

Une (petite) introduction à Spring

1 Introduction

Le framework Spring¹ est une boîte à outils très riche permettant de structurer, d'améliorer et de simplifier l'écriture d'application JEE. Spring est organisé en module

- Gestion des instances de classes (JavaBean et/ou métier),
- Programmation orientée Aspect,
- Modèle MVC et outils pour les application WEB,
- Outils pour la DAO (JDBC),
- Outils pour les ORM (Hibernate, iBatis, ...),
- Outils pour les applications JEE (JMX, JMA, JCA, EJB, ...),

Plus d'informations ici².

2 Programmation par contrat

2.1 Principe

La *programmation par contrat* consiste à séparer la spécification d'une couche logicielle (aussi appelée service) de sa réalisation. La spécification donne lieu à la création d'une **interface** et la réalisation fournit une **classe qui implante cette interface**. Ce ne sont pas nécessairement les mêmes personnes qui développent l'interface et son implantation. On peut également remarquer que la phase de réalisation peut produire plusieurs implantations différentes d'une même interface. Le choix entre ces implantations est réalisé à l'intégration entre les couches. Les objectifs de cette approche :

Réduire les dépendances. Les classes d'implantation ne se connaissent pas. Elles dialoguent au moyen des interfaces. De ce fait, on peut facilement changer un implantation contre une autre sans avoir à mettre à jour la totalité du logiciel.

Faciliter les tests. Chaque couche logicielle ayant une spécification claire, il est facile de lui associer un jeu de tests utilisable quelque soit la nature de l'implantation.

Simplifier le code. Dans certains cas de figure, le code d'une méthode est une suite de considérations sans liaison directe entre-elles. La programmation par contrat va faciliter la construction d'implantations façade qui se chargent chacune d'une partie du travail.

Organisation du développement.

2.2 Un exemple

Prenons un exemple pour éclairer ces principes. Nous avons besoin dans nos applications de pouvoir tracer un certain nombre d'évènements. Nous allons donc créer un service de trace (un *logger* en anglais). Ce service est spécifié par l'interface ci-dessous :

¹<http://www.springframework.org/>

²<http://static.springframework.org/spring/docs/3.0.x/spring-framework-reference/html/overview.html>

```

package fr.ssa.log;

public interface ILog {

    void log(String message);

    void log(String message, String arg1);

}

```

2.2.1 Une première implantation

Pour utiliser ce service, nous avons besoin d'une classe qui implante ce service. Il existe plusieurs manières de faire. Nous allons, dans un premier temps, envoyer les messages de trace sur la console de sortie d'erreur standard :

```

package fr.ssa.log.imp;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import fr.ssa.log.ILog;

public class StderrLogger implements ILog {
    // format des dates
    private final DateFormat format = new SimpleDateFormat(
        "yyyy/MM/dd hh:mm:ss | ");

    public void init() {
        System.err.println("Init " + this);
    }

    public void close() {
        System.err.println("Close " + this);
    }

    public void log(String message) {
        System.err.print(format.format(new Date()));
        System.err.println(message);
    }

    public void log(String message, String arg1) {
        log(message.replaceAll("\\$1", arg1));
    }

}

```

Les méthodes `init` et `close` correspondent à la phase d'initialisation et de clôture du service. Nous retrouverons ces méthodes dans toutes les implantations (même si elles sont vides).

A propos des packages. Vous pouvez noter dans cet exemple que l'interface est dans un package et que la classe d'implantation est dans un autre. Dans un cas réel, la spécification d'un service peut être composé de plusieurs interfaces accompagnées de javaBeans ou de classes d'exception. L'implantation de ce service peut également contenir plusieurs classes ce qui justifie clairement l'utilisation de plusieurs packages.

Nous pouvons maintenant utiliser ce service :

```

package fr.ssa.appli;

import fr.ssa.log.ILog;
import fr.ssa.log.imp.StderrLogger;

public class MyApp1 {

    // Utilisation du service "logger"
    void use(ILog logger) {
        logger.log("result = $1", "hello");
    }

    // integration
    void run() {
        StderrLogger logger = new StderrLogger();
        logger.init();
        use(logger);
        logger.close();
    }

    public static void main(String[] args) {
        MyApp1 app = new MyApp1();
        app.run();
    }
}

```

A ce stade, nous pouvons remarquer que :

- Les utilisateurs (la méthode use ci-dessus) n'ont aucune connaissance des détails de l'implantation. Seule l'interface est utilisée.
- Il existe tout de même une dépendance puisque le nom de la classe d'implantation apparaît en clair dans le code de l'utilisateur (méthode run).
- L'intégrateur (methode run) est responsable du respect du contrat (on appelle d'abord init, puis on utilise, puis on appelle close).

2.2.2 Une deuxième implantation

Nous pouvons aussi donner une deuxième implantation qui stocke les traces dans un fichier :

```

package fr.ssa.log.imp;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import fr.ssa.log.ILog;

public class FileLogger implements ILog {

    // format de sortie des dates
    private final DateFormat format = new SimpleDateFormat(
        "yyyy/MM/dd hh:mm:ss | ");

    // fichier de sortie
    private final PrintWriter file;

    public FileLogger(String fileName) {
        try {
            this.file = new PrintWriter(new FileOutputStream(fileName, true));
        } catch (FileNotFoundException e) {
            throw new IllegalArgumentException("bad fileName");
        }
    }

    public void init() {
        System.err.println("Init " + this);
    }

    public void close() {
        file.close();
        System.err.println("Close " + this);
    }

    public void log(String message) {
        file.print(format.format(new Date()));
        file.println(message);
    }

    public void log(String message, String arg1) {
        log(message.replaceAll("\\$1", arg1));
    }

}

```

Cette nouvelle implantation a absolument besoin d'un paramètre (le nom du fichier) pour être fonctionnelle. La solution retenue est la plus simple : ajouter un argument au constructeur. Nous remarquons que de ce fait, la méthode `init` n'a plus vraiment d'intérêt.

On remarque que le code d'intégration est très peu modifié (une ligne), alors que le fonctionnement est totalement différent. On remarque également que le code d'utilisation (méthode `use`) n'est pas altéré. Les modifications portent uniquement sur le code d'intégration :

```
// integration de l'application
void run() {
    FileLogger logger = new FileLogger("/tmp/myapp.log");
    logger.init();
    use(logger);
    logger.close();
}
```

2.2.3 Une troisième implantation

La plupart des classes d'implantation ont besoin de paramètres pour assurer leur service. Le choix de placer ces paramètres en argument du constructeur pose plusieurs problèmes :

- La classe obtenue **n'est pas un javaBean** (pas de constructeur vide). C'est particulièrement gênant car l'intérêt de ces composants élémentaires est très important.
- Les paramètres du service sont fixés à sa création (par le constructeur). Il n'est donc **pas possible de les changer en cours de route**, voir même d'envisager un recyclage du service (changement des paramètres et nouvelle initialisation).
- Si nous avons **beaucoup de paramètres**, le constructeur est difficile à utiliser.
- Il n'est pas possible de prévoir des **valeurs par défaut** pour certains paramètres.

Nous allons donc introduire une nouvelle solution au problème des paramètres : les paramètres vont être codés comme des propriétés de la classe d'implantation et la méthode `init` devra les utiliser pour initialiser le service. Nous obtenons donc cette nouvelle version :

```

package fr.ssa.log.imp;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import fr.ssa.log.ILog;

public class BeanFileLogger implements ILog {

    // format des dates
    private final DateFormat format = new SimpleDateFormat(
        "yyyy/MM/dd hh:mm:ss | ");

    // parametre du nom de fichier
    private String fileName;

    private PrintWriter file;

    // initialisation du service
    public void init() {
        if (fileName == null) {
            throw new IllegalStateException("no fileName");
        }
        try {
            OutputStream os = new FileOutputStream(fileName, true);
            this.file = new PrintWriter(os);
        } catch (FileNotFoundException e) {
            throw new IllegalStateException("bad fileName");
        }
    }

    // fermeture du service
    public void close() {
        file.close();
    }

    public void log(String message) {
        file.print(format.format(new Date()));
        file.println(message);
    }

    public void log(String message, String arg1) {
        log(message.replaceAll("\\$1", arg1));
    }

    public String getFileName() {
        return fileName;
    }

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }
}

```

Le code d'intégration a maintenant la responsabilité de fixer les paramètres du service avant d'appeler la méthode d'initialisation. Cette solution est plus simple et plus systématique quand le nombre de paramètres est important :

```

// integration de l'application
void run() {
    BeanFileLogger logger = new BeanFileLogger();
    logger.setFileName("/tmp/myapp.log");
    logger.init();
    use(logger);
    logger.close();
}

```

3 Injection des dépendances

L'**injection des dépendances** traite le délicat problème de la communication et de la dépendance entre services logiciels. Prenons l'exemple d'une classe métier :

```

package fr.ssa.bus;

public interface IBusiness {

    int doAddition(int a, int b);

    boolean doLogin(String name, String pass);
}

```

Construisons maintenant une implantation de ce service qui génère une trace après chaque appel d'une méthode métier. Cette implantation a donc besoin d'une couche *logger* pour s'exécuter correctement. Nous pourrions envisager de placer dans cette implantation la variable de classe suivante :

```

package fr.ssa.bus.imp;

import fr.ssa.bus.IBusiness;
import fr.ssa.log.ILog;

public class SimpleBusiness implements IBusiness {

    private ILog logger = new fr.ssa.log.imp.StderrLogger();

    ...
}

```

Cette solution pose deux problèmes :

1. Une dépendance directe vient d'être introduite entre cette implantation du service métier et une implantation particulière de la couche *logger*. Cette dépendance est regrettable car inutile. La couche métier doit utiliser l'interface *ILog* et pas une implantation.
2. Si nous avons choisi une couche de trace ayant besoin d'un paramètre (comme celle vue précédemment), nous aurions sans doute dû inclure ce paramètre (le fichier de sortie) comme un paramètre de la couche métier. En d'autres termes, les paramètres d'une couche *A* doivent inclure tous les paramètres des couches utilisées par *A*.

Pour éviter ces problèmes, nous allons simplement introduire dans l'implantation de la couche métier un paramètre faisant référence à une implantation de la couche *logger*. De ce fait, les deux implantations resteront indépendantes l'une de l'autre. Le seul point de contact sera l'interface *ILog* :

```

package fr.ssa.bus.imp;

import fr.ssa.bus.IBusiness;
import fr.ssa.log.ILog;

public class SimpleBusiness implements IBusiness {

    private ILog logger;

    public void init() {
        if (logger == null) {
            throw new IllegalStateException("null logger");
        }
        logger.log("Start " + this);
    }

    public void close() {
        logger.log("Stop " + this);
    }

    public int doAddition(int a, int b) {
        logger.log("doAddition(" + a + ", " + b + ")");
        return (a + b);
    }

    public boolean doLogin(String name, String pass) {
        logger.log("doLogin(\"" + name + "\",\"" + pass + "\")");
        return (name.equals(pass));
    }

    public ILog getLogger() {
        return logger;
    }

    public void setLogger(ILog logger) {
        this.logger = logger;
    }

}

```

La phase d'intégration devient plus délicate puisqu'il faut créer et initialiser deux couches logicielles :

```

void run() {
    // creation de la couche logger
    StderrLogger logger = new StderrLogger();
    logger.init();
    // creation de la couche metier
    SimpleBusiness business = new SimpleBusiness();
    business.setLogger(logger);
    business.init();
    // utiliser la couche metier
    use(business);
    // fermer la couche metier
    business.close();
    // fermer la couche logger
    logger.close();
}

```

Nous pouvons très facilement et **sans modifier la couche métier** changer la politique de trace en utilisant un fichier. Il suffit de changer les quatre premières lignes du code d'intégration :

```
// creation de la couche logger
BeanFileLogger logger = new BeanFileLogger();
logger.setFileName("/tmp/myapp.log");
logger.init();
```

Nous venons de mettre en oeuvre le principe de *l'injection de dépendances*. C'est la partie intégration qui se charge d'injecter dans la couche métier la référence vers la couche *logger*. Initialiser une application revient à créer les couches logicielles, injecter les dépendances et appeler les méthodes d'initialisation.

4 Gestion des beans par Spring

La création des instances de javaBeans, l'injection des dépendances et le nommage des instances sont le coeur du framework *Spring*. Nous allons donc reprendre ces exemples en utilisant les outils fournis par *Spring*.

4.1 Installation de Spring

- Téléchargez le framework Spring³ avec ses dépendances (**en version 3.0.x**) [disponible ici⁴].
- Créez un projet Eclipse en plaçant dans le répertoire lib les archives java suivantes :
 - org.springframework.asm-3.0.6.RELEASE.jar
 - org.springframework.beans-3.0.6.RELEASE.jar
 - org.springframework.context-3.0.6.RELEASE.jar
 - org.springframework.core-3.0.6.RELEASE.jar
 - org.springframework.expression-3.0.6.RELEASE.jar
 - commons-logging-1.1.1.jar (disponible ici⁵ ou ici⁶)
- Préparez, dans un navigateur, un onglet vers la la documentation Spring⁷.

4.2 Créer et utiliser des javaBeans

Dans Spring, la définition des javaBeans, leurs noms et leurs paramètres sont définis dans un fichier de configuration au format XML. Une fois ce fichier préparé, Spring va être capable de créer les classes de service en effectuant les injections de dépendances et les initialisations nécessaires.

Préparez dans votre projet le fichier XML « *spring.xml* » suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 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">

    <bean id="logger" class="fr.ssa.log.imp.StderrLogger"
          init-method="init" destroy-method="close">
    </bean>

</beans>
```

³<http://www.springframework.org/>

⁴[ress-spring](http://www.springframework.org/resources-spring)

⁵[ress-spring](http://www.springframework.org/resources-spring)

⁶<http://commons.apache.org/logging/>

⁷<http://static.springframework.org/spring/docs/3.0.x/spring-framework-reference/html/>

Nous allons pouvoir utiliser Spring pour exploiter la classe *logger* :

```
package fr.ssa.appli;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import fr.ssa.log.ILog;

public class MyApp1 {

    // Utilisation du service "logger"
    void use(ILog logger) {
        logger.log("result = $1", "hello");
    }

    // intégration par Spring
    void run() {
        String conf = "spring.xml";
        AbstractApplicationContext context = new ClassPathXmlApplicationContext(conf);
        context.registerShutdownHook();

        // recuperer les beans
        ILog logger = context.getBean("logger", ILog.class);
        use(logger);
    }

    public static void main(String[] args) {
        MyApp1 app = new MyApp1();
        app.run();
    }
}
```

Si nous voulons changer de *logger*, il **n'est pas nécessaire de modifier notre programme**. Il suffit de changer le fichier XML :

```
<bean id="logger" class="fr.ssa.log.imp.BeanFileLogger"
      init-method="init" destroy-method="close">
  <property name="fileName">
    <value>/tmp/myapp.log</value>
  </property>
</bean>
```

Testez cette nouvelle version.

4.3 Utiliser les annotations pour déclarer les callbacks

Indiquer par des annotations dans la classe d'implantation les méthodes d'initialisation et de fermeture :

```

package fr.ssa.log.imp;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
...

public class BeanFileLogger implements ILog {

    ...

    // initialisation du service
    @PostConstruct
    public void init() {
        ...
    }

    // fermeture du service
    @PreDestroy
    public void close() {
        ...
    }

    ...

}

```

Le code XML de configuration peut maintenant être simplifié :

```

<bean id="logger" class="fr.ssa.log.imp.BeanFileLogger">
    <property name="fileName">
        <value>/tmp/myapp.log</value>
    </property>
</bean>

```

Important : Faites de même dans toutes vos classes d'implantation.

4.4 Préparer la couche métier

Utilisez cette nouvelle version, pour initialiser la couche métier et lui injecter la référence vers le *logger* utilisant un fichier :

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 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">

  <bean id="logger" class="fr.ssa.log.imp.BeanFileLogger">
    <property name="fileName">
      <value>/tmp/myapp.log</value>
    </property>
  </bean>

  <bean id="business" class="fr.ssa.bus.imp.SimpleBusiness">
    <property name="logger" ref="logger" />
  </bean>

</beans>

```

Vous pouvez parcourir avec profit les trois premières sections de ce chapitre⁸.

5 Utilisez les annotations de Spring

Depuis java 1.5, le framework Spring est également capable de décrire la configuration des beans par un jeu d'annotations. Nous allons pouvoir décrire l'injection de dépendances et la création des instances de manière déclarative et laisser le soin à Spring de réaliser les opérations.

5.1 Injection par annotation

Dans la classe SimpleBusiness, indiquez par une annotation que vous souhaitez l'injection d'un classe logger :

```

package fr.ssa.bus.imp;

import org.springframework.beans.factory.annotation.Autowired;
...

public class SimpleBusiness implements IBusiness {

    private ILog logger;

    ...

    public ILog getLogger() {
        return logger;
    }

    @Autowired
    public void setLogger(ILog logger) {
        this.logger = logger;
    }

}

```

Le code XML est maintenant simplifié (j'ai remplacé BeanFileLogger par StderrLogger) :

⁸<http://static.springframework.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html>

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 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 /><!-- pour traiter les annotations -->

  <bean id="logger" class="fr.ssa.log.imp.StderrLogger" />

  <bean id="business" class="fr.ssa.bus.imp.SimpleBusiness" />

</beans>

```

L'injection, c'est à dire la liaison entre la classe utilisée et la classe utilisatrice, se réalise sur la base du type (ici l'interface ILog) et du nom (ici « logger »).

5.2 Création des instances par annotation

Nous pouvons également déclarer une classe comme étant un service. Cette annotation va permettre de l'instancier automatiquement en cas de besoin. Appliquons ce principe sur la classe StderrLogger :

```

package fr.ssa.log.imp;

...

import org.springframework.stereotype.Service;

@Service("logger")
public class StderrLogger implements ILog {

    ...

}

```

Le code XML devient maintenant beaucoup plus simple :

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 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">

  <!-- pour traiter les annotations -->
  <context:annotation-config />

  <!-- pour décrire les classes à instancier -->
  <context:component-scan base-package="fr.ssa.log.imp" />

  <bean id="business" class="fr.ssa.bus.imp.SimpleBusiness" />

</beans>

```

Si la classe SimpleBusiness est elle aussi dotée d'une annotation @Service, le code XML devient :

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 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">

  <!-- pour traiter les annotations -->
  <context:annotation-config />

  <!-- pour décrire les classes à instancier -->
  <context:component-scan base-package="fr.ssa.log.imp" />
  <context:component-scan base-package="fr.ssa.bus.imp" />

</beans>

```

Découvrez d'autres possibilités en parcourant cette documentation⁹

6 Aller plus loin

6.1 Nouvelles implantations

- Proposez une nouvelle implantation de ILog qui se charge de transmettre les demandes de trace vers d'autres couches de trace. Cette classe est construite autour d'un paramètre qui est une Collection<ILog>.
- Notre première version de la couche métier mélange du code métier (addition, authentification) et du code de trace. Ce n'est pas une très bonne idée. Proposez une nouvelle implantation de décoration de la couche métier qui est construite sur deux paramètres :
 - une référence vers une autre implantation de la couche métier (qui ne fait aucune trace),
 - une référence vers une implantation de la couche *logger*,

Ce décorateur va retransmettre les requêtes métier et y associer une trace.

- Faites la même chose mais pour un décorateur qui va synchroniser les appels à la couche métier. cette implantation de la couche métier est donc valide dans un environnement multi-threads.

6.2 Tests unitaires

- Mettez en place un test unitaire pour chaque classe d'implantation des interfaces ILog et IBusiness. Pour ce faire, suivez, pour chaque implantation, les étapes ci-dessous :
 - Créez dans votre projet Eclipse une « **JUnit Test Case** » qui se nomme « *TestNomDeLaClasseTestée* » (de préférence en version 4).
 - Sélectionnez les méthodes métier à tester.
 - Exécutez cette classe en tant que « **JUnit Test** ».
 - Remplissez la méthode `setUp` qui est appelée avec chaque test (annotation `@Before`). Elle doit préparer une instance de la classe à tester.
 - Remplissez la méthode `tearDown` qui est appelée après chaque test (annotation `@After`). Elle doit fermer l'instance de test.

⁹<http://static.springframework.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-annotation-config>

- Vous pouvez maintenant remplir les fonctions de test (annotation `@Test`) dont le but est de vérifier le bon fonctionnement des méthodes de la classe testée. Pour ce faire, appelez ces méthodes et testez le résultat avec les nombreuses méthodes de test offertes par la classe `org.junit.Assert`¹⁰ (utilisez la complétion sur une ligne vide pour voir toutes les méthodes disponibles).
- Vous trouverez plus d'information sur ces sites :
 - <http://junit.sourceforge.net/>
 - <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>
 - <http://www.cavdar.net/2008/07/21/junit-4-in-60-seconds/>

6.3 Un service d'accès aux données (DAO)

Nous retrouvons dans pratiquement toutes les applications un service d'accès aux données (DAO pour *Data Access Object*). Je vous propose de spécifier et d'implanter un tel service en se basant sur deux *beans* :

- le premier représente une promotion de l'annuaire des anciens du Master¹¹ (année, diplôme, identifiant, étudiants) ;
- le second représente une entrée de cette annuaire (identifiant, nom, prénom, mail, site WEB, commentaires, date de mise à jour).

Votre service doit permettre de sauver et de récupérer les objets. Je vous propose d'utiliser la librairie XStream¹² pour sauvegarder vos données dans des fichiers XML. Je vous conseille notamment le le cours en deux minutes¹³ et la gestion (très simple) des collections persistantes¹⁴.

¹⁰<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

¹¹<http://masterinfo.univ-mrs.fr/annuaire/index.html>

¹²<http://xstream.codehaus.org/index.html>

¹³<http://xstream.codehaus.org/tutorial.html>

¹⁴<http://xstream.codehaus.org/persistence-tutorial.html>