

# Projet Informatique

2019

Steve Hostettler

Software Modeling and Verification Group

University of Geneva



---

# Injection de dépendances & inversion de contrôle



# Objectifs

Moins de couplage (connaissance “hard-codée”) entre les composants



# Couplage?

$A \rightarrow B \rightarrow C$

$A \rightarrow B \rightarrow A$

$A \rightarrow B, C \rightarrow G, D, E \rightarrow A$

<http://depfind.sourceforge.net/>

# Concepts

- Patron de conception d'architecture
- Le flot d'exécution du logiciel n'est pas sous le contrôle du programmeur mais du conteneur ou du framework
- Hollywood principle: Don't call us, we'll call you!



# Concepts

- On se concentre sur la valeur ajoutée
- JSF est une forme d'inversion de contrôle puisque c'est le conteneur qui gère la mécanique d'exécution.
- Différentes implémentations: injection de dépendances, factory pattern...



---

# Patron de conception : Factory + Service Locator



# Factory: objectifs

Séparer l'utilisation d'un composant par un autre du choix de l'implémentation et de son instantiation





# Factory: mise en oeuvre

```
public class MyClass {  
  
    IMyComponent component = new MyComponent();  
  
    public void myMethod() {  
        component.doSomething();  
    }  
}
```

Implémentation  
choisie

mécanisme  
d'instantiation

Utilisation

# Factory: mise en oeuvre

```
public class MyComponentFactory {  
  
    public static IMyComponent instance() {  
        return new MyComponent();  
    }  
}
```

mécanisme  
d'instantiation

Implémentation  
choisie

```
public class MyClass {  
  
    IMyComponent component =  
        MyComponentFactory.instance();  
  
    public void myMethod() {  
        component.doSomething();  
    }  
}
```

Utilisation

# Service Locator: objectifs

Séparer l'utilisation d'un composant par un autre de sa position effective

Contrairement à Factory, le service locator n'instancie pas nécessairement le service



# Service Locator: mise en oeuvre

```
public class MyServiceLocator {  
  
    private MyServiceLocator() {}  
  
    public static MyServiceLocator instance() {  
        return new MyServiceLocator();  
    }  
    public IMyComponent GetComponent() {  
        /*Get the service from wherever it is*/  
    }  
}  
  
public class MyClass {  
  
    public void myMethod() {  
        MyServiceLocator.GetComponent().doSomething();  
    }  
  
}
```



# Problèmes de ces approches

- Toujours des références vers la Factory ou le ServiceLocator
- Le singleton...c'est le mal
- Pas très souple pour les objets "mock"



---

# Injection de dépendances



# Concepts

- PUSH vs PULL
- Sélection de dépendances @Runtime et non pas @Compiletime
- Pour chaque objet, il suffit de préciser le contrat que doit satisfaire ses dépendances et le conteneur se chargera de trouver un objet valide



# Exemple avec JSR-299

```
public class MyClass {  
  
    @Inject  
    private IMyComponent component;  
  
    public void myMethod() {  
        component.doSomething();  
    }  
  
}
```



# Avantages

- Elimination des patrons de conception “Singleton”, “Factory” et “Service Locator”
- Elimination de code non-métier “inutile”
- Flexibilité de la configuration
  - pratique pour les tests
  - fausses implémentations (mock)



# Inconvénients

- Les moins
  - Risque de “magie noire”
  - Besoin de code de “bootstrap”
  - Possible contraintes sur la forme des classes contenues (pas final)



# Types d'injections

- Type 1 : injection d'interface
- Type 2 : injection par mutateurs
- Type 3 : injection par constructeurs



# Injection d'interface

```
public interface Injected {  
    void inject(IMyComponent);  
}
```

```
public class Container  
    public Container() {  
        Injected c = new MyClass();  
        c.inject(  
            new MyComponentImpl());  
    }  
}
```

```
public class MyClass implements Injected {  
    @Inject  
    private IMyComponent component;  
    public void inject(IMyComponent c) {  
        this.component = c;  
    }  
  
    public void myMethod() {  
        component.doSomething();  
    }  
}
```

# Injection par mutateurs

```
public class MyClass {  
    @Inject  
    IMyComponent component;  
  
    public void setComponent(IMyComponent c) {  
        this.component = c;  
    }  
  
    public void myMethod() {  
        component.doSomething();  
    }  
}  
  
public class Container  
    public Container() {  
        MyClass c = new MyClass();  
        c.setComponent(new MyComponentImpl());  
    }  
}
```

# Injection par constructeur

```
public class MyClass {  
    @Inject  
    IMyComponent component;
```

```
    public MyClass(IMyComponent c) {  
        this.component = c;  
    }  
}
```

```
public class Container  
    public Container() {  
        MyClass c = new MyClass(new MyComponentImpl());  
    }  
}
```

# Injection de dépendances

Le code en bleu peut-être géré par un framework externe appelé conteneur



---

# JSR-299

## constructor injection





# Principales Fonctionnalités

- Un bean n'a pas besoin d'être explicitement nommé
- Caractérisation possible par typage fort (annotations) ou par chaîne de caractères
- Pas besoin de getters et de setters
- Gestion des implémentations alternatives
- Ajout du scope de Conversation permet une gestion très souple des flows de l'application
- Un mécanisme d'interception "à la AOP"



## **Règle principale**

# The Highlander Rule

**There can be only one**

**...**

**implementation that fulfils  
the contract**



# Limites

Un service doit implémenter Serializable et ne pas être “final” → possible problème de design



# Injecter un bean

```
/** Only one instance per flow. */  
@ConversationScoped  
public class ManageStudentRegistration implements Serializable {  
  
    /** The serial-id. */  
    private static final long serialVersionUID = 2123342218792192804L;  
    ...  
  
    /** The service that provides the business logic for the student registration process. */  
    @Inject  
    private StudentService mService;  
  
    public ManageStudentRegistration() {  
        ...  
    }  
}
```

Injection d'un service qui respecte l'interface StudentService. Une classe qui satisfait ce service doit exister dans le Classpath



# Déclaration d'un bean

Une seule instance par application

```
@ApplicationScoped
public class StudentServiceImpl implements StudentService, Serializable {
    ...
    /** The serial-id. */
    private static final long serialVersionUID = 1386508985359072399L;
    ...

    /**
     * Empty constructor.
     */
    public StudentServiceImpl() {
        LOGGER.info("This is the real implementation");
    }
    ...
}
```

# Configurer l'injection

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans\_1\_0.xsd>  
</beans>
```

Ce fichier configure les services  
alternatifs,  
les intercepteurs et les  
décorateurs

Un fichier beans.xml peut être présent soit dans META-INF ou dans WEB-INF. Il peut être vide

# Configuration du conteneur ([web.xml](#))

Cet servlet initialise le conteneur

```
<listener>  
  <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>  
</listener>
```

# Le scope “ConversationScoped”

- Permet de fixer la durée de vie d'un objet manuellement : sur plusieurs requêtes par exemple
- Le programmeur doit définir le point de départ de la conversation et la terminer





# Le scope “ConversationScoped”

Le scope est “ConversationScoped”

```
@ConversationScoped  
public class ManageStudentRegistration implements Serializable {  
...
```

```
/** Inject the current conversation. */  
@Inject  
private Conversation mConversation;
```

Injection de la conversation courante.  
Si pas de conversation alors création  
d'une nouvelle

# Le scope “ConversationScoped”

## Début de conversation

```
/**  
 * Action that forwards to the registration page.  
 *  
 * @return the next action to perform  
 */  
public String toRegistration() {  
    LOGGER.debug("registration")  
    this.mStudent = new Student()  
    //Starts the flow  
    this.mConversation.begin();  
    return "register";  
}
```

Démarre une nouvelle conversation  
à l'intérieur de la session

# Le scope “ConversationScoped”

## Fin de conversation

```
public String add() {  
    this.mStudent.validate();  
    this.mService.add(this.mStudent);  
    //Ends the flow  
    this.mConversation.end();  
    return "success";  
}
```

Termine une conversation

```
public String toList() {  
    LOGGER.debug("list");  
    //Ends the flow  
    this.mConversation.end();  
    return "list";  
}
```

Termine une conversation

# Gérer les conflits

Plusieurs implémentation du service mais spécifiques à des besoins particuliers

Utilisation de qualificateurs pour les distinguer.

```
@ApplicationScoped  
public class StudentServiceImpl implements StudentService, Serializable {
```

```
@RestImpl  
@ApplicationScoped  
public class StudentServiceRestImpl extends StudentServiceImpl {
```

```
@SoapImpl  
@ApplicationScoped  
public class StudentServiceSoapImpl extends StudentServiceImpl {
```

qualificateur

qualificateur



# Gérer les conflits

qualificateur

```
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER})  
public @interface RestImpl { }
```

Important

```
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})  
public @interface SoapImpl { }
```

Applicable sur un champs, un type,  
une méthode ou un paramètre

# Gérer les conflits.

```
/** The name of the instance of this object used in the JSF. */
@Named
/** Only one instance per flow. */
@ConversationScoped
public class ManageStudentRegistration implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = 2123342218792192804L;
    ...

    /** The service that provides the business logic for the student registration process. */
    @Inject @RestImpl
    private StudentService mService;

    public ManageStudentRegistration() {
        ...
    }
}
```



# Gérer les conflits.

```
/** The name of the instance of this object used in the JSF. */
@Named
/** Only one instance per flow. */
@ConversationScoped
public class ManageStudentRegistration implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = 2123342218792192804L;
    ...

    /** The service that provides the business logic for the student registration process. */
    @Inject @SoapImpl
    private StudentService mService;

    public ManageStudentRegistration() {
        ...
    }
}
```



# Déclaration d'une implémentation alternative

```
@Alternative  
@ApplicationScoped  
public class ...
```

Annotation (stereotype) de la classe. Il s'agit d'une alternative.  
Les beans annotés avec alternatives sont ignorés par défaut.  
Il faut les activer dans le fichier beans.xml

```
/**  
 *  
 */
```

```
private List<Student> mStudentList;
```

```
/**  
 *  
 */
```

```
public StudentServiceMockImpl() {  
    LOGGER.info("This is the mock implementation");  
    this.mStudentList = new ArrayList<Student>();  
}
```



# Utiliser un service alternatif

Cette ligne active une implémentation alternatives. Elle est inactive par défaut

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <alternatives>
    <class>ch.unige.business.service.mock.StudentServiceMockImpl</class>
  </alternatives>
</beans>
```

# Exercices

- Exercice 1
  - Programmer un service simple
  - Injecter ce service dans une servlet
  - Enlever le service précédent du classpath et en créer un autre
- Exercice 2
  - Transformer les deux service précédents en alternatives
  - Créer des annotations pour différencier les services



# Bibliographie

- <http://martinfowler.com/articles/injection.html>
- [http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/dependency\\_injection.html](http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/dependency_injection.html)