Spring 3.x

Introduction

jnesis

plan

- Présentation
- Configuration de l'architecture
- Solutions
- Conteneur léger et inversion de contrôle (IOC)
- Mise en œuvre
- Spring et l'injection de dépendance
- Conclusion

Présentation

- Dans cette formation nous étudierons la version 3 de Spring, sur la base de Java 6.
- Spring 3 date de 2010 et nombre d'applications fonctionnent encore avec Spring 2.X, notamment la version 2.5.
  - Les différences entre la version majeure précédente (Spring 2.5) et la version 3 sont assez légères mais seront évoquées dans le support lorsque cela s'impose.

Configuration de l'architecture

Prise en compte des évolutions fonctionnelles

- La décomposition du code en composants vise plusieurs objectifs
  - Favoriser la lisibilité du code, la documentation
  - Simplifier la testabilité (on y reviendra)
  - Mais surtout simplifier les maintenances futures et notamment :
    - Les évolutions fonctionnelles

Configuration de l'architecture

- Reprenons le service de facturation.
  - Certains clients de votre application vous demandent de produire des numéros de factures avec un pas de 10.
  - D'autres clients vous demandent de produire des numéros de factures au format alphanumérique FACT_Numéro.
  - D'autres encore vous demandent de pouvoir récupérer ce numéro d'un système externe (ERP par exemple)
- Plusieurs solutions s'offrent à vous, l'une d'entre-elle va nous intéresser tout particulièrement pour la suite :
  - Créer différentes implémentations de services

Configuration de l'architecture

Créer de nouvelles implémentations de services

- On ajoutera autant de classes de Services qu'il existe de spécificités

```
public class FactureService1 {

   private static int counter=1;

   public Facture createFacture(Facture facture){

      facture.setNumero(counter++);

      return facture;
   }
...
```

```
public class FactureService2 {

   private static int counter=1;

   public Facture createFacture(Facture facture){

      facture.setNumero(counter);
      counter=counter+10;

      return facture;
   }
...
```

```
public class FactureService3 {

   private static int counter=1;

   public Facture createFacture(Facture facture){

      facture.setNumero(...);
      //D'un ERP par exemple

      return facture;
   }
...
```

Configuration de l'architecture

Pour permettre l'utilisation de plusieurs implémentations en fonction du contexte, on pourrait exploiter un paramètre de configuration (d'où provient ce paramètre a peu d'importance ici) :

```
package com.mycompany.controller;

import java.util.Scanner;

import com.mycompany.model.Facture;
import com.mycompany.service.FactureService1;
import com.mycompany.service.FactureService2;
import com.mycompany.service.FactureService3;

public class FactureController {


	public void createFactureFromConsoleInput(){
		Scanner sc=new Scanner(System.in);
		System.out.println("Quel est le nom du client ?");
		String nomClient=sc.next();
		Facture fact=new Facture();
		fact.setNomClient(nomClient);
		if (param==1){
			FactureService1 fs=new FactureService1();
			fact=fs.createFacture(fact);
		}
		else if (param==2){
			FactureService2 fs=new FactureService2();
			fact=fs.createFacture(fact);
		}
		else{
			FactureService3 fs=new FactureService3();
			fact=fs.createFacture(fact);
		}
		System.out.println("La facture "+fact.getNumero()+" à été créée");
	}

}
```

Configuration de l'architecture

- Cette solution engendre des contraintes significatives, chaque nouvelle spécificité client demande
  - la modification du code existant
  - la recompilation du code
  - et le plus gênant, une montée de version ce qui implique généralement un redéploiement chez l'ensemble des clients
- Alors quelle est la solution ?
  - C'est ici que l'on introduit la notion de **configuration** de l'architecture applicative.
  - Il va s'agir d'améliorer la solution précédente et d'**instrumenter** le code, en indiquant quelle implémentation doit être utilisée par simple déclaration dans un fichier de configuration.
    - Pour chaque client, seule l'implémentation qui lui est nécessaire sera fournie
    - Il ne sera plus nécessaire de modifier le code existant à chaque nouvelle implémentation

Solutions

Programmation par contrat

- Il va tout d'abord être nécessaire d'adapter le code existant de manière à créer une abstraction (Interface ou classe Abstraite) des implémentations existantes, c'est la *programmation dite par contrat*.

```
package com.mycompany.service;

import com.mycompany.model.Avoir;
import com.mycompany.model.Facture;

public interface FactureServiceInterface {

   public Facture createFacture(Facture facture);

   public Avoir createAvoirOnFacture(Facture facture);
}
```

```
public class FactureService1 implements FactureServiceInterface{

      private static int counter = 1;

      @Override
      public Facture createFacture(Facture facture) {
            facture.setNumero(counter++);
            return facture;
      }
      @Override
      public Avoir createAvoirOnFacture(Facture facture) {
            Avoir newAvoir = ...;
            return newAvoir;
      }
...
```

Solutions

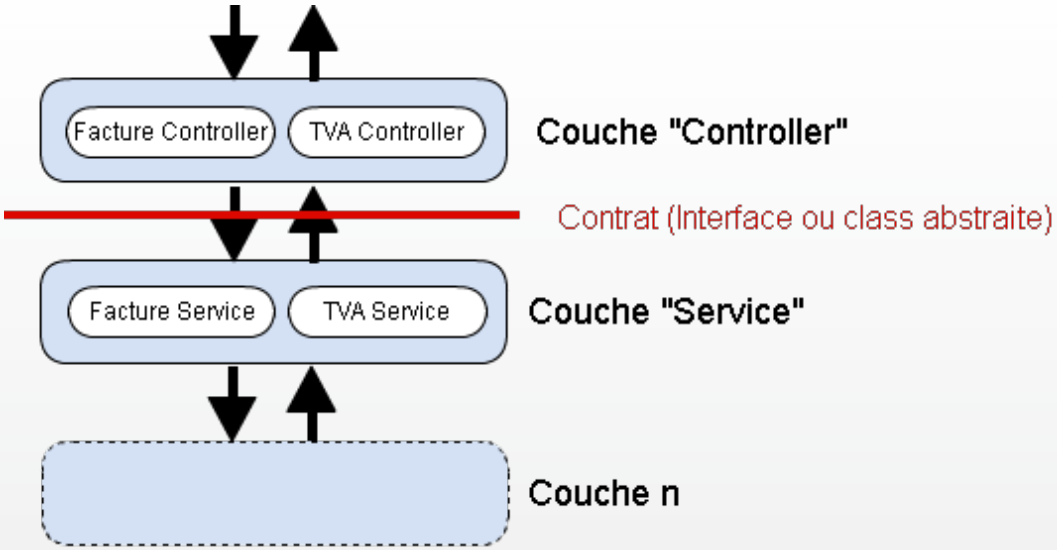
- Le composant qui exploite le service (Controller) ne fera désormais plus référence explicite à une implémentation

```
public class FactureController {

	private FactureServiceInterface fs;

	public void createFactureFromConsoleInput(){
		Scanner sc=new Scanner(System.in);
		System.out.println("Quel est le nom du client ?");
		String nomClient=sc.next();
		Facture fact=new Facture();
		fact.setNomClient(nomClient);
		fact=fs.createFacture(fact);
		System.out.println("La facture "+fact.getNumero()+" à été créée");
	}
	...
}
```

Variable qui devra être valorisée à un moment



Solutions

L'instrumentation aura pour effet de valoriser le contrat par une instance concrète de l'interface ou la classe abstraite :

```
public class FactureController {

	private FactureServiceInterface fs;

	public void createFactureFromConsoleInput(){
		Scanner sc=new Scanner(System.in);
		System.out.println("Quel est le nom du client ?");
		String nomClient=sc.next();
		Facture fact=new Facture();
		fact.setNomClient(nomClient);
		fact=fs.createFacture(fact);
		System.out.println("La facture "+fact.getNumero()+" à été créée");
	}

	public FactureServiceInterface getFs() {
		return fs;
	}

	public void setFs(FactureServiceInterface fs) {
		this.fs = fs;
	}
}
```

Instrumentation

```
FactureController fc=new FactureController();
FactureServiceInterface fs=new FactureService2();
fc.setFs(fs);
```

Solutions

Managed Beans

- Les opérations d'instrumentation qui vont s'appliquer aux composants de l'application vont impliquer que le cycle de vie du composant (instanciation entre autres) sera géré par une tierce partie (souvent un framework ou de conteneur). Le composant (ou bean) est alors qualifié de managed bean. De nombreuses technologies utilisent cette notion :
  - *JSF* Managed Beans
  - *JMX* Managed Beans
  - *Spring* Managed Beans
    - et même les *Form Beans* de *Struts* auraient en leur temps déjà pu être qualifiés de « Managed Beans » !
- Le terme managed bean ne peut donc se comprendre que s'il est mis en relation avec le (ou les) composants en charge de sa gestion.

Solutions

Réponse apportée par les EJBs

- Selon Sun, l'avènement des EJB il y a maintenant plus de 10 ans devait répondre de manière très large au besoin de composants managés (sous-entendus, dont les aspects techniques sont gérés par le conteneur EJB).
- L'expérience EJB 1 – EJB 2 fut un échec :
  - Nécessité d'utiliser des serveurs d'application J2EE alors hors de prix
  - Composants métier avec une dépendance forte vis à vis de la norme EJB, les objets devaient alors être adaptés à cette technologie (à contrario des POJO).
  - L'intégration avec d'autres frameworks tiers parfois complexes voir ingérables.

Solutions

Réponse apportée par SPRING

Spring est promu par SpringSource qui est désormais la propriété de l'entreprise VMware.

Spring s'est dès sa création posé en alternative à la complexité des EJB 2 en proposant :

- Un découplage systématique des composants en charge des différentes couches logicielles.
- Une configuration de l'architecture logicielle par le biais d'une programmation par « contrat », ce qui permet l'interchangeabilité des composants.
- Une simplification des tests permettant de réaliser un véritable développement piloté par les tests (TDD).

Solutions

Concrètement, Spring propose d'instrumenter le code grâce à deux techniques :

- L'emploi d'un conteneur léger

Il s'agit d'un logiciel qui n'est PAS intégré à la norme JEE. Il va permettre :

  - D'initialiser des composants
  - Gérer le cycle de vie de ces composants
  - Gérer les dépendances entre ces composants

- La possibilité d'utiliser l'AOP (programmation orientée aspect)

  - Facilite l'intégration des couches logicielles transversales (transactions, sécurité, logs, ...)

Conteneur léger et inversion de contrôle

- Un conteneur léger est en réalité un moteur capable de mettre en œuvre le modèle de conception (design pattern) d'inversion de contrôle.
- La notion d'inversion de contrôle (Inversion of Control) est intimement liée à un autre concept, celui d'injection de dépendance (Dependency Injection). Les 2 notions sont proches.
  - L'injection de dépendance est un sous ensemble de l'inversion de contrôle. Lorsque le conteneur léger va contrôler le cycle de vie des composants d'une application, il va à certains moments devoir recourir à l'injection de dépendance pour lier les composants entre eux.

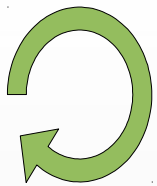
Conteneur léger et inversion de contrôle

- Dans une programmation objet classique, c'est le programmeur qui va :
  - Construire les objets dont il a besoin (new)
  - Les mettre en relation avec d'autres objets créés ou livrés par le système
    - En résumé, les relations entre les objets sont instituées dans le code.

Conteneur léger et inversion de contrôle

- Avec l'IOC c'est le conteneur léger qui va :
  - Instancier les objets (plus d'instanciation explicite)
  - Mettre en relation les objets entre eux grâce à l'injection de dépendance à l'aide notamment de la norme JavaBean
- Rappelez vous l'exemple du service de facturation :

Instrumentation



```
FactureController fc=new FactureController();
FactureServiceInterface fs=new FactureService2();
fc.setFs(fs);
```

Conteneur léger et inversion de contrôle

- L'inversion de contrôle est en réalité un design pattern déjà utilisé par d'autres outils bien connus, comme le conteneur de Servlet ou n'importe quel serveur applicatif JEE « lourd ».
  - Pensez à la manière dont une Servlet, une queue JMS, un EJB où n'importe quel composant géré par un conteneur.
  - Le conteneur créé ces objets et gère leur cycle de vie.

Mise en oeuvre

- La configuration de Spring est stockée dans un fichier XML :
- L'élément principal est `<beans>` :

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

- Chaque classe (Bean) à instancier par le conteneur léger sera déclarée dans une balise `<bean>`. Dans l'exemple qui suit, l'instance de classe qui implémente `FactureServiceInterface` sera nommée « factureService ». De la même manière, on fera instancier notre contrôleur `FactureController` et sera nommé «factureController» :

```
<bean id="factureService" class="com.mycompany.td.service.FactureService2"/>

<bean id="factureController" class="com.mycompany.td.controller.FactureController"/>
```

Spring et l'injection de dépendance

- La valeur d'une propriété d'un Bean peut faire référence à un autre objet Java (un autre Bean). Cet autre bean est alors appelé « collaborateur ».
- L'attribut `ref` de la balise `<property>` permet de faire référence à un autre bean instancié par le conteneur.

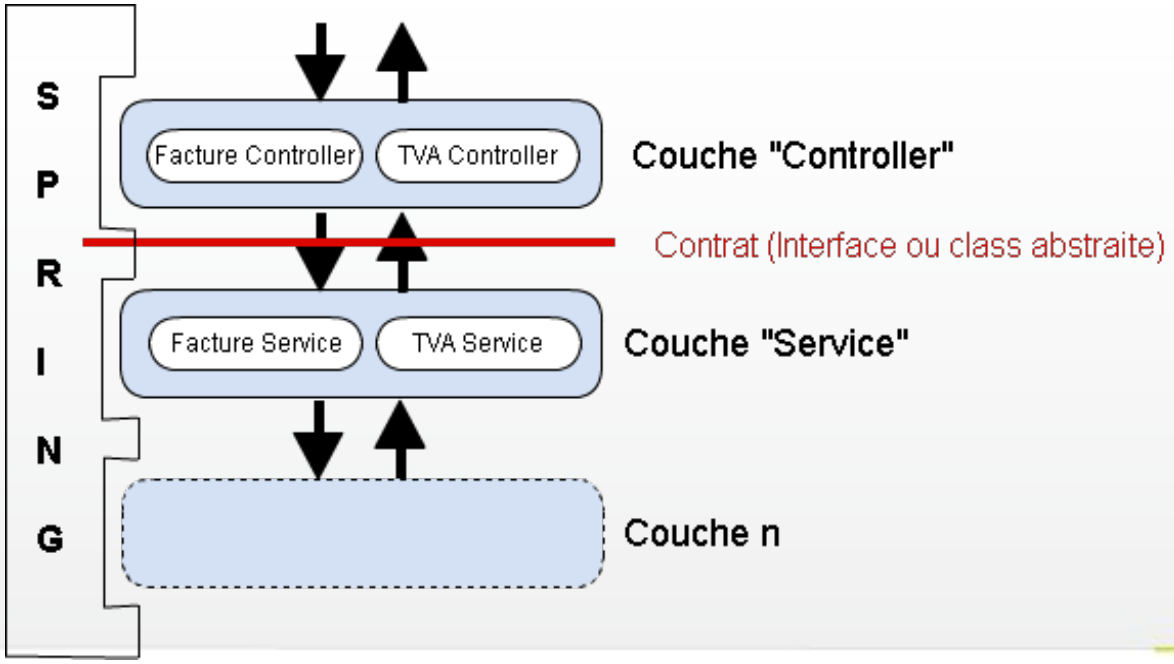
```
<bean id="factureService" class="com.mycompany.td.service.FactureService2"/>

<bean id="factureController" class="com.mycompany.td.controller.FactureController"/>
    <property name="fs" ref="factureService" />
</bean>
```

Conclusion

A quoi sert réellement Spring ?

- Spring permet de configurer l'architecture même d'une application. Cette architecture devient malléable et évolutive. L'implémentation concrète d'une couche logicielle peut à tout moment être modifiée.
  - Spring devient le **ciment** de l'application.



Conclusion

- Spring n'a pas été conçu pour réduire la taille du code.
  - Les lignes de code sont remplacées par des lignes dans un fichier de configuration.
- Spring permet de configurer l'architecture d'une application et en cela :
  - Adapter les logiciels à des nouvelles exigences fonctionnelles ou techniques
  - Favoriser la réutilisation
  - Rester ouvert aux évolutions technologiques
  - Et on le verra, « bouchonner » très facilement une application (Mock, Stubs). Ce qui favorise le développement agile et facilite les tests d'intégration.