JAVA - TD

Deux exemples d'architectures Java à trois couches

serge.tahe@istia.univ-angers.fr, septembre 2006

Objectifs du TD :

- architectures à 3 couches
- interfaces graphiques
- interfaces web
- accès aux données avec Spring / iBATIS

Outils utilisés :

- **Spring** : http://www.springframework.org/
- **Ibatis SqlMap** : http://www.ibatis.com/
- **Junit** : http://www.junit.org/index.htm
- **Eclipse 3.2** : http://www.eclipse.org/
- **Tomcat 5.5.17** : http://tomcat.apache.org/
- **JDK 1.5** : http://java.sun.com/j2se/1.5.0/
- **Firebird** : http://firebird.sourceforge.net/ : SGBD, pilote JDBC. En fait, toute source JDBC fait l'affaire.
- **IBExpert, personal edition** : http://www.hksoftware.net/download/ibep_2005.2.14.1_full.exe (mars 2005). IBExpert permet d'administrer graphiquement le SGBD Firebird.

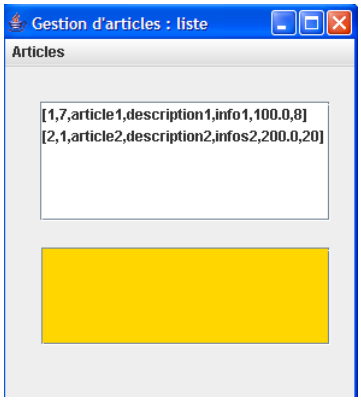
La compréhension de ce document nécessite divers pré-requis. Certains d'entre-eux peuvent être acquis dans des documents que j'ai écrits. Dans ce cas, je les cite. Il est bien évident que ce n'est qu'une suggestion et que le lecteur peut utiliser ses documents favoris.

- langage Java : [http://tahe.developpez.com/java/cours]
- utilisation de l'aspect IoC de Spring : [http://tahe.developpez.com/java/springioc]
- documentation Ibatis SqlMap : [http://cvs.apache.org/dist/ibatis/ibatis.java/docs/iBATIS-SqlMaps-2-Tutorial_fr.pdf]
- l'utilisation d'Ibatis Sqlmap est décrite notamment dans [Bases de la programmation web MVC en Java, paragraphe 17.3] disponible à l'url [http://tahe.developpez.com/java/baseswebmvc].
- Firebird : [http://firebird.sourceforge.net/pdfmanual/Firebird-1.5-QuickStart.pdf] (mars 2005).

1 L'application [dbarticles-swing]

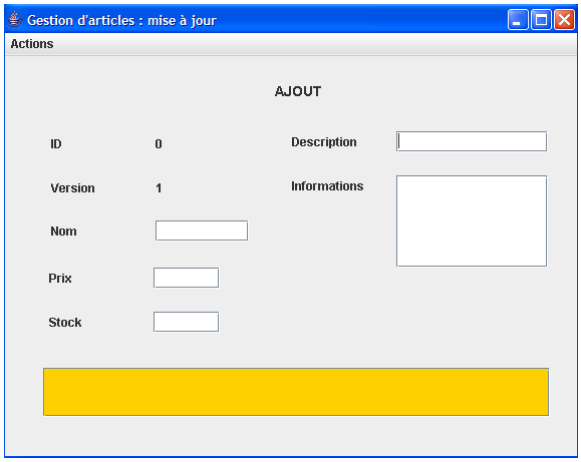
L'application [dbarticles-swing] est une application graphique Java permettant de gérer une table d'articles vendus par un magasin. Les différentes vues présentées à l'utilisateur seront les suivantes :

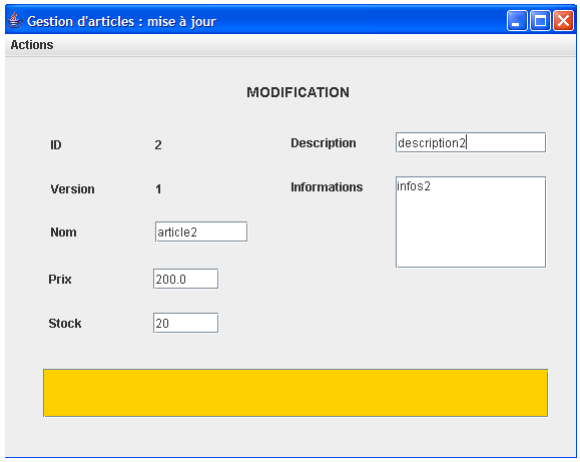
- la vue [LISTE] qui présente une liste des articles en vente



la vue [AJOUT] qui permet d'ajouter un article :

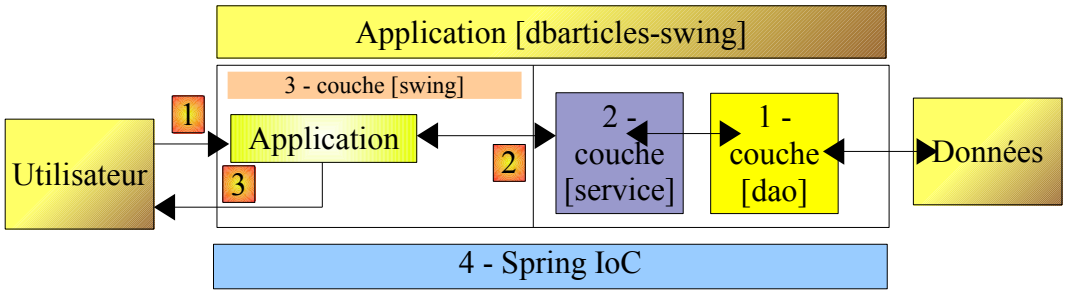
la vue [MODIFICATION] qui permet de modifier un article :





1.1 Architecture générale de l'application

On souhaite construire une application ayant la structure à trois couches suivante :



- la couche [1-dao] s'occupe de l'accès aux données. Celles-ci seront placées dans une base de données.
- la couche [2-service] s'occupe des accès transactionnels à la base de données.
- la couche [3-swing] s'occupe de la présentation des données à l'utilisateur et de l'exécution de ses requêtes.
- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces Java
- l'intégration des différentes couches est réalisée par [4 - **Spring IoC]**

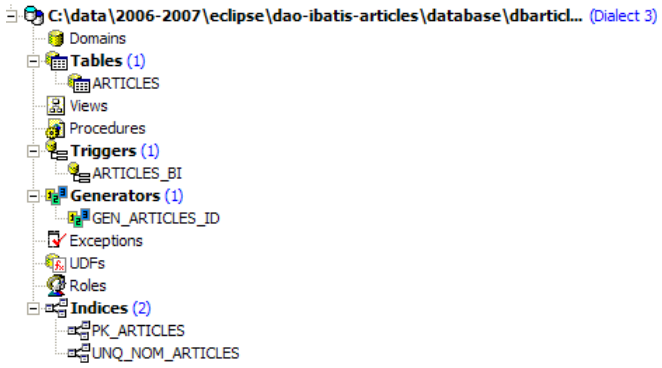
Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

1. le client fait une demande à l'application.
2. l'application traite cette demande. Pour ce faire, elle peut avoir besoin de l'aide de la couche [service] qui elle-même peut avoir besoin de la couche [dao] si des données doivent être échangées avec la base de données.
3. l'application reçoit une réponse de la couche [service]. Selon celle-ci, elle envoie la vue appropriée au client.

1.2 La base de données

La base de données ne contient qu'une table, appelée ARTICLES. Nous utiliserons ici le SGBD Firebird comme exemple d'implémentation. Cependant, nous ferons en sorte d'écrire une application indépendante du type de SGBD utilisé.

La base Firebird s'appelle [dbarticles.gdb] et a la structure suivante (copie d'écran IBExpert) :



La BD Firebird [dbarticles.gdb] a été générée avec les commandes SQL suivantes :

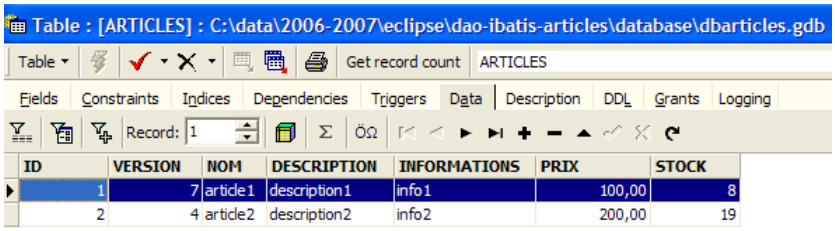
```
1.
2.  /******************************************************************************/
3.  /***                                 Tables                                 ***/
4.  /******************************************************************************/
5.
6.
7.  CREATE GENERATOR GEN_ARTICLES_ID;
8.
9.  CREATE TABLE ARTICLES (
10.     ID              INTEGER NOT NULL,
11.     "VERSION"       INTEGER NOT NULL,
12.     NOM             VARCHAR(30) NOT NULL,
13.     DESCRIPTION     VARCHAR(100) NOT NULL,
14.     INFORMATIONS    VARCHAR(1000) NOT NULL,
15.     PRIX            DECIMAL(15,2) NOT NULL,
16.     STOCK           INTEGER NOT NULL
17. );
18.
19.
20.
21.
22. /* Check constraints definition */
23.
24. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM_ARTICLES check (nom<>'');
25. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX_ARTICLES check (prix>=0);
26. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_VERSION_ARTICLES check (version>0);
27. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCK_ARTICLES check (stock>=0);
28. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_DESCRIPTION_ARTICLES check (description<>'');
29. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_INFORMATIONS_ARTICLES check (informations<>'');
30.
31.
32. /******************************************************************************/
33. /***                            Unique Constraints                          ***/
34. /******************************************************************************/
35.
36. ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM_ARTICLES UNIQUE (NOM);
37.
38.
39. /******************************************************************************/
40. /***                              Primary Keys                              ***/
41. /******************************************************************************/
42.
43. ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);
44.
45.
46. /******************************************************************************/
47. /***                           Triggers for tables                          ***/
48. /******************************************************************************/
49.
50.
51. /* Trigger: ARTICLES_BI */
52. CREATE TRIGGER ARTICLES_BI FOR ARTICLES
53. ACTIVE BEFORE INSERT POSITION 0
54. AS
55. BEGIN
56.   IF (NEW.ID IS NULL) THEN
57.     NEW.ID = GEN_ID(GEN_ARTICLES_ID,1);
58. END
```

- lignes 9-17 : définition de la table [ARTICLES]

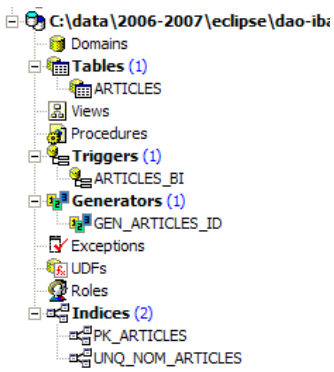
`id`	clé primaire identifiant un article de façon unique. Un générateur nommé " GEN_ARTICLES_ID " a été créé. C'est lui qui fournira les valeurs successives de la colonne id en commençant par 1.
`version`	n° de version. A chaque fois, que l'article est modifié, son n° de version est incrémenté de 1.
`nom`	nom de l'article
`description`	description courte de l'article
`informations`	description longue de l'article
`prix`	le prix de l'article
`stock`	le stock actuel de l'article

- lignes 24-29 : contraintes d'intégrité sur les colonnes de la table
- ligne 36 : contrainte d'unicité sur la colonne NOM. On ne pourra pas avoir deux noms identiques.
- ligne 43 : contrainte de clé primaire sur la colonne ID
- lignes 52-58 : procédure stockée exécutée avant chaque insertion dans la table ARTICLES. Si l'ordre d'insertion ne contient pas de valeur pour la clé primaire ID, cette procédure lui en affecte automatiquement une à partir du générateur [GEN_ARTICLES_ID].

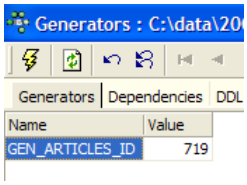
La table [ARTICLES] pourrait avoir le contenu suivant :



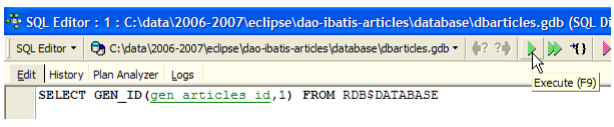
La base [dbarticles.gdb] a, outre la table [ARTICLES], un objet appelé générateur et nommé [GEN_ARTICLES_ID]. Ce générateur délivre des nombres entiers successifs que nous utiliserons pour donner sa valeur, à la clé primaire [ID] de la classe [PERSONNES]. Prenons un exemple pour illustrer son fonctionnement :



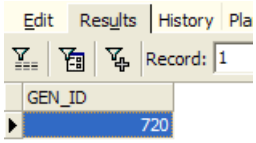
- double-clic sur [GEN_ARTICLES_ID]



- le générateur a actuellement la valeur 719

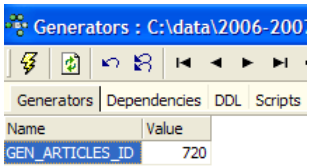


- émettons l'ordre SQL ci-dessus (F12) ->



- la valeur obtenue est l'ancienne valeur du générateur +1

On peut constater que la valeur du générateur [GEN_PERSONNES_ID] a changé (double-clic dessus + F5 pour rafraîchir) :



L'ordre SQL

```
SELECT GEN_ID ( GEN_ARTICLES_ID,1 ) FROM RDB$DATABASE
```

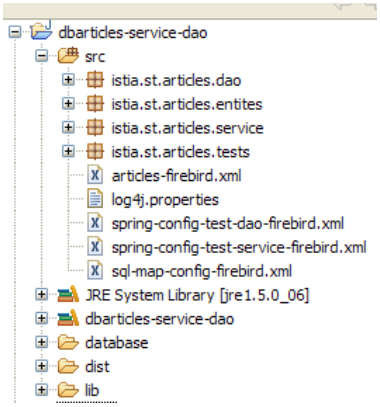
permet donc d'avoir la valeur suivante du générateur [GEN_ARTICLES_ID]. GEN_ID est une fonction interne de Firebird et [RDB$DATABASE], une table système de ce SGBD.

1.3 Les couches [dao] et [service] d'accès aux articles

Nous allons tout d'abord développer les couches [dao] et [service] d'accès aux articles. Ces deux couches permettront aux applications de faire les opération de base sur la table [ARTICLES] :

- obtenir tous les articles de la table
- modifier un article
- supprimer un article
- ajouter un article

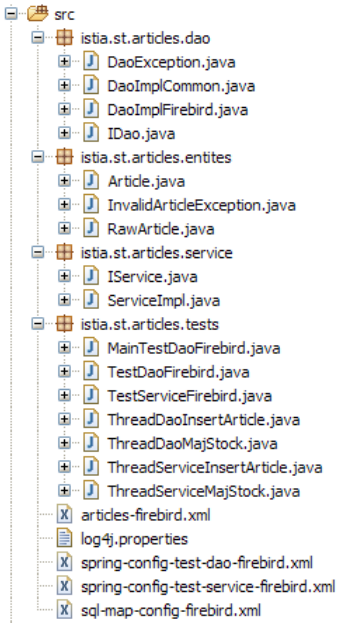
Pour ce faire, nous utiliserons le projet Eclipse [dbarticles-service-dao] suivant :



Le projet est un projet Java.

Dossier [src]

Ce dossier contient les codes source des couches [dao] et [service] :



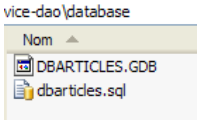
On y trouve différents paquetages :

- [istia.st.articles.entites] : contient les classes [Article] et [RawArticle] qui serviront à encapsuler une ligne de la table [ARTICLES].
- [istia.st.articles.dao] : contient la couche [dao]. Cette couche est en contact avec la base de données via le pilote JDBC de celle-ci.
- [istia.st.articles.service] : contient la couche [service] qui va assurer les transactions sur la base de données
- [istia.st.articles.tests] : contient les tests JUnit des couches [dao] et [service]

ainsi que des fichiers de configuration qui doivent être dans le *ClassPath* de l'application. Nous y reviendrons.

Dossier [database]

Ce dossier contient la base de données Firebird des articles :



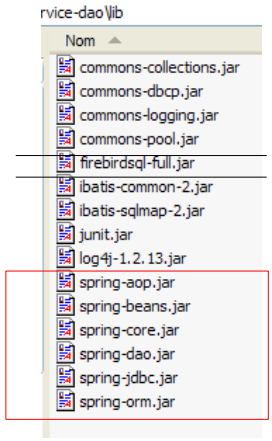
- [dbarticles.gdb] est la base de données.
- [dbarticles.sql] est le script SQL de génération de la base :

```
1.  /******************************************************************************/
2.  /***          Generated by IBExpert 2006.03.07 21/08/2006 16:55:36          ***/
3.  /******************************************************************************/
4.
5.  SET SQL DIALECT 3;
6.
7.  SET NAMES NONE;
8.
9.  CREATE DATABASE 'C:\data\2006-2007\webjava\dbarticles-service-dao\database\DBARTICLES.GDB'
10. USER 'SYSDBA' PASSWORD 'masterkey'
11. PAGE_SIZE 16384
12. DEFAULT CHARACTER SET NONE;
13.
14.
15.
16. /******************************************************************************/
17. /***                                 Generators                             ***/
18. /******************************************************************************/
19.
20. CREATE GENERATOR GEN_ARTICLES_ID;
21. SET GENERATOR GEN_ARTICLES_ID TO 720;
22.
23.
24.
25. /******************************************************************************/
26. /***                                   Tables                               ***/
27. /******************************************************************************/
28.
```

```
29.
30.
31. CREATE TABLE ARTICLES (
32.     ID            INTEGER NOT NULL,
33.     "VERSION"     INTEGER NOT NULL,
34.     NOM           VARCHAR(30) NOT NULL,
35.     DESCRIPTION   VARCHAR(100) NOT NULL,
36.     INFORMATIONS  VARCHAR(1000) NOT NULL,
37.     PRIX          DECIMAL(15,2) NOT NULL,
38.     STOCK         INTEGER NOT NULL
39. );
40.
41. INSERT INTO ARTICLES (ID, "VERSION", NOM, DESCRIPTION, INFORMATIONS, PRIX, STOCK) VALUES (1, 7,
    'article1', 'description1', 'info1', 100, 8);
42. INSERT INTO ARTICLES (ID, "VERSION", NOM, DESCRIPTION, INFORMATIONS, PRIX, STOCK) VALUES (2, 4,
    'article2', 'description2', 'info2', 200, 19);
43.
44. COMMIT WORK;
45.
46.
47.
48. /* Check constraints definition */
49.
50. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM_ARTICLES check (nom<>'');
51. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX_ARTICLES check (prix>=0);
52. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_VERSION_ARTICLES check (version>0);
53. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCK_ARTICLES check (stock>=0);
54. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_DESCRIPTION_ARTICLES check (description<>'');
55. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_INFORMATIONS_ARTICLES check (informations<>'');
56.
57.
58. /******************************************************************************/
59. /***                           Unique Constraints                           ***/
60. /******************************************************************************/
61.
62. ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM_ARTICLES UNIQUE (NOM);
63.
64.
65. /******************************************************************************/
66. /***                              Primary Keys                              ***/
67. /******************************************************************************/
68.
69. ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);
70.
71.
72. /******************************************************************************/
73. /***                                 Triggers                               ***/
74. /******************************************************************************/
75.
76.
77. SET TERM ^ ;
78.
79.
80. /******************************************************************************/
81. /***                          Triggers for tables                           ***/
82. /******************************************************************************/
83.
84.
85.
86. /* Trigger: ARTICLES_BI */
87. CREATE TRIGGER ARTICLES_BI FOR ARTICLES
88. ACTIVE BEFORE INSERT POSITION 0
89. AS
90. BEGIN
91.   IF (NEW.ID IS NULL) THEN
92.     NEW.ID = GEN_ID(GEN_ARTICLES_ID,1);
93. END
94. ^
95.
96. SET TERM ; ^
```

Dossier [lib]

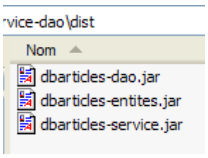
Ce dossier contient les archives nécessaires à l'application :



On notera la présence du pilote JDBC [firebirdsql-full.jar] du SGBD Firebird ainsi que d'un certain nombre d'archives [spring-*.jar]. Nous pourrions utiliser l'unique archive [spring.jar] que l'on trouve dans le dossier [dist] de la distribution Spring et qui contient la totalité des classes de Spring. On peut aussi n'utiliser que les seules archives nécessaires au projet. C'est ce que nous faisons ici. Toutes les archives du dossier [lib] ont été placées dans le *Classpath* du projet.

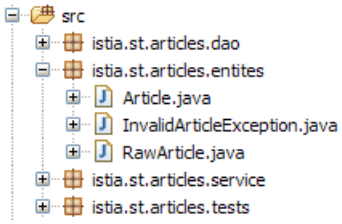
Dossier [dist]

Ce dossier contiendra les archives issues de la compilation des classes de l'application :



- [dbarticles-dao.jar] : archive du paquetage [dao]
- [dbarticles-service.jar] : archive du paquetage [service]
- [dbarticles-entites.jar] : archive du paquetage [entites]

1.3.1 Le paquetage [entites]



`Article`	classe qui encapsule une ligne de la table [ARTICLES]. Les constructeurs de cette classe vérifient que les données fournies pour initialiser un objet de type [Article] sont valides. Un stock négatif par exemple, provoquera une exception de type [InvalidArticleException].
`InvalidArticleException`	classe dérivée de [RuntimeException] qui servira à signaler des erreurs d'instanciation d'objets de type [Article].
`RawArticle`	classe qui encapsule également une ligne de la table [ARTICLES] mais dans laquelle les constructeurs ne vérifient pas la validité des données qui servent à initialiser l'objet.

1.3.1.1 La classe [InvalidArticleException]

Cette classe dérivée de [RuntimeException] nous servira à signaler des erreurs lors de l'instanciation de la table [ARTICLES]. Son code est le suivant :

```
1.  package istia.st.articles.entites;
2.
3.  @SuppressWarnings("serial")
4.  public class InvalidArticleException extends RuntimeException {
```

```
5.   // code erreur
6.   private int code;
7.
8.   // getter
9.   public int getCode() {
10.    return code;
11.  }
12.
13.  // constructeur
14.  public InvalidArticleException(String message, int code) {
15.    super(message);
16.    this.code = code;
17.  }
18. }
```

La seule nouveauté par rapport à une exception classique est apportée par la ligne 6. Nous distinguerons les différentes causes ayant provoqué l'exception [InvalidArticleException] par un code d'erreur mémorisé par le champ [code]. La présence de ce champ entraîne la création :

- de la méthode [getCode] des lignes 9-11 qui permet de récupérer le code de l'exception
- du constructeur défini lignes 14-17 qui permet de définir les attributs d'une exception de type [InvalidArticleException] :
  - un message d'erreur qui est passé à la classe parent [RuntimeException] (ligne 15)
  - le code de l'erreur enregistré dans le champ privé [code] (ligne 16)

On notera enfin (ligne 4) que le type [InvalidArticleException] dérive du type [RunTimeException] ce qui en fait une exception non contrôlée. Cela implique deux choses :

- une méthode lançant cette exception n'est pas obligée de la déclarer dans sa signature par [throws InvalidArticleException]
- un bloc de code incluant un appel à une méthode lançant l'exception [InvalidArticleException] n'est pas obligé de la gérer avec un try / catch.

Une exception de type [InvalidArticleException] sera construite par un code du genre :

```
throw new InvalidArticleException(" stock invalide ",4);
```

1.3.1.2 La classe [Article]

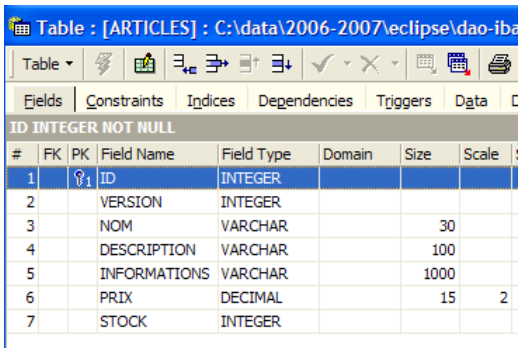
Un objet [Article] encapsule les données d'une ligne de la table [ARTICLES]. Son code est le suivant :

```
1.  package istia.st.articles.entites;
2.
3.  public class Article {
4.
5.    // identifiant unique de l'article
6.    protected int id;
7.
8.    // la version actuelle
9.    protected long version;
10.
11.   // le nom
12.   protected String nom;
13.
14.   // la description
15.   protected String description;
16.
17.   // les informations
18.   protected String informations;
19.
20.   // le prix
21.   protected double prix;
22.
23.   // le stock
24.   protected int stock;
25.
26.   // constructeur par défaut
27.   public Article() {
28.
29.   }
30.
31.   // constructeur par recopie
32.   public Article(Article article) {
33.     // argument valide ?
34.     if (article == null) {
35.       throw new InvalidArticleException("argument article=null invalide",
36.           10);
37.     }
38.     // recopie
39.     setId(article.getId());
```

```
40.     setVersion(article.getVersion());
41.     setNom(article.getNom());
42.     setDescription(article.getDescription());
43.     setInformations(article.getInformations());
44.     setPrix(article.getPrix());
45.     setStock(article.getStock());
46.   }
47.
48.   // constructeur avec initialisations
49.   public Article(int id, long version, String nom, String description,
50.       String informations, double prix, int stock) {
51.     setId(id);
52.     setVersion(version);
53.     setNom(nom);
54.     setDescription(description);
55.     setInformations(informations);
56.     setPrix(prix);
57.     setStock(stock);
58.   }
59.
60.   // toString
61.   public String toString() {
62.     return "[" + id + "," + version + "," + nom + "," + description + ","
63.         + informations + "," + prix + "," + stock + "]";
64.   }
65.
66.   // getters - setters
67.   public int getId() {
68.     return id;
69.   }
70.
71.   public void setId(int id) {
72.     if (id <= -1)
73.       throw new InvalidArticleException("id [" + id + "]invalide", 1);
74.     this.id = id;
75.   }
76.
77.   public long getVersion() {
78.     return version;
79.   }
80.
81.   public void setVersion(long version) {
82.     if (version <= -1)
83.       throw new InvalidArticleException("version [" + version
84.           + "]invalide", 2);
85.     this.version = version;
86.   }
87.
88.   public String getNom() {
89.     return nom;
90.   }
91.
92.   public void setNom(String nom) {
93.     if (nom == null || nom.trim().length() == 0)
94.       throw new InvalidArticleException("nom [" + nom + "]invalide", 3);
95.     this.nom = nom;
96.   }
97.
98.   public String getDescription() {
99.     return description;
100.  }
101.
102.  public void setDescription(String description) {
103.    if (description == null || description.trim().length() == 0)
104.      throw new InvalidArticleException("description [" + description
105.          + "]invalide", 4);
106.    this.description = description;
107.  }
108.
109.  public String getInformations() {
110.    return informations;
111.  }
112.
113.  public void setInformations(String informations) {
114.    if (informations == null || informations.trim().length() == 0)
115.      throw new InvalidArticleException("informations [" + informations
116.          + "]invalide", 7);
117.    this.informations = informations;
118.  }
119.
120.  public double getPrix() {
121.    return prix;
122.  }
123.
```

```
124. public void setPrix(double prix) {
125.   if (prix < 0)
126.     throw new InvalidArticleException("prix [" + prix + "]invalide", 5);
127.   this.prix = prix;
128. }
129.
130. public int getStock() {
131.   return stock;
132. }
133.
134. public void setStock(int stock) {
135.   if (stock < 0)
136.     throw new InvalidArticleException("stock [" + stock + "]invalide",
137.         6);
138.   this.stock = stock;
139. }
140.
141.}
```

- lignes 6-24 : les champs correspondant un à un aux colonnes de la table [ARTICLES] :



- lignes 27-29 : le constructeur sans arguments de la classe.
- lignes 32-46 : le constructeur par recopie de la classe : construit un clône de l'article passé en argument.
- lignes 49-58 : le constructeur qui construit un objet [Article] à partir des sept informations d'une ligne de la table [ARTICLES].
- lignes 61-64 : une méthode *toString* qui nous permet d'identifier un objet [Article] par une chaîne de caractères.
- lignes 67-139 : les méthodes *get* / *set* permettant d'obtenir ou de fixer les valeurs des champs de la classe. Les méthodes [set] vérifient la validité de la donnée passée en paramètre à la méthode et lance une exception de type [InvalidArticleException] si celle-ci est invalide. On notera que les constructeurs avec arguments utilisent ces méthodes [set]. La validité des données passées à ces constructeurs est donc vérifiée. On notera également que la signature de ces méthodes et constructeurs ne comporte pas la déclaration [throws InvalidArticleException] parce que cette exception est non contrôlée.

1.3.1.3 La classe [RawArticle]

Un objet [Article] est toujours dans un état valide grâce aux méthodes [set] qui vérifient la validité des données passées pour fixer l'état de l'objet. Ce comportement n'est pas toujours souhaitable. C'est notamment le cas où une ligne de la table [ARTICLES] doit être transférée dans un objet [Article]. Si les constraintes sur les colonnes de la table [ARTICLES] ne sont pas les mêmes que celles sur les champs de l'objet [Article], une exception sera lancée lorsque on transfèrera certaines lignes de la table [ARTICLES] dans les objets [Article].

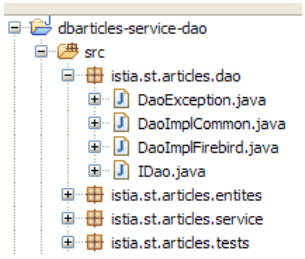
Prenons par exemple un SGBD peu évolué qui n'accepte pas les contraintes. Une ligne de la table [ARTICLES] peut alors avoir un stock<0 suite à une erreur de saisie. Le transfert de cette ligne dans un objet [Article] sera refusé et une exception de type [InvalidArticleException] lancée. Ce peut être très gênant. On peut vouloir récupérer toutes les lignes de la table [ARTICLES] même celles invalides. La classe [**RawArticle**] est créée pour ce cas de figure. Elle est identique à la classe [Article] sauf que ses méthodes [set] ne vérifient pas la validité du paramètre qui leur est passé. Pour ne pas perdre l'existant, la classe [**RawArticle**] est dérivée de la classe [**Article**].

Question : écrire la classe [RawArticle]

1.3.2 Le paquetage [dao]

1.3.2.1 Les éléments du paquetage [dao]

Le paquetage [dao] et le paquetage [entites] constituent la couche [dao] de notre application. Il est formé des classes et interfaces suivantes :



- [IDao] est l'interface présentée par la couche [dao]
- [DaoImplCommon] est une implémentation de celle-ci où le groupe de personnes se trouve dans une table de base de données. [DaoImplCommon] regroupe des fonctionnalités indépendantes du SGBD.
- [DaoImplFirebird] est une classe dérivée de [DaoImplCommon] pour gérer spécifiquement une base Firebird.
- [DaoException] est le type des exceptions non contrôlées, lancées par la couche [dao].

La classe [DaoException] est la suivante :

```
1.  package istia.st.articles.dao;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  @SuppressWarnings("serial")
7.  public class DaoException extends RuntimeException {
8.
9.    // code erreur
10.   private int code;
11.
12.   // causes
13.   List<DaoException> daoExceptions = new ArrayList<DaoException>();
14.
15.   // getters - setters
16.   public int getCode() {
17.     return code;
18.   }
19.
20.   public List<DaoException> getDaoExceptions() {
21.     return daoExceptions;
22.   }
23.
24.   public void setDaoExceptions(List<DaoException> daoExceptions) {
25.     this.daoExceptions = daoExceptions;
26.   }
27.
28.   // constructeurs
29.   public DaoException() {
30.     super();
31.   }
32.
33.   public DaoException(String message, int code) {
34.     super(message);
35.     this.code = code;
36.   }
37. }
```

Cette classe peut s'utiliser pour encapsuler une erreur particulière ou pour encapsuler une liste d'erreurs :

- pour encapsuler une erreur particulière, on utilisera le constructeur des lignes 33-36.
- pour encapsuler une liste d'exceptions de type [DaoException], on utilisera le constructeur sans arguments des lignes 29-31, on mettra la valeur 100 dans le champ [code] et on utilisera le champ [daoExceptions] pour encapsuler les différentes exceptions.

L'interface [IDao] de la couche [dao] est la suivante :

```
1.  package istia.st.articles.dao;
2.
```

```
3.  import istia.st.articles.entites.Article;
4.
5.  import java.util.List;
6.
7.  public interface IDao {
8.    // liste de tous les articles
9.    List<Article> getAll();
10.   // obtenir un article particulier
11.   Article getOne(int id);
12.   // ajouter/modifier un article
13.   void saveOne(Article article);
14.   // supprimer un article
15.   void deleteOne(int id);
16. }
```

- l'interface a les quatre méthodes permettant les opérations dites CRUD (Create Read Update Delete) sur la table [ARTICLES]
- ligne 9 : méthode [getAll()] qui rend une liste de tous les articles de la table [ARTICLES]
- ligne 11 : méthode [getOne(int id)] qui rend l'article de la table [ARTICLES] identifié par [id]
- ligne 13 : méthode [saveOne(Article article)] qui
  - ajoute [article] dans la table [ARTICLES] si [article] a son champ id=0
  - remplace par [article], la ligne de la table [ARTICLES] identifiée par id
- ligne 15 : méthode [deleteOne(int id)] qui supprime de la table [ARTICLES] l'article identifié par [id]

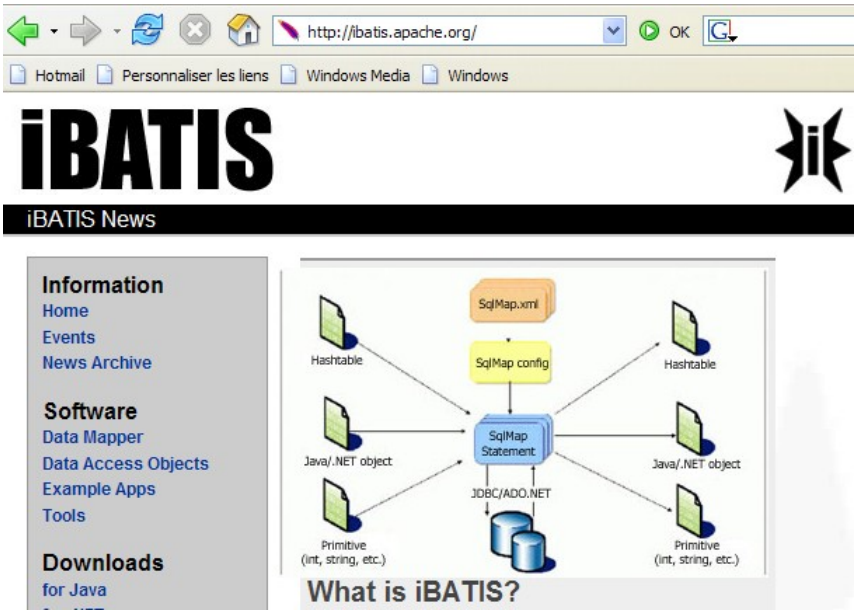
La classe [DaoImplCommon] implémentant cette interface sera la suivante :

```
1.  package istia.st.articles.dao;
2.
3.  import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;
4.  import istia.st.articles.entites.Article;
5.
6.  import java.util.List;
7.
8.  public class DaoImplCommon extends SqlMapClientDaoSupport implements IDao {
9.
10.   // liste des personnes
11.   @SuppressWarnings("unchecked")
12.   public List<Article> getAll() {
13. ...
14.   }
15.
16.   // obtenir un article en particulier
17.   public Article getOne(int id) {
18. ...
19.   }
20.
21.   // suppression d'un article
22.   public void deleteOne(int id) {
23. ...
24.   }
25.
26.   // ajouter ou modifier un article
27.   public void saveOne(Article article) {
28.     // ajout ou modification ?
29.     if (article.getId() == 0) {
30.       // ajout
31.       insertArticle(article);
32.     } else {
33.       // modification
34.       updateArticle(article);
35.     }
36.   }
37.
38.   // ajouter un article
39.   protected void insertArticle(Article article) {
40. ...
41.   }
42.
43.   // modifier un article
44.   protected void updateArticle(Article article) {
45. ...
46.   }
47.
48. ...
49. }
```

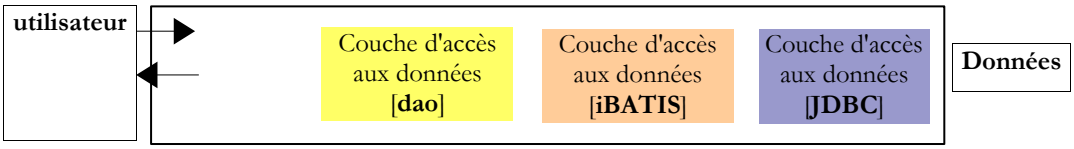
- ligne 8 : la classe [DaoImplCommon] implémente l'interface [IDao] et donc les quatre méthodes [getAll, getOne, saveOne, deleteOne].
- lignes 27-36 : la méthode [saveOne] utilise deux méthodes internes [insertPersonne] et [updatePersonne] selon qu'on doit faire un ajout ou une modification d'article.
- ligne 8 : pour implémenter l'interface [IDao], la classe [DaoImpl] dérive de la classe Spring [**SqlMapClientDaoSupport**].

1.3.2.2 La couche d'accès aux données [iBATIS]

La classe Spring [SqlMapClientDaoSupport] utilise un framework tierce [Ibatis SqlMap] disponible à l'url [http://ibatis.apache.org/] :



[iBATIS] est un projet Apache qui facilite la construction de couches [dao] s'appuyant sur des bases de données. Avec [iBATIS], l'architecture de la couche d'accès aux données est la suivante :



[iBATIS] s'insère entre la couche [dao] de l'application et le pilote JDBC de la base de données.

Question : écrire la classe [DaoImplCommon] qui implémente ci-dessus la couche [dao]. On suivra l'exemple donné dans le document [Bases de la programmation web MVC en Java, paragraphe 17.3].

L'instanciation du singleton [DaoImplCommon] implémentant la couche [dao] sera réalisée par le fichier Spring suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>org.firebirdsql.jdbc.FBDriver</value>
9.      </property>
10.     <!-- attention : ne pas laisser d'espaces entre les deux balises <value> -->
11.     <property name="url">
12.       <value>jdbc:firebirdsql:localhost/3050:...</value>
13.     </property>
14.     <property name="username">
15.       <value>sysdba</value>
16.     </property>
17.     <property name="password">
18.       <value>masterkey</value>
19.     </property>
20.   </bean>
21.   <!-- SqlMapCllient -->
22.   <bean id="sqlMapClient"
23.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
24.     <property name="dataSource">
25.       <ref local="dataSource"/>
26.     </property>
27.     <property name="configLocation">
```

```
28.       <value>classpath:sql-map-config-firebird.xml</value>
29.     </property>
30.   </bean>
31.   <!-- la classes d'accè à la couche [dao] -->
32.   <bean id="dao" class="istia.st.articles.dao.DaoImplCommon">
33.     <property name="sqlMapClient">
34.       <ref local="sqlMapClient"/>
35.     </property>
36.   </bean>
37. </beans>
```

- ligne 12 : on mettra le chemin exact du fichier Firebird [dbarticles.gdb]
- ligne 28 : référence le fichier qui configure la couche [iBATIS]. Le contenu de celui-ci est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.  <!DOCTYPE sqlMapConfig
3.      PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
4.      "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
5.
6.  <sqlMapConfig>
7.    <sqlMap resource="articles-firebird.xml"/>
8.  </sqlMapConfig>
```

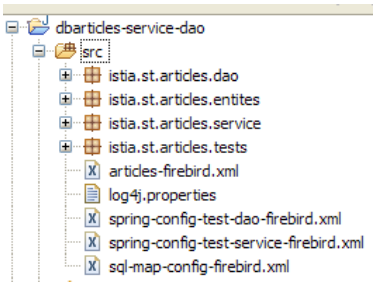
- ligne 7 : référence le fichier qui
  - fait la correspondance entre les colonnes de la table [ARTICLES] et les champs des objets [Article] et [RawArticle]
  - définit les commandes SQL à émettre sur la BD

Le contenu du fichier de " mapping " [articles-firebird.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.
3.  <!DOCTYPE sqlMap
4.      PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
5.      "http://www.ibatis.com/dtd/sql-map-2.dtd">
6.
7.  <sqlMap>
8.    <!-- alias classe [Article] -->
9.    <typeAlias alias="Article.classe"
10.     type="istia.st.articles.entites.Article" />
11.   <!-- mapping table [ARTICLES] - objet [Article] -->
12.   <resultMap id="Article.map" class="Article.classe">
13.     <result property="id" column="ID" />
14.     <result property="version" column="VERSION" />
15.     <result property="nom" column="NOM" />
16.     <result property="description" column="DESCRIPTION" />
17.     <result property="informations" column="INFORMATIONS" />
18.     <result property="prix" column="PRIX" />
19.     <result property="stock" column="STOCK" />
20.   </resultMap>
21.   <resultMap id="ArticlePartiel.map" class="Article.classe">
22.     <result property="id" column="ID" />
23.     <result property="version" column="VERSION" />
24.     <result property="nom" column="NOM" />
25.     <result property="description" column="DESCRIPTION" />
26.     <result property="prix" column="PRIX" />
27.   </resultMap>
28.   <!-- liste de tous les articles -->
29.   <select id="Articles.getAll" resultMap="Article.map">
30.     select ID, VERSION, NOM, DESCRIPTION, INFORMATIONS, PRIX, STOCK FROM ARTICLES
31.   </select>
32.   <!-- obtenir un article en particulier -->
33.   <select id="Articles.getOne" resultMap="Article.map"
34.     parameterClass="int">
35.     select ID, VERSION, NOM, DESCRIPTION, INFORMATIONS, PRIX, STOCK
36.     FROM ARTICLES WHERE ID=#value#
37.   </select>
38.   <!-- ajouter un article -->
39.   <insert id="Articles.insertOne" parameterClass="Article.classe">
40.     <selectKey keyProperty="id">
41.       SELECT GEN_ID(GEN_ARTICLES_ID,1) as "value" FROM
42.       RDB$$DATABASE
43.     </selectKey>
44.     INSERT INTO ARTICLES(ID, VERSION, NOM, DESCRIPTION,
45.     INFORMATIONS, PRIX, STOCK) VALUES(#id#, #version#, #nom#,
46.     #description#, #informations#, #prix#, #stock#)
47.   </insert>
48.   <!-- mettre à jour un article -->
49.   <update id="Articles.updateOne" parameterClass="Article.classe">
50.     UPDATE ARTICLES SET VERSION=#version#+1, NOM=#nom#,
51.     DESCRIPTION=#description#,
52.     INFORMATIONS=#informations#,PRIX=#prix#, STOCK=#stock# WHERE
53.     ID=#id# and VERSION=#version#
54.   </update>
```

```
55.  <!-- supprimer un article -->
56.  <delete id="Articles.deleteOne" parameterClass="int">
57.    DELETE FROM ARTICLES WHERE ID=#value#
58.  </delete>
59. </sqlMap>
```

Les trois fichiers de configuration de la couche [dao] doivent être dans le ClassPath de l'application. Ici, ils sont placés dans le dossier [src] du projet Eclipse :



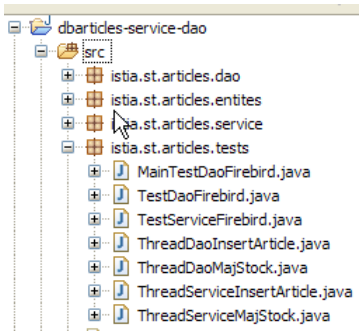
`spring-config-test-dao-firebird.xml`	le fichier de configuration Spring pour les tests de la couche [dao]. Son contenu est celui qui vient d'être décrit.
`spring-config-test-service-firebird.xml`	le fichier de configuration Spring pour les tests de la couche [service]
`sql-map-config-firebird.xml`	le fichier de configuration de la couche [iBATIS]
`articles-firebird.xml`	le fichier de " mappings " de la couche [iBATIS]
`log4j.properties`	le fichier de configuration de l'outil de logs [log4j] utilisé par Spring pour écrire des logs sur la console

Eclipse recopie automatiquement dans un dossier [/bin] tout fichier présent dans le dossier [/src] et qui n'est pas un fichier Java. Par ailleurs, le dossier [/bin] fait partie du ClassPath d'un projet Eclipse. Donc tout fichier placé dans [/src] fera partie de ce ClassPath. Ce sera le cas ici, des fichiers de configuration de la couche [dao].

1.3.3 Tests de la couche [dao]

1.3.3.1 Tests de l'implémentation [DaoImplCommon]

La couche [dao] sera testée avec des tests JUnit :



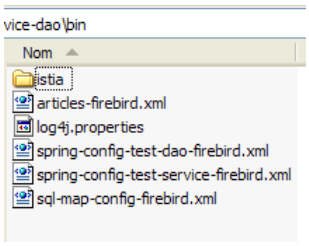
Avant de faire des tests intensifs, nous pouvons commencer par un simple programme de type [main] qui va afficher le contenu de la table [ARTICLES]. C'est la classe [MainTestDaoFirebird] :

```
1.  package istia.st.articles.tests;
2.
3.  import istia.st.articles.dao.IDao;
4.  import istia.st.articles.entites.Article;
5.
6.  import java.util.Iterator;
7.  import java.util.List;
8.
9.  import org.springframework.beans.factory.xml.XmlBeanFactory;
10. import org.springframework.core.io.ClassPathResource;
11.
12. public class MainTestDaoFirebird {
13.   public static void main(String[] args) {
14.     // instanciation couche [dao]
```

```
15.     IDao dao = (IDao) (new XmlBeanFactory(new ClassPathResource(
16.         "spring-config-test-dao-firebird.xml"))).getBean("dao");
17.     // liste d'articles actuelle
18.     List<Article> articles = dao.getAll();
19.     // affichage console
20.     Iterator iter = articles.iterator();
21.     while (iter.hasNext()) {
22.       System.out.println(iter.next());
23.     }
24.   }
25. }
```

Instanciation de la couche [dao]

Les lignes 15-16 de la méthode [main] fournissent une référence appelée [dao] sur un objet implémentant l'interface [IDao] de la couche [dao]. On notera bien que nous utilisons ici une interface et non une classe concrète. Cette dernière est fournie par le fichier Spring [spring-config-test-dao-firebird.xml] qui sera cherché dans le ClassPath de l'application (ClassPathResource). Nous savons que par construction, le fichier [spring-config-test-dao-firebird.xml] sera dans le dossier [bin] de notre projet Eclipse, dossier qui fait automatiquement partie du Classpath du projet Eclipse :

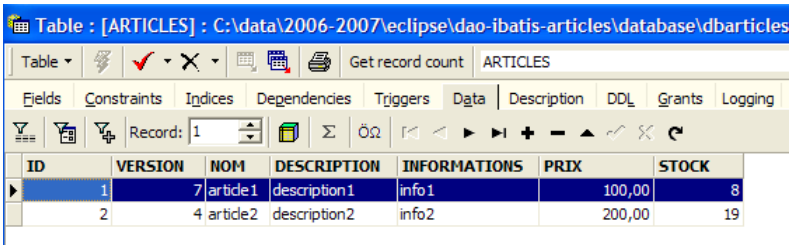


La référence sur la classe implémentant la couche [dao] est fournie par le bean nommé " dao " (getBean(" dao ")) du fichier de configuration Spring [spring-config-test-dao-firebird.xml]. Le contenu de ce dernier a été présenté page 15. Nous en rappelons ci-dessous un court extrait :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.  ...
6.    <!-- SqlMapCllient -->
7.  ...
8.    <!-- la classes d'accè à la couche [dao] -->
9.    <bean id="dao" class="istia.st.articles.dao.DaoImplCommon">
10.     <property name="sqlMapClient">
11.       <ref local="sqlMapClient"/>
12.     </property>
13.   </bean>
14. </beans>
```

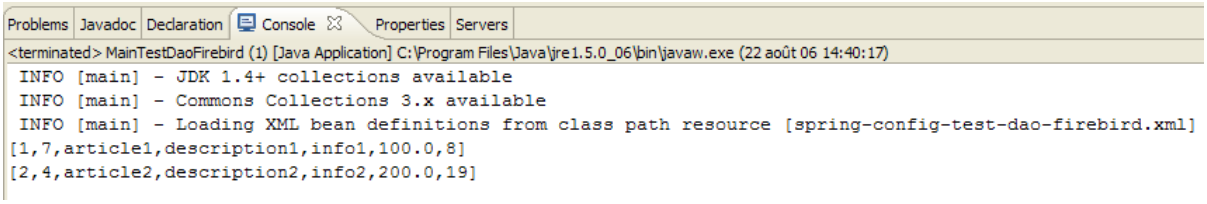
La ligne 9 montre que la méthode [main] du programme de test va obtenir comme référence à la couche [dao], une référence sur un type [DaoImplCommon].

Pour le test, le SGBD Firebird est lancé. Le contenu de la table [ARTICLES] est le suivant :



ID	VERSION	NOM	DESCRIPTION	INFORMATIONS	PRIX	STOCK
1	7	article1	description1	info1	100,00	8
2	4	article2	description2	info2	200,00	19

L'exécution du programme [MainTestDaoFirebird] donne les résultats écran suivants :



On a bien obtenu la liste des articles. On peut passer au test JUnit.

Le test JUnit [TestDaoFirebird] est le suivant :

```
1.  package istia.st.articles.tests;
2.
3.  import java.text.ParseException;
4.  import java.util.Iterator;
5.  import java.util.List;
6.
7.  import org.springframework.beans.factory.xml.XmlBeanFactory;
8.  import org.springframework.core.io.ClassPathResource;
9.  import org.springframework.dao.DataAccessException;
10.
11. import istia.st.articles.dao.DaoException;
12. import istia.st.articles.dao.IDao;
13. import istia.st.articles.entites.Article;
14. import istia.st.articles.entites.InvalidArticleException;
15. import istia.st.articles.entites.RawArticle;
16. import junit.framework.TestCase;
17.
18. public class TestDaoFirebird extends TestCase {
19.
20.  // couche [dao]
21.  private IDao dao;
22.
23.  public IDao getDao() {
24.    return dao;
25.  }
26.
27.  public void setDao(IDao dao) {
28.    this.dao = dao;
29.  }
30.
31.  // initialiseur
32.  public void setUp() {
33.    dao = (IDao) (new XmlBeanFactory(new ClassPathResource(
34.        "spring-config-test-dao-firebird.xml"))).getBean("dao");
35.  }
36.
37.  // liste des articles
38.  private void doListe(List<Article> articles) {
39.    Iterator iter = articles.iterator();
40.    while (iter.hasNext()) {
41.      System.out.println(iter.next());
42.    }
43.  }
44.
45.  // test1
46.  public void test1() throws ParseException {
47.    // liste actuelle
48.    List<Article> articles = dao.getAll();
49.    int nbarticles = articles.size();
50.    // affichage
51.    doListe(articles);
52.    // ajout d'un article
53.    Article a1 = new Article(0, 0L, "X", "X", "X", 100D, 10);
54.    dao.saveOne(a1);
55.    int id1 = a1.getId();
56.    // vérification - on aura un plantage si l'article n'est pas trouvé
57.    a1 = dao.getOne(id1);
58.    assertEquals("X", a1.getNom());
59.    // modification
60.    a1.setNom("Y");
61.    dao.saveOne(a1);
62.    // vérification - on aura un plantage si l'article n'est pas trouvé
63.    a1 = dao.getOne(id1);
64.    assertEquals("Y", a1.getNom());
65.    // suppression
66.    dao.deleteOne(id1);
67.    // vérification
68.    int codeErreur = 0;
69.    boolean erreur = false;
70.    try {
71.      a1 = dao.getOne(id1);
72.    } catch (DaoException ex) {
73.      erreur = true;
74.      codeErreur = ex.getCode();
75.    }
76.    // on doit avoir une erreur de code 1
77.    assertTrue(erreur);
78.    assertEquals(1, codeErreur);
79.    // liste des articles
80.    articles = dao.getAll();
81.    assertEquals(nbarticles, articles.size());
82.  }
```

```
83.
84.  // modification-suppression d'un élément inexistant
85.  public void test2() throws ParseException {
86.    // d'abord ajout d'un article
87.    Article a1 = new Article(0, 0L, "X", "X", "X", 100D, 10);
88.    dao.saveOne(a1);
89.    // récupération copie de l'article
90.    Article a2 = dao.getOne(a1.getId());
91.    // modification copie avec un id inexistant
92.    a2.setId(a1.getId() + 1);
93.    // modification nom de la copie
94.    a2.setNom("Y");
95.    // sauvegarde copie - on doit avoir une DaoException de code 3
96.    boolean erreur = false;
97.    int codeErreur = 0;
98.    try {
99.      dao.saveOne(a2);
100.   } catch (DaoException ex) {
101.     erreur = true;
102.     codeErreur = ex.getCode();
103.   }
104.   // vérification - on doit avoir une erreur de code 3
105.   assertTrue(erreur);
106.   assertEquals(3, codeErreur);
107.   // suppression élément inexistant - on doit avoir une DaoException de
108.   // code 2
109.   erreur = false;
110.   codeErreur = 0;
111.   try {
112.     dao.deleteOne(-2);
113.   } catch (DaoException ex) {
114.     erreur = true;
115.     codeErreur = ex.getCode();
116.   }
117.   // vérification - on doit avoir une erreur de code 2
118.   assertTrue(erreur);
119.   assertEquals(2, codeErreur);
120.   // suppression personne p1
121.   dao.deleteOne(a1.getId());
122.   // vérification
123.   try {
124.     a1 = dao.getOne(a1.getId());
125.   } catch (DaoException ex) {
126.     erreur = true;
127.     codeErreur = ex.getCode();
128.   }
129.   // on doit avoir une erreur de code 1
130.   assertTrue(erreur);
131.   assertEquals(1, codeErreur);
132. }
133.
134. // gestion des versions de personne
135. public void test3() throws ParseException, InterruptedException {
136.   // d'abord ajout
137.   Article a1 = new Article(0, 0L, "X", "X", "X", 100D, 10);
138.   dao.saveOne(a1);
139.   // récupération copie a2 de l'article a1
140.   Article a2 = dao.getOne(a1.getId());
141.   // récupération copie a3 de l'article a1
142.   Article a3 = dao.getOne(a1.getId());
143.   // on vérifie qu'on a bien la même version
144.   assertEquals(a2.getVersion(), a3.getVersion());
145.   // attente 10 ms
146.   Thread.sleep(10);
147.   // sauvegarde copie a2 - la version de a1 va changer
148.   dao.saveOne(a2);
149.   // sauvegarde copie a3 - on doit avoir une DaoException de code 3
150.   boolean erreur = false;
151.   int codeErreur = 0;
152.   try {
153.     dao.saveOne(a3);
154.   } catch (DaoException ex) {
155.     erreur = true;
156.     codeErreur = ex.getCode();
157.   }
158.   // vérification - on doit avoir une erreur de code 3
159.   assertTrue(erreur);
160.   assertEquals(codeErreur, 3);
161.   // suppression article a1
162.   dao.deleteOne(a1.getId());
163.   // vérification
164.   try {
165.     a1 = dao.getOne(a1.getId());
166.   } catch (DaoException ex) {
167.     erreur = true;
168.     codeErreur = ex.getCode();
169.   }
```

```
170.   // on doit avoir une erreur de code 1
171.   assertTrue(erreur);
172.   assertEquals(1, codeErreur);
173. }
174.
175. // optimistic locking - accès multi-threads
176. public void test4() throws Exception {
177.   // ajout d'une personne
178.   Article a1 = new Article(0, 0L, "X", "X", "X", 100D, 0);
179.   dao.saveOne(a1);
180.   int id1 = a1.getId();
181.   // création de N threads de mise à jour du stock
182.   final int N = 100;
183.   Thread[] taches = new Thread[N];
184.   for (int i = 0; i < taches.length; i++) {
185.     taches[i] = new ThreadDaoMajStock("thread n° " + i, dao, id1);
186.     taches[i].start();
187.   }
188.   // on attend la fin des threads
189.   for (int i = 0; i < taches.length; i++) {
190.     taches[i].join();
191.   }
192.   // on récupère l'article a1
193.   a1 = dao.getOne(id1);
194.   // le stock doit être égal à N
195.   assertEquals(N, a1.getStock());
196.   // suppression article a1
197.   dao.deleteOne(a1.getId());
198.   // vérification
199.   boolean erreur = false;
200.   int codeErreur = 0;
201.   try {
202.     a1 = dao.getOne(a1.getId());
203.   } catch (DaoException ex) {
204.     erreur = true;
205.     codeErreur = ex.getCode();
206.   }
207.   // on doit avoir une erreur de code 1
208.   assertTrue(erreur);
209.   assertEquals(1, codeErreur);
210. }
211.
212. // tests de validité de saveOne avec Article
213. public void test5() throws ParseException {
214.   // nom erroné
215.   boolean erreur = false;
216.   try {
217.     dao.saveOne(new Article(0, 0, "", "X", "X", 100D, 10));
218.   } catch (InvalidArticleException ex) {
219.     erreur = true;
220.   }
221.   assertTrue(erreur);
222.   // nom erroné
223.   erreur = false;
224.   try {
225.     dao.saveOne(new Article(0, 0, null, "X", "X", 100D, 10));
226.   } catch (InvalidArticleException ex) {
227.     erreur = true;
228.   }
229.   assertTrue(erreur);
230.   // description erronée
231.   erreur = false;
232.   try {
233.     dao.saveOne(new Article(0, 0, "X", "", "X", 100D, 10));
234.   } catch (InvalidArticleException ex) {
235.     erreur = true;
236.   }
237.   assertTrue(erreur);
238.   erreur = false;
239.   try {
240.     dao.saveOne(new Article(0, 0, "X", null, "X", 100D, 10));
241.   } catch (InvalidArticleException ex) {
242.     erreur = true;
243.   }
244.   assertTrue(erreur);
245.   // informations erronées
246.   erreur = false;
247.   try {
248.     dao.saveOne(new Article(0, 0, "X", "X", null, 100D, 10));
249.   } catch (InvalidArticleException ex) {
250.     erreur = true;
251.   }
252.   assertTrue(erreur);
253.   erreur = false;
254.   try {
255.     dao.saveOne(new Article(0, 0, "X", "X", "", 100D, 10));
256.   } catch (InvalidArticleException ex) {
```

```
257.     erreur = true;
258.   }
259.   assertTrue(erreur);
260.   // prix erroné
261.   erreur = false;
262.   try {
263.     dao.saveOne(new Article(0, 0, "X", "X", "X", -1D, 10));
264.   } catch (InvalidArticleException ex) {
265.     erreur = true;
266.   }
267.   assertTrue(erreur);
268.   // stock erroné
269.   erreur = false;
270.   try {
271.     dao.saveOne(new Article(0, 0, "X", "X", "X", 100D, -1));
272.   } catch (InvalidArticleException ex) {
273.     erreur = true;
274.   }
275.   assertTrue(erreur);
276. }
277.
278. // insertions multi-threads
279. public void test6() throws ParseException, InterruptedException {
280.   // création article
281.   Article a = new Article(0, 0L, "X", "X", "X", 100D, 0);
282.   // qu'on duplique N fois dans un tableau
283.   final int N = 10;
284.   Article[] articles = new Article[N];
285.   for (int i = 0; i < articles.length; i++) {
286.     articles[i] = new Article(a);
287.     articles[i].setNom("X" + i);
288.   }
289.   // création de N threads d'insertion - chaque thread insère 1 personne
290.   Thread[] taches = new Thread[N];
291.   for (int i = 0; i < taches.length; i++) {
292.     taches[i] = new ThreadDaoInsertArticle("thread n° " + i, dao,
293.         articles[i]);
294.     taches[i].start();
295.   }
296.   // on attend la fin des threads
297.   for (int i = 0; i < taches.length; i++) {
298.     // thread n° i
299.     taches[i].join();
300.     // supression personne
301.     dao.deleteOne(articles[i].getId());
302.   }
303. }
304.
305. // tests de validité de saveOne avec RawArticle
306. public void test7() throws ParseException {
307.   // nom erroné
308.   boolean erreur = false;
309.   try {
310.     dao.saveOne(new RawArticle(0, 0, "", "X", "X", 100D, 10));
311.   } catch (DataAccessException ex) {
312.     erreur = true;
313.   }
314.   assertTrue(erreur);
315.   // nom erroné
316.   erreur = false;
317.   try {
318.     dao.saveOne(new RawArticle(0, 0, null, "X", "X", 100D, 10));
319.   } catch (DataAccessException ex) {
320.     erreur = true;
321.   }
322.   assertTrue(erreur);
323.   // description erronée
324.   erreur = false;
325.   try {
326.     dao.saveOne(new RawArticle(0, 0, "X", "", "X", 100D, 10));
327.   } catch (DataAccessException ex) {
328.     erreur = true;
329.   }
330.   assertTrue(erreur);
331.   erreur = false;
332.   try {
333.     dao.saveOne(new RawArticle(0, 0, "X", null, "X", 100D, 10));
334.   } catch (DataAccessException ex) {
335.     erreur = true;
336.   }
337.   assertTrue(erreur);
338.   // informations erronées
339.   erreur = false;
340.   try {
341.     dao.saveOne(new RawArticle(0, 0, "X", "X", null, 100D, 10));
342.   } catch (DataAccessException ex) {
343.     erreur = true;
```

```
344.   }
345.   assertTrue(erreur);
346.   erreur = false;
347.   try {
348.     dao.saveOne(new RawArticle(0, 0, "X", "X", "", 100D, 10));
349.   } catch (DataAccessException ex) {
350.     erreur = true;
351.   }
352.   assertTrue(erreur);
353.   // prix erroné
354.   erreur = false;
355.   try {
356.     dao.saveOne(new RawArticle(0, 0, "X", "X", "X", -1D, 10));
357.   } catch (DataAccessException ex) {
358.     erreur = true;
359.   }
360.   assertTrue(erreur);
361.   // stock erroné
362.   erreur = false;
363.   try {
364.     dao.saveOne(new RawArticle(0, 0, "X", "X", "X", 100D, -1));
365.   } catch (DataAccessException ex) {
366.     erreur = true;
367.   }
368.   assertTrue(erreur);
369. }
370.
371.}
```

Nous ne commenterons que certains points du test et laisserons le lecteur découvrir le reste.

Instanciation de la couche [dao]

La classe de test [TestDaoFirebird] n'a qu'un champ, le champ [dao] de la ligne 21. Ce champ est une référence sur un objet implémentant l'interface [IDao] de la couche [dao]. Cette référence est fournie par la méthode [setUp] (lignes 32-35), méthode exécutée avant chaque test [text**X**]. Le champ privé [dao] de la classe de test a pour valeur le bean " dao " du fichier Spring [spring-config-test-dao-firebird.xml]. Nous obtenons donc la même référence que celle obtenue dans la méthode [main] de la classe [MainTestDaoFirebird] étudiée précédemment, c.a.d. une référence sur un objet de type [DaoImplCommon].

[test1]

[test1] a pour objectif de tester les méthodes de l'interface [IDao] :

- ligne 48 : on demande la liste de tous les articles
- ligne 51 : on les affiche
- lignes 53-54 : on crée un article et on l'ajoute à la table [ARTICLES]
- lignes 57-58 : on vérifie qu'il est bien présent dans la table
- lignes 60-61 : on le modifie et on sauvegarde l'article modifié
- lignes 63-64 : on vérifie que l'article modifié a bien été sauvegardé
- ligne 66 : on le supprime de la table [ARTICLES]
- lignes 70-78 : on vérifie que l'article n'est plus dans la table

---

[test4]

[test4] a pour objectif de tester le comportement de la couche [dao] face à des threads concurrents. Il

- crée et sauvegarde un article avec un stock de 0
- crée 100 threads chargés d'incrémenter simultanément le stock de l'article sauvegardé, de la quantité 1
- attend la fin des 100 threads
- vérifie que le nouveau stock de l'article est de 100

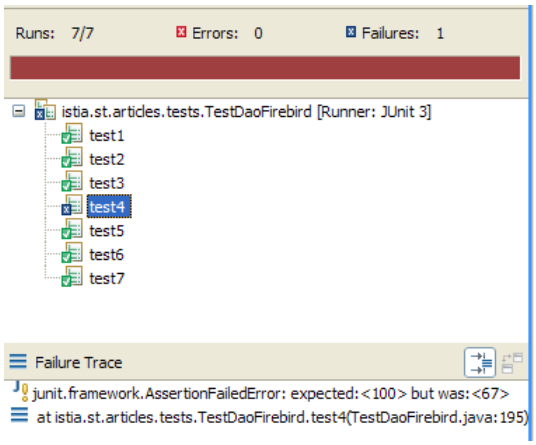
Le code des threads est le suivant :

```
1.  package istia.st.articles.tests;
2.
3.  import java.util.Date;
4.
5.  import istia.st.articles.dao.DaoException;
6.  import istia.st.articles.dao.IDao;
7.  import istia.st.articles.entites.Article;
8.
9.  public class ThreadDaoMajStock extends Thread {
10.  // nom du thread
11.  private String name;
12.
13.  // référence sur la couche [dao]
```

```
14.  private IDao dao;
15.
16.  // id de l'article à modifier
17.  private int idArticle;
18.
19.
20.  // constructeur
21.  public ThreadDaoMajStock(String name, IDao dao, int idArticle) {
22.    this.name = name;
23.    this.dao = dao;
24.    this.idArticle = idArticle;
25.  }
26.
27.  // coeur du thread
28.  public void run() {
29.    // suivi
30.    suivi("lancé");
31.    // on boucle tant qu'on n'a pas réussi à incrémenter de 1
32.    // le stock de l'article idArticle
33.    boolean fini = false;
34.    int stock = 0;
35.    while (!fini) {
36.      // on récupère une copie de l'article
37.      Article article = dao.getOne(idArticle);
38.      // stock
39.      stock=article.getStock();
40.      // suivi
41.      suivi("" + stock + " -> " + (stock + 1)
42.          + " pour la version " + article.getVersion());
43.      // attente de 10 ms pour abandonner le processeur
44.      try {
45.        // suivi
46.        suivi("début attente");
47.        // on s'interrompt pour laisser le processeur
48.        Thread.sleep(10);
49.        // suivi
50.        suivi("fin attente");
51.      } catch (Exception ex) {
52.        throw new RuntimeException(ex.toString());
53.      }
54.      // attente terminée - on essaie de valider la copie
55.      // entre-temps d'autres threads ont pu modifier l'original
56.      int codeErreur = 0;
57.      try {
58.        // incrémente de 1 le stock de la copie
59.        article.setStock(stock + 1);
60.        // on essaie de modifier l'original
61.        dao.saveOne(article);
62.        // on est passé - l'original a été modifié
63.        fini = true;
64.      } catch (DaoException ex) {
65.        // on récupère le code erreur
66.        codeErreur = ex.getCode();
67.        // si une erreur d'ID ou de version de code 3, on réessaie la mise à jour
68.        switch (codeErreur) {
69.        case 3:
70.          suivi("version corrompue ou article inexistant");
71.          break;
72.        default:
73.          // exception non gérée - on laisse remonter
74.          throw ex;
75.        }
76.      }
77.    }
78.    // suivi
79.    suivi("a terminé et passé le stock à " + (stock + 1));
80.  }
81.
82.  // suivi
83.  private void suivi(String message) {
84.    System.out.println(name + " [" + new Date().getTime() + "] : "
85.        + message);
86.  }
87. }
```

- lignes 44-77 : le thread essaie de façon répétée d'incrémenter de 1 le stock de l'article identifié par [idArticle].
- ligne 37 : l'article est lu dans la base
- lignes 43-53 : avant de mettre à jour son stock, on attend 10 ms afin de donner le temps à d'autres threads de lire le même article
- lignes 59-61 : on incrémente le stock de l'article récupé en ligne 37 et on demande la sauvegarde de cet article modifié. Celle-ci ne sera acceptée que si un autre thread n'a pas modifié l'original entre-temps. Si ce n'est pas le cas, la couche [dao] renvoie une exception de type [DaoException] de code 3. On doit alors refaire toute la boucle [lecture, attente, mise à jour].

Le lecteur est invité à lire et comprendre le code des autres tests à partir des fichiers source qui lui seront fournis. Ces tests seront utilisés pour valider la classe [DaoImplCommon]. Aux tests, on obtient les résultats suivants :



- le test [test4] échoue, les autres réussissent.

Le test [test4] échoue donc. Le stock de l'article est à 67 au lieu de 100 attendu. Que s'est-il passé ? Examinons les logs écran. Ils montrent l'existence d'exceptions lancées par Firebird :

```
1.  Exception in thread "Thread-31" org.springframework.jdbc.UncategorizedSQLException: SqlMapClient operation;
    uncategorized SQLException for SQL []; SQL state [HY000]; error code [335544336];
2.  --- The error occurred in articles-firebird.xml.
3.  --- The error occurred while applying a parameter map.
4.  --- Check the Articles.updateOne-InlineParameterMap.
5.  --- Check the statement (update failed).
6.  --- Cause: org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544336. deadlock
7.  update conflicts with concurrent update; nested exception is
    com.ibatis.common.jdbc.exception.NestedSQLException:
8.  --- The error occurred in articles-firebird.xml.
9.  --- The error occurred while applying a parameter map.
10. --- Check the Articles.updateOne-InlineParameterMap.
11. --- Check the statement (update failed).
12.
```

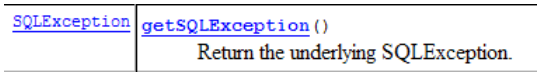
- ligne 1 – on a eu une exception Spring [org.springframework.jdbc.UncategorizedSQLException]. C'est une exception non contrôlée qui a été utilisée pour encapsuler une exception lancée par le pilote JDBC de Firebird, décrite ligne 6.
- ligne 6 – le pilote JDBC de Firebird a lancé une exception de type [org.firebirdsql.jdbc.FBSQLException] et de code d'erreur 335544336.
- ligne 7 : indique qu'on a eu un conflit d'accès entre deux threads qui voulaient mettre à jour en même temps la même ligne de la table [ARTICLES].

Ce n'est pas une erreur irrécupérable. Le thread qui intercepte cette exception peut retenter la mise à jour. Il faut pour cela modifier le code de [ThreadDaoMajStock] :

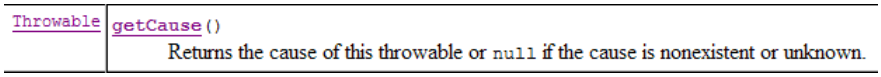
```
1.  try {
2.          // incrémente de 1 le stock de la copie
3.          article.setStock(stock + 1);
4.          // on essaie de modifier l'original
5.          dao.saveOne(article);
6.          // on est passé - l'original a été modifié
7.          fini = true;
8.        } catch (DaoException ex) {
9.          // on récupère le code erreur
10.         codeErreur = ex.getCode();
11.         // si une erreur d'ID ou de version de code 3, on
12.         // réessaie la mise à jour
13.         switch (codeErreur) {
14.         case 3:
15.           suivi("version corrompue ou article inexistant");
16.           break;
17.         default:
18.           // exception non gérée - on laisse remonter
19.           throw ex;
20.         }
21.       }
22.
```

- ligne 8 : on gère une exception de type [DaoException]. D'après ce qui a été dit, il nous faudrait gérer l'exception qui est apparue aux tests, le type [org.springframework.jdbc.UncategorizedSQLException]. On ne peut cependant pas se contenter de gérer ce type qui est un type générique de Spring destiné à encapsuler des exceptions qu'il ne connaît pas. Spring connaît les exceptions émises par les pilotes JDBC d'un certain nombre de SGBD tels Oracle, MySQL, Postgres, DB2, SQL Server, ... mais pas Firebird. Aussi toute exception lancée par le pilote JDBC de Firebird se trouve-t-elle encapsulée dans le type Spring [org.springframework.jdbc.UncategorizedSQLException] :

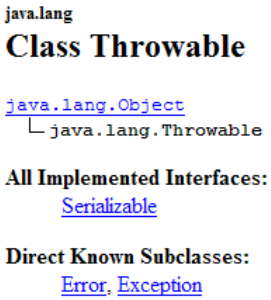
On voit ci-dessus, que la classe [UncategorizedSQLException] dérive de la classe [DataAccessException] qui est le type des exceptions lancées par la classe Spring [SqlMapClientDaoSupport] dont dérive la classe [DaoImplCommon]. Il est possible de connaître l'exception qui a été encapsulée dans [UncategorizedSQLException] grâce à sa méthode [getSQLException] :



Cette exception de type [SQLException] sera ici celle lancée par la couche [iBATIS] qui elle même encapsule l'exception lancée par le pilote JDBC de la base de données. La cause exacte de l'exception de type [SQLException] peut être obtenue par la méthode :



On obtient l'objet de type [Throwable] qui a été lancé par le pilote JDBC :



Le type [Throwable] est la classe parent de [Exception].

D'après les logs du test JUnit, il nous faut vérifier que l'objet de type [Throwable] lancé par le pilote JDBC de Firebird et cause de l'exception [SQLException] lancée par la couche [iBATIS] est bien une exception de type [org.firebirdsql.gds.GDSException] et de code d'erreur 335544336. Pour récupérer le code erreur de l'exception lancée par le pilote JDBC de Firebird, nous pourrons utiliser la méthode [getErrorCode()] de la classe [org.firebirdsql.gds.GDSException].

Si nous utilisons dans le code de [ThreadDaoMajStock] l'exception [org.firebirdsql.gds.GDSException], alors ce thread ne pourra travailler qu'avec le SGBD Firebird. Il en sera de même du test [test4] qui utilise ce thread. Nous voulons éviter cela. En effet, nous souhaitons que nos tests JUnit restent valables quelque soit le SGBD utilisé. Pour arriver à ce résultat, on décide que la couche [dao] lancera une [DaoException] de code 4 lorsqu'une exception de type " conflit de mise à jour " est détectée et ce, quelque soit le SGBD sous-jacent. Ainsi, le thread [ThreadDaoMajStock] pourra-t-il être réécrit comme suit :

```
1.  package istia.st.articles.tests;
2.  ...
3.
```

```
4.  public class ThreadDaoMajStock extends Thread {
5.    // nom du thread
6.    private String name;
7.
8.    // référence sur la couche [dao]
9.    private IDao dao;
10.
11.   // id de l'article à modifier
12.   private int idArticle;
13.
14.   // constructeur
15.   public ThreadDaoMajStock(String name, IDao dao, int idArticle) {
16. ...
17.   }
18.
19.   // coeur du thread
20.   public void run() {
21. ...
22.     while (!fini) {
23.       // on récupère une copie de l'article
24.       Article article = dao.getOne(idArticle);
25.       // stock
26.       stock = article.getStock();
27.       // suivi
28.       suivi("" + stock + " -> " + (stock + 1) + " pour la version "
29.           + article.getVersion());
30.       // attente de 10 ms pour abandonner le processeur
31.       try {
32.         // suivi
33.         suivi("début attente");
34.         // on s'interrompt pour laisser le processeur
35.         Thread.sleep(10);
36.         // suivi
37.         suivi("fin attente");
38.       } catch (Exception ex) {
39.         throw new RuntimeException(ex.toString());
40.       }
41.       // attente terminée - on essaie de valider la copie
42.       // entre-temps d'autres threads ont pu modifier l'original
43.       int codeErreur = 0;
44.       try {
45.         // incrémente de 1 le stock de la copie
46.         article.setStock(stock + 1);
47.         // on essaie de modifier l'original
48.         dao.saveOne(article);
49.         // on est passé - l'original a été modifié
50.         fini = true;
51.       } catch (DaoException ex) {
52.         // on récupère le code erreur
53.         codeErreur = ex.getCode();
54.         // si une erreur d'ID ou de version de code 3 ou un deadlock de
55.         // code 4, on réessaie la mise à jour
56.         switch (codeErreur) {
57.         case 3:
58.           suivi("version corrompue ou article inexistant");
59.           break;
60.         case 4:
61.           suivi("conflit de mise à jour");
62.           break;
63.         default:
64.           // exception non gérée - on laisse remonter
65.           throw ex;
66.         }
67.       }
68.     }
69.     // suivi
70.     suivi("a terminé et passé le stock à " + (stock + 1));
71.   }
72.
73.   // suivi
74.   private void suivi(String message) {
75.     System.out.println(name + " [" + new Date().getTime() + "] : "
76.         + message);
77.   }
78. }
```

- lignes 60-62 : l'exception de type [DaoException] de code 4 est interceptée. Le thread [ThreadDaoMajStock] va être forcé de recommencer la procédure de mise à jour à son début (ligne 10)

Notre couche [dao] doit donc être capable de reconnaître une exception de type " conflit de mise à jour ". Celle-ci est émise par un pilote JDBC et lui est spécifique. Cette exception doit être gérée dans la méthode [updateArticle] de la classe [DaoImplCommon]. Avec le SGBD Firebird, l'exception lancée lors d'un conflit de mise à jour est de type [org.firebirdsql.gds.GDSException] et a comme code d'erreur 335544336. Pour un autre SGBD, l'exception lancée par le pilote JDBC sera d'un autre type. Si on veut garder

un caractère généraliste à la classe [DaoImplCommon], il nous faut la dériver et gérer l'exception de " conflit de mise à jour " dans une classe propre à chaque SGBD. C'est ce que nous faisons maintenant pour le SGBD Firebird.

1.3.3.2 La classe [DaoImplFirebird]

Son code est le suivant :

```
1.  package istia.st.articles.dao;
2.
3.  import istia.st.articles.entites.Article;
4.
5.  public class DaoImplFirebird extends DaoImplCommon {
6.
7.    // modifier un article
8.    protected void updateArticle(Article article) {
9.  ...
10.     }
11.   }
12. ...
13.
14.}
```

- ligne 5 : la classe [DaoImplFirebird] dérive de [DaoImplCommon], la classe que nous venons d'étudier. Elle redéfinit, lignes 8-10, la méthode [updateArticle] qui nous pose problème.

Question : en suivant l'exemple décrit au paragraphe 17.4.2 du document [Les bases de la programmation web MVC en Java], écrire la classe [DaoImplFirebird].

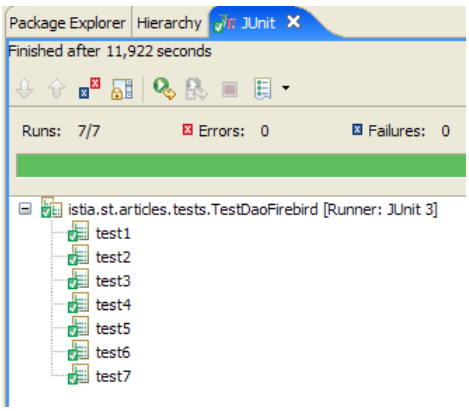
1.3.3.3 Tests de l'implémentation [DaoImplFirebird]

Le fichier de configuration des tests [spring-config-test-dao-firebird.xml] est modifié pour utiliser l'implémentation [DaoImplFirebird] :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.  ...
6.    <!-- SqlMapCllient -->
7.  ...
8.    <!-- la classes d'accè à la couche [dao] -->
9.    <bean id="dao" class="istia.st.articles.dao.DaoImplFirebird">
10.     <property name="sqlMapClient">
11.       <ref local="sqlMapClient"/>
12.     </property>
13.   </bean>
14. </beans>
```

- ligne 9 : la nouvelle implémentation [DaoImplFirebird] de la couche [dao].

Rejouons le test JUnit :



Le test [test4] a été réussi. Les dernières lignes de logs écran sont les suivantes :

```
1.  thread n° 64 [1156257448656] : a terminé et passé le stock à 99
```

```
2.  thread n° 55 [1156257448656] : conflit de mise à jour
3.  thread n° 55 [1156257448656] : 99 -> 100 pour la version 100
4.  thread n° 55 [1156257448656] : début attente
5.  thread n° 55 [1156257448671] : fin attente
6.  thread n° 55 [1156257448671] : a terminé et passé le stock à 100
```

La dernière ligne indique que c'est le thread n° 55 qui a terminé le dernier. La ligne 2 montre un conflit de version qui a forcé le thread n° 55 à reprendre sa procédure de mise à jour de l'article (ligne 3). D'autres logs montrent des conflits d'accès lors des mises à jour :

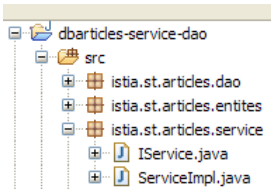
```
1.  thread n° 64 [1156257448625] : version corrompue ou article inexistant
2.  thread n° 64 [1156257448625] : 97 -> 98 pour la version 98
3.  thread n° 64 [1156257448625] : début attente
```

La ligne 1 montre que le thread n° 64 a échoué lors de sa mise à jour parce qu'un autre thread a modifié l'article entre le moment où le thread n° 64 a lu l'article original et celui il va sauvegarder la mise à jour qu'il a faite de cet article. Ce conflit d'accès va obliger le thread n° 64 à retenter sa mise à jour (lignes 2-3).

Nous considèrerons désormais la couche [dao] comme opérationnelle.

1.3.4 Le paquetage [service]

Le paquetage [service] est constitué des classes et interfaces suivantes :



- [IService] est l'interface présentée par la couche [service]
- [ServiceImpl] est une implémentation de celle-ci

L'interface [IService] est la suivante :

```
1.  package istia.st.articles.service;
2.
3.  import istia.st.articles.entites.Article;
4.
5.  import java.util.List;
6.
7.  public interface IService {
8.    // liste de tous les articles
9.    List<Article> getAll();
10.
11.   // obtenir un article particulier
12.   Article getOne(int id);
13.
14.   // ajouter/modifier un article
15.   void saveOne(Article article);
16.
17.   // supprimer un article
18.   void deleteOne(int id);
19.
20.   // ajouter/modifier plusieurs articles
21.   void saveMany(Article[] articles);
22.
23.   // supprimer plusieurs articles
24.   void deleteMany(int ids[]);
25. }
```

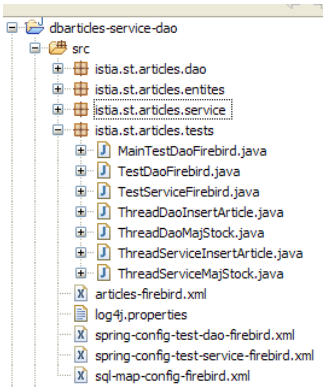
- l'interface a les mêmes quatre méthodes que dans la version 1 mais elle en a deux de plus :
  - **saveMany** : permet de sauvegarder plusieurs articles en même temps de façon atomique. Soit elles sont toutes sauvegardées, soit aucune ne l'est.
  - **deleteMany** : permet de supprimer plusieurs articles en même temps de façon atomique. Soit elles sont toutes supprimées, soit aucune ne l'est.

Les deux méthodes devront être exécutées au sein d'une transaction pour obtenir l'atomicité désirée.

Question : en suivant l'exemple du paragraphe 17.5 du document [Les bases de la programmation web MVC en Java], écrire la classe [ServiceImpl] ainsi que le fichier de configuration Spring qui servira à l'instancier.

1.3.5 Tests de la couche [service]

Maintenant que nous avons écrit et configuré la couche [service], nous nous proposons de la tester avec des tests JUnit :



Le fichier de configuration [spring-config-test-service-firebird.xml] de la couche [service] est celui que vous venez de créer à la question précédente.

Le test JUnit [TestServiceFirebird] est le suivant :

```
1.  package istia.st.articles.tests;
2.
3.  import java.text.ParseException;
4.  import java.util.Iterator;
5.  import java.util.List;
6.
7.  import org.springframework.beans.factory.xml.XmlBeanFactory;
8.  import org.springframework.core.io.ClassPathResource;
9.  import org.springframework.dao.DataAccessException;
10.
11. import istia.st.articles.dao.DaoException;
12. import istia.st.articles.entites.Article;
13. import istia.st.articles.entites.InvalidArticleException;
14. import istia.st.articles.entites.RawArticle;
15. import istia.st.articles.service.IService;
16. import junit.framework.TestCase;
17.
18. public class TestServiceFirebird extends TestCase {
19.
20.   // couche [service]
21.   private IService service;
22.
23.   public IService getService() {
24.     return service;
25.   }
26.
27.   public void setService(IService service) {
28.     this.service = service;
29.   }
30.
31.   // initialiseur
32.   public void setUp() {
33.     service = (IService) (new XmlBeanFactory(new ClassPathResource(
34.         "spring-config-test-service-firebird.xml"))).getBean("service");
35.   }
36.
37.   // liste des articles
38.   private void doListe(List<Article> articles) {
39.     Iterator iter = articles.iterator();
40.     while (iter.hasNext()) {
41.       System.out.println(iter.next());
42.     }
43.   }
44.
45.   // test1
46.   public void test1() throws ParseException {
47.     // liste actuelle
48.     List<Article> articles = service.getAll();
49.     int nbarticles = articles.size();
```

```
50.    // affichage
51.    doListe(articles);
52.    // ajout d'un article
53.    Article a1 = new Article(0, 0L, "X", "X", "X", 100D, 10);
54.    service.saveOne(a1);
55.    int id1 = a1.getId();
56.    // vérification - on aura un plantage si l'article n'est pas trouvé
57.    a1 = service.getOne(id1);
58.    assertEquals("X", a1.getNom());
59.    // modification
60.    a1.setNom("Y");
61.    service.saveOne(a1);
62.    // vérification - on aura un plantage si l'article n'est pas trouvé
63.    a1 = service.getOne(id1);
64.    assertEquals("Y", a1.getNom());
65.    // suppression
66.    service.deleteOne(id1);
67.    // vérification
68.    int codeErreur = 0;
69.    boolean erreur = false;
70.    try {
71.      a1 = service.getOne(id1);
72.    } catch (DaoException ex) {
73.      erreur = true;
74.      codeErreur = ex.getCode();
75.    }
76.    // on doit avoir une erreur de code 1
77.    assertTrue(erreur);
78.    assertEquals(1, codeErreur);
79.    // liste des articles
80.    articles = service.getAll();
81.    assertEquals(nbarticles, articles.size());
82.  }
83.
84.  // modification-suppression d'un élément inexistant
85.  public void test2() throws ParseException {
86. ...
87.  }
88.
89.  // gestion des versions de personne
90.  public void test3() throws ParseException, InterruptedException {
91. ...
92.  }
93.
94.  // optimistic locking - accès multi-threads
95.  public void test4() throws Exception {
96. ...
97.  }
98.
99.  // tests de validité de saveOne
100. public void test5() throws ParseException {
101....
102. }
103.
104. // insertions multi-threads
105. public void test6() throws ParseException, InterruptedException {
106....
107. }
108. // tests de validité de saveOne avec RawArticle
109. public void test7() throws ParseException {
110....
111. }
112.
113.
114. // tests des méthodes saveMany / deleteMany
115. public void test8() throws ParseException {
116.   // liste actuelle
117.   List<Article> articles = service.getAll();
118.   int nbArticles1 = articles.size();
119.   // affichage
120.   doListe(articles);
121.   // création de trois articles
122.   Article a1 = new Article(0, 0L, "X1", "X1", "X1", 100D, 10);
123.   Article a2 = new Article(0, 0L, "X2", "X2", "X2", 200D, 20);
124.   Article a3 = new Article(0, 0L, "X2", "X2", "X2", 300D, 30);
125.   // ajout des 3 articles - l'article a3 avec le nom "X2" va provoquer
126.   // une exception (violation de la contrainte d'unicité du nom)
127.   boolean erreur = false;
128.   try {
129.     service.saveMany(new Article[] { a1, a2, a3 });
130.   } catch (Exception ex) {
131.     erreur = true;
132.     System.out.println(ex.toString());
133.   }
134.   // vérification
135.   assertTrue(erreur);
136.   // nouvelle liste - le nombre d'éléments n'a pas du changer
```

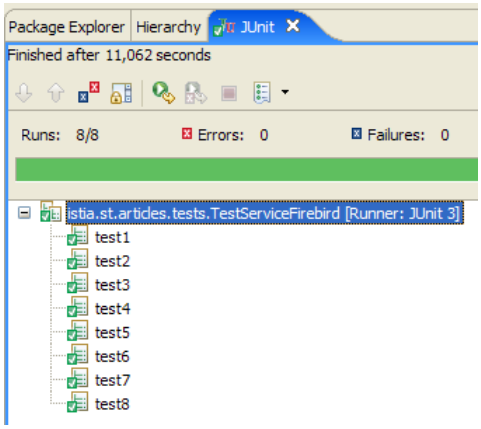
```
137.   // à cause du rollback automatique de la transaction
138.   int nbArticles2 = service.getAll().size();
139.   assertEquals(nbArticles1, nbArticles2);
140.   // ajout des deux articles valides
141.   // on remet leur id à 0 car entre-temps il a changé à cause de leurs
142.   // insertions dans la table
143.   a1.setId(0);
144.   a2.setId(0);
145.   service.saveMany(new Article[] { a1, a2 });
146.   // on récupère leurs id
147.   int id1 = a1.getId();
148.   int id2 = a2.getId();
149.   // vérifications
150.   a1 = service.getOne(id1);
151.   assertEquals(a1.getNom(), "X1");
152.   a2 = service.getOne(id2);
153.   assertEquals(a2.getNom(), "X2");
154.   // nouvelle liste - on doit avoir 2 éléments de +
155.   int nbArticles3 = service.getAll().size();
156.   assertEquals(nbArticles1 + 2, nbArticles3);
157.   // modification des articles
158.   a1.setNom("X");
159.   a2.setNom(a1.getNom());
160.   // persistence en bloc des modifications
161.   // la mise à jour doit échouer à cause de la contrainte d'unicité du nom
162.   // ici, a1 et a2 ont le même nom
163.   erreur = false;
164.   try {
165.     service.saveMany(new Article[] { a1, a2 });
166.   } catch (Exception ex) {
167.     erreur = true;
168.   }
169.   // vérification
170.   assertTrue(erreur);
171.   // on récupère les deux articles dans la base
172.   a1 = service.getOne(id1);
173.   a2 = service.getOne(id2);
174.   // leurs noms n'ont pas du changer
175.   assertEquals("X1", a1.getNom());
176.   assertEquals("X2", a2.getNom());
177.   // nouvelle modification des articles
178.   a1.setStock(1);
179.   a2.setStock(2);
180.   // persistence en bloc des modifications
181.   service.saveMany(new Article[] { a1, a2 });
182.   // vérification
183.   // on récupère les deux articles dans la base
184.   a1 = service.getOne(id1);
185.   a2 = service.getOne(id2);
186.   // on vérifie leurs stocks
187.   assertEquals(1, a1.getStock());
188.   assertEquals(2, a2.getStock());
189.   // suppression de a1 et a2 et d'un article inexistant
190.   // une exception doit se produire
191.   erreur = false;
192.   try {
193.     service.deleteMany(new int[] { id1, id2, 0 });
194.   } catch (Exception ex) {
195.     erreur = true;
196.   }
197.   // vérification
198.   assertTrue(erreur);
199.   // nouvelle liste
200.   articles = service.getAll();
201.   int nbArticles4 = articles.size();
202.   // aucun article n'a du être supprimé (rollback
203.   // automatique de la transaction)
204.   assertEquals(nbArticles4, nbArticles3);
205.   // on supprime les deux articles valides
206.   service.deleteMany(new int[] { id1, id2 });
207.   // vérifications
208.   // article a1
209.   erreur = false;
210.   int codeErreur = 0;
211.   try {
212.     a1 = service.getOne(id1);
213.   } catch (DaoException ex) {
214.     erreur = true;
215.     codeErreur = ex.getCode();
216.   }
217.   // on doit avoir une erreur de code 1
218.   assertTrue(erreur);
219.   assertEquals(1, codeErreur);
220.   // article a2
221.   erreur = false;
222.   codeErreur = 0;
223.   try {
```

```
224.     a2 = service.getOne(id2);
225.   } catch (DaoException ex) {
226.     erreur = true;
227.     codeErreur = ex.getCode();
228.   }
229.   // on doit avoir une erreur de code 1
230.   assertTrue(erreur);
231.   assertEquals(1, codeErreur);
232.   // nouvelle liste
233.   articles = service.getAll();
234.   int nbArticles5 = articles.size();
235.   // vérification - on doit être revenu au point de départ
236.   assertEquals(nbArticles5, nbArticles1);
237.   // affichage
238.   doListe(articles);
239. }
240.}
```

- lignes 21-29 : le programme teste la couche [service] configurée par le fichier [spring-config-test-service-firebird.xml] (lignes 33-34), celui étudié dans la section précédente.
- les tests [test1] à [test7] sont identiques dans leur esprit à leurs homologues de même nom dans la classe de test [TestDaoFirebird] de la couche [dao]. La différence réside dans le fait qu'on teste les méthodes [getAll, getOne, saveOne, deleteOne] de la couche [service] au lieu des méthodes de noms identiques de la couche [dao]. Le code de la méthode [test1] est donné comme exemple.
- la méthode [test8] a pour but de tester les méthodes [saveMany] et [deleteMany]. On veut vérifier qu'elles s'exécutent bien dans une transaction. Commentons le code de cette méthode :

- lignes 117-118 : on compte le nombre d'articles [nbArticles1] actuellement dans la liste
- ligne 120 : on affiche la liste
- lignes 122-124 : on crée trois articles
- lignes 127-133 : ces trois articles sont sauvegardés par la méthode [saveMany] – ligne 129. Les deux premières articles a1 et a2 ayant un id égal à 0 vont être ajoutées à la table [ARTICLES]. La personne a3 a également un id égal à 0 mais un nom égal à celui de l'article a2. L'insertion va échouer car la table [ARTICLES] a une contrainte d'unicité sur sa colonne [NOM]. La couche [dao] va donc lancer une exception qui va remonter jusqu'à la couche [service]. L'existence de cette exception est testée ligne 130.
- à cause de l'exception précédente, la couche [service] devrait faire un [rollback] de l'ensemble des ordres SQL émis pendant l'exécution de la méthode [saveMany], ceci parce que cette méthode s'exécute dans une transaction. Lignes 138-139, on vérifie que le nombre d'articles de la table [ARTICLES] n'a pas bougé et que donc les insertions des articles a1 et a2 n'ont pas eu lieu.
- ligne 145 : on ajoute les seules articles a1 et a2
- lignes 150-153 : on vérifie la présence dans la table des deux articles ajoutés
- lignes 155-156 : on vérifie qu'on a deux articles de plus dans la table.
- lignes 158-159 : on donne le même nom aux deux articles a1 et a2
- lignes 163-168 : on sauvegarde les deux articles ainsi modifiés. Parce que a2 a le même nom que a1, sa mise à jour doit échouer et une exception remonter jusqu'à la couche [service] qui devrait alors faire un [rollback] de l'ensemble des ordres SQL émis pendant l'exécution de la méthode [saveMany], ceci parce que cette méthode s'exécute dans une transaction.
- ligne 170 : on vérifie qu'il y a bien eu exception
- lignes 172-176 : on vérifie que les articles a1 et a2 sont restés inchangés dans la table
- lignes 178-181 : on retente une mise à jour en bloc des articles a1 et a2 en modifiant cette fois leurs stocks.
- lignes 184-188 : on vérifie que les stocks des deux articles ont bien été mis à jour dans la table
- lignes 191-196 : on supprime un groupe d'articles constitué des articles a1 et a2 et d'un article inexistant (id= 0). La méthode [deleteMany] est utilisée pour cela, ligne 193. Cette méthode va échouer car il n'y a aucun article avec un id égal à 0 dans la table [ARTICLES]. La couche [dao] va donc lancer une exception qui va remonter jusqu'à la couche [service]. L'existence de cette exception est testée ligne 198.
- à cause de l'exception précédente, la couche [service] devrait faire un [rollback] de l'ensemble des ordres SQL émis pendant l'exécution de la méthode [deleteMany], ceci parce que cette méthode s'exécute dans une transaction. Lignes 200-204, on vérifie que le nombre d'articles de la liste n'a pas bougé et que donc les suppressions de a1 et a2 n'ont pas eu lieu.
- ligne 206 : on supprime un groupe constitué des seuls articles a1 et a2. Cela devrait réussir. Le reste de la méthode vérifie que c'est bien le cas.

L'exécution des tests donne les résultats suivants :

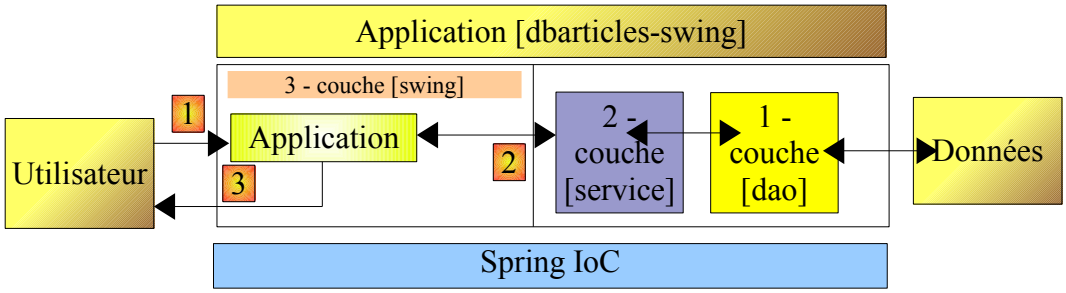


Les tests ont été réussis. Nous considèrerons notre couche [service] comme opérationnelle.

1.4 Implémentation de l'interface graphique

1.4.1 L'architecture de l'application

Rappelons l'architecture à trois couches de l'application à construire :



- la couche [1-dao] s'occupe de l'accès aux données. Celles-ci seront ici placées dans une base de données.
- la couche [2-service] s'occupe des accès transactionnels à la base de données.
- la couche [3-swing] s'occupe de la présentation des données à l'utilisateur et de l'exécution de ses requêtes.
- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces Java
- l'intégration des différentes couches est réalisée par **Spring IoC**

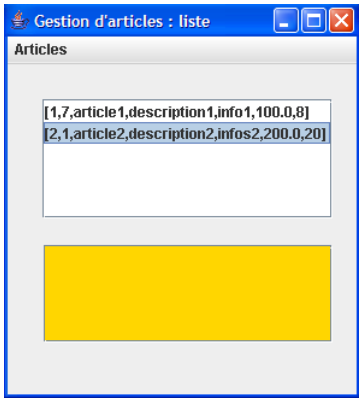
Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

1. le client fait une demande à l'application.
2. l'application traite cette demande. Pour ce faire, elle peut avoir besoin de l'aide de la couche [service] qui elle-même peut avoir besoin de la couche [dao] si des données doivent être échangées avec la base de données.
3. l'application reçoit une réponse de la couche [service]. Selon celle-ci, elle envoie la vue appropriée au client.

Nous avons étudié les couches [1-dao, 2-service]. Nous souhaitons maintenant construire une interface graphique Swing qui viendra se placer dans la couche [3-swing]. Elle aura pour rôle de permettre à un utilisateur de gérer la table des articles (liste, insertion, modification, suppression).

1.4.2 Fonctionnement de l'interface graphique

Au démarrage de l'application graphique, on présentera à l'utilisateur la vue suivante :

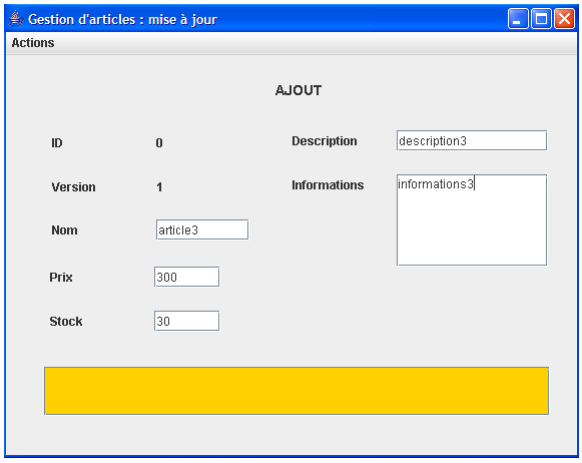


La page affiche la liste des articles de la table [ARTICLES] de la BD.

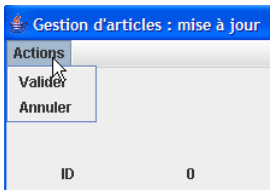
Le menu [Articles] offre plusieurs possibilités :

`Lister`	liste tous les articles de la table
`Ajouter`	fait afficher un formulaire d'ajout
`Modifier`	fait afficher le formulaire de modification de l'article sélectionné dans la liste.
`Supprimer`	supprime de la table l'article sélectionné dans la liste et fait réafficher le nouveau contenu de la table
`Quitter`	pour quitter l'application

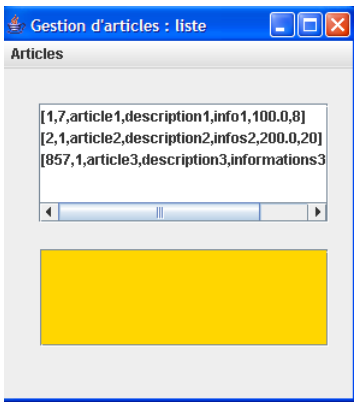
L'option [Ajouter] fait apparaître la vue d'ajout :



Le menu [Actions] permet de valider ou d'invalider l'action en cours :



L'option [Valider] tente d'opérer l'ajout. Si celui-ci réussit, on revient à la page de présentation des articles avec l'article ajouté présent dans la nouvelle liste.

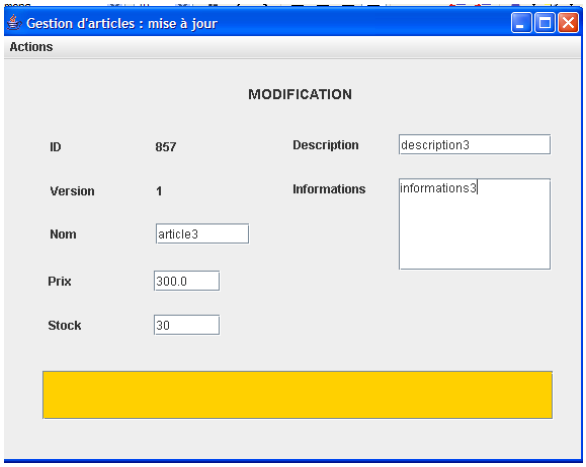


L'option [Annuler] fait simplement revenir à la page de présentation des articles sans opération aucune sur la base des articles.

L'opération [Modification] sur un article sélectionné dans la liste fait afficher la page de modification. Elle est identique à celle d'ajout aux différences suivantes près :

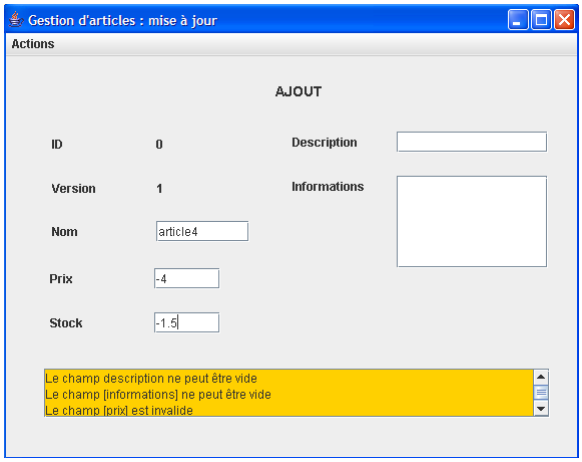
1. les champs sont pré-remplis avec les valeurs de l'article à modifier
2. le champ [id] de l'article ne peut être modifié
3. le titre de la page change

Le menu [Actions] est identique à celui de la page d'ajout.

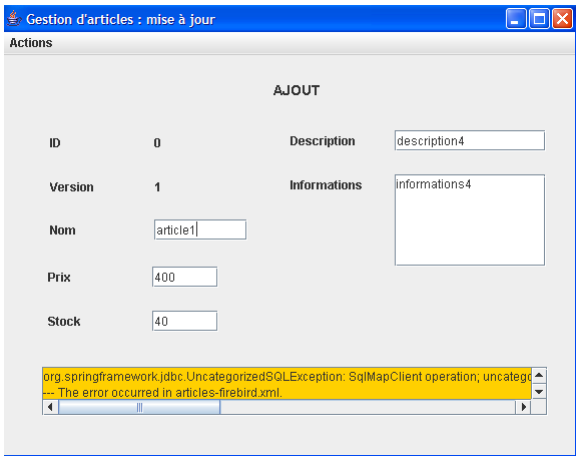


L'opération [Suppression] sur un article sélectionné dans la liste supprime l'article désigné dans la table des articles et rafraîchit la liste présentée à l'utilisateur.

Il y a deux vues dans l'application qu'on appellera vue [Liste] et vue [Ajout-Modification]. Chacune d'elles a un champ pour afficher des messages de réussite ou d'échec. Ainsi si on essaie d'ajouter un article invalide :



Les erreurs peuvent provenir de la base de données. Ainsi, si on essaie d'ajouter un article avec un nom qui existe déjà :



Le message complet ici est :

```
org.springframework.jdbc.UncategorizedSQLException: SqlMapClient operation; uncategorized SQLException for SQL []; SQL
state [HY000]; error code [335544665];
--- The error occurred in articles-firebird.xml.
--- The error occurred while applying a parameter map.
--- Check the Articles.insertOne-InlineParameterMap.
--- Check the statement (update failed).
--- Cause: org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544665. violation of PRIMARY or UNIQUE KEY constraint
"UNQ_NOM_ARTICLES" on table "ARTICLES"; nested exception is com.ibatis.common.jdbc.exception.NestedSQLException:
--- The error occurred in articles-firebird.xml.
--- The error occurred while applying a parameter map.
--- Check the Articles.insertOne-InlineParameterMap.
```

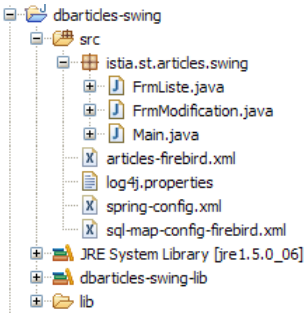
```
--- Check the statement (update failed).
--- Cause: org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544665. violation of PRIMARY or UNIQUE KEY constraint
"UNQ_NOM_ARTICLES" on table "ARTICLES"
Caused by: org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544665. violation of PRIMARY or UNIQUE KEY constraint
"UNQ_NOM_ARTICLES" on table "ARTICLES".
```

Il n'est pas ergonomique mais on peut cependant voir que l'erreur de départ est une violation de la contrainte "UNQ_NOM_ARTICLES" sur la table [ARTICLES]. Cette contrainte interdit les doublons dans la colonne NOM de la table [ARTICLES].

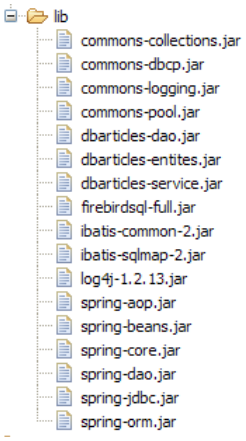
1.4.3 Le projet Eclipse

Le projet Eclipse de l'application [dbarticles-swing] est le suivant :

- le projet complet



- le dossier [lib]



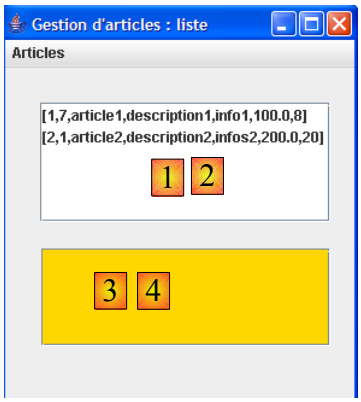
dossier [src]

Le dossier [src] contient les codes source des classes Java de la couche [swing] de l'application ainsi que les fichiers de configuration nécessaires aux couches [service] et [dao]. Le dossier [lib] contient les archives .jar nécessaires aux couches [service] et [dao] ainsi que les archives des trois paquetages utilisés par ces deux couches : [dbarticles-dao.jar, dbarticles-service.jar, dbarticles-entites.jar].

1.4.4 Construire les éléments de l'application graphique

Nous construisons maintenant l'interface graphique avec le plugin [Visual Editor] d'Eclipse. Elle pourrait être construite avec d'autres outils, notamment JBuilder qui, à mon avis, est plus performant.

Les éléments de l'interface graphique de la vue [Liste] sont les suivants :



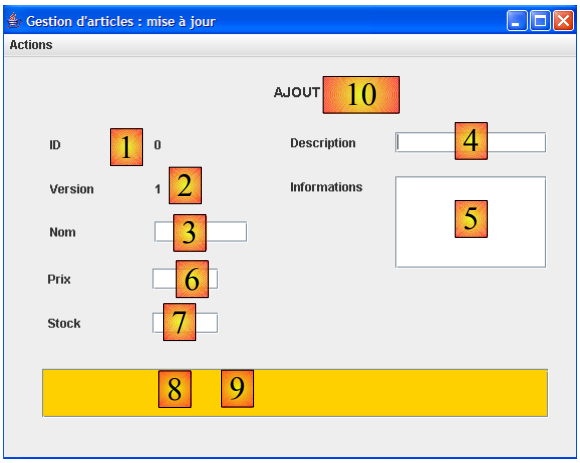
n°	nom	type	rôle	caractéristiques
1	jScrollPaneArticles	JScrollPane	conteneur déroulant de composants	

n°	nom	type	rôle	caractéristiques
2	jListArticles	JList	contient la liste des articles	
3	jScrollPaneMessages	JScrollPane	conteneur déroulant de composants	
4	jTextAreaMessages	JTextArea	contient d'éventuels messages d'erreur	editable=false

Le menu [Articles] contient les composants suivants :

nom	type	rôle
jMenuBar1	JMenuBar	la barre de menus associée à la fenêtre
jMenuArticles	JMenu	le menu [Articles]
jMenuItemLister	JMenuItem	option [Lister]
jMenuItemAjouter	JMenuItem	option [Ajouter]
jMenuItemModifier	JMenuItem	option [Modifier]
jMenuItemSupprimer	JMenuItem	option [Supprimer]
jMenuItemQuitter	JMenuItem	option [Quitter]

Les éléments de l'interface graphique de la vue [Ajout-Modification] sont les suivants :



n°	nom	type	rôle	caractéristiques
1	jLabelIdValue	JTextField	valeur du champ [id] de l'article : 0 pour un ajout	
2	jLabelVersionValue	JLabel	valeur du champ [version] de l'article : 1 pour un ajout	
3	jTextFieldNom	JTextField	saisie du champ [nom] de l'article	
4	jTextFieldDescription	JTextField	saisie du champ [description] de l'article	
5	jTextAreaInformations	JTextArea	saisie du champ [informations] de l'article	
6	jTextFieldPrix	JTextField	saisie du champ [prix] de l'article	
7	jTextFieldStock	JTextField	saisie du champ [stock] de l'article	
8	jScrollPaneMessages	JScrollPane	conteneur déroulant	
9	jTextAreaMessages	JTextArea	zone d'affichage d'éventuels messages d'erreurs	editable=false
10	jLabelAction	JLabel	type de l'action en cours : AJOUT / MODIFICATION	

Le menu [Actions] fait partie de la barre de menu [jMenuBar1] évoquée dans la vue [Liste]. Il contient les composants suivants :

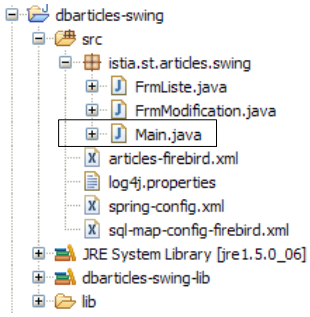
nom	type	rôle
jMenuActions	JMenu	le menu [Actions]
jMenuItemValider	JMenuItem	valide l'action d'ajout ou de modification – si l'action réussit, on revient à la vue [Liste] mise à jour avec la nouvelle liste – si l'action échoue, le message d'erreur est affiché dans [9]
jMenuItemAnnuler	JMenuItem	annule l'action en cours et revient à la vue [Liste]

1.4.5 La logique des vues

Nous avons dans cette application deux vues nommées [Liste] et [Ajout-Modification]. Elles seront toutes deux implémentées par une classe dérivée de JFrame :

- **FrmListe** pour la vue [Liste]
- **FrmModification** pour la vue [Ajout – Modification]

L'application sera lancée via la classe [Main.java] du projet Eclipse :



La classe [Main.java] est la suivante :

```
1.  package istia.st.articles.swing;
2.
3.  import javax.swing.SwingUtilities;
4.
5.  import org.springframework.beans.factory.BeanFactory;
6.  import org.springframework.beans.factory.xml.XmlBeanFactory;
7.  import org.springframework.core.io.ClassPathResource;
8.
9.  public class Main {
10.
11.   /**
12.    * lance l'application graphique
13.    */
14.   public static void main(String[] args) {
15.     SwingUtilities.invokeLater(new Runnable() {
16.       public void run() {
17.         // initialisations environnement de l'application
18.         FrmListe frmListe = null;
19.         FrmModification frmModification = null;
20.         BeanFactory bf;
21.         try {
22.           bf = new XmlBeanFactory(new ClassPathResource("spring-config.xml"));
23.           frmListe = (FrmListe) bf.getBean("frmListe");
24.           frmModification = (FrmModification) bf.getBean("frmModification");
25.         } catch (Exception ex1) {
26.           ex1.printStackTrace();
27.           System.exit(1);
28.         }
29.         // mise en place interfaces graphiques
30.         frmListe.setFrmModification(frmModification);
31.         frmModification.setFrmListe(frmListe);
32.         // affichage vue [Liste]
33.         frmListe.init();
34.         frmListe.setVisible(true);
35.         frmModification.setVisible(false);
36.       }
37.     });
38.   }
39.
40. }
```

- les deux vues sont créées lignes 22-24 à partir de leurs définitions trouvées dans le fichier [spring-config.xml] suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource"
6.      class="org.apache.commons.dbcp.BasicDataSource"
7.      destroy-method="close">
8.      <property name="driverClassName">
9.        <value>org.firebirdsql.jdbc.FBDriver</value>
10.     </property>
11.     <!-- attention : ne pas laisser d'espaces entre les deux balises <value> -->
12.     <property name="url">
13.       <value>jdbc:firebirdsql:localhost/3050:C:/.../dbarticles.gdb</value>
14.     </property>
15.     <property name="username">
16.       <value>sysdba</value>
17.     </property>
18.     <property name="password">
19.       <value>masterkey</value>
20.     </property>
21.   </bean>
22.   <!-- SqlMapCllient -->
23.   <bean id="sqlMapClient"
24.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
25.     <property name="dataSource">
26.       <ref local="dataSource" />
27.     </property>
28.     <property name="configLocation">
29.       <value>classpath:sql-map-config-firebird.xml</value>
30.     </property>
31.   </bean>
32.   <!-- la classes d'accè à la couche [dao] -->
33.   <bean id="dao" class="istia.st.articles.dao.DaoImplFirebird">
34.     <property name="sqlMapClient">
35.       <ref local="sqlMapClient" />
36.     </property>
37.   </bean>
38.   <!-- gestionnaire de transactions -->
39.   <bean id="transactionManager"
40.     class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
41.     <property name="dataSource">
42.       <ref local="dataSource" />
43.     </property>
44.   </bean>
45.   <!-- la classes d'accès à la couche [service] -->
46.   <bean id="service"
47.     class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
48.     <property name="transactionManager">
49.       <ref local="transactionManager" />
50.     </property>
51.     <property name="target">
52.       <bean class="istia.st.articles.service.ServiceImpl">
53.         <property name="dao">
54.           <ref local="dao" />
55.         </property>
56.       </bean>
57.     </property>
58.     <property name="transactionAttributes">
59.       <props>
60.         <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
61.         <prop key="save*">PROPAGATION_REQUIRED</prop>
62.         <prop key="delete*">PROPAGATION_REQUIRED</prop>
63.       </props>
64.     </property>
65.   </bean>
66.   <!--  la couche swing -->
67.   <bean id="frmListe" class="istia.st.articles.swing.FrmListe">
68.     <property name="service">
69.       <ref local="service" />
70.     </property>
71.   </bean>
72.   <bean id="frmModification"
73.     class="istia.st.articles.swing.FrmModification">
74.     <property name="service">
75.       <ref local="service" />
76.     </property>
77.   </bean>
78.
79. </beans>
```

- les lignes 1-65 du fichier ci-dessus servent à configurer les couches [service] et [dao] de l'application. Nous les avons rencontrées lors de l'étude de ces couches.
- les lignes 67-77 configurent la couche [swing]

- lignes 67-71 : création d'une instance [FrmListe] pour la vue [Liste]
- lignes 72-77 : création d'une instance [FrmModification] pour la vue [Ajout – Modification]
- les deux vues ont besoin d'accéder à la couche [service] pour exécuter certaines des actions demandées par l'utilisateur. Aussi leur injecte-t-on à toutes les deux, une référence sur cette couche (lignes 68-70 et 74-76)

Revenons au code de la classe [Main.java] chargée de lancer l'application graphique :

- lignes 25-28 : si l'exploitation du fichier de configuration [spring-config.xml] lance une exception, on écrit celle-ci sur la console et on s'arrête.
- chaque vue a besoin d'une référence sur l'autre. En effet,
  - les options [Ajout, Modification] de la vue [Liste] font passer à la vue [Ajout – Modification]
  - les options du menu [Actions] de la vue [Ajout – Modification] font revenir à la vue [Liste]
- ligne 30 : on donne à la vue [Liste] une référence sur la vue [Ajout – Modification]
- ligne 31 : on donne à la vue [Ajout – Modification] une référence sur la vue [Liste]
- ligne 33 : on demande à la vue [Liste] de s'initialiser. Il s'agit pour la vue [Liste] de demander la liste des articles à la couche [service] pour les afficher.
- ligne 34 : la vue [Liste] est affichée
- ligne 35 : la vue [Ajout – Modification] est cachée

Le squelette de la classe [FrmListe] est le suivant :

```
1.  package istia.st.articles.swing;
2.
3.  import istia.st.articles.entites.Article;
4.  import istia.st.articles.service.IService;
5.
6.  ...
7.
8.  public class FrmListe extends JFrame {
9.
10.  // ----------------------------------------
11.  // code non généré
12.  // ----------------------------------------
13.
14.  // service de l'architecture 3 couches
15.  private IService service = null;
16.
17.  public void setService(IService service) {
18.    this.service = service;
19.  }
20.
21.  // formulaire de modification
22.  private FrmModification frmModification;
23.
24.  public void setFrmModification(FrmModification frmModification) {
25.    this.frmModification = frmModification;
26.  }
27.
28.  // modèle du composant jListArticles
29.  private DefaultListModel vArticles = null;
30.
31.  // init interface
32.  public void init() {
33.    // on affiche la liste des articles
34.    doLister();
35.  }
36.
37.  // ajout d'un article
38.  protected void doAjouter() {
39. ...
40.  }
41.
42.  // modification d'un article
43.  protected void doModifier() {
44. ...
45.  }
46.
47.  // liste des articles
48.  protected void doLister() {
49. ...
50.  }
51.
52.  // suppression de l'article sélectionné dans la liste
53.  protected void doSupprimer() {
54. ...
55.  }
56.
57.  // ----------------------------------------
58.  // code généré
59.  // ----------------------------------------
```

```
60.  private static final long serialVersionUID = 1L;
61.
62.  private JPanel jContentPane = null;
63.
64.  private JMenuBar jJMenuBar = null;
65.
66. ....
```

La vue [Liste] est construite visuellement avec Eclipse qui génère lui-même le code Java correspondant. Celui-ci commence ligne 63. Ce code est insuffisant pour répondre à notre besoin de communication entre les classes [Main, FrmListe, FrmModification] d'une part et avec la couche [service] d'autre part.

- les lignes 14 – 19 définissent le champ [service] qui sera une référence sur la couche [service] et qui sera initialisé par Spring au démarrage.
- les lignes 22 – 26 définissent le champ [frmModification] qui sera une référence sur la vue [Modification - Ajout] et qui sera initialisé par la classe [Main] au démarrage.
- les lignes 28 – 55 forment le code de gestion des événements de la vue.

On retrouve une structure analogue pour la vue [Ajout – Modification] :

```
1.  package istia.st.articles.swing;
2.
3.  import istia.st.articles.dao.DaoException;
4.  import istia.st.articles.entites.Article;
5.  import istia.st.articles.entites.InvalidArticleException;
6.  import istia.st.articles.service.IService;
7.  import org.springframework.dao.DataAccessException;
8.
9.  ...
10.
11. public class FrmModification extends JFrame {
12.
13.  // ----------------------------------------
14.  // code non généré
15.  // ----------------------------------------
16.
17.  // service de l'architecture 3 couches
18.  private IService service = null;
19.
20.  public void setService(IService service) {
21.    this.service = service;
22.  }
23.
24.  // formulaire de liste
25.  private FrmListe frmListe;
26.
27.  // setter
28.  public void setFrmListe(FrmListe frmListe) {
29.    this.frmListe = frmListe;
30.  }
31.
32.  // l'article géré par le formulaire
33.  private Article article;
34.
35.  public void setArticle(Article article) {
36.    this.article = article;
37.  }
38.
39.  // init interface
40.  public void init(String action) {
41.    // préparation de l'interface
42.    if ("ajout".equals(action)) {
43. ...
44.    } else {
45.      if ("modification".equals(action)) {
46. ...
47.      }
48.    }
49. ...
50.  }
51.
52.  // annulation d'un ajout ou modification
53.  protected void doAnnuler() {
54. ...
55.  }
56.
57.  // validation d'une modification
58.  protected void doValider() {
59. ....
60.  }
61.
62.  // ----------------------------------------
63.  // code généré
64.  // ----------------------------------------
```

```
65.
66.  private static final long serialVersionUID = 1L;
67.
68.  // interface graphique
69.
70.  private JPanel jContentPane = null;
71.
72.  private JLabel jLabelAction = null;
```

- les lignes 17 – 22 définissent le champ [service] qui sera une référence sur la couche [service] et qui sera initialisé par Spring au démarrage.
- les lignes 25 – 30 définissent le champ [frmListe] qui sera une référence sur la vue [Liste] et qui sera initialisé par la classe [Main] au démarrage.
- les lignes 33 - 37 définissent le champ [article] qui sera une référence sur l'article à modifier. C'est la vue [Liste] qui initialisera ce champ lors d'une modification.
- les lignes 39 – 60 forment le code de gestion des événements de la vue.

Les vues sont implémentées par des fenêtres JFrame que l'utilisateur peut fermer. Le comportement par défaut du gestionnaire de cet événement est d'arrêter l'application. Ce n'est pas souhaitable. En effet, si l'utilisateur ferme la vue [Ajout – Modification], il faut éventuellement revenir à la vue [Liste] mais pas arrêter l'application. Pour éviter ce problème, on inhibera le comportement par défaut d'une fenêtre qui se ferme en écrivant :

```
this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

où [this] désigne la classe de la vue.

Les vues [Liste] et [Ajout-Modification] seront associées à des gestionnaires de disposition (Layout Managers) "absolus", c.a.d. que les éléments de l'interface seront placés à un endroit précis de l'interface et ne peuvent en bouger. C'est en général déconseillé pour les raisons suivantes :

1. cela nuit à la portabilité entre machines. Selon les systèmes d'exploitation sous-jacents, les polices de caractères utilisées n'ont pas les mêmes caractéristiques. Ainsi, le texte d'un bouton créé sous Windows 2000, peut s'avérer déborder du bouton sous Windows XP.
2. cela nuit à l'internationalisation de l'application. Une interface graphique peut être construite de telle façon qu'elle prenne ses libellés dans un fichier de ressources. Ainsi on peut avoir un fichier de libellés en français, un autre en allemand, ... Ainsi un logiciel peut se vendre dans différents pays. Seulement les libellés, d'une langue à l'autre, n'auront pas le même nombre de caractères. Ainsi un codage en dur de la position et taille des composants créé alors qu'on travaillait en français peut s'avérer désastreux lorsqu'on passe en allemand.

Il existe des gestionnaires de disposition permettant aux composants d'adapter leurs taille et position au moment de l'exécution en fonction de la taille de la fenêtre dans laquelle ils sont ou des libellés qu'ils doivent afficher. C'est cette voie qu'il faudrait suivre. Ici, nous choisirons une voie de facilité plutôt qu'une voie professionnelle.

1.5 Travail à faire

Construire l'interface graphique décrite précédemment.

2 L'application [boutique-web]

Nous souhaitons donner ici quelques éléments d'une application web de commerce électronique. Celle-ci permettra à des clients du web :

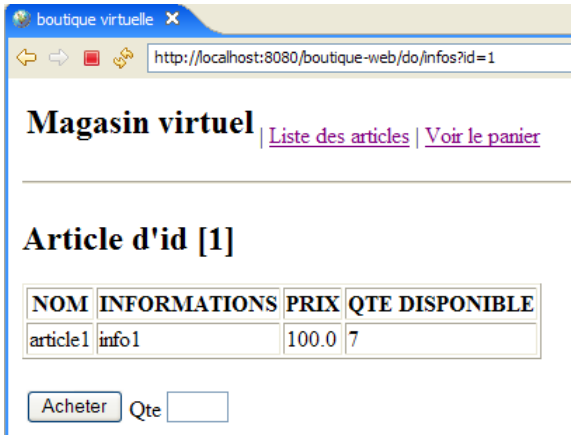
- de consulter une liste d'articles provenant d'une base de données
- d'en mettre certains dans un panier électronique
- de valider celui-ci. Cette validation aura pour seul effet de mettre à jour, dans la base, les stocks des articles achetés.

2.1 L'interface web

Les différentes vues présentées à l'utilisateur seront les suivantes :

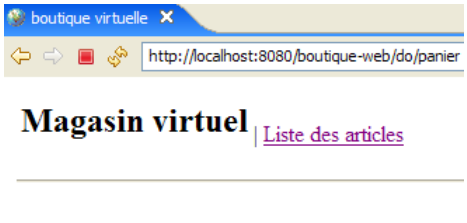
- la vue [LISTE] qui présente une liste des articles en vente

- la vue [INFOS] qui donne des informations supplémentaires sur un produit :



- la vue [PANIER] qui donne le contenu du panier du client

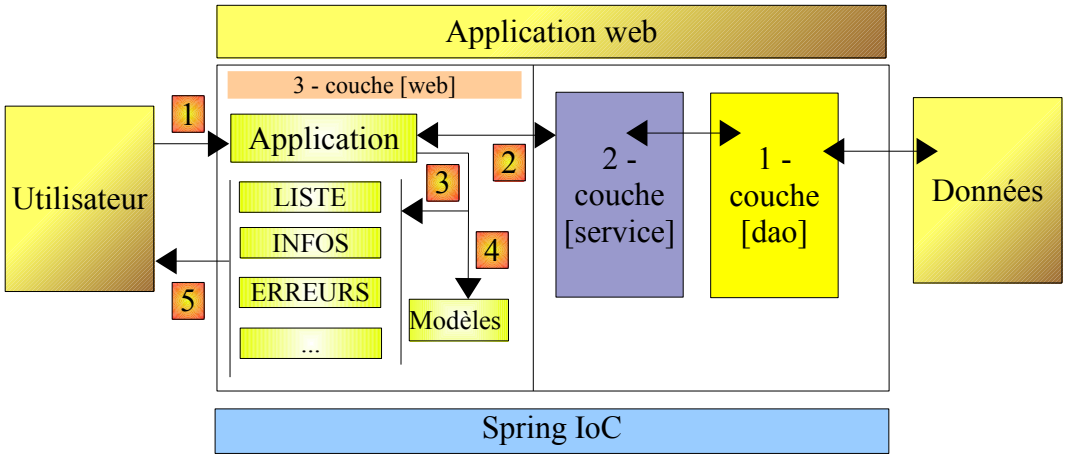
- la vue [PANIERVIDE] pour le cas où le panier du client est vide



- la vue [ERREURS] qui signale toute erreur de l'application

2.2 Architecture générale de l'application

L'application aura l'architecture à trois couches suivante :



- la couche [1-dao] s'occupe de l'accès aux données. Celles-ci seront ici placées dans une base de données.
- la couche [2-service] s'occupe de la gestion du panier du client ainsi que des accès transactionnels à la base de données.
- la couche [3-web] s'occupe de la présentation des données à l'utilisateur et de l'exécution de ses requêtes.
- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces Java
- l'intégration des différentes couches est réalisée par **Spring IoC**
- la couche de présentation [3-web] implémente une architecture MVC (Modèle – Vue – Contrôleur)
  - C : c'est la servlet [Application] qui traite toutes les demandes de l'utilisateur.
  - V : c'est l'ensemble des pages JSP qui génèrent la réponse envoyée au client, ici un flux HTML.
  - M : c'est l'ensemble des informations affichées par les différentes vues. C'est la définition que nous utiliserons ici. Pour d'autres auteurs, le M est le modèle de l'application qui a été implémenté par les couches [service] et [dao].

Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

1. le client fait une demande au contrôleur C. Ce contrôleur est ici la servlet [Application] qui voit passer toutes les demandes des clients.
2. le contrôleur traite cette demande. Pour ce faire, il peut avoir besoin de l'aide de la couche [service] qui elle-même peut avoir besoin de la couche [dao] si des données doivent être échangées avec la base de données.
3. le contrôleur reçoit une réponse de la couche [service]. La demande du client a été traitée. Celle-ci peut appeler plusieurs réponses possibles. Un exemple classique est
   - une page d'erreurs si la demande n'a pu être traitée correctement
   - une page de confirmation sinon
4. le contrôleur choisit la réponse (= vue) à envoyer au client. Celle-ci est le plus souvent une page contenant des éléments dynamiques. Le contrôleur fournit ceux-ci à la vue. C'est ce que nous appelons dans ce texte, le modèle M de la vue.
5. la vue est envoyée au client. C'est le V de MVC.

2.3 L'existant

Nous utiliserons un certain nombre d'éléments de l'application [dbarticles-swing] développée précédemment :

- la base de données des articles en vente décrite au paragraphe , page .
- les archives .jar implémentant les interfaces et classes suivantes :

`dbarticles-entites.jar`	**classes** : Article, RawArticle, InvalidArticleException
`dbarticles-dao.jar`	**interface** : IDao **classes** : DaoImplCommon, DaoImplFirebird, DaoException
`dbarticles-service.jar`	**interface** : IService **classes** : ServiceImpl

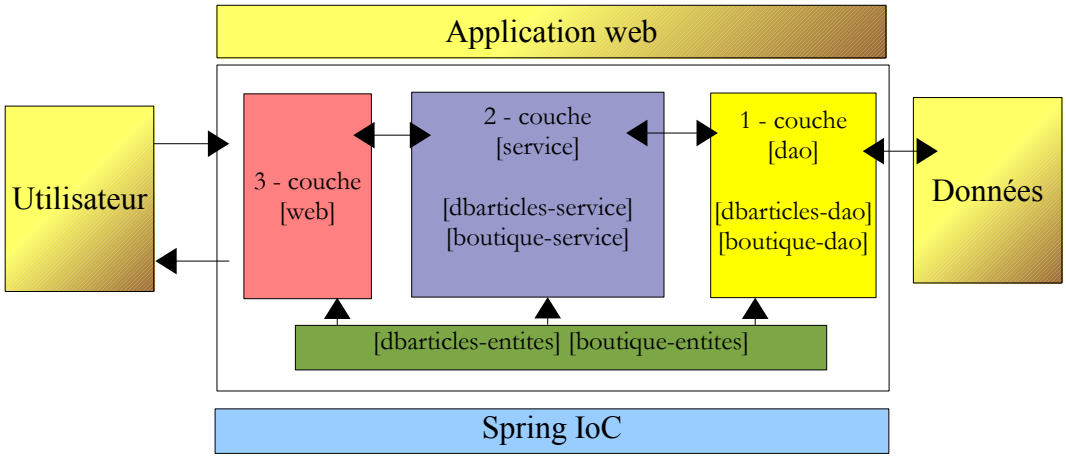
2.4 Les couches [dao] et [service] d'accès aux articles

2.4.1 Architecture de l'application

Nous allons tout d'abord développer les couches [dao] et [service] de la boutique en ligne. Ces deux couches permettront à l'application web de faire les opérations suivantes :

- obtenir tous les articles disponibles pour les présenter à l'acheteur
- obtenir les détails d'un article particulier choisi par l'utilisateur
- modifier les stocks des articles achetés

L'architecture de ces couches au sein de l'architecture générale sera la suivante :

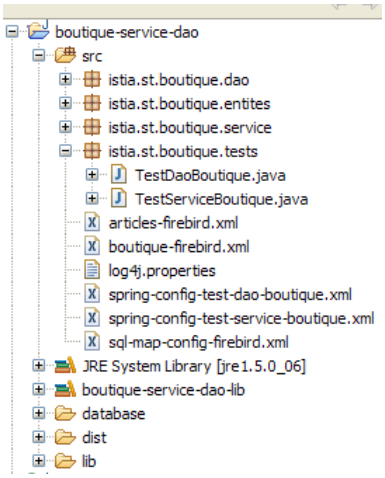


On notera que les couches [service] et [dao] intègrent l'existant : les archives [dbarticles-dao], [dbarticles-service] et [dbarticles-entites] développées précédemment. L'application [boutique-web] amène des éléments supplémentaires :

- [boutique-entites] regroupe les nouveaux objets nécessités par la gestion de la boutique virtuelle
- [boutique-dao] implémente de nouvelles requêtes aux données
- [boutique-service] implémente les fonctionnalités transactionnelles de l'application ainsi qu'une partie métier.

2.4.2 Le projet Eclipse

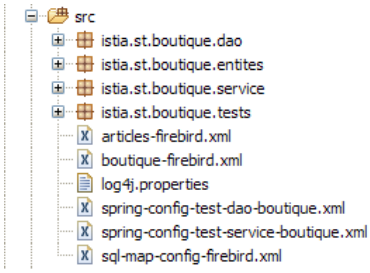
Pour réaliser cette architecture, nous utiliserons le projet Eclipse [boutique-service-dao] suivant :

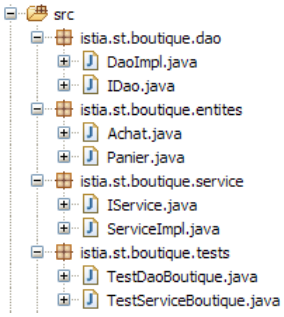


Le projet est un projet Java.

Dossier [src]

Ce dossier contient les codes source des couches [dao] et [service] ainsi que les classes de tests de ces couches :





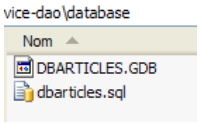
On y trouve différents paquetages :

- [istia.st.boutique.entites] : contient les classes [Achat] et [Panier] qui représentent respectivement un achat fait par le client et l'ensemble de ces achats réunis dans un panier.
- [istia.st.boutique.dao] : contient la couche [dao] qui est en contact avec le SGBD. Elle va permettre :
  - d'obtenir une liste simplifiée des articles disponibles pour les présenter à l'utilisateur
  - d'obtenir les détails d'un article particulier
  - de décrémenter les stocks des articles achetés par un utilisateur
- [istia.st.articles.service] : contient la couche [service] qui va assurer les transactions sur la base de données.
- [istia.st.articles.tests] : contient les tests JUnit des couches [dao] et [service]

ainsi que des fichiers de configuration qui doivent être dans le *ClassPath* de l'application.

Dossier [database]

Ce dossier contient la base de données Firebird des articles :

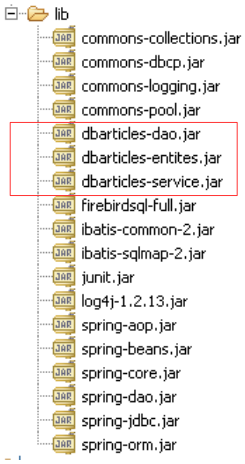


- [dbarticles.gdb] est la base de données.
- [dbarticles.sql] est le script SQL de génération de la base

Ces éléments ont été décrits au paragraphe , page .

Dossier [lib]

Ce dossier contient les archives nécessaires à l'application :



On retrouve toutes les archives utilisées dans l'application [dbarticles-swing] et décrites au paragraphe , page , ainsi que les trois archives [dbarticles-dao.jar, dbarticles-entites.jar, dbarticles-service.jar] contenant les couches [service] et [dao] de l'application [dbarticles-swing].

Dossier [dist]

Ce dossier contiendra les archives issues de la compilation des classes de l'application :

- [boutique-dao.jar] : archive du paquetage [dao]
- [boutique-service.jar] : archive du paquetage [service]
- [boutique-entites.jar] : archive du paquetage [entites]

2.4.3 Le paquetage [entites]

`Achat`	modélise l'achat d'un client (article, quantité)
`Panier`	rassemble les achats d'un client

La classe [Achat] représente un achat du client :

```
1.  package istia.st.boutique.entites;
2.
3.  import istia.st.articles.entites.Article;
4.
5.  public class Achat {
6.
7.    // article acheté
8.    private Article article;
9.
10.   // qté achetée
11.   private int qte;
12.
13.   // total à payer
14.   public double getTotal() {
15.     return article.getPrix() * qte;
16.   }
17.
18.   // constructeur par défaut
19.   public Achat() {
20.
21.   }
22.
23.   // constructeur avec arguments
24.   public Achat(Article article, int qte) {
25.     setArticle(article);
26.     setQte(qte);
27.   }
28.
29.   // getters and setters
30. ...
31.
32.   // toString
33.   public String toString() {
34.     return "[" + this.article + "," + this.qte + "]";
35.   }
36.}
```

`article`	l'article acheté
`qte`	la quantité achetée
`double getTotal()`	rend le montant de l'achat
`String toString()`	chaîne d'identité de l'objet

La classe [Panier] représente l'ensemble des achats du client :

```
1.  package istia.st.boutique.entites;
2.
3.  ...
4.  // panier d'achats
```

```
5.  public class Panier {
6.
7.    // liste des achats
8.    private List<Achat> achats = new ArrayList<Achat>();
9.
10.   // ajoute un achat
11.   public void ajouter(Achat unAchat) {
12. ...
13.   }
14.
15.   // enlever un article acheté
16.   public void enlever(int idArticle) {
17. ...
18.   }
19.
20.   // vider le panier
21.   public void clear(){
22. ...
23.   }
24.
25.   // valeur du panier
26.   public double getTotal() {
27. ...
28.   }
29.
30.   // toString
31.   public String toString() {
32.     return this.achats.toString();
33.   }
34.
35.   // getters / setters
36.   public List<Achat> getAchats() {
37.     return achats;
38.   }
39.
40.   public void setAchats(List<Achat> achats) {
41.     this.achats = achats;
42.   }
43.
44. }
```

`List<Achat> achats`	la liste des achats du client - liste d'objets de type [Achat]
`void ajouter(Achat unAchat)`	ajoute un achat à la liste des achats. Si l'article acheté est déjà présent dans le panier, on met à jour la quantité achetée de celui-ci. Dans le panier, on ne trouve donc pas deux fois le même article.
`void enlever(int idArticle)`	enlève l'achat de l'article idArticle
`void clear()`	vide le panier de tous ses achats
`double getTotal()`	rend le montant total des achats
`String toString()`	rend la chaîne d'identité du panier
`List<Achat> getAchats()`	rend la liste des achats

Question : écrire la classe [Panier].

2.4.4 Le paquetage [dao]

2.4.4.1 Les éléments de la couche [dao]

Le paquetage [dao] est formé des classes et interfaces suivantes :

- [IDao] est l'interface présentée par la couche [dao]
- [DaoImpl] est une implémentation de celle-ci

L'interface [IDao] est la suivante :

```
1.  package istia.st.boutique.dao;
```

```
2.
3.  import java.util.List;
4.
5.  import istia.st.articles.entites.Article;
6.  import istia.st.boutique.entites.Panier;
7.
8.  public interface IDao {
9.    // obtenir tous les articles dont le stock est # de 0
10.   public List<Article> getAllArticlesDisponibles();
11.   // valider le panier du client
12.   public void acheter(Panier panier);
13. }
```

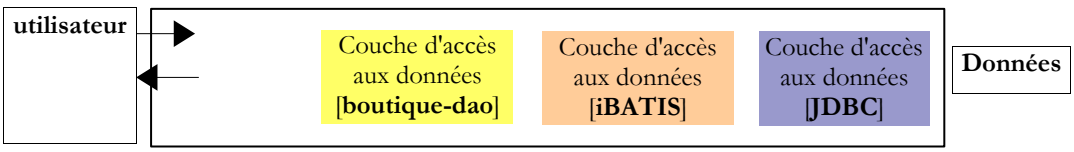
`getAllArticlesDisponibles`	rend la liste des articles disponibles avec pour chacun d'eux les informations [ID, VERSION, NOM, DESCRIPTION, PRIX]. Les articles dont le stock est nul ne font pas partie de la liste.
`acheter(Panier panier)`	passe en revue tous les articles du panier et pour chacun d'eux, décrémente dans la table [ARTICLES] le stock de l'article acheté de la quantité achetée. Cette opération peut échouer si le stock actuel de l'article est inférieur à la quantité demandée. Dans ce cas, le stock de l'article reste inchangé et une exception est lancée.

L'interface [istia.st.boutique.IDao] fait remonter des exceptions non contrôlées, dérivées de [RuntimeException]. Dans l'implémentation [[DaoImpl] que nous allons en faire, nous utiliserons des exceptions du type [istia.st.articles.DaoException] défini au paragraphe , page .

Le squelette de la classe d'implémentation [DaoImpl.java] sera le suivant :

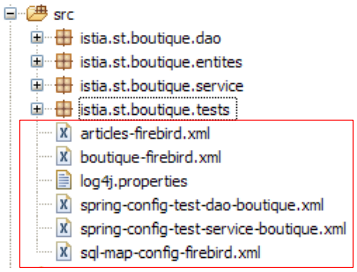
```
1.  package istia.st.boutique.dao;
2.
3.  ...
4.
5.  public class DaoImpl extends SqlMapClientDaoSupport implements IDao {
6.
7.    // achat du panier
8.    public void acheter(Panier panier) {
9.  ....
10.   }
11.
12.   public List<Article> getAllArticlesDisponibles() {
13. ...
14.   }
15. }
```

- ligne 5 : la classe dérive de la classe [SqlMapClientDaoSupport] du framework Spring. Elle utilise le framework [Ibatis] pour accéder au SGBD :



2.4.4.2 Configuration de la couche [dao]

La couche [dao] est configurée par les fichiers suivants du Classpath de l'application :



spring-config-test-dao-boutique.xml

C'est la configuration Spring de la couche [dao] :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource"
6.      class="org.apache.commons.dbcp.BasicDataSource"
7.      destroy-method="close">
8.      <property name="driverClassName"
9.        value="org.firebirdsql.jdbc.FBDriver" />
10.     <property name="url"
11.       value="jdbc:firebirdsql:localhost/3050:C:\...\dbarticles.gdb" />
12.     <property name="username" value="sysdba" />
13.     <property name="password" value="masterkey" />
14.   </bean>
15.   <!-- SqlMapCllient -->
16.   <bean id="sqlMapClient"
17.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
18.     <property name="dataSource">
19.       <ref local="dataSource" />
20.     </property>
21.     <property name="configLocation">
22.       <value>classpath:sql-map-config-firebird.xml</value>
23.     </property>
24.   </bean>
25.   <!-- la classes d'accè à la couche [dao] -->
26.   <bean id="daoArticles"
27.     class="istia.st.articles.dao.DaoImplFirebird">
28.     <property name="sqlMapClient">
29.       <ref local="sqlMapClient" />
30.     </property>
31.   </bean>
32.   <!-- la classes d'accès à la couche [dao] de la boutique -->
33.   <bean id="daoBoutique" class="istia.st.boutique.dao.DaoImpl">
34.     <property name="sqlMapClient">
35.       <ref local="sqlMapClient" />
36.     </property>
37.   </bean>
38. </beans>
```

Le lecteur est invité à relire les explications données précédemment pour un fichier analogue au paragraphe 1.3.2.2, page 15.

- ligne 16 : définit le bean implémentant la couche [Ibatis]
- ligne 26 : définit le bean implémentant la couche [dao] étudiée précédemment : [istia.st.articles.dao.DaoImplFirebird]
- ligne 33 : définit le bean implémentant la couche [dao] spécifique à la gestion de la boutique. Cette couche utilise la classe [istia.st.boutique.dao.DaoImpl] que nous venons de présenter.
- lignes 28 et 33 : les deux couches [dao] utilisent le même bean [Ibatis]

La ligne 22 indique que la couche [Ibatis] est configurée par le fichier [sql-map-config-firebird.xml].

---

sql-map-config-firebird.xml

Ce fichier est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.  <!DOCTYPE sqlMapConfig
3.      PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
4.      "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
5.
6.  <sqlMapConfig>
7.    <sqlMap resource="articles-firebird.xml"/>
8.    <sqlMap resource="boutique-firebird.xml"/>
9.  </sqlMapConfig>
```

- ligne 7 : [articles-firebird.xml] décrit les ordres SQL permettant de gérer la table des articles. Ce fichier a déjà été utilisé et présenté au paragraphe , page .
- ligne 8 : [boutique-firebird.xml] décrit les nouveaux ordres SQL nécessaires à la gestion de la boutique

---

boutique-firebird.xml

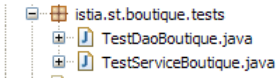
Le contenu de ce fichier est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.
3.  <!DOCTYPE sqlMap
4.      PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
```

```
5.      "http://www.ibatis.com/dtd/sql-map-2.dtd">
6.
7.  <sqlMap>
8.    <!-- alias classe [Article] -->
9.    <typeAlias alias="Article.classe"
10.     type="istia.st.articles.entites.Article" />
11.   <!-- mapping table [ARTICLES] - objet [Article] -->
12.   <resultMap id="ArticlePartiel.map" class="Article.classe">
13.     <result property="id" column="ID" />
14.     <result property="version" column="VERSION" />
15.     <result property="nom" column="NOM" />
16.     <result property="description" column="DESCRIPTION" />
17.     <result property="prix" column="PRIX" />
18.   </resultMap>
19.   <!-- liste de certaines colonnes de tous les articles-->
20.   <select id="Boutique.getAllArticlesDisponibles" resultMap="ArticlePartiel.map">
21.     select ID, VERSION, NOM, DESCRIPTION, PRIX FROM ARTICLES WHERE STOCK>0
22.   </select>
23.   <!-- modifier le stock d'un article -->
24.   <update id="Boutique.changeStock" parameterClass="java.util.Hashtable">
25.     UPDATE ARTICLES SET STOCK=STOCK-#qte#, VERSION=VERSION+1 WHERE ID=#id#
26.   </update>
27. </sqlMap>
```

- ligne 21 : l'ordre SQL qui va permettre d'obtenir la liste des articles disponibles à la vente
- ligne 25 : l'ordre SQL qui va permettre de décrémenter le stock d'un article du panier de la quantité achetée. Cet ordre peut lancer une exception. En effet si l'opération [UPDATE] amène le stock à être négatif, la contrainte d'intégrité sur la colonne STOCK de la table ARTICLES ne sera plus vérifiée (cf contrainte CHK_STOCK_ARTICLES page , ligne 27) et le pilote JDBC du SGBD va lancer une exception d'un type généralement spécifique au pilote JDBC.

2.4.5 Tests de la couche [dao]



La classe [TestDaoBoutique] est un test JUnit chargé de tester la couche [dao] de la boutique. Son code est le suivant :

```
1.  package istia.st.boutique.tests;
2.
3.  import istia.st.articles.dao.DaoException;
4.  import istia.st.articles.entites.Article;
5.  import istia.st.boutique.entites.Achat;
6.  import istia.st.boutique.entites.Panier;
7.
8.  import java.util.List;
9.  import junit.framework.TestCase;
10.
11. import org.springframework.beans.factory.BeanFactory;
12. import org.springframework.beans.factory.xml.XmlBeanFactory;
13. import org.springframework.core.io.ClassPathResource;
14.
15. // test de la classe ArticlesDaoSqlMap
16. public class TestDaoBoutique extends TestCase {
17.
18.   // une instance de la couche [dao] de la boutique
19.   private istia.st.boutique.dao.IDao daoBoutique;
20.
21.   // une instance de la couche [dao] de la BD articles
22.   private istia.st.articles.dao.IDao daoArticles;
23.
24.   public TestDaoBoutique() throws Exception {
25.     // on récupère la configuration du test
26.     BeanFactory bf = new XmlBeanFactory(new ClassPathResource(
27.         "spring-config-test-dao-boutique.xml"));
28.     // récupère une instance de la couche dao de la boutique
29.     daoBoutique = (istia.st.boutique.dao.IDao) bf.getBean("daoBoutique");
30.     // récupère une instance de la couche dao de la BD articles
31.     daoArticles = (istia.st.articles.dao.IDao) bf.getBean("daoArticles");
32.   }
33.
34.   // affichage console liste d'articles
35.   private void doListe(List<Article> articles) {
36.     for (Article article : articles) {
37.       System.out.println(article);
38.     }
39.   }
40.
41.   // liste des articles disponibles
42.   public void test0() {
```

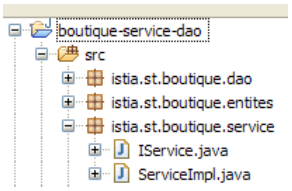
```
43.     doListe(daoBoutique.getAllArticlesDisponibles());
44.   }
45.
46.   // achat d'un panier
47.   public void test1() {
48.     // création de deux articles
49.     Article a1 = new Article(0, 0L, "X1", "X1", "X1", 100D, 10);
50.     Article a2 = new Article(0, 0L, "X2", "X2", "X2", 200D, 20);
51.     // persistence des deux articles
52.     daoArticles.saveOne(a1);
53.     daoArticles.saveOne(a2);
54.     // création d'un panier
55.     Panier panier = new Panier();
56.     // premier achat
57.     panier.ajouter(new Achat(a1, 1));
58.     // vérifications
59.     List<Achat> achats = panier.getAchats();
60.     assertEquals(1, achats.size());
61.     Achat achat = achats.get(0);
62.     assertEquals("X1", achat.getArticle().getNom());
63.     assertEquals(1, achat.getQte());
64.     assertEquals(100.0, panier.getTotal(), 1e-6);
65.     // nouvel achat du même article
66.     panier.ajouter(new Achat(a1, 1));
67.     // vérifications
68.     achats = panier.getAchats();
69.     assertEquals(1, achats.size());
70.     achat = achats.get(0);
71.     assertEquals("X1", achat.getArticle().getNom());
72.     assertEquals(2, achat.getQte());
73.     assertEquals(200.0, panier.getTotal(), 1e-6);
74.     // achat article a2
75.     panier.ajouter(new Achat(a2, 1));
76.     // vérifications
77.     achats = panier.getAchats();
78.     assertEquals(2, achats.size());
79.     achat = achats.get(1);
80.     assertEquals("X2", achat.getArticle().getNom());
81.     assertEquals(1, achat.getQte());
82.     assertEquals(400.0, panier.getTotal(), 1e-6);
83.     // validation panier
84.     daoBoutique.acheter(panier);
85.     // vérifications
86.     // recherche article a1 dans la BD
87.     a1 = daoArticles.getOne(a1.getId());
88.     // le stock doit être de 8
89.     assertEquals(8, a1.getStock());
90.     // recherche article a2 dans la BD
91.     a2 = daoArticles.getOne(a2.getId());
92.     // le stock doit être de 19
93.     assertEquals(19, a2.getStock());
94.     // suppression des articles ajoutés en début de test
95.     daoArticles.deleteOne(a1.getId());
96.     daoArticles.deleteOne(a2.getId());
97.   }
98.
99.   // échec d'un achat de panier
100.  public void test2() {
101.    // création de deux articles
102.    Article a1 = new Article(0, 0L, "X1", "X1", "X1", 100D, 10);
103.    Article a2 = new Article(0, 0L, "X2", "X2", "X2", 200D, 20);
104.    // persistence des deux articles
105.    daoArticles.saveOne(a1);
106.    daoArticles.saveOne(a2);
107.    // création d'un panier
108.    Panier panier = new Panier();
109.    // achats
110.    panier.ajouter(new Achat(a1, 1));
111.    panier.ajouter(new Achat(a2, 21));
112.    // validation panier
113.    DaoException daoException = null;
114.    boolean erreur = false;
115.    try {
116.      daoBoutique.acheter(panier);
117.    } catch (DaoException ex) {
118.      erreur = true;
119.      daoException = ex;
120.    }
121.    // vérifications
122.    // une erreur a du se produire car le stock de a2 (20) est insuffisant
123.    // pour l'achat (21)
124.    assertTrue(erreur);
125.    // l'exception récupérée doit avoir le code 100
126.    int codeErreur = daoException.getCode();
```

```
127.   assertEquals(100, codeErreur);
128.   // la liste des exceptions internes doit avoir un élément
129.   List<DaoException> daoExceptions = daoException.getDaoExceptions();
130.   assertEquals(1, daoExceptions.size());
131.   // la 1ère exception de la liste doit avoir le code 20
132.   codeErreur = daoExceptions.get(0).getCode();
133.   assertEquals(20, codeErreur);
134.   // recherche article a1 dans la BD
135.   a1 = daoArticles.getOne(a1.getId());
136.   // le stock doit être passé à 9
137.   assertEquals(9, a1.getStock());
138.   // recherche article a2 dans la BD
139.   a2 = daoArticles.getOne(a2.getId());
140.   // le stock doit être resté à 20
141.   assertEquals(20, a2.getStock());
142.   // suppression des articles ajoutés en début de test
143.   daoArticles.deleteOne(a1.getId());
144.   daoArticles.deleteOne(a2.getId());
145. }
146.
147. // retirer des achats du panier
148. public void test3() {
149.   // création de trois articles
150.   Article a1 = new Article(1, 1L, "X1", "X1", "X1", 100D, 10);
151.   Article a2 = new Article(2, 1L, "X2", "X2", "X2", 200D, 20);
152.   Article a3 = new Article(3, 1L, "X3", "X3", "X3", 300D, 30);
153.   // création d'un panier
154.   Panier panier = new Panier();
155.   // achats
156.   panier.ajouter(new Achat(a1, 1));
157.   panier.ajouter(new Achat(a2, 1));
158.   panier.ajouter(new Achat(a3, 1));
159.   panier.ajouter(new Achat(a1, 1));
160.   panier.ajouter(new Achat(a2, 1));
161.   panier.ajouter(new Achat(a3, 1));
162.   // vérifications
163.   // liste des achats
164.   List<Achat> achats = panier.getAchats();
165.   // il doit y avoir 3 articles #
166.   assertEquals(3, achats.size());
167.   // retirer l'achat a2
168.   panier.enlever(a2.getId());
169.   // vérifications
170.   // liste des achats
171.   achats = panier.getAchats();
172.   // il doit y avoir 2 articles #
173.   assertEquals(2, achats.size());
174.   // le 1er article acheté est (X1,2)
175.   Achat achat = achats.get(0);
176.   assertEquals("X1", achat.getArticle().getNom());
177.   assertEquals(2, achat.getQte());
178.   // le second article acheté est (X3,2)
179.   achat = achats.get(1);
180.   assertEquals("X3", achat.getArticle().getNom());
181.   assertEquals(2, achat.getQte());
182. }
183.
184.}
```

Question : écrire une classe [istia.st.boutique.dao.DaoImpl] (cf page 51) qui satisfasse aux tests ci-dessus.

2.4.6 Le paquetage [service]

Le paquetage [service] de l'application [boutique-web] est constitué des classes et interfaces suivantes :



- [IService] est l'interface présentée par la couche [service]
- [ServiceImpl] est une implémentation de celle-ci

L'interface [IService] est la suivante :

```
1.  package istia.st.boutique.service;
2.
3.  import java.util.List;
4.
5.  import istia.st.articles.entites.Article;
6.  import istia.st.boutique.entites.Panier;
7.
8.  public interface IService {
9.    // liste des articles disponibles
10.  public List<Article> getAllArticlesDisponibles();
11.
12.  // validation des achats du panier
13.  public void acheter(Panier panier);
14. }
```

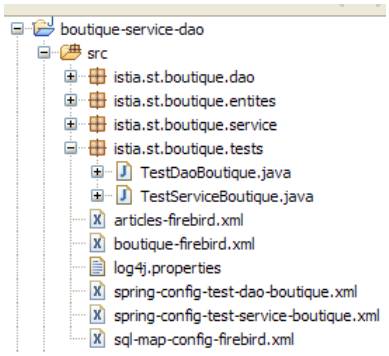
- l'interface a les mêmes deux méthodes que l'interface de la couche [dao]
- ligne 13 : la méthode [acheter] valide de façon atomique les achats du panier. Soit tous les achats sont validés (les stocks des articles achetés sont suffisants), soit aucun.

L'implémentation [istia.st.boutique.service.ServiceImpl] de l'interface [istia.st.boutique.service.IService] travaillant avec un SGBD, placera l'exécution de la méthode [acheter] au sein d'une transaction afin d'assurer l'achat de type tout ou rien souhaité.

Question : en suivant l'exemple du paragraphe 17.5 du document [Les bases de la programmation web MVC en Java], écrire la classe [istia.st.boutique.service.ServiceImpl] ainsi que le fichier de configuration Spring qui servira à l'instancier.

2.4.7 Tests de la couche [service]

Maintenant que nous avons écrit et configuré la couche [service], nous nous proposons de la tester avec des tests JUnit :



Le fichier de configuration [spring-config-test-service-boutique.xml] de la couche [service] est celui qui vient d'être présenté.

Le test JUnit [TestServiceFirebird] est le suivant :

```
1.  package istia.st.boutique.tests;
2.
3.  import istia.st.articles.dao.DaoException;
4.  import istia.st.articles.entites.Article;
5.  import istia.st.boutique.entites.Achat;
6.  import istia.st.boutique.entites.Panier;
7.
8.  import java.util.List;
9.  import junit.framework.TestCase;
10.
11. import org.springframework.beans.factory.BeanFactory;
12. import org.springframework.beans.factory.xml.XmlBeanFactory;
13. import org.springframework.core.io.ClassPathResource;
14.
15. // test de la classe ArticlesDaoSqlMap
16. public class TestServiceBoutique extends TestCase {
17.
18.  // une instance de la couche service de la boutique
19.  private istia.st.boutique.service.IService serviceBoutique;
20.
21.  // une instance de la couche service de la BD articles
22.  private istia.st.articles.service.IService serviceArticles;
23.
24.  public TestServiceBoutique() throws Exception {
25.    // on récupère la configuration du test
```

```
26.    BeanFactory bf = new XmlBeanFactory(new ClassPathResource(
27.        "spring-config-test-service-boutique.xml"));
28.    // récupère une instance de la couche service de la boutique
29.    serviceBoutique = (istia.st.boutique.service.IService) bf
30.        .getBean("serviceBoutique");
31.    // récupère une instance de la couche service de la BD articles
32.    serviceArticles = (istia.st.articles.service.IService) bf
33.        .getBean("serviceArticles");
34.  }
35.
36.  // affichage console liste d'articles
37.  private void doListe(List<Article> articles) {
38.    for (Article article : articles) {
39.      System.out.println(article);
40.    }
41.  }
42.
43.  // liste des articles disponibles
44.  public void test0() {
45.    doListe(serviceBoutique.getAllArticlesDisponibles());
46.  }
47.
48.  // achat d'un panier
49.  public void test1() {
50.    // création de deux articles
51.    Article a1 = new Article(0, 0L, "X1", "X1", "X1", 100D, 10);
52.    Article a2 = new Article(0, 0L, "X2", "X2", "X2", 200D, 20);
53.    // persistence des deux articles
54.    serviceArticles.saveMany(new Article[] { a1, a2 });
55.    // création d'un panier
56.    Panier panier = new Panier();
57.    // premier achat
58.    panier.ajouter(new Achat(a1, 1));
59.    // vérifications
60.    List<Achat> achats = panier.getAchats();
61.    assertEquals(1, achats.size());
62.    Achat achat = achats.get(0);
63.    assertEquals("X1", achat.getArticle().getNom());
64.    assertEquals(1, achat.getQte());
65.    assertEquals(100.0, panier.getTotal(), 1e-6);
66.    // nouvel achat du même article
67.    panier.ajouter(new Achat(a1, 1));
68.    // vérifications
69.    achats = panier.getAchats();
70.    assertEquals(1, achats.size());
71.    achat = achats.get(0);
72.    assertEquals("X1", achat.getArticle().getNom());
73.    assertEquals(2, achat.getQte());
74.    assertEquals(200.0, panier.getTotal(), 1e-6);
75.    // achat article a2
76.    panier.ajouter(new Achat(a2, 1));
77.    // vérifications
78.    achats = panier.getAchats();
79.    assertEquals(2, achats.size());
80.    achat = achats.get(1);
81.    assertEquals("X2", achat.getArticle().getNom());
82.    assertEquals(1, achat.getQte());
83.    assertEquals(400.0, panier.getTotal(), 1e-6);
84.    // validation panier
85.    serviceBoutique.acheter(panier);
86.    // vérifications
87.    // le panier doit être vide
88.    achats = panier.getAchats();
89.    assertEquals(0, achats.size());
90.    // recherche article a1 dans la BD
91.    a1 = serviceArticles.getOne(a1.getId());
92.    // le stock doit être de 8
93.    assertEquals(8, a1.getStock());
94.    // recherche article a2 dans la BD
95.    a2 = serviceArticles.getOne(a2.getId());
96.    // le stock doit être de 19
97.    assertEquals(19, a2.getStock());
98.    // suppression des articles ajoutés en début de test
99.    serviceArticles.deleteMany(new int[] { a1.getId(), a2.getId() });
100. }
101.
102. // échec d'un achat de panier
103. public void test2() {
104.   // création de deux articles
105.   Article a1 = new Article(0, 0L, "X1", "X1", "X1", 100D, 10);
106.   Article a2 = new Article(0, 0L, "X2", "X2", "X2", 200D, 20);
107.   // persistence des deux articles
108.   serviceArticles.saveMany(new Article[] { a1, a2 });
109.   // création d'un panier
110.   Panier panier = new Panier();
```

```
111.   // achats
112.   panier.ajouter(new Achat(a1, 1));
113.   panier.ajouter(new Achat(a2, 21));
114.   // validation panier
115.   DaoException daoException = null;
116.   boolean erreur = false;
117.   try {
118.     serviceBoutique.acheter(panier);
119.   } catch (DaoException ex) {
120.     erreur = true;
121.     daoException = ex;
122.   }
123.   // vérifications
124.   // une erreur a du se produire car le stock de a2 (20) est insuffisant
125.   // pour l'achat (21)
126.   assertTrue(erreur);
127.   // l'exception récupérée doit avoir le code 100
128.   int codeErreur = daoException.getCode();
129.   assertEquals(100, codeErreur);
130.   // la liste des exceptions internes doit avoir un élément
131.   List<DaoException> daoExceptions = daoException.getDaoExceptions();
132.   assertEquals(1, daoExceptions.size());
133.   // la 1ère exception de la liste doit avoir le code 20
134.   codeErreur = daoExceptions.get(0).getCode();
135.   assertEquals(20, codeErreur);
136.   // le panier doit être inchangé
137.   List<Achat> achats = panier.getAchats();
138.   assertEquals(2, achats.size());
139.   Achat achat = achats.get(0);
140.   assertEquals("X1", achat.getArticle().getNom());
141.   assertEquals(10, achat.getArticle().getStock());
142.   assertEquals(1, achat.getQte());
143.   achat = achats.get(1);
144.   assertEquals("X2", achat.getArticle().getNom());
145.   assertEquals(20, achat.getArticle().getStock());
146.   assertEquals(21, achat.getQte());
147.   // recherche article a1 dans la BD
148.   a1 = serviceArticles.getOne(a1.getId());
149.   // le stock doit être resté à 10
150.   assertEquals(10, a1.getStock());
151.   // recherche article a2 dans la BD
152.   a2 = serviceArticles.getOne(a2.getId());
153.   // le stock doit être resté à 20
154.   assertEquals(20, a2.getStock());
155.   // suppression des articles ajoutés en début de test
156.   serviceArticles.deleteMany(new int[] { a1.getId(), a2.getId() });
157. }
158.
159. // retirer des achats du panier
160. public void test3() {
161.   // création de trois articles
162.   Article a1 = new Article(1, 1L, "X1", "X1", "X1", 100D, 10);
163.   Article a2 = new Article(2, 1L, "X2", "X2", "X2", 200D, 20);
164.   Article a3 = new Article(3, 1L, "X3", "X3", "X3", 300D, 30);
165.   // création d'un panier
166.   Panier panier = new Panier();
167.   // achats
168.   panier.ajouter(new Achat(a1, 1));
169.   panier.ajouter(new Achat(a2, 1));
170.   panier.ajouter(new Achat(a3, 1));
171.   panier.ajouter(new Achat(a1, 1));
172.   panier.ajouter(new Achat(a2, 1));
173.   panier.ajouter(new Achat(a3, 1));
174.   // vérifications
175.   // liste des achats
176.   List<Achat> achats = panier.getAchats();
177.   // il doit y avoir 3 articles #
178.   assertEquals(3, achats.size());
179.   // retirer l'achat a2
180.   panier.enlever(a2.getId());
181.   // vérifications
182.   // liste des achats
183.   achats = panier.getAchats();
184.   // il doit y avoir 2 articles #
185.   assertEquals(2, achats.size());
186.   // le 1er article acheté est (X1,2)
187.   Achat achat = achats.get(0);
188.   assertEquals("X1", achat.getArticle().getNom());
189.   assertEquals(2, achat.getQte());
190.   // le second article acheté est (X3,2)
191.   achat = achats.get(1);
192.   assertEquals("X3", achat.getArticle().getNom());
193.   assertEquals(2, achat.getQte());
194. }
```

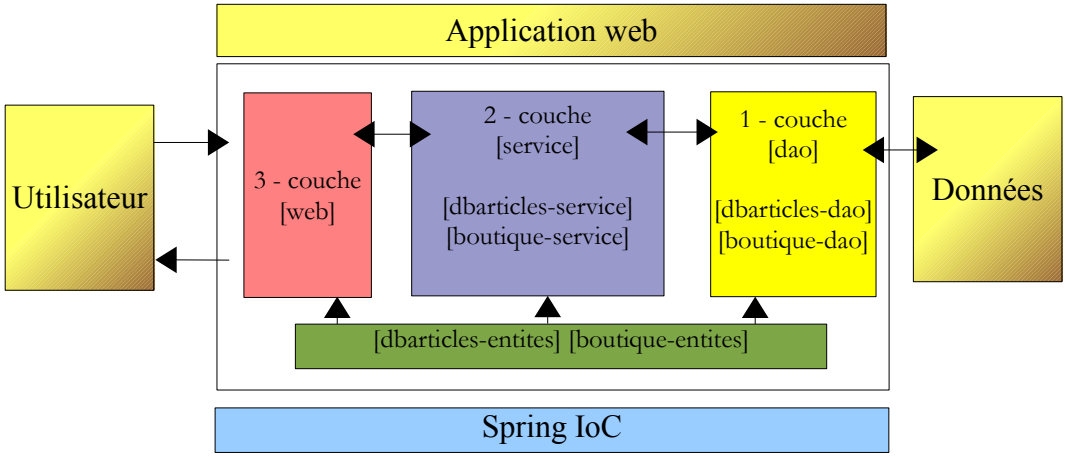
```
195.
196.}
```

Les tests sont analogues à ceux faits pour la couche [boutique-dao]. La nouveauté vient des tests sur l'achat, fait en mode tout ou rien, du panier.

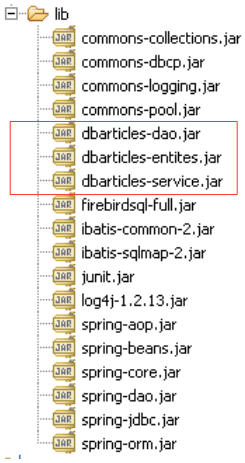
Question : écrire une classe [istia.st.boutique.service.ServiceImpl] qui satisfasse aux tests ci-dessus.

2.4.8 Archives des couches [entites, dao, service]

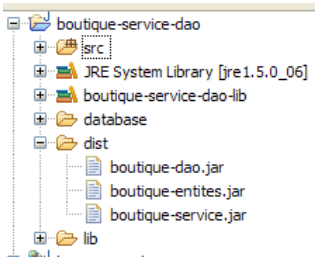
Rappelons la place des couches [entites, dao, service] au sein de l'architecture générale de l'application web que nous voulons construire :



- les éléments [dbarticles-entites, dbarticles-dao, dbarticles-service] avaient été construits précédemment et intégrés dans le projet Eclipse actuel en tant qu'archives présentes dans le Classpath de celui-ci :



- de façon analogue, nous créons les archives [boutiques-entites, boutique-dao, boutique-service] pour les paquetages [istia.st.boutique.entites, istia.st.boutique.dao, istia.st.boutique.service] du projet actuel. Celles-ci sont générées dans le dossier [dist] du projet :



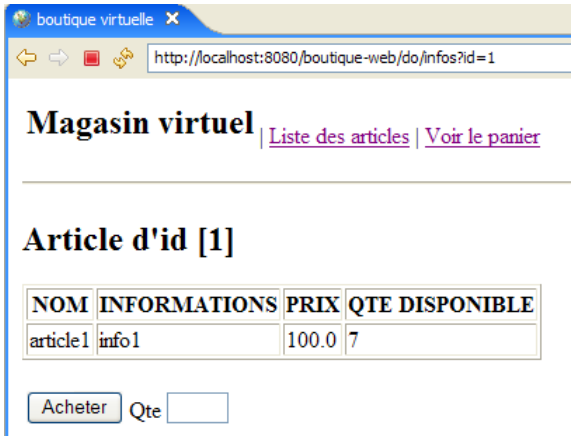
Ces archives seront utilisées dans le projet Eclipse de construction de l'interface web de l'application.

2.5 Implémentation de l'interface web

Rappelons l'interface web que nous souhaitons présenter à l'utilisateur pour lui permettre de faire des achats en ligne. Les différentes vues présentées à l'utilisateur seront les suivantes :

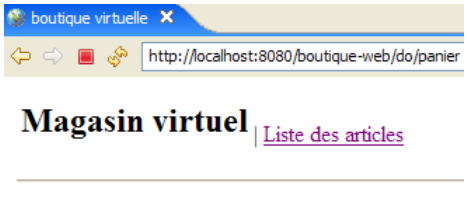
- la vue [LISTE] qui présente une liste des articles en vente

- la vue [INFOS] qui donne des informations supplémentaires sur un produit :



- la vue [PANIER] qui donne le contenu du panier du client

- la vue [PANIERVIDE] pour le cas où le panier du client est vide



- la vue [ERREURS] qui signale toute erreur de l'application

Cette interface web peut être construite en Java de plusieurs façons :

- sans framework en utilisant les servlets de Java
- avec le framework **Struts**
- avec le framework **Spring**
- ...

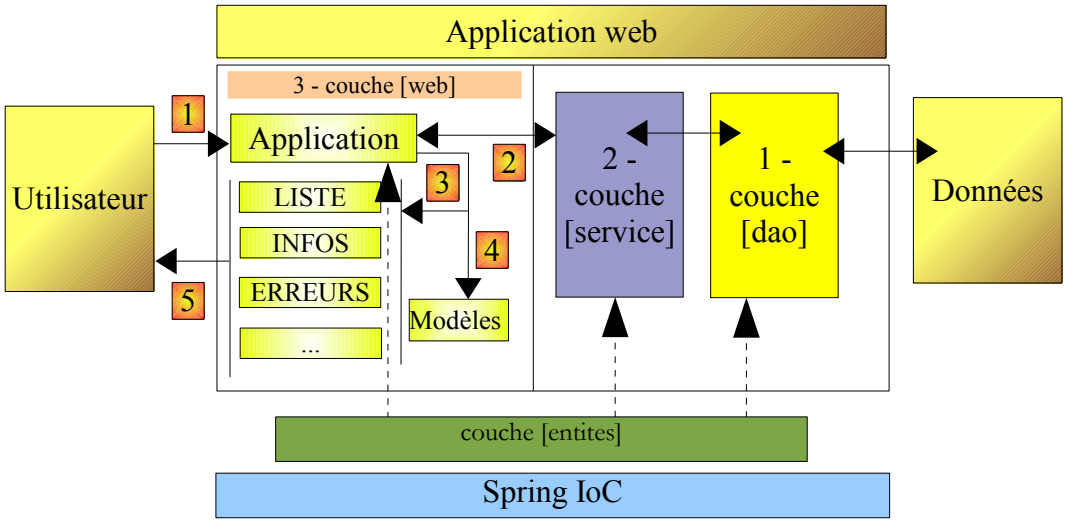
Nous construisons ici une interface web avec les outils de base fournis par Java. La maîtrise de ces outils permet de comprendre les mécanismes de base des applications web et ultérieurement d'apprécier l'aide apportée par des frameworks tels que **Struts** ou **Spring**.

Lectures conseillées :

- programmation web en Java : [http://tahe.developpez.com/java/web/]
- programmation web avec Java, Eclipse, Tomcat : [http://tahe.developpez.com/java/baseswebmvc/]

2.5.1 L'architecture de l'application web

L'application aura l'architecture à trois couches suivante où la couche [web] aura une architecture MVC :



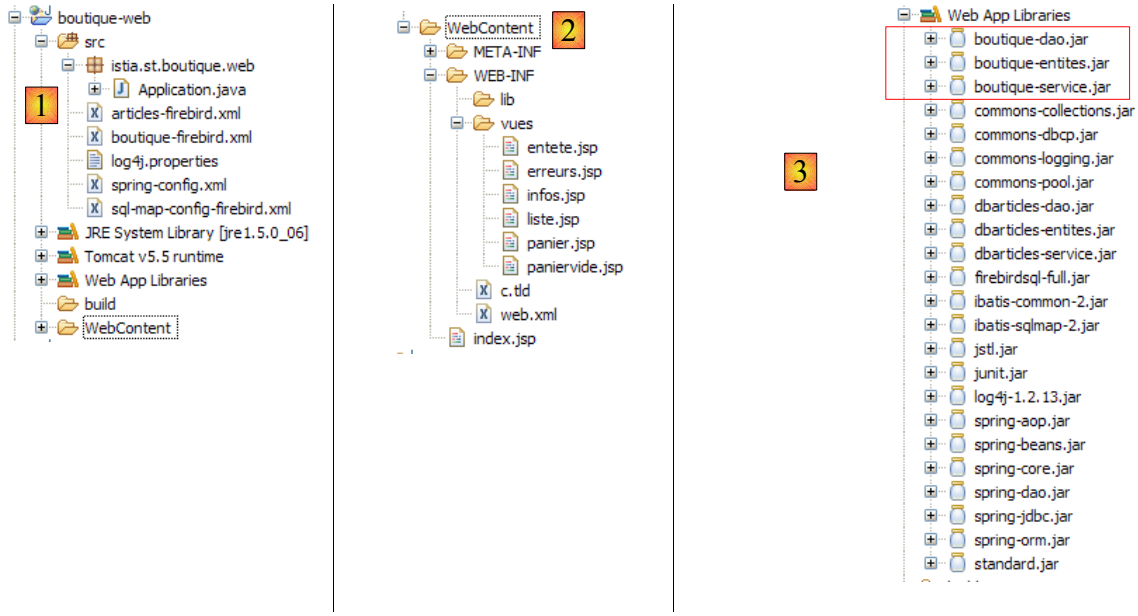
- la couche [1-dao] s'occupe de l'accès aux données. Celles-ci seront ici placées dans une base de données.
- la couche [2-service] s'occupe principalement des accès transactionnels à la base de données.
- la couche [3-web] s'occupe de la présentation des données à l'utilisateur et de l'exécution de ses requêtes.
- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces Java
- l'intégration des différentes couches est réalisée par **Spring IoC**
- la couche de présentation [3-web] implémente une architecture MVC (Modèle – Vue – Contrôleur)
  - C : c'est la servlet [Application] qui traite toutes les demandes de l'utilisateur.
  - V : c'est l'ensemble des pages JSP qui génèrent la réponse envoyée au client, ici un flux HTML.
  - M : c'est l'ensemble des informations affichées par les différentes vues.

Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

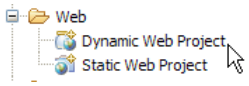
1. le client fait une demande au contrôleur C. Ce contrôleur est ici la servlet [Application] qui voit passer toutes les demandes des clients.
2. le contrôleur traite cette demande. Pour ce faire, il peut avoir besoin de l'aide de la couche [service] qui elle-même peut avoir besoin de la couche [dao] si des données doivent être échangées avec la base de données.
3. le contrôleur reçoit une réponse de la couche [service]. La demande du client a été traitée. Celle-ci peut appeler plusieurs réponses possibles. Un exemple classique est
   - une page d'erreurs si la demande n'a pu être traitée correctement
   - une page de confirmation sinon
4. le contrôleur choisit la réponse (= vue) à envoyer au client. Celle-ci est le plus souvent une page contenant des éléments dynamiques. Le contrôleur fournit ceux-ci à la vue. C'est ce que nous appelons dans ce texte, le modèle M de la vue.
5. la vue est envoyée au client. C'est le V de MVC.

2.5.2 Le projet Eclipse

Pour réaliser cette architecture, nous utiliserons le projet Eclipse [boutique-web] suivant :



Le projet est un projet Eclipse de type [Dynamic Web Project] :



Dossier [src] (1)

Ce dossier contient :

- le contrôleur de l'application web implémenté par une unique classe [Application.java]
- les fichiers de configuration des couches [dao] et [service] de l'application : [spring-config.xml, sql-map-config-firebird.xml, articles-firebird.xml, boutique-firebird.xml]. Ce sont ceux utilisés dans la construction des couches [dao] et [service] du projet Eclipse précédent.

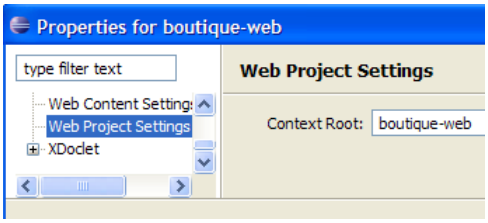
Dossier [WebContent] (2)

Il contient les éléments suivants :

- dossier [WEB-INF/vues] : les pages JSP implémentant l'interface utilisateur dont les principales vues ont été présentées au paragraphe 2.5, page 60.
- dossier [WEB-INF/lib] : contient toutes les archives nécessaires au projet. Leur liste est présentée en (3) ci-dessus. On retrouve toutes les archives utilisées dans les projets précédents ainsi que les trois archives [boutique-entites, boutique-dao, boutique-service] créées lors du dernier projet. Deux nouvelles archives apparaissent : [standard.jar] et [jstl.jar], nécessaires à l'utilisation des balises de la bibliothèque JSTL dans les pages JSP.
- [WEB-INF/web.xml] : le descripteur XML de l'application web.
- [WEB-INF/c.tld] : le fichier de définition de certaines des balises de la bibliothèque JSTL
- [index.jsp] : la page d'entrée de l'application web

2.5.3 La configuration de l'application web [boutique-web]

Le contexte de l'application [boutique-web] sera **/boutique-web**. Il est défini au sein d'Eclipse en cliquant droit sur le projet puis Properties / Web Project Settings :



Le premier fichier de configuration utilisé par une application web est le fichier [WEB-INF/web.xml]. Celui de l'application [boutique-web] sera le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app id="WebApp_ID" version="2.4"
3.    xmlns="http://java.sun.com/xml/ns/j2ee"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee          http://java.sun.com/xml/ns/j2ee/web-
    app_2_4.xsd">
6.    <display-name>boutique</display-name>
7.    <!--  ServletPersonne -->
8.    <servlet>
9.      <servlet-name>boutique</servlet-name>
10.     <servlet-class>istia.st.boutique.web.Application</servlet-class>
11.     <init-param>
12.       <param-name>springConfigFileName</param-name>
13.       <param-value>spring-config.xml</param-value>
14.     </init-param>
15.     <init-param>
16.       <param-name>urlErreurs</param-name>
17.       <param-value>/WEB-INF/vues/erreurs.jsp</param-value>
18.     </init-param>
19.     <init-param>
20.       <param-name>urlListe</param-name>
21.       <param-value>/WEB-INF/vues/liste.jsp</param-value>
22.     </init-param>
23.     <init-param>
24.       <param-name>urlInfos</param-name>
25.       <param-value>/WEB-INF/vues/infos.jsp</param-value>
26.     </init-param>
27.     <init-param>
28.       <param-name>urlPanier</param-name>
29.       <param-value>/WEB-INF/vues/panier.jsp</param-value>
30.     </init-param>
31.     <init-param>
32.       <param-name>urlPanierVide</param-name>
33.       <param-value>/WEB-INF/vues/paniervide.jsp</param-value>
34.     </init-param>
35.   </servlet>
36.   <!--  Mapping Servlet boutique-->
37.   <servlet-mapping>
38.     <servlet-name>boutique</servlet-name>
39.     <url-pattern>/do/*</url-pattern>
40.   </servlet-mapping>
41.   <!--  fichiers d'accueil -->
42.   <welcome-file-list>
43.     <welcome-file>index.jsp</welcome-file>
44.   </welcome-file-list>
45.   <!--  Page d'erreur inattendue -->
46.   <error-page>
47.     <exception-type>java.lang.Exception</exception-type>
48.     <location>/WEB-INF/vues/exception.jsp</location>
49.   </error-page>
50. </web-app>
```

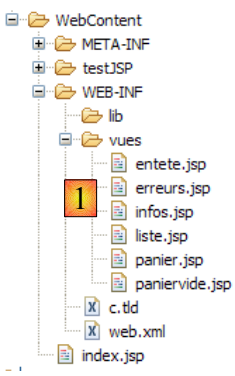
- lignes 37-40 : les URL traitées par la servlet [**boutique**] (ligne 38) seront du type **/do/*** (ligne 39). Puisque le contexte de l'application est [/boutique-web], le serveur web redirigera vers la servlet [**boutique**] les URL de la forme [http://machine:port/boutique-web/do/*].
- lignes 9-10 : la servlet [**boutique**] est implémentée par la classe [istia.st.boutique.web.Application]. Ce sera le contrôleur C de l'architecture MVC de la couce web. Il est configuré par divers paramètres :
  - lignes 12-13 : le nom du fichier Spring chargé de configurer les couches [dao] et [service] de l'application
  - lignes 11-34 : les liens nom <--> url des différentes vues de l'application
- lignes 42-44 : définissent [/vues/index.jsp] comme point d'entrée de l'application

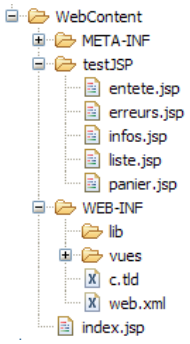
Les fichiers [spring-config.xml, sql-map-config-firebird.xml, articles-firebird.xml, boutique-firebird.xml] utilisés pour initialiser les couches [dao] et [service] de l'application sont ceux décrits au paragraphe 2.4.7, page 56, lors de l'étude de ces deux couches. On a simplement renommé le fichier [spring-config-test-service-boutique.xml] en [spring-config.xml].

2.5.4 Les vues

Nous passons maintenant en revue les différentes pages JSP de l'application. Celles-ci ne nécessitent pas la présence du contrôleur de l'application web pour être construites et testées et leur étude permet au développeur de mieux cerner les interactions attendues entre l'utilisateur et l'application web. Ce peut être ainsi une bonne chose que de commencer l'écriture d'une application web par celle de ses vues.

Les pages JSP de l'application se trouvent dans le dossier [WEB-INF/vues] de l'application :





Les pages JSP du dossier (1) [vues] seront testées à l'aide d'autres pages JSP, placées elles dans le dossier (2) [testJSP] ci-dessus.

2.5.4.1 index.jsp

C'est la page d'accueil de l'application. Si l'url initiale demandée par l'utilisateur est [http://machine:port/boutique-web/], le serveur web va examiner le fichier [web.xml] de l'application [boutique-web]. Comme l'Url pointe sur la racine de l'application [boutique-web], le serveur web va chercher si l'application définit une balise <welcome-file-list>. C'est le cas ici (lignes 42-44) du fichier [web.xml] page 64. Le serveur web va alors passer la requête du client à la page [index.jsp] située à la racine de l'application.

Son code est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.
4.  <c:redirect url="/do/liste"/>
```

Le code de la page [index.jsp] est constitué de la seule ligne 4. Celle-ci utilise la balise <c:redirect> de la bibliothèque JSTL pour demander au client de se rediriger vers l'url [http://machine:port/boutique-web/do/liste]. Nous allons voir que cette Url est celle qui affiche la liste des articles disponibles à la vente.

2.5.4.2 entete.jsp

Afin de donner une certaine homogénéité aux différentes vues, celles-ci partageront un même entête, celui qui affiche le nom de l'application avec le menu :

Le menu est dynamique et fixé par le contrôleur. Celui-ci met dans la requête [request] transmise à la page JSP, un tableau d'objets de type [HashMap] où chacun d'eux modélise l'un des liens du menu. Chaque objet [HashMap] a deux clés :

- **href** : la valeur associée est la cible du lien
- **lien** : la valeur associée est le texte du lien

La vue [entete.jsp] pourrait être testée avec la page [testJSP/entete.jsp] suivante :

```
1.  <%@ page language="java"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c"%>
3.
4.  <!-- initialisation page [entete.jsp] -->
5.  <%@ page import="java.util.Map"%>
6.  <%@ page import="java.util.HashMap"%>
7.  <%
8.    // lien 1
9.    Map<String, String> action1 = new HashMap<String, String>();
```

```
10.  action1.put("href", "href1");
11.  action1.put("lien", "lien1");
12.  // lien 2
13.  Map<String, String> action2 = new HashMap<String, String>();
14.  action2.put("href", "href2");
15.  action2.put("lien", "lien2");
16.  // tableau des liens
17.  Map[] actions = new Map[] { action1, action2 };
18.  // le tableau des liens est placé dans le requête du client
19.  request.setAttribute("actions", actions);
20. %>
21.
22. <!--  on passe le flux d'exécution à la page [entete.jsp] -->
23. <jsp:forward page="/WEB-INF/vues/entete.jsp"></jsp:forward>
```

Les lignes 4-20 définissent du code permettant de tester la vue [entete.jsp] en l'absence du contrôleur.

- lignes 9-11 : on définit un premier dictionnaire pour une première option du menu
- lignes 13-15 : on définit un second dictionnaire pour une seconde option du menu
- ligne 17 : on définit un tableau avec ces deux options
- ligne 19 : on met le tableau dans la requête courante, associé à la clé " actions "
- ligne 23 : on demande à la page [/WEB-INF/vues/entete.jsp] de continuer l'exploitation de la requête du client. [entete.jsp] doit récupérer l'attribut " actions " placé dans la requête et afficher le menu comme le montre l'exécution de la page de test [testJSP/entete.jsp] que nous venons de décrire :

Le code HTML source de la page ci-dessus est le suivant :

```
1.  <table>
2.    <tr>
3.      <td><h2>Magasin virtuel</h2></td>
4.
5.        <td>|</td>
6.        <td><a href="href1">lien1</a></td>
7.
8.        <td>|</td>
9.        <td><a href="href2">lien2</a></td>
10.
11.   </tr>
12. </table>
13. <hr>
```

Question : écrire la page JSP [/vues/entete.jsp]

Les autres vues de l'application utiliseront l'entête défini par [entete.jsp] à l'aide de la balise JSP suivante :

```
<jsp:include page="entete.jsp"/>
```

A l'exécution, cette balise aura pour effet d'inclure dans le code de la page JSP qui la contient, celui de la page [entete.jsp]. L'url de la page étant une url relative (absence de /), la page [entete.jsp] sera cherchée dans le même dossier que la page possédant la balise <jsp:include>.

2.5.4.3 liste.jsp

La page [liste.jsp] affiche la liste des articles disponibles à la vente :

La partie [1] de la vue est générée par inclusion de la page [entete.jsp], la partie [2] par [liste.jsp] elle-même.

La vue [liste.jsp] pourrait être testée avec la page [testJSP/liste.jsp] suivante :

```
1.  <%@ page language="java"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c"%>
3.
4.  <!-- initialisation page [entete.jsp] -->
5.  <%@ page import="java.util.Map"%>
6.  <%@ page import="java.util.HashMap"%>
7.  <%
8.    // lien 1
9.    Map<String, String> action1 = new HashMap<String, String>();
10.   action1.put("href", "href1");
11.   action1.put("lien", "lien1");
12.   // lien 2
13.   Map<String, String> action2 = new HashMap<String, String>();
14.   action2.put("href", "href2");
15.   action2.put("lien", "lien2");
16.   // tableau des liens
17.   Map[] actions = new Map[] { action1, action2 };
18.   // le tableau des liens est placé dans le requête du client
19.   request.setAttribute("actions", actions);
20. %>
21.
22. <!-- initialisation page [liste.jsp]-->
23. <%@ page import="istia.st.articles.entites.Article"%>
24. <%@ page import="java.util.List"%>
25. <%@ page import="java.util.ArrayList"%>
26. <%
27.   // création de deux articles
28.   Article a1 = new Article(1, 10L, "article1", "description1",
29.       "infos1", 100D, 10);
30.   Article a2 = new Article(2, 20L, "article2", "description2",
31.       "infos2", 200D, 20);
32.   // on les met dans une liste
33.   List<Article> articles = new ArrayList<Article>();
34.   articles.add(a1);
35.   articles.add(a2);
36.   // la liste des articles est placés dans le requête du client
37.   request.setAttribute("listarticles", articles);
38. %>
39.
40. <!--  on passe le flux d'exécution à la page [liste.jsp] -->
41. <jsp:forward page="/WEB-INF/vues/liste.jsp"></jsp:forward>
```

- lignes 4-20 : initialisation de l'entête
- ligne 28 : création d'un premier article
- ligne 30 : création d'un second article
- lignes 33-35 : création d'une liste de ces deux articles
- ligne 37 : la liste des articles est mise dans la requête du client, associée à la clé " listarticles "
- ligne 41 : la requête du client est passée à la page JSP [/WEB-INF/vues/liste.jsp]

L'exécution de cette page de test donne le résultat suivant :

Le code HTML source de la page ci-dessus est le suivant :

```
1.  <html>
2.  <head>
3.  <title>boutique virtuelle</title>
4.  </head>
5.  <body>
6.
7.
8.
9.  <table>
10.  <tr>
11.    <td><h2>Magasin virtuel</h2></td>
12.
13.      <td>|</td>
14.      <td><a href="href1">lien1</a></td>
15.
16.      <td>|</td>
17.      <td><a href="href2">lien2</a></td>
18.
19.  </tr>
20. </table>
21. <hr>
22.
23. <h2>Liste des articles disponibles</h2>
24. <table border="1">
25.  <tr>
26.    <th>NOM</th>
27.    <th>DESCRIPTION</th>
28.    <th>PRIX</th>
29.  </tr>
30.
31.    <tr>
32.      <td>article1</td>
33.      <td>description1</td>
34.      <td>100.0</td>
35.      <td><a href="/boutique-web/do/infos?id=1">Infos</a></td>
36.    </tr>
37.
38.    <tr>
39.      <td>article2</td>
40.      <td>description2</td>
41.      <td>200.0</td>
42.      <td><a href="/boutique-web/do/infos?id=2">Infos</a></td>
43.    </tr>
44.
45. </table>
46. <p>
47. </body>
48. </html>
```

Question : écrire la page JSP [/vues/liste.jsp]

2.5.4.4 infos.jsp

La page [infos.jsp] affiche les détails d'un article donné et permet son achat :

La partie [1] de la vue est générée par inclusion de la page [entete.jsp], la partie [2] par [infos.jsp] elle-même.

La vue [infos.jsp] pourrait être testée avec la page [testJSP/infos.jsp] suivante :

```
1.  <%@ page language="java"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c"%>
3.
4.  <!-- initialisation page [entete.jsp] -->
5.  <%@ page import="java.util.Map"%>
6.  <%@ page import="java.util.HashMap"%>
7.  <%
8.    // lien 1
9.    Map<String, String> action1 = new HashMap<String, String>();
10.   action1.put("href", "href1");
11.   action1.put("lien", "lien1");
12.   // lien 2
13.   Map<String, String> action2 = new HashMap<String, String>();
14.   action2.put("href", "href2");
15.   action2.put("lien", "lien2");
16.   // tableau des liens
17.   Map[] actions = new Map[] { action1, action2 };
18.   // le tableau des liens est placé dans le requête du client
19.   request.setAttribute("actions", actions);
20. %>
21.
22. <!-- initialisation page [infos.jsp]-->
23. <%@ page import="istia.st.articles.entites.Article"%>
24. <%
25.   // article dont on veut les détails
26.   Article a1 = new Article(1, 10L, "article1", "description1",
27.       "infos1", 100D, 10);
28.   // qté achetée
29.   String strQte = "yy";
30.   // message d'erreur
31.   String message = "Quantité erronée";
32.   // on met ces informations dans le requête du client
33.   request.setAttribute("article", a1);
34.   request.setAttribute("qte", strQte);
35.   request.setAttribute("msg", message);
36. %>
37.
38. <!--  on passe le flux d'exécution à la page [infos.jsp] -->
39. <jsp:forward page="/WEB-INF/vues/infos.jsp"></jsp:forward>
```

- lignes 4-20 : initialisation de l'entête
- lignes 26-27 : création d'un article
- ligne 29 : la quantité achetée sous forme de chaîne de caractères
- ligne 31 : un message destiné à l'utilisateur
- lignes 33-35 : ces trois informations sont mises dans la requête du client
- ligne 39 : la requête du client est passée à la page JSP [/WEB-INF/vues/infos.jsp]

L'exécution de cette page de test donne le résultat suivant :

Le code HTML source de la page ci-dessus est le suivant :

```
1.  <html>
2.    <head>
3.      <title>boutique virtuelle</title>
4.    </head>
5.    <body>
6.
7.
8.
9.  <table>
10.   <tr>
11.     <td><h2>Magasin virtuel</h2></td>
12.
13.       <td>|</td>
14.       <td><a href="href1">lien1</a></td>
15.
16.       <td>|</td>
17.       <td><a href="href2">lien2</a></td>
18.
19.   </tr>
20. </table>
21. <hr>
22.
23.     <h2>Article d'id [1]</h2>
24.     <table border="1">
25.       <tr>
26.         <th>NOM</th><th>INFORMATIONS</th><th>PRIX</th><th>QTE DISPONIBLE</th>
27.       </tr>
28.         <tr>
29.           <td>article1</td>
30.           <td>infos1</td>
31.           <td>100.0</td>
32.           <td>10</td>
33.         </tr>
34.     </table>
35.     <p>
36.     <form method="post" action="/boutique-web/do/achat?id=1"/>
37.       <table>
38.         <tr>
39.           <td><input type="submit" value="Acheter"></td>
40.           <td>Qte <input type="text" name="qte" size="3" value="yy"></td>
41.           <td>Quantité erronée</td>
42.         </tr>
43.       </table>
44.     </form>
45.   </body>
46. </html>
```

Question : écrire la page JSP [/vues/infos.jsp]

2.5.4.5 erreurs.jsp

La page [erreurs.jsp] affiche une liste de messages d'erreur :

La partie [1] de la vue est générée par inclusion de la page [entete.jsp], la partie [2] par [erreurs.jsp] elle-même.

La vue [erreurs.jsp] pourrait être testée avec la page [testJSP/erreurs.jsp] suivante :

```
1.  <%@ page language="java"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c"%>
3.
4.  <!-- initialisation page [entete.jsp] -->
5.  <%@ page import="java.util.Map"%>
6.  <%@ page import="java.util.HashMap"%>
7.  <%
8.    // lien 1
9.    Map<String, String> action1 = new HashMap<String, String>();
10.   action1.put("href", "href1");
11.   action1.put("lien", "lien1");
12.   // lien 2
13.   Map<String, String> action2 = new HashMap<String, String>();
14.   action2.put("href", "href2");
15.   action2.put("lien", "lien2");
16.   // tableau des liens
17.   Map[] actions = new Map[] { action1, action2 };
18.   // le tableau des liens est placé dans le requête du client
19.   request.setAttribute("actions", actions);
20. %>
21.
22. <!-- initialisation page [erreurs.jsp]-->
23. <%@ page import="java.util.List"%>
24. <%@ page import="java.util.ArrayList"%>
25. <%
26.   // liste d'erreurs
27.   List<String> erreurs = new ArrayList<String>();
28.   erreurs.add("erreur1");
29.   erreurs.add("erreur2");
30.   // on met la liste des erreurs dans le requête du client
31.   request.setAttribute("erreurs", erreurs);
32. %>
33.
34. <!--  on passe le flux d'exécution à la page [erreurs.jsp] -->
35. <jsp:forward page="/WEB-INF/vues/erreurs.jsp"></jsp:forward>
```

- lignes 4-20 : initialisation de l'entête
- lignes 27-29 : création d'une liste d'erreurs
- ligne 31 : la liste d'erreurs est mise dans la requête du client associée à la clé " erreurs "
- ligne 39 : la requête du client est passée à la page JSP [/WEB-INF/vues/erreurs.jsp]

L'exécution de cette page de test donne le résultat suivant :

Le code HTML source de la page ci-dessus est le suivant :

```
1.  <html>
2.    <head>
3.      <title>boutique virtuelle</title>
4.    </head>
5.    <body>
6.  <table>
7.    <tr>
8.      <td><h2>Magasin virtuel</h2></td>
9.        <td>|</td>
10.       <td><a href="href1">lien1</a></td>
11.       <td>|</td>
12.       <td><a href="href2">lien2</a></td>
13.   </tr>
14. </table>
15. <hr>
16.     <h2>Les erreurs suivantes se sont produites</h2>
17.     <ul>
18.         <li>erreur1</li>
19.         <li>erreur2</li>
20.     </ul>
21.   </body>
22. </html>
```

Question : écrire la page JSP [/vues/erreurs.jsp]

2.5.4.6 panier.jsp

La page [panier.jsp] affiche le contenu du panier du client :

La partie [1] de la vue est générée par inclusion de la page [entete.jsp], la partie [2] par la page [panier.jsp] elle-même.

La vue [panier.jsp] pourrait être testée avec la page [testJSP/panier.jsp] suivante :

```
1.  <%@ page language="java"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c"%>
3.
4.  <!-- initialisation page [entete.jsp] -->
5.  <%@ page import="java.util.Map"%>
6.  <%@ page import="java.util.HashMap"%>
```

```
7.  <%
8.    // lien 1
9.    Map<String, String> action1 = new HashMap<String, String>();
10.   action1.put("href", "href1");
11.   action1.put("lien", "lien1");
12.   // lien 2
13.   Map<String, String> action2 = new HashMap<String, String>();
14.   action2.put("href", "href2");
15.   action2.put("lien", "lien2");
16.   // tableau des liens
17.   Map[] actions = new Map[] { action1, action2 };
18.   // le tableau des liens est placé dans le requête du client
19.   request.setAttribute("actions", actions);
20. %>
21.
22. <!-- initialisation page [panier.jsp]-->
23. <%@ page import="istia.st.articles.entites.Article"%>
24. <%@ page import="istia.st.boutique.entites.Panier"%>
25. <%@ page import="istia.st.boutique.entites.Achat"%>
26. <%
27.   // création du panier d'achats
28.   Panier panier = new Panier();
29.   // premier achat
30.   panier.ajouter(new Achat(new Article(1, 10L, "article1",
31.       "description1", "infos1", 100D, 10), 2));
32.   // second achat
33.   panier.ajouter(new Achat(new Article(2, 20L, "article2",
34.       "description2", "infos2", 200D, 20), 5));
35.   // le panier est placé dans le requête du client
36.   request.setAttribute("panier", panier);
37. %>
38.
39. <!--  on passe le flux d'exécution à la page [panier.jsp] -->
40. <jsp:forward page="/WEB-INF/vues/panier.jsp"></jsp:forward>
```

- lignes 4-20 : initialisation de l'entête
- ligne 28 : création d'un panier vide
- lignes 30-31 : un premier achat est ajouté au panier d'achats
- lignes 33-34 : un second achat est ajouté au panier d'achats
- ligne 36 : le panier est mis dans la requête du client, associé à la clé " panier "
- ligne 39 : la requête du client est passée à la page JSP [/WEB-INF/vues/panier.jsp]

L'exécution de cette page de test donne le résultat suivant :

Le code HTML source de la page ci-dessus est le suivant :

```
1.  <html>
2.    <head>
3.      <title>boutique virtuelle</title>
4.    </head>
5.    <body>
6.
7.
8.
9.  <table>
10.   <tr>
11.     <td><h2>Magasin virtuel</h2></td>
12.       <td>|</td>
13.       <td><a href="href1">lien1</a></td>
14.       <td>|</td>
```

```
15.       <td><a href="href2">lien2</a></td>
16.   </tr>
17. </table>
18. <hr>
19.     <h2>Contenu de votre panier</h2>
20.     <table border="1">
21.       <tr>
22.         <th>Article</th><th>Qt&eacute; achet&eacute;e</th><th>Qt&eacute;
    disponible</th><th>Pu</th><th>Total</th>
23.       </tr>
24.         <tr>
25.           <td>article1</td>
26.           <td>2</td>
27.           <td>10</td>
28.           <td>100.0</td>
29.           <td>200.0</td>
30.           <td><a href="/boutique-web/do/retirerachat?id=1">Retirer</a></td>
31.           <td><a href="/boutique-web/do/infos?id=1">Modifier</a></td>
32.         </tr>
33.         <tr>
34.           <td>article2</td>
35.           <td>5</td>
36.           <td>20</td>
37.           <td>200.0</td>
38.           <td>1000.0</td>
39.           <td><a href="/boutique-web/do/retirerachat?id=2">Retirer</a></td>
40.           <td><a href="/boutique-web/do/infos?id=2">Modifier</a></td>
41.         </tr>
42.     </table>
43.     <p>
44.     Total de la commande : 1200.0 euros
45.   </body>
46. </html>
```

Question : écrire la page JSP [/vues/panier.jsp]

2.5.4.7 paniervide.jsp

La page [paniervide.jsp] est utilisée lorsque l'utilisateur demande l'affichage de son panier et que celui-ci est vide :

La partie [1] de la vue est générée par inclusion de la page [entete.jsp], la partie [2] par la page [panier.jsp] elle-même.

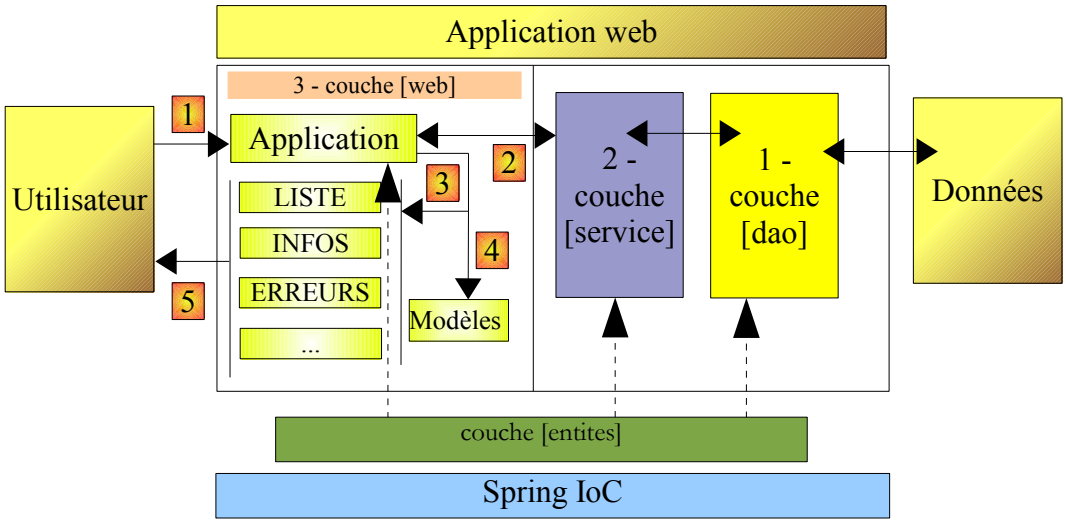
Le code de la page JSP [/vues/paniervide.jsp] est le suivant :

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>

<html>
  <head>
    <title>boutique virtuelle</title>
  </head>
  <body>
    <jsp:include page="entete.jsp" />
    <h2>Contenu de votre panier</h2>
    <p>
    Votre panier est vide.
  </body>
</html>
```

2.5.5 Le contrôleur

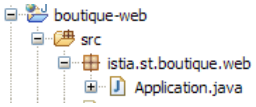
Rappelons l'architecture de l'application [boutique-web] :



Il reste à écrire le coeur de notre application web, le contrôleur [Application]. Son rôle consiste à :

- récupérer la requête du client,
- traiter l'action demandée par celui-ci, éventuellement à l'aide de la couche [service],
- envoyer en réponse la vue appropriée.

La classe [Application.java] du contrôleur est dans le dossier [src] du projet Eclipse :



Son squelette pourrait être le suivant :

```
1.  package istia.st.boutique.web;
2.  ...
3.  public class Application extends HttpServlet {
4.
5.    // constantes de l'application
6.    // les URL
7.    private final String URL_JSP_ERREURS = "urlErreurs";
8.    private final String URL_JSP_LISTE = "urlListe";
9.    private final String URL_JSP_INFOS = "urlInfos";
10.   private final String URL_JSP_PANIER = "urlPanier";
11.   private final String URL_JSP_PANIER_VIDE = "urlPanierVide";
12.   // le fichier de configuratio Spring
13.   private final String SPRING_CONFIG_FILENAME = "springConfigFileName";
14.   // les paramètres de configuration de l'application trouvés dans [web.xml]
15.   private final String[] paramètres = { URL_JSP_ERREURS, URL_JSP_LISTE,
16.       URL_JSP_INFOS, URL_JSP_PANIER, URL_JSP_PANIER_VIDE,
17.       SPRING_CONFIG_FILENAME };
18.   // les actions
19.   private final String ACTION_LISTE = "liste";
20.   private final String ACTION_PANIER = "panier";
21.   private final String ACTION_ACHAT = "achat";
22.   private final String ACTION_INFOS = "infos";
23.   private final String ACTION_RETIRER_ACHAT = "retirerachat";
24.   private final String ACTION_MODIFIER_ACHAT = "modifierachat";
25.   private final String ACTION_VALIDATION_PANIER = "validationpanier";
26.   private final String ACTION_RAFRAICHIR_PANIER = "rafraichirpanier";
27.   // les textes des options de menu
28.   private final String lienActionListe = "Liste des articles";
29.   private final String lienActionPanier = "Voir le panier";
30.   private final String lienActionValidationPanier = "Valider le panier";
31.   // les dictionnaires des options de menu
32.   private HashMap<String,String> hActionListe = new HashMap<String,String>(2);
33.   private HashMap<String,String> hActionPanier = new HashMap<String,String>(2);
34.   private HashMap<String,String> hActionValidationPanier = new HashMap<String,String>(2);
35.   private HashMap<String,String> hActionRafraichirPanier = new HashMap<String,String>(2);
36.
37.   // paramètres d'instance
38. ...
39.   // services
40.   private istia.st.articles.service.IService serviceArticles = null;
41.   private istia.st.boutique.service.IService serviceBoutique = null;
```

```
42.
43.  // init application
44.  @SuppressWarnings("unchecked")
45.  public void init() throws ServletException {
46. ...
47.  }
48.
49.  // requêtes GET / POST
50.  public void doGet(HttpServletRequest request, HttpServletResponse response)
51.      throws IOException, ServletException {
52. ....
53.  }
54.
55.  public void doPost(HttpServletRequest request, HttpServletResponse response)
56.      throws IOException, ServletException {
57. ...
58.  }
59. }
```

2.5.5.1 Initialisation du contrôleur

Lorsque la classe [Application] du contrôleur est chargée par le conteneur de servlets, sa méthode [init] est exécutée. Ce sera la seule fois. Une fois chargée en mémoire, le contrôleur y restera et traitera les requêtes des différents clients. Chaque client fait l'objet d'un thread d'exécution et les méthodes du contrôleur sont ainsi exécutées simultanément par différents threads. On rappelle que, pour cette raison, le contrôleur ne doit pas avoir de champs que ses méthodes pourraient modifier. Ses champs doivent être en lecture seule. Ils sont initialisés par la méthode [init] dont c'est le rôle principal. Cette méthode a en effet la particularité d'être exécutée une unique fois par un seul thread. Il n'y a donc pas de problèmes d'accès concurrents aux champs du contrôleur dans cette méthode. La méthode [init] a pour but d'initialiser les objets nécessaires à l'application web et qui seront partagés en lecture seule par tous les threads clients. Ces objets partagés peuvent être placés en deux endroits :

- les champs privés du contrôleur
- le contexte d'exécution de l'application (ServletContext)

La méthode [init] du contrôleur de l'application [boutique-web] fera les actions suivantes :

- vérifiera la présence dans le fichier [web.xml], des paramètres nécessaires au bon fonctionnement de l'application. Ceux-ci ont été décrits au paragraphe 2.5.3, page 63. Chaque paramètre absent sera signalé par un message d'erreur dans un champ privé [ArrayList erreursInitialisations] du contrôleur.
- si le paramètre [urlErreurs] est absent, une exception de type [ServletException] sera lancée.
- si le paramètre [springConfigFileName] est présent, on l'utilisera pour instancier les couches [dao] et [service] de l'application. Si cette instanciation échoue, on signalera l'erreur en plaçant un message dans le champ [erreursInitialisations].

Question : écrire la méthode [init] du contrôleur.

2.5.5.2 Méthodes doGet, doPost

Ces deux méthodes traitent les requêtes les requêtes HTTP GET et POST des clients. La requête client sera traitée de la façon suivante :

- le champ [erreursInitialisation] sera vérifié. S'il référence une liste non vide, cela signifie qu'il y a eu des erreurs lors de l'initialisation de l'application et que celle-ci ne peut fonctionner. On enverra alors en réponse, la vue [erreurs.jsp] avec [erreursInitialisation] comme liste d'erreurs et un menu vide d'options.
- l'application est construite pour que toutes les requêtes d'un utilisateur aient un paramètre nommé [action] dont la valeur est le nom de l'action à exécuter. Si la valeur de ce paramètre ne correspond pas à une action connue, la vue [erreurs.jsp] est envoyée avec le message d'erreur adéquat. Si le paramètre [action] est valide, la requête du client est passée à une procédure spécifique à l'action, pour traitement.

Avec ces spécifications, les méthodes [doGet, doPost] pourraient être les suivantes :

```
1.// requêtes GET / POST
2.  public void doGet(HttpServletRequest request, HttpServletResponse response)
3.      throws IOException, ServletException {
4.    execute(request, response);
5.  }
6.
7.  public void doPost(HttpServletRequest request, HttpServletResponse response)
8.      throws IOException, ServletException {
9.    execute(request, response);
10. }
11.
12. // requêtes GET / POST
```

```
13. protected void execute(HttpServletRequest request,
14.     HttpServletResponse response) throws IOException, ServletException {
15.   // on vérifie comment s'est passée l'initialisation de la servlet
16.   if (erreursInitialisation.size() != 0) {
17.     // on affiche la page des erreurs
18.     afficheErreurs(request, response, erreursInitialisation,
19.         new HashMap[] {});
20.     // fin
21.     return;
22.   }
23.   // on récupère la méthode d'envoi de la requête
24.   String méthode = request.getMethod().toLowerCase();
25.   // on récupère l'action à exécuter
26.   String action = request.getPathInfo();
27.   if (action != null) {
28.     action = action.substring(1);
29.   }
30.   // on traite l'action
31.   if (action == null) {
32.     // liste des articles
33.     doListe(request, response);
34.     return;
35.   }
36.   if (méthode.equals("get") && action.equals(ACTION_LISTE)) {
37.     // liste des articles
38.     doListe(request, response);
39.     return;
40.   }
41.   if (action.equals(ACTION_INFOS)) {
42.     // infos sur un article
43.     doInfos(request, response);
44.     return;
45.   }
46.   if (méthode.equals("post") && action.equals(ACTION_ACHAT)) {
47.     // achat d'un article
48.     doAchat(request, response);
49.     return;
50.   }
51.   if (méthode.equals("get") && action.equals(ACTION_PANIER)) {
52.     // affichage du panier
53.     doPanier(request, response);
54.     return;
55.   }
56.   if (méthode.equals("get") && action.equals(ACTION_RETIRER_ACHAT)) {
57.     // suppression d'un article du panier
58.     doRetirerAchat(request, response);
59.     return;
60.   }
61.   if (méthode.equals("get") && action.equals(ACTION_MODIFIER_ACHAT)) {
62.     // modification qté achetée d'un article du panier
63.     doInfos(request, response);
64.     return;
65.   }
66.   if (méthode.equals("get") && action.equals(ACTION_VALIDATION_PANIER)) {
67.     // validation du panier
68.     doValidationPanier(request, response);
69.     return;
70.   }
71.   if (méthode.equals("get") && action.equals(ACTION_RAFRAICHIR_PANIER)) {
72.     // validation du panier
73.     doRafraichirPanier(request, response);
74.     return;
75.   }
76.   // action inconnue
77.   ArrayList<String> erreurs = new ArrayList<String>();
78.   erreurs.add("action [" + action + "] inconnue");
79.   // on affiche la page des erreurs
80.   afficheErreurs(request, response, erreurs,
81.       new HashMap[] { hActionListe });
82.   // fin
83.   return;
84. }
```

2.5.5.3 Traitement des différentes actions demandées par l'utilisateur

2.5.5.3.1 /do/liste

Cette action est disponible dans de nombreuses vues, par exemple dans la suivante :

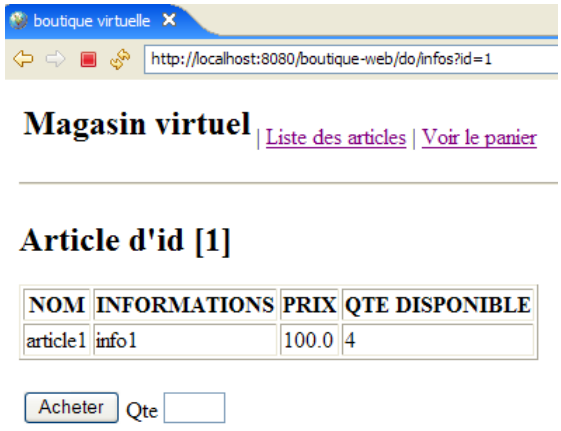
- en [1], on demande la liste des articles
- en [2], on l'obtient avec la possibilité, pour chaque article, d'avoir des informations complémentaires

méthode du contrôleur	demande du client	traitement
- doListe	GET /do/liste	– envoyer la vue [liste.jsp] vide avec dedans la liste des articles disponibles à la vente.

Question : écrire la méthode **doListe** du contrôleur.

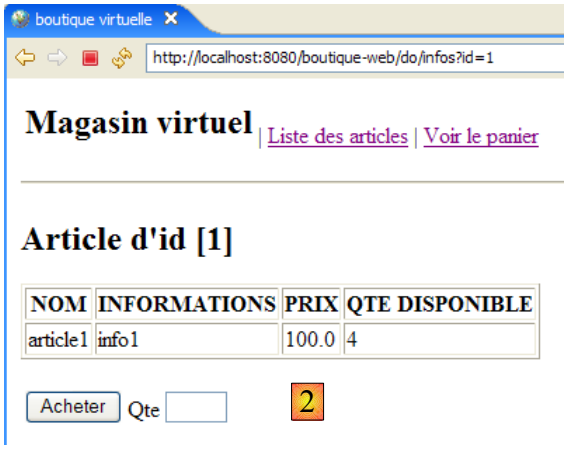
2.5.5.3.2 /do/infos

Cette action est disponible dans la vue [Liste] :



- en [1], l'utilisateur demande des détails sur l'article " article1 "
- en [2], on lui donne ces détails et en même temps on lui propose un formulaire d'achat

ainsi que dans la vue [PANIER] :



- en [1], l'utilisateur demande à modifier la quantité achetée de " article1 "
- en [2], on lui présente le formulaire d'achat de l'article " article1 "

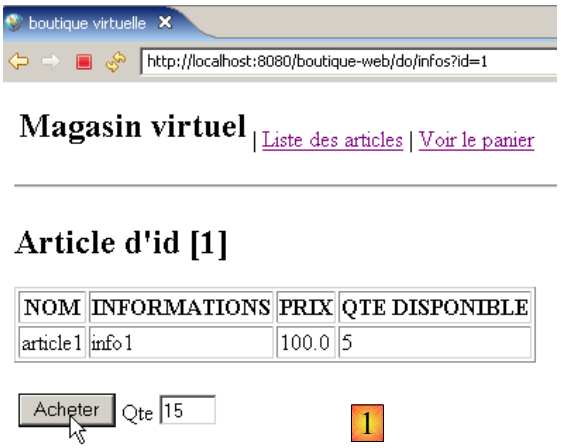
méthode du contrôleur	demande du client	traitement
- doInfos	GET /do/infos?id=XX	– vérifier la validité du paramètre id. S'il est incorrect, envoyer la vue [erreurs.jsp]. – s'il est correct, demander à la BD les détails de l'article identifié par [id] – si l'article a été obtenu, afficher la vue [infos.jsp] et mettre l'article dans la session. – si l'article n'a pas été obtenu, afficher la vue [erreurs.jsp]

Question : écrire la méthode **doInfos** du contrôleur.

2.5.5.3.3 /do/achat

Cette action est disponible dans la vue [Infos] :

- en [1], on achète deux articles " article1 "
- en [2], l'article acheté est placé dans le panier. S'il était déjà présent dans le panier, sa quantité est remplacée par la nouvelle.



- en [1], on demande une quantité plus grande que la quantité disponible
- en [2], on indique l'erreur

- en [1], on fait une erreur de saisie
- en [2], on indique l'erreur

méthode du contrôleur	demande du client	traitement
- doAchat	POST /do/achat?id=XX le champ de saisie HTML **qte** est posté	– récupérer l'article dans la session – vérifier la validité de la quantité – afficher la vue [panier] si l'achat est valide – afficher la vue [infos] avec un message d'erreur sinon

Question : écrire la méthode **doAchat** du contrôleur.

2.5.5.3.4 /do/panier

Cette action est disponible par exemple dans la vue [Liste] :

- en [1], on demande à voir le panier
- en [2], le contenu du panier est présenté

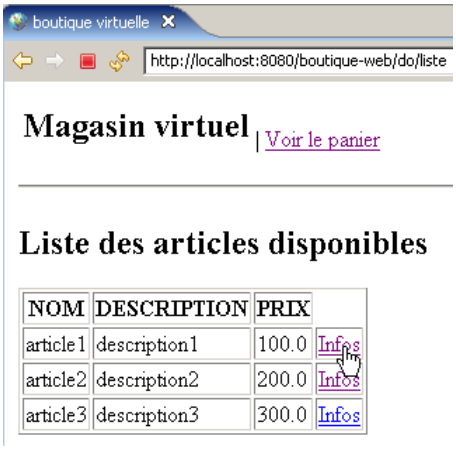
méthode du contrôleur	demande du client	traitement
- doPanier	GET /do/panier	– afficher le contenu du panier

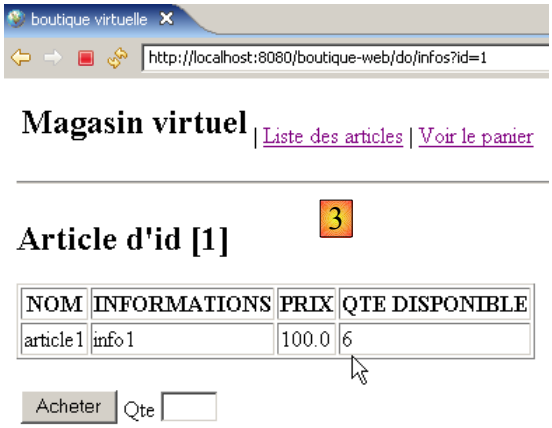
Question : écrire la méthode **doPanier** du contrôleur.

2.5.5.3.5 /do/validationpanier

Cette action est disponible dans la vue [Panier] :

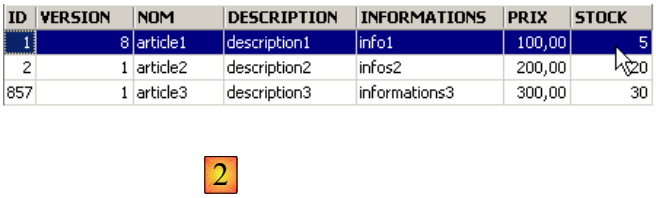

boutique virtuelle
http://localhost:8080/boutique-web/do/panier
Magasin virtuel | Liste des articles | Valider le panier
Contenu de votre panier
Article | Qté achetée | Qté disponible | Pu | Total
article1 | 2 | 8 | 100.0 | 200.0 | Retirer | Modifier
Total de la commande : 200.0 euros
1


boutique virtuelle
http://localhost:8080/boutique-web/do/liste
Magasin virtuel | Voir le panier
Liste des articles disponibles
NOM | DESCRIPTION | PRIX
article1 | description1 | 100.0 | Infos
article2 | description2 | 200.0 | Infos
article3 | description3 | 300.0 | Infos
2



- en [1], on achète 2 articles " article1 " dont le stock actuel est 8
- en [2], l'achat a réussi et on présente de nouveau la liste des articles
- en [3], on vérifie le nouveau stock de " article1 ". Il est bien passé à 6.

Ci-dessus, l'achat du panier a réussi. Il peut échouer si l'un des articles a un stock insuffisant comme montré ci-dessous :



- en [1], on s'apprête à acheter 6 articles " article1 "
- en [2], on change " à la main " le stock de " article1 ". On le passe à 5 pour que l'achat du panier échoue.
- en [3], on valide le panier
- en [4], l'achat échoue et on obtient la vue [erreurs]

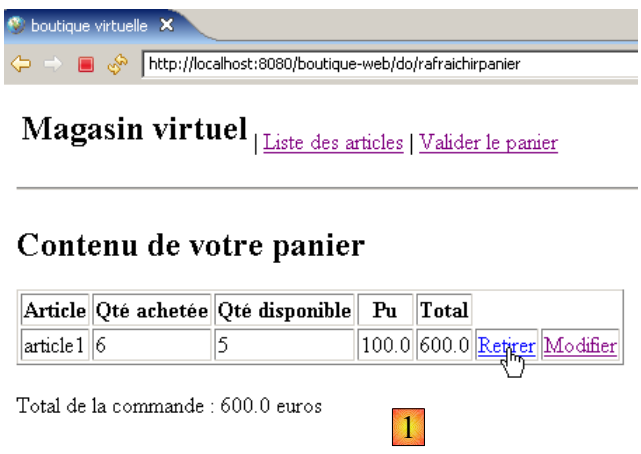
- en [5], on demande à voir le panier
- en [6], on affiche le panier avec dans " Qté disponible ", les stocks réels en base de données. Ainsi l'utilisateur voit que le stock de " article1 " est de 5 alors qu'il en demandait 6.

méthode du contrôleur	demande du client	traitement
- doValidationPanier	GET /do/validationpanier	– valide l'achat du panier en mode " tout ou rien ". Après un achat réussi, le panier est vide et les stocks des articles achetés décrémentés en BD. Après un achat raté, le panier n'a pas changé et aucun stock n'a été décrémenté.

Question : écrire la méthode **doValidationPanier** du contrôleur.

2.5.5.3.6 /do/retirerachat

Cette action est disponible dans la vue [Panier] :



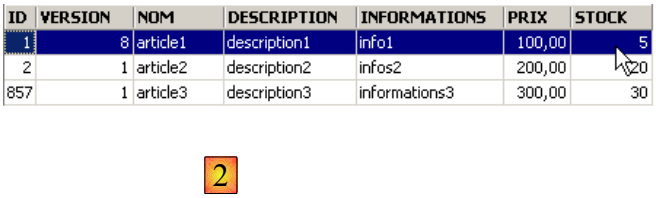
- en [1], on retire du panier l'achat de l'article " article1 "
- en [2], on représente le panier, une fois l'achat retiré

méthode du contrôleur	demande du client	traitement
- doRetirerAchat	GET /do/retirerachat?id=XX	– retire l'achat de l'article d'ID=XX du panier. Si le paramètre id est invalide, on ne fait rien et on laisse le panier en l'état.

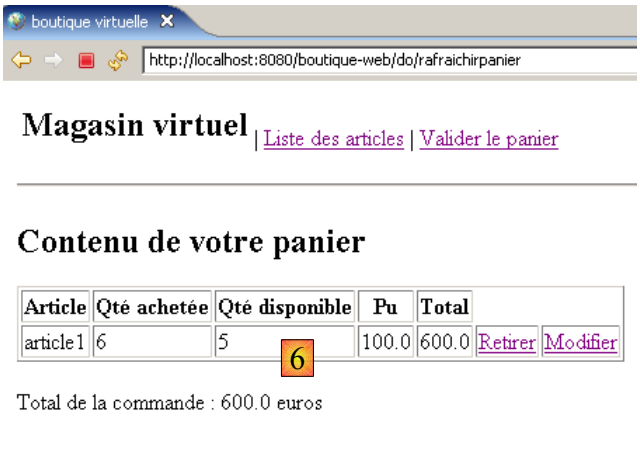
Question : écrire la méthode **doRetirerAchat** du contrôleur.

2.5.5.3.7 /do/rafraichirpanier

Cette action est disponible dans la vue [Erreurs] après l'échec de l'achat d'un panier :



- en [1], on s'apprête à acheter 6 articles " article1 "
- en [2], on change " à la main " le stock de " article1 ". On le passe à 5 pour que l'achat du panier échoue.
- en [3], on valide le panier
- en [4], l'achat échoue et on obtient la vue [erreurs]



- en [5], on demande à voir le panier
- en [6], on affiche le panier avec dans " Qté disponible ", les stocks réels en base de données. Ainsi l'utilisateur voit que le stock de " article1 " est de 5 alors qu'il en demandait 6.

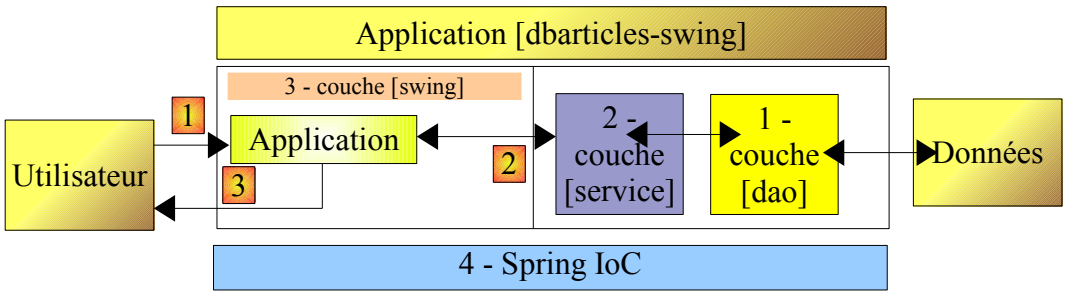
méthode du contrôleur	demande du client	traitement
- doRafraichirPanier	GET /do/rafraichirpanier	– affiche le panier courant après avoir récupéré en BD les stocks de chacun des articles qu'il contient.

Question : écrire la méthode **doRafraichirPanier** du contrôleur.

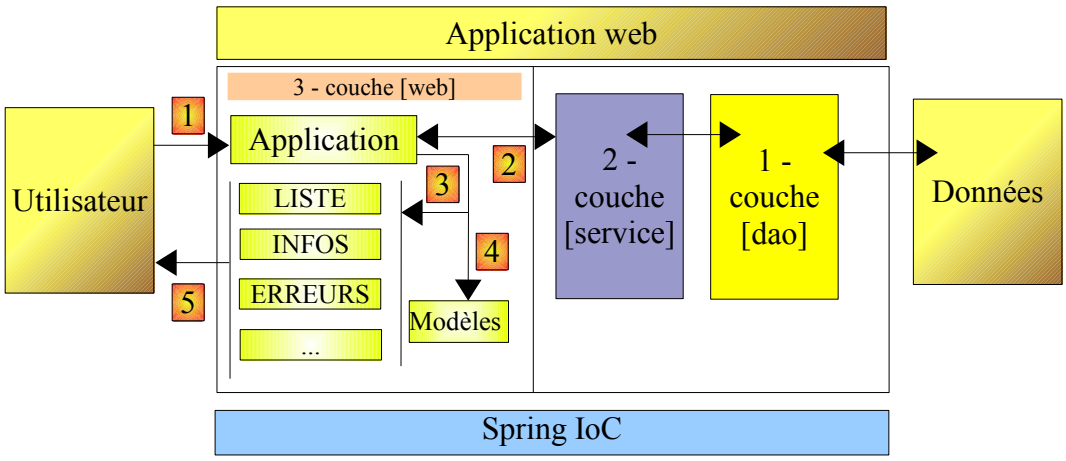
3 Conclusion

Nous avons construit deux applications avec une architecture à trois couches :

L'application [dbarticles-swing] qui présentait à l'utilisateur une interface swing :



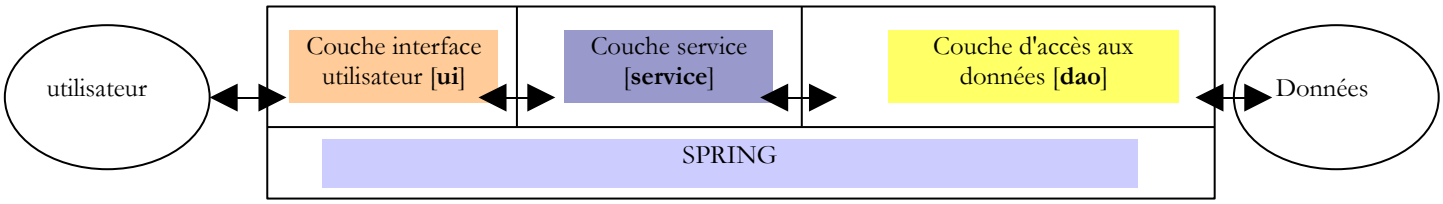
L'application [boutique-web] qui présentait à l'utilisateur une interface web :



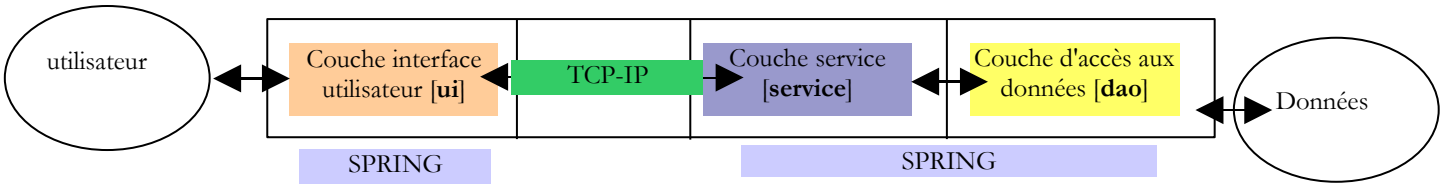
Nous pourrions maintenant construire une synthèse des deux applications précédentes en implémentant l'application [boutique-web] en client / serveur :

- côté client, on présenterait à l'utilisateur une interface swing
- côté serveur, on retrouverait les couches [service] et [dao] de l'application [boutique-web]. Elles seraient reprises à l'identique.
- le dialogue client /serveur serait un dialogue HTTP assuré par des classes spécialisées d'un module de Spring, appelé **Spring Remoting**.

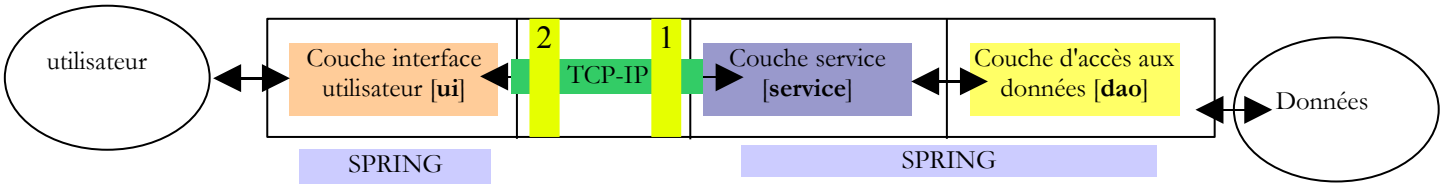
Considérons une application à trois couches où les trois couches sont sur une même machine :



Spring Remoting permet de transformer cette architecture en l'architecture client-serveur suivante :



Si les trois couches ont été encapsulées dans des archives, Spring Remoting permet de construire l'architecture client / serveur précédente par simple adjonction de couches d'adaptation entre le client et le serveur. L'acquis (les archives) de l'application précédente est préservé :



Deux couches notées ci-dessus 1 et 2 ont été ajoutées :

- la couche 1 va exposer un service distant d'accès à la couche [service]. Cette couche a pour rôle de cacher le réseau à la couche [service] qui ne saura pas qu'elle est utilisée par des clients distants. Nous l'appellerons couche [proxyServeur].
- la couche 2 va implémenter les proxy d'accès aux services web de la couche 1, de façon transparente pour la couche [ui] qui ne saura pas que les données qu'elle reçoit et envoie vont sur le réseau. Nous l'appellerons couche [proxyClient].

Spring Remoting permet d'implémenter les couches d'adaptation ci-dessus de différentes façons :

1. avec la technologie Spring RMI
2. avec la technologie Spring RMI-IIOP
3. avec la technologie Spring HttpInvoker
4. avec la technologie Spring Caucho Hessian
5. avec la technologie Spring Caucho Burlap
6. avec la technologie JaxRpc

On pourra lire à ce sujet l'article "Construction d'une application web MVC distribuée avec Spring Remoting" à l'adresse [http://tahe.developpez.com/java/remote3tier-part1].