



Développement par composants (modulaires) Spring

Intervenant : Chouki TIBERMACHINE

Bureau : LIRMM (E.311)

Tél. : 04.67.14.97.24

Mél. : Chouki.Tibermachine@lirmm.fr

Web : <http://www.lirmm.fr/~tibermacin/ens/hmin304/>

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- Injection de dépendances avec Spring

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- Injection de dépendances avec Spring

Qu'est-ce qu'un module ?

- Module = unité de code indépendante dans un système logiciel
- Deux caractéristiques fondamentales :
 - Forte cohésion : le module se focalise sur une tâche unique
 - **Couplage minimal** : le module a de faibles dépendances avec les autres modules
- La modularité existe depuis longtemps (le langage Modula dans les années 70, un des ancêtres de Java)

Description d'un module

- A l'origine, un module est décrit par :
 - Une spécification du module : signatures des fonctions publiques fournies par le module
 - ~ une interface Java
 - Une implémentation du module : implémentation des fonctions fournies
 - ~ une classe Java qui implémente l'interface
- La plupart des langages de programmation par objets fournissent les moyens pour organiser son code ainsi

Intérêt de la modularité

- **Maintenabilité :**

- Module = unité facilement compréhensible indépendamment des autres modules (comparativement à un système complet)
- Module = unité pouvant être changée (son implémentation) sans grand impact sur les autres modules (qui sont liés à l'interface)

- **Réutilisabilité :**

- Module = unité facilement réutilisable dans d'autres contextes (faisant partie d'autres applications)

- **Testabilité :**

- Module = unité sur laquelle les tests unitaires peuvent être facilement réalisés

Du module au composant

De la « Séparation Spécification-Implémentation »



À la « Séparation Logique métier-Architecture »

Composant = Module++

Rappel de ce qu'est un composant

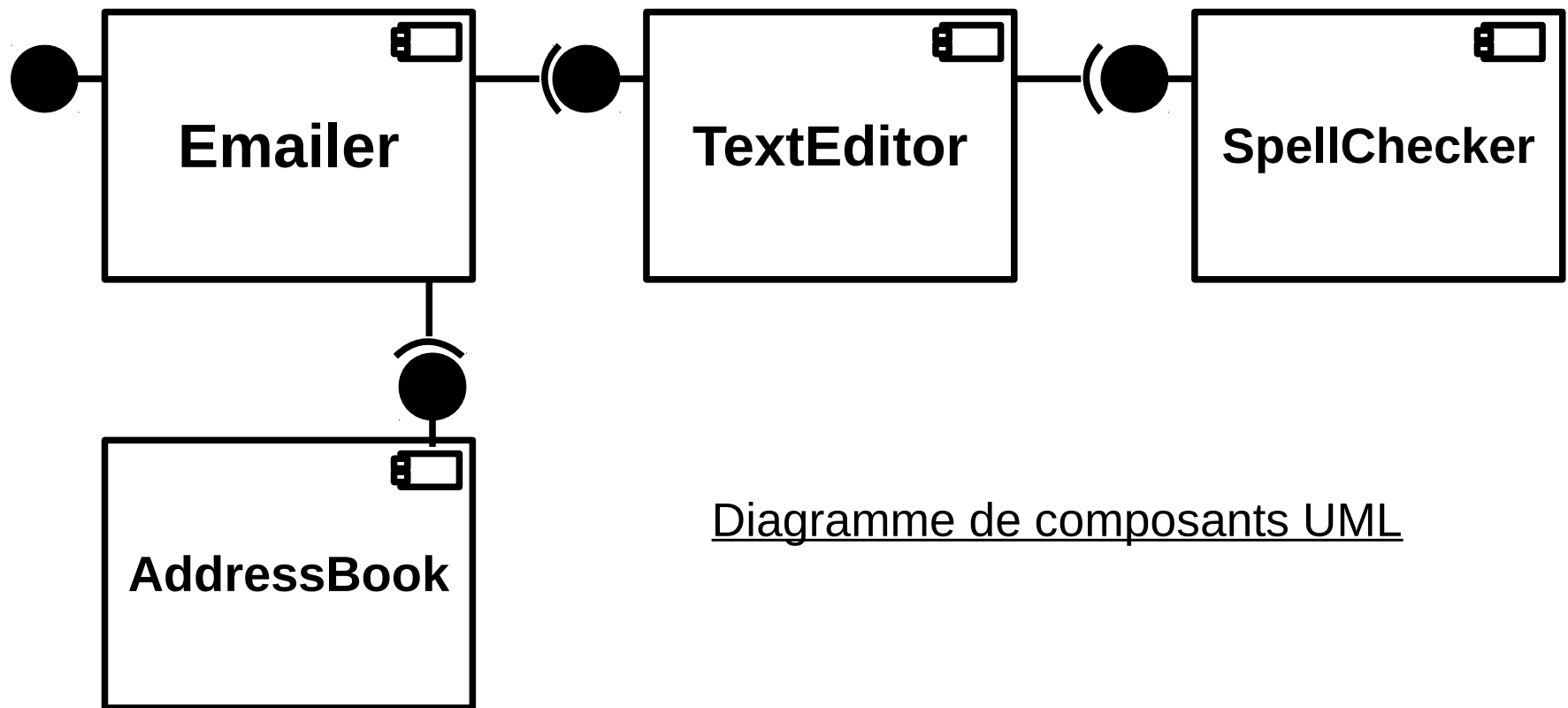
- Un composant est une unité logicielle qui :
 - décrit de façon explicite ses :
 - fonctionnalités fournies : signatures des opérations qu'il offre (comme dans les modules)
 - fonctionnalités requises : signatures des opérations dont il a besoin pour implémenter les fonctionnalités fournies
 - [son architecture interne : liste des instances de ses composants internes et leurs interconnexions]
 - est sujet à une instanciation et à une connexion avec d'autres composants (qui ont besoin de ses opérations / qui répondent à ses besoins -fournissent ses opérations requises-)

Modularité dans les composants logiciels

- Interfaces requises explicites :
 - Un composant ne dépend pas directement d'un autre composant
 - Il requiert un certain nombre d'opérations (spécifiées dans une ou plusieurs interface(s) : type abstrait) que peut fournir (implémenter) un autre composant
- Dépendances entre composants mises en place par un architecte :
 - Un rôle différent de celui qui développe les composants (qui sont à connecter, et destinés pour la réutilisation) : développeur par la réutilisation

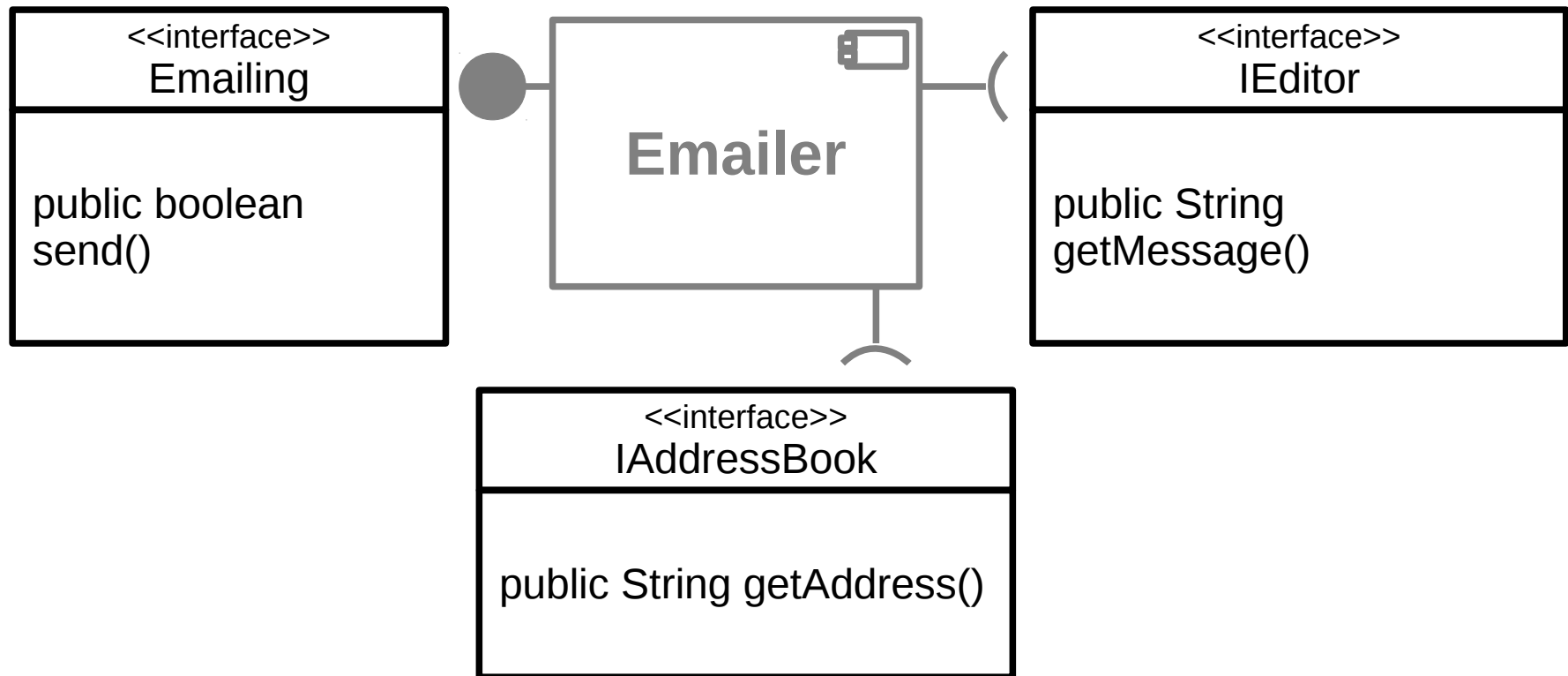
Modularité dans Java : un exemple

- Architecture simplifiée d'un exemple jouet :



Composant Emailer en « UML »

- Interface fournie et interfaces requises :



Composant Emlaler en Java


■ Classe Emlaler :

```
public class Emlaler implements Emlaling {  
  
    public IEditor editor;  
    public IAddressBook addressBook;  
  
    // Constructeurs ...  
  
    public boolean send() {  
        String msg = editor.getMessage();  
        String address = addressBook.getAddress();  
        return sendMessage(msg,address); // Méthode locale  
    }  
}
```

Composant Emailer en Java

■ Classe Emailer :

```
public class Emailer implements Emailing {  
  
    public IEditor editor;  
    public IAddressBook addressBook;  
  
    // Constructeurs ...  
  
    public boolean send() {  
        String msg = editor.getMessage();  
        String address = addressBook.getAddress();  
        return sendMessage(msg, address); // Méthode locale  
    }  
}
```



Composant Emlaler en Java

■ Classe Emlaler :

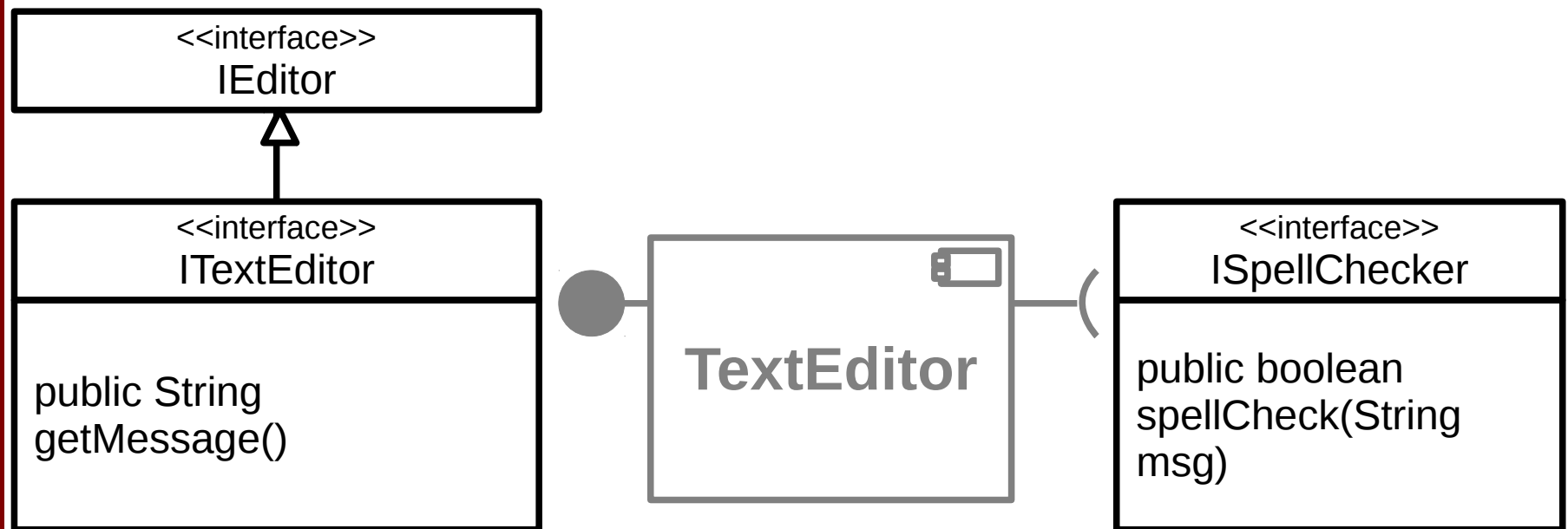
```
public class Emlaler implements Emlaling {  
  
    public IEditor editor;  
    public IAddressBook addressBook;  
  
    // Constructeurs ...  
  
    public boolean send() {  
        String msg = editor.getMessage();  
        String address = addressBook.getAddress();  
        return sendMessage(msg, address); // Méthode locale  
    }  
}
```

Interface fournie

Interfaces requises

Composant TextEditor en « UML »

- Interface fournie et interface requise :



Composant TextEditor en Java

■ Classe TextEditor :

```
public class TextEditor implements ITextEditor {  
  
    public ISpellChecker checker;  
  
    // Constructeurs ...  
  
    public String getMessage() {  
        ... {  
            String msg = getUserInput(); // Méthode locale  
            boolean isCorrect = checker.spellCheck(msg);  
            if(isCorrect) return msg;  
        }  
    }  
}
```


Composant TextEditor en Java

- Classe TextEditor :

```
public class TextEditor implements ITextEditor {  
  
    public ISpellChecker checker;  
  
    // Constructeurs ...  
  
    public String getMessage() {  
        ... {  
            String msg = getUserInput();// Méthode locale  
            boolean isCorrect = checker.spellCheck(msg);  
            if(isCorrect) return msg;  
        }  
    }  
}
```

Décrire l'architecture et lancer l'application

- Définir une classe avec un main dans laquelle on instancie les différentes classes qui représentent les composants
- Les dépendances entre ces composants sont résolues de deux façons :
 - En initialisant les attributs (interfaces requises) avec des références aux instances créés – solution 1
 - En passant ces références sous la forme d'arguments aux constructeurs des différentes classes au moment de l'instanciation – solution 2

Décrire l'architecture – solution 1

- Classe de description d'archi. et de lancement de l'application :

```
public class MainApplication {  
    private static Emailing emailer;  
    private static void configureApplication() {  
        emailer = new Emailer();  
        ITextEditor editor = new TextEditor();  
        editor.setChecker(new SpellChecker());  
        emailer.setEditor(editor); // ou setter  
        emailer.setAddressBook(new AddressBook());  
    }  
    public static void main(String[] args) {  
        configureApplication();  
        System.out.println(emailer.send());  
    }  
}
```

Décrire l'architecture – solution 2

- Classe de description d'archi. et de lancement de l'application :

```
public class MainApplication {  
    private static Emailing emailer;  
    private static void configureApplication() {  
        ISpellChecker checker = new SpellChecker();  
        IEditor editor = new TextEditor(checker);  
        IAddressBook book = new AddressBook();  
        emailer = new Emailer(editor, book);  
    }  
    public static void main(String[] args) {  
        configureApplication();  
        System.out.println(emailer.send());  
    }  
}
```

Constructeur de la classe Emlaler

```
public class Emlaler implements Emlaling {  
  
    public IEditor editor;  
    public IAddressBook addressBook;  
  
    public Emlaler(IEditor e, IAddressBook a) {  
        this.editor = e;  
        this.addressBook = a;  
    }  
  
    public boolean send() {  
        // ...  
    }  
}
```

Avoir des dépendances noyées dans le code

```
public class TextEditor implements ITextEditor {  
    public ISpellChecker checker;  
  
    public TextEditor() {  
        checker = new EnglishSpellChecker();  
    }  
  
    // ...  
}
```

Avoir des dépendances noyées dans le code

```
public class TextEditor implements ITextEditor {  
    public ISpellChecker checker;  
  
    public TextEditor() {  
        checker = new EnglishSpellChecker();  
    }  
    // ...  
}
```

↑
Et si on veut utiliser un
new FrenchSpellChecker() ?

Avantages de la solution : constructeur avec arguments

- Dépendances entre composants peuvent être définies depuis l'extérieur par l'architecte de l'application et non par le développeur du composant Mailer
- Séparation des préoccupations :
 - Code représentant la logique métier de l'application séparé du code décrivant l'architecture de celle-ci
 - C'est la classe MainApplication qui s'occupe de la description d'architecture

Avantages de la solution : attributs = interfaces requises

- Possibilité de connecter n'importe quel composant, qui implémente l'interface requise (ou bien une interface dérivée de celle-ci)
 - Le composant requiert une spécification abstraite
 - Il ne dépend pas d'une implémentation particulière

- Exemple :

```
public class FrenchSpellChecker
    implements ISpellChecker {
    public String spellCheck() { /* ... */ }
}
```

Le composant `TextEditor` peut être connecté à une instance de `FrenchSpellChecker` :

```
IEditor editor = new TextEditor(new FrenchSpellChecker());
```

Améliorations possibles

- Au lieu de gérer la création de ces instances et leur connexion, demander ça à des objets tiers
- Ces objets peuvent être distants. Ils centralisent des pools d'objets (qui peuvent être uniques), qui peuvent avoir accès à certaines ressources « précieuses » (bases de données, ...)
- Le programmeur est déchargé de cela. Il doit s'occuper de la logique métier de son application

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- Injection de dépendances avec Spring

Patron « Factory » ou « Abstract Factory »

- Le patron « Factory » (ou « Abstract Factory ») préconise la définition d'une classe qui crée des instances et gère leurs dépendances
- Pour notre exemple, on peut écrire une classe `EmailerFactory` avec une méthode qui ressemble d'une certaine manière à la méthode `configureApplication()`

Exemple avec ce patron

```
public class EmlerFactory {  
    public Emling newFrenchEmler() {  
        ISpellChecker checker = new FrenchSpellChecker();  
        IEditor editor = new TextEditor(checker);  
        IAddressBook book = new AddressBook();  
        Emler emler = new Emler(editor,book);  
        return emler;  
    }  
    public Emling newGermanEmler() {  
        ISpellChecker checker = new GermanSpellChecker();  
        // ...  
    }  
}
```

Avantages de ce patron

- Un composant client (notre MainApplication, dans l'exemple) est déchargé de ce travail d'instanciation et de gestion des dépendances
- Un composant client n'a besoin de connaître que le composant EMailerFactory (et non les autres composants)
- Encore de la séparation des préoccupations :-)
 - Code de création d'instances et de gestion de leurs dépendances séparé du code d'utilisation du composant EMailer :
`new EMailerFactory().newFrenchEmailer() ...`
ou souvent (méthode factory statique)
`EMailerFactory.newFrenchEmailer() ...`

Inconvénients de ce patron

- Il faudra créer autant de méthodes que de variations, dans les composants et leurs dépendances, qu'on souhaite obtenir avec ce « factory »
- Il faudra définir autant de méthodes `new...Emailer()` que de langues qu'on souhaite avoir pour le *spell checker*
- Et si on voudrait changer `AddressBook` par `PhoneAndAddressBook`
 - Il faudra changer chacune de ces méthodes (+ combinaisons)

Patron « Service Locator »

- Le patron « Service Locator » est une sorte de « factory »
- Il s'agit d'un objet tiers utilisé pour obtenir une instance ayant toutes ses dépendances résolues
- Ça peut être un objet qui s'exécute sur la même machine virtuelle que les autres composants de l'application qu'on construit, comme ça peut être un objet distant
- JNDI est un bon exemple de « Service Locator » utilisé dans les composants EJB, Web Java, ..

Exemple avec ce patron

- Utilisation de ce patron :

```
Emailer emailer = (Emailer) new ServiceLocator()  
                        .get("emailer");
```

- Utilisation de **clés** pour obtenir une référence vers un composant
 - Le mot "emailer" dans notre exemple
- Les mêmes problèmes que nous avons avec le « Factory » se posent avec ce patron
- De plus, les clés utilisées sont opaques et peuvent être ambiguës
 - Possibles erreurs à l'exécution (si la clé n'est pas correcte)

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- Injection de dépendances (DI) avec Spring :
 - Généralités sur la DI et introduction à Spring
 - Définition de composants Spring
 - Connexion de composants Spring avec la DI
 - Connexion de composants via le langage SpEL
 - Auto-connexion et auto-découverte de composants Spring

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- Injection de dépendances (DI) avec Spring :
 - Généralités sur la DI et introduction à Spring
 - Définition de composants Spring
 - Connexion de composants Spring avec la DI
 - Connexion de composants via le langage SpEL
 - Auto-connexion et auto-découverte de composants Spring

Injection de dépendances (DI)

- Une entité externe à l'application :
 - crée les instances nécessaires
 - les lie ensemble
- De cette façon, les développeurs d'une application sont déchargés de cette tâche et peuvent se focaliser uniquement sur la programmation de la logique métier

Exemple d'injection manuelle de dépendances

- Le constructeur de la classe `Emler` (vu précédemment) :

```
public Emler(IEditor e, IAddressBook a) {  
    this.editor = e;  
    this.addressBook = a;  
}
```

- Dans une autre classe, on écrit :

```
new Emler(new TextEditor(), new AddressBook());
```

- On dit ici qu'on a injecté manuellement des dépendances dans une instance de la classe `Emler` à travers son constructeur

Frameworks d'injection automatique de dépendances

- L'injection de dépendances peut être réalisée de façon automatique
- Il existe des frameworks Java proposant des solutions d'injection automatique de dépendances
- Quelques frameworks gratuits :
 - Spring : AOP, Web, ... (**celui qui a popularisé la DI**)
 - Guice de Google (le plus simple)
 - PicoContainer (l'un des premiers frameworks)
 - Quelques projets chez Apache : Avalon et HiveMind
 - Chez Oracle (Sun) : DI dans JEE (inspirée de Spring et Guice)

Un peu d'histoire sur Spring

- Décembre 1996 : JavaBeans - modèle de composants Java
 - se limitant principalement aux interfaces graphiques
- Mars 1998 : EJB - modèle de composants plus riche (transactions, persistance, ...)
 - répondant aux besoins réels des applications d'entreprise (mais rendant le code assez complexe et verbeux)
- Juin 2003 : Spring - une simplification du développement des applications d'entreprise (DI, AOP, ...)
 - développement de la logique métier des applications avec des POJOs (Plain-Old Java Objects)
 - mise en place des autres aspects en utilisant un modèle de programmation déclarative

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- **Injection de dépendances (DI) avec Spring :**
 - Généralités sur la DI et introduction à Spring
 - **Définition de composants Spring**
 - Connexion de composants Spring avec la DI
 - Connexion de composants via le langage SpEL
 - Auto-connexion et auto-découverte de composants Spring

Mettre en place l'environnement Spring

- Configuration Spring (fichier XML) :

```
<?xml version="1.0" encoding="UTF-8" ?>
<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">

  <!-- Déclarations de composants (beans) -->

</beans>
```

- Trois modes de description d'architecture :

- Dans des fichiers XML (déclarations de beans ci-dessus)
- Annotations dans le code (avec un minimum d'XML)
- Code Java ordinaire, mais utilisé de façon spécifique par le framework

Déclarer des composants (beans) dans Spring

- Un bean = une classe Java ordinaire implémentant une ou plusieurs interfaces (les interfaces fournies du composant)
- Déclaration du bean dans le fichier de configuration XML :
`<bean id="emailer" class="fr.lirmm.marel.di.Emailer"/>`
- Le framework Spring est informé alors qu'il doit créer un objet de la classe ci-dessus et l'enregistrer avec l'id « emailer »

Obtenir une référence vers un bean

- Charger le contexte de l'application :

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(  
    "fr/lirmm/marel/di/spring-emailer.xml");
```

Charger un fichier depuis le CLASSPATH (même dans un JAR référencé dans le CP)

Ou

```
ApplicationContext ctx = new FileSystemXmlApplicationContext(  
    "./spring-emailer.xml");
```

Charger un fichier depuis le disque

Ou

```
ApplicationContext ctx = new XmlWebApplicationContext(  
    ".../spring-emailer.xml");
```

Charger un fichier depuis une application Web

- Demander au framework une référence vers un bean :

- Une implémentation d'un Service Locator

```
Emailer em = (Emailer) ctx.getBean("emailer");  
em.send(...);
```

Définir la portée (*scope*) d'un bean

- Plusieurs portées possibles. Celles qui nous intéressent ici sont :
 - Singleton :
 - Une seule instance est créée pour un contexte d'application
 - C'est la valeur par défaut
 - Prototype :
 - Plusieurs instances peuvent être créées
- Définir la portée :
`<bean id="emailer" class="fr.lirmm.mare1.di.Emailer" scope="prototype"/>`
- Les autres portées sont liées aux applications Web (request, ...)

Réagir à la création et la destruction de beans

- Il est possible de définir des méthodes qui seront invoquées au moment de la création et de la destruction d'un bean
- Plusieurs façons possibles :
 - Ajouter dans la déclaration du bean :

```
<bean id="emailer" class="fr.lirmm.marel.di.Emailer"  
      init-method="configureEmailAcount"  
      destroy-method="unconfigureEmailAcount" />
```
 - Implémenter les interfaces `InitializingBean` et `DisposableBean`
 - Le faire une fois pour toute l'application avec les attributs : `default-init-method` et `default-destroy-method` dans `<beans />`

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- **Injection de dépendances (DI) avec Spring :**
 - Généralités sur la DI et introduction à Spring
 - Définition de composants Spring
 - **Connexion de composants Spring avec la DI**
 - Connexion de composants via le langage SpEL
 - Auto-connexion et auto-découverte de composants Spring

Injecter les constructeurs du bean

- Deux cas de figure :

- Injecter une valeur de type primitif ou String :

```
<bean id="message" class="fr.lirmm.mare1.di.Message">  
    <constructor-arg value="120" />  
</bean>
```

Taille max du message

Si aucun constructor-arg n'est défini, le constructeur par défaut est utilisé

- Injecter une référence :

```
<bean id="emailer" class="fr.lirmm.mare1.di.Emailer">  
    <constructor-arg ref="editor" />  
    <constructor-arg ref="addressBook" />  
</bean>  
<bean id="editor" class="fr.lirmm.mare1.di.Editor">  
    ...
```

Identifiants de beans

Injecter les propriétés des beans

- Propriété d'un bean \iff Propriété d'un JavaBean :

- Un attribut privé ayant un getter et un setter

- Injecter une valeur de type primitif ou String :

```
<bean id="message" class="fr.lirmm.marel.di.Message">  
    <property name="maxSize" value="120" />  
</bean>
```

- Injecter une référence :

```
<bean id="textEditor" class="fr.lirmm.marel.di.TextEditor">  
    <property name="spellChecker" ref="frSpellChecker"/>  
</bean>
```

```
<bean id="frSpellChecker" class="fr.lirmm.marel.di.FrenchSC">
```

- Ne pas oublier de définir un constructeur sans paramètres (si vous avez défini d'autres constructeurs dans la classe du bean) : dans la DI, il est utilisé en premier, puis le setter est invoqué

Changer la configuration des beans

- Exemple de reconfiguration de notre application :

```
...  
<bean id="textEditor" class="fr.lirmm.marel.di.TextEditor">  
    <property name="spellChecker" ref="enSpellChecker"/>  
</bean>  
<bean id="enSpellChecker" class="fr.lirmm.marel.di.EnglishSC">  
...  

```

- Aucune modification dans le code des classes des beans
- Aucune re-compilation du code
- Simple relancement de l'exécution du code de chargement de l'application

Définir des beans composites (grâce aux *inner beans*)

- Possibilité de définir des beans qui ne sont pas partageables
- Déclarer un bean à l'intérieur d'un autre bean (à la façon des classes internes) : *inner bean*

- Bean anonyme, sans identifiant :

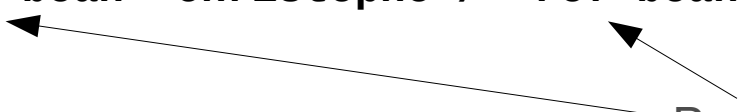
```
<bean id="textEditor" class="fr.lirmm.mare1.di.TextEditor">  
  <property name="spellChecker">  
    <bean class="fr.lirmm.mare1.di.EnglishSC"/>  
  </property>  
</bean>
```

Connecter un bean à une collection de beans

- Quatre sortes de collections dans Spring :

- Listes : Pas de doublons d'un bean dans la collection

```
<bean id="message" class="fr.lirmm.marel.di.Message">  
  <property name="destinataires">  
    <list>  
      <ref bean="christophe"/><ref bean="marianne"/>  
    </list>  
  </property>  
</bean>
```



Possibilité de remplacer ref par value, bean ou null

- Ensembles : Doublons d'un bean possibles

```
<set>  
  <!-- Des ref, value, bean ou null -->  
</set>
```

- Maps (tableaux associatifs key-value) ou Properties (tableaux associatifs de Strings)

- Dans le code, nous utilisons l'interface Collection pour les set et list, l'interface Map pour les map et Properties pour props

Connecter un bean à une map de beans

- Utiliser une map :

```
<bean id="message" class="fr.lirmm.marel.di.Message">
  <property name="destinataires">
    <map>
      <entry key="Christophe DONY" value-ref="christophe"/>
      <entry key="Marianne HUCHARD" value-ref="marianne"/>
      <entry key="Abdelhak SERIAI" value-ref="djamel"/>
    </map>
  </property>
</bean>
```

- Utiliser une props : comme une map mais String-String

```
<bean id="message" class="fr.lirmm.marel.di.Message">
  <property name="destinataires">
    <props>
      <prop key="DONY">dony@lirmm.fr</prop>
      <prop key="HUCHARD">huchard@lirmm.fr</prop>
    </props>
  </property>
</bean>
```

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- Injection de dépendances (DI) avec Spring :
 - Généralités sur la DI et introduction à Spring
 - Définition de composants Spring
 - Connexion de composants Spring avec la DI
 - Connexion de composants via le langage SpEL
 - Auto-connexion et auto-découverte de composants Spring

Pourquoi SpEL ?

- Pour le moment, les connexions sont déterminées de façon statique (connecter un bean à un autre bean connu, ou injecter une valeur de type primitif)
- Et si on voudrait définir des connexions qui sont résolues dynamiquement ?
- Nous pouvons utiliser SpEL : Spring Expression Language
 - Référencer des beans (on sait déjà le faire sans ce langage)
 - Invoquer des méthodes et accéder aux propriétés des beans
 - Utiliser des opérations arithmétiques, logiques et de comparaison
 - Tester des expressions régulières
 - Manipuler des collections, ...

Exemples avec SpEL

- Utiliser des valeurs littérales :

```
<property name="maxSize" value="#{120}"/>  
<property name="message" value="The value is #{120}"/>
```

Possibilité aussi d'utiliser des booléens, des réels ou des Strings

- Référencer d'autres beans, invoquer leurs méthodes et accéder à leurs propriétés :

```
<property name="emailer" value="#{emailer}"/>  
<property name="config" value="#{emailer1.config}"/>  
<property name="config" value="#{configs.selectConfig(...)}"/>  
<property name="pi" value="#{T(java.lang.Math).PI}"/>
```

- Utiliser des opérateurs, expressions régulières et collections :

```
<property name="circonference"  
    value="#{2 * T(java.lang.Math).PI * cercle.r}"/>  
<property name="estVide" value="#{message.size == 0}"/>  
<property name="validEmail"  
    value="#{admin.email matches '[a-z]...}'"/>  
<property name="defaultDest" value="#{destinataires[0]}"/>
```

Plan du cours

- Introduction à la programmation modulaire en JAVA
- Quelques patrons de conception pour la modularité
- Injection de dépendances (DI) avec Spring :
 - Généralités sur la DI et introduction à Spring
 - Définition de composants Spring
 - Connexion de composants Spring avec la DI
 - Connexion de composants via le langage SpEL
 - Auto-connexion et auto-découverte de composants Spring

Auto-connecter des beans

- Objectif : éliminer du XML et laisser le framework Spring connecter automatiquement les beans
- Il existe quatre sortes d'auto-connexion :
 - **Par nom** : demander au framework de trouver un bean ayant un id qui correspond au nom de la propriété du bean à auto-connecter
 - **Par type** : trouver un bean ayant un type affectable à une propriété du bean à auto-connecter
 - **Par constructeur** : chercher des beans qui sont de types affectables aux paramètres du constructeur du bean à auto-connecter
 - **Par auto-détection** : par constructeur, puis par type

Auto-connecter des beans par nom

- Le framework va chercher un bean qui a un id correspondant au nom de la propriété du bean à auto-connecter

- Exemple :

```
<bean id="spellChecker" class="fr.lirmm.marel.di.EnglishSC"/>
```

```
<bean id="textEditor" class="fr.lirmm.marel.di.TextEditor"  
    autowire="byName"/>
```

Ici, le bean qui a comme id « *spellChecker* » est connecté à la propriété nommée « *spellChecker* » du bean « *textEditor* »

Auto-connecter des beans par type

- Chercher un bean qui a un type affectable à la propriété du bean à auto-connecter : `autowire="byType"`
- Cela ressemble plus aux connexions entre composants ayant une interface requise et une interface fournie : typer la propriété du bean à auto-connecter avec une interface (requise) et typer le bean recherché avec une interface (fournie)
- Et si le framework trouve plusieurs candidats à la connexion ?
 - Anticiper qui sera le candidat principal à l'auto-connexion :
`<bean id="frSpellChecker" class="fr.lirmm.marel.di.FrenchSC" primary="false"/>`
Problème : par défaut, c'est true pour tous les beans (mettre false partout, sauf ...)
 - Éliminer des beans de la candidature à l'auto-connexion :
`<bean id="frSpellChecker" class="fr.lirmm.marel.di.FrenchSC" autowire-candidate="false"/>`

Auto-connecter des beans par constructeur

- Même principe que l'auto-connexion par type, sauf que la DI se fait à travers les constructeurs
- Chercher les beans qui vont correspondre aux types des arguments du constructeur du bean à auto-connecter puis utiliser le constructeur pour la DI
- S'il existe plusieurs constructeurs, choisir celui qui a le plus d'arguments satisfaits, sinon le constructeur sans arguments, sinon une exception est levée
- Exemple :

```
<bean id="textEditor" class="fr.lirmm.marel.di.TextEditor"
      autowire="constructor"/>
```

Auto-connecter des beans par « Auto-détection » et Auto-connecter des beans par défaut

- Auto-détecter le mode d'auto-connexion :
 - Auto-connecter d'abord par constructeur puis par type
- Exemple :

```
<bean id="textEditor" class="fr.lirmm.mare1.di.TextEditor"
    autowire="autodetect"/>
```
- Même mode d'auto-connexion pour tous :
 - Demander à Spring d'appliquer le même mode d'auto-connexion partout (pour tous les beans)
 - Exemple :

```
<beans ...
    default-autowire="byType"/>
```

Par défaut, c'est « *none* » (pas d'auto-connexion).

Mélanger l'auto-connexion et les connexions explicites

- Il est possible de surcharger l'auto-connexion en mettant des connexions explicites

- Exemple :

```
<bean id="emailer" class="fr.lirmm.marel.di.Emailer"
      autowire="byType">
  <property name="editor" ref="textEditor"/>
</bean>
```

Ici, la propriété « *addressBook* » est auto-connectée par type et « *editor* » est explicitement connecté au bean qui a comme id « *textEditor* »

- Impossibilité de mélanger l'auto-connexion par constructeur et les connexions explicites de quelques arguments

Auto-connecter des beans grâce aux annotations

- Activer les annotations pour l'auto-connexion :

```
<beans xmlns="http://www.springframework.org/schema/beans"
  ..
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd"
>
<context:annotation-config/>
<!-- Déclarations de beans -->
</beans>
```

- Il est possible ensuite d'utiliser :
 - Les annotations Spring : @Autowired, @Component, ...
 - Les annotations standards de Java EE > 6 : (implémentées dans Spring) : @Inject, @Qualifier, ...

Auto-connecter un bean avec @Autowired

- Appliquer l'annotation à un setter :

@Autowired

```
public void setEditor(Editor e) {this.editor = e;}
```

Ici, l'auto-connexion se fera par type

- L'appliquer sur n'importe quelle méthode :

@Autowired

```
public void voiciUnEditor(Editor e) {this.editor = e;}
```

- L'appliquer sur une propriété directement :

@Autowired

```
private Editor editor;
```

- S'il y a aucun ou plusieurs beans qui peuvent être utilisés dans l'auto-connexion, une exception est levée

Définir des auto-connexions optionnelles

- L'annotation `@Autowired` impose un contrat fort : auto-connexion obligatoire
- Possibilité d'avoir une exception : `NoSuchBeanDefinitionException` (c'est mieux qu'un `NullPointerException` plus loin dans l'exécution !)
- Définir une interface requise optionnelle d'un composant :
`@Autowired(required=false)`
`private SpellChecker spellChecker;`

Si aucun spell checker n'est trouvé, cette propriété reste = null
(Avec un peu de chance, on aurait prévu dans notre code l'édition de texte sans spell checking : `if(spellChecker != null) ...`)

Préciser (qualifier) les dépendances ambiguës

- Si plusieurs beans peuvent être connectés à un bean, on peut préciser à Spring lequel utiliser grâce à l'annotation `@Qualifier`

- Exemple :

```
@Autowired
```

```
@Qualifier("frSpellChecker")
```

```
private SpellChecker spellChecker;
```

Switcher de l'auto-connexion « par type » à l'auto-connexion « par nom »

- Nous pouvons aller plus loin dans la qualification des beans :

- Déclarer le bean :

```
<bean id="frSpellChecker" class="fr.lirmm.marel.di.FrenchSC">
```

```
  <qualifier value="latinChecker"/>
```

```
</bean>
```

- Déclarer l'auto-connexion qualifiée :

```
@Autowired
```

```
@Qualifier("latinChecker")
```

```
private SpellChecker spellChecker;
```

Créer des *Qualifier* personnalisés

- Spécifier son propre type d'annotation :

```
package fr.lirmm.marel.di;  
import java.lang.annotation.*;  
import org.springframework.beans.factory.annotation.Qualifier;  
@Target({ElementType.FIELD, ElementType.PARAMETER,  
        ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Qualifier  
public @interface LatinChecker { }
```

- L'utiliser pour qualifier des auto-connexions :

```
@Autowired  
@LatinChecker  
private SpellChecker spellChecker;
```

- Possibilité d'utiliser plusieurs « qualifieurs » :

```
@Autowired  
@LatinChecker  
@LexicalAndSyntacticChecker  
private SpellChecker spellChecker;
```

Appliquer les auto-connexions standards

- Utiliser les annotations spécifiées par JAVA EE (JSR330)
- Ces annotations sont implémentées par Spring, mais aussi par d'autres frameworks de DI (comme Guice) : unification et standardisation de la DI entre ces frameworks
- Exemple :
@Inject
`private SpellChecker spellChecker;`
- Pas de membre « required » dans cette annotation
- Qualifier une auto-connexion :
@Inject
@Named("frSpellChecker")
`private SpellChecker spellChecker;`
Ici, auto-connexion par nom directement
- Possibilité de définir des *qualifiers* personnalisés : `javax.inject.Qualifier`

Utiliser des expressions SpEL dans les annotations

- Il est possible d'utiliser les annotations Spring pour « connecter » des valeurs calculées dynamiquement :

- Exemple :

```
@Value("dony@lirmm.fr")  
private String destinataire;
```

- Pas intéressant pour des valeurs simples comme ci-dessous (à mettre directement dans le constructeur)

- Utiliser des expressions SpEL :

```
@Value("#{addressBook.DefaultContact}")  
private String destinataire;
```

Auto-découvrir des beans

- Déclarer à Spring quels beans il doit charger automatiquement

- Pas besoin de déclarer les beans avec la balise `<bean>` :

```
<beans xmlns="http://www.springframework.org/schema/beans"
    ...>
    <context:component-scan
        base-package="fr.lirmm.marel.di"
    />
</beans>
```

Ici, tout le contenu du package et ses sous-packages est scanné pour chercher les classes qui sont à charger en tant que beans

Marquer les beans à charger automatiquement

- Marquer une classe avec @Component :

```
package fr.lirmm.marel.di;  
import org.springframework.stereotype.Component;  
@Component  
public class Emitter ...
```

La classe est alors chargée en tant que bean

- Marquer une classe avec un identifiant :

```
package fr.lirmm.marel.di;  
import org.springframework.stereotype.Component;  
@Component("emailer")  
public class Emitter ...
```

Restreindre la découverte automatique de composants

- Il est possible de restreindre la recherche de beans à charger :

- Inclure explicitement les beans à charger :

```
<context:component-scan
  base-package="fr.lirmm.marel.di">
  <context:include-filter
    type="assignable"
    expression="fr.lirmm.marel.di.Emailer"
  />
</context:component-scan>
```

Toutes les classes qui sont affectables au type Emailer

- Exclure explicitement les beans à charger :

```
<context:component-scan
  base-package="fr.lirmm.marel.di">
  <context:exclude-filter
    type="annotation"
    expression="fr.lirmm.marel.di.annotations.SkipIt"
  />
</context:component-scan>
```


Décrire une architecture en programmant

- Pour les fans du « tout-programmer »
- Le fait d'avoir écrit <context:component scan> permet de charger des beans (classes annotées), mais aussi les classes annotées @Configuration
- Définir une classe décrivant l'architecture et l'annoter @Configuration :

```
package fr.lirmm.marel.di;  
import org.springframework.context.annotation.Configuration;  
@Configuration  
public class EMailerConfig {  
    // Mettre ici les méthodes de déclaration de beans  
}
```

Déclarer un bean et le connecter

- Déclarer un simple bean :

@Bean

```
public SpellChecker frSpellChecker() {  
    return new FrenchSC();  
}
```

Le nom de la méthode correspond à l'id du bean chargé

- Connexion entre beans :

@Bean

```
public Editor textEditor() {  
    return new TextEditor(frSpellChecker());  
}
```

L'appel à la méthode frSpellChecker() ne crée pas à chaque fois une nouvelle instance (Spring va chercher le bean s'il est déjà chargé, si scope = Singleton)

Avantage de la description d'architecture en programme

- Détecter des erreurs de connexion entre composants au plus tôt (impossibilité de détecter les erreurs sur du XML) :
 - Les id des beans par exemple sont des chaînes de caractères en XML qui ne sont pas vérifiées statiquement (à la compil.)
 - Dans le programme précédent, le type et l'id des beans sont exprimés dans la signature d'une méthode
 - Nous pouvons donc, dans ce dernier cas, faire des vérifications à la compilation (savoir si dans les connexions, on utilise des noms de beans qui existent)

Avantages et limites de la DI dans Spring

- Avantages de l'utilisation de ce type de frameworks :
 - Un développeur n'a pas à définir un objet (Factory) qui retourne les instances créés et liés entre elles (tout est fait par le framework)
 - Plusieurs modes de DI auto. : par constructeur, par setter, ...
 - Le framework s'occupe de l'injection virale des dépendances : les dépendances sont propagées dans toutes les instances
- Limites (pour la programmation par composants) :
 - Aucune obligation à utiliser des interfaces requises et fournies
 - Pas d'interface matching (sans sous-typage)

Quelques références

- **Modular Java: Creating Flexible Applications with OSGi and Spring.** Craig Walls. The Pragmatic Programmers, 2009.
- **Dependency Injection: Design Patterns Using Spring and Guice.** Dhanji R. Prasanna. Manning Editions, 2009.
- **Spring in Action, 3rd edition.** Craig Walls. Manning Publications Co, 2011.

Questions

