

# Architecture des applications

---

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Les classes valeurs</b>	<b>2</b>
2.1	Les JavaBeans . . . . .	2
2.2	Les différents types de JavaBeans . . . . .	3
2.3	Où sont les traitements? . . . . .	3
<b>3</b>	<b>Les classes de service</b>	<b>5</b>
3.1	Les services d'accès aux données . . . . .	5
3.2	Les contrôleurs . . . . .	6
3.3	Les vues . . . . .	6
3.4	Les services métier . . . . .	6
3.5	Les types de services . . . . .	7
<b>4</b>	<b>Spécification d'un service</b>	<b>7</b>
4.1	Gestion des erreurs . . . . .	7
4.2	Les données manipulées . . . . .	8
4.3	Un service pour les données . . . . .	8
<b>5</b>	<b>L'implantation d'un service</b>	<b>9</b>
5.1	Factoriser l'implantation . . . . .	9
5.2	Le code d'implantation . . . . .	9
5.3	Un JavaBean pour l'implantation . . . . .	10
5.4	Les singletons . . . . .	11
5.5	Tests unitaires . . . . .	11
<b>6</b>	<b>Injection de dépendances</b>	<b>12</b>
6.1	Composant logiciel . . . . .	13
6.2	Inversion de contrôle . . . . .	13
6.3	Paralléliser le développement . . . . .	14

## 1 Introduction

Une **application** c'est

- des données (**le modèle**),
- des traitements (**couches logicielles** ou **services**),
- un intégrateur (**point de démarrage**) ou un **environnement d'exécution** (**serveur d'applications**).

## 2 Les classes valeurs

**Objectif** : représenter les données manipulées par l'application sans référence aux traitements.

La définition des classes valeurs peut être obtenue par une phase de conception UML (**Diagramme de classes**) ou par une méthode d'analyse des données (**Modèle conceptuel des données**).

L'écriture de ces classes peut être prise en charge par des outils automatiques (à partir de schémas UML, relationnels ou XML).

### 2.1 Les JavaBeans

En Java, les classes valeurs sont représentées par des **JavaBeans**. Un *JavaBean* est une classe qui respecte les contraintes suivantes :

- Chaque classe décrit **une entité** (les voitures, les commandes, les personnes, etc.). C'est la notion de table en relationnel.
- Chaque instance décrit **une entité particulière** (une voiture, une commande, une personne, etc.). C'est une ligne d'une table.
- Une entité particulière est entièrement décrite par les propriétés de la classe (poids, nom, volume, couleur, propriétaire, etc.)
- deux méthodes publiques et optionnelles sont associées à chaque propriété : l'une pour l'accès (appelée **getter**) et l'autre pour la modification (appelée **setter**).
- Il existe un constructeur public sans argument.

Plus d'information sur les javaBeans<sup>1</sup>.

Un exemple de JavaBean :

```
package fr.sample.beans;

import java.util.Set;

public class Person {
    // properties
    private String name;
    private boolean student;
    private Set<Person> friends;

    // constructor
    public Person() { }

    // getters
    public String getName() { return name; }
    public boolean isStudent() { return student; }
    public Set<Person> getFriends() { return friends; }

    // setters
    public void setName(String name) { this.name = name; }
    public void setStudent(boolean student) { this.student = student; }
    public void setFriends(Set<Person> friends) { this.friends = friends; }
}
```

<sup>1</sup><http://www.dil.univ-mrs.fr/~garreta/>

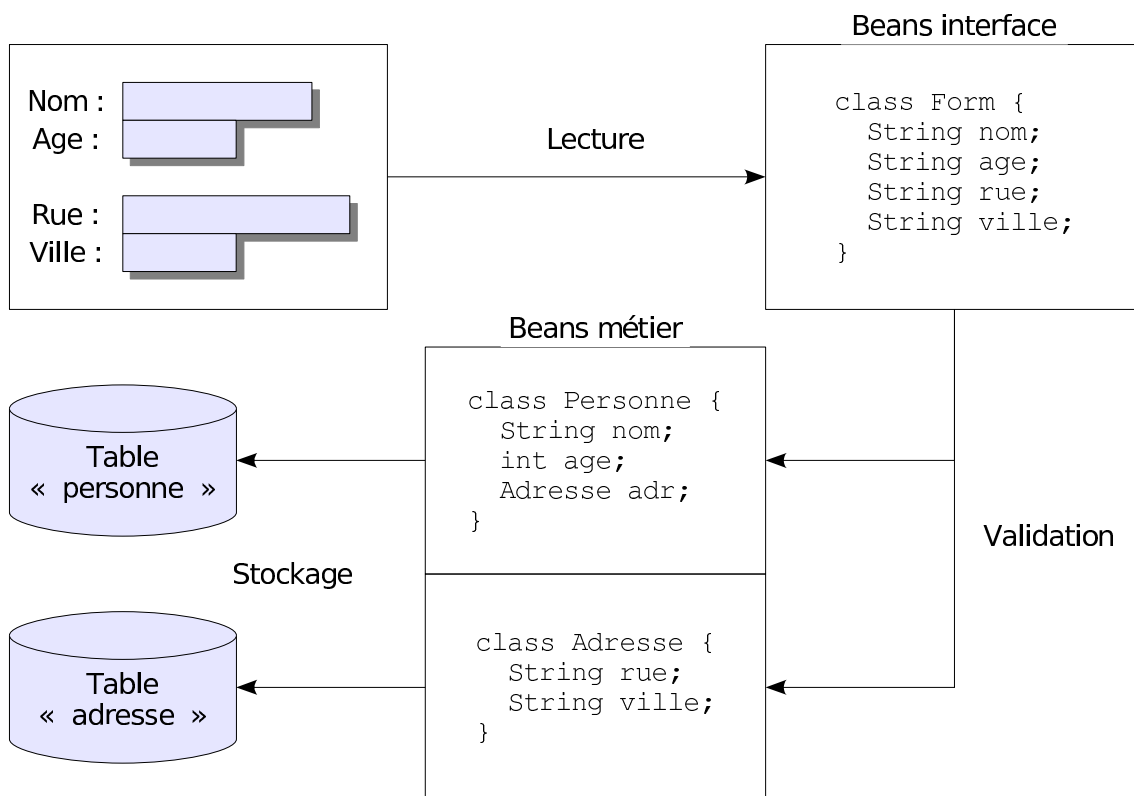
Vous pouvez noter la construction normalisée des noms de méthodes et l'utilisation du changement de casse pour construire les identificateurs. Il faut également noter la forme particulière des *getters* quand la propriété est un booléen.

Dernières remarques :

- Pour les collections, les interfaces sont à privilégier.
- L'accès aux propriétés passe obligatoirement par les méthodes (l'implantation est donc forcément privée).
- Ce sont les méthodes qui fixent le nom des propriétés (l'anglais est à favoriser).
- La version 1.7 de Java va introduire une notation plus compacte (nouveau mot-clé `property` et construction automatique des *getters* et *setters*).
- Les environnements de développement offrent toujours des facilités pour rédiger les JavaBeans.

## 2.2 Les différents types de JavaBeans

Suivant le contexte, nous pouvons avoir plusieurs JavaBeans pour représenter une donnée :



## 2.3 Où sont les traitements ?

Un exemple : comment implanter une méthode de sauvegarde ?

```
public class Person {
    ...
    public void save() { ... }
    ...
}
```

Pour implanter cette méthode nous devons avoir des **informations** sur la **nature** (BDR, XML, etc.) et les **paramètres** (login, mot de passe, etc.) du système de persistance.

Ces informations **ne peuvent pas** être représentées par une propriété du bean Person :

- ce ne sont pas des données de l'application,
- ils seraient dupliqués dans chaque instance,

Nous devons donc ajouter un paramètre à cette méthode :

```
public class Person {
    ...

    public void save(JDBCParameters where) { ... }

    ...
}
```

Le paramètre where est **obligatoire** car il indique où effectuer la sauvegarde (classe JDBCParameters). Cette approche pose d'autres problèmes :

- La définition du bean est **polluée** par des considérations techniques. Nous nous éloignons de l'objectif des classes valeurs (représentation des données).
- La méthode de persistance est **dupliquée** dans chaque bean (très difficile à changer).
- Il est **délicat** d'offrir plusieurs méthodes de sauvegarde. Nous avons créé une **dépendance** artificielle entre une donnée et sa manipulation.

**Nouveaux objectifs :**

- supprimer les **dépendances** entre données et traitements,
- rassembler les traitements **éparpillés**,

**Solution :** il faut ranger le **code technique** de sauvegarde dans une classe **spécialisée** qui va se charger de la sauvegarde de **tous** les beans :

```
public class JDBCStorage {
    ...

    public void save(Person p) {
        ...
    }

    ...
}
```

Il peut exister plusieurs versions de cette classe (JDBCStorage, FileStorage, XmlStorage) qui rendent le même service mais de manière différente.

```
public class JDBCStorage implements Storage {
    ...
}

public class FileStorage implements Storage {
    ...
}
```

Ces implantations **partagent** la même interface ou peuvent hériter de la même **classe abstraite**. Le partage d'interfaces est une solution préférable.

Nous venons de définir les **classes de service**.

### 3 Les classes de service

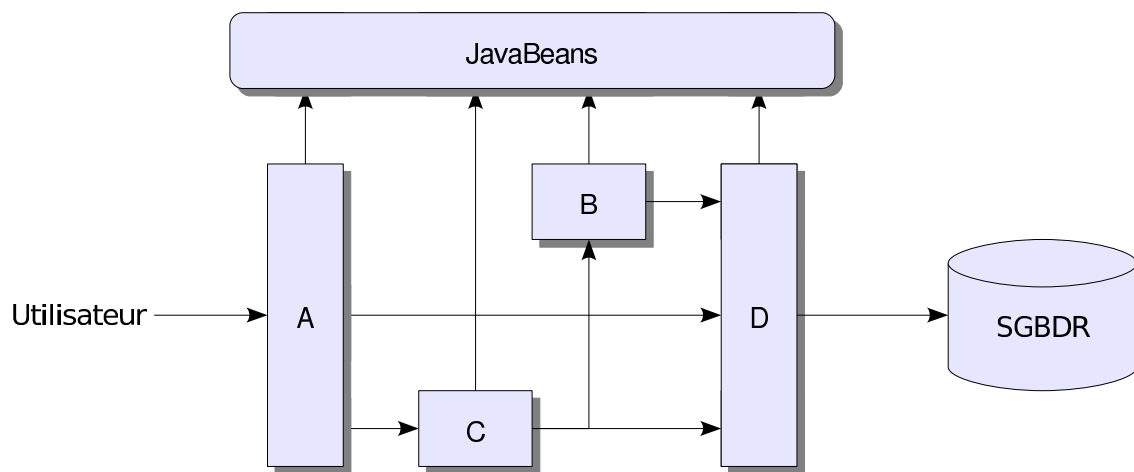
Un **service logiciel** c'est

- une spécification (en Java elle se traduit par une ou plusieurs interfaces),
- une ou plusieurs implantations (réalisées par une ou plusieurs classes de service qui agissent sur les données).

Les utilisateurs d'un service travaillent à partir de la **spécification** (interface) et ignorent les **détails** de l'implantation sous-jacente.

Le rôle de la **couche d'intégration** est de faire le lien entre les utilisateurs d'une spécification et une implantation particulière.

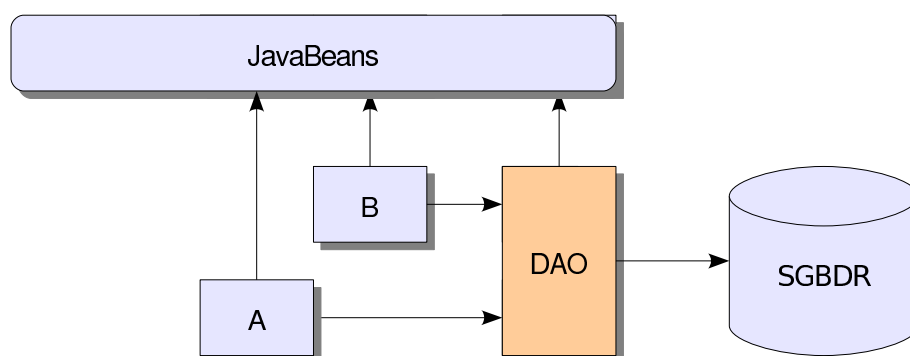
Une application doit être conçue comme un ensemble de services construits les uns à partir des autres en vue de répondre aux spécifications détaillées.



Les services sont développés **indépendamment** et la couche d'intégration va faire le lien entre A/C, A/D, B/D, C/B et C/D.

On peut **classer** les services en fonction de leur rôle.

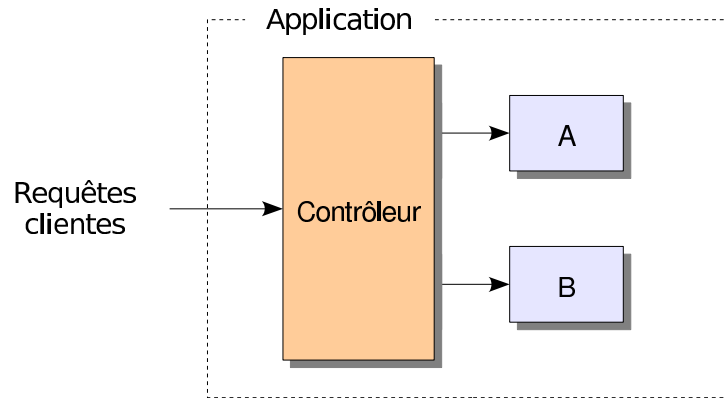
#### 3.1 Les services d'accès aux données



Une couche DAO (*Data Access Object*) offrent plusieurs services :

- centralisation des accès aux données,
- simplification de l'accès aux données,
- abstraction du support de stockage,
- travail sur les entités principales,

### 3.2 Les contrôleurs

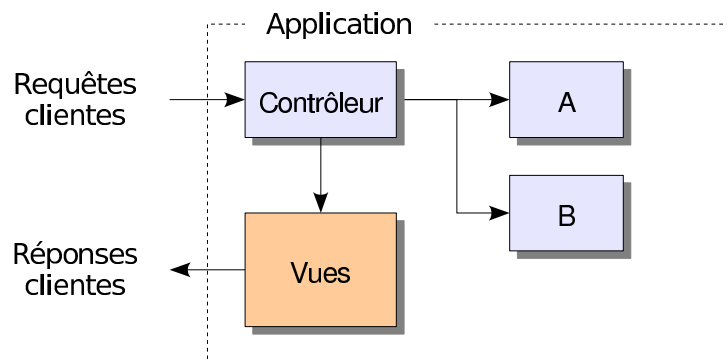


Un contrôleur assure :

- l'implantation du protocole d'entrée,
- le traitement et la validation des requêtes clientes,
- l'appel aux couches internes.

Un contrôleur est un **point d'entrée** d'une couche logicielle.

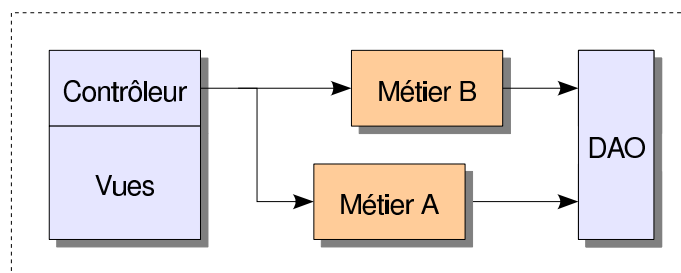
### 3.3 Les vues



Les vues assurent :

- l'implantation du protocole de sortie,
- la construction des résultats à partir des données,
- l'envoi de ces résultats.

### 3.4 Les services métier



Les services « métier » assurent les opérations de traitement des données.

**Caractéristiques :**

- Ils sont **indépendants** d'une source de données.
- Ils sont **indépendants** de la logique applicative (suite de requêtes clientes).
- Ils sont **réutilisables**.

### 3.5 Les types de services

Les services peuvent :

- offrir des fonctions spécialisées (DAO, contrôleur, etc.),
- simplifier un service trop complexe (**facade**),
- enrichir les fonctions d'un service existant (**decorator**),
- rechercher un service (**locator**),
- se charger de l'accès à un service (**proxy**),
- construire un service particulier (**factory**).

## 4 Spécification d'un service

Une spécification décrit « **ce qui est fait** » sans préciser « **comment le faire** ». En Java, elle est exprimée par une ou plusieurs interfaces :

```
package fr.sample.services.mailer;

public interface IMailer {

    void sendMail(String subject, String body,
                  String from, String to, String cc);

}
```

Les interfaces d'un service sont rangées dans un **paquetage particulier**. Ce n'est peut-être pas la même personne qui va développer l'interface et son implantation.

### 4.1 Gestion des erreurs

Les erreurs émises ne doivent pas **dévoiler** les choix d'implantation.

```
package fr.sample.services.mailer;

public interface IMailer {

    void sendMail(String subject, String body,
                  String from, String to, String cc)
        throws MailerException;

}
```

Pour éviter cette fuite d'information, le paquetage doit regrouper l'interface et la définition des exceptions du service :

```
package fr.sample.services.mailer :
| IMailer.class
| MailerException.class
```

## 4.2 Les données manipulées

Une spécification peut **utiliser**, voir **définir** les classes valeurs dont elle a besoin :

```
package fr.sample.services.mailer;

public interface IMailer {

    void sendMail(Mail mail) throws MailerException;

}
```

Le paquetage regroupe maintenant les interfaces, les javaBeans et les exceptions. C'est le cas le plus général.

```
package fr.sample.services.mailer :
| IMailer.class
| MailerException.class
| Mail.class
```

## 4.3 Un service pour les données

La définition des données peut être vue comme un service. Le paquetage contient les JavaBeans (classes) sans interface ni exception.

On peut également définir les données par des interfaces :

```
package fr.sample.service.values;

import java.util.Collection;

public interface IPerson {

    // getters
    String getName();
    boolean isStudent();
    Collection<IPerson> getFriends();

    // setters
    void setName(String name);
    void setStudent(boolean student);
    void setFriends(Collection<IPerson> friends);

}
```

Dans ce cas, il faut également fournir un service de création des instances des JavaBeans qui respectent l'interface :

```
package fr.sample.service.values;

public interface IValuesFactory {

    IPerson newPerson();

}
```

Nous obtenons une **indépendance complète** entre la définition des données, leur implantation et leur utilisation.

## 5 L'implantation d'un service

Les classes d'implantation doivent :

- respecter la lettre et l'esprit de la spécification,
- regrouper les ressources dans un paquetage d'implantation,
- offrir un moyen souple pour paramétrer leur fonctionnement,
- interagir avec son environnement.

**Attention** : il est assez difficile d'avoir plusieurs implantations interchangeables.

Cet **objectif** est néanmoins important car il permet de découpler client et fournisseur de service et donc

- de diminuer la complexité globale,
- d'améliorer la réutilisabilité du code d'implantation.

### 5.1 Factoriser l'implantation

Si nous offrons plusieurs implantations d'un même service, il est probable que certaines méthodes puissent **partager leur définition**.

Pour ce faire, nous devons définir une classe d'implantation abstraite qui va regrouper ce code commun :

```
package fr.sample.imp.mailer;

import fr.sample.services.mailer.IMailer;
import fr.sample.services.mailer.Mail;
import fr.sample.services.mailer.MailerException;

public abstract class AbstractMailer implements IMailer {

    protected void checkMail(Mail m) throws MailerException {
        // check addresses, body and destination
    }

    ... autres méthodes implantées

}
```

### 5.2 Le code d'implantation

Une première solution consiste à créer une classe d'implantation dont le constructeur permet de récupérer les paramètres de fonctionnement :

```
package fr.sample.imp.mailer;

import fr.sample.services.mailer.*;

public class SmtplibMailer extends AbstractMailer implements IMailer {
    // SMTP server name
    final private String host;

    public SmtplibMailer(String host) {
        super();
        if (host == null) throw new NullPointerException();
        this.host = host;
    }

    public void sendMail(Mail mailToSend) throws MailerException {
        checkMail(mailToSend);
        // send mail to the SMTP server
        ...
    }
}
```

son utilisation est simple :

```
IMailer mailer = new SmtplibMailer("mail.dil.univ-mrs.fr");
mailer.sendMail(mail);
```

Mais

- les paramètres ne peuvent pas être changés,
- ce service n'est pas recyclable,
- il est difficile de prévoir des valeurs par défaut,
- si le nombre de paramètres est important, le constructeur est délicat à appeler.

### 5.3 Un JavaBean pour l'implantation

Nous allons utiliser les **getters** et les **setters** pour gérer les paramètres et nous introduisons deux nouvelles méthodes d'initialisation et de clôture.

```

package fr.sample.imp.mailer;
import fr.sample.services.mailer.*;

public class JavaBeanSmtpMailer extends AbstractMailer implements IMailer {
    // SMTP server name
    private String host = "localhost";

    // getter and setter for parameters
    public String getHost() { return host; }
    public void setHost(String host) { this.host = host; }

    // initialize service and ressources
    public void init() {
        if (host == null) throw new IllegalStateException("no SMTP host");
        ...
    }

    // close service and ressources
    public void close() { ... }

    public void sendMail(Mail mailToSend) throws MailerException { ... }
}

```

son utilisation est un peu moins simple :

```

JavaBeanSmtpMailer mailer = new JavaBeanSmtpMailer();
mailer.setHost("mail.dil.univ-mrs.fr");
mailer.init();
mailer.sendMail(mail);
mailer.close();

```

Mais

- les paramètres peuvent être changés (il faut ensuite appeler `init`),
- le service est totalement recyclable (plusieurs phases de paramétrage, `init` et `close`),
- les valeurs par défaut sont possibles (par exemple un paramètre `debug` à `false`).
- la partie initialisation qui est un peu moins simple ne doit être réalisée qu'une seule fois. C'est le rôle du **code d'intégration**.

## 5.4 Les singletons

De nombreux services sont souvent implémentés par des **singletons** c'est-à-dire des classes à instance unique.

Dans un environnement **Multi-Threads**, il est primordial de prévoir des sections critiques qui protègent les ressources critiques (ressources extérieures ou propriétés de la classe d'implantation).

Une autre solution consiste à créer une **instance par client**. Dans cette optique, il est nécessaire de mettre en place un service de création des instances (**factory**) qui lui, est un singleton.

Si le service ne gère pas le **multi-threading**, c'est le client qui doit le prévoir. **Attention** : les classes façades peuvent masquer le service réel et donc rendre inefficace les clauses `synchronized`.

## 5.5 Tests unitaires

Une fois les services spécifiés, nous pouvons prévoir une **campagne de tests**.

Cette étape de **tests unitaires** permet de vérifier

- qu'une implantation répond bien aux spécifications,
- que deux implantations sont bien équivalentes,
- la non-régression en cas de modification.

La recherche d'erreur est facilitée par le fait que les couches sont clairement isolées.

**Conseil 1** : les jeux de tests doivent être rédigés avant l'implantation du service.

**Conseil 2** : Testez un service à chaque étape de son développement.

Dans un deuxième temps, ces tests unitaires doivent être complétés par des **tests d'intégration**.

## 6 Injection de dépendances

Ce principe traite le délicat problème de la **communication** et de la **dépendance** entre service logiciel.

Imaginons que nous ayons un service d'envoi de mail par un utilisateur authentifié. L'implantation de ce service nécessite le service **Mailer**.

```
public class AuthMailer implements IAuthMailer {
    // a mailer
    JavaBeanSmtpMailer mailer = new JavaBeanSmtpMailer();

    // init service
    public void init() {
        mailer.setHost("mail.dil.univ-mrs.fr");
        mailer.init();
    }

    ...
}
```

Dans cette version nous introduisons un **dépendance involontaire** avec une implantation particulière de IAuthMailer et le nom du serveur de mail n'est pas **correctement paramétré**.

Pour régler ce problème, nous allons

- remplacer l'utilisation de JavaBeanSmtpMailer par l'interface IMailer,
- traiter le *mailer* comme un paramètre,
- préparer le *mailer* à l'extérieur.

Le code devient

```
public class AuthMailer implements IAuthMailer {
    // mailer parameter
    IMailer mailer;

    // init service
    public void init() {
        if (mailer == null) throw new IllegalStateException("no mailer");
    }

    // setter and getter
    public IMailer getMailer() { ... }
    public void setMailer(IMailer mailer) { ... }

    ...
}
```

Le code d'intégration (programme principal) devient :

```

// a SMTP mailer
JavaBeanSmtpMailer mailer = new JavaBeanSmtpMailer();
mailer.setHost("mail.dil.univ-mrs.fr");
mailer.init();
// mailer for users
AuthMailer am = new AuthMailer();
am.setMailer(mailer);
am.init();
... use am
am.close();
mailer.close();

```

Nous pouvons très facilement et sans **modifier la couche** AuthMailer changer la stratégie d'envoi des mails. Il suffit de changer les quatre premières lignes du code d'intégration.

C'est la partie intégration qui se charge d'injecter dans le service **utilisateur** la référence vers le service **utilisé**. Initialiser une application revient à **créer les services logiciels, injecter les dépendances** et appeler les **méthodes d'initialisation** (callback).

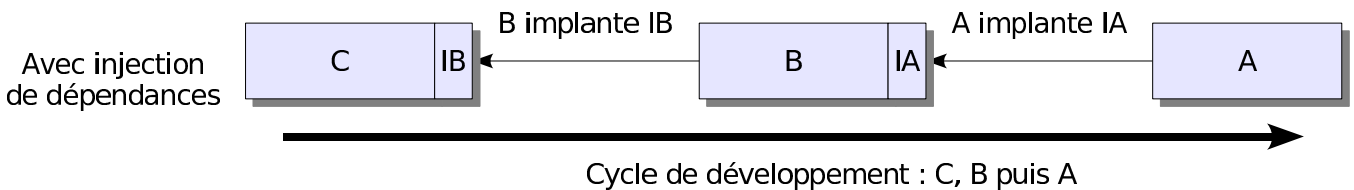
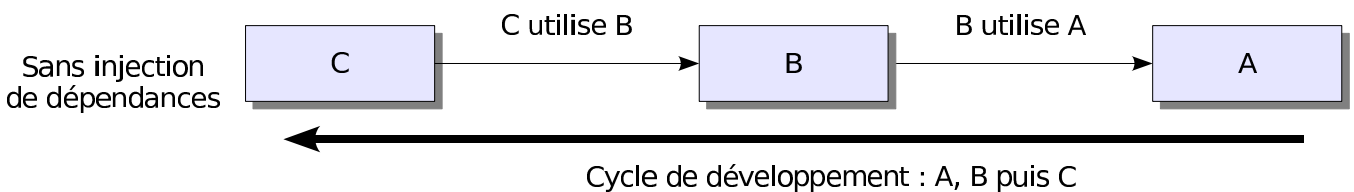
### 6.1 Composant logiciel

Nous venons de définir la notion de **composant logiciel** :

- C'est une brique de base réutilisable.
- Les interfaces d'entrée et de sortie sont clairement définies.
- Ce composant est déployé dans un environnement (via le code d'intégration, EJB, Spring, RMI, Servlet, Test JUnit).
- La création, la vie et l'arrêt de ce composant correspondent à des évènements générés par l'environnement et envoyés au composant via des *callback* (utilisation des annotations Java).

### 6.2 Inversion de contrôle

L'injection des dépendances peut aussi être vu comme une inversion de contrôle :



Dans la deuxième approche nous décomposons un problème complexe (C) en problèmes plus simples (B puis A).

### 6.3 Paralléliser le développement

Nous pouvons même paralléliser le développement des couches en prévoyant une implantation vide de chaque spécification (**bouchon**) et en suivant les étapes ci-dessous :

- réaliser une implantation (par exemple de B),
- lui fournir le bouchon A par injection de dépendances,
- valider l'implantation de B.

Nous devons ensuite réaliser un **test d'intégration**.

**Note** : Cette démarche correspond bien aux méthodes agiles qui sont dirigées par les tests et qui préconisent une première implantation **simple** et **juste** puis une série **d'améliorations**.