Architecture des applications

Table des matières

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 (JavaBeans)

Objectif : représenter les données manipulées par l'application sans référence aux traitements (**Modèle des données**). Chaque classe décrit **une entité** :

```java
package fr.myapp.model;

import java.util.Set;
import java.io.Serializable;

// La classe doit être sérialisable
public class Person implements Serializable {

   private static final long serialVersionUID = 1L;

   // les propriétés sont privées (interfaces à privilégier)
   private String name;
   private boolean student;
   private Set<Person> friends;

   // il existe un constructeur public sans argument
   public Person() { }

   ...
```

```java
   ...

   // 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; }

}
```

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**).

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.

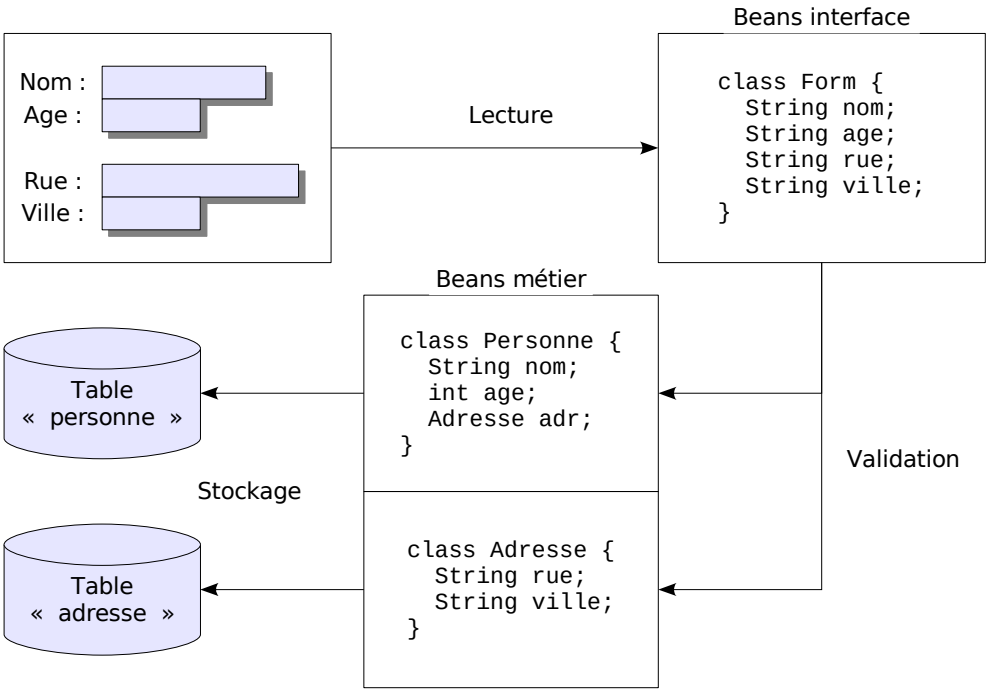
La librairie **Lombok** permet de simplifier l'écriture des JavaBeans :

```java
package fr.myapp.model;

import java.io.Serializable;
import java.util.Set;

import lombok.Data;
import lombok.ToString;

@Data
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private boolean student;
    @ToString.Exclude
    private Set<Person> friends;
}
```

Le constructeur sans argument ainsi que les méthodes `get...`, `set...`, `toString`, `equals` sont générées automatiquement à la compilation.

2.1 Les différents types de JavaBeans

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



3 Les classes de services

Un exemple : comment proposer une méthode de sauvegarde ?

```
@Data
public class Person {
   private String name;
   private int age;

   public void save() { ... }      // Traitement
}
```

Critiques :

- La méthode de persistance est **dupliquée** dans chaque bean. Nous ne respectons pas la règle : **une responsabilité, une classe** ni la règle **une classe, une responsabilité**.
- Il est **délicat** d'offrir plusieurs méthodes de sauvegarde. Nous avons créé une **dépendance** artificielle entre une donnée et sa manipulation.
- Il est **difficile** de paramétrer le service de sauvegarde.
- Ou placer les traitements qui s'appliquent sur plusieurs JavaBeans ?

3.1 Rassembler les traitements

Nouveaux objectifs :

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

Solution : il faut ranger le **code 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) {
      ...
   }

   ...
}
```

3.2 Plusieurs versions

Il peut exister plusieurs versions (`JDBCStorage` , `FileStorage` , `XmlStorage`) qui rendent le même service (**défini par une interface**) mais de manière différente.

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

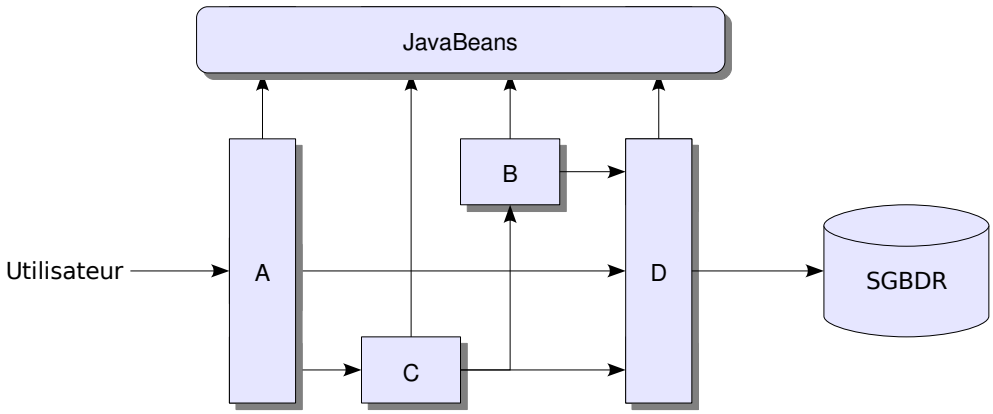
```
public class FileStorage
   implements IStorage {
   ...
}
```

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

- une spécification (une ou plusieurs interfaces),
- une ou plusieurs implémentations.
- Les utilisateurs d'un service travaillent à partir de la **spécification** (interface) et **ignorent les détails** de l'implémentation sous-jacente.
- La **couche d'intégration** fait le lien entre les utilisateurs d'une spécification et une implémentation particulière.

3.3 Architecture d'une application

Une application doit être concue 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.

- `D` : **Data Access Object** (abstraction/simplification du stockage),
- `A` : **Contrôleur** (implémentation du protocole d'entrée),
- `B,C` : **Services métier** (opérations sur les données métier),

3.4 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 :

```
public interface IMailer {

   void sendMail(Mail mail) throws MailerException;

}
```

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

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

3.5 L'implémentation d'un service

Les classes d'implémentation doivent :

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

Attention : il est difficile d'avoir plusieurs implémentations 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 **ré-utilisabilité** du code d'implémentation.

3.6 Un JavaBean pour l'implémentation

Des propriétés pour les paramètres

```
@Data // utilisation de Lombok
public class SmtpMailer implements IMailer {

   // SMTP server name
   private String host = "localhost";

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

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

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

Utilisation de SmtpMailer

```
// Création et paramétrage
SmtpMailer mailer = new SmtpMailer();
mailer.setHost("smtp.univ-amu.fr");
// Initialisation
mailer.init();

// Utilisation
Mail mail = prepareMail();
mailer.sendMail(mail);

// Fermeture
mailer.close();
```

- les paramètres peuvent être changés (il faut ensuite appeler `init`),
- le service est recyclable (plusieurs phases de paramétrage, `init` et `close`),
- les valeurs par défaut sont possibles (par exemple `localhost`),
- la partie initialisation n'est réalisée qu'une seule fois (**code d'intégration**).

4 Injection de dépendances

Ce principe traite le problème de la **dépendance** entre service logiciel. Imaginons que nous ayons un service d'envoi de mail au format HTML.

```
public interface IHtmlMailer {
    void sendMail(Mail mail) throws MailerException;
}
```

L'implémentation nécessite le service Mailer traité comme un paramètre :

```java
@Data // utilisation de Lombok
public class HtmlMailer implements IHtmlMailer {

   IMailer mailer;     // mailer parameter

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

   ...
}
```

Code d'intégration des services :

```java
// a SMTP mailer (initialisation)
SmtpMailer mailer = new SmtpMailer();
mailer.setHost("smtp.univ-amu.fr");
mailer.init();

// a HTML mailer
HtmlMailer htmlMailer = new HtmlMailer();
htmlMailer.setMailer(mailer);
htmlMailer.init();

// use htmlMailer
Mail mail = prepareMail();
htmlMailer.sendMail(mail);

// closing
htmlMailer.close();
mailer.close();
```

L'intégration **injecte** dans le service **utilisateur** la référence vers le service **utilisé**.

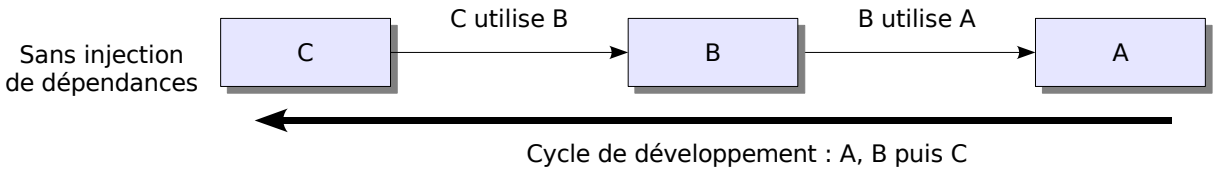
Changer l'envoi des mails ne nécessite pas de **modifier le service** `HtmlMailer`.

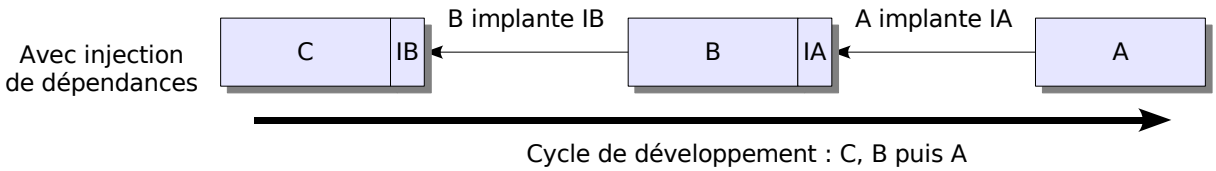
Initialiser une application revient à

- **créer les services logiciels**,
- **injecter les paramètres** et
- **injecter les dépendances** et
- appeler les **méthodes d'initialisation** (callback).

4.1 Inversion de contrôle

L'injection des dépendances comme une inversion de contrôle :





Nous pouvons développer en parallèle `A`, `B` et `C` si nous définissons des bouchons pour `NullA` et `NullB`.

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

4.2 Retour sur le projet de génie logiciel

À la place de

```
          use                 use
Game ----------> Kernel ----------> Engine
```

Nous pouvons avoir

```
  Game ------ use -------> IKernel

Kernel ------ use -------> IEngine
       --- implements ---> IKernel

Engine --- implements ---> IEngine
```

Les services `Game`, `Kernel` et `Engine` deviennent indépendants.

4.3 Injection de dépendances avec Spring

Déclaration d'un service (avec initialisation et fermeture) :

```java
@Service // Un service géré par Spring
@Data   // Utilisation de Lombok
public class SpringSmtpMailer implements IMailer {

   // SMTP server name
   private String host = "localhost";

   // initialize service and ressources
   @PostConstruct
   public void init() { ... }

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

   ...
}
```

Déclaration d'un service (avec injection de dépendance) :

```
@Service
@Data
public class SpringHtmlMailer implements IHtmlMailer {

   // mailer parameter
   @Autowired
   IMailer mailer;

   ...
}
```

Utilisation d'un service (avec injection de dépendance) :

```
@SpringBootTest
public class TestMailer {

    @Autowired
    ApplicationContext context;

    @Autowired
    IHtmlMailer mailer;

    @Test
    public void testHtmlMailer() {
        assertTrue(mailer instanceof SpringHtmlMailer);
    }

    @Test
    public void testMailerByContext() {
        var mailer2 = context.getBean(IHtmlMailer.class);
        assertEquals(mailer, mailer2);
    }

}
```