Les composants et les services

F. Dagnat, M.T. Segarra

3e année, Filière SLR – UV IDL

Année 2016-2017

Progress

Un peu d'histoire de l'informatique

- Quels besoins ?
  - Réduire la complexité du dév (donc son coût) et le coût de maintenance
  - Comprendre les grands systèmes
  - Accroître le niveau de réutilisabilité (COTS)
- Solutions
  - Structurer/organiser le logiciel
  - Réutiliser ce qui est produit (par nous ou d'autres)
  - Programmer en assemblant
  - Utilisation de contrats
- Deux approches
  - Amont : l'architecture logicielle
  - Aval : les modèles et plate-formes de composants

Les objets ?

- Ils ont fourni une solution à certains problèmes
  - Structuration en classes, réutilisation de classes
- Mais
  - Limites structurelles
    - La structure d'une appli objet est peu lisible (ens. de fichiers)
    - Beaucoup de dépendances entre objets
    - Des dépendances cachées (variable locale)
  - Manque de mécanisme de gestion
    - Création des instances, gestion des dépendances entre classes, appels explicites de méthodes ...
    - Pas d'outils pour déployer les exécutables sur différents sites
  - Ne conviennent pas pour l'architecture de grands systèmes
    - Spécifient les services fournis mais pas les requis
    - Une seule forme de connexion (invoc. de méthodes)

Développer et assembler des composants

- Construire des applications par assemblage de « briques » logicielles réutilisables
- Pour l'assemblage, les briques doivent être
  - « Assemblables » : problème d'intéropérabilité
    - Modèle de composant : définit comment un composant est construit, comment on assemble des composants, comment on les *« package »* [et déploie]
    - Compatibilité des composants
  - Remplaçables/interchangeables : maîtrise du couplage
    - Indépendance de l'implantation

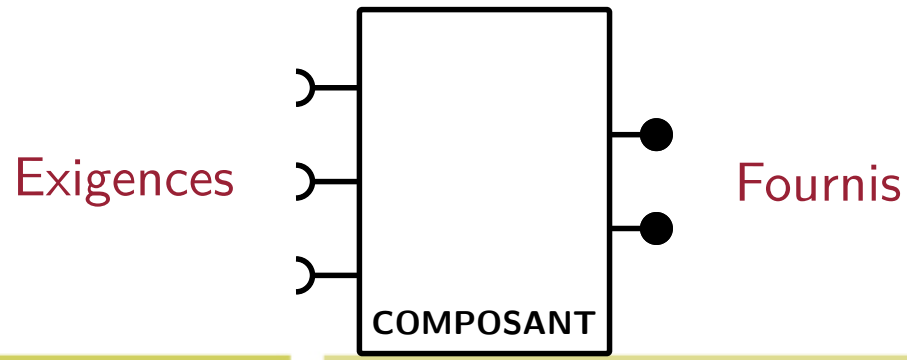
Développer et assembler des composants

- Construire des applications par assemblage de « briques » logicielles réutilisables
- Pour l'assemblage, les briques doivent être ...
- Pour la réutilisation, les briques doivent
  - « Exposer » ce qu'elles offrent et ce dont elles ont besoin
    - Compatibilité des composants = ce qui est offert par un composant est « compatible » avec ce qui est requis par un autre

Vers une définition de composant

- Unité logique (pas forcément de déploiement)
- Configurable, Réutilisable, Composable
- Qui possède une description explicite :
  - De ce qu'elle fait
  - De ce dont elle a besoin

  + ou - formelle (documentation, contrat, ...)
- Et qui est faite pour être assemblée (et désassemblée ...)



Modèle de composant : description

- Partie du modèle de composant qui spécifie ce qu'un composant fait, son contrat
- Différents « *types* » de contrat selon le modèle de composant
  - Syntaxique : assure la compilation
  - Sémantique (effet de l'invocation) : assure la compatibilité fonctionnelle pour une requête
  - Synchronisation : assure la compatibilité fonctionnelle en cas de concurrence
  - Qualité de service : tout ce qui est non fonctionnel (performances, fiabilité, ...)

Exemple

BankAccount $\{balance \geq lowest\}$

`deposit(amount : Money)`
$\{pre : amount > 0\}$
$\{post : balance = balance@pre + amount\}$
`withdraw(amount : Money)`
$\{pre : amount > 0 \wedge amount \leq balance - lowest\}$
$\{post : balance = balance@pre - amount\}$

Lifecycle = init.(`deposit` + `withdraw`)*.close
$Response - time($`deposit`$) < 1s$ when $system - users < 1000$ $Response - time($`withdraw`$) < 1s$ when $system - users < 1000$ Availability(`BankAccount`) all days from 1:00 to 0:00

Utilisation des contrats

- Expression
  - Contrat syntaxique : langages - types (IDL, ...)
  - Contrat sémantique : OCL, assertions, Eiffel, JML
  - Contrat de synchronisation : Automates, Protocol State Machine, logiques temporelles, ...
  - Contrat de qualité de service : QML
- Assembler = Vérifier des contrats
  - Entre composants
  - Vis-à-vis de la plate-forme d'exécution des composants (logiciel qui implante les mécanismes permettant l'assemblage/desassemblage de composants)

Exemple : contrat sémantique

```
1  public class BankAccount {
2      private int lowest = 0 ;
3      private int balance ;
4      //@invariant balance >= lowest ;
5      //@ensures balance == lowest ;
6      public BankAccount() { ... }
7      //@requires amount > 0 ;
8      //@ensures balance == \old(balance) + amount ;
9      public void deposit(int amount) { ... }
10     //@requires amount > 0 ;
11     //@ensures balance == \old(balance) - amount ;
12     public void withdraw(int amount) { ... }
13 }
```

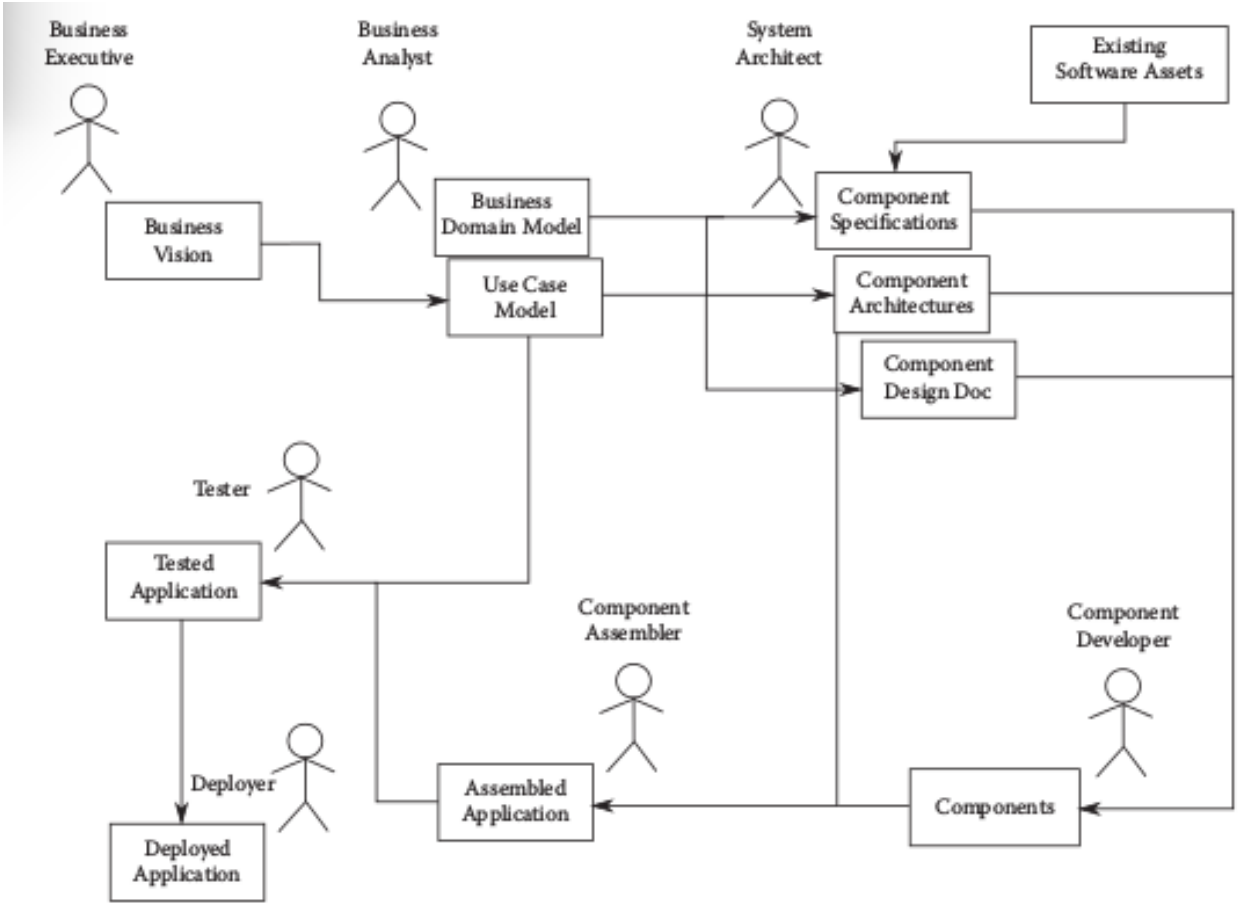
La taille d'un composant

- Gros composants
  - L'application, la bibliothèque
  - Spécialisé pour un problème (risque de ne pas être adapté)
  - Beaucoup de choses réutilisées, mais pas souvent
- Petits composants
  - La macro, la procédure, la fonction, la classe
  - Peu de choses réutilisées, mais fréquemment
  - Beaucoup d'assemblage à faire
- Besoin de plusieurs niveaux de réutilisation : hiérarchie de composants

Ingénierie des composants

- Eléments : composants, [connecteurs]
- Activités :
  - Spécification
    - Quels composants, quel contrat
  - Approvisionnement
    - En fonction de la spécification, chercher des composants existants et en développer si besoin
  - Développement
    - Des traitements / boîtes (composants)
    - De colle (connecteurs) moins évident
  - Assemblage (architecture et configuration)
    - Choix (des boîtes, de la colle)
    - Liaison
  - Test, déploiement, monitoring, ...
  - Gestion des « entrepôts »

Ingénierie des composants



Source : *Component-Oriented Development and Assembly*, Infosys Press, 2013

Différents niveaux d'abstraction

- Composant d'analyse (frontière, exigence et fournis en terme de service)
- Composant de conception (spécification des contrats => un type de composant)
- Implantation de composant (suivant une plate-forme ou dans un langage)
- Paquet de composant pour le déploiement
- Instance de composant à l'exécution
- Les frontières peuvent changer...
- Contrat pour les différents niveaux

Type vs instance de composant

- Différence entre l'entité statique (spécifiée, développée, installée) et l'entité dynamique qui s'exécute (et se déploie à l'instanciation) : Modèles différents
- Contrat de type de composant
- Contrat d'instance de composant

Histoire des composants

- Des débuts
  - Idée de Douglas McIlroy, conf de l'OTAN en 1968
  - Mise en place des *pipe* Unix
  - Repris par Brad Cox modèle des composants électroniques
  - A conduit au langage de programmation Objective-C
- Des expériences fondatrices pour les IHM
  - Smalltalk Parts (90)
  - Microsoft avec OLE et COM (90)
  - Beans/JavaBeans (2000)
- Et maintenant
  - Pour des architectures : CCM, EJB, .NET, OSGi, SCA ...
  - Avec des méthodes pour le développement de composants : les lignes de produits

Progress

Interfaces

- La notion de composant repose sur le découplage interface/réalisation
- Une forme de contrat syntaxique très simple : la signature des méthodes
- Un composant n'utilise pas directement un objet externe mais dépend de ses interfaces requises
- Un composant ne fournit pas de référence interne aux objets externes mais uniquement à travers son interface

$\Rightarrow$ Indépendance contrat/réalisation du contrat

Factory

- Pas de constructeurs pour ne pas créer de dépendance avec l'implémentation
- Des composants (ou objets) qui construisent des composants
- Souvent à partir d'un modèle (*template*)
- Pour les composants distribués : *Locator*
- Mais dépendance sur la *Factory*

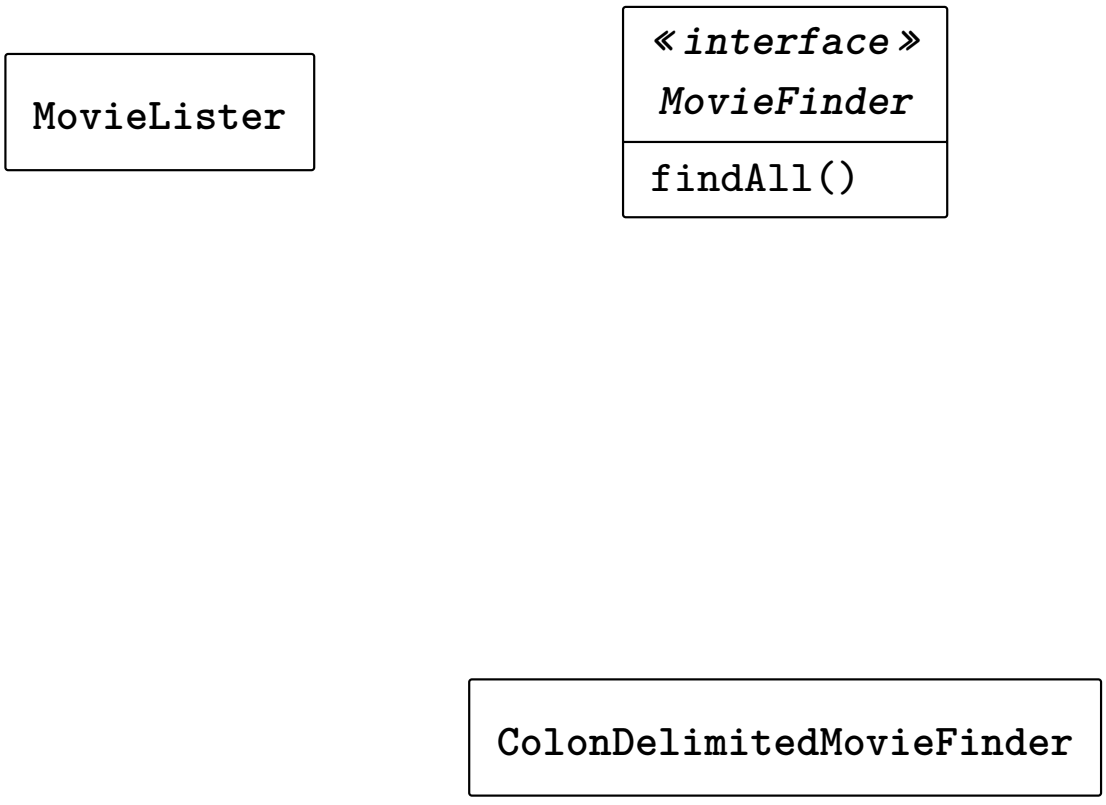
Un exemple : lister films d'un réalisateur

- Composant `MovieLister` et *Factory*

```
1  private MovieFinder finder ;
2  public MovieLister() {
3    finder = new ColonDelimitedMovieFinder("movies1.txt") ;
4  }
5  public List<Movie> moviesDirectedBy(String arg) {
6    List<Movie> allMovies = finder.findAll() ;
7    List<Movie> returnedMovies = finder.findAll() ;
8    for (Movie movie  : allMovies)
9      if ( !movie.getDirector().equals(arg))
10       returnedMovies.remove(movie) ;
11   return returnedMovies ;
12 }
```

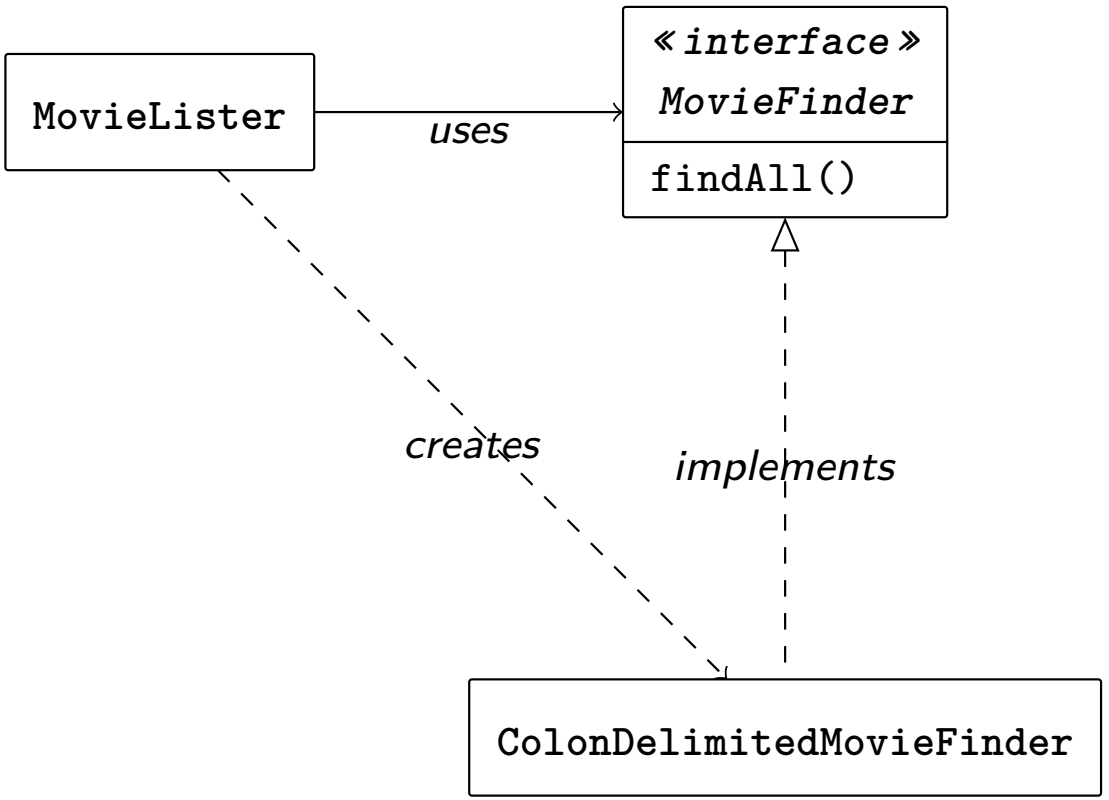
Un exemple : lister films d'un réalisateur

- Dépendances



Un exemple : lister films d'un réalisateur

- Dépendences

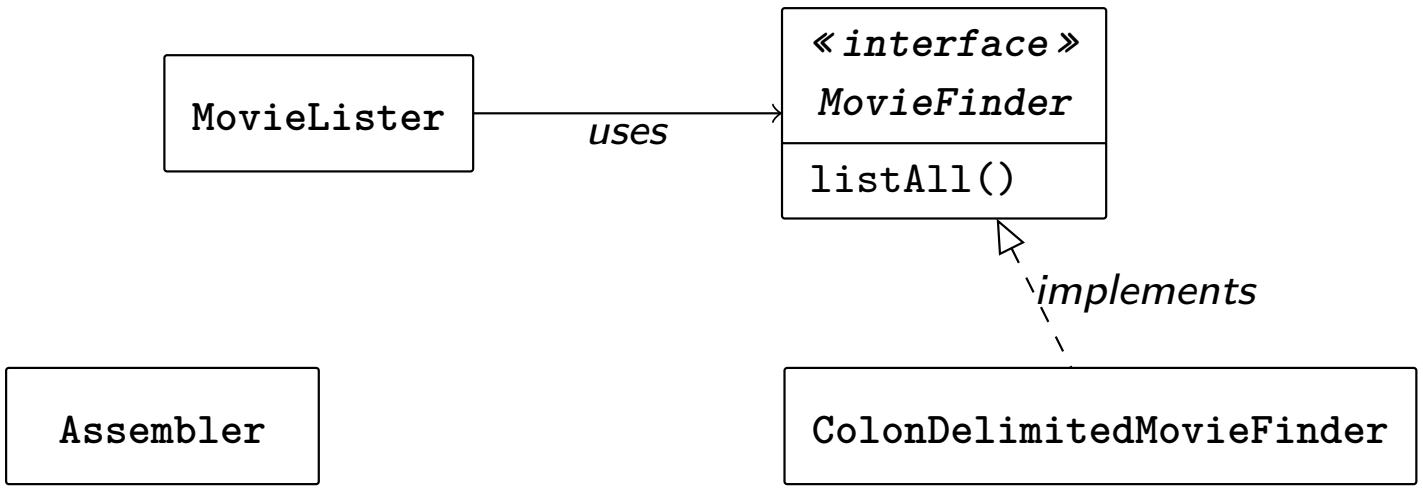


Inversion du contrôle

- Principe d'Hollywood
- Ce n'est pas l'applicatif qui appelle le support mais l'inverse
- Repose sur des conventions ou des interfaces
- L'applicatif s'engage à fournir des méthodes que le support s'engage à appeler
- $\Rightarrow$ contrat entre le composant et son support
- Exemple : le composant fournit une méthode `set`

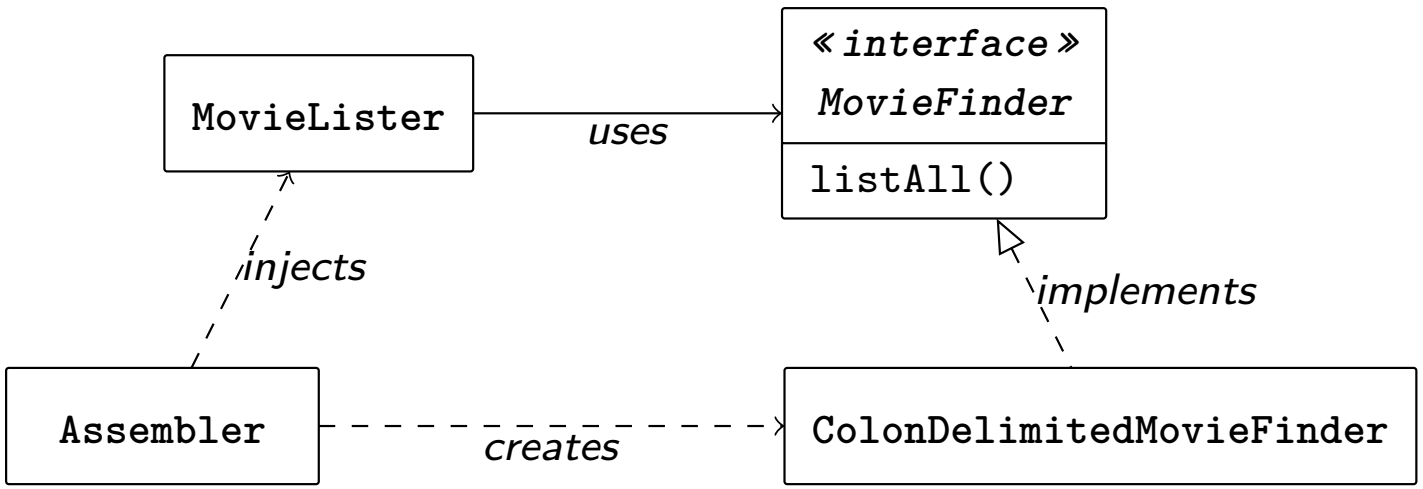
Injection de dépendances

- Réalisation de l'inversion de contrôle
- Très populaire dans les systèmes comme Spring ou EJB
- Le support crée l'instance



Injection de dépendances

- Réalisation de l'inversion de contrôle
- Très populaire dans les systèmes comme Spring ou EJB
- Le support crée l'instance



Injection de dépendances

- Résultat : aucune dépendance directe
- L'implémentation à injecter doit être configurée quelque part ...
- Pour aller plus loin :
  - Article « référence » de Martin Fowler : `martinfowler.com/articles/injection.html`
  - Article « critique » : `www.improve-technologies.com/blog/2011/07/29/le-syndrome-de-linversion-de-controle/`

Interceptor

- Les indirections (permet de modifier dynamiquement)
- Peuvent mémoriser plusieurs fournisseurs (selon une clé par exemple)
- Peuvent avoir du comportement (filtrer, convertir, effectuer un pré ou un post traitement)
- Fournit un point de contrôle
- Combiné systématiquement avec les interfaces $\Rightarrow$ Conteneur / membrane
- `www.masterthebоss.com/java-ee-16-articles/177-ejb-interceptors-in-depth.html`
- `www.byteslounge.com/tutorials/java-ee-add-cdi-interceptor-programmatically`

Progress

Où en est-on ?

- Des premiers produits industriels :
  - Sun : Java Beans => EJB
  - Microsoft : COM => DCOM => COM+ => .NET
- Du travail de normalisation :
  - OMG avec CCM (Corba 3)
  - OSGi
  - OASIS avec Web Services et SCA
- Des modèles plus avancés dans les laboratoires :
  - Fractal, openCOM
  - Les *Architecture Description Languages*

Entreprise Java Beans

Un ancêtre : les Java Beans

- But : application *client* (souvent graphique) par composition *dynamique* et *graphique*
- Le bean
  - est composé de propriétés (attributs et accesseurs)
  - communique par évènements
  - est persistant (`java.io.Serializable`)
  - ses propriétés peuvent être découvertes par introspection
- Réalisation : un bean est une classe Java
- Bilan, une ébauche de composant !
  - Modèle simple bien adapté pour les composants graphiques
  - Outils de construction et de manipulation des beans
  - Pas de notion d'architecture et uniquement évènements

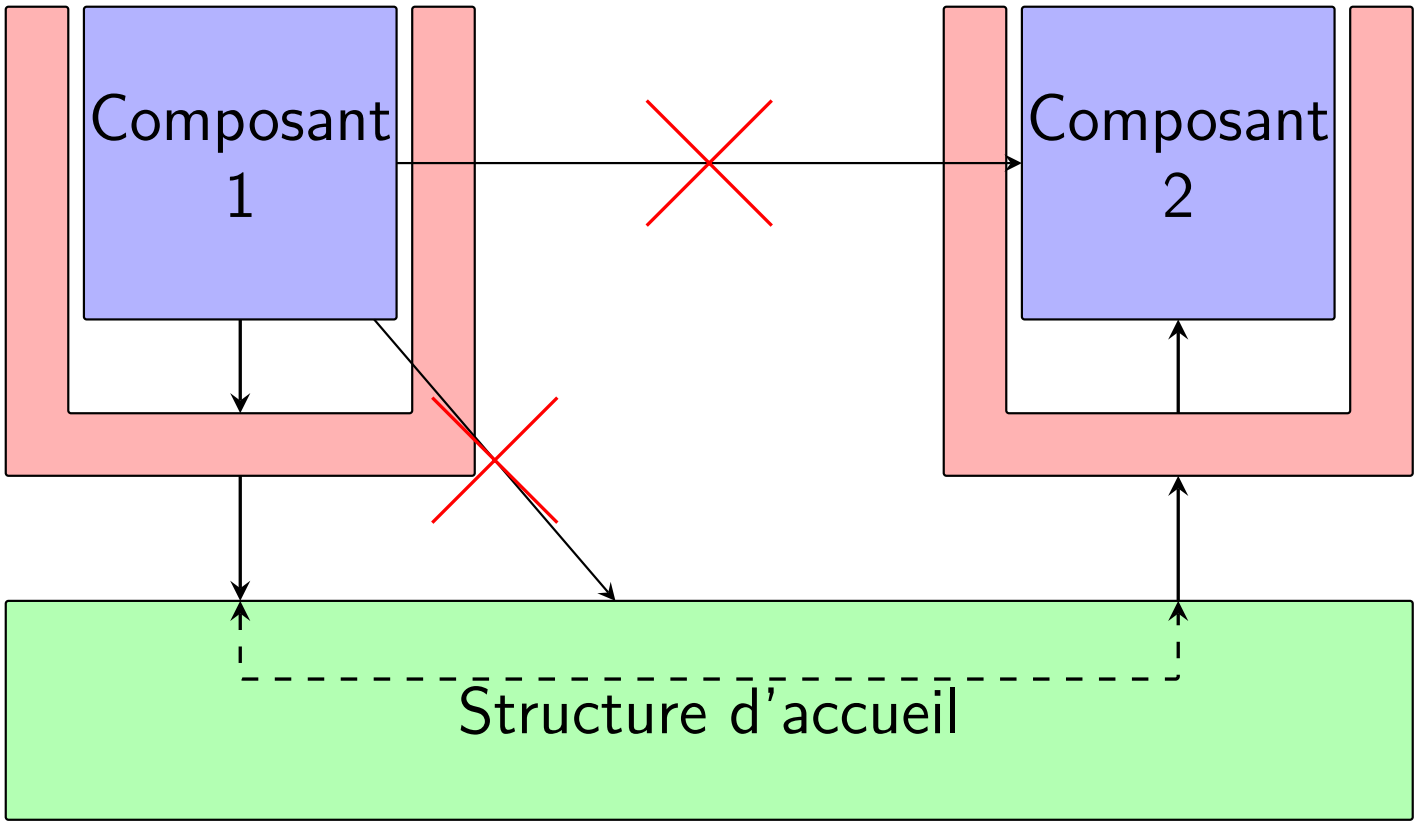
Entreprise Java Beans

- Composant Java pour applications pour serveur
  - Utilisation d'une architecture à trois niveaux (*3 tiers*)
  - Services fournis par plateforme (persistance, transaction, ...)
  - Objectif : une industrie de fabrication d'*Entreprise Beans*
- Concrètement
  - Une spécification (version 3.1 depuis 2008)
  - Un ensemble d'interfaces Java
- Basé sur la notion de conteneur et structure d'accueil
  - But : découplage
    - Éviter code technique dans les composants
    - Pas de lien direct entre composants et avec services plate-forme
  - Rôles : gestion
    - du déploiement, du cycle de vie,
    - des liens/échanges entre composants et avec services plate-forme

Conteneur et structure d'accueil

- Conteneur + structure d'accueil = serveur d'applications
- Structure d'accueil
  - Fournit des services techniques
  - Espace d'exécution des conteneurs et des composants
- Conteneur
  - Encapsule les composants d'une application
  - Intercepte la communication avec composants
  - Ajoute l'utilisation des services techniques

Qu'est-ce qu'un conteneur ? II

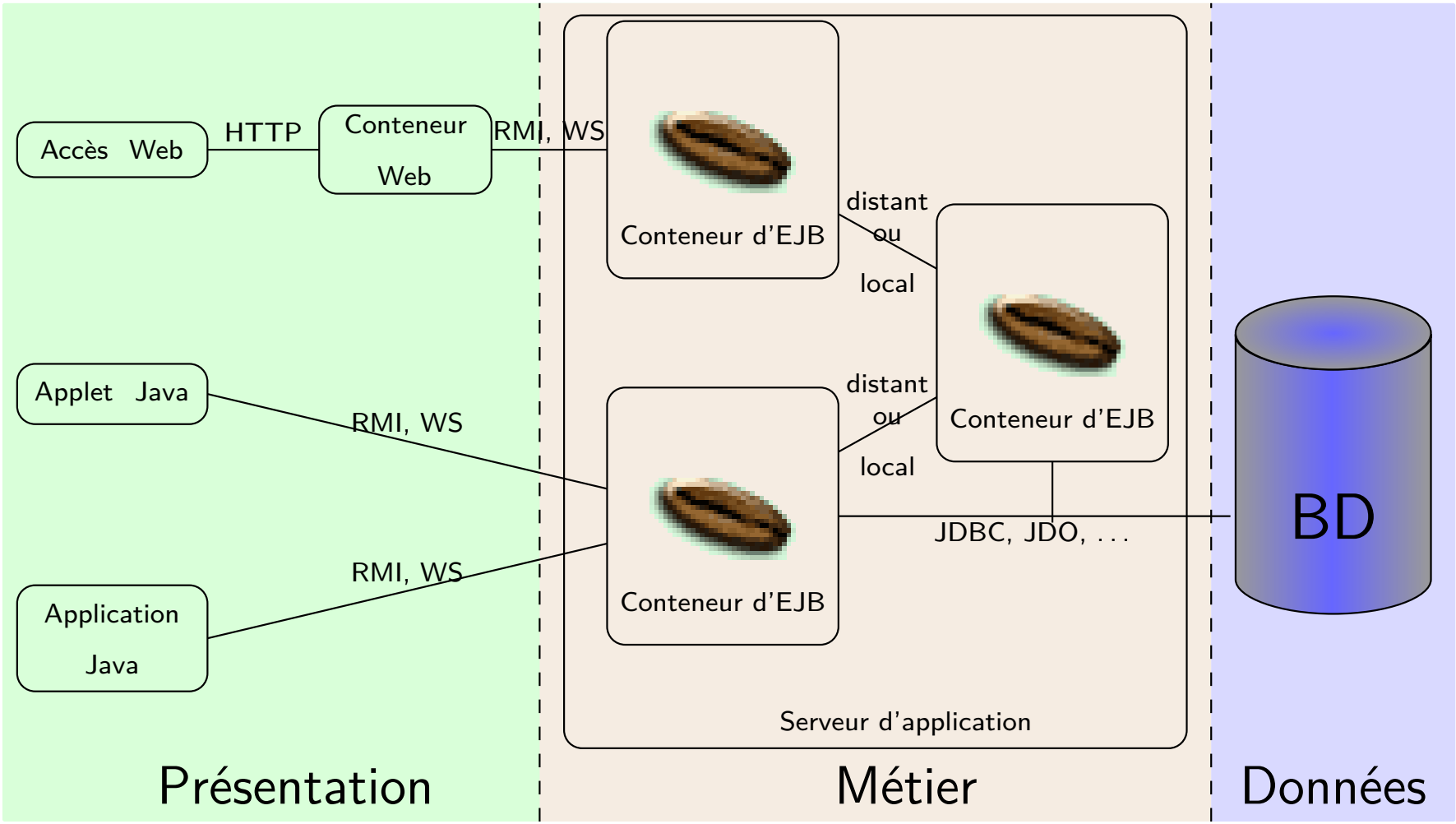


Conteneur et structure d'accueil

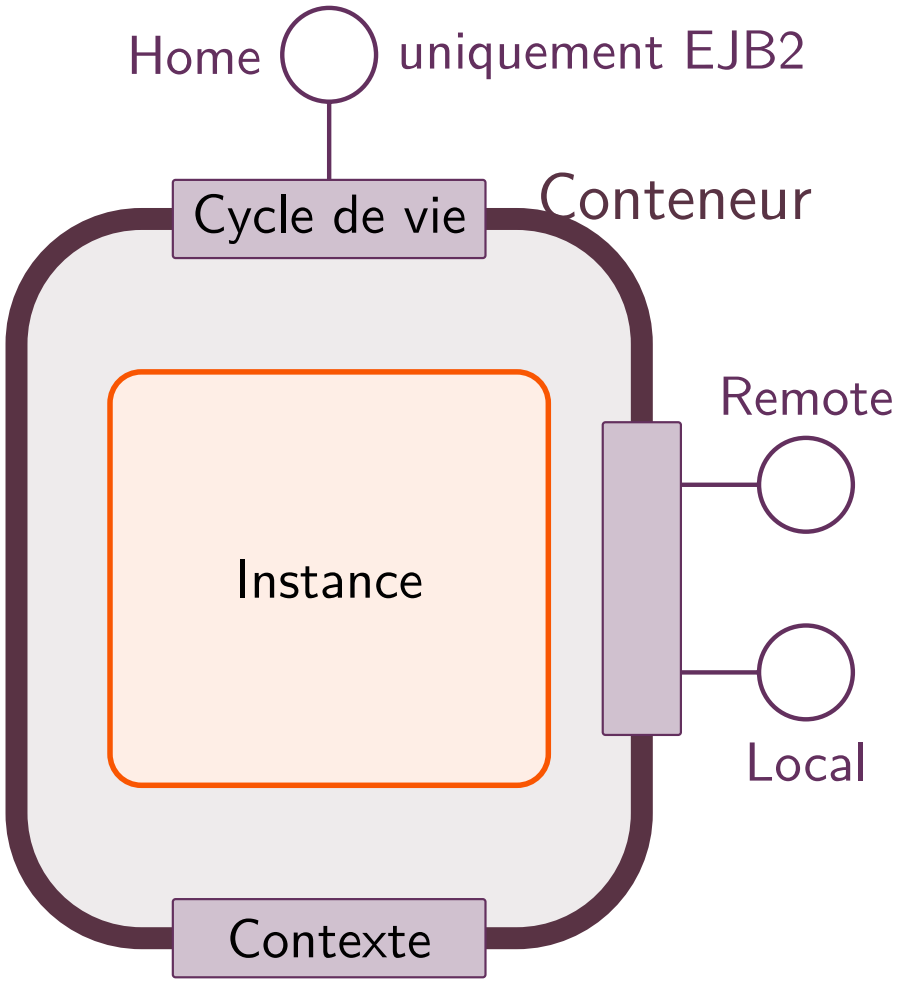
- Pas de lien direct entre composants
- Pas de lien direct entre composants et services techniques
- Limiter le code technique dans les composants

$\Rightarrow$ Remplacer un composant devient plus simple

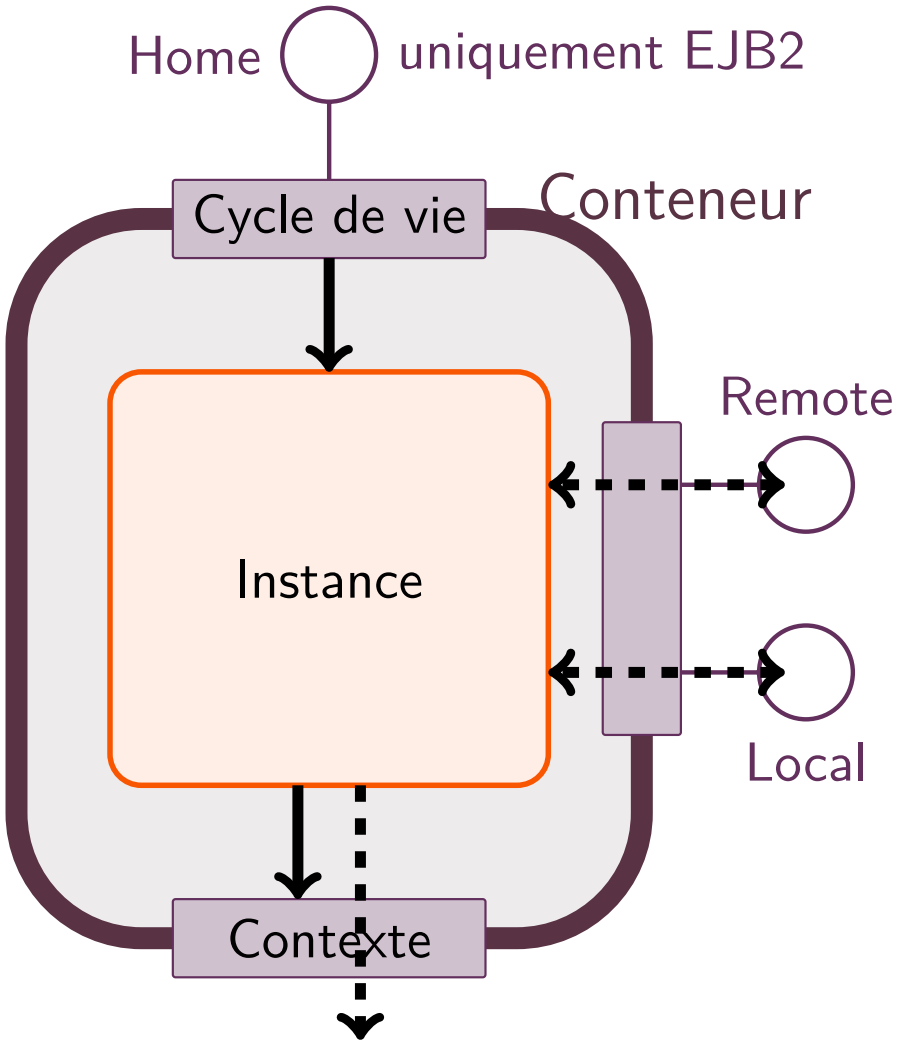
Architecture 3-tier



EJB


Home
uniquement EJB2
Cycle de vie
Conteneur
Instance
Remote
Local
Contexte

EJB



- Interactions par les interfaces
- Conteneur chargé
  - du cycle de vie
  - de l'utilisation des services de la plate-forme
  - réalise les interfaces
- Code
  - généré en EJB2 (nouvelle classe)
  - injecté en EJB3

Un Entreprise Bean

- Est composé de
  - la classe décrivant son comportement
  - une ou deux (ou quatre [1]) interfaces de services offerts
    - les interfaces distante et/ou locale (Remote et Local Interface) qui décrivent ses tâches métiers
    - les maisons local et/ou distantes (Home et LocalHomeInterface) qui décrivent les services de gestion du bean (sa création, sa destruction, sa localisation, ...)
  - des méta-informations
    - un descripteur en XML
    - des annotations dans le code (uniquement EJB3)
    - pour sécurité du *bean*, persistance, transaction

1. EJB2 en Bleu

Les interfaces

- Définition d'une interface distante
  - EJB3 : annotation `@Remote` sur l'interface

```
@javax.ejb.Remote
public interface Hello { public String hello() ; }
```

  - EJB2 : il faut étendre `javax.ejb.EJBObject` ou `EJBHome`

```
public interface Hello extends javax.ejb.EJBObject {
  public String hello() throws java.rmi.RemoteException ;
}

public interface HelloHome extends javax.ejb.EJBHome {
  Hello create() throws java.rmi.RemoteException,
      javax.ejb.CreateException ;
}
```

- Définition d'une interface locale
  - EJB3 : rien à faire
  - EJB2 : il faut étendre `EJBLocalObject` ou `EJBLocalHome`

Trois types d'Entreprise Beans

1. Session (Session) – SB : Modélise les processus métiers, du simple service sans état au processus complexe avec *workflow* (et donc état)
2. Entité (Entity) – EB : Modélise les données métiers persistantes ainsi que leur comportement
   - EJB3, ne sont plus des composants mais des objets usuels (POJO)
3. Orienté message (Message Driven) – MDB : Modélise un processus asynchrone déclenché par la réception d'un message

Un bean session

- Ne survit pas à la fin du processus (d'où leur nom)
- Deux types
  - sans état (*stateless*), il n'a pas d'attribut. Toutes les instances sont interchangeables. ex : calcul du prix
  - avec état (*stateful*), il mémorise l'état conversationnel dans ses attributs. Doit fournir méthodes activation / passivation. ex : achat avec panier dans une appli de e-commerce
- Seuls les beans sans état peuvent être concurrents

```
@javax.ejb.Stateless(name="Hello example", mappedName="ejb/HelloBeanJNDI")
public class HelloBean implements Hello {
  // Méthode pour l'interface Hello locale et distante
  public String hello() {
    System.out.println("Methods hello() from Hello bean activated !") ;
    return "Hello world" ;
  }
}
```

Un client EJB3

```
import javax.naming.* ;
public class HelloClient {
  public static void main(String[] args) throws Exception {
    // initialisation JNDI
    java.util.Properties props = System.getProperties() ;
    Context ctx = new InitialContext(props) ;
    // recherche du bean et reconstruction de son type
    // accès et création du bean à partir de sa maison inutile
    Hello bean = (Hello) ctx.lookup("ejb/HelloBeanJNDI") ;
    // Enifn, on l'utilise
    System.out.println(bean.hello()) ;
  }
}
```

Session avec état

- Instance du *bean* associée à un client (existe tant que le client est présent ou jusqu'à un *timeout*)
- Deux annotations principales
  - `@Stateful` : déclare un *bean* avec état
  - `@Remove` : définit la méthode de fin de session

```
@Stateful
public class CartBean implements CartItf {
  private List<Item> items = new ArrayList<Item>() ;
  public void addItem(int ref, int qte) { ... }
  public void removeItem(int ref) { ... }
  @Remove public void confirmOrder() { ... }
}
```

Un bean entité

- Est persistant même à une panne système
- Correspond généralement à une table d'un SGBDR et une instance de bean à une ligne
- Possède une clé primaire
- Est concurrent (il peut servir plusieurs clients)
- La gestion de la persistance peut être réalisée par
  - le bean lui-même : BMP (EJB2)
  - le conteneur : CMP (EJB2)
  - le conteneur de persistance, JPA (EJB3)
- En EJB3, les entités sont des objets usuels
  - annotation `@Entity`
  - synchronisation explicite `persist` ou implicite attachement

Un composant persistant en EJB3

- Deux modes de définition d'attributs persistants
  - Mode *field*, attribut classique
  - Mode *property*, méthodes `getX` et `setX`
- Types supportés pour les attributs
  - Types primitifs (et leurs *wrappers*), String, Date, etc.
  - Références d'entité (relation ?-1)
  - Collections de références d'entité (relation ?-n)
- Support
  - de l'héritage
  - des classes abstraites
  - des classes internes
- Descripteur de persistance

Le compte en banque EJB3

```
@Entity
@Table(name="ACCOUNT")
public class Account implements java.io.Serializable {
  private int id;
  private String owner;
  private float balance;
  @Id
  @Column(name="ID")
  public int getId() { return this.id; }
  public void setId(int id) { this.id = id; }
  @Column(name="OWNER")
  public String getCustomer() { return this.owner; }
  public void setCustomer(String customer) { this.owner = customer; }
  @Column(name="BALANCE")
  public float getBalance() { return this.balance; }
  public void setBalance(float balance) { this.balance = balance; }
  public Account() { ... }
  public Account(String owner) { ... }
  public String toString() { ... }
}
```

Les finders

- Méthode de la maison (EJB2) ou Annotation `@NamedQuery` (EJB3)
- Méthode `X findByPrimaryKey(String nom)` est générée (nom et type de retour fixe)
- Les autres
  - commencent par *find* en EJB2
  - retourne soit l'interface, soit une collection
  - implantées par une requête EJB QL
    - Dans descripteur de déploiement (EJB2)
    - Paramètre de la méthodes accessible par `?num` ou `:nom`
  - Par exemple
    - `findAll`: `SELECT OBJECT(c) FROM Compte c`
    - `findBySolde`: `SELECT OBJECT(c) FROM Compte c WHERE c.solde>=?1`

Accès au compte en banque EJB3

```
1  @Stateful
2  @Remote(AccountInterface.class)
3  public class AccountBean implements AccountInterface {
4    @PersistenceContext(type = PersistenceContextType.EXTENDED)
5    private EntityManager manager;
6    private Account account;
7    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
8    public void open(int accountID) {
9      System.out.println("Avant");
10     account = manager.find(Account.class, accountID);
11     if (account == null)
12       account = new Account(accountID);
13     manager.persist(account);
14   }
15   @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
16     public float getBalance() {
17     if (account == null)
18       throw new IllegalStateException();
19     return account.getBalance();
20   }
```

Les relations en EJB3

- Définition de relations entre entités
  - Cardinalité 1-1, 1-n, n-1, n-n
  - Gestion optimisée par la base de donnée
  - Possibilité de détruire de façon transitive (*cascade delete*)
- Un exemple

```
1 Class Auteur {
2   String nom;
3   Collection<Livre> livres;
4
5   @OneToMany(mappedBy="auteur")
6   public Collection<Livre> getLivres() {}
7 }
```

```
1 Class Livre {
2   String titre;
3   Auteur auteur;
4
5   @ManyToOne
6   @JoinColumn(name="Auteur_id")
7   public Auteur getAuteur() {}
8 }
```

Le déploiement

- Une application d'entreprise (archive ear) contient
  - un module EJB
  - un module WAR (optionnel)
- Module EJB (tier métier) : archive .jar
  - Descripteur du module (XML)
  - Ensemble des EJBs qui constituent le module
  - Une(des) unités de persistance et les entités correspondantes (optionnel)
- Module Web (tier présentation) : archive .war
  - Descripteur du module (XML)
  - Ressources : html, images, . . . , JSP, .class des *servlets*
- Il suffit alors de déployer l'archive ear au sein du serveur d'application

Bilan des EJB

- Intérêts
  - Très utilisé (forte influence sur les concurrents)
  - Gestion déclarative d'aspects non métier (conteneur)
  - Paquetage et déploiement assez complet
  - Machine d'exécution (le serveur d'application)
- Limites
  - Lourd, très Java et très web
  - Une seule interface métier, pas de composition (d'archi)
  - Beaucoup de règles et peu d'outils pour vérifier leur respect
- Références
  - Glassfish, JBoss, OpenEJB, . . . , Websphere, Oracle 9i, . . .
  - Ed Roman, Scott Ambler et Tyler Powel, « *Mastering EJB* », `http://www.theserverside.com/books/wiley/masteringEJB`
  - `java.sun.com/products/ejb`

ArchJava

Présentation

- Disposer d'un modèle (et d'un compilateur) permettant de rendre exécutable une architecture
- Utiliser un système de types pour vérifier une architecture
  - Vérifier que les composants n'échangent des messages qu'en passant par les liens de l'architecture
  - S'assurer que les données ne sont partagées qu'entre des composants en liaison

Un exemple

```
public component class C1 {
  public port entrée { }
  public port sortie { }
}
public component class C2 {
  public port entrée { }
  public port sortie { }
}
public component class System1 {
  private final C2 c2 = new C2();
  private final C1 c1 = new C1();
  connect c2.entrée, c1.sortie;
  public static void main(String[] args) {
    System.out.println("Welcome to ArchJava");
  }
}
```

Conclusion

Faire des composants

- En s'appuyant sur des modèles et leurs implantations
  - Fractal, EJB, CCM, UML2.0, .NET, etc
  - Avantages, des outils existent
    - Déploiement, vérification d'assemblage (contrat syntaxique), documentation
  - Limites
    - Les modèles de composant sont-ils assez riches ?
    - Verbeux
- En définissant des règles de développement :
  - Ex : découpler spécification et implantation, contrat sémantique, etc
  - Avantages : Adaptabilité aux besoins, Réutilisation (contrainte) des outils actuels
  - Inconvénient : pas (encore) d'outils spécifiques

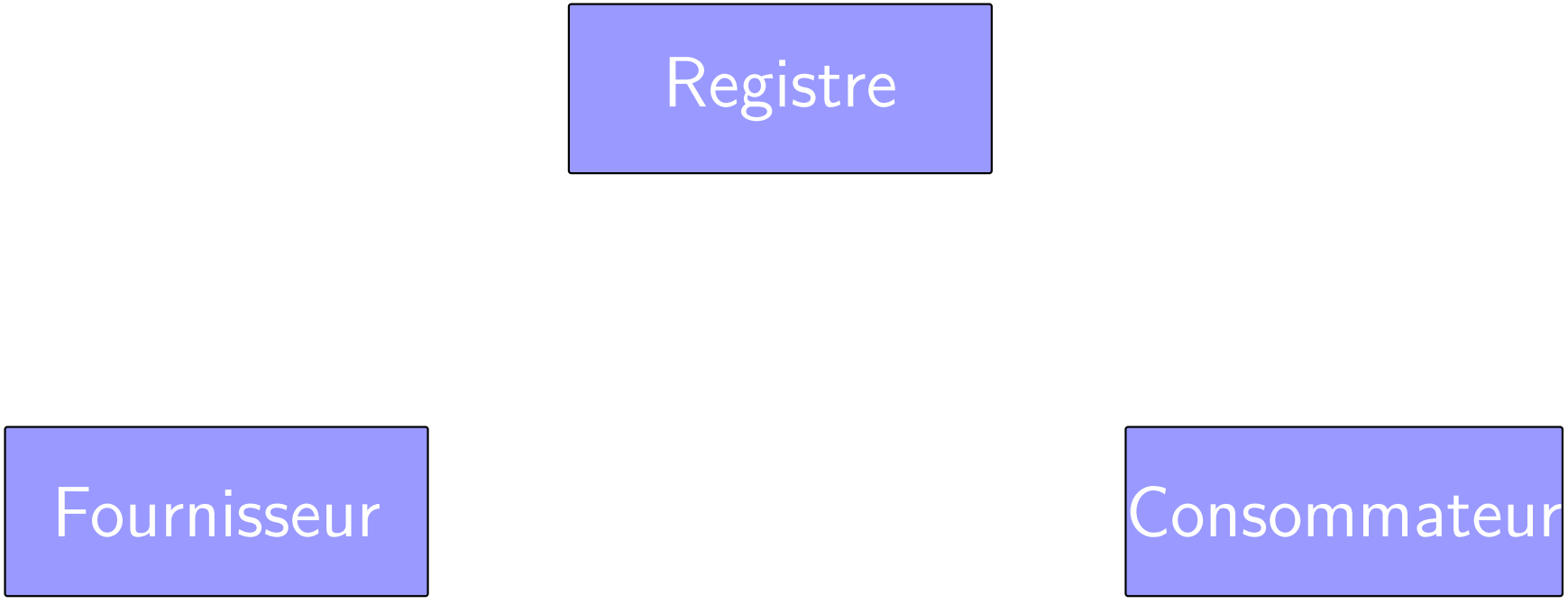
Bilan

- Domaine en plein mouvement
- Pour l'instant, modèles pragmatiques
  - Pas très composant, un peu lourd
  - Bien supportés, plein d'outils
- Pour l'instant, modèles académiques
  - Pas très outillés, peu d'automatisation
  - Des modèles complets et sophistiqués

Progress

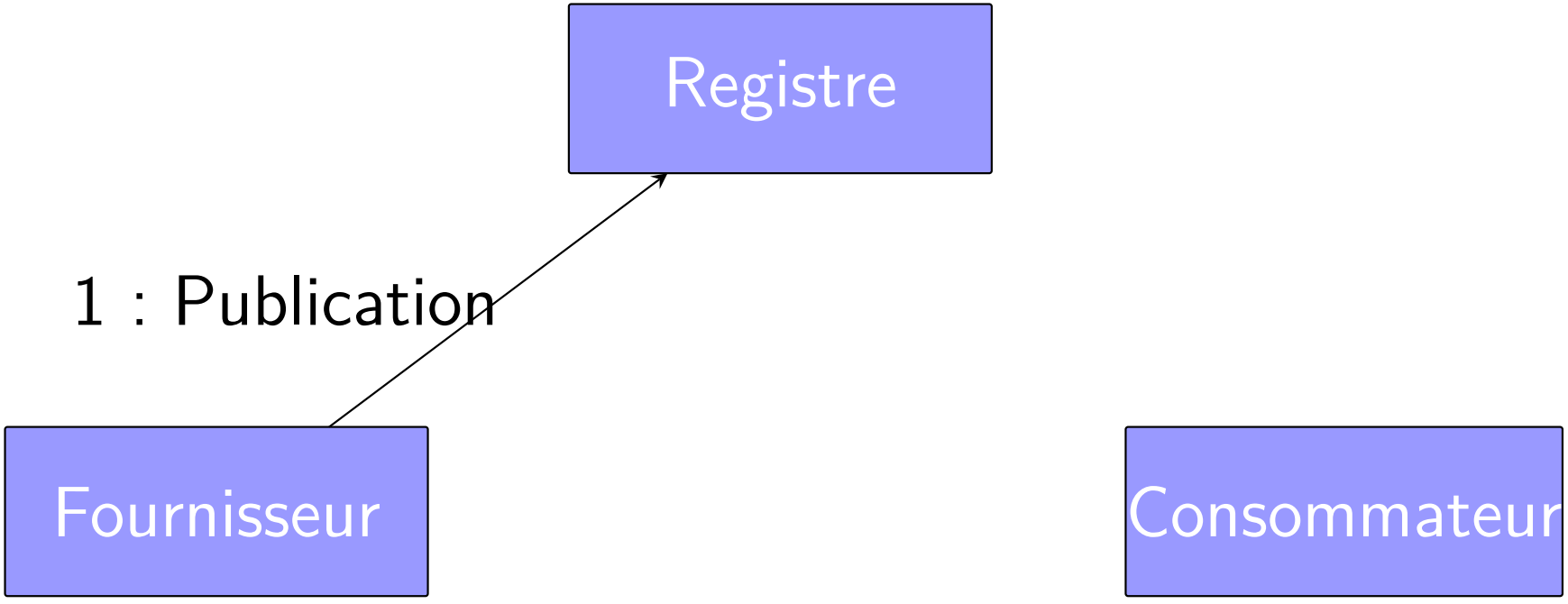
Architectures à base de services

- Modèle d'interaction applicative mettant en oeuvre des connexions en **couplage lâche** entre divers composants logiciels (ou agents)



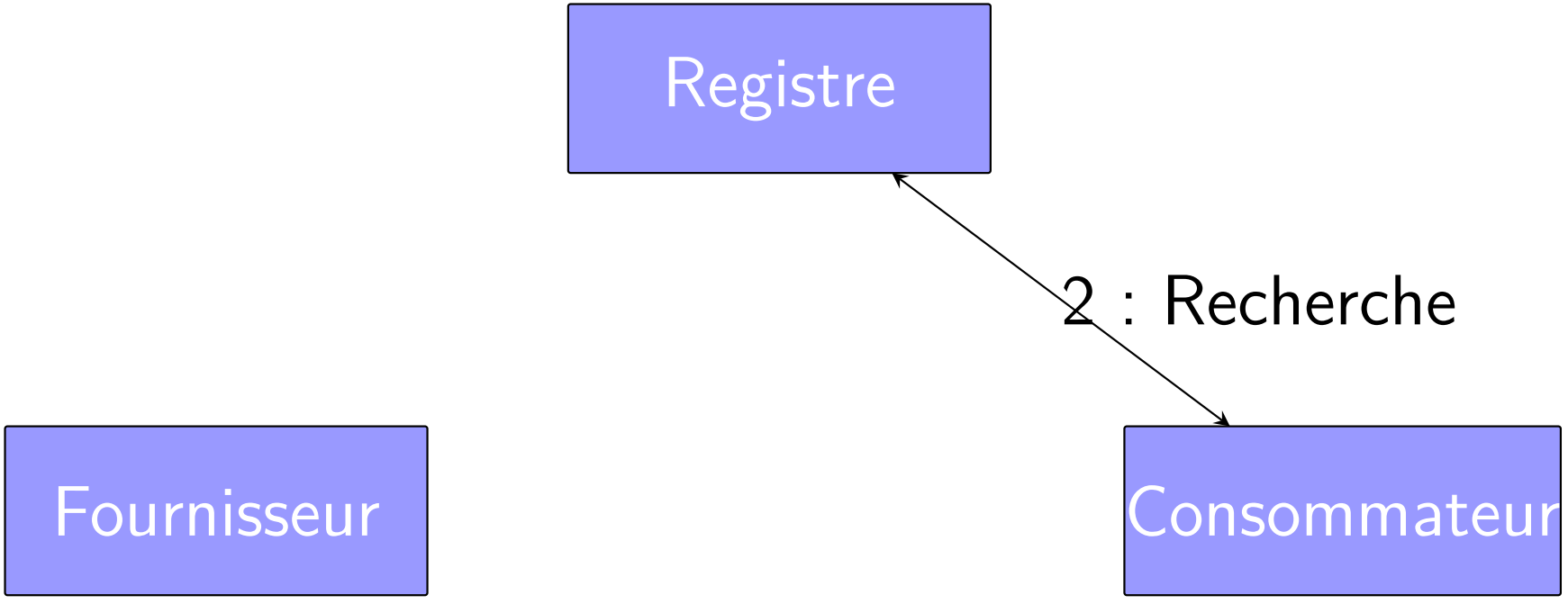
Architectures à base de services

- Modèle d'interaction applicative mettant en oeuvre des connexions en **couplage lâche** entre divers composants logiciels (ou agents)



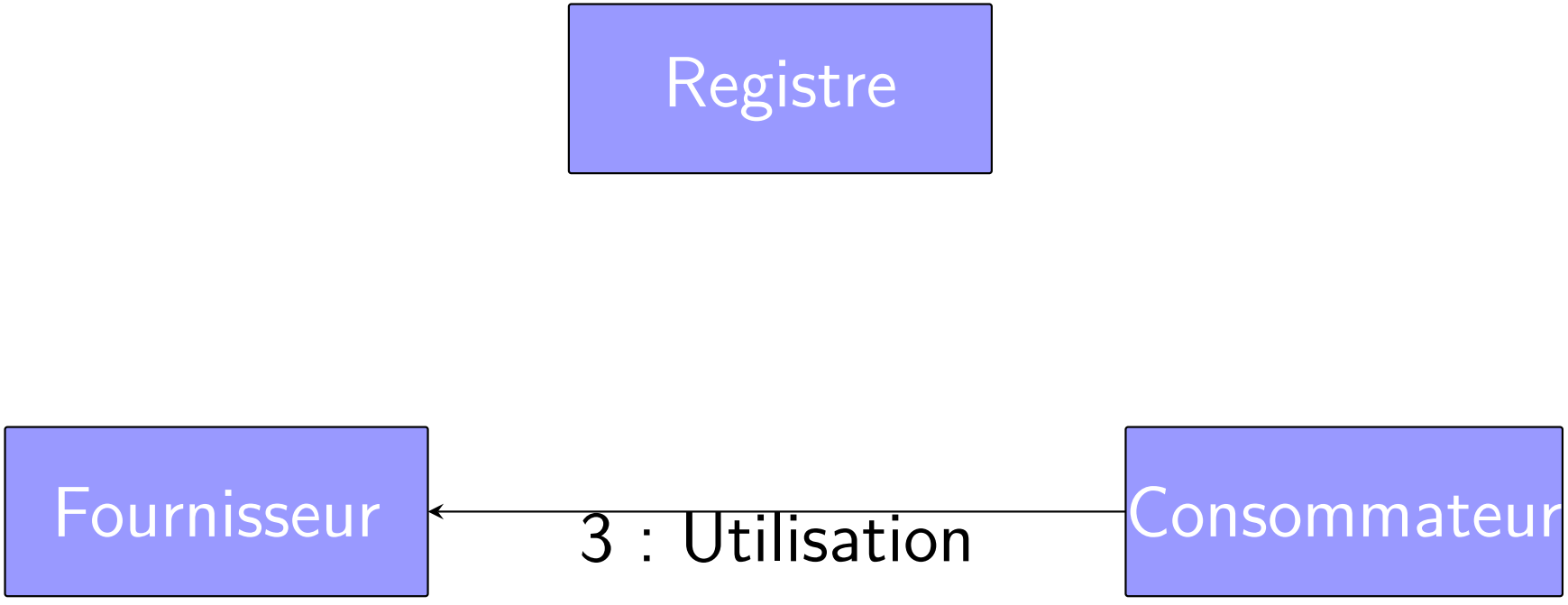
Architectures à base de services

- Modèle d'interaction applicative mettant en oeuvre des connexions en **couplage lâche** entre divers composants logiciels (ou agents)



Architectures à base de services

- Modèle d'interaction applicative mettant en oeuvre des connexions en **couplage lâche** entre divers composants logiciels (ou agents)



Architectures à base de services

- Fournisseurs : définissent les « services » offerts, leur contrat
- Consommateurs : doivent connaître les services qui existent et leur contrat $\Rightarrow$ Registre (annuaire de services)
- Accord dans la définition des interfaces, format d'échanges, protocoles de transport... $\Rightarrow$ Standardisation

Conséquence d'un couplage lâche

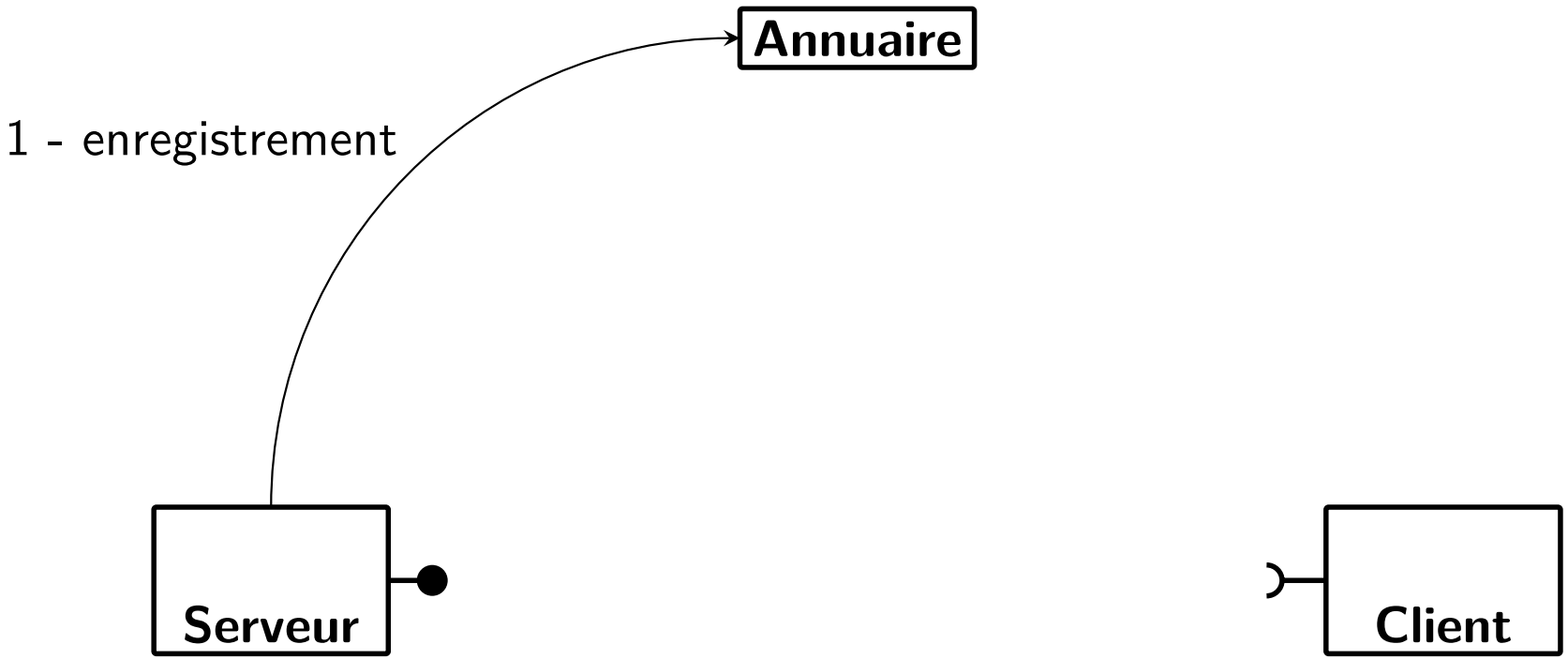
- Fournisseurs substituables
  - Consommateur ne connaît pas l'identité du fournisseur, uniquement son contrat
  - Pas de lien direct sur le fournisseur dans un consommateur
- Les échanges entre fournisseurs/consommateurs et annuaire doivent être normalisés
- Vocabulaire de description de contrats connu par les deux parties
  - Interface d'invocation, protocoles d'interaction, données échangées...

Adaptabilité des plate-formes de services

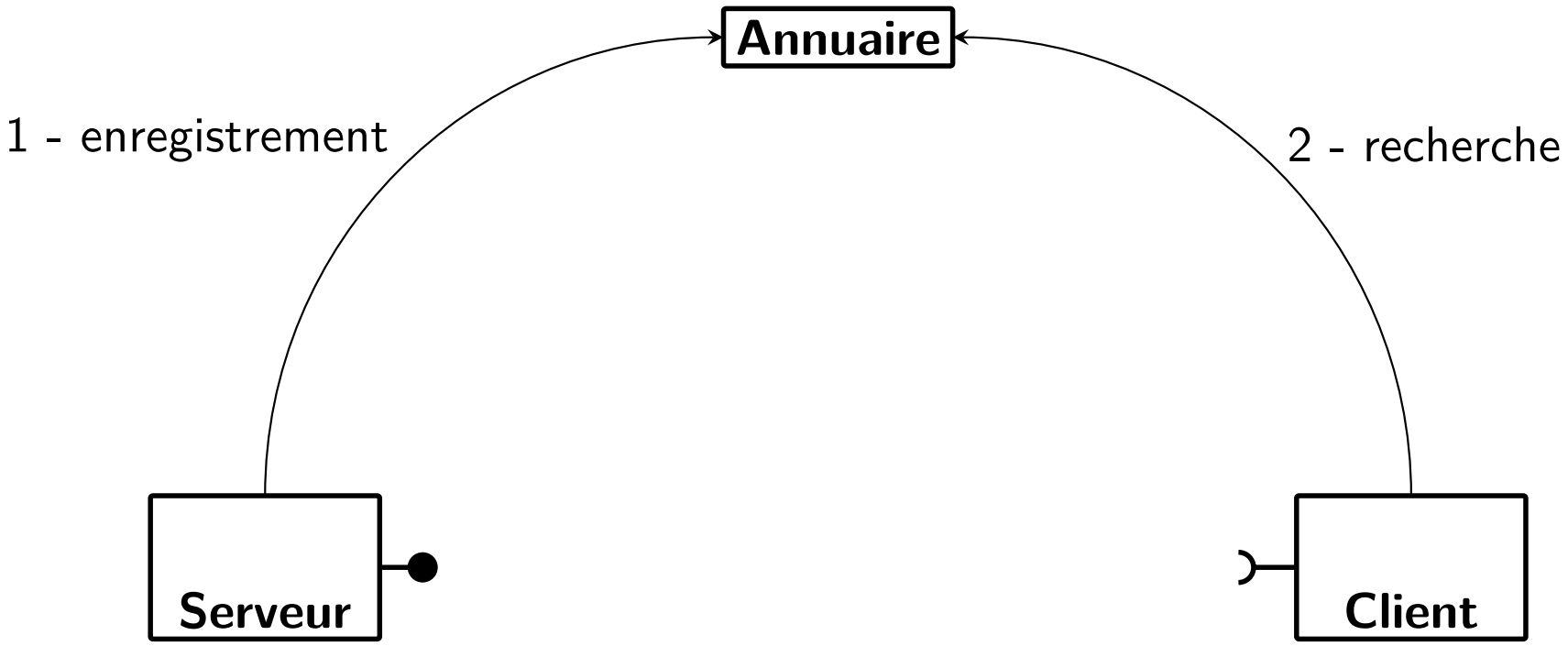
- Plate-forme de services statique : résolution des contrats à la conception
  - – Impossible de sélectionner le fournisseur à l'exécution
  - \+ Facile à programmer, client indépendant de l'implém
- Plate-forme de services dynamique : résolution des contrats au chargement
  - – Impossible de changer de fournisseur en cours d'exécution
  - \+ Facile à programmer
- Plate-forme de services dynamique et adaptable : résolution des contrats au chargement
  - \+ Modification des contrats en fonction des départs et arrivées dynamique des services
  - – Difficile à programmer

Principe

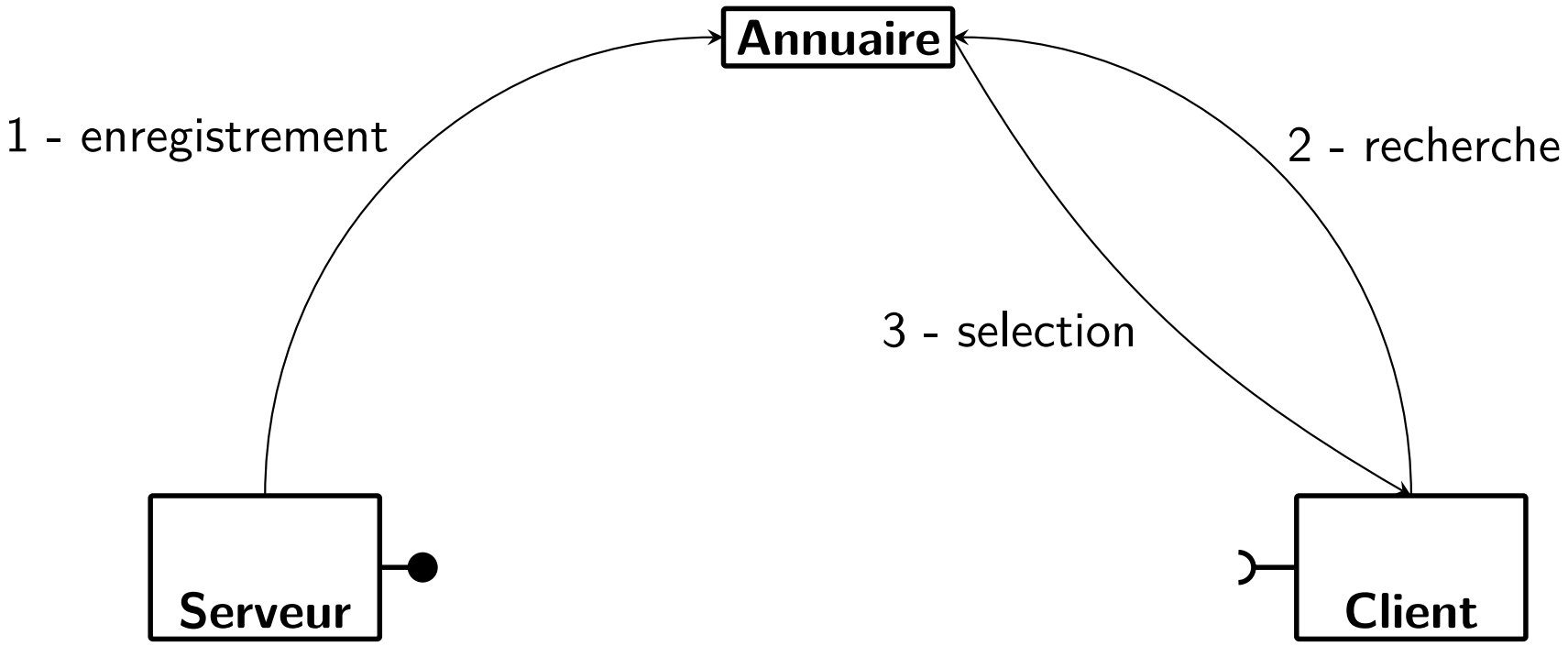
Principe



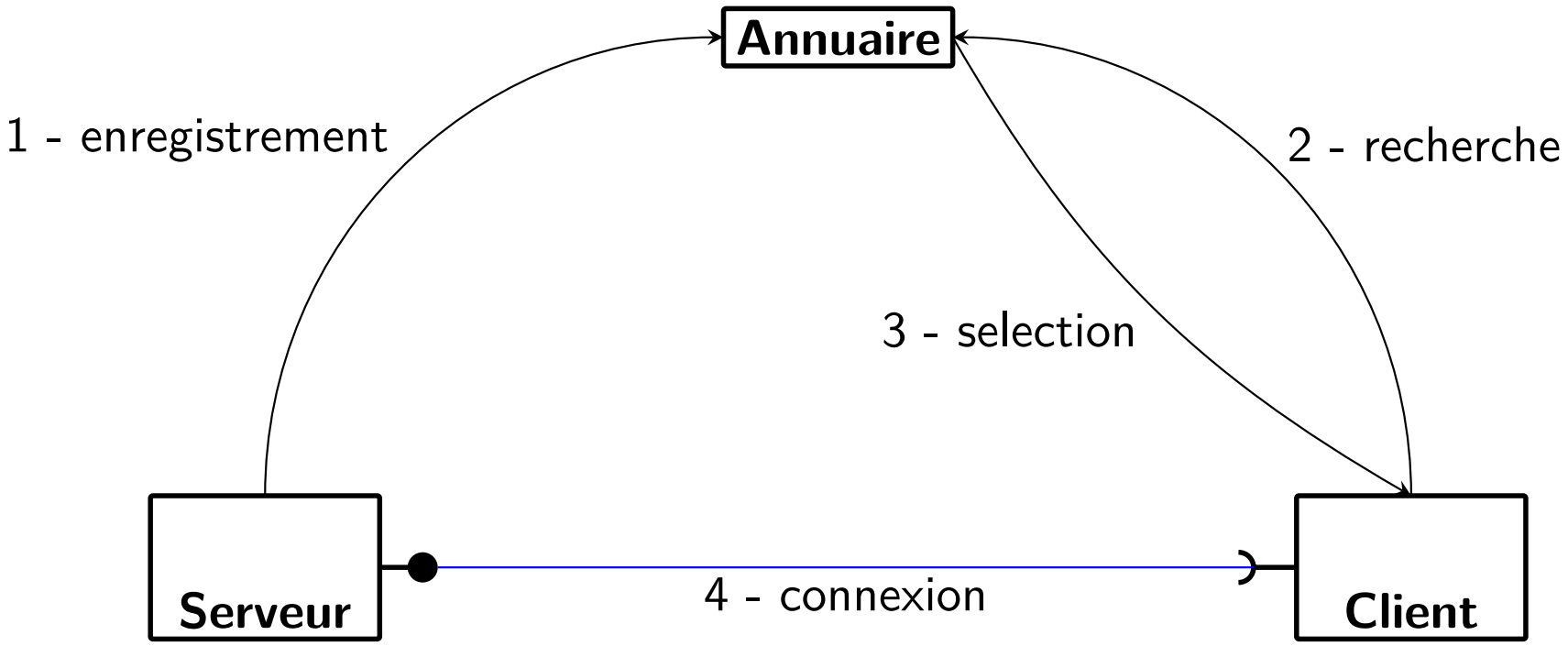
Principe



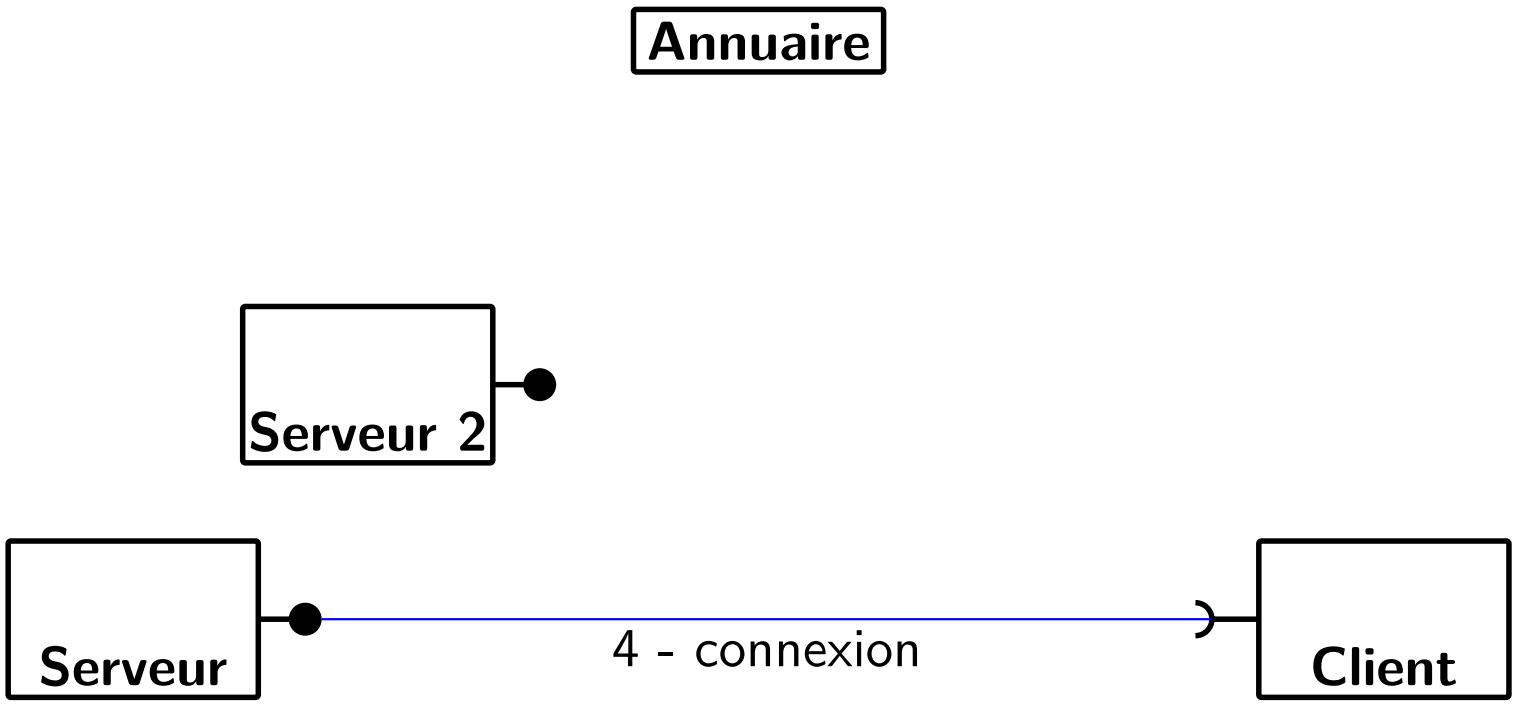
Principe


Annuaire
1 - enregistrement
2 - recherche
3 - selection
Serveur
Client

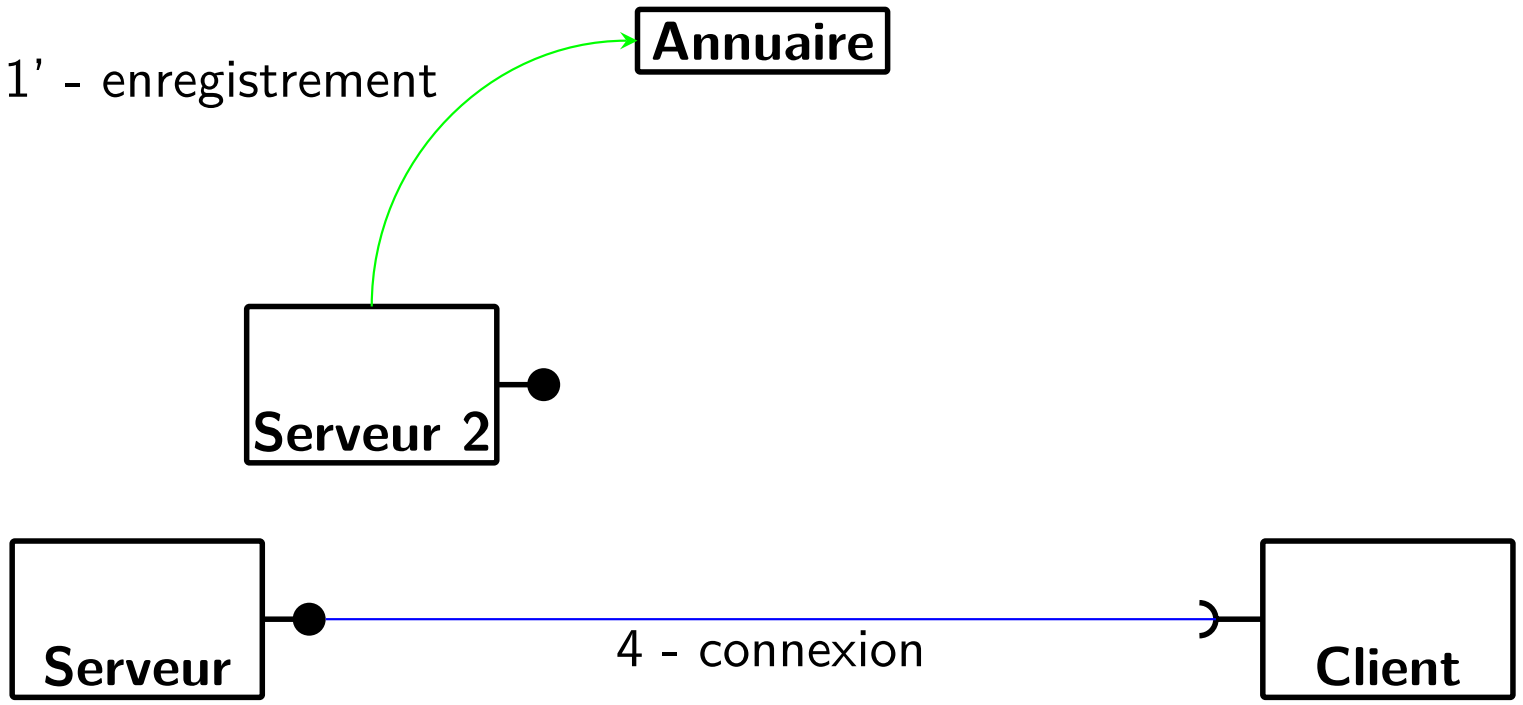
Principe


Annuaire
1 - enregistrement
2 - recherche
3 - selection
Serveur
4 - connexion
Client

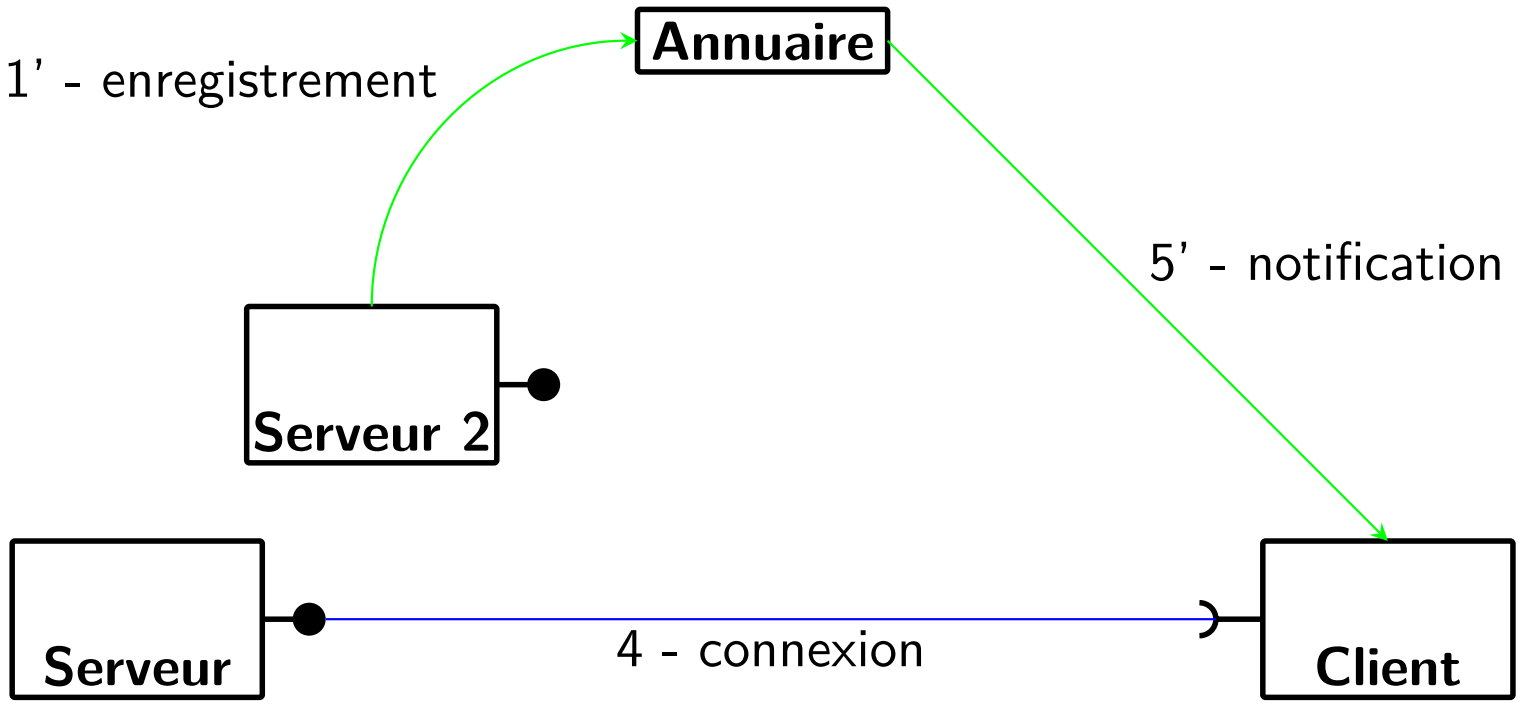
Principe



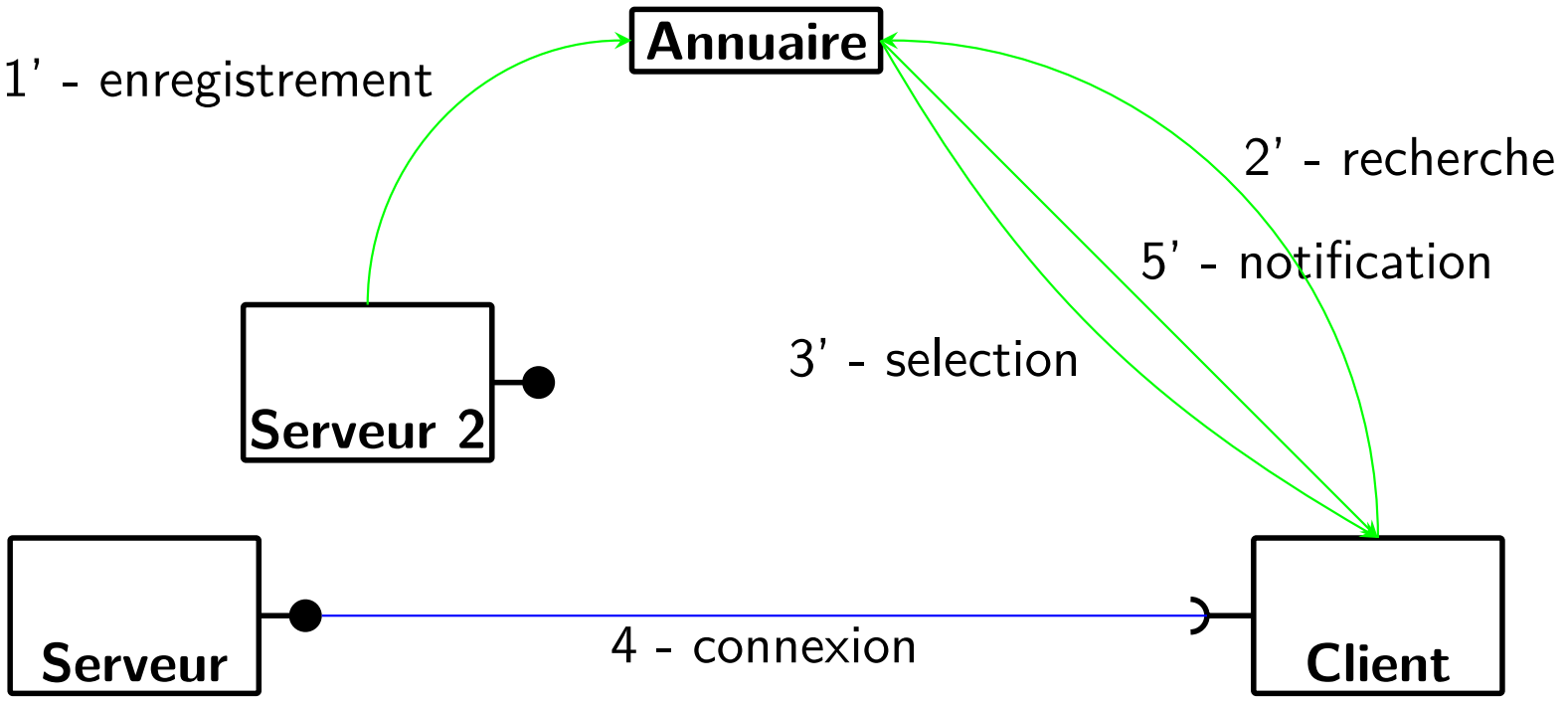
Principe


1' - enregistrement
Annuaire
Serveur 2
Serveur
4 - connexion
Client

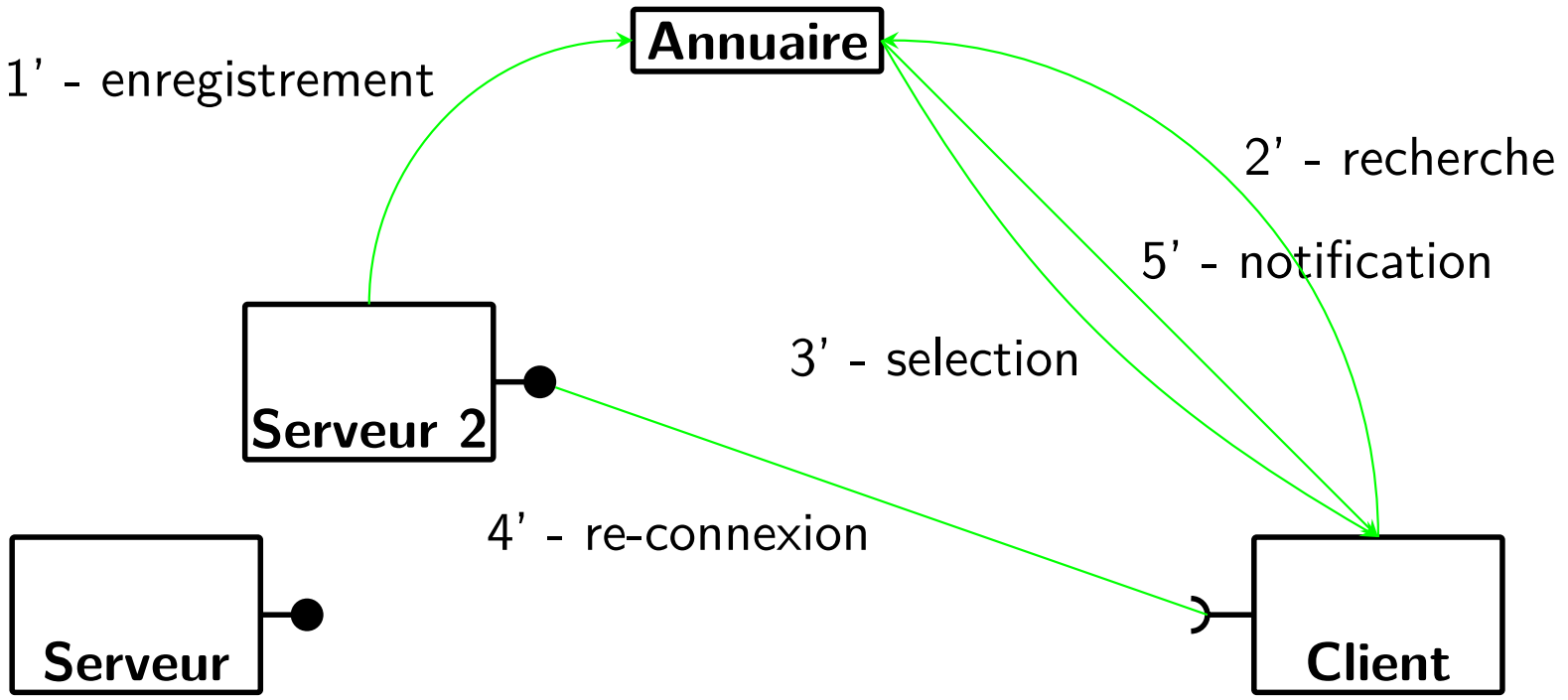
Principe


Annuaire
1' - enregistrement
5' - notification
Serveur 2
Serveur
4 - connexion
Client

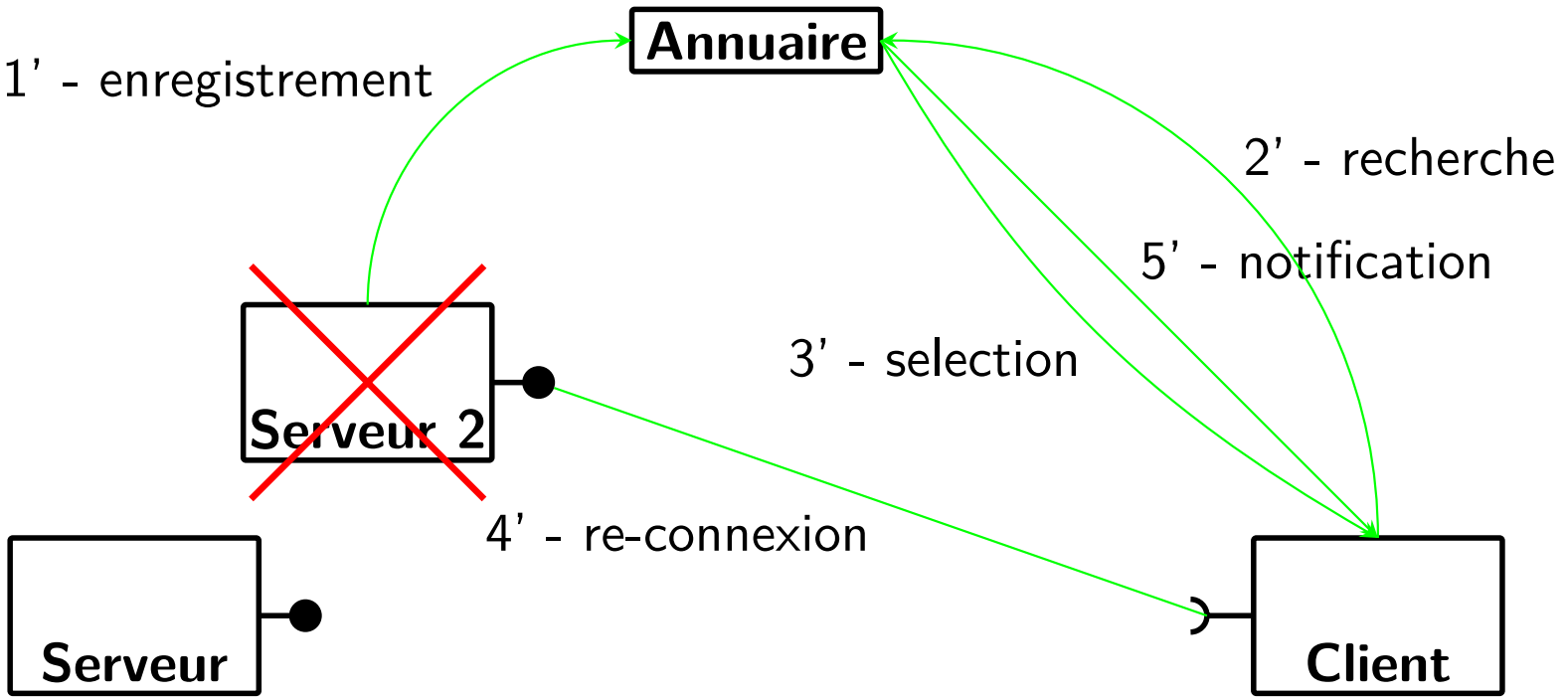
Principe


Annuaire
1' - enregistrement
2' - recherche
5' - notification
3' - selection
Serveur 2
Serveur
4 - connexion
Client

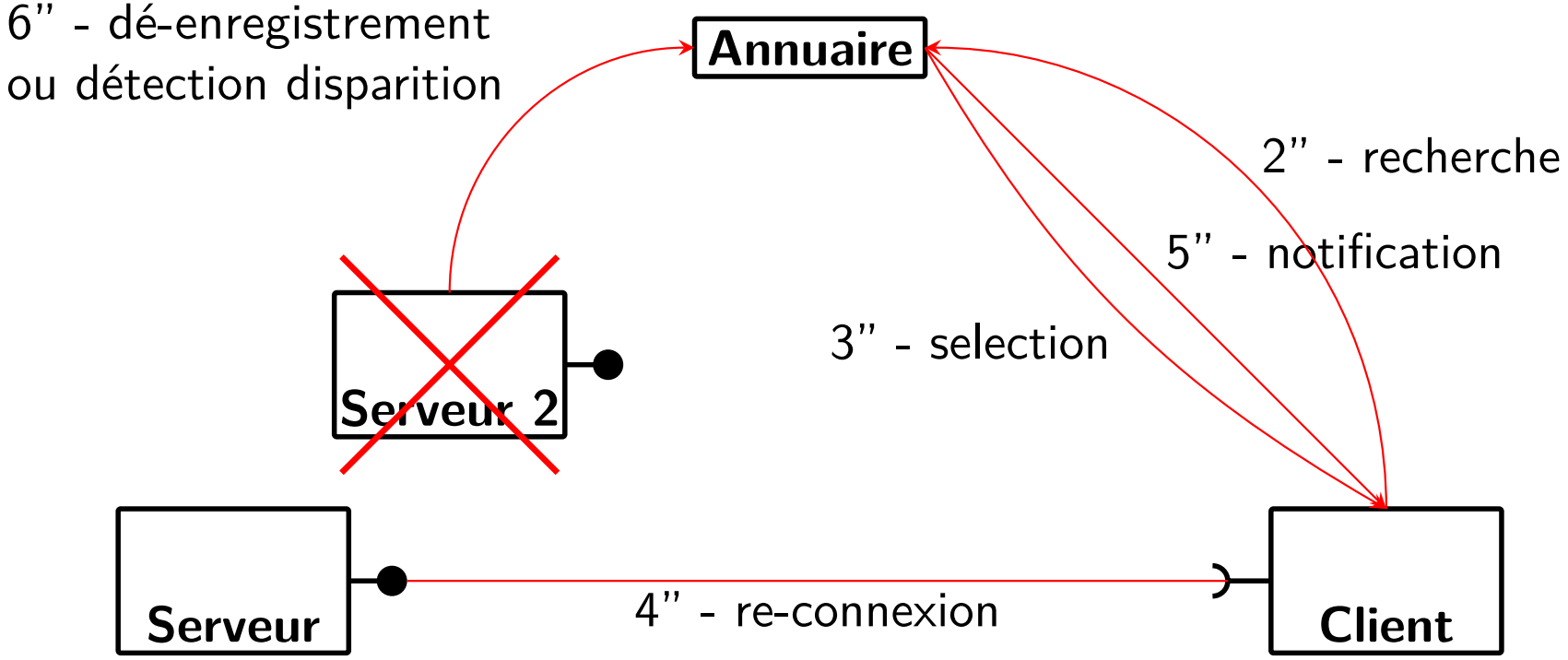
Principe


Annuaire
1' - enregistrement
2' - recherche
5' - notification
3' - selection
Serveur 2
4' - re-connexion
Serveur
Client

Principe


Annuaire
1' - enregistrement
2' - recherche
5' - notification
3' - selection
Serveur 2
4' - re-connexion
Serveur
Client

Principe


6” - dé-enregistrement
ou détection disparition
Annuaire
2” - recherche
5” - notification
3” - selection
Serveur 2
Serveur
4” - re-connexion
Client

Intérêt et contraintes

+ Sépare spécification (service) et implantation
+ Connexion dynamique (pas à la conception)
c Un contrat ne change pas

Architectures à base de services : exemples

- Registre
  - SSDP d'UPnP, Registry d'OSGi, UDDi des services Web SOAP
- Standardisation des interfaces et du format de données
  - Spécifications UPnP, spécifications OSGi (Java), WSDL des services Web SOAP, WADL des services Web REST

Des composants ou des services

- Communauté composants
  - Structuration des traitements
- Communauté services
  - Interaction des traitements (composants avec interconnexions « implicites »)
- Couplage entre les deux
  - Des services structurés à l'aide de composants
  - Ex. spécifications SCA (*Service Component Architecture*) `http://www.osoa.org`

Services Web

Services Web

- Initialement pour intégration d'applications des SI
  - Utilisation d'une seule plate-forme pour gérer l'hétérogénéité des applications en utilisant un « format pivot »
- Mais maintenant
  - Utilisés pour offrir une vue « service » (processus (métier) d'une entreprise)
- Plate-forme de services dynamique

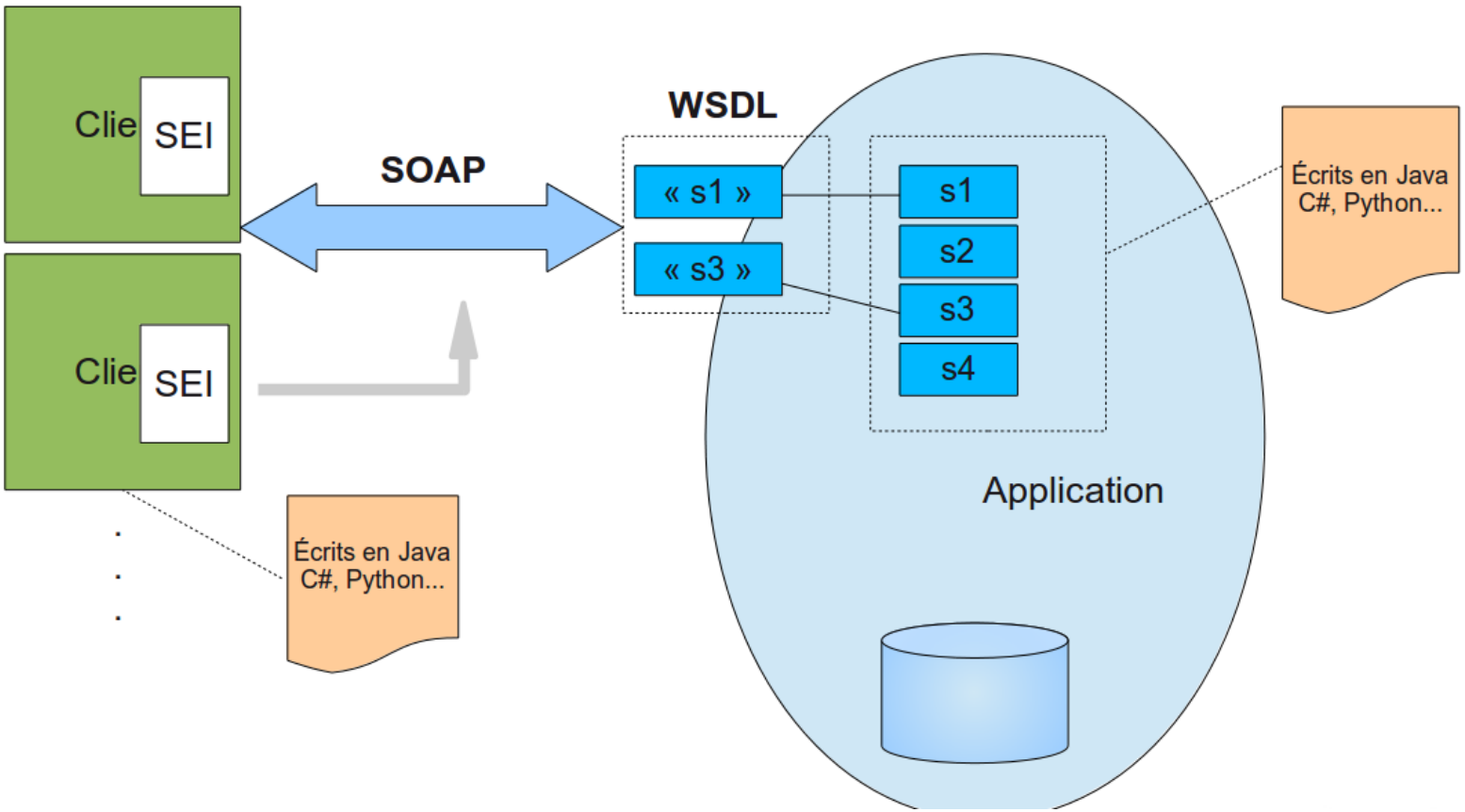
Service Web et contrat

- Service = Traitement métier ou technique (exécuté à distance)
- Services sans état
- Contrat : deux « chapelles »
  - Les services SOAP :
    - Contrat explicite (WSDL)
    - Contrat de base et *« étendu »*
  - Les services REST

Service Web et contrat

- Service = Traitement métier ou technique (exécuté à distance)
- Services sans état
- Contrat : deux « chapelles »
  - Les services SOAP
  - Les services REST
    - Contrat implicite (requêtes HTTP)
    - Contrat de base

Services Web SOAP



Services Web SOAP : technologies

- Spécifications proposées par des consortiums (W3C, OASIS(WS-I))
- À la base HTTP et XML
- SOAP
  - Protocole sur HTTP mais aussi d'autres
  - Appel de méthode à distance (comme Corba, RMI ...)
  - Message SOAP = document XML
- Contrat et données échangées codés en dialecte XML : WSDL
- Annuaire de services : UDDI (*deprected*)

Bilan des services Web

- Intérêts :
  - Indépendance du langage et distribué
  - Notion de qualité de service très poussée (SOAP)
- Limites :
  - Pas vraiment de notion d'architecture
  - Pas de dynamicité
  - Lourd et complexe à programmer

Services OSGi

OSGi, www.osgi.org

- Plate-forme OSGi
  - Canevas de déploiement et d'exécution de services Java
  - Centralisée (VM unique)
- Consortium OSGi
  - Fondé en 1999
  - Inclut de nombreux acteurs majeurs (IT, téléphonie, Eclipse, Apache...)
- Marchés visés
  - Intialement : passerelles résidentielles (set-top-box)
  - Industrie automobile
  - Téléphonie mobile
  - Contrôle industriel
  - Mais se généralise maintenant ...

Les spécifications et implantations

- Novembre 1999 : transfert du JST008 à OSGi
- Mai 2000 : 1.0 (189 pages)
- Octobre 2001 : 2.0 (288 pages)
- Mars 2003 : 3.0 (602 pages)
- Octobre 2005 : 4.0 (1000 pages)
- Juin 2012 : 5.0 (1400 pages)

`http://www.osgi.org/Specifications/HomePage`

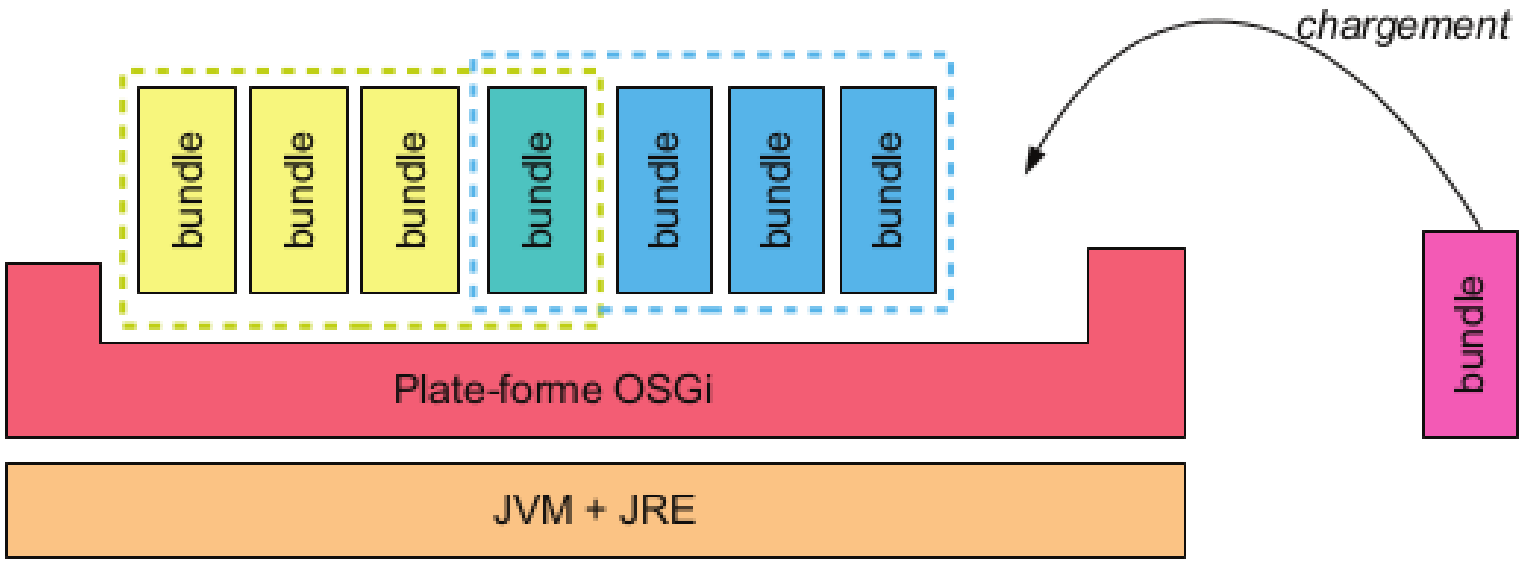
- Principales implantations : Felix (Apache), Equinox (IBM/eclipse), Knoplerfish

Motivations

- Programmation orientée service en Java dynamique et adaptable
  - Notification arrivées/départs de services
- Installation, mise à jour, lancement, arrêt, retrait
- Résolution des dépendances de versions de code

Application OSGi

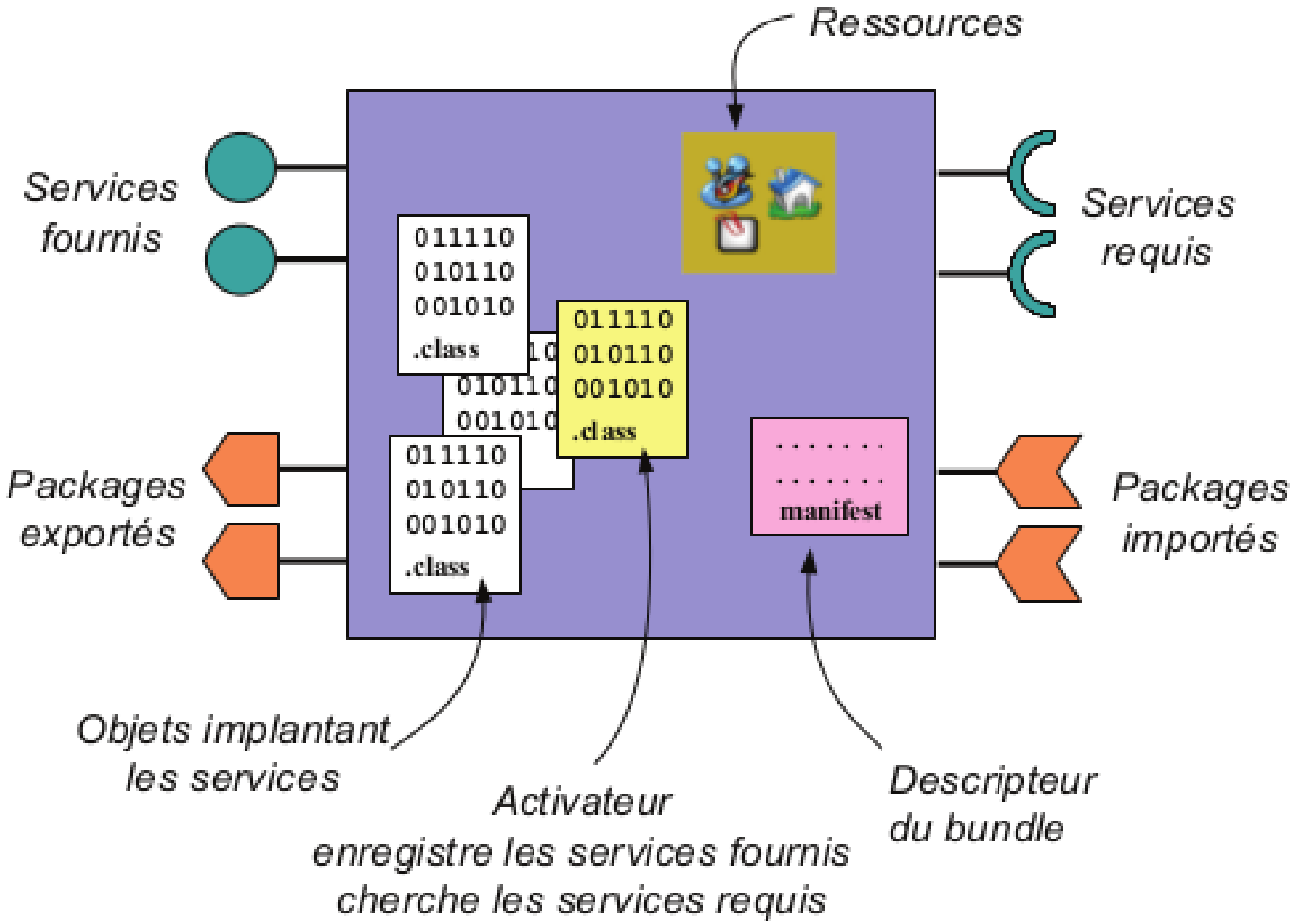
- Ensemble de bundles
  - Livrés dynamiquement
  - Administrés à partir d'une console (ou un autre outil)
    - Lister
    - Déployer ou supprimer
    - Résoudre des dépendances



Bundles OSGi

- Livrés dynamiquement
- Partageables par différentes applications
- Des unités de déploiement (archive JAR)
- Des unités fonctionnelles (offrent des services)
- Reliés fonctionnellement par les services (offerts et requis)
- Explicitant leurs dépendances de classes Java (import/export)

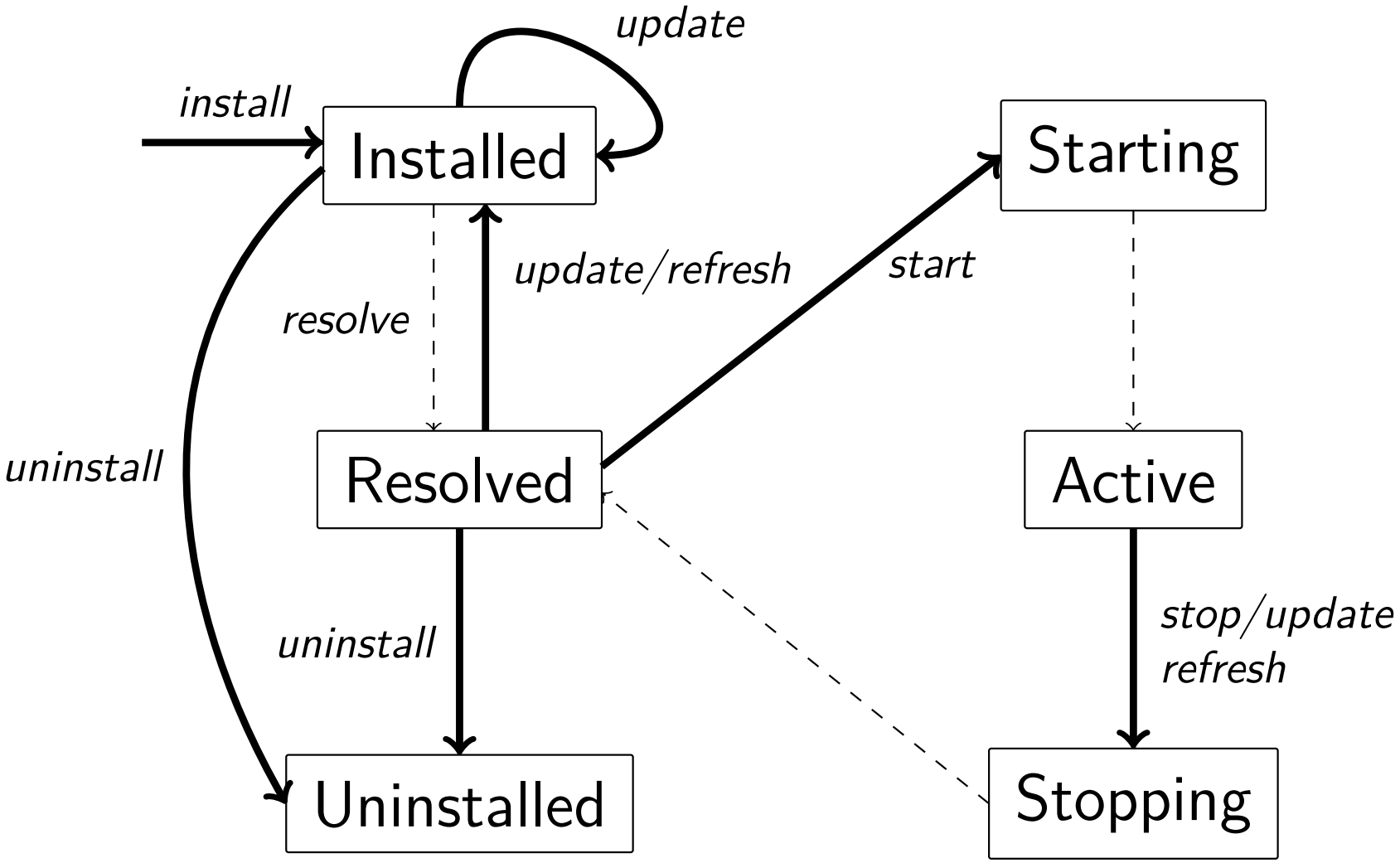
Structure d'un bundle



Manifest d'un bundle (MANIFEST.MF)

- Nom et description du bundle
- Les packages importés
- Les packages exportés
- L'activateur (gère une partie du cycle de vie du bundle)
- Un *classpath* interne
- Le numéro de version
- Le vendeur, un mail de contact, un copyright ...

Cycle de vie d'un bundle



Actions possibles sur un bundle

- *install* : charge dans le système de fichier
- *resolve* : charge dans la VM
- *start* : appelle `start()` de l'activateur
- *stop* : appelle `stop()` de l'activateur
- *uninstall* : supprime du système de fichier
- *update* : ré-installe
- *refresh* : reconstruit les dépendances

Les états d'un bundle

- *installed* : correctement (télé)chargé
- *resolved* : toutes les classes nécessaires au bundle sont disponibles (résolution de dépendances)
- *starting* : démarrage du bundle. `start()` de l'activateur s'exécute
- *active* : le bundle peut rendre des services
- *stopping* : arrêt du bundle. `stop()` de l'activateur s'exécute
- *uninstalled* : le bundle ne peut plus changer d'état

Activateur d'un bundle

- Démarre et arrête bundle (méthodes `start()` et `stop()`)
- Démarrage
  - Instancie et enregistre les services fournis
  - Interroge la plate-forme OSGi pour trouver les services requis et se lier à eux
  - Démarre des *threads*
- Arrêt
  - Dés-enregistre les services fournis
  - Relâche les références vers les services requis
  - Arrêt des *threads*

OSGi et SOA

- Couplage lâche
  - Annuaire centralisé ; référence d'interface Java + propriétés
- Contrat de service
  - Signature des interfaces fonctionnelles + propriétés
  - Protocole de transport et d'échanges (JVM/TCP-RMI) et règles d'invocation supposées connues
- Mécanisme de notification aux *bundles* des arrivées et départs des *bundles*

Bilan d'OSGi

- Intérêts :
  - Bonne gestion du déploiement
  - Très léger
  - Très dynamique
- Limites :
  - Pas vraiment de notion d'architecture
  - Implémentations souvent opaques

Les slides sont inspirés et contiennent des documents de J. Aldrich, A. Beugnard, E. Bruneton, H. Cervantes, D. Donsez, L. Duchien, S. Krakowiak, Y. Mahéo, L. Seinturier, T. Teschke et J. Ritter, B. Traverson, ...