie la base
  - l'instrospecte
  - la modifie
  - l' "ouvre"

niveau méta

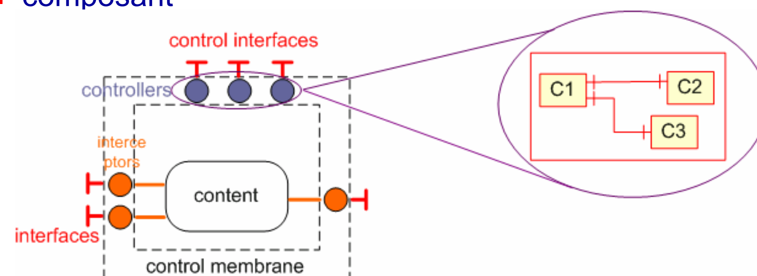


niveau de base

3.1 Adaptation niveau plate-forme

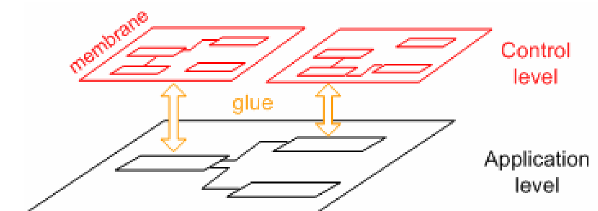
Programmation du contrôle

- classe/objet
  - implémentation de référence Fractal/Julia
- composant



3.1 Adaptation niveau plate-forme

Programmation du contrôle



- même outils, API, @, XML pour développer les 2 niveaux
- modèle de composant réflexif

3.1 Adaptation niveau plate-forme

Issues: controlling the controllers?

M2	Meta Control level
M1	Control level
M0	Business level

Component controllers are regular Fractal components

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

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

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

3.1 Adaptation niveau plate-forme

Les notions fondamentales des *frameworks* de composants

- dependency injection
- property injection
- component identity & naming

- component instantiation & initialization
- component destruction

- stub for accessing components
- asynchronous method invocation
- callback
- conversation management

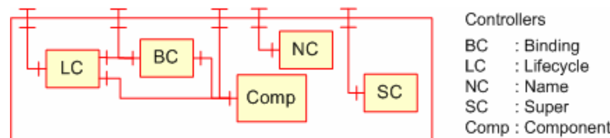
- introspection & reconfiguration API
- component hierarchy

Introduction de variabilité dans ces notions

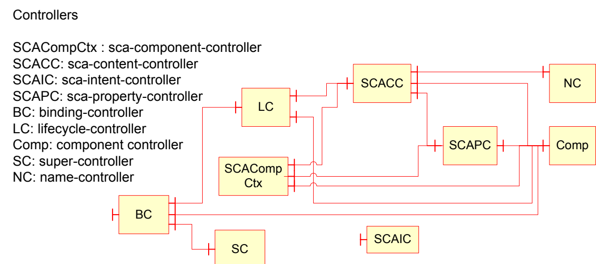
3.1 Adaptation niveau plate-forme

Exemples de membranes de contrôle

Primitif Fractal



Primitif SCA



3.1 Adaptation niveau plate-forme

- Variabilité dans les notions de base
  - supporter ≠ **personnalités** de composants
    - définition d'un micro-noyau et dérivations
    - promeut l'adoption
      - 1 modèle de composant ne répond pas à tous les besoins
    - promeut interopérabilité
    - promeut le multi-échelle
      - des composants à différents niveaux de granularité : de l'embarqué à la grille en passant par les serveurs d'applications

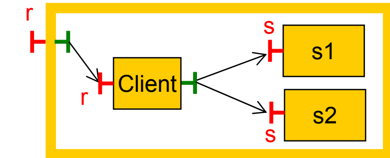
Plan

1. Introduction
2. Deux exemples de modèles de composants
   - 2.1 SCA
   - 2.2 Fractal
   - 2.3 Synthèse
3. Problématique de recherche
   - 3.1 Adaptation niveau plate-forme
   - 3.2 Adaptation niveau application
4. Conclusion

3.2 Adaptation niveau application

Reconfiguration

- passer d'une version A de l'application à une version B
  - dynamiquement



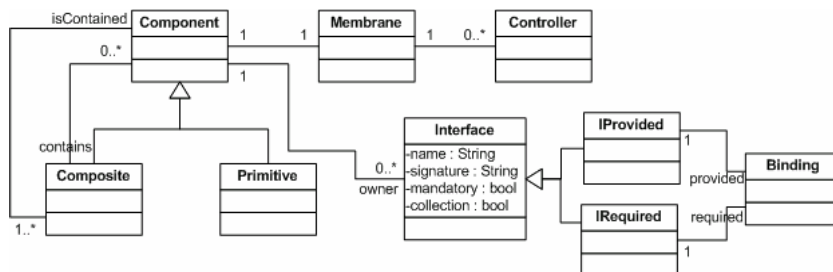
2 solutions possibles (parmi d'autres)

- API de transformation
- techniques de développement par aspect au niveau composant (AOP)

3.2 Adaptation niveau application

Reconfiguration par API

- architecture composant = structure de données
- modèle de cette structure (métamodèle)
- API de lecture et d'écriture de cette structure



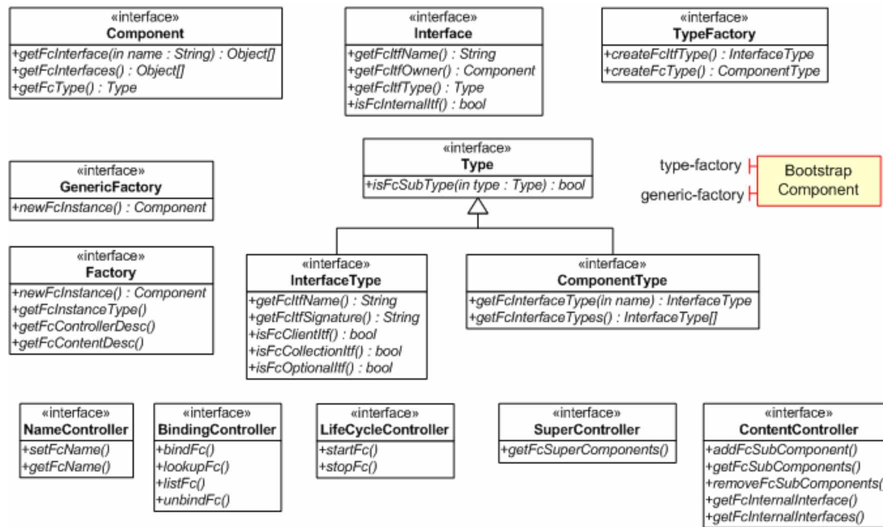
3.2 Adaptation niveau application

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

Introspection et modification

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

3.2 Adaptation niveau application



3.2 Adaptation niveau application

Reconfiguration par aspect (AOP)

Aux limites de la programmation objet

- 1968 : Simula
- 70s : SmallTalk
- 80s : C++
- 90s : Java puis C#

- indéniablement un succès
- programmation plus claire, modulaire, efficace (par rapport procédural, …)

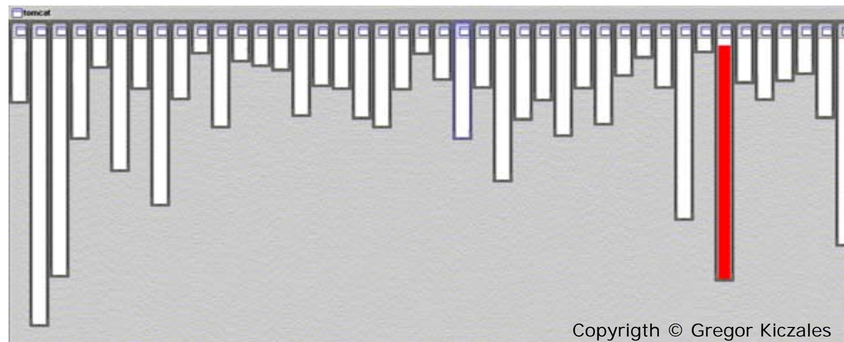
Mais plusieurs exemples de "résistance" à l'*objectisation*

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

3.2 Adaptation niveau application

Tomcat

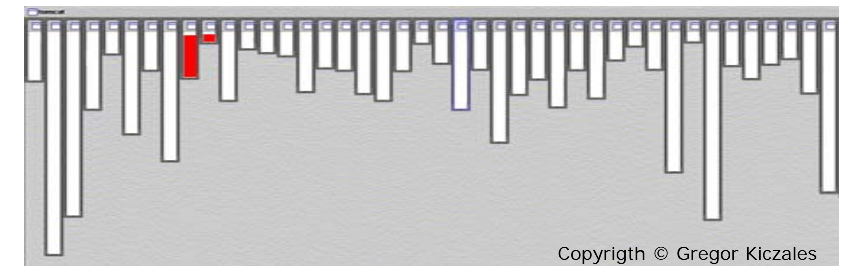
- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box

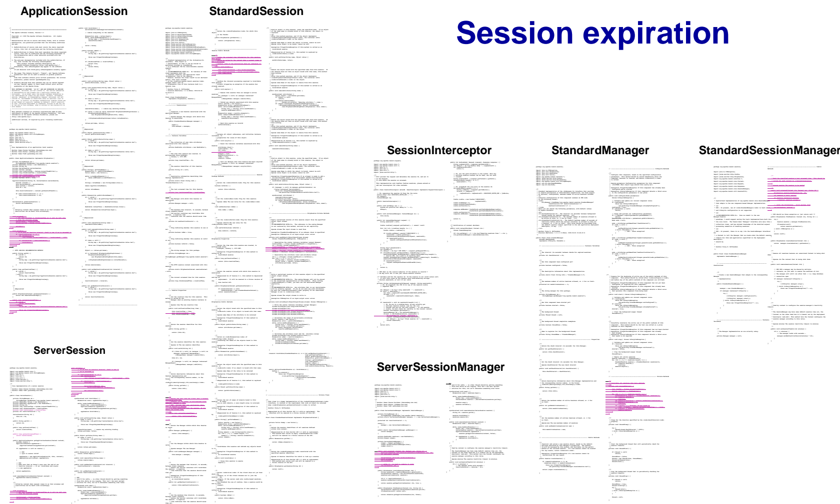


3.2 Adaptation niveau application

Tomcat

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





Copyrigth © Gregor Kiczales

Analyse

Code

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

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

Autre exemple de code dispersé et entremélé

```
class BlockingBuffer {

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

```
/* Politique buffer vide */
/* Politique buffer plein */
/* Politique FIFO */
```

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

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


Autre exemple de code entremélé

Programme JDBC, Java EE, …

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

3.2 Adaptation niveau application

Contraintes d'intégrité référentielles

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

3.2 Adaptation niveau application

AOP : Aspect-Oriented Programming [Kiczales 97]

Aspect

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

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

3.2 Adaptation niveau application

Tissage

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

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

- tissage statique (AspectJ, …) aspect = entité *compile time*
- tissage dynamique (JAC, …) aspect = entité *run time*

- performance vs flexibilité

3.2 Adaptation niveau application

Langages / *frameworks* AOP

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

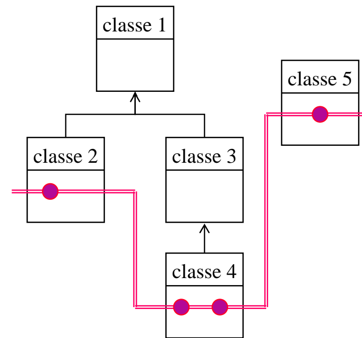
⇨ Smalltalk : AOP/ST, Apostole, AspectS, FracTalk-AOP
⇨ C/C++ : AspectC, AspectC++, Arachne
⇨ C#
  - AspectDNG, AOP.NET, AspectC#, Loom.NET, Weave.NET, AopDotNetAop, AOPEngine.NET, Aspect.Net, Aspect#, Eos, PostSharp, SetPoint!, NAspect

⇨ PHP : AOPHP, phpaspect
⇨ Python : Pythius
⇨ COBOL : AspectCobol, cobble

3.2 Adaptation niveau application

4 notions de base

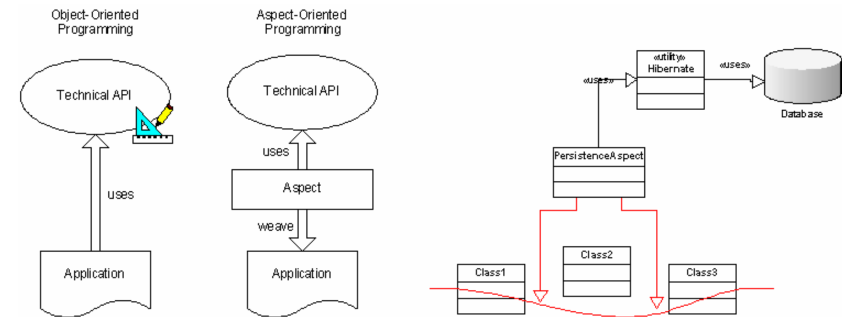
- **point de jonction** (*join point*)
  - ⇨ point dans le flot d'exec. du prog.
- **coupe** (*pointcut*)
  - ⇨ ensemble de points de jonction
  - ⇨ langage de *pattern* (* .. +)
  - ⇨ quantification sur le programme
- ***advice***
  - ⇨ le comportement d'un aspect (≈ méthode)
- **aspect**
  - ⇨ coupe(s) associée à advice
  - ⇨ *advice(s)* exécutés avant/après points de jonction de la coupe



3.2 Adaptation niveau application

2 constats sur l'AOP

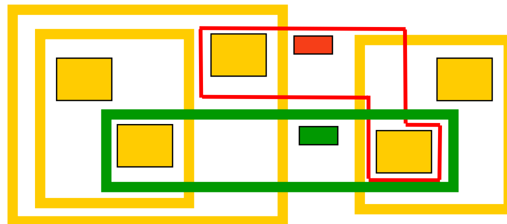
- ⇨ conduit à une inversion du contrôle (IoC: *Inversion of Control*)
- ⇨ technique d'intégration



3.2 Adaptation niveau application

Évolution de la notion d'aspect

- de l'objet aux composants/architecture logicielle
- raisonner en terme de "plans" sur l'architecture de base



3.2 Adaptation niveau application

3 notions

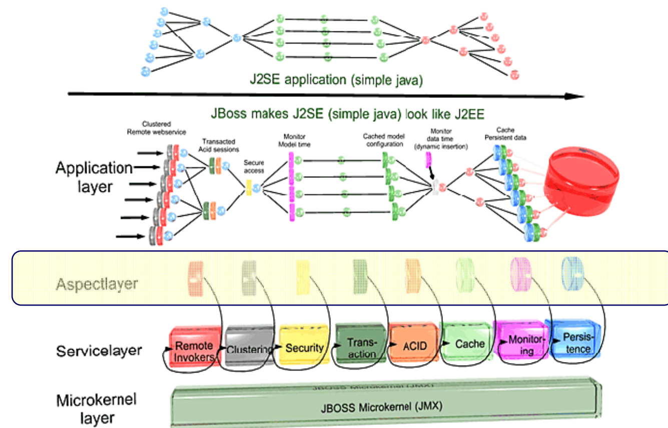
- 1 aspect = 1 composant
- domaine d'aspect = composant composite
  - capture les composants métier impactés par l'aspect
- liaison d'aspect
  - interception des invocations entrantes et/ou sortante d'un composant métier
  - déroutement vers le composant d'aspect)

- 1 forme de reconfiguration de l'architecture
- exemple de mise en œuvre
  - FAC http://fac.gforge.inria.fr
  - thèse de Nicolas Pessemier (2007 – ADAM)

3.2 Adaptation niveau application

Autre exemple d'unification aspect/composant

- Serveur Java EE JBoss et JBoss AOP



Plan

1. Introduction
2. Deux exemples de modèles de composants
   - 2.1 SCA
   - 2.2 Fractal
   - 2.3 Synthèse
3. Problématique de recherche
   - 3.1 Adaptation niveau plate-forme
   - 3.2 Adaptation niveau application
4. Conclusion

4. Conclusion

Objet -> composant -> architecture logicielle (ADL)

- monter en abstraction
- trouver des concepts de haut niveau qui vont améliorer/faciliter/rendre plus rapide, … la conception du *middleware* et des applications réparties
- construction intellectuelle qui se matérialise au niveau du code et/ou des modèles

- focus sur une problématique de recherche : Adaptation
  - montée en abstraction
    - composant de contrôle (modèle comp réflexif)
    - aspects (AOP) au niveau composant
      - API : relativement bas niveau (l'assembleur de la reconfiguration)
      - AOP : une vision architecturale et composant de la notion d'aspect
      - concepts de plus haut niveau qui s'appuient sur les notions de l'API

- mais il y en a d'autres : passage à l'échelle (*scalability*), sûreté, typage, langage basé composant, …

4. Conclusion

Conférences de recherche du domaine

- composant
  - CBSE, Euromicro CBSE, CD, WCOP, FACS, FMCO, GPCE, JC, OCM, SAVCBS, WPG, …
- architecture logicielle
  - CAL, ECSA-EWSA, FOCLASA, QoSA, WICSA, …
  - ASE, ASPEC, WCAT, ESEC, ICSE, RST, SAC, SC, SEDE, SEM, …
- *middleware*
  - CFSE, DAIS, WDAG, Euro-Par, ICDCS, ICPADS, IPDPS, Middleware, MidSens, NOTERE, OPODIS, PDCN, PDCS, PODC, RenPar, SOSP, SRDS, WDAS, …

Revue

- IEEE DS Online, JOT, JUCS, Automated Soft. Eng., CACM, COTS, Dist. Syst. Eng., IEE Software, IEEE Transac. on Par. and Dist. Syst., IET Software, Software Practise and Experience, Trans. on AOSD, …