

# Spring et Injection de dépendances

Serge Rosmorduc  
CNAM - GLG 203

# Code associé à ce cours

Disponible sur

[https://gitlab.cnam.fr/gitlab/glg203\\_204\\_demos/03\\_spring\\_intro.git](https://gitlab.cnam.fr/gitlab/glg203_204_demos/03_spring_intro.git)

**git clone [https://gitlab.cnam.fr/gitlab/glg203\\_204\\_demos/03\\_spring\\_intro.git](https://gitlab.cnam.fr/gitlab/glg203_204_demos/03_spring_intro.git)**  
pour le récupérer.

# Assemblage des applications

- Une application: des objets de différentes couches reliés entre eux...
- On souhaite minimiser le couplage entre les couches...

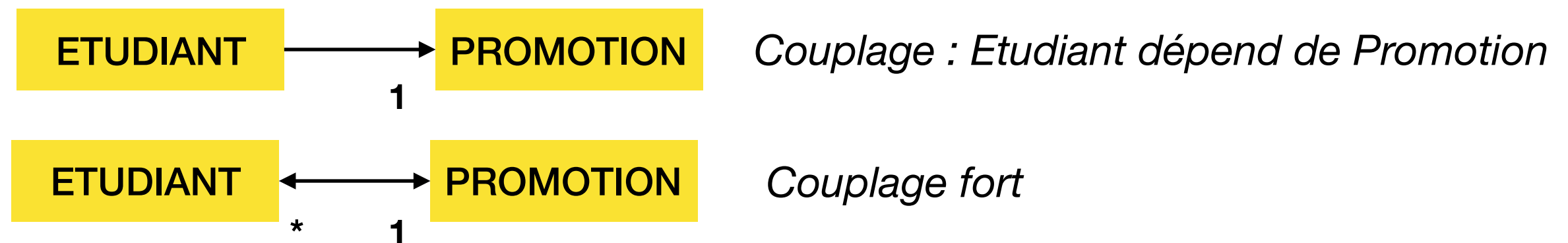
```
class ControleFacture {  
    private FactureUi factureUi;  
  
    public void sauverFacture() {  
        Facture f = factureUi.getData();  
        DbConnection db = new DbConnection("jdbc:mysql:localhost/db",  
            "root", "secretPassword");  
        FactureService service = new FactureService(db);  
        service.sauver(f);  
    }  
}
```

Ouille!

Notre couche UI est couplée avec le type de bd!

# Le couplage

- La classe A est couplée à la classe B si le nom de la classe B apparaît dans le code de la classe A (A a besoin de B pour compiler)
- Le couplage est fort s'il existe un cycle entre A et B (A dépend directement ou indirectement de B, et B dépend directement ou indirectement de A).



# Pourquoi minimiser le couplage?

- Diviser pour régner: plutôt beaucoup de classes simples que peu de classes complexes
- facilité de tests **unitaires** : la classe ControleFacture n'est pas testable sans la BD
- possibilité de modifier simplement l'architecture (par exemple la technologie de sauvegarde des objets)

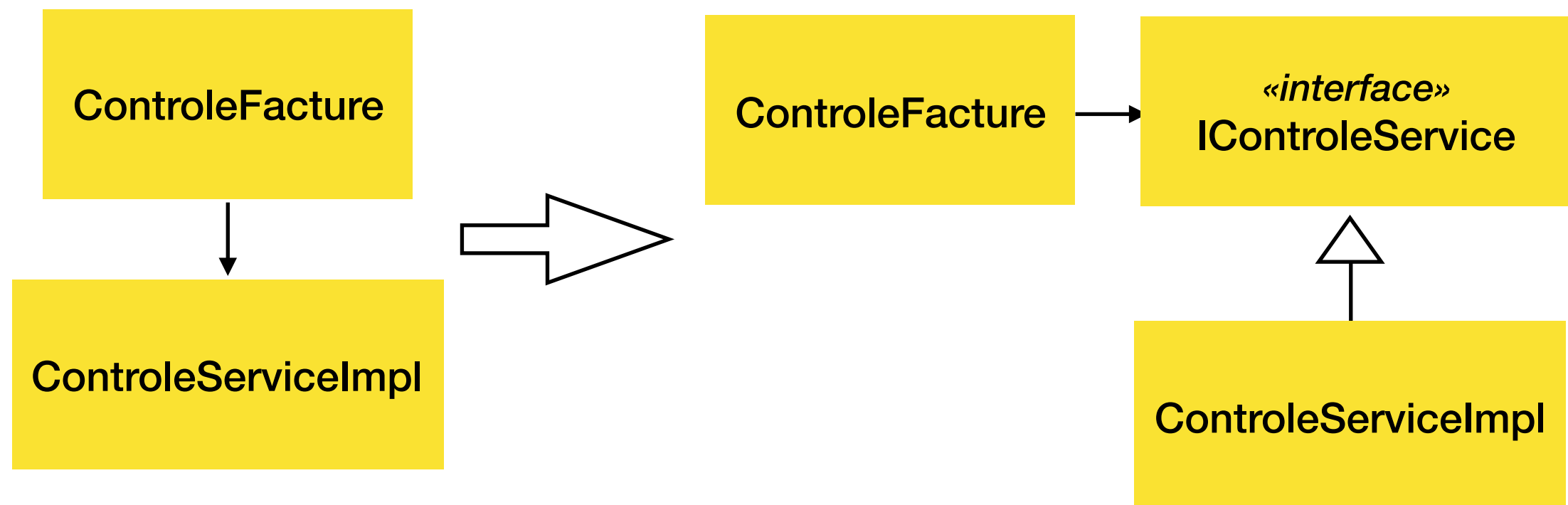
# Le "D" de SOLID

- **Dependency Inversion Principle**

- A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
- B. Abstractions should not depend upon details. Details should depend upon abstractions.

# Une solution...

- Abstraire les fonctionnalités utiles à une classe : *en faire des interfaces.*



# Une solution...

- Abstraire les fonctionnalités utiles à une classe : *en faire des interfaces.*

```
class ControleFacture {  
    private IFactureService service;  
    private FactureUi factureUi;  
  
    public void sauverFacture() {  
        Facture f = factureUi.getData();  
        service.sauver(f);  
    }  
}
```

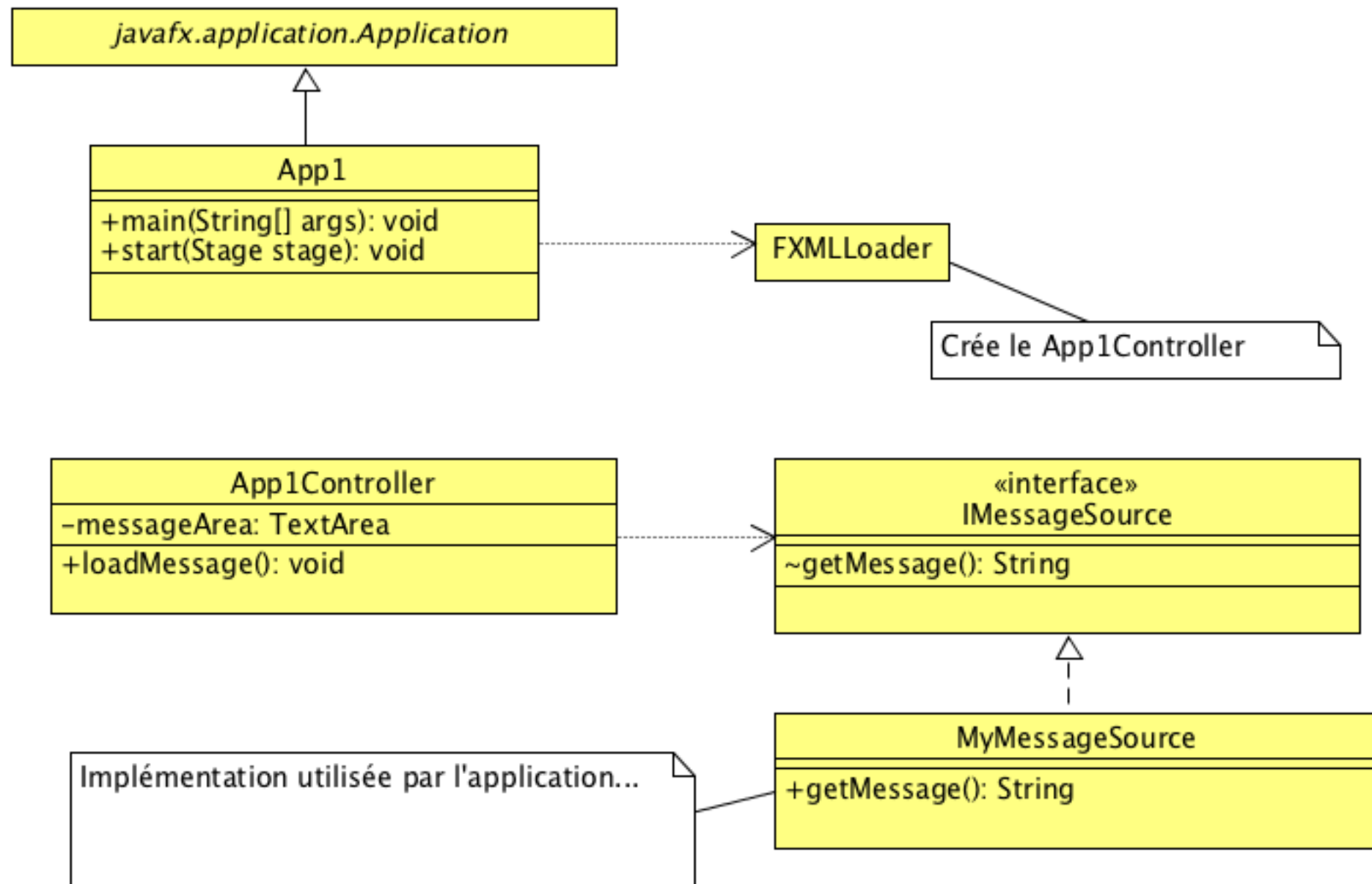


# Une solution...

- Abstraire les fonctionnalités utiles à une classe : *en faire des interfaces*.
- Reste le problème de savoir comment implémenter ces interfaces!

```
class ControleFacture {  
    private IFactureService service;  
    private FactureUi factureUi;  
  
    public void sauverFacture() {  
        Facture f = factureUi.getData();  
        service.sauver(f);  
    }  
}
```

# Autre exemple: Application JavaFx



# Autre exemple: configuration en JavaFx

- En JavaFX, les Contrôleurs sont naturellement créés par les vues quand on utilise FXML
- Du coup, il est difficile de leur passer des éléments de configuration...

```
<GridPane hgap="10.0" vgap="10.0"  
  xmlns="http://javafx.com/javafx/8.0.171"  
  xmlns:fx="http://javafx.com/fxml/1"  
  fx:controller="glg203.demoJFX.demo1.App1Controller">  
...  
</GridPane>
```

créé par constructeur par défaut ?

# Caractéristiques des ressources

- Pour les implémentations des ressources :
  - on ne veut pas dépendre de leur implémentation
  - on veut généralement que ce soient des singletons (pas forcément au sens pattern). Les données sont partagées
  - Exemple typique : connexion base de données.

# Utilisation du pattern singleton?

- Simplifie le code... dans un premier temps.
- Ne résout pas le problème du couplage.

```
class ControleFacture {  
    private FactureUi factureUi;  
  
    public void sauverFacture() {  
        Facture f = factureUi.getData();  
        IFactureService service = FactureServiceImpl.getInstance();  
        service.sauver(f);  
    }  
}
```

...

# Utilisation d'un annuaire

- Annuaire (Directory): une classe qui permet d'enregistrer des implémentations d'objets
- Les objets sont généralement identifiés par un nom (une String), plus ou moins formalisée
- Exemple: JNDI
- *Le test reste difficile*

```
class ControleFacture {  
  
    public void sauverFacture() {  
        Facture f = factureUi.getData();  
        Context context = new InitialContext();  
        FactureService service =  
            (FactureService)context.lookup("FService");  
        service.sauver(f);  
    }  
}
```

# Injection de dépendances

- Hollywood Principle: « *Don't call us, we'll call you* ».
- Les dépendances sont de bêtes variables d'instances
- Leurs implémentations sont passées, soit au constructeur, soit à des setters

# Constructeur...

```
class ControleFacture {  
    private FactureUi factureUi;  
    private IFactureService service;  
  
    public ControleFacture(IFactureService service, FactureUi ui) {  
        this.service = service;  
        this.ui = ui;  
    }  
  
    public void sauverFacture() {  
        Facture f = factureUi.getData();  
        service.sauver(f);  
    }  
}
```



# Setter...

```
class ControleFacture {  
    private FactureUi factureUi;  
    private IFactureService service;  
  
    public ControleFacture(FactureUi factureUi) {  
        this.factureUi = ui;  
    }  
  
    public void setFactureService(IFactureService service) {  
        this.service = service;  
    }  
}
```

- L'objet n'est pas complètement initialisé après l'appel du constructeur...
- invariants non garantis ?

# Mais qui crée les objets?

- Il faut un chef d'orchestre pour monter l'application...
- peu pratique à écrire « à la main »...
- d'où l'utilisation d'un injecteur de dépendance
- **Spring**, Guice, J2EE (en partie), spécifications JSR 330 (Java Specification Requests 330)

# Spring Framework

- Rod Johnson : *Expert One-on-One J2EE Design and Development* (2002)
- réaction aux lourdeurs de JEE (à l'époque)
- permet l'utilisation de POJO: « plain old java objects » ; objets java « normaux » non liés à un framework
- au départ: configuration en XML
- de plus en plus: configuration par annotations, et par conventions de nommage.

# Spring

- framework applicatif « léger » : environnement pour l'exécution des programmes ;
- « léger » : peu de contraintes sur les classes qu'on écrit ;
- cœur de Spring: injection de dépendances et Aspect Oriented Programming;
- accent mis sur la testabilité ;
- nombreuses autres fonctionnalités fournies: accès aux bases de données, web, sécurité, messagerie intergicielle, Cloud...

# Aspect Oriented Programming

- Permet d'ajouter des traitement en amont et en aval des appels de méthodes
- Approche sophistiquée du pattern Decorator
- peut utiliser la classe Proxy de java
- ou des bibliothèques de manipulation de bytecode
- on en reparle à la fin du semestre

# Les POJO

- Spring encourage l'utilisation de « POJO »
- « Plain Old Java Objects »
- Objets non liés à un framework ou à une bibliothèque
- Attention, ce ne sont pas *a priori* des objets anémiques (avec juste des getters et des setters). Ils ont de « vraies » méthodes
- Voir <https://martinfowler.com/bliki/POJO.html>

# SpringBoot

- Assembler de nombreux composants divers n'est pas simple.
- Problèmes de compatibilités entre les versions des bibliothèques, etc...
- Mise en place d'une infrastructure assez lourde
- → Spring boot
  - bibliothèques pré-configurées embarquant les dépendances
  - plus outil d'initialisation <https://start.spring.io>

# Spring boot

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- ***Automatically configure Spring and 3rd party libraries whenever possible***
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration



# Une petite appli web en 5 minutes...

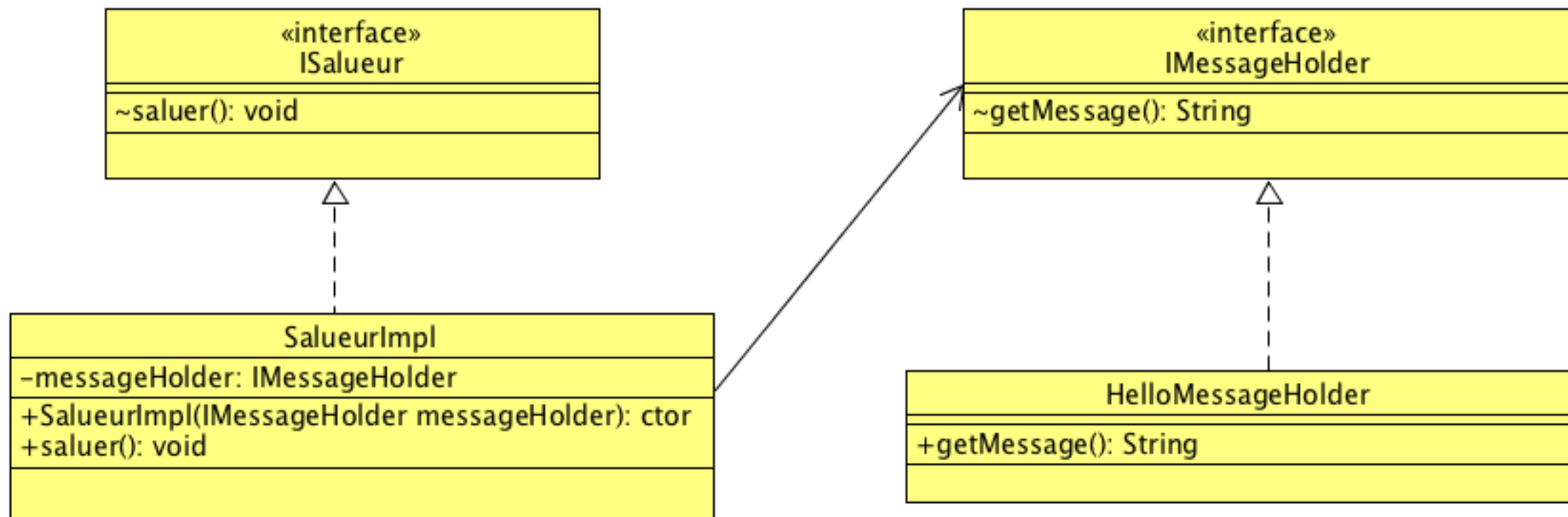
- Initialisation:
- `spring init -d=web --build=gradle simple-web`
- Ajout de la gestion de servlets : `@ServletComponentScan` dans le main
- Création d'un Service (interface et implémentation)
- injection de celui-ci dans une servlet simple...

# Bases de Spring

*(Projet: 02\_spring-basic-app)*

- Les objets créés par Spring sont appelés des **beans** (terminologie java - cf. javabeans) ;
- Création/gestion des beans: objet **ApplicationContext**
- gère le **cycle de vie** des beans
- plusieurs type d'applicationContext: annotations java, xml, groovy
- Les bean forme généralement l'infrastructure de l'application. Le modèle lui-même n'est pas géré par Spring

# Petit Exemple...



# Déclaration de beans en XML

- Le main :

```
public static void main(String[] args) {  
    // ApplicationContext ne permet pas de fermer la ressource...  
    try (ClassPathXmlApplicationContext ctx =  
        new ClassPathXmlApplicationContext(  
            "conf/config.xml")) {  
        ISaleur saleur= (ISaleur)ctx.getBean("saleur");  
        saleur.saler();  
    }  
}
```

# Déclarations 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
                           https://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="salueur" class="glg203.cours03.app00Xml.SalueurImpl">
    <constructor-arg ref="messageHolder"/>
  </bean>
  <bean id="messageHolder"
        class="glg203.cours03.app00Xml.HelloMessageHolder">
    <constructor-arg type="java.lang.String"
                     value="Bonjour depuis un bean XML"/>
  </bean>
</beans>
```

# Les classes

```
public class SalueurImpl implements ISalueur{
    private final IMessageHolder messageHolder;
    public SalueurImpl(IMessageHolder messageHolder) {
        this.messageHolder = messageHolder;
    }
    @Override
    public void saluer() {
        System.out.print("Voici un message : ");
        System.out.println(messageHolder.getMessage());
    }
}
```

Aucune dépendance à Spring... pojo...

# Les classes...

```
<bean id="messageHolder"  
      class="glg203.cours03.app00Xml.HelloMessageHolder">  
    <constructor-arg type="java.lang.String"  
                    value="Bonjour depuis un bean XML"/>  
</bean>
```

```
public class HelloMessageHolder implements IMessageHolder {  
    private final String message;  
    public HelloMessageHolder(String message) {  
        this.message = message;  
    }  
    @Override  
    public String getMessage() {  
        return message;  
    }  
}
```

# La balise bean

- Définit un bean
- Attributs
  - id : le nom (unique) du bean. Permet d'y faire référence.
  - class : la classe Java du bean
- Le bean est initialisé à travers un de ses constructeurs et/ou des setters.



# constructor-arg

- Permet de passer une valeur à un constructeur
- attributs:
  - value : valeur de l'argument (type primitif)
  - ref: référence à un autre bean.
  - type: (optionnel) de l'argument
  - index: position de l'argument

# property

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">  
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>  
  <property name="username" value="root"/>  
  <property name="password" value="masterkaoli"/>  
</bean>
```

- Permet d'initialiser une propriété en passant par un « setter ».
- attributs:
  - name: nom de la propriété
  - value, type, ref...

# @Lazy

- Déclare qu'un bean ne sera créé que quand on en aura vraiment besoin (allocation paresseuse ou tardive)

# Valeurs complexes

- On peut spécifier la valeur d'un argument à l'intérieur de la balise property ou constructor-arg
- On peut construire des collections si nécessaire...

```
<bean id="messageHolder"  
      class="glg203.cours03.app00Xml.ListMessageHolder">  
  <constructor-arg>  
    <list>  
      <value>premier message</value>  
      <value>second message</value>  
      <value>troisième message</value>  
    </list>  
  </constructor-arg>  
</bean>
```

# Cycle de vie du bean...

- On peut lier des appels de méthodes à des étapes dans le cycle de vie d'un bean

```
<bean id="myDataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/mydb" />
  <property name="username" value="root" />
  <property name="password" value="masterkaoli" />
</bean>
```

# Déclaration de beans en Java

- Alternative au XML
- Pas forcément plus invasif - ne touche pas forcément aux classes des beans
- Le principe: des factory method créent les beans, elles sont **annotées** pour cela
- Le système les trouve et les utilise

# Injection en java

```
public static void main(String[] args) {  
    // ApplicationContext ne permet pas de fermer la ressource...  
    try (AnnotationConfigApplicationContext ctx =  
        new AnnotationConfigApplicationContext(MyConfig.class)) {  
  
        ISaleur messenger = ctx.getBean(ISaleur.class);  
        messenger.saler();  
        // par nom...  
        ISaleur messenger1 = (ISaleur) ctx.getBean("saleur");  
        messenger1.saler();  
    }  
}
```

# Classe de configuration

```
@Configuration
public class MyConfig {
    @Bean
    public IMessageHolder messageHolder() {
        return new HelloMessageHolder(
            "depuis une application configurée par annotations");
    }

    @Bean
    public ISalueur salueur() {
        return new SalueurImpl(messageHolder());
        // IMPORTANT : on appelle la méthode annotée @Bean, on
        // ne fait pas un new explicitement sur l'argument.
    }
}
```



# Les beans sont des singletons par défaut

Contrairement à ce que laisse imaginer une ligne comme

```
return new SalueurImpl(messageHolder());
```

chaque bean est un singleton. **Il est créé une seule fois.**

Si plusieurs autres beans utilisent messageHolder(), un seul objet IMessageHolder sera créé.

# Recherche automatique de composants

- On lie les classes à Spring (annotation dans les classes)
- Une classe annotée par **@Component** définit un bean
- On déclenche la recherche des composants en annotant la configuration avec **@ComponentScan**

```
@Configuration  
@ComponentScan  
public class MyConfig {}
```

peut (ou non) être vide !

```
@Component  
public class SalueurImpl implements ISalueur{  
    private final IMessageHolder messageHolder;  
    ...  
}
```

# Injection de dépendances

- On peut annoter les classes des beans avec `@Autowired` (c'est intrusif!) pour spécifier qu'on veut injecter tel ou tel bean à un endroit donné
- L'Application context se chargera alors de fournir le bean
- On peut annoter:
  - une variable d'instance
  - un setter
  - un constructeur
- ça n'est pas toujours nécessaire pour les constructeurs.

# @Autowired

```
@Component
public class EtudiantDao {
    @Autowired
    private DataSource
        connection;

    . . . .
}
```

```
@Component
public class EtudiantDao {
    private DataSource connection;

    @Autowired
    public EtudiantDao(DataSource connection) {
        this.connection = connection;
    }
}
```

```
@Component
public class EtudiantDao {
    private DataSource connection;

    @Autowired
    public void setConnection(DataSource connection) {
        this.connection = connection;
    }
}
```

# @Autowired

- required

# @Autowired et interfaces

- Attention, @Autowired va chercher des beans compatibles en assignement avec la propriété concernée.

```
@Autowired IMessageHolder messageHolder
```

- cherche un bean d'une classe qui étend l'interface IMessageHolder
- possibles ambiguïtés

# Levée d'ambiguïtés

- On peut nommer les beans
  - Dans la configuration java: **@Bean(name="nom")**
  - Dans l'annotation de la classe **@Component("nom")**
- On peut utiliser l'annotation @Qualifier pour lever une ambiguïté:

```
@Autowired  
@Qualifier("userSource")  
DataSource dataSource;
```

# Levée d'ambiguïtés

- @Primary : marque un composant comme prioritaire s'il y a un choix
- Injection dans une liste : si la cible d'Autowired est une collection, elle réunira tous les composants candidats !



# Guider la recherche

- @ComponentScan peut prendre les formes suivantes:

@ComponentScan(basePackages = {"glg203.cours.comp"})

- recherche récursivement dans les packages nommés

@ComponentScan(basePackageClasses = {  
glg203.cours.comp.App.class})

- recherche dans les à partir des packages contenant les classes passées dans la liste.

# Importer des configurations

- Importer explicitement une configuration depuis une autre (pour découper la configuration par couche de l'application) :
  - `@Import(AutreClasseDeConfig.class)`
- Sinon, les configurations sont aussi trouvées et utilisées par `@ComponentScan`

# Injection de Properties

- Classe Properties de java ;
- Couples clefs/valeurs (String)
- Pratique pour passer des informations de configuration au programme
- `@PropertySource("classpath:foo.properties")`
- Injection de valeur:
- `@Value("${db.url}")` : injection de la propriété (\$ = propriétés)
- `@Value("#{systemProperties['user.home']}")` : '#' : SpEl, plus complet

# Que choisir

- Déclaration XML : la moins intrusive, mais lourde ;
- Déclaration java :
  - la saisie se fait dans l'environnement java. Sécurité du typage ;
  - Plus léger à manipuler ;
- Autocablage : simple et agréable. **Couple le code java à Spring.**
  - mais : annotations J2EE possible : fonctionne avec d'autres framework
  - ne concerne (normalement) qu'une petite partie du code.
  - garder tout ça testable...

# Et SpringBoot ?

- Simplifie la configuration de base...
- Et pré-configurer les composants standards selon les bibliothèques de Springboot installées par l'application...

```
@SpringBootApplication
public class DvdApplication {
    public static void main(String[] args) {
        SpringApplication.run(DvdApplication.class, args);
    }
}
```

# SpringBoot

- Dans le fichier gradle...

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'  
    developmentOnly 'org.springframework.boot:spring-boot-devtools'  
    runtimeOnly 'com.h2database:h2'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```

- Intègre : le web, avec un serveur tomcat intégré, et une visualisation avec thyme leaf, plus une couche d'accès au données jpa, et une base h2
- Pas de configuration supplémentaire : met tout en place automatiquement !

# SpringBoot

- Une bonne partie de la configuration peut être modifiée en utilisant le fichier application.properties (intégré dans les ressources)

```
logging.level.web=DEBUG
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
spring.datasource.url=jdbc:h2:/tmp/demoDB
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
```

Remplace la base en mémoire par une base sur disque

# Portée des beans Spring

- singleton (par défaut) : un seul bean est créé pour une déclaration donnée
- prototype : le bean est créé à chaque demande.
- request et session: portées utilisées pour le Web (plus tard)



# Bean prototypes

- Assez peu utilisés ;
- annotation `@Scope("prototype")`
- La méthode de création du bean renvoie *un nouveau bean à chaque injection ou demande*.
- Intérêt éventuel (par rapport à new !) : ce bean bénéficie de l'injection de dépendances
- Utilisation possible: injecter l'ApplicationContext dans un bean client, et utiliser celle-ci pour créer les beans prototypes.

# Cycle de vie des beans

- Un bean:
  - est créé par le conteneur avec `new`, et initialisé avec l'un de ses constructeurs ;
  - l'injection de dépendances par setter est réalisée ;
  - les méthodes annotées par **@PostConstruct** sont exécutées
  - ...
  - L'application ferme (ou décide de supprimer le bean)
  - les méthodes annotées par **@PreDestroy** sont exécutées
  - le bean est passé au GC.

# Et avec @Bean ?

- Si on crée un bean avec @Bean, et qu'on ne veut pas annoter l'objet directement:

`@Bean(initMethod = "afterPropertiesSet", destroyMethod = "destroy")`

# Cycle de vie des beans

- **Important !!!** Si des variables d'instances sont injectées, elle ne sont pas initialisées au moment où le constructeur est appelé.
- Du coup, on ne peut pas les utiliser pour l'initialisation d'autres données
- exemple: liaison BD utilisée pour lire des données fixes ?
- la solution est d'attendre l'appel de la méthode `@PostConstruct`
- ... ou d'utiliser l'initialisation par constructeur.

# Composants spécifiques

- Annotations alternatives à @Component :
  - @Service : opérations sans états (DDD)
  - @Repository : généralisation de la notion de DAO (DDD)
  - @Controller : contrôleur dans Spring MVC (Web)

# Annotations JSR 330

- <https://docs.oracle.com/javaee/6/tutorial/doc/giwhb.html>
- Standardisées : fonctionne a priori dans d'autres containers (mais il y a des chances qu'on utilise d'autres fonctionnalités de Spring...)
- @Inject : équivalent de @Autowired
- @Named :
  - sur une classe : la marque comme un composant nommé
  - sur un champ injecté : permet de sélectionner le composant injecté par nom.

# JSR330 et portée

- Selon la JSR330, les beans sont des équivalents des prototypes de Spring
- Pour avoir l'équivalent d'un Bean Spring, on écrit en théorie `@Named(...)` `@Singleton`
- Mais en Spring, même les beans `@Named` sont par défaut des singletons
- On peut utiliser l'annotation pour la portabilité.

# Tests en Spring

- Tests « normaux » : Spring encourage l'écriture de POJO
- Tests d'intégration : Spring permet d'utiliser l'injection de dépendance dans les tests.



# Test avec injection Spring standard

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestConfig.class} )
public class SalueurImplTest {
    @Autowired
    ISalueur salueur;

    @Test
    public void testSaluer() {
        salueur.saluer();
    }
}
```

Classe pour faire tourner le test

Config a priori différente de celle d'exécution

L'injection fonctionne

# Avec SpringBoot

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class DemoApplicationTests {

    @Autowired
    ICalculeurService calculeurService;

    @Test
    public void contextLoads() {
        Assert.assertNotNull(calculeurService);
    }

    @Test
    public void testCalc() {
        Assert.assertEquals(4.0, calculeurService.doubler(2), 1e-10);
    }
}
```

# Annotations utiles

`@TestPropertySource(locations=« classpath:test.properties »)`

- sur la classe de test
- permet de remplacer `application.properties` pour les tests
- usage: par exemple ne pas utiliser la même base de données

# Annotations utiles

- `@DirtyContext` (sur un test particulier) : le test modifie/détruit des objets importants, et le contexte n'est pas réutilisable pour les tests suivants. Force la recreation du contexte
- `@Transactional` : support de commit et rollback. À peu près automatique avec SpringBoot
- `@Rollback` : demande un rollback après le test

# Remplacement de DB Unit

- Spring peut utiliser l'annotation @SQL, qui lui permet d'exécuter du SQL avant un test:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@TestPropertySource(locations="classpath:test.properties")
@Transactional
public class DvdServiceTest {
    @Autowired
    private DvdService service;
    @Test
    @SqlGroup({
        @Sql(statements = {"delete from dvd"})
    })
    @Rollback
    public void testFindAllEmpty() {
        assertTrue("", service.findAll().isEmpty());
    }
}
```

# Les profils

- Permettent d'avoir plusieurs configurations différentes (production, développement, test...)

```
@Configuration
@Profile("development")
@Import(RamsesMockUserConfig.class)
@ComponentScan(basePackages = {
    "org.genherkhopeshef.ramses.mock",
    "org.genherkhopeshef.ramses.users.service"})
public class DevDBConfig {
```

- Lors d'un scan, le fichier de configuration n'est utilisé que si le profil correspond au profil courant
- @ActiveProfiles : permet de sélectionner les profils actifs

# Fixer les profils

- variable d'environnement `SPRING_PROFILE_ACTIVE`
- `java -Dspring.profiles.active=dev ...`
- définir `spring.profiles.active` dans `application.properties`

# Exemple

- Dans une application, on a défini l'interface MailService, qui sert à prévenir un utilisateur que son compte a été créé ou modifié
- L'implémentation réelle envoie vraiment des mails - pas très pratique pour les tests de l'UI
- On fait une implémentation « bidon » (Dummy/Mock), qui ne fait rien
- La première est définie dans une configuration annotée avec `@Profile("production")`
- La seconde dans une configuration annotée avec `@Profile("development")`
- Les implémentations communes sont importées.



# Ecosystème Spring

- gradle et maven disposent de plugins adaptés
- dans gradle, spring-boot-devtools permet par exemple de relancer le serveur à chaud, d'avoir des informations de configuration, etc...
- spring initializer <https://start.spring.io> permet de créer facilement un squelette de code. Il existe en stand-alone
- springToolSuite : version d'eclipse pour spring
- nombreux plugins pour les IDEs.

# Dependency Injection et Inversion of Control

- On a des concepts un peu différents :
  - ***Dependency Inversion Principle*** : le haut niveau ne devrait pas être couplé au bas niveau
  - ***Inversion of Control*** : « don't call us, we'll call you » : le framework appelle les méthodes de l'utilisateur, pas l'inverse. En fait, caractéristique de ***tous*** les frameworks
  - ***Dependency Injection*** : mécanisme particulier d'inversion de contrôle pour la configuration d'objets, en passant par un « assembleur ».

# Bibliographie

- Les docs de Spring sur <https://spring.io>
- Craig Walls, *Spring in Action*, Manning
- Cosmina et al, Pro Spring 5, Appress
- <https://martinfowler.com/bliki/InversionOfControl.html>
- <https://www.martinfowler.com/articles/injection.html>
- <https://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection>