TD JAVA

Mots clés : Bases Java, Tableaux, Listes, Classes et Interfaces, Exceptions, Fichiers texte, Bases de données, Architectures trois couches, Interface graphique swing

1 Partie 1

On désire écrire un programme qui, au soir d'élections, puisse calculer le nombre de sièges obtenus pas les différentes listes en présence. On trouvera sur les feuilles suivantes (journal Ouest-France du 15 mars 1986) le mode de calcul des sièges, pour une élection proportionnelle à la plus forte moyenne.

On écrira une application Java "console", c.a.d. sans interface graphique. Elle demandera les renseignements suivants à l'utilisateur (tapés au clavier) :

- nombre de sièges à pourvoir
- nombre de listes en compétition
- pour chaque liste : son nom, son nombre de voix

Avec ces renseignements, l'application calcule les sièges obtenus par chacune des listes et les affiche à l'écran sous la forme suivante :

- La liste [X1] a obtenu [N1] sièges
- La liste [X2] a obtenu [N2] sièges
- ...

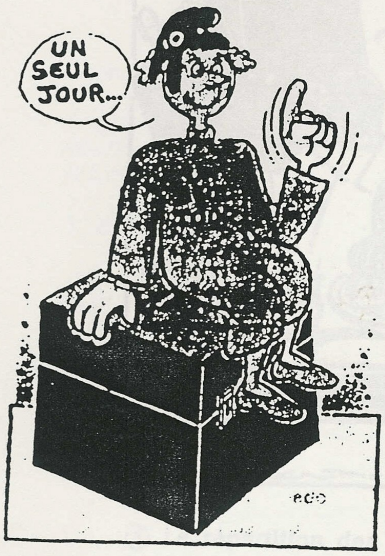
où [Xi] est le nom de la liste n° i et [Ni] le nombre de sièges qu'elle a obtenus.

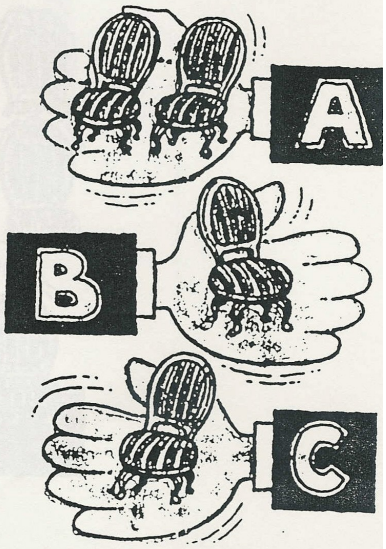
Q1 : écrire l'algorithme de cette application. Les erreurs de saisie seront signalées à l'utilisateur qui devra alors resaisir la donnée erronée. L'élection est invalidée si toutes les listes ont un pourcentage de voix inférieur à un certain seuil (ici 5%). L'algorithme signalera également ce cas.

Q2 : traduire l'algorithme en Java. On utilisera la méthode statique [main] d'une classe [MainElections1] pour faire le traitement demandé.

La proportionnelle, mode d'emploi

2





PRENONS UN DÉPARTEMENT où six sièges sont à pourvoir, sept listes en présence et où 100 000 suffrages ont été exprimés. Voici la règle du jeu, valable aussi bien pour les législatives que pour les régionales.

OF 15/3/84

① La barre des 5 %

Ont obtenu, liste A, 32 000 voix (32 %) ; liste B, 25 000 (25 %) ; liste C, 16 000 (16 %) ; liste D, 12 000 (12 %) ; liste E, 8 000 (8 %) ; liste F, 4 500 (4,5 %) ; liste G, 2 500 (2,5 %).

Les deux dernières, n'ayant pas atteint la barre des 5 %, sont éliminées.

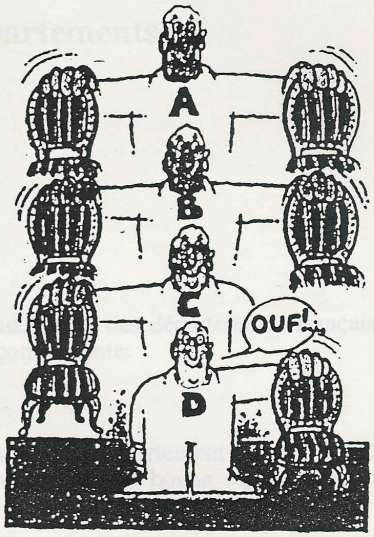
Première opération : calculer les **suffrages exprimés utiles**, en additionnant les voix des listes ayant dépassé les 5 %, soit : A + B + C + D + E = 93 000

② Le calcul du quotient électoral

On calcule le quotient électoral en divisant le nombre de suffrages exprimés utiles par le nombre de sièges à pourvoir : 93 000 : 6 = 15 500. On attribue ensuite à chaque liste autant de sièges que son nombre de voix contient de fois le quotient. Ainsi, la liste A, ayant obtenu plus de deux fois le quotient, aura deux sièges.

Liste A : 2 sièges.
Liste B : 1 siège.
Liste C : 1 siège.

Au total, quatre sièges sont attribués « au quotient ».



③ La répartition des restes

Restent, dans notre exemple, à répartir deux sièges à la plus forte moyenne. On attribue d'abord fictivement un siège supplémentaire à chacune des listes. On divise ensuite le nombre de suffrages recueillis par chaque liste par le nombre de sièges déjà attribués plus un. Celle qui a le plus fort résultat obtient un siège.

Liste A, 32 000 : 3 (2 + 1) = 10 666.
Liste B, 25 000 : 2 (1 + 1) = 12 500.
Liste C, 16 000 : 2 (1 + 1) = 8 000.
Liste D, 12 000 : 1 (0 + 1) = 12 000.
Liste E, 8 000 : 1 (0 + 1) = 8 000.

Le premier siège restant va à la liste B, qui a la plus forte moyenne. On recommence l'opération pour l'attribution du dernier siège, qui ira à la liste D.

④ Le classement final

Les deux sièges restants sont donc attribués aux listes B et D. A l'arrivée, les six sièges à pourvoir sont ainsi répartis :

Liste A : 2 sièges.
Liste B : 2 sièges.
Liste C : 1 siège.
Liste D : 1 siège.

OF 15/3/96

c.o.d liste A, 32000 : 3 (2+1) = 10666
" B, 25000 : 3 (2+1) = 8333 ← liste B a maintenant un siège de plus
" C, 16000 : 2 (1+1) = 8000
" D, 12000 : 1 (0+1) = 12000
" E, 8000 : 1 (0+1) = 8000

le siège va à la liste D

2 Partie 2

Mots clés : classe, interface, héritage, exception, polymorphisme

Dans la partie 1 de l'exercice ELECTIONS aucune classe n'a été utilisée. On a construit une solution comme on l'aurait construite en langage C. Nous allons maintenant introduire la notion de classe et d'objets.

2.1 La classe [ListeElectorale]

En langage C, nous aurions probablement utilisé une structure pour représenter une liste participant à l'élection. Elle aurait pu être de la forme suivante :

```
struct t_liste
   {
     char nom[15];
     long voix;
     int  elimine;
     int sieges;
   };
```

La notion de structure n'existe pas dans le langage Java. Il faut la remplacer par celle de classe. On décide donc de créer une classe pour mémoriser les informations sur une liste. Celle-ci aurait le squelette suivant :

```
1. package istia.st.elections;
2.
3. public class ListeElectorale {
4.
5.     /**
6.      * identité de la liste
7.      */
8.     private int id;
9.
10.    /**
11.     * nom de la liste
12.     */
13.    private String nom;
14.    /**
15.     * nombre de voix de la liste
16.     */
17.    private int voix;
18.    /**
19.     * nombre de sièges de la liste
20.     */
21.    private int sieges;
22.    /**
23.     * indique si la liste est éliminée ou non
24.     */
25.    private boolean elimine;
26.
27.    /**
28.     * constructeur par défaut
29.     */
30.    public ListeElectorale() {
31.    }
32.
33.    /**
34.     *
35.     * @param nom String : le nom de la liste
36.     * @param voix int : son nombre de voix
37.     * @param sieges int : son nombre de sieges
38.     * @param elimine boolean : son état éliminé ou non
39.     */
40.    public ListeElectorale(int id, String nom, int voix, int sieges, boolean elimine) {
41.....
42.    }
43.
44.    /**
45.     *
46.     * @return int : l'identifiant de la liste
47.     */
48.    public int getId() {
49....
50.    }
51.
52.    /**
53.     * initialise l'identifiant de liste
54.     * @param id int : identifiant de la liste
```

```
55.     * @throws ElectionsException si id<1
56.     */
57.    public void setId(int id) {
58....
59.    }
60.
61.    /**
62.     *
63.     * @return String : le nom de la liste
64.     */
65.    public String getNom() {
66....
67.    }
68.
69.    /**
70.     * initialise le nom de la liste
71.     * @param nom String : nom de la liste
72.     *  @throws ElectionsException si le nom est vide ou blanc
73.     */
74.    public void setNom(String nom) {
75....
76.    }
77.
78.    /**
79.     *
80.     * @return int : le nombre de voix de la liste
81.     */
82.    public int getVoix() {
83....
84.    }
85.
86.    /**
87.     * initialise le nombre de voix de la liste
88.     * @param voix int : le nombre de voix de la liste
89.     * @throws ElectionsException si voix<0
90.     */
91.    public void setVoix(int voix) {
92....
93.    }
94.
95.    /**
96.     *
97.     * @return int : le nombre de sièges de la liste
98.     */
99.    public int getSieges() {
100....
101.    }
102.
103.    /**
104.     * fixe le nombre de sièges de la liste
105.     * @param sieges int : le nombre de sièges de la liste
106.     *  @throws ElectionsException si sieges<0
107.     */
108.    public void setSieges(int sieges) {
109....
110.    }
111.
112.    /**
113.     *
114.     * @return boolean : valeur du champ elimine
115.     */
116.    public boolean isElimine() {
117....
118.    }
119.
120.    /**
121.     *
122.     * @param sieges int
123.     */
124.    public void setElimine(boolean elimine) {
125....
126.    }
127.
128.    /**
129.     *
130.     * @return String : identité de la liste électorale
131.     */
132.    public String toString() {
133....
134.    }
135.}
```

- ligne 8 : n° identifiant une liste de façon unique. N'est pas indispensable ici mais est prévu pour une utilisation future.
- ligne 13 : le nom de la liste.

- ligne 17 : le nombre de voix de la liste
- ligne 21 : le nombre de sièges de la liste
- ligne 25 : booléen indiquant si la liste est éliminée (au-dessous du seuil électoral) ou non.

Chaque champ privé nommé [xyz] peut être initialisé par une méthode nommée [setXyz]. La méthode [getXyz] permet elle d'obtenir la valeur du champ privé [xyz]. Dans le cas particulier où [xyz] est un champ de type booléen, la méthode [getXyz] peut être remplacée par la méthode [isXyz]. Le nommage particulier de ces méthodes obéit à une norme de codage appelée norme **JavaBean**. Ainsi nous définissons les méthodes publiques suivantes :

- getId (ligne 48), setId (ligne 57)
- getNom (ligne 65), setNom (ligne 74)
- getVoix (ligne 82), setVoix (ligne 91)
- getSieges (ligne 99), setSieges (ligne 108)
- isElimine (ligne 116), setElimine (ligne 124)

- lignes 30-31 : définissent un constructeur sans paramètres. Celui-ci permet de créer un objet [ListeElectorale] sans l'initialiser. Celui-ci peut ensuite être initialisé grâce aux méthodes **set**.
- lignes 40-42 : définissent un constructeur permettant de créer un objet [ListeElectorale] tout en initialisant ses cinq champs privés.
- ligne 132-134 : définissent la méthode [toString] qui rend une chaîne de caractères donnant les valeurs des cinq champs de l'objet.

Un programme de test de la classe **ListeElectorale** pourrait être le suivant :

```
1. package istia.st.elections.tests;
2.
3. import istia.st.elections.ListeElectorale;
4. import istia.st.elections.ElectionsException;
5.
6. public class MainTest1ListeElectorale {
7.     public static void main(String[] args) {
8.       // création d'une liste électorale
9.         ListeElectorale listeElectorale1 = new ListeElectorale(1, "A", 32000, 0, false);
10.        // affichage identité liste
11.        System.out.println("listeElectorale1=" + listeElectorale1);
12.        // modification du nombre de sièges
13.        listeElectorale1.setSieges(2);
14.        // affichage identité liste 1
15.        System.out.println("listeElectorale1=" + listeElectorale1);
16.        // une nouvelle liste électorale
17.        ListeElectorale listeElectorale2 = listeElectorale1;
18.        // affichage identité liste 2
19.        System.out.println("listeElectorale2=" + listeElectorale2);
20.        // modification du nombre de sièges
21.        listeElectorale2.setSieges(3);
22.        // affichage identité des 2 listes
23.        System.out.println("listeElectorale2=" + listeElectorale2);
24.        System.out.println("listeElectorale1=" + listeElectorale1);
25.    }
26.}
```

L'affichage écran obtenu après exécution du programme ci-dessus est le suivant :

```
1. listeElectorale1=[A,32000,0,false]
2. listeElectorale1=[A,32000,2,false]
3. listeElectorale2=[A,32000,2,false]
4. listeElectorale2=[A,32000,3,false]
5. listeElectorale1=[A,32000,3,false]
```

Question 1 : En vous aidant de tout ce qui précède, compléter le code de la classe **ListeElectorale**.

2.2 Création d'une classe d'exception

Parmi les différentes classes d'exception du langage Java, il en est une appelée [RuntimeException]. Cette classe dérive de la classe [Exception], racine de toutes les classes d'exception. La particularité des instances de [RuntimeException] ou instances dérivées est que l'on n'est pas obligé de les déclarer ou de les gérer. On les appelle des **exceptions non contrôlées**.

Prenons un premier exemple. La classe [BufferedReader] est une classe dont les instances permettent de lire des lignes de texte dans un flux de données. Elle possède une méthode [readLine] dont la signature est la suivante :

```
public String readLine()throws IOException
```

On voit que la méthode peut lancer une exception de type [IOException]. L'arborescence de cette classe est la suivante :

```
1. java.lang.Object
2.   java.lang.Throwable
3.       java.lang.Exception
4.           java.io.IOException
```

La classe [IOException] dérive de la classe [Exception] (ligne 3). Le compilateur nous force à gérer et à déclarer les exceptions de type [java.lang.Exception] ou dérivé (sauf pour la branche RuntimeException). Ainsi, pour lire une ligne de texte tapée au clavier, on sera obligé d'écrire quelque chose comme :

```
1. BufferedReader clavier=....;
2. String ligne=null;
3. try{
4.     ligne=clavier.readLine();
5. }catch (IOException ex){
6.     // gérer l'exception
7.     ....
8. }
```

Prenons un autre exemple. Pour transformer une chaîne en entier on peut utiliser la méthode statique [Integer.parseInt] dont la signature est la suivante :

```
public static int parseInt(String s) throws NumberFormatException
```

L'argument [s] est la chaîne de caractères à transformer en entier. On voit que la méthode peut lancer une exception de type [NumberFormatException]. L'arborescence de cette classe est la suivante :

```
1. java.lang.Object
2.   java.lang.Throwable
3.       java.lang.Exception
4.           java.lang.RuntimeException
5.               java.lang.IllegalArgumentException
6.                   java.lang.NumberFormatException
```

La classe [NumberFormatException] dérive de la classe [RuntimeException] (ligne 4). Le compilateur ne nous force pas à gérer et à déclarer les exceptions de type [java.lang.RuntimeException] ou dérivé. Ainsi, on pourra écrire quelque chose comme :

```
1. BufferedReader clavier=....;
2. String ligne=null;
3. try{
4.     ligne=clavier.readLine();
5. }catch (IOException ex){
6.     // gérer l'exception
7.     ....
8. }
9. int age=Integer.parseInt(ligne);
```

Nous ne sommes pas obligés de mettre une clause [try - catch] pour gérer l'éventuelle exception générée par [Integer.parseInt] (ligne 9).

Il y a des avantages et inconvénients à créer et utiliser des classes d'exception dérivées de [RuntimeException] :

- au chapitre des avantages : le code est plus léger
- au chapitre des inconvénients : on peut être ramené aux méthodes du C où chaque fonction rend un code d'erreur que peu de gens utilisent, justement pour avoir un code plus léger. Lorsqu'une telle erreur non gérée se produit, le programme plante, généralement de façon fort inélégante.

Nous décidons de créer une classe spéciale regroupant toutes les exceptions pouvant survenir dans notre application ELECTIONS. Elle s'appellera [ElectionsException] et dérivera de la classe [RuntimeException]. Son code est le suivant :

```
1. package istia.st.elections;
2.
3. // classe d'exception pour l'application Elections
4. // l'exception est non contrôlée
5.
6. public class ElectionsException extends RuntimeException {
7.  // constructeur sans paramètre
8.     public ElectionsException() {
9.         super();
10.    }
11.    /**
12.     *
13.     * @param message String : le message d'erreur à associer à l'exception
14.     */
```

```
15.    public ElectionsException(String message) {
16.        super(message);
17.    }
18.
19.    /**
20.     *
21.     * @param cause Throwable : une exception à encapsuler dans ElectionsException
22.     */
23.    public ElectionsException(Throwable cause) {
24.        super(cause);
25.    }
26.
27.    /**
28.     *
29.     * @param message String : le message d'erreur à associer à l'exception
30.     * @param cause Throwable : une exception à encapsuler dans ElectionsException
31.     */
32.    public ElectionsException(String message, Throwable cause) {
33.        super(message, cause);
34.    }
35.}
```

- ligne 1 : nous plaçons la classe dans le paquetage [istia.st.elections]
- ligne 6 : la classe dérive de [RuntimeException]. Elle est donc non contrôlée.
- nous utiliserons dans notre application deux sortes de constructeur :
  - celui classique des lignes 15-17 comme ci-dessous :

```
throw new ElectionsException("Le nombre de sièges doit être >0")
```

  - ou celui des lignes 32-34 destiné à faire remonter une exception déjà survenue, en l'encapsulant dans une exception de type [ElectionsException] :

```
try{
....
}catch(IOException ex){
  // on encapsule l'exception
  throw new ElectionsException("Problème d'accès aux données",ex);
}
```

Cette seconde méthode a l'avantage de conserver l'information que peut contenir la première exception.

Question 2 : Reprendre le code de la classe **ListeElectorale** de façon à ce que les méthodes **set** lancent une exception de type [ElectionsException] si l'initialisation demandée est incorrecte, tel que initialiser le nom avec une chaîne vide.

2.3 Une classe de test unitaire

Nous avons déjà présenté une classe de test qui était la suivante :

```
1. package istia.st.elections.tests;
2.
3. import istia.st.elections.ListeElectorale;
4. import istia.st.elections.ElectionsException;
5.
6. public class MainTest1ListeElectorale {
7.     public static void main(String[] args) {
8.       // création d'une liste électorale
9.         ListeElectorale listeElectorale1 = new ListeElectorale(1, "A", 32000, 0, false);
10.        // affichage identité liste
11.        System.out.println("listeElectorale1=" + listeElectorale1);
12.        // modification du nombre de sièges
13.        listeElectorale1.setSieges(2);
14.        // affichage identité liste 1
15.        System.out.println("listeElectorale1=" + listeElectorale1);
16.        // une nouvelle liste électorale
17.        ListeElectorale listeElectorale2 = listeElectorale1;
18.        // affichage identité liste 2
19.        System.out.println("listeElectorale2=" + listeElectorale2);
20.        // modification du nombre de sièges
21.        listeElectorale2.setSieges(3);
22.        // affichage identité des 2 listes
23.        System.out.println("listeElectorale2=" + listeElectorale2);
24.        System.out.println("listeElectorale1=" + listeElectorale1);
25.        // test d'une exception
26.        try {
27.            listeElectorale2.setSieges( -3);
28.        } catch (ElectionsException ex) {
29.            System.err.println("L'exception suivante s'est produite : [" +
30.                               ex.toString() + "]");
```

```
31.         }
32.     }
33.}
```

Nous avons rajouté lignes 26-31, du code testant le comportement de la classe face à une initialisation incorrecte.

Ce type de test repose sur une vérification visuelle. On vérifie qu'on obtient à l'écran ce qui est attendu. C'est une méthode à déconseiller en milieu professionnel. Les tests doivent toujours être automatisés au maximum et viser à ne nécessiter aucune intervention humaine. L'être humain est en effet sujet à la fatigue et sa capacité à vérifier des tests s'émousse au fil de la journée.

Une application évolue au fil du temps. A chaque évolution, on doit vérifier que l'application ne "régresse" pas, c.a.d. qu'elle continue à passer les tests de bon fonctionnement qui avaient été faits lors de son écriture initiale. On appelle ces tests, tests de "non régression". Une grosse aplication peut nécessiter des centaines de tests. On teste en effet chaque méthode de chaque classe de l'application. On appelle cela des **tests unitaires**. Ceux-ci peuvent mobiliser beaucoup de développeurs s'ils n'ont pas été automatisés.

Des outils ont été développés pour automatiser les tests. L'un d'eux s'appelle [JUnit]. C'est une bibliothèque de classes destinées à gérer les tests. Nous allons utiliser cet outil pour tester la classe [ListeElectorale].

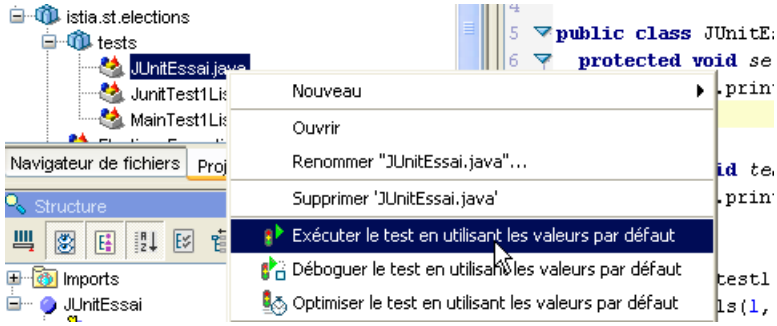
Un programme de test JUnit a la forme suivante :

```
1. package istia.st.elections.tests;
2.
3. import junit.framework.TestCase;
4.
5. public class JUnitEssai extends TestCase {
6.   protected void setUp() {
7.     System.out.println("setUp");
8.   }
9.
10.  protected void tearDown() {
11.    System.out.println("tearDown");
12.  }
13.
14.  public void test1() {
15.    System.out.println("test1");
16.    assertEquals(1, 1);
17.  }
18.
19.  public void test2() {
20.    System.out.println("test2");
21.    assertEquals(2, 3);
22.  }
23.}
```

- ligne 1 : la classe a été placée dans le paquetage [istia.st.elections.tests].
- ligne 3 : la classe [junit.framework.TestCase] est importée. C'est elle qui contient les méthodes permettant d'automatiser les tests.
- ligne 5 : la classe de test dérive de la classe [TestCase]. Cette classe possède les méthodes [assert] qui vont nous permettre de vérifier qu'un résultat attendu a bien été obtenu. Il n'y aura plus de vérification visuelle. La classe [junit.framework.TestCase] a été importée ligne 3.
- lignes 6-8 : définissent la méthode [**setUp**]. Cette méthode est exécutée **avant** chaque test unitaire.
- lignes 10-12 : définissent la méthode [**tearDown**]. Cette méthode est exécutée **après** chaque test unitaire.
- lignes 14-17 : définissent la méthode [test1]. Dans une classe dérivée de [TestCase], toutes les méthodes dont le nom commence par [test] sont considérées comme des méthodes à tester. Elles seront exécutées les unes après les autres sauf indication contraire du testeur qui peut sélectionner lui-même les méthodes [test*] à tester. Avant chaque exécution d'une méthode [test*], la méthode [setUp] est exécutée. Après chaque exécution d'une méthode [test*], la méthode [tearDown] est exécutée.
- ligne 16 : une des méthodes [assert*] définies par la classe [TestCase]. On trouve les méthodes [assert] suivantes :
    - **assertEquals(expression1, expression2)** : vérifie que les valeurs des deux expressions sont égales. De nombreux types d'expression sont acceptés (int, String, float, double, boolean, char, short). Si les deux expressions ne sont pas égales, alors une exception de type [AssertionFailedError] est lancée.
    - **assertEquals(réel1, réel2, delta)** : vérifie que deux réels sont égaux à **delta** près, c.a.d abs(réel1-réel2)<=delta. On pourra écrire par exemple **assertEquals(réel1, réel2, 1E-6)** pour vérifier que deux valeurs sont égales à 10^{-6} près.
    - **assertEquals(message, expression1, expression2)** et **assertEquals(message, réel1, réel2, delta)** sont des variantes permettant de préciser le message d'erreur à associer à l'exception de type [AssertionFailedError] lancée lorsque la méthode [assertEquals] échoue.
    - **assertNotNull(Object)** et **assertNotNull(message, Object)** : vérifie que **Object** n'est pas égal à **null**.
    - **assertNull(Object)** et **assertNull(message, Object)** : vérifie que **Object** est égal à **null**.
    - **assertSame(Object1, Object2)** et **assertSame(message, Object1, Object2)** : vérifie que les références **Object1** et **Object2** pointent sur le même objet.
    - **assertNotSame(Object1, Object2)** et **assertNotSame(message, Object1, Object2)** : vérifie que les références **Object1** et **Object2** ne pointent pas sur le même objet.
- ligne 16 : le test doit réussir

- ligne 21 : le test doit échouer

L'exécution d'une classe [TestCase] dépend de l'environnement de développement utilisé. Sous Jbuilder, un clic droit sur la classe de test permet de l'exécuter :



Les résultats obtenus sont les suivants :

On voit que la méthode [test2] a échoué. A chaque fois qu'un test échoue, un message d'erreur lui est associé. Pour [test2] c'est le suivant :

```
junit.framework.AssertionFailedError: expected:<2> but was:<3>
...(Cliquez pour un suivi de pile complet)...
  at istia.st.elections.tests.JUnitEssai.test2(JUnitEssai.java:21)
...
```

L'appel qui a échoué était

```
    assertEquals(2, 3);
```

Le premier paramètre est appelé la valeur attendue, le second la valeur réelle. Le message ci-dessus indique que la valeur attendue était 2 mais que la valeur réelle était 3. On a par ailleurs, le n° de la ligne où l'erreur s'est produite (ligne 21).

Enfin, les messages écrits sur la console ont été les suivants :

```
setUp
test1
tearDown
setUp
test2
tearDown
```

Ces messages montrent que les méthodes [setUp] et [tearDown] ont bien été appelées respectivement avant et après chaque méthode de test.

Dans une application professionnelle en cours de développement, les classes de tests sont exécutées chaque nuit de façon automatisée. A chaque début de journée, les développeurs savent alors ce qui marche et ce qui ne marche pas. Cela assure que l'application ne régresse pas. Une application a régressé si des tests qui passaient il y a une semaine ne passent plus à cause d'une modification apportée par un des développeurs.

Les classes de test ne sont pas nécessairement écrites par les développeurs eux-mêmes. Elles peuvent l'être par les personnes qui ont écrit les spécifications de l'application. Certaines méthodes de développement (TDD : Test Driven Development) préconisent l'écriture des classes de tests avant même l'écriture des classes à tester. Cela permet parfois de clarifier des spécifications qui pourraient être interprétées de plusieurs façons.

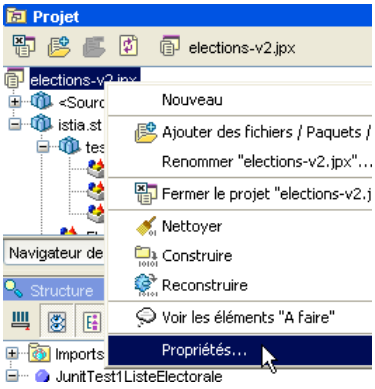
Un programme de test de la classe [ListeElectorale] pourrait être le suivant :

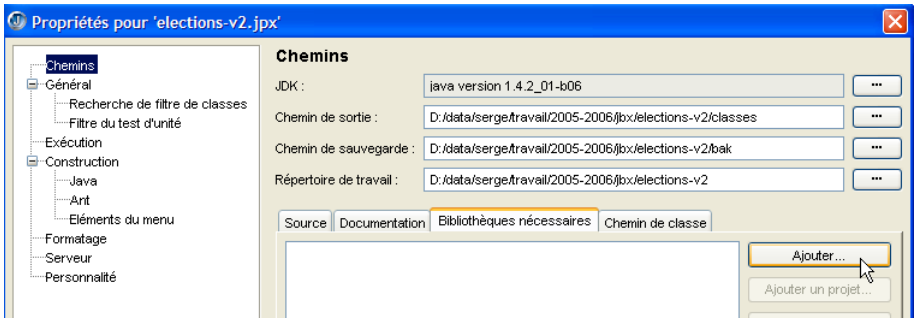
```
1. package istia.st.elections.tests;
2.
3. import junit.framework.TestCase;
4. import istia.st.elections.ListeElectorale;
5. import istia.st.elections.ElectionsException;
6.
```

```
7. public class JunitTest1ListeElectorale extends TestCase {
8.     // méthode de test
9.     public void test1()  {
10.        // création liste électorale
11.      ListeElectorale liste = new ListeElectorale(1,"a", 32000, 0,false);
12.      // vérifications
13.      assertEquals("a", liste.getNom());
14.      assertEquals(32000, liste.getVoix());
15.      assertEquals(false, liste.isElimine());
16.      assertEquals(0, liste.getSieges());
17.      // vérification validité id
18.      boolean erreur = false;
19.      try {
20.        liste.setId( -4);
21.      }
22.      catch (ElectionsException e) {
23.        erreur = true;
24.      }
25.      assertEquals(true, erreur);
26.      // vérification validité nom
27.      erreur = false;
28.      try {
29.        liste.setNom("");
30.      }
31.      catch (ElectionsException e) {
32.        erreur = true;
33.      }
34.      assertEquals(true, erreur);
35.      // vérification validité voix
36.      erreur = false;
37.      try {
38.        liste.setVoix( -4);
39.      }
40.      catch (ElectionsException e) {
41.        erreur = true;
42.      }
43.      assertEquals(true, erreur);
44.      // vérification validité sièges
45.      erreur = false;
46.      try {
47.        liste.setSieges( -4);
48.      }
49.      catch (ElectionsException e) {
50.        erreur = true;
51.      }
52.      assertEquals(true, erreur);
53.    }
54.}
```

La classe [TestCase] est importée ligne 3. Cette classe se trouve dans une archive appelée **junit.jar**. Quelque soit l'outil utilisé pour développer l'application Java, il faut que [junit.jar] soit dans le CLASSPATH de l'application. Le CLASSPATH d'une l'application liste les archives .jar et les répertoires contenant des fichiers .class qui doivent être explorés, soit par le compilateur au moment de la compilation, soit par la machine virtuelle Java au moment de l'exécution, pour trouver les classes externes référencées par l'application Java compilée ou exécutée.

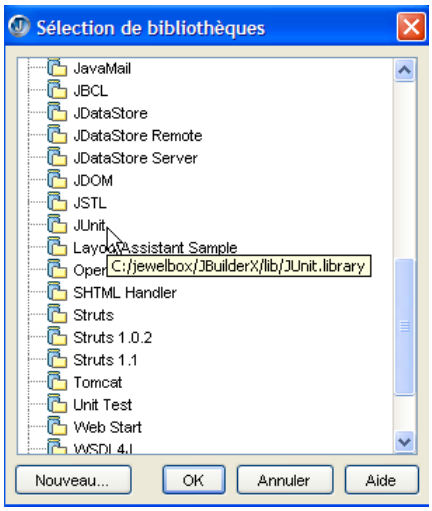
Sous Jbuilder, on procèdera ainsi :

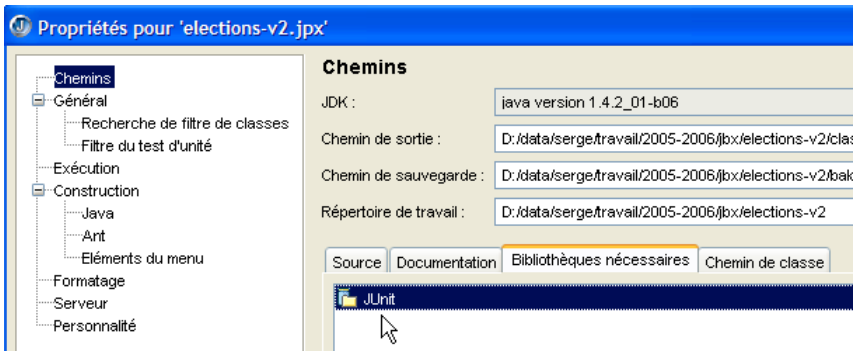




- onglet [Bibliothèques nécessaires]
- [Ajouter]

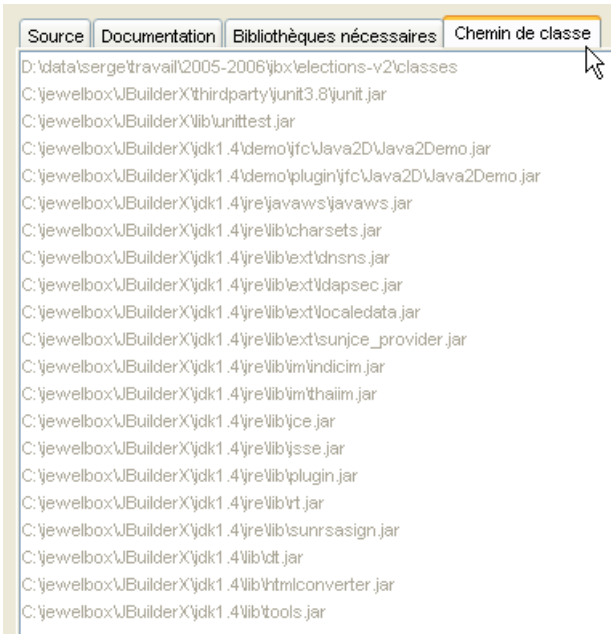
Propriétés du projet





- la bibliothèque [JUnit] fait désormais partie du CLASSPATH du projet

- sélectionner la bibliothèque [JUnit]



- la liste des archives et dossiers du CLASSPATH du projet. On y trouve [junit.jar]

2.4 MainElections : version 2

On désire réécrire l'application [Elections] en y ajoutant les nouvelles contraintes suivantes :

- on utilisera la classe [ListeElectorale] pour représenter une liste
- l'application demandera au clavier les informations suivantes :
    - le nombre de sièges à pourvoir
    - les noms et voix des listes. On ne sait pas à priori combien il y a de listes. La dernière liste est signalée par son nom égal à la chaîne "*".
- parce qu'on ne connaît pas à priori le nombre de listes, celles-ci seront tout d'abord mémorisées dans un objet de type [ArrayList]. Puis, lorsque toutes les listes auront été saisies, elles seront transférées dans un tableau de listes.
- les résultats seront affichés par ordre décroissant du nombre de sièges obtenus.

Pour trier un tableau T, on dispose de différentes méthodes statiques de la classe [Arrays] :

- **Arrays.sort(T)** : trie le tableau T selon un ordre naturel s'il en a un (croissant pour les nombres, les dates, alphabétique pour les chaînes, ...)
- **Arrays.sort(T,comparateur)** : pour trier des tableaux n'ayant pas un ordre naturel. C'est le cas ici du tableau des listes qu'il faut trier selon un champ particulier de la liste.

Dans la méthode **Arrays.sort(T,comparateur)**, **comparateur** est un objet implémentant l'interface **Comparator**. Cette interface impose d'implémenter une méthode ayant la signature suivante :

```
public int compare(Object obj1, Object obj2)
```

Cette méthode est appelée de façon répétée par la méthode [**Arrays.sort**]. Celle-ci va à chaque fois passer comme paramètres **obj1** et **obj2**, deux éléments du tableau **T** à trier. Dans notre cas, ces éléments seront de type [ListeElectorale]. On notera ici que le polymorphisme est à l'oeuvre. La méthode [compare] est définie comme recevant des paramètres de type [Object]. Cela veut dire qu'elle peut recevoir des paramètres de type [Object] ou dérivé (polymorphisme). Comme [Object] est la classe parent de toutes les classes Java, les paramètres effectifs peuvent avoir le type [ListeElectorale].

Pour un tri dans l'ordre croissant, la méthode [compare] doit rendre :

- +1 si obj1>obj2
- -1 si obj1<obj2
- 0 si obj1=obj2

Pour un tri dans l'ordre décroissant, les valeurs +1 et -1 sont inversées.

Ici, les signes <, > et = expriment une relation d'ordre. Pour des objets de type [ListeElectorale], on aura la relation **liste1<liste2** si **liste1** a moins de sièges que **liste2**.

Dans le même fichier source que la classe [MainElections], on pourra ajouter une deuxième classe :

```
1. // classe de comparaison de listes électorales
2. class CompareListesElectorales implements Comparator {
3.
4.     // comparaison de deux listes électorales selon le nombre de sièges
5.     public int compare(Object obj1, Object obj2) {
6.         // on récupère les listes électorales
7.         ListeElectorale listeElectorale1 = (ListeElectorale) obj1;
8.         ListeElectorale listeElectorale2 = (ListeElectorale) obj2;
9.         // on compare les sièges de ces deux listes
10.....
11.     }
12.}
```

- ligne 2 : la classe n'est pas déclarée publique. Dans un fichier source Java, il peut y avoir plusieurs classes mais une seule peut avoir l'attribut **public**, celle qui porte le nom du fichier source.

Question 3 : Réécrire l'application [Elections] en tenant compte des nouvelles spécifications.

---

3 Partie 3

Mots clés : architecture 3tier, Spring, XML

3.1 Introduction

Rappelons ce qui a été fait :

- dans la partie 1 de l'exercice ELECTIONS aucune classe n'a été utilisée. On a construit une solution comme on l'aurait construite en langage C.
- dans la partie 2 de l'exercice, deux classes ont été introduites :
    - [ListeElectorale] qui représente les attributs (id, nom, voix, sièges, élimine) d'une liste
    - [ElectionsException] une classe d'exceptions non contrôlées. Ce type d'exception est utilisé à chaque fois que se produit une erreur fatale dans l'application des élections. Elle est non contrôlée, c.a.d. que le développeur n'est pas obligé de la gérer avec un try-catch.

Le calcul du résultat des élections a été confié jusqu'à maintenant à une méthode [main] d'une classe [MainElections]

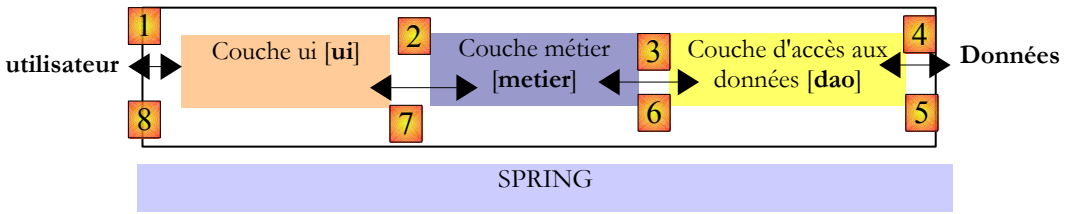
```
1. package istia.st.elections;
2.
3. import java.io.*;
4.
5. public class MainElections {
6.
7.   // qqs données
8.   private static final double barre = 0.05;
9.
10.  // ----------------------------------------------------------------------
11.  // la procédure principale
12.  public static void main(String[] arguments) throws IOException {
13.
14.    // on prépare le flux d'entrée clavier
15.    BufferedReader clavier = new BufferedReader(new InputStreamReader(System.in));
16.
17.    // saisie des données nécessaires au calcul des sièges
18....
19.    // calcul des sièges obtenus par les différentes listes
20.....
21.    // affichage des résultats
22....
23.  } // main
24.} // classe
```

La solution précédente inclut trois phases classiques :

- l'acquisition des données, lignes 17-18
- le calcul de la solution, lignes 19-20
- l'affichage ou la persistance des résultats, lignes 21-22

Seule la phase 2 est vraiment constante. La phase 1 peut varier : les données peuvent venir du clavier comme ci-dessus, d'un fichier texte, d'une interface graphique, d'une base de données, du réseau, ... De même il y a de multiples façons de restituer les résultats dans la phase 3 : les afficher à l'écran comme cela est fait ici, les enregistrer dans un fichier, dans une base de données, les envoyer sur le réseau, ...

De façon plus générale, une application peut souvent être découpée en trois couches ayant chacune un rôle bien défini :



On appelle également cette architecture, architecture " trois tiers ", traduction de l'anglais " three tier architecture ". Le terme "trois tiers" désigne normalement une architecture où chaque tier est sur une machine différente. Lorsque les tiers sont sur une même machine, l'architecture devient une architecture "trois couches".

- la couche [metier] est celle qui contient les règles métier de l'application. Pour notre application d'élections, ce sont les règles qui permettent de calculer les sièges obtenus par les différentes listes, une fois que l'on connaît les voix obtenues par chacune d'elles. Cette couche a besoin de données pour travailler. Par exemple dans l'application d'élections :
  - les listes avec pour chacune son nom et son nombre de voix
  - le nombre de sièges à pourvoir
  - le seuil électoral au-dessous duquel, une liste est éliminée

  Dans le schéma ci-dessus, les données peuvent provenir de deux endroits :

  - la couche d'accès aux données ou [dao] (DAO = Data Access Object) pour les données déjà enregistrées dans des fichiers ou bases de données. Ce pourrait être le cas ici du nom des listes, du nombre de sièges à pourvoir, du seuil électoral. On connaît en effet ces informations avant l'élection elle-même.
  - la couche d'interface avec l'utilisateur ou [ui] (UI = User Interface) pour les données saisies par l'utilisateur ou affichées à l'utilisateur. Ce pourrait être le cas ici des voix des listes qui ne sont connues qu'au dernier moment ainsi que de l'affichage des résultats de l'élection.

- de façon générale, la couche [dao] s'occupe de l'accès aux données persistantes (fichiers, bases de données) ou non persistantes (réseau, ...).
- la couche [ui] elle, s'occupe des interactions avec l'utilisateur s'il y en a un.
- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces Java.
- Pour intégrer ces couches ensemble dans l'application, il existe différentes méthodes. Nous serons amenés à utiliser un outil appelé " Spring ". Sur le schéma, il est transversal aux autres couches.

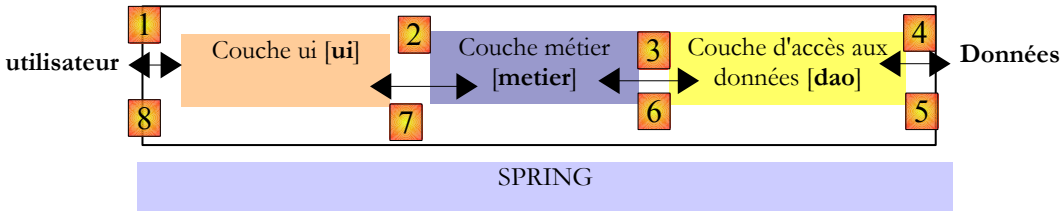
Nous allons reprendre l'application [Elections] développée précédemment pour lui donner une architecture 3tier. Pour cela, nous allons étudier les couches [ui, metier, dao] les unes après les autres, en commençant par la couche [dao], couche qui s'occupe des données persistantes.

Auparavant, il nous faut définir les interfaces des différentes couches de l'application [Elections].

3.2 Les interfaces de l'application [Elections]

Rappelons qu'une interface définit un ensemble de signatures de méthodes. Les classes implémentant l'interface donnent un contenu à ces méthodes.

Revenons à l'architecture 3 couches de notre application :

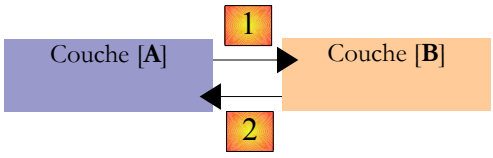


Dans ce type d'architecture, c'est souvent l'utilisateur qui prend les initiatives. Il fait une demande en [1] et reçoit une réponse en [8]. On appelle cela le cycle demande - réponse. Prenons l'exemple du calcul des sièges obtenus au soir des élections. Celui-ci va nécessiter plusieurs étapes :

(a) la couche [ui] va devoir demander à l'utilisateur le nombre de voix obtenues par chacune des listes. Pour cela elle va devoir présenter à celui-ci le nom des listes en compétition. L'utilisateur n'aura alors qu'à mettre le nombre de voix en face de chaque liste puis à demander le calcul des sièges.
(b) la couche [ui] ne dispose pas du nom des listes. Celles-ci sont enregistrées dans la source de données à droite du schéma. Elle va utiliser le chemin [2, 3, 4, 5, 6, 7] pour les obtenir. L'opération [2] est la demande des listes, l'opération [7] la réponse à cette demande. Ceci fait, elle peut les présenter à l'utilisateur par [8].
(c) l'utilisateur va transmettre à la couche [ui] le nombre de voix obtenues par chacune des listes. C'est l'opération [1] ci-dessus. Au cours de cette étape, l'utilisateur n'interagit qu'avec la couche [ui]. C'est celle-ci qui va notamment vérifier la validité des données saisies. Ceci fait, l'utilisateur va demander la liste des sièges obtenus par chacune des listes.

(d) la couche [ui] va demander à la couche métier de faire le calcul des sièges. Pour cela elle va lui transmettre les données qu'elle a reçues de l'utilisateur. C'est l'opération [2].
(e) la couche [metier] a besoin de certaines informations pour mener à bien son travail. Elle a déjà les listes depuis l'opération (b). Il lui faut également le nombre de sièges à pourvoir ainsi que la valeur du seuil électoral. Elle va demander ces informations à la couche [dao] avec le chemin [3, 4, 5, 6]. [3] est la demande initiale et [6] la réponse à cette demande.
(f) ayant toutes les données dont elle avait besoin, la couche [metier] calcule les sièges obtenus par chacune des listes.
(g) la couche [metier] peut maintenant répondre à la demande de la couche [ui] faite en (d). C'est le chemin [7].
(h) la couche [ui] va mettre en forme ces résultats pour les présenter à l'utilisateur sous une forme appropriée puis les présenter. C'est le chemin [8].
(i) on peut imaginer que ces résultats doivent être mémorisés dans un fichier ou une base de données. Cela peut être fait de façon automatique. Dans ce cas, après l'opération (f), la couche [metier] va demander à la couche [dao] d'enregistrer les résultats. Ce sera le chemin [3, 4, 5, 6]. Cela peut être fait également seulement sur demande de l'utilisateur. Ce sera le chemin [1-8] qui sera utilisé par le cycle demande - réponse.

On voit dans cette description qu'une couche est amenée à utiliser les ressources de la couche qui est à sa droite, jamais de celle qui est à sa gauche. Considérons deux couches contigües :



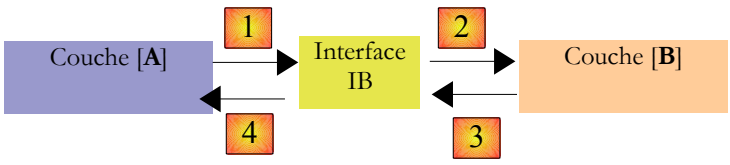
La couche [A] fait des demandes à la couche [B]. Dans les cas les plus simples, une couche est implémentée par une unique classe. Une application évolue au cours du temps. Ainsi la couche [B] peut avoir des classes d'implémentation différentes [B1, B2, ...]. Si la couche [B] est la couche [dao], celle-ci peut avoir une première implémentation [B1] qui va chercher des données dans un fichier. Quelques années plus tard, on peut vouloir mettre les données dans une base de données. On va alors construire une seconde classe d'implémentation [B2]. Si dans l'application initiale, la couche [A] travaillait directement avec la classe [B1] on est obligés de réécrire partiellement le code de la couche [A]. Supposons par exemple qu'on ait écrit dans la couche [A] quelque chose comme suit :

```
1. B1 b1=new B1(...);
2. ..
3. b1.getData(...);
```

- ligne 1 : une instance de la classe [B1] est créée
- ligne 3 : des données sont demandées à cette instance

Si on suppose, que la nouvelle classe d'implémentation [B2] utilise des méthodes de même signature que celle de la classe [B1], il faudra changer tous les [B1] en [B2]. Ca, c'est le cas très favorable et assez improbable si on n'a pas prêté attention à ces signatures de méthodes. Dans la pratique, il est fréquent que les classes [B1] et [B2] n'aient pas les mêmes signatures de méthodes et que donc une bonne partie de la couche [A] doive être totalement réécrite.

On peut améliorer les choses si on met une interface entre les couches [A] et [B]. Cela signifie qu'on fige dans une interface les signatures des méthodes présentées par la couche [B] à la couche [A]. Le schéma précédent devient alors le suivant :



La couche [A] ne s'adresse désormais plus directement à la couche [B] mais à son interface [IB]. Ainsi dans le code de la couche [A], la classe d'implémentation [Bi] de la couche [B] n'apparaît qu'une fois, au moment de l'implémentation de l'interface [IB]. Ailleurs, c'est le nom de l'interface [IB] qui apparaît. Le code précédent devient celui-ci :

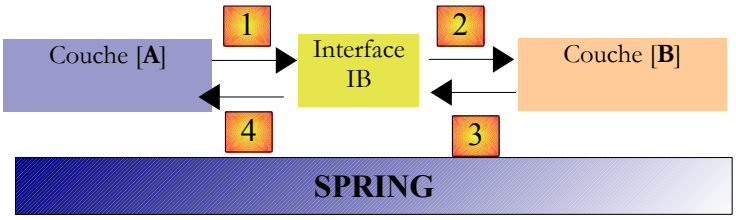
```
1. IB ib=new B1(...);
2. ..
3. ib.getData(...);
```

- ligne 1 : une instance [ib] implémentant l'interface [IB] est créée par instanciation de la classe [B1]
- ligne 3 : des données sont demandées à l'instance [ib]

Désormais si on remplace l'implémentation [B1] de la couche [B] par une implémentation [B2], et que ces deux implémentations respectent la même interface [IB], alors seule la ligne 1 de la couche [A] doit être modifiée et aucune autre. C'est un grand avantage qui à lui seul justifie l'usage systématique des interfaces entre deux couches.

On peut aller encore plus loin et rendre la couche [A] totalement indépendante de la couche [B]. Dans le code ci-dessus, la ligne 1 pose problème parce qu'elle référence en dur la classe [B1]. L'idéal serait que la couche [A] puisse disposer d'une implémentation de l'interface [IB] sans avoir à nommer de classe. Ce serait cohérent avec notre schéma ci-dessus. On y voit que la couche [A] s'adresse à l'interface [IB] et on ne voit pas pourquoi elle aurait besoin de connaître le nom de la classe qui implémente cette interface. Ce détail n'est pas utile à la couche [A].

Le framework Spring (http://www.springframework.org) va nous permettre d'obtenir ce résultat. L'architecture précédente évolue de la façon suivante :



La couche transversale [Spring] va permettre à une couche d'obtenir par configuration une référence sur la couche située à sa droite sans avoir à connaître le nom de la classe d'implémentation de la couche. Ce nom sera dans les fichiers de configuration et non pas dans le code Java. Le code Java de la couche [A] prend alors la forme suivante :

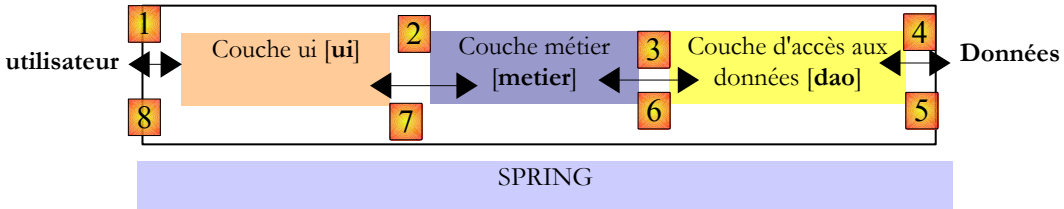
```
IB ib; // initialisé par Spring
..
ib.getData(...);
```

- ligne 1 : une instance [ib] implémentant l'interface [IB] de la couche [B]. Cette instance est créée par Spring sur la base d'informations trouvées dans un fichier de configuration. Spring va s'occuper de créer :
  - l'instance [b] implémentant la couche [B]
  - l'instance [a] implémentant la couche [A]. Cette instance sera initialisée. Le champ [ib] ci-dessus recevra pour valeur la référence [b] de l'objet implémentant la couche [B]
- ligne 3 : des données sont demandées à l'instance [ib]

On voit maintenant que, la classe d'implémentation [B1] de la couche B n'apparaît nulle part dans le code de la couche [A]. Lorsque l'implémentation [B1] sera remplacée par une nouvelle implémentation [B2], rien ne changera dans le code de la classe [A]. On changera simplement les fichiers de configuration de Spring pour instancier [B2] au lieu de [B1].

Le couple **Spring** et **interfaces Java** apporte une amélioration décisive à la maintenance d'applications en rendant les couches de celles-ci complètement étanches entre elles. C'est cette solution que nous utiliserons pour l'application [Elections].

Revenons à l'architecture trois couches de notre application :



Dans les cas simples, on peut partir de la couche [metier] pour découvrir les interfaces de l'application. Pour travailler, elle a besoin de données :

- déjà disponibles dans des fichiers, bases de données ou via le réseau. Elles sont fournies par la couche [dao].
- pas encore disponibles. Elles sont alors fournies par la couche [ui] qui les obtient auprès de l'utilisateur de l'application.

Quelle interface doit offrir la couche [dao] à la couche [metier] ? Quelles sont les interactions possibles entre ces deux couches ? La couche [dao] doit fournir les données suivantes à la couche [metier] :

- le nombre de sièges à pourvoir

- la valeur du seuil électoral au-dessous duquel une liste est éliminée
- les noms des listes

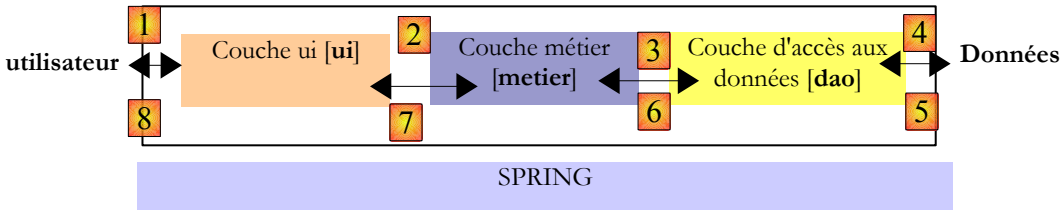
Ces informations sont en effet connues avant l'élection et peuvent donc être mémorisées. Dans le sens [metier] -> [dao], la couche [metier] peut demander à la couche [dao] d'enregistrer le résultat des élections, notamment les sièges obtenus par les différentes listes.

Avec ces informations, on pourrait tenter une première définition de l'interface de la couche [dao] :

```
1. package istia.st.elections.dao;
2.
3. import istia.st.elections.data.ListeElectorale;
4.
5. public interface IElectionsDao {
6.   /**
7.    * demande le seuil électoral
8.    * @return double : le seuil électoral
9.    */
10.  public double getSeuilElectoral();
11.
12.  /**
13.   * demande le nombre de sièges à pourvoir
14.   * @return int : le nombre de sièges à pourvoir
15.   */
16.  public int getNbSiegesAPourvoir();
17.
18.  /**
19.   * demande le tableau des listes en compétition
20.   * @return ListeElectorale[] : les listes en compétition
21.   */
22.  public ListeElectorale[] getListesElectorales();
23.
24.  /**
25.   * enregistre les résultats des listes en compétition
26.   * @param listesElectorales ListeElectorale[] : les listes en compétition
27.   */
28.  public void setListesElectorales(ListeElectorale[] listesElectorales);
29.}
```

- ligne 1 : on mettra tout ce qui concerne la couche [dao] dans le paquetage [istia.st.elections.dao].
- ligne 5 : l'interface s'appelle [IElectionsDao]. Elle définit quatre méthodes :
    - trois méthodes pour lire des données venant de la source de données : [getSeuilElectoral, getNbSiegesAPourvoir, getListesElectorales]. Ces trois méthodes permettront à la couche [metier] d'obtenir les données qui caractérisent l'élection courante.
    - une méthode pour écrire des données dans la source de données : [setListesElectorales]. Cette méthode permettra à la couche [metier] de demander l'enregistrement des résultats qu'elle aura calculés.

Revenons à l'architecture trois couches de notre application :



Quelle interface la couche [metier] doit-elle présenter à la couche [ui] ? Examinons les interactions possibles entre ces deux couches.

1. la couche [ui] va avoir pour rôle de demander à l'utilisateur les voix des différentes listes en compétition. Pour cela, elle doit connaître le nombre de listes. Elle peut demander ce renseignement à la couche [metier] qui peut demander à son tour le tableau des listes en compétition à la couche [dao]. Si la couche [metier] a ce tableau, autant transférer celui-ci dans la couche [ui]. Celle-ci disposera ainsi des noms des listes et pourra affiner ses messages à l'utilisateur en demandant par exemple " Nombre de voix de la liste A ".
2. lorsque la couche [ui] aura obtenu les voix de toutes les listes, elle demandera le calcul des sièges à la couche [metier]. Celle-ci pourra faire ce calcul et rendre le résultat à la couche [ui].
3. la couche [ui] pourra alors présenter ces résultats à l'utilisateur. Celui-ci pourra également demander leur enregistrement.
4. la couche [ui] peut vouloir par ailleurs présenter des informations complémentaires à l'utilisateur, telles que le seuil électoral ou le nombre de sièges à pourvoir.

Avec ces informations, on pourrait tenter une première définition de l'interface de la couche [metier] :

```
1.  package istia.st.elections.metier;
2.
3.  import istia.st.elections.data.ListeElectorale;
4.
5.  public interface IElectionsMetier {
6.    /**
7.     * demande le seuil électoral
8.     * @return double : le seuil électoral
9.     */
10.   public double getSeuilElectoral();
11.
12.   /**
13.    * demande le nombre de sièges à pourvoir
14.    * @return int : le nombre de sièges à pourvoir
15.    */
16.   public int getNbSiegesAPourvoir();
17.
18.   /**
19.    * rend le tableau des listes en compétition
20.    * @return ListeElectorale[] : le tableau des listes en compétition
21.    */
22.   ListeElectorale[] getListesElectorales();
23.
24.   /**
25.    * calcul des sièges des listes candidates
26.    * @param listesElectorales ListeElectorale[] : les listes avec les voix mais sans les sièges
27.    * @return ListeElectorale[] : les listes avec les sièges
28.    */
29.   ListeElectorale[] calculerSieges(ListeElectorale[] listesElectorales);
30.
31.   /**
32.    * enregistre les résultats de l'élection
33.    * @param listesElectorales ListeElectorale[] : les listes candidates à enregistrer
34.    */
35.   public void recordResultats(ListeElectorale[] listesElectorales);
36.
37. }
```

- ligne 1 : on mettra tout ce qui concerne la couche [metier] dans le paquetage [istia.st.elections.metier].
- ligne 5 : l'interface s'appelle [IElectionsMetier]. Elle définit trois méthodes :
    - une méthode [getListesElectorales] (ligne 17) qui permettra à la couche [ui] d'obtenir le tableau des listes en compétition.
    - une méthode [calculerSieges] (ligne 11) qui permettra à la couche [ui] de demander le calcul des sièges une fois que les nombres de voix des différentes listes seront connus.
    - une méthode [recordResultats] (ligne 23) qui permettra à la couche [ui] de demander l'enregistrement des résultats.

3.3 La couche [dao]

Nous allons étudier deux implémentations de la couche [dao] :

- l'une où les données sont dans des fichiers texte
- l'autre où les données sont dans une base de données

Pour fixer les idées, nous allons commencer par écrire un test JUnit de la couche [dao]. Certaines méthodes de développement préconisent d'écrire les tests avant même d'écrire le code qui doit être testé. Cette méthode d'appelle **Tests Driven Development** (TDD). L'écriture des tests JUnit permet de clarifier les spécifications des différentes couches : " Que doivent faire exactement les différentes méthodes des interfaces ? ". Dans les équipes importantes, les tests sont écrits par des gens qui ne font que cela. Les spécifications et les classes de tests sont ensuite transmises aux développeurs qui doivent s'assurer que les tests passent. En imposant les tests qui doivent être passés sur les classes et interfaces, on est assuré que les développeurs vont respecter à la lettre les spécifications de l'application.

3.3.1 La classe de test JUnit

Pour tester la couche [dao], il nous faut spécifier son interface. Ce sera celle qui a déjà été présentée :

```
1. package istia.st.elections.dao;
2.
3. import istia.st.elections.data.ListeElectorale;
4.
5. public interface IElectionsDao {
6.   /**
7.    * demande le seuil électoral
```

```
8.    * @return double : le seuil électoral
9.    */
10.  public double getSeuilElectoral();
11.
12.  /**
13.   * demande le nombre de sièges à pourvoir
14.   * @return int : le nombre de sièges à pourvoir
15.   */
16.  public int getNbSiegesAPourvoir();
17.
18.  /**
19.   * demande le tableau des listes en compétition
20.   * @return ListeElectorale[] : les listes en compétition
21.   */
22.  public ListeElectorale[] getListesElectorales();
23.
24.  /**
25.   * enregistre les résultats des listes en compétition
26.   * @param listesElectorales ListeElectorale[] : les listes en compétition
27.   */
28.  public void setListesElectorales(ListeElectorale[] listesElectorales);
29.}
```

Pour tester cette interface, nous pourrions utiliser la classe de test JUnit suivante :

```
1. package istia.st.elections.dao.tests;
2.
3. import org.springframework.beans.factory.xml.XmlBeanFactory;
4. import org.springframework.core.io.ClassPathResource;
5. import istia.st.elections.dao.IElectionsDao;
6. import junit.framework.TestCase;
7. import istia.st.elections.data.ListeElectorale;
8.
9. public class JUnitTestsElectionsDaoFile extends TestCase {
10.
11.  /**
12.   * instance d'accès à la couche [dao]
13.   */
14.  private IElectionsDao electionsDao = null;
15.
16.  /**
17.   * constructeur par défaut
18.   * <p>on récupère auprès de Spring une instance de l'interface d'accès aux données</p>
19.   */
20.  public JUnitTestsElectionsDaoFile() {
21.    super();
22.    electionsDao = ...;
23.  }
24.
25.  /**
26.   * affichage des données de l'élection
27.   */
28.  public void testLectureDataElections() {
29.    // on affiche les données de l'élection
30.....
31.  }
32.
33.  public void testEcritureResultatsElections(){
34.      // on crée un tableau de trois listes
35....
36.      // on rend ces données persistantes
37. ...
38.  }
39.}
```

Ce code mérite des explications :

- ligne 14 : une instance implémentant l'interface [IElectionsDao] est définie. Elle va être initialisée par construction ligne 22. Nous y reviendrons.
- lignes 29-32 : une méthode pour tester les méthodes de lecture de la couche [dao] : [getSeuilElectoral, getNbSiegesAPourvoir, getListesElectorales].
- ligne 34-39 : une méthode pour tester la méthode d'écriture de la couche [dao] : [setListesElectorales]
- lignes 20-24 : le constructeur de la classe de test. Son but premier est de donner une valeur au champ [electionsDao] de la ligne 14. Nous y reviendrons ultérieurement.

La méthode [testLectureDataElections] fait un simple affichage à l'écran des données qu'elle demande à la couche [dao] via le champ [electionsDao]. Cet affichage est le suivant :

```
1. Nombre de sièges à pourvoir : 6
2. Seuil électoral : 0.05
3. [A,32000,0,false]
```

```
4. [B,25000,0,false]
5. [C,16000,0,false]
6. [D,12000,0,false]
7. [E,8000,0,false]
8. [F,4500,0,false]
```

Question 1 : écrire la méthode [testLectureDataElections]. On se rappellera que la méthode n'a pas à savoir comment la couche [dao] obtient ces données. Elle se contente de les demander via les méthodes d'interface de la couche [dao].

La méthode [testEcritureResultatsElections] se contente de créer un tableau de trois listes et demande son enregistrement à la couche [dao]. L'enregistrement du tableau pourrait donner naissance au fichier texte suivant :

```
1. La liste [A] a obtenu [2] siège(s)
2. La liste [B] a obtenu [2] siège(s)
3. La liste [C] a obtenu [0] siège(s)
```

Ce n'est qu'un exemple. La méthode n'a pas à savoir où la couche [dao] enregistre les données qu'elle lui transmet et comment elle les enregistre. Elle se contente de demander leur enregistrement via la méthode d'interface de la couche [dao] adéquate.

Question 2 : écrire la méthode [testEcritureResultatsElections].

3.3.2 Configuration des tests unitaires

Dans un premier temps, nous allons supposer que nous disposons d'une classe [ElectionsDaoFile] implémentant l'interface [IElectionsDao] de la façon suivante :

- les données nécessaires à l'élection sont enregistrées avant l'élection dans un fichier texte sous la forme suivante :

```
(a)6
(b)0.05
(c)A
(d)B
(e)C
(f)D
(g)E
(h)F
(i)G
```

- ligne (a) : nombre de sièges à pourvoir
- ligne (b) : seuil électoral
- lignes suivantes : noms des listes à raison d'un nom par ligne

- une fois connus les résultats de l'élection, ils seront enregistrés eux-aussi dans un fichier texte sous la forme suivante :

```
(a)[A,32000,2,false]
(b)[B,25000,2,false]
(c)[C,16000,0,true]
```

Chaque liste fait l'objet d'une ligne de la forme [nom, voix, sièges, éliminé].

Le fichier des données utilisées par l'application peut être erroné comme le suivant :

```
6x
0.05
```

Dans ce cas, la construction de la classe [ElectionsDaoFile] échoue et une exception est lancée. Les erreurs sont mémorisées dans un fichier de logs :

```
Ligne [1] dans fichier [data\elections-in-bad.txt] : Nombre de sièges à pourvoir [6x] incorrect
Le fichier [data\elections-in-bad.txt] est incomplet
```

Ce fichier permet à l'utilisateur d'identifier les erreurs.

La classe [ElectionsDaoFile] pourrait avoir le squelette suivant :

```
1. package istia.st.elections.dao;
2.
3. ...
4. public class ElectionsDaoFile implements IElectionsDao {
5.
6.     /**
7.      * le nom du fichier qui contient les données nécessaires au calcul des sièges
```

```
8.      */
9.     private String inFileName = null;
10.
11.    /**
12.     * le nom du fichier qui contiendra les résultats
13.     */
14.    String outFileName = null;
15.
16.    /**
17.     * le nom du fichier de logs
18.     */
19.    private String logFileName = null;
20.
21.    /**
22.     * le seuil électoral
23.     */
24.    private double seuilElectoral;
25.
26.
27.    /**
28.     * le nombre de sièges à pourvoir
29.     */
30.    private int nbSiegesAPourvoir;
31.
32.
33.    /**
34.     * les listes en compétition
35.     */
36.    private ListeElectorale[] listesElectorales = null;
37.
38.
39.    /**
40.     * constructeur avec paramètres
41.     * @param inFileFileName String : le nom du fichier qui contient les données nécessaires au calcul
   des sièges
42.     * @param outFileName String : le nom du fichier qui contiendra les résultats
43.     * @param logFileName String : le nom du fichier qui contiendra les messages d'erreurs éventuels
44.     * @throws ElectionsException si problème quelconque
45.     *
46.     */
47.    public ElectionsDaoFile(String inFileName, String outFileName,
48.                            String logFileName) {
49.        // on enregistre les noms des fichiers
50.        this.inFileName = inFileName;
51.        this.outFileName = outFileName;
52.        this.logFileName = logFileName;
53....
54.    }
55.
56.    /**
57.     * getSeuilElectoral
58.     *
59.     * @return double : seuil électoral
60.     */
61.    public double getSeuilElectoral() {
62.        return seuilElectoral;
63.    }
64.
65.
66.    /**
67.     * getNbSiegesAPourvoir
68.     *
69.     * @return int : nombre de sièges à pourvoir
70.     */
71.    public int getNbSiegesAPourvoir() {
72.        return nbSiegesAPourvoir;
73.    }
74.
75.
76.    /**
77.     * getListesElectorales
78.     *
79.     * @return ListeElectorale[] : tableau des listes électorales
80.     */
81.    public ListeElectorale[] getListesElectorales() {
82.        return listesElectorales;
83.    }
84.
85.    /**
86.     * enregistrement des résultats dans le fichier [outFileName]
87.     * @param listesElectorales ListeElectorale[]
88.     */
89.    public void setListesElectorales(ListeElectorale[] listesElectorales) {
90....
91.    }
92.}
```

Nous retiendrons de ce squelette que la classe a un constructeur a trois paramètres :

```
1.     /**
2.      * constructeur avec paramètres
3.      * @param inFileFileName String : le nom du fichier qui contient les données nécessaires au calcul des sièges
4.      * @param outFileName String : le nom du fichier qui contiendra les résultats de l'élection
5.      * @param logFileName String : le nom du fichier qui contiendra les messages d'erreurs éventuels
6.      * @throws ElectionsException si problème quelconque
7.      *
8.      */
9.     public ElectionsDaoFile(String inFileName, String outFileName, String logFileName) {
```

Revenons sur la classe de test [JUnit] et à son constructeur :

```
1. public class JUnitTestsElectionsDaoFile extends TestCase {
2.
3.   /**
4.    * instance d'accès à la couche [dao]
5.    */
6.   private IElectionsDao electionsDao = null;
7.
8.     /**
9.      * constructeur par défaut
10.     * <p>on récupère auprès de Spring une instance de l'interface d'accès aux données</p>
11.     */
12.    public JUnitTestsElectionsDaoFile() {
13.        // classe parent
14.        super();
15.        // instanciation couche [dao]
16.        electionsDao = (IElectionsDao) (new XmlBeanFactory(new ClassPathResource(
17.                        "spring-config-electionsDaoFile.xml"))).getBean("electionsDao");
18.    }
19.
20.  /**
21.   * affichage des données de l'élection
22.   */
23.  public void testLectureDataElections() {
24.    // on affiche les données de l'élection
25.....
26.  }
27.
28.  public void testEcritureResultatsElections(){
29.      // on crée un tableau de trois listes
30....
31.      // on rend ces données persistantes
32. ...
33.  }
34.}
```

Nous n'avions pas détaillé le constructeur de la classe de test, lignes 12-18. Nous le faisons maintenant.

- ligne 13 : la classe parent est tout d'abord construite.
- lignes 16-17 : le champ [electionsDao] est initialisé grâce au framework Spring.
- ligne 16 : on dit à Spring d'utiliser le fichier de configuration appelé [spring-config-electionsDaoFile.xml]. Celui-ci aura le contenu suivant :

```
(a)<?xml version="1.0" encoding="ISO_8859-1"?>
(b)<!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
(c)<beans>
(d)     <!-- la source de données -->
(e)     <!-- les chemins des fichiers doivent être absolus ou relatifs au répertoire courant au moment de
   l'exécution-->
(f)     <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoFile">
(g)             <constructor-arg index="0">
(h)                     <value>data\elections-in-good.txt</value>
(i)             </constructor-arg>
(j)             <constructor-arg index="1">
(k)                     <value>data\elections-out.txt</value>
(l)             </constructor-arg>
(m)             <constructor-arg index="2">
(n)                     <value>data\elections-log.txt</value>
(o)             </constructor-arg>
(p)     </bean>
(q)</beans>
```

Ce fichier est un fichier XML. C'est la ligne (a) qui le dit.

- ligne (b) : un fichier XML est constitué d'une arborescence de balises de la forme :

```
<balise att1="val1" att2= "val2">contenu</balise>
```

ou

```
<balise att1="val1" att2="val2"/>
```

Une balise a donc de façon facultative :
- des attributs de la forme attribut="valeur"
- un contenu. Ce contenu peut alors être lui-même composé de balises. On a alors une arborescence de balises.

Un fichier XML doit être " bien formé ", c.a.d. que toute balise ouverte doit être fermée. Ce n'est en général pas suffisant. On veut également que le fichier respecte un modèle. Un modèle va dire par exemple que la balise <personne> a deux attributs facultatifs *nom* et *age*. La balise

```
<personne nom="martin"/>
```

sera alors déclarée correcte, alors que la balise

```
<personne prenom="paul"/>
```

sera déclarée incorrecte car le modèle ne prévoit pas d'attribut [prenom] pour la balise <personne>. Ce modèle s'appelle une DTD (Document Type Description).

- ligne (b) : indique où trouver le modèle du document XML. Cette ligne n'est pas obligatoire. Si elle est présente, les programmes Java utilisant le fichier XML peuvent vérifier qu'il est non seulement " bien formé " mais également conforme à sa DTD. Avec Spring, nous placerons toujours cette ligne car elle permet à Spring de vérifier la conformité du fichier XML à son modèle. Si le poste n'est pas connecté à Internet, il faut supprimer cette ligne ou la mettre en commentaires afin d'éviter une erreur d'accès réseau.
- ligne (c) : <beans> est la balise " racine " d'un fichier Spring. On peut traduire " coffee bean " par " grain de café ". C'est un clin d'oeil à " Java " qu'on a souvent lié au café. Un " bean " est un objet Java.
- lignes (d) et (e) : des commentaires sous la forme <!-- commentaire -->
- lignes (f)-(p) : définissent un " bean " donc un objet.
- ligne (f) : un bean a un attribut " id " qui est l'identifiant du bean au sein du fichier de configuration. Il a également un attribut " class " qui indique la nom de la classe à instancier. Ici c'est la classe [istia.st.elections.dao.ElectionsDaoFile]. On donne le nom complet, paquetage inclus.
- lignes (g)-(o) : paramètres de construction du bean. On peut utiliser deux méthodes pour créer un bean avec Spring :
  1. utiliser le paramètre sans constructeur du bean et ensuite utiliser ses " setters ". Dans ce cas, le " bean " doit respecter la norme Javabean (présence du constructeur sans paramètres, présence des setters).
  2. utiliser l'un des constructeurs de la classe en lui passant les paramètres dont il a besoin. C'est ce qui a été fait ici. La signature du constructeur de la classe [istia.st.elections.dao.ElectionsDaoFile] est la suivante :

     ```
     public ElectionsDaoFile(String inFileName, String outFileName,String logFileName)
     ```

     On fournit à Spring les trois paramètres nécessaires au constructeur. Cela se fait avec une balise <constructor-arg index= "i ">. L'index est le n° du paramètre dans la liste des paramètres du constructeur.
- lignes (g)-(i) : le nom du fichier qui contient les données caractérisant l'élection, paramètre n° 0.
- lignes (j)-(l) : le nom du fichier qui contiendra les résultats de l'élection, paramètre n° 1.
- lignes (m)-(o) : le nom du fichier qui contient les éventuels messages d'erreurs, paramètre n° 2.

Maintenant que le fichier de configuration de Spring a été décrit, revenons au constructeur de la classe de test unitaire dela couche [dao] :

```
1.     public JUnitTestsElectionsDaoFile() {
2.         // classe parent
3.         super();
4.         // instanciation couche [dao]
5.         electionsDao = (IElectionsDao) (new XmlBeanFactory(new ClassPathResource(
6.                         "spring-config-electionsDaoFile.xml"))).getBean("electionsDao");
7.     }
```

- **new ClassPathResource("spring-config-electionsDaoFile.xml")** est un objet construit par Spring à partir du fichier de configuration [spring-config-electionsDaoFile.xml]. Celui-ci peut être à divers endroits. Il sera cherché ici dans le " ClassPath " de l'application. En effet, l'objet **ClassPathResource** sert à localiser des ressources dans le " ClassPath " de l'application. On rappelle que ce terme désigne la liste des dossiers explorés par la machine virtuelle Java pour chercher les classes dont l'application a besoin. Le fichier [spring-config-electionsDaoFile.xml] sera donc cherché par Spring dans ces mêmes dossiers. Avec JBuilder, le dossier [**classes**] d'un projet Jbuilder fait automatiquement partie du " ClassPath " de l'application. Aussi placerons-nous le fichier [spring-config-electionsDaoFile.xml] dans ce dossier.
- **new XmlBeanFactory(ClassPathResource resource)** est une fabrique d'objets. Spring la construit à partir d'un fichier XML (utilisation de la classe XmlBeanFactory). Spring va exploiter le fichier [spring-config-electionsDaoFile.xml] et mémoriser la configuration de chacun des beans du fichier. Dans notre exemple, nous n'avons qu'un bean, le bean nommé " electionsDao " (ligne f du fichier de configuration).
- **[XmlBeanFactory].getBean(" electionsDao ")** demande à Spring une référence sur l'objet nommé " electionsDao ". Si Spring n'a pas encore construit ce bean, il le construit. Si le bean a déjà été construit, Spring se contente d'en rendre une

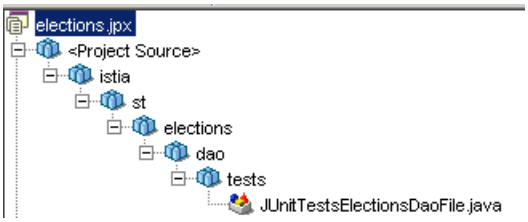
référence. Ce type d'objet est appelé un **singleton**. Il n'existe qu'en **un seul exemplaire**. Dans les applications multi-couches, les couches de service sont implémentées par des singletons. Par exemple, la couche [metier] de notre application d'élections va être instanciée par un singleton. Une fois qu'on a un objet qui sait comment calculer des résultats d'élections, on n'en a pas besoin de deux.
- la méthode **getBean** rend une référence sur un type [Object]. Pour attribuer cette valeur au champ [electionsDao], on est obligés de faire un changement de type :

```
        electionsDao = (IElectionsDao) (new XmlBeanFactory(...));
```

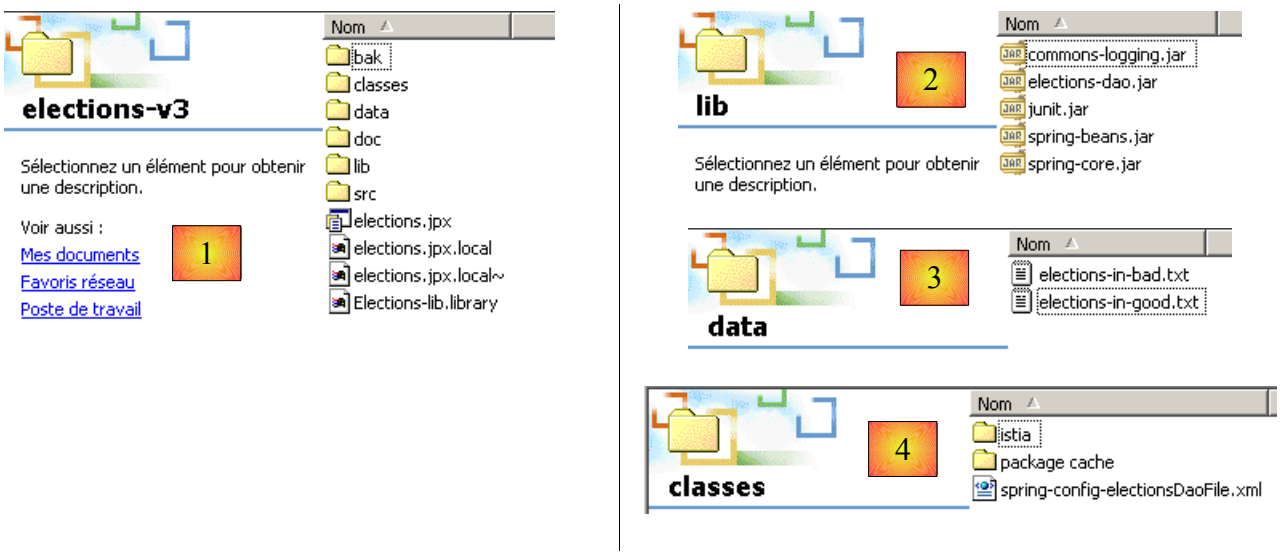
- au final, [electionsDao] est du type de l'interface [IElectionsDao]. Cela entraîne que nous ne pouvons utiliser dans le code que les seules méthodes de l'interface. Nous ne pouvons pas par exemple utiliser des méthodes de la classe d'implémentation utilisée [ElectionsDaoFile] qui ne seraient pas dans l'interface. Le compilateur le refuserait.
- terminons en attirant l'attention sur le fait que la classe de test est totalement indépendante de la classe d'implémentation [ElectionsDaoFile] dont le nom n'apparaît jamais dans le code Java. Ce nom se trouve uniquement dans le fichier de configuration de Spring. Si nous changeons de classe d'implémentation dans ce fichier de configuration, notre classe de test reste valide. C'est un avantage important apporté par l'utilisation de Spring.

3.3.3 Mise en oeuvre des tests unitaires

Pour mettre en oeuvre les tests unitaires de la couche [dao], nous créons le nouveau projet JBuilder suivant :



- Le projet Jbuilder n'a qu'une classe, la classe [JUnitTestsElectionsDaoFile] dont on a décrit les fonctionnalités au paragraphe 3.3.2, page 21.
- Le dossier du projet est le suivant :



- le dossier (1) est un dossier standard JBuilder auquel on a ajouté les dossiers [lib] et [data]
- le dossier [lib] (2) contient les archives nécessaires à l'exécution du test :
  - **junit.jar** pour les classes JUnit, **spring-core.jar**, **spring-beans.jar**, **commons-logging.jar** pour Spring
  - **elections-dao.jar** contient les classes de la couche [dao] à tester. Son contenu est le suivant :

Name	Type	Modified	Size	Ratio	Pac...	Path
ElectionsDaoFile.class	class File	28/09/2005 15:29	5 061	51%	2 502	istia\st\elections\dao\
IElectionsDao.class	class File	28/09/2005 15:29	358	41%	213	istia\st\elections\dao\
ElectionsException.class	class File	28/09/2005 15:29	769	51%	373	istia\st\elections\data\
ListeElectorale.class	class File	28/09/2005 15:29	2 325	54%	1 063	istia\st\elections\data\
MANIFEST.MF	MF File	28/09/2005 15:30	25	00%	27	META-INF\

Les archives du dossier [lib] nous seront fournies. **Le " ClassPath " du projet doit être modifié pour les inclure**.

- le dossier [data] (3) contient deux fichiers contenant des données pour l'élection, l'un avec des données valides, l'autre avec des données invalides.

`elections-in-good.txt`	`elections-in-bad.txt`
6	6x
0.05	0.05
A	
B	
C	
D	
E	
F	
G	

- le dossier [classes] (4) contient le fichier de configuration Spring décrit au paragraphe 3.3.2, page 23.

Travail pratique : Mettre en oeuvre ces tests unitaires sur machine. On fera les tests dans deux configurations successives :

- la première utilisera le fichier [elections-in-good.txt] pour lire les données de l'élection
- la seconde utilisera le fichier [elections-in-bad.txt]

Après chaque test, on vérifiera le contenu des fichiers [elections-out.txt] et [elections-log.txt].

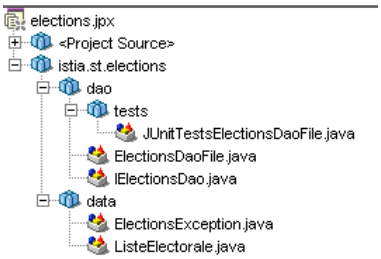
3.3.4 Implémentation [ElectionsDaoFile] de la couche [dao]

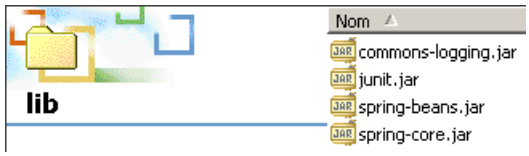
Nous revenons sur la classe [ElectionsDaoFile] que nous avons testée à partir d'une archive qui nous avait été donnée.

Question 3 : A partir de ce qui a été écrit précédemment et des tests que vous avez faits, écrire le code de la classe [ElectionsDaoFile].

Travail pratique : Mettre en oeuvre les tests unitaires de cette classe. On utilisera une configuration analogue à celle des précédents tests aux changements près suivants :

- l'archive [elections-dao.jar] n'est plus dans le dossier [lib]
- les classes que contenait cette archive sont désormais présentes en " clair " dans le projet.



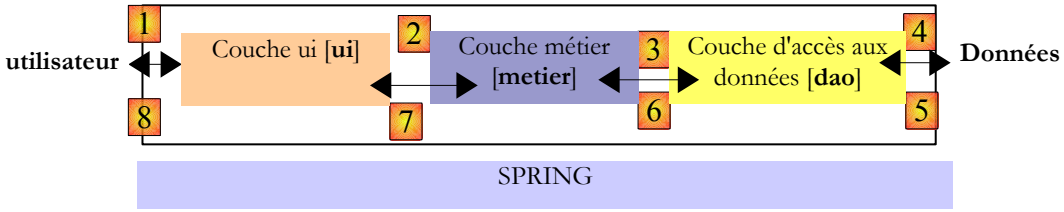


3.3.5 Création de l'archive [elections-dao.jar] de la couche [dao]

Travail pratique : Créer l'archive [elections-dao.jar] décrite au paragraphe 3.3.3, page 25. Cette archive sera utilisée pour construire la couche [metier] de l'application.

3.3.6 Conclusion

Rappelons l'architecture générale de l'application [Elections] que nous sommes en train de construire :



Nous venons de construire une première implémentation de la couche [dao]. Nous poursuivons par la construction des couches [metier] et [ui].

3.4 La couche [metier]

3.4.1 L'interface [IElectionsMetier]

La couche [metier] aura l'interface déjà présentée au paragraphe 3.2, page 19. Nous la rappelons :

```
1.  package istia.st.elections.metier;
2.
3.  import istia.st.elections.data.ListeElectorale;
4.
5.  public interface IElectionsMetier {
6.    /**
7.     * demande le seuil électoral
8.     * @return double : le seuil électoral
9.     */
10.   public double getSeuilElectoral();
11.
12.   /**
13.    * demande le nombre de sièges à pourvoir
14.    * @return int : le nombre de sièges à pourvoir
15.    */
16.   public int getNbSiegesAPourvoir();
17.
18.   /**
19.    * rend le tableau des listes en compétition
20.    * @return ListeElectorale[] : le tableau des listes en compétition
21.    */
22.   ListeElectorale[] getListesElectorales();
23.
24.   /**
25.    * calcul des sièges des listes candidates
26.    * @param listesElectorales ListeElectorale[] : en entrée les listes avec les voix mais sans les
    sièges, en sortie les listes avec les sièges
27.    */
28.   void calculerSieges(ListeElectorale[] listesElectorales);
29.
30.   /**
31.    * enregistre les résultats de l'élection
32.    * @param listesElectorales ListeElectorale[] : les listes candidates à enregistrer
33.    */
34.   public void recordResultats(ListeElectorale[] listesElectorales);
35.
36. }
```

- ligne 1 : on mettra tout ce qui concerne la couche [metier] dans le paquetage [istia.st.elections.metier].
- ligne 5 : l'interface s'appelle [IElectionsMetier]. Elle définit cinq méthodes :
  - une méthode [getSeuilElectoral] (ligne 10) pour obtenir la valeur du seuil électoral
  - une méthode [getNbSiègesAPourvoir] (ligne 16) pour obtenir le nombre de sièges à pourvoir
  - une méthode [getListesElectorales] (ligne 22) qui permettra à la couche [ui] d'obtenir le tableau des listes en compétition.
  - une méthode [calculerSieges] (ligne 29) qui permettra à la couche [ui] de demander le calcul des sièges une fois que les nombres de voix des différentes listes seront connus.
  - une méthode [recordResultats] (ligne 35) qui permettra à la couche [ui] de demander l'enregistrement des résultats.

3.4.2 La classe de test

Nous allons suivre les méthodes dites TDD (Test Driven Development) qui consistent à écrire les classes de test avant de coder ce qui va être testé. La classe de test JUnit aura la forme suivante :

```
1.  package istia.st.elections.metier.tests;
2.
3.  import org.springframework.beans.factory.xml.XmlBeanFactory;
4.  import org.springframework.core.io.ClassPathResource;
5.  import junit.framework.TestCase;
6.  import istia.st.elections.data.ListeElectorale;
7.  import istia.st.elections.metier.IElectionsMetier;
8.  import istia.st.elections.data.ElectionsException;
9.
10.
11. public class JUnitTestsElectionsMetierDaoFile extends TestCase {
12.
13.   /**
14.    * instance d'accès à la couche [metier]
15.    * sera instanciée par Spring
16.    */
17.   private IElectionsMetier electionsMetier = null;
18.
19.   /**
20.    * constructeur par défaut
21.    * <p>on récupère auprès de Spring une instance de l'interface métier</p>
22.    */
23.   public JUnitTestsElectionsMetierDaoFile() {
24.     // classe parent
25.     super();
26.     // instanciation couche [metier]
27.     electionsMetier = (IElectionsMetier) (new XmlBeanFactory(new
28.         ClassPathResource(
29.         "spring-config-electionsMetierDaoFile.xml"))).getBean(
30.         "electionsMetier");
31.   }
32.
33.
34.   /**
35.    * vérification 1 méthode de calcul des sièges
36.    * on fixe en dur les listes
37.    */
38.   public void test1CalculSieges() {
39.     // on crée le tableau des listes candidates
40.     ListeElectorale[] listes = new ListeElectorale[7];
41. ...
42.     // on calcule les sièges
43. ..
44.     // on vérifie les résultats
45.     assertEquals(2, listes[0].getSieges());
46. ...
47.   }
48.
49.   /**
50.    * vérification 2 méthode de calcul des sièges
51.    * on demande les listes à la couche [metier]
52.    * puis on fixe en dur les voix
53.    */
54.   public void test2CalculSieges() {
55.     // on récupère le tableau des listes candidates auprès de la couche [metier]
56.     ListeElectorale[] listes = ...
57.     // on fixe en dur les voix
58.     listes[0].setVoix(32000);
59. ...
60.     // on calcule les sièges
61. ...
62.     // on vérifie les résultats
63.     assertEquals(2, listes[0].getSieges());
64. ...
65.   }
66.
67.   /**
68.    * vérification 3 méthode de calcul des sièges
69.    * on provoque une exception
70.    */
71.   public void test3CalculSieges() {
72.     // on crée un tableau de 25 listes candidates ayant toutes 1 voix
73.     // les 25 listes auront le même nombre de voix (4%)
74. ...
75.     // calcul des sièges - normalement on doit avoir une ElectionsException
76.     // avec un seuil électoral de 5% - on la gère
77.     boolean erreur = false;
78. ...
79.     // on vérifie les résultats
80.     assertTrue(erreur);
81.   }
82.
83.   /**
84.    * enregistrement des résultats de l'élection
85.    */
86.   public void testEcritureResultatsElections() {
```

```
87.     // on récupère le tableau des listes candidates auprès de la couche [metier]
88.     ListeElectorale[] listes = ...
89.     // on fixe en dur les voix
90. ...
91.     // on calcule les sièges
92. ...
93.     // on affiche les résultats
94. ...
95.     // on enregistre les résultats
96. ...
97.   }
98. }
```

- ligne 11 : déclaration de la classe de test JUnit
- ligne 17 : l'instance d'accès à la couche métier. Elle sera initialisée par Spring lignes 27-30 lors de la construction de la classe de test.
- lignes 23-31 : le constructeur de la classe
- ligne 25 : d'abord construire la classe parent
- ligne 27 : initialiser le champ [electionsMetier] avec Spring

---

Question 4 : Écrire les quatre méthodes de tests en vous aidant des commentaires. On se rappellera que lorsque ces méthodes s'exécutent, le champ [electionsMetier] a déjà été initialisé.

3.4.3 La classe d'implémentation [ElectionsMetier]

Cette classe implémente l'interface [IElectionsMetier]. Nous ne proposerons qu'une implémentation qui aura la structure suivante :

```
1.  package istia.st.elections.metier;
2.
3.  import istia.st.elections.data.ListeElectorale;
4.  import istia.st.elections.dao.IElectionsDao;
5.  import istia.st.elections.data.ElectionsException;
6.  import java.util.Comparator;
7.  import java.util.Arrays;
8.
9.  public class ElectionsMetier implements IElectionsMetier {
10.
11.   /**
12.    * le point d'accès à la couche [dao]
13.    * sera instanciée par [Spring]
14.    */
15.   private IElectionsDao electionsDao;
16.
17.   /**
18.    *
19.    * @param electionsDao IElectionsDao : instance de la couche [dao]
20.    */
21.   public void setElectionsDao(IElectionsDao electionsDao) {
22.     this.electionsDao = electionsDao;
23.   }
24.
25.
26.   /**
27.    * constructeur par défaut
28.    */
29.   public ElectionsMetier() {
30.   }
31.
32.
33.   /**
34.    * calculerSieges : calcule les résultats de l'élection
35.    *
36.    * @param listesElectorales ListeElectorale[] : le tableau des listes candidates
37.    * @return ListeElectorale[] : ce même tableau avec cette fois-ci les sièges obtenus
38.    */
39.   public ListeElectorale[] calculerSieges(ListeElectorale[] listesElectorales) {
40. ...
41.   }
42.
43.
44.   /**
45.    * gelistesElectoralesElectorales : rend le tableau des listes candidates
46.    *
47.    * @return ListeElectorale[] : tanleau des listes en compétition
48.    */
49.   public ListeElectorale[] getListesElectorales() {
50.         // demande les listes à la couche [dao]
51.  ...
52.   }
53.
```

```
54.     /**
55.      * rend le nombre de sièges à pourvoir
56.      * @return int : le nombre de sièges à pourvoir
57.      */
58.     public int getNbSiegesAPourvoir() {
59.        // demande l'information à la couche [dao]
60.  ...
61.     }
62.
63.     /**
64.      * rend le seuil électoral
65.      * @return double : le seuil électoral
66.      */
67.     public double getSeuilElectoral() {
68.        // demande l'information à la couche [dao]
69.  ...
70.     }
71.
72.   /**
73.    * recordResultats : enregistre les résultats de l'élection
74.    *
75.    * @param listesElectorales ListeElectorale[] : le tableau des listes candidates
76.    */
77.   public void recordResultats(ListeElectorale[] listesElectorales) {
78.        // demande l'enregistrement des listes à la couche [dao]
79. ....
80.   }
81. }
```

- ligne 9 : la classe [ElectionsMetier] implémente l'interface [IElectionsMetier]
- ligne 15 : le champ [electionsDao] est une référence sur une implémentation de la couche [dao]. Ce champ sera initialisé par configuration avec Spring. Spring utilisera pour cela la méthode [set] associée au champ et définie lignes 21-23
- lignes 29-30 : le constructeur sans paramètres - sera utilisé par Spring pour instancier la classe.
- lignes suivantes : les implémentations des cinq méthodes de l'interface [IElectionsMetier].

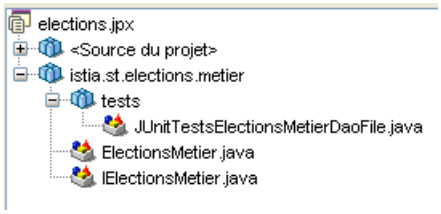
Question 5 :

- écrire les méthodes [getSeuilElectoral, getNbSiègesAPourvoir, getListesElectorales]
- écrire la méthode [recordResultats]
- écrire la méthode [calculerSieges]

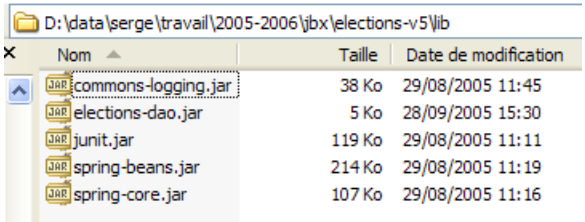
On s'aidera des commentaires lorsqu'il y en a.

3.4.4 Mise en oeuvre des tests unitaires de la couche [metier]

Le projet JBuilder aura la forme suivante :



Le dossier du projet aura un dossier [lib] contenant les archives de classes qui lui sont nécessaires :



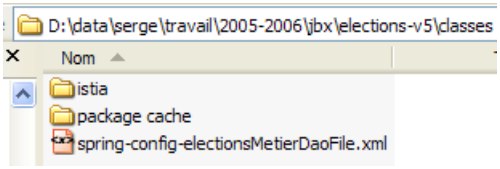
On remarquera la présence de l'archive de la couche [dao] : **elections-dao.jar**. Les archives du dossier [lib] sont à inclure dans le "ClassPath" du projet JBuilder.

Revenons sur le constructeur de la classe de test JUnit :

```
1.  /**
2.   * constructeur par défaut
```

```
3.    * <p>on récupère auprès de Spring une instance de l'interface d'accès aux données</p>
4.    */
5.   public JUnitTestsElectionsMetierDaoFile() {
6.     // classe parent
7.     super();
8.     // instanciation couche [metier]
9.     electionsMetier = (IElectionsMetier) (new XmlBeanFactory(new ClassPathResource(
10.         "spring-config-electionsMetierDaoFile.xml"))).getBean("electionsMetier");
11.}
```

- ligne 10 : le fichier de configuration Spring s'appelle [spring-config-electionsMetierDaoFile.xml]. Il doit être dans le "ClassPath" du projet. Nous le placerons dans le dossier [classes] :



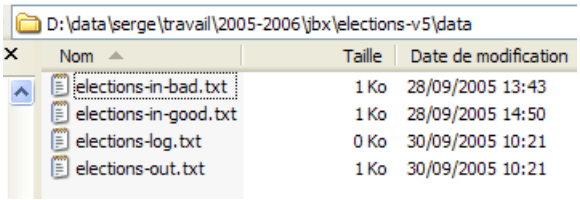
- lignes 9-10 : le champ privé [electionsMetier] est initialisé avec le bean Spring appelé "electionsMetier".

Le fichier de configuration Spring [spring-config-electionsMetierDaoFile.xml] aura le contenu 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 couche [dao] -->
5.          <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoFile">
6.                  <constructor-arg index="0">
7.                          <value>data\elections-in-good.txt</value>
8.                  </constructor-arg>
9.                  <constructor-arg index="1">
10.                         <value>data\elections-out.txt</value>
11.                 </constructor-arg>
12.                 <constructor-arg index="2">
13.                         <value>data\elections-log.txt</value>
14.                 </constructor-arg>
15.         </bean>
16.         <!-- la couche [metier] -->
17.         <bean id="electionsMetier" class="istia.st.elections.metier.ElectionsMetier">
18.                 <property name="electionsDao">
19.                         <ref local="electionsDao"/>
20.                 </property>
21.         </bean>
22. </beans>
```

- lignes 5-15 : définissent le bean [impotsDao] qui implémente la couche [dao]. Nous avons déjà présenté la définition de ce bean au paragraphe 3.3.2, page 23.
- lignes 17-21 : définissent le bean [impotsMetier] qui implémente la couche [metier].
- ligne 17 : la classe d'implémentation sera [istia.st.elections.metier.ElectionsMetier]. Cette classe sera instanciée ici grâce à son constructeur sans paramètres. L'objet instancié sera ensuite initialisé grâce aux méthodes **set** de la classe.
- ligne 18 : la balise <**property**> sert à initialiser un champ C de l'instance créée. Le nom du champ est précisé par l'attribut **name** de la balise. Spring utilise la méthode **set** associée à ce champ pour lui donner sa valeur. Il faut donc que cette méthode existe dans la classe.
- ligne 19 : la valeur affectée au champ [electionsDao] sera la référence du bean [electionsDao] défini lignes 5-15. La couche [metier] détient donc une référence à la couche [dao] ce qui lui permettra d'utiliser les services (= méthodes) offerts par cette couche.

Les trois fichiers texte référencés lignes 7, 10, 13 seront placés dans le dossier [data] du projet :



Le contenu des fichiers d'entrée sera le suivant :

elections-in-bad.txt	elections-in-good.txt
6x 0.05	6 0.05 A B C D E F G

Travail pratique : Mette en oeuvre sur machine les tests unitaires de la couche [metier]. Après exécution réussie des tests on consultera le fichier texte [elections-out.txt] et en cas d'échec le fichier [elections-log.txt].

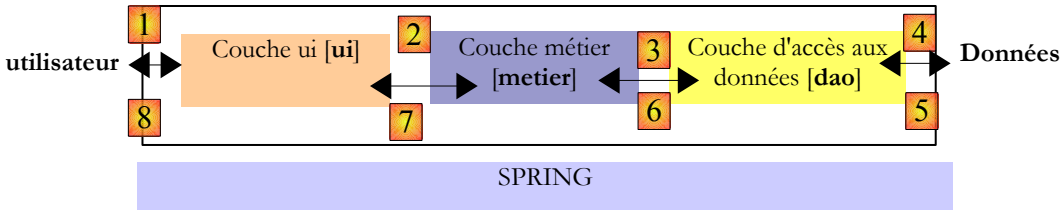
3.4.5 Création de l'archive [elections-metier.jar] de la couche [metier]

Travail pratique : Créer l'archive [elections-metier.jar] ayant le contenu suivant :

Nom	Type	Modifié	Taille	Ratio	En...	Chemin+
IElectionsMetier.class	Fichier CLASS	30/09/2005 11:31	2 446	00%	2 446	istia\st\elections\metier\
IElectionsMetier.class	Fichier CLASS	30/09/2005 11:31	405	00%	405	istia\st\elections\metier\
MANIFEST.MF	Fichier MF	30/09/2005 11:31	25	00%	25	META-INF\

3.4.6 Conclusion

Rappelons l'architecture générale de l'application [Elections] que nous sommes en train de construire :

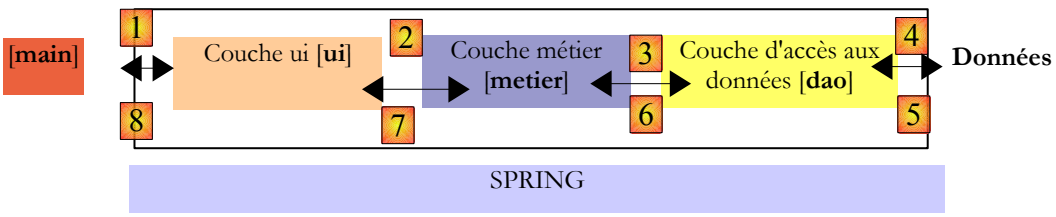


Nous avons construit les couche [metier] et [dao]. Nous allons terminer par la construction de la couche [ui].

3.5 La couche [ui]

3.5.1 L'interface [IElectionsUI]

Pour comprendre ce que peut être l'interface Java de la couche [ui], il nous faut savoir qui va utiliser cette interface. Il ne s'agit pas de l'utilisateur du schéma ci-dessus mais du programme qui va lancer l'application dans son ensemble.



L'interface [IElectionsUI] est présentée au programme principal [main] qui va lancer l'application. Que peut demander [main] à la couche [ui] ? Elle peut lui demander de commencer les échanges avec l'utilisateur qui va saisir les données manquantes de l'élection. On adoptera l'interface minimale suivante :

```
1.  package istia.st.elections.ui;
2.
3.  public interface IElectionsUI {
4.      /**
5.       * lance le dialogue avec l'utilisateur
6.       */
7.      public void run();
8.  }
```

- ligne 7 : l'interface n'a qu'une unique méthode : **run**. En appelant cette méthode, on demande à la couche [ui] de commencer les échanges avec l'utilisateur.

3.5.2 Le lanceur de l'application

Le lanceur de l'application est une classe Java ayant une méthode statique [main]. Cette méthode doit créer des instances des couches [ui, metier, demo] et demander à la couche [ui] de commencer le dialogue avec l'utilisateur. Cette classe pourrait être la suivante :

```
1.  package istia.st.elections.main;
2.
3.  import istia.st.elections.ui.IElectionsUI;
4.  import org.springframework.beans.factory.xml.XmlBeanFactory;
5.  import org.springframework.core.io.ClassPathResource;
6.
7.  public class MainElections {
8.      public static void main(String[] arguments) {
9.          // instanciation couche [ui]
10.         IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
11.                 ClassPathResource("spring-config-electionsUIMetierDaoFile.xml"))).
12.                                 getBean("electionsUI");
13.         // exécution
14.         electionsUI.run();
15.     }
16. }
```

- ligne 8 : la méthode statique [main] responsable du lancement de l'application [Elections].
- ligne 10 : la couche [ui] est instanciée. Nous allons voir que cette instanciation va provoquer celle des couches [metier] et [dao].
- ligne 14 : on demande à la couche [ui] de commencer le dialogue avec l'utilisateur.

Voyons maintenant le fichier de configuration Spring qui est utilisé pour instancier la couche [ui]. Son contenu est le 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 couche [dao] -->
5.          <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoFile">
6.                  <constructor-arg index="0">
7.                          <value>data\elections-in-good.txt</value>
8.                  </constructor-arg>
9.                  <constructor-arg index="1">
10.                         <value>data\elections-out.txt</value>
11.                 </constructor-arg>
12.                 <constructor-arg index="2">
13.                         <value>data\elections-log.txt</value>
14.                 </constructor-arg>
15.         </bean>
16.         <!-- la couche [metier] -->
17.         <bean id="electionsMetier" class="istia.st.elections.metier.ElectionsMetier">
18.                 <property name="electionsDao">
19.                         <ref local="electionsDao"/>
20.                 </property>
21.         </bean>
22.         <!-- la couche [ui] -->
23.         <bean id="electionsUI" class="istia.st.elections.ui.ElectionsConsole">
24.                 <property name="electionsMetier">
25.                         <ref local="electionsMetier"/>
26.                 </property>
27.         </bean>
28. </beans>
```

- lignes 5-15 : définissent l'instance de la couche [dao] - déjà commentées
- lignes 17-21 : définissent l'instance de la couche [metier] - déjà commentées
- lignes 23-27 : définissent l'instance de la couche [ui]
- ligne 23 : l'instance sera de type [istia.st.elections.ui.ElectionsConsole], une classe que nous allons bientôt définir. Cette classe implémentera l'interface [IElectionsUI] précédemment définie.

- lignes 24-27 : le champ [electionsMetier] de cette classe est initialisée avec le bean [electionsMetier] défini lignes 17-21

Revenons à l'instanciation de la couche [ui] dans la méthode statique [main] du programme principal de l'application :

```
1.      public static void main(String[] arguments) {
2.          // instanciation couche [ui]
3.          IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
4.                  ClassPathResource("spring-config-electionsUIMetierDaoFile.xml"))).
5.                                  getBean("electionsUI");
6.          // exécution
7.          electionsUI.run();
8.  }
```

- ligne 5, le bean [electionsUI] est demandé à Spring
- le bean [electionsUI] est défini lignes 23-26 du fichier de configuration
- Spring utilise le constructeur sans paramètres pour construire une instance de la classe [ElectionsConsole]. Puis, ligne 24, il utilise la méthode [setElectionsMetier] de cette classe pour initialiser le champ privé [electionsMetier]. Mais pour cette initialisation, il a besoin du bean [electionsMetier] (ligne 25). Spring va donc devoir instancier ce bean. Celui-ci est défini lignes 17-21. On voit que son instanciation va nécessiter celle du bean [electionsDao] (ligne 19). Spring va donc devoir instancier le bean [electionsDao]. Au final, on voit que pour instancier la couche [ui], Spring va devoir instancier les deux autres couches de l'application.
- lorsque ligne 7 du [main] ci-dessus, on demande à la couche [ui] de commencer le dialogue avec l'utilisateur, les trois couches [ui, metier, dao] sont en place et l'application est complète.

3.5.3 La classe d'implémentation [ElectionsConsole]

Notre première classe d'implémentation de la couche [ui] sera une classe utilisant la console pour communiquer avec l'utilisateur. Voici un exemple de dialogue obtenu sur la console JBuilder, avec la classe [MainElections] ci-dessus et les divers fichiers de configuration déjà rencontrés :

```
1.  30 sept. 2005 17:39:46 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
    loadBeanDefinitions
2.  INFO: Loading XML bean definitions from class path resource [spring-config-
    electionsUIMetierDaoFile.xml]
3.  30 sept. 2005 17:39:47 org.springframework.beans.factory.support.AbstractBeanFactory getBean
4.  INFO: Creating shared instance of singleton bean 'electionsUI'
5.  30 sept. 2005 17:39:47 org.springframework.core.CollectionFactory <clinit>
6.  INFO: JDK 1.4+ collections available
7.  30 sept. 2005 17:39:47 org.springframework.beans.factory.support.AbstractBeanFactory getBean
8.  INFO: Creating shared instance of singleton bean 'electionsMetier'
9.  30 sept. 2005 17:39:47 org.springframework.beans.factory.support.AbstractBeanFactory getBean
10. INFO: Creating shared instance of singleton bean 'electionsDao'
11.
12. Il y a 7 listes en compétition. Veuillez indiquer le nombre de voix de chacune d'elles :
13. Nombre de voix de la liste [A] : x
14. Nombre de voix incorrect. Veuillez recommencer
15. Nombre de voix de la liste [A] : -1
16. Nombre de voix incorrect. Veuillez recommencer
17. Nombre de voix de la liste [A] : 32000
18. Nombre de voix de la liste [B] : 25000
19. Nombre de voix de la liste [C] : 16000
20. Nombre de voix de la liste [D] : 12000
21. Nombre de voix de la liste [E] : 8000
22. Nombre de voix de la liste [F] : 4500
23. Nombre de voix de la liste [G] : 2500
24.
25. Résultats de l'élection
26.
27. [A,32000,2,false]
28. [B,25000,2,false]
29. [C,16000,1,false]
30. [D,12000,1,false]
31. [E,8000,0,false]
32. [F,4500,0,true]
33. [G,2500,0,true]
```

- lignes 1-10 : les informations loguées par Spring. On peut remarquer les points suivants :
  - ligne 4 : il commence la création du bean [electionsUI]. Pour cela il a besoin du bean [electionsMetier].
  - ligne 8 : il commence la création du bean [electionsMetier]. Pour cela il a besoin du bean [electionsDao].
  - ligne 10 : il commence la création du bean [electionsDao]
- ligne 12 : début du dialogue avec l'utilisateur

La classe [ElectionsConsole] pourrait avoir le squelette suivant :

```
1.  package istia.st.elections.ui;
```

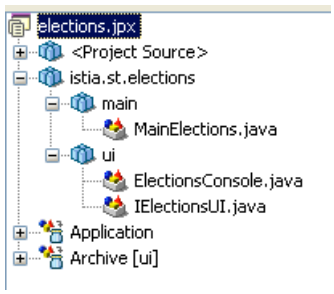
```
2.
3.  import istia.st.elections.metier.IElectionsMetier;
4.  import istia.st.elections.data.ListeElectorale;
5.  import istia.st.elections.data.ElectionsException;
6.  import java.io.IOException;
7.  import java.io.BufferedReader;
8.  import java.io.InputStreamReader;
9.
10. public class ElectionsConsole implements IElectionsUI {
11.     /**
12.      * instance de la couche [metier]
13.      */
14.     private IElectionsMetier electionsMetier;
15.
16.     /**
17.      *
18.      * @param electionsMetier IElectionsMetier : instance couche [metier]
19.      */
20.     public void setElectionsMetier(IElectionsMetier electionsMetier) {
21.         this.electionsMetier = electionsMetier;
22.     }
23.
24.     /**
25.      * constructeur par défaut
26.      */
27.     public ElectionsConsole() {
28.     }
29.
30.     /**
31.      * exécute le dialogue avec l'utilisateur
32.      */
33.     public void run() {
34. ...
35.             // on demande les listes en compétition à la couche [metier]
36.             ListeElectorale[] listes = electionsMetier.getListesElectorales();
37.             // on fait la saisie des voix
38. ....
39.             // on fait le calcul des sièges
40. ...
41.             // on enregistre les résultats
42. ...
43.             // on les affiche sur la console
44. ...
45.     }
46. }
```

- ligne 10 : la classe implémente l'interface [IElectionsUI]
- ligne 14 : référence sur la couche [metier] qui sera initialisée par Spring lors de la construction de la classe.
- lignes 20-22 : la méthode [set] correspondant au champ précédent
- lignes 27-28 : le constructeur sans paramètres qui sera utilisé par Spring pour instancier la classe. Pourrait ne pas être présent car lorsqu'une classe n'a aucun constructeur, c'est le constructeur sans paramètres qui est utilisé pour en construire des instances.
- lignes 33-45 : la méthode [run] de l'interface [IElectionsUI].

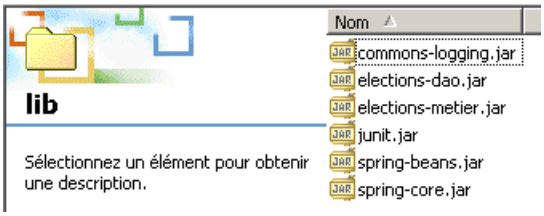
Question : Écrire le code de la méthode [run]. On s'aidera des commentaires.

3.5.4 Tests de l'application [Elections]

Le projet JBuilder de l'application [Elections] pourrait être le suivant :



Le dossier du projet aura un dossier [lib] contenant les archives de classes qui lui sont nécessaires :

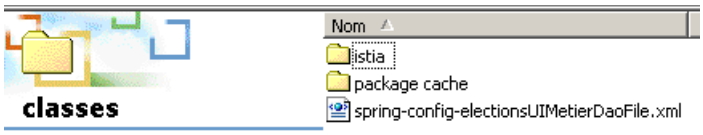


On remarquera la présence des archives de la couche [dao] (**elections-dao.jar**) et de la couche [metier] (**elections-metier.jar**). Les archives du dossier [lib] sont à inclure dans le "ClassPath" du projet JBuilder.

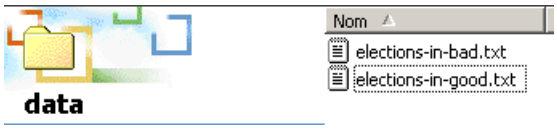
Revenons sur la méthode [main] du programme principal :

```
1.      public static void main(String[] arguments) {
2.          // instanciation couche [ui]
3.          IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
4.                  ClassPathResource("spring-config-electionsUIMetierDaoFile.xml"))).
5.                                  getBean("electionsUI");
6.          // exécution
7.          electionsUI.run();
8.  }
```

- ligne 4 : le fichier de configuration Spring s'appelle [spring-config-electionsUIMetierDaoFile.xml]. Il doit être dans le "ClassPath" du projet. Nous le placerons dans le dossier [classes] :



Les fichiers texte contenant les données caractérisant l'élection seront placés dans le dossier [data] du projet :



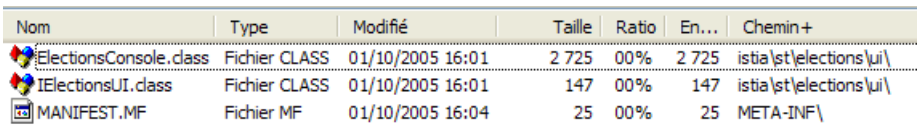
Le contenu des fichiers d'entrée sera le suivant :

elections-in-bad.txt	elections-in-good.txt
6x 0.05	6 0.05 A B C D E F G

Travail pratique : Mette en oeuvre sur machine l'application [Elections]. Après une exécution réussie on consultera le fichier texte [elections-out.txt] et en cas d'échec le fichier [elections-log.txt].

3.5.5 Création de l'archive [elections-ui.jar] de la couche [ui]

Travail pratique : Créer l'archive [elections-ui.jar] ayant le contenu suivant :



Nom	Type	Modifié	Taille	Ratio	En...	Chemin+
ElectionsConsole.class	Fichier CLASS	01/10/2005 16:01	2 725	00%	2 725	istia\st\elections\ui\
IElectionsUI.class	Fichier CLASS	01/10/2005 16:01	147	00%	147	istia\st\elections\ui\
MANIFEST.MF	Fichier MF	01/10/2005 16:04	25	00%	25	META-INF\

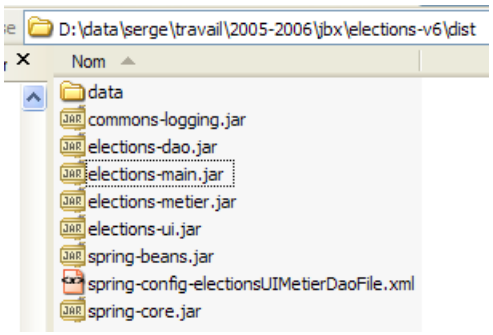
3.6 Création de l'archive [elections-main.jar] du programme principal

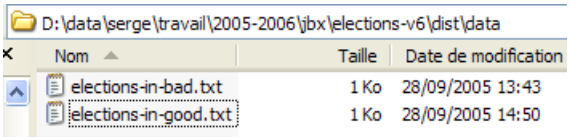
Travail pratique : Créer l'archive [elections-main.jar] ayant le contenu suivant :

Nom	Type	Modifié	Taille	Ratio	En...	Chemin+
MainElections.class	Fichier CLASS	01/10/2005 17:08	906	45%	500	istia\st\elections\main\
MANIFEST.MF	Fichier MF	01/10/2005 17:09	76	12%	67	META-INF\

3.7 Déploiement de l'application [Elections]

Cherchons maintenant à exécuter l'application [Elections] en-dehors de JBuilder. Rassemblons tous les fichiers nécessaires à l'application [Elections] dans le dossier [dist] suivant :





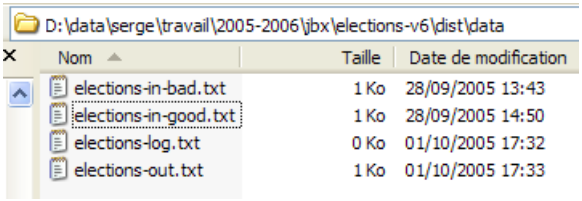
Ouvrons une fenêtre [DOS], plaçons-nous sur le dossier [dist] puis demandons l'exécution de la classe [istia.st.elections.main.MainElections] :

```
1.  dist>java -classpath elections-main.jar;elections-ui.jar;elections-metier.jar;elections-
    dao.jar;spring-core.jar;spring-beans.jar;commons-logging.jar;.
    istia.st.elections.main.MainElections
2.  1 oct. 2005 17:09:40 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
    loadBeanDefinitions
3.  INFO: Loading XML bean definitions from class path resource [spring-config-
    electionsUIMetierDaoFile.xml]
4.  1 oct. 2005 17:09:40 org.springframework.beans.factory.support.AbstractBeanFactory getBean
5.  INFO: Creating shared instance of singleton bean 'electionsUI'
6.  1 oct. 2005 17:09:40 org.springframework.core.CollectionFactory <clinit>
7.  INFO: JDK 1.4+ collections available
8.  1 oct. 2005 17:09:40 org.springframework.beans.factory.support.AbstractBeanFactory getBean
9.  INFO: Creating shared instance of singleton bean 'electionsMetier'
10. 1 oct. 2005 17:09:40 org.springframework.beans.factory.support.AbstractBeanFactory getBean
11. INFO: Creating shared instance of singleton bean 'electionsDao'
12.
13. Il y a 7 listes en compétition. Veuillez indiquer le nombre de voix de chacune d'elles :
14. Nombre de voix de la liste [A] : 32000
15. Nombre de voix de la liste [B] : 25000
16. Nombre de voix de la liste [C] : 16000
17. Nombre de voix de la liste [D] : 12000
18. Nombre de voix de la liste [E] : 8000
19. Nombre de voix de la liste [F] : 4500
20. Nombre de voix de la liste [G] : 2500
21.
22. Résultats de l'élection
23.
24. [A,32000,2,false]
25. [B,25000,2,false]
26. [C,16000,1,false]
27. [D,12000,1,false]
28. [E,8000,0,false]
29. [F,4500,0,true]
30. [G,2500,0,true]
```

- ligne 1 : on appelle la JVM [java]. Ici, cette JVM est dans le [PATH] de [Dos], ce qui explique qu'on n'a pas eu besoin de donner son nom complet. En réalité son nom complet était [C:\jewelbox\j2sdk1.4.2\bin\java.exe].
- ligne 1 : l'attribut [classpath] fixe les archives .jar et les dossier que la JVM doit explorer pour trouver les classes dont l'application a besoin. On cite ici tous les .jar nécessaires à l'application [Elections] et qui ont été placés dans le dossier [dist]. On ajoute de plus le dossier . qui représente le dossier courant au moment de l'exécution, c.a.d. le dossier [dist].

- ligne 1 : le dernier paramètre est le nom de la classe contenant la méthode statique [main] qui doit être exécutée au lancement de l'application. On donne son nom complet : [istia.st.elections.main.MainElections]
- lignes 2-11 : les logs de Spring.
- ligne 3 : le nom du fichier de configuration exploité. On se rappelle que ce fichier est cherché dans le [classpath] de l'application donc dans la liste des archives et dossiers listés derrière l'attribut [-classpath]. Parce qu'on a mis le dossier courant . dans cette liste, le fichier [spring-config-electionsUIMetierDaoFile.xml] est trouvé. Sinon, il ne l'aurait pas été. On se rappelle que les fichiers nécessaires à l'élection doivent se trouver dans un dossier [data] placé dans le répertoire de l'application. C'est pourquoi nous l'y avons mis.
- ligne 13 : début du dialogue avec l'utilisateur.

Après exécution, regardons le contenu du dossier [data] :



Le contenu de [elections-out.txt] est le suivant :

```
[A,32000,2,false]
[B,25000,2,false]
[C,16000,1,false]
[D,12000,1,false]
[E,8000,0,false]
[F,4500,0,true]
[G,2500,0,true]
```

4 Partie 4

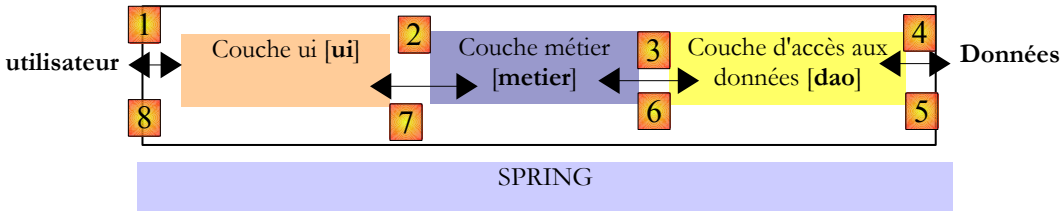
Mots clés : architecture 3tier, Spring, XML, Base de données, Interface graphique Swing

Lectures conseillées du polycopié Java :
- chapitre sur les bases de données
- chapitre sur les interfaces graphiques

4.1 Introduction

Rappelons ce qui a été fait :

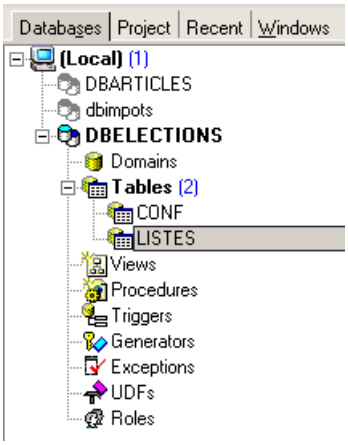
- dans la partie 1 de l'exercice ELECTIONS aucune classe n'a été utilisée. On a construit une solution comme on l'aurait construite en langage C.
- dans la partie 2 de l'exercice, deux classes ont été introduites :
  - [ListeElectorale] qui représente les attributs (id, nom, voix, sièges, élimine) d'une liste
  - [ElectionsException] une classe d'exceptions non contrôlées. Ce type d'exception est utilisé à chaque fois que se produit une erreur fatale dans l'application des élections. Elle est non contrôlée, c.a.d. que le développeur n'est pas obligé de la gérer avec un try-catch.
- dans la partie 3 de l'exercice, une solution 3tier a été élaborée :



Dans cette solution, les données nécessaires à l'élection avaient été placées dans des fichiers texte. Nous les plaçons désormais dans une base de données.

4.2 La base de données [dbelections]

Les données nécessaires à l'élection seront placées dans une base Firebird (http://firebird.sourceforge.net/). On pourra utiliser IBExpert (http://www.ibexpert.com) ou (EMS Interbase / Firebird Manager (http://sqlmanager.net/) pour créer la base. Ci-dessous, on trouvera des copies d'écran d'EMS Interbase / Firebird Manager. La base [**dbelections**] a deux tables : **CONF** et **LISTES** :



- la base [dbelections] et ses objets

SAP	SEUILELECTORAL
6	0,050

- contenu de la table [CONF]

ID	NOM	VOIX	SIEGES	ELIMINE
1	A	0	0	N
2	B	0	0	N
3	C	0	0	N
4	D	0	0	N
5	E	0	0	N
6	F	0	0	N
7	G	0	0	N

- contenu de la table [LISTES]

La table [CONF] a été construite avec les commandes SQL suivantes :

```
1.  /* Table: CONF */
2.
3.  CREATE TABLE CONF (
4.      SAP INTEGER,
5.      SEUILELECTORAL DOUBLE PRECISION);
6.
7.
8.
9.  /* Check constraints definition */
10.
11. ALTER TABLE CONF ADD CONSTRAINT CHK_CONF_SAP check (SAP>=1);
12. ALTER TABLE CONF ADD CONSTRAINT CHK_CONF_SEUILELECTORAL check (SEUILELECTORAL>=0);
```

- la table [CONF] a pour rôle de mémoriser les caractéristiques de l'élection. Elle n'a qu'une ligne.
- lignes 3-5 : la commande SQL CREATE (SAP : nombre de sièges à pourvoir, SEUILELECTORAL : le seuil électoral)
- ligne 11 : contrainte sur la colonne SAP
- ligne 12 : contrainte sur la colonne SEUILELECTORAL

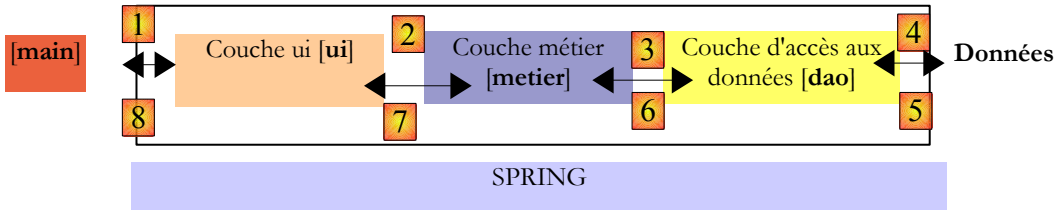
La table [LISTES] a été construite avec les commandes SQL suivantes :

```
1.  /* Table: LISTES */
2.
3.  CREATE TABLE LISTES (
4.      ID INTEGER NOT NULL,
5.      NOM VARCHAR (20) CHARACTER SET NONE NOT NULL COLLATE NONE,
6.      VOIX INTEGER DEFAULT 0 NOT NULL,
7.      SIEGES INTEGER DEFAULT 0 NOT NULL,
8.      ELIMINE CHAR (1) CHARACTER SET NONE DEFAULT 'N' NOT NULL COLLATE NONE);
9.
10.
11.
12. /* Check constraints definition */
13.
14. ALTER TABLE LISTES ADD CONSTRAINT CHK_LISTES_ID check (ID>=1);
15. ALTER TABLE LISTES ADD CONSTRAINT CHK_LISTES_NOM check (NOM<>'');
16. ALTER TABLE LISTES ADD CONSTRAINT CHK_LISTES_SIEGES check (SIEGES>=0);
17. ALTER TABLE LISTES ADD CONSTRAINT CHK_LISTES_VOIX check (VOIX>=0);
18. ALTER TABLE LISTES ADD CONSTRAINT CHK_LISTES_ELIMINE check (UPPER(ELIMINE)='O' OR
    UPPER(ELIMINE)='N');
19.
20.
21. /* Primary keys definition */
22.
23. ALTER TABLE LISTES ADD CONSTRAINT PK_LISTES PRIMARY KEY (ID);
```

- la table [LISTES] a pour rôle de mémoriser les informations sur les listes en compétition dans l'élection.
- lignes 3-8 : la définition des colonnes de la table, ID : identifiant liste, NOM : nom de la liste, VOIX : voix de la liste, SIEGES : nombre de sièges obtenus par la liste, ELIMINE : 'O' si liste éliminée, 'N' si liste non éliminée.
- lignes 14-18 : les contraintes sur les colonnes de la table
- ligne 23 : la clé primaire de la table

4.3 Implémentation de la couche [dao] avec une base de données

Revenons sur l'architecture de l'application [Elections] :



Nous construisons une couche [dao] capable d'exploiter la base de données présentée précédemment.

4.3.1 L'interface [IElectionsDao]

L'interface de la couche [dao] n'a pas changé. C'est toujours l'interface [IElectionsDao] suivante :

```
1. package istia.st.elections.dao;
2.
3. import istia.st.elections.data.ListeElectorale;
4.
5. public interface IElectionsDao {
6.   /**
7.    * demande le seuil électoral
8.    * @return double : le seuil électoral
9.    */
10.  public double getSeuilElectoral();
11.
12.  /**
13.   * demande le nombre de sièges à pourvoir
14.   * @return int : le nombre de sièges à pourvoir
15.   */
16.  public int getNbSiegesAPourvoir();
17.
18.  /**
19.   * demande le tableau des listes en compétition
20.   * @return ListeElectorale[] : les listes en compétition
21.   */
22.  public ListeElectorale[] getListesElectorales();
23.
24.  /**
25.   * enregistre les résultats des listes en compétition
26.   * @param listesElectorales ListeElectorale[] : les listes en compétition
27.   */
28.  public void setListesElectorales(ListeElectorale[] listesElectorales);
29.}
```

4.3.2 Les tests unitaires

La classe de tests unitaires de la couche [dao] change très peu. Il s'agit toujours de tester l'interface [IElectionsDao] mais cette fois-ci avec une autre implémentation de cette interface. Avec Spring, nous savons que le choix de la classe d'implémentation se fait par configuration. La classe de test s'appellera [JUnitTestsElectionsDaoJdbc] et son code sera le suivant :

```
1.  package istia.st.elections.dao.tests;
2.
3.  import org.springframework.beans.factory.xml.XmlBeanFactory;
4.  import org.springframework.core.io.ClassPathResource;
5.  import istia.st.elections.dao.IElectionsDao;
6.  import junit.framework.TestCase;
7.  import istia.st.elections.data.ListeElectorale;
8.
9.  /**
10.  *
11.  * <p>Title: JUnitTestsElectionsDaoJdbc</p>
12.  * <p>Description: Classe de test de la couche [dao] de l'application [Elections]</p>
13.  * <p>Copyright: Copyright (c) 2005</p>
14.  * <p>Company: ISTIA</p>
15.  * @author ST
16.  * @version 1.0
17.  */
18. public class JUnitTestsElectionsDaoJdbc extends TestCase {
```

```
19.
20.     /**
21.      * instance d'accès à la couche [dao]
22.      */
23.     private IElectionsDao electionsDao = null;
24.
25.     /**
26.      * constructeur par défaut
27.      */
28.     public JUnitTestsElectionsDaoJdbc() {
29.         // construction parent
30.         super();
31.         // instanciation couche [dao]
32.         electionsDao = (IElectionsDao) (new XmlBeanFactory(new
33.                 ClassPathResource(
34.                         "spring-config-ElectionsDaoJdbc.xml"))).getBean(
35.                                 "electionsDao");
36.     }
37.
38.     /**
39.      * affichage des données de l'élection
40.      */
41.     public void testLectureDataElections() {
42.         // on affiche les données de l'élection
43.         System.out.println("Nombre de sièges à pourvoir : " +
44.                             electionsDao.getNbSiegesAPourvoir());
45.         System.out.println("Seuil électoral : " +
46.                             electionsDao.getSeuilElectoral());
47.         ListeElectorale[] listes = electionsDao.getListesElectorales();
48.         for (int i = 0; i < listes.length; i++) {
49.             System.out.println(listes[i]);
50.         }
51.     }
52.
53.     public void testEcritureResultatsElections() {
54.         // on crée un tableau de listes
55.         ListeElectorale[] listesElectorales = new ListeElectorale[7];
56.         listesElectorales[0] = new ListeElectorale(1, "A", 32000, 2, false);
57.         listesElectorales[1] = new ListeElectorale(2, "B", 25000, 2, false);
58.         listesElectorales[2] = new ListeElectorale(3, "C", 16000, 1, false);
59.         listesElectorales[3] = new ListeElectorale(4, "D", 12000, 1, false);
60.         listesElectorales[4] = new ListeElectorale(5, "E", 8000, 0, false);
61.         listesElectorales[5] = new ListeElectorale(6, "F", 4500, 0, true);
62.         listesElectorales[6] = new ListeElectorale(7, "G", 2500, 0, true);
63.         // on rend ces données persistantes
64.         electionsDao.setListesElectorales(listesElectorales);
65.     }
66. }
```

- la seule modification notable par rapport au test unitaire utilisé avec l'implémentation précédente de la couche [dao] est le nom du fichier de configuration, ligne 34.

4.3.3 La classe [ElectionsDaoJDBC]

Le squelette de la classe [ElectionsDaoJDBC] implémentant la couche [dao] avec une base de données sera le suivant :

```
1.  package istia.st.elections.dao;
2.
3.  //paquetages importés
4.  import java.sql.*;
5.  import java.util.*;
6.  import istia.st.elections.data.ListeElectorale;
7.  import istia.st.elections.data.ElectionsException;
8.
9.  public class ElectionsDaoJDBC implements IElectionsDao {
10.
11.     /**
12.      * pilote de la BD
13.      */
14.     private String driverClassName;
15.
16.     /**
17.      * url de la BD
18.      */
19.     private String url;
20.
21.     /**
22.      * login utilisateur
23.      */
24.     private String userName;
25.
26.     /**
27.      * mot de passe de l'utilisateur de la BD
```

```
28.      */
29.     private String password;
30.
31.
32.     /**
33.      * le seuil électoral
34.      */
35.     double seuilElectoral;
36.     /**
37.      * obtient le seuil électoral
38.      * @return double : le seuil électoral
39.      */
40.     public double getSeuilElectoral() {
41.         return seuilElectoral;
42.     }
43.
44.
45.     /**
46.      * le nombre de sièges à pourvoir
47.      */
48.     int nbSiegesAPourvoir;
49.     /**
50.      * obtient le nombre de sièges à pourvoir
51.      * @return int : le nombre de sièges à pourvoir
52.      */
53.     public int getNbSiegesAPourvoir() {
54.         return nbSiegesAPourvoir;
55.     }
56.
57.
58.     /**
59.      * les listes en compétition
60.      */
61.     ListeElectorale[] listesElectorales = null;
62.     /**
63.      * obtient le tableau des listes en compétition
64.      * @return ListeElectorale[] : le tableau des listes en compétition
65.      */
66.     public ListeElectorale[] getListesElectorales() {
67.         return listesElectorales;
68.     }
69.
70.     /**
71.      * constructeur sans paramètres
72.      *
73.      */
74.     public ElectionsDaoJDBC() {
75.
76.     }
77.
78.     /**
79.      *
80.      * @param driverClassName : le pilote JDBC du SGBD
81.      * @param url : url de la BD
82.      * @param userName : le login de l'utilisateur
83.      * @param password : son mot de passe
84.      */
85.
86.     public ElectionsDaoJDBC(String driverClassName, String url,
87.                             String userName, String password) {
88.         // on mémorise les paramètres
89. ...
90.
91.         // exploitation de la base de données
92.         try {
93.             // connexion à la base
94. ...
95.             // création d'un objet PreparedStatement
96. ...
97.             // exécution de la requête SELECT sur la table CONF
98. ...
99.             // on exploite l'unique ligne de résultats
100....
101.            // création d'un nouvel objet PreparedStatement
102....
103.            // exécution et exploitation de la requête select sur la table LISTES
104....
105.            // fermeture ressources
106....
107.        } catch (Exception ex1) {
108....
109.        } finally {
110....
111.        }
112....
113.    }
114.
```

```
115.    /**
116.     * met à jour les champs (sieges, voix, elimine) des listes
117.     * @param listesElectorales ListeElectorale[] : le tableau des listes après calcul des sièges
118.     */
119.    public void setListesElectorales(ListeElectorale[] listesElectorales) {
120.        // exploitation de la base
121...
122.        try {
123.            // connexion à la base
124....
125.            // création d'un objet PreparedStatement pour la requête UPDATE
126...
127.            // on boucle sur le tableau des listes
128....
129.            // on libère les ressources
130....
131.        } catch (Exception ex1) {
132....
133.        } finally {
134....
135.        }
136.    }
137.
138.}
```

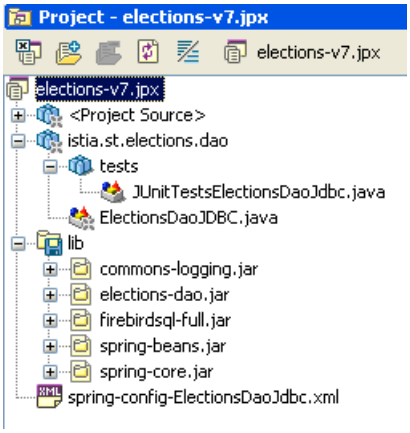
- la classe [ElectionsDaoJdbc] implémente l'interface [IElectionsDao], ligne 9. On retrouve donc les méthodes à implémenter :
    - [getSeuilElectoral] - ligne 40
    - [getNbSiegesAPourvoir] - ligne 53
    - [getListesElectorales] - ligne 66
    - [setListesElectorales] - ligne 119
- pour exploiter la base de données, la couche [dao] doit créer une connexion avec elle. En Java, il s'agit de créer un objet [Connection]. Cette création nécessite quatre arguments, tous de type String :
    - le nom du pilote JDBC qui va faire l'intermédiaire entre la couche [dao] et le SGBD
    - l'identifiant de la base à exploiter. En effet, un SGBD gère plusieurs bases de données. Une connexion ne vaut que pour une base de données. Un identifiant, appelé également URL de la base, doit donc être fourni.
    - l'utilisateur à qui va appartenir la connexion. Un SGBD gère des utilisateurs. Ceux-ci ont des droits (select, insert, update, delete, ...) sur certaines tables de certaines bases de données. On ouvre une connexion pour un utilisateur donné. Il faut préciser le nom de celui-ci (appelé parfois login) ainsi que son mot de passe.
- les quatre arguments nécessaires à l'ouverture d'une connexion au SGBD seront fournis au constructeur de la classe [ElectionsDaoJdbc], lignes 86-87. Ces informations seront mémorisées dans quatre champs privés : **driverClassName** - ligne 14, **url** - ligne 19, **userName** - ligne 24, **password** - ligne 29. Ils pourront ainsi être utilisés également par la méthode **setListesElectorales**, ligne 119.
- les arguments nécessaires à la construction d'un objet [ElectionsDaoJDBC] seront fournis par un fichier de configuration Spring.

Question 1 : Écrire le constructeur à quatre arguments de la classe. Il a pour fonction première de donner une valeur aux champs privés [seuilElectoral, nbSiegesAPourvoir, listesElectorales]. Si un problème surgit, on lancera une exception de type [ElectionsException]. Le constructeur mémorisera également les valeurs de ses quatre paramètres dans les champs privés correspondants.

Question 2 : Écrire la méthode [setListesElectorales]. Si un problème apparaît, on lancera une exception de type [ElectionsException].

4.3.4 Configuration des tests de la couche [dao]

Le projet JBuilder des tests de la classe [ElectionsDaoJDBC] aura la structure suivante :



- le dossier [lib] contiendra les archives nécessaires aux tests :
  - [elections-dao.jar] : pour l'interface [IElectionsDao] et le paquetage [istia.st.elections.data]
  - [spring-core.jar, spring-beans.jar, commons-logging.jar] : pour Spring
  - [firebirdsql-full.jar] : pour le pilote JDBC du SGBD Firebird
- le fichier de configuration Spring sera le 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 données -->
5.    <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoJDBC">
6.      <constructor-arg index="0">
7.        <value>org.firebirdsql.jdbc.FBDriver</value>
8.      </constructor-arg>
9.      <constructor-arg index="1">
10.       <value>jdbc:firebirdsql:localhost/3050:d:/data/serge/databases/firebird/dbelections.gd
    b</value>
11.     </constructor-arg>
12.     <constructor-arg index="2">
13.       <value>SYSDBA</value>
14.     </constructor-arg>
15.     <constructor-arg index="3">
16.       <value>masterkey</value>
17.     </constructor-arg>
18.   </bean>
19. </beans>
```

  - ligne 5 : l'implémentation de la couche [dao] est assurée par une instance de la classe [istia.st.elections.dao.ElectionsDaoJDBC]
  - lignes 5-18 : la classe [ElectionsDaoJDBC] est instanciée grâce à son constructeur à quatre paramètres
  - ligne 7 : le 1er paramètre est le nom du pilote JDBC à utiliser. Ici c'est un pilote pour le SGBD Firebird
  - ligne 10 : l'url de la base à utiliser. La syntaxe de cette url est la suivante :
    - jdbc:firebirdsql : ces deux termes sont constants pour toute base Firebird
    - localhost : machine sur laquelle se trouve le SGBD. Ici la machine locale sur laquelle se trouve le programme Java exécuté.
    - 3050 : le port d'écoute standard du SGBD. C'est là qu'il attend les demandes de ses clients.
    - d:/data/... : le nom du fichier Firebird contenant la base de données.
  - ligne 13 : utilisateur sous l'identité de laquelle sera ouverte la connexion avec le SGBD. SYSDBA est l'administrateur du SGBD Firebird. Il a tous les droits sur toutes les bases.
  - ligne 16 : le mot de passe de l'utilisateur SYSDBA

Pour comprendre cette configuration de test, il faut se rappeler la teneur du programme de tests unitaires qui est exécuté :

```
1.  package istia.st.elections.dao.tests;
2.
3.  ...
4.  public class JUnitTestsElectionsDaoJdbc extends TestCase {
5.
6.      /**
7.       * instance d'accès à la couche [dao]
8.       */
9.      private IElectionsDao electionsDao = null;
10.
11.     /**
12.      * constructeur par défaut
13.      */
14.     public JUnitTestsElectionsDaoJdbc() {
15.         // construction parent
16.         super();
```

```
17.         // instanciation couche [dao]
18.         electionsDao = (IElectionsDao) (new XmlBeanFactory(new
19.                 ClassPathResource(
20.                         "spring-config-ElectionsDaoJdbc.xml"))).getBean(
21.                                 "electionsDao");
22.     }
23.
24.     /**
25.      * affichage des données de l'élection
26.      */
27.     public void testLectureDataElections() {
28. ...
29.     }
30.
31.     public void testEcritureResultatsElections() {
32. ...
33.     }
34. }
```

- lignes 18-21 : la couche [dao] est instanciée avec le bean nommé "electionsDao" (ligne 21). Ce bean est défini lignes 5-18 du fichier de configuration Spring [spring-config-ElectionsDaoJdbc.xml] exploité. Spring va instancier ce bean et en rendre une référence qui sera stockée par le constructeur de la classe de test dans le champ privé [electionsDao] (ligne 18).
- les tests des lignes 27 et 31 vont utiliser la référence [electionsDao] pour tester la couche [dao].

Travail pratique : Mettre en oeuvre ces tests unitaires. Ne pas oublier de lancer le SGBD Firebird.

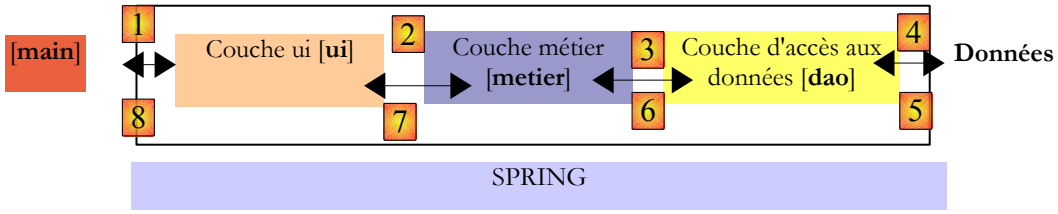
4.3.5 Création de l'archive [elections-dao-jdbc.jar]

Travail pratique : Créer l'archive [elections-dao.jdbc.jar] suivante

Nom	Type	Modifié	Taille	Ratio	En...	Chemin+
ElectionsDaoJDBC.class	Fichier CLASS	04/10/2005 08:56	5 716	51%	2 795	istia\st\elections\dao\

4.4 Implémentation de la couche [ui] avec une interface Swing

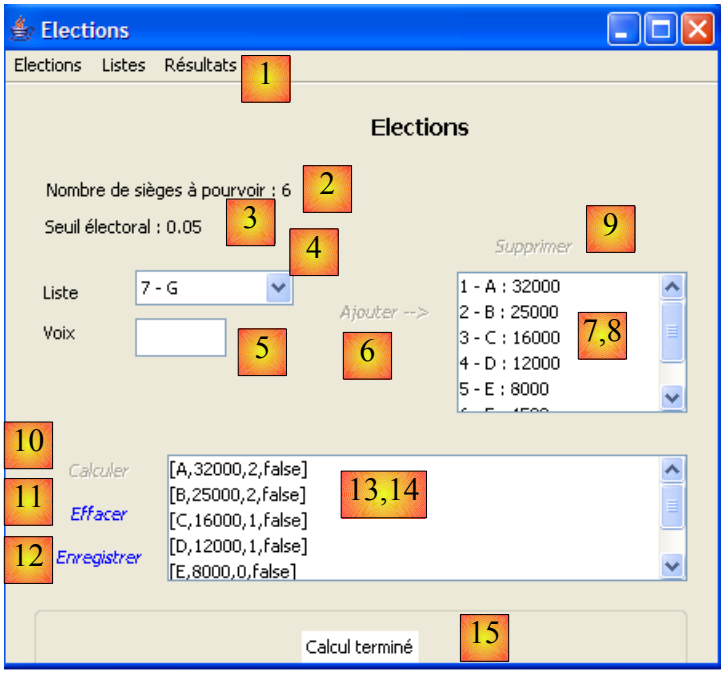
Revenons à l'architecture globale de l'application [Elections] :



Nous nous intéressons maintenant à une nouvelle implémentation de la couche [ui]. L'unique implémentation réalisée pour le moment est une interface console. Nous créons maintenant une interface graphique.

4.4.1 Description de l'interface swing

L'utilisateur disposera de l'interface suivante pour interagir avec l'application [Elections] :

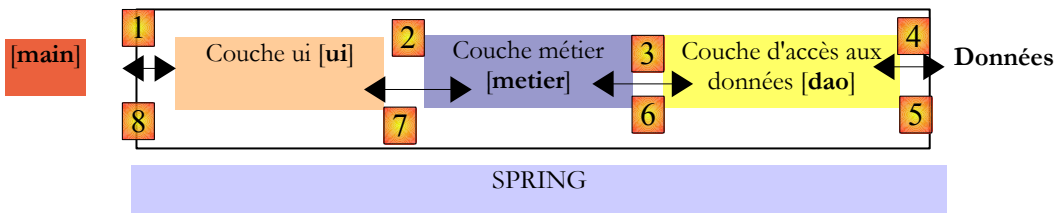


Les composants de l'interface sont les suivants :

n°	type	nom	rôle
1	JMenuBar	jMenuBar1	un menu
2	JLabel	jLabelSAP	le nombre de sièges à pourvoir
3	JLabel	jLabelSE	le seuil électoral
4	JComboBox	jComboBoxNomsListes	liste des noms des listes en compétition
5	JTextField	jTextFieldVoixListe	le nombre de voix d'une liste
6	JLabel	jLabelAjouter	pour ajouter une liste à (8)
7,8	(JScrollPane, JList)	jListNomsVoix	les noms et voix des listes
9	JLabel	jLabelSupprimer	pour supprimer de (8) la liste sélectionnée dans (8)
10	JLabel	jLabelCalculer	pour calculer les résultats de l'élection
11	JLabel	jLabelEffacer	pour effacer les résultats de l'élection
12	JLabel	jLabelEnregistrer	pour enregistrer les résultats de l'élection
13,14	(JScrollPane, JList)	jListResultats	pour afficher les résultats de l'élection
15	JTextPane	jTextPaneMessages	pour afficher des messages de suivi

4.4.2 Fonctionnement de l'application

Revenons sur l'architecture générale de l'application :



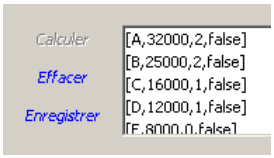
L'interface graphique se trouve dans la couche [ui]. C'est elle qui interagit avec l'utilisateur.

- au démarrage, l'application console [**main**] instancie les trois couches de l'application grâce à Spring. Ceci est fait avant même que l'interface graphique ne soit visible. Toujours dans cette phase d'initialisation, les renseignements caractérisant l'élection (nombre de sièges à pourvoir, seuil électoral, listes en compétition) sont demandés à la couche [dao]. Si cette phase d'initialisation échoue (impossibilité d'accéder aux données par exemple), un message d'erreur est affiché sur la console et l'interface graphique n'est pas affichée.

- si la lecture des données s'est bien passée, l'interface graphique est affichée avec les renseignements suivants (cf copie d'écran plus haut) :
  - le nombre de sièges à pourvoir dans (2)
  - le seuil électoral dans (3)
  - les identifiants et noms des listes candidates dans (4)

- l'utilisateur affecte alors à chaque liste candidate son nombre de voix à l'aide des champs 4 (id - nom), 5 (voix), 6 (pour ajouter).

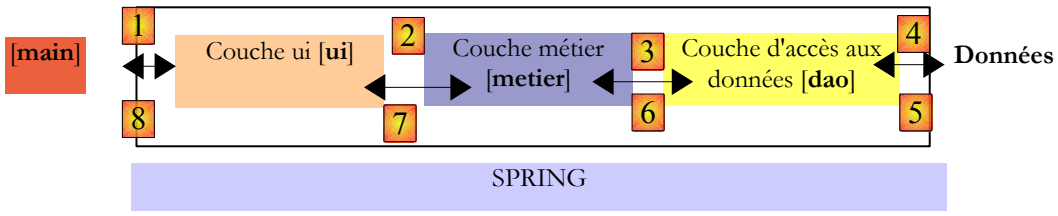
- on peut alors utiliser le lien (10) pour calculer les sièges :



- le lien [Enregistrer] permet d'enregistrer les résultats dans la source de données.

4.4.3 La classe [ElectionsSwing] d'implémentation de la couche [ui]

Revenons sur la structure de l'application [Elections] lorsqu'on l'avait implémentée avec une interface console. Elle présentait l'architecture 3tier suivante :



La couche [ui] présentait l'interface [IElectionsUI] suivante :

```
1.  package istia.st.elections.ui;
2.
3.  public interface IElectionsUI {
4.      /**
5.       * lance le dialogue avec l'utilisateur
6.       */
7.      public void run();
8.  }
```

L'application était lancée par la classe [MainElections] suivante :

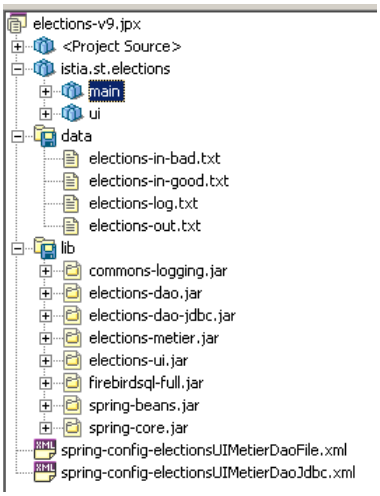
```
1.  package istia.st.elections.main;
2.
3.  import istia.st.elections.ui.IElectionsUI;
4.  import org.springframework.beans.factory.xml.XmlBeanFactory;
5.  import org.springframework.core.io.ClassPathResource;
6.
7.  public class MainElections {
8.      public static void main(String[] arguments) {
```

```
9.          // instanciation couche [ui]
10.         IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
11.                 ClassPathResource("spring-config-electionsUIMetierDaoFile.xml"))).
12.                                 getBean("electionsUI");
13.         // exécution
14.         electionsUI.run();
15.     }
16. }
```

Le fichier de configuration Spring utilisé par le lanceur était le 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 couche [dao] -->
5.    <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoFile">
6.      <constructor-arg index="0">
7.        <value>data\elections-in-good.txt</value>
8.      </constructor-arg>
9.      <constructor-arg index="1">
10.       <value>data\elections-out.txt</value>
11.     </constructor-arg>
12.     <constructor-arg index="2">
13.       <value>data\elections-log.txt</value>
14.     </constructor-arg>
15.   </bean>
16.   <!-- la couche [metier] -->
17.   <bean id="electionsMetier" class="istia.st.elections.metier.ElectionsMetier">
18.     <property name="electionsDao">
19.       <ref local="electionsDao"/>
20.     </property>
21.   </bean>
22.   <!-- la couche [ui] -->
23.   <bean id="electionsUI" class="istia.st.elections.ui.ElectionsConsole">
24.     <property name="electionsMetier">
25.       <ref local="electionsMetier"/>
26.     </property>
27.   </bean>
28. </beans>
```

Par souci de cohérence avec ce qui a été fait, il est souhaitable que la nouvelle couche [ui] offre également l'interface [IElectionsUI]. Ainsi on pourra basculer d'une interface console à une interface graphique par simple changement du fichier de configuration Spring. Le projet Jbuilder de l'application aura la forme suivante :



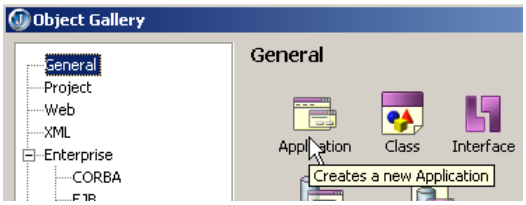
Nous testerons la couche [ui] avec deux configurations de la couche [dao] : celle implémentée à l'aide de fichiers texte et celle implémentée avec une base de données Firebird.

- [lib] est le dossier contenant les archives nécessaires à l'application :
  - [elections-dao.jar, elections-dao-jdbc] : pour la couche [dao]
  - [elections-metier.jar] : pour la couche [metier]
  - [elections-ui.jar] : pour l'interface [IElectionsUi] qui sera implémentée par la classe [ElectionsSwing.java]
  - [spring-core.jar, spring-beans.jar, commons-logging.jar] : pour Spring
  - [firebirdsql-full.jar] : pour le pilote JDBC du SGBD Firebird
- toutes les archives de [lib] sont placées dans le " ClassPath " du projet.
- [spring-config-electionsUIMetierDaoFile.xml] : le fichier de configuration Spring utilisé pour instancier les couches [ui, metier, dao] avec une couche [dao] implémentée avec des fichiers texte.

- [spring-config-electionsUIMetierDaoJdbc.xml] : le fichier de configuration Spring utilisé par [MainElectionsDaoJdbc] pour instancier les couches [ui, metier, dao] avec une couche [dao] implémentée avec une base de données.

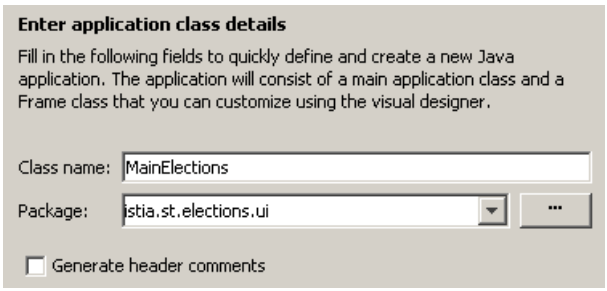
Lorsque nous utilisons Jbuilder pour construire une interface graphique, nous utilisons habituellement un assistant qui donne naissance à deux fichiers. Suivons le processus de cet assistant :

- choisissons l'option File/New/Application :

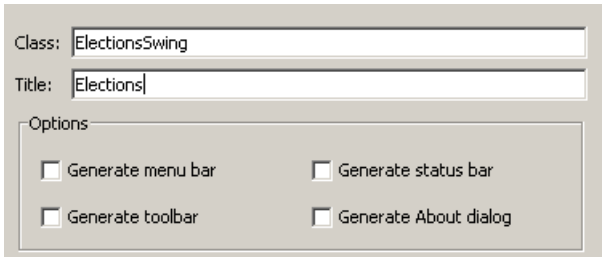


- l'assistant va générer deux classes :
  - une classe qui contiendra la méthode statique [main] qui lancera l'application
  - la classe de l'interface graphique dérivée de [JFrame]

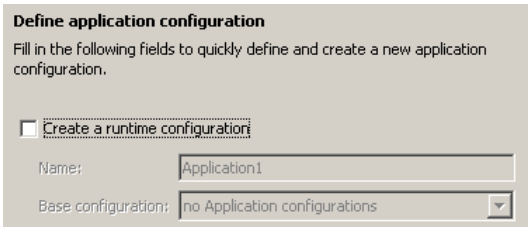
  L'assistant nous demande de caractériser la classe de lancement de l'application :



- l'étape suivante nous demande de caractériser la classe de l'interface graphique :



- la dernière étape nous demande de définir une configuration d'exécution. Nous n'en définissons pas :



Une fois l'assistant terminé, deux classes Java sont générées :

La classe [MainElections.java] ressemble à ce qui suit :

```
1.  package istia.st.elections.ui;
2.
3.  ...
4.
5.  public class MainElections {
6.      boolean packFrame = false;
7.
```

```
8.      /**
9.       * Construct and show the application.
10.      */
11.     public MainElections() {
12.         ElectionsSwing frame = new ElectionsSwing();
13.         frame.setVisible(true);
14.     }
15.
16.     /**
17.      * Application entry point.
18.      *
19.      * @param args String[]
20.      */
21.     public static void main(String[] args) {
22.         SwingUtilities.invokeLater(new Runnable() {
23.             public void run() {
24.                 try {
25.                     UIManager.setLookAndFeel(UIManager.
26.                                              getSystemLookAndFeelClassName());
27.                 } catch (Exception exception) {
28.                     exception.printStackTrace();
29.                 }
30.
31.                 new MainElections();
32.             }
33.         });
34.     }
35. }
```

- ligne 25, on demande à la classe [UIManager] qui gère l'aspect des interfaces graphiques de donner à la nôtre celui des interfaces graphiques du système d'exploitation sous-jacent. Dans les copies d'écran ci-dessus, ce système était Windows 2000.
- ligne 31 : demande la construction d'un objet de type [MainElections]
- ligne 12 : un objet [ElectionsSwing] est construit. Rappelons que [ElectionsSwing] est la classe de l'interface graphique
- ligne 13 : l'interface graphique est rendue visible

Au final, la méthode [main] exécutée au démarrage de l'application :
- construit un objet [ElectionsSwing]
- lui demande de s'afficher

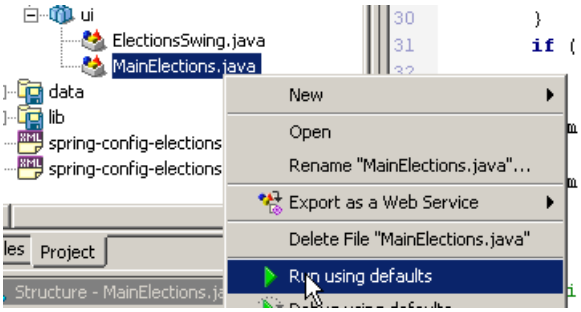
La classe [ElectionsSwing] ressemble elle à ce qui suit :

```
1.  package istia.st.elections.ui;
2.
3.  ...
4.  public class ElectionsSwing extends JFrame {
5.      JPanel contentPane;
6.      BorderLayout borderLayout1 = new BorderLayout();
7.
8.      public ElectionsSwing() {
9.          try {
10.             setDefaultCloseOperation(EXIT_ON_CLOSE);
11.             jbInit();
12.         } catch (Exception exception) {
13.             exception.printStackTrace();
14.         }
15.     }
16.
17.     /**
18.      * Component initialization.
19.      *
20.      * @throws java.lang.Exception
21.      */
22.     private void jbInit() throws Exception {
23.         contentPane = (JPanel) getContentPane();
24.         contentPane.setLayout(borderLayout1);
25.         setSize(new Dimension(400, 300));
26.         setTitle("Elections");
27.     }
28. }
```

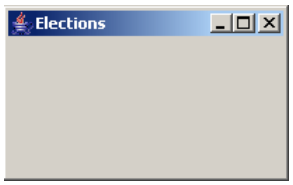
- rappelons-nous que la méthode [main] fait appel au constructeur de la classe [ElectionsSwing]. Celui-ci est défini lignes 8-15 ci-dessus.
- ligne 4 : la classe [ElectionsSwing] dérive de la classe [JFrame]
- ligne 10 : indique que lorsque l'utilisateur ferme la fenêtre, l'application doit s'arrêter.
- ligne 11 : appel à la méthode [jbInit] qui va créer la fenêtre
- ligne 23 : [contentPane] est une référence sur le conteneur de composants de la fenêtre
- ligne 24 : fixe la méthode de disposition des composants dans ce conteneur
- ligne 25 : fixe les dimensions de la fenêtre

- ligne 26 : fixe le titre de la fenêtre

On a là une application exécutable. Tentons une première exécution :



On obtient la fenêtre suivante :



Si on ferme la fenêtre, l'application s'arrête.

La classe [ElectionsSwing] générée par JBuilder n'implémente pas l'interface [IElectionsUI] que nous avons définie. Rappelons la définition de cette dernière :

```
1.  package istia.st.elections.ui;
2.
3.  public interface IElectionsUI {
4.      /**
5.       * lance le dialogue avec l'utilisateur
6.       */
7.      public void run();
8.  }
```

La classe [ElectionsSwing] doit donc implémenter la méthode [run]. Faisons évoluer la code de cette classe de la façon suivante :

```
1.  package istia.st.elections.ui;
2.
3.  ...
4.  public class ElectionsSwing extends JFrame implements IElectionsUI {
5.      JPanel contentPane;
6.      BorderLayout borderLayout1 = new BorderLayout();
7.
8.      public ElectionsSwing() {
9.     ...
10. }
11.     /**
12.      * Component initialization.
13.      *
14.      * @throws java.lang.Exception
15.      */
16.     private void jbInit() throws Exception {
17. ...
18.     }
19.
20.     /**
21.      * affichage interface graphqiue
22.      */
23.     public void run() {
24.         // on s'affiche
25.         this.setVisible(true);
26.     }
27. }
```

- ligne 4 : la classe [ElectionsSwing] implémente désormais l'interface [IElectionsUI]
- lignes 23-26 : la méthode [run] est implémentée. Elle ne fait qu'une chose : rendre visible la fenêtre rendant possible le début du dialogue avec l'utilisateur.

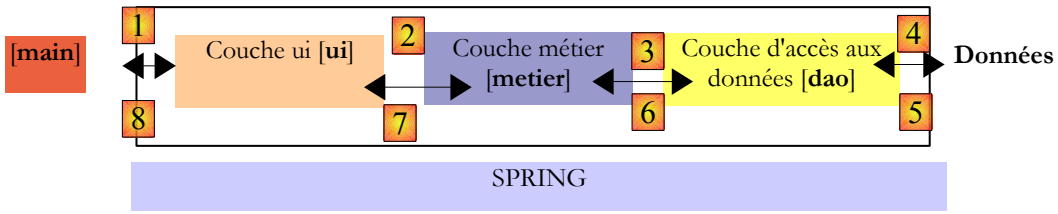
Avec cette nouvelle écriture de la classe [ElectionsSwing], la classe [MainElections] devient la suivante :

```
1.  package istia.st.elections.ui;
2.
3.  ...
4.
5.  public class MainElections {
6.      boolean packFrame = false;
7.
8.      /**
9.       * Construct and show the application.
10.      */
11.     public MainElections() {
12.         ElectionsSwing frame = new ElectionsSwing();
13.         //frame.setVisible(true);
14.         frame.run();
15.     }
16.
17.     /**
18.      * Application entry point.
19.      *
20.      * @param args String[]
21.      */
22.     public static void main(String[] args) {
23. ...
24.     }
25. }
```

- la modification intervient lignes 13-14. Au lieu d'afficher la fenêtre [frame] par la méthode [setVisible], on utilise la nouvelle méthode [run].

Si on exécute la nouvelle classe [MainElections], on obtient bien l'affichage de la fenêtre :

Nous devons aller un peu plus loin dans la modification de la classe [ElectionsSwing] générée par JBuilder. Revenons sur la structure générale de l'application :



On voit que la couche [ui] doit communiquer avec la couche [metier]. La classe [ElectionsSwing] implémente la couche [ui]. Il faut que cette classe ait une référence sur la couche [metier] pour pouvoir communiquer avec elle. Nous faisons évoluer la structure de la classe [ElectionsSwing] de la façon suivante :

```
1.  package istia.st.elections.ui;
2.
3.  ...
4.  public class ElectionsSwing extends JFrame implements IElectionsUI {
5.
6.      // couche [metier]
7.      IElectionsMetier electionsMetier;
8.      public void setElectionsMetier(IElectionsMetier electionsMetier) {
9.          this.electionsMetier = electionsMetier;
10.     }
11.
12.     JPanel contentPane;
13.     BorderLayout borderLayout1 = new BorderLayout();
14.
15.     public ElectionsSwing() {
16.   ...
17. }
18.     /**
19.      * Component initialization.
20.      *
21.      * @throws java.lang.Exception
22.      */
23.     private void jbInit() throws Exception {
```

```
24. ...
25.     }
26.
27.     /**
28.      * affichage interface graphqiue
29.      */
30.     public void run() {
31.         // on s'affiche
32.         this.setVisible(true);
33.     }
34. }
```

- ligne 7 : définit un champ privé de type [IElectionsMetier]. Ce champ sera initialisé par Spring.
- lignes 8-10 : le setter nécessaire à Spring pour initialiser le champ précédent.

Parce que la classe [ElectionsSwing] implémente l'interface [IElectionsUI], nous pouvons utiliser la même classe de lancement que celle utilisée lorsque la couche [ui] était implémentée par la classe [ElectionsConsole] :

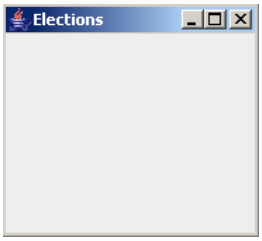
```
1.  package istia.st.elections.ui;
2.
3.  import istia.st.elections.ui.IElectionsUI;
4.  import org.springframework.beans.factory.xml.XmlBeanFactory;
5.  import org.springframework.core.io.ClassPathResource;
6.
7.  public class MainElections {
8.      public static void main(String[] arguments) {
9.          // instanciation couche [ui]
10.         IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
11.                 ClassPathResource("spring-config-electionsUIMetierDaoFile.xml"))).
12.                                 getBean("electionsUI");
13.         // exécution
14.         electionsUI.run();
15.     }
16. }
```

Le fichier de configuration [Spring] utilisé pourrait être le suivant :

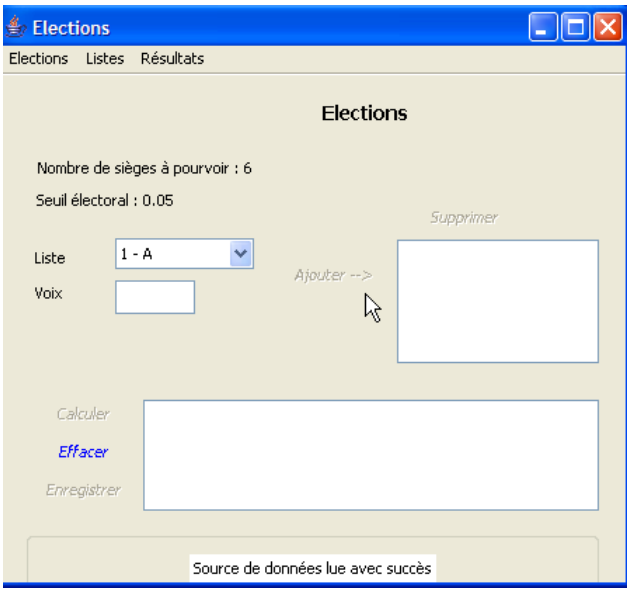
```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la couche [dao] -->
5.    <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoFile">
6.      <constructor-arg index="0">
7.        <value>data\elections-in-good.txt</value>
8.      </constructor-arg>
9.      <constructor-arg index="1">
10.       <value>data\elections-out.txt</value>
11.     </constructor-arg>
12.     <constructor-arg index="2">
13.       <value>data\elections-log.txt</value>
14.     </constructor-arg>
15.   </bean>
16.   <!-- la couche [metier] -->
17.   <bean id="electionsMetier" class="istia.st.elections.metier.ElectionsMetier">
18.     <property name="electionsDao">
19.       <ref local="electionsDao"/>
20.     </property>
21.   </bean>
22.   <!-- la couche [ui] -->
23.   <bean id="electionsUI" class="istia.st.elections.ui.ElectionsSwing">
24.     <property name="electionsMetier">
25.       <ref local="electionsMetier"/>
26.     </property>
27.   </bean>
28. </beans>
```

- lignes 23-27 : instancient la classe [ElectionsSwing] en initialisant son champ privé [electionsMetier] avec une référence sur la couche [metier] instanciée lignes 17-21.

Si on exécute la nouvelle application on obtient la fenêtre suivante :



Il nous reste un dernier point à régler. Lorsque l'interface graphique est affichée, certains de ses composants ont été initialisés :



On voit ci-dessus :

- que le combo a été rempli avec les noms des listes
- que le nombre de sièges à pourvoir et le seuil électoral sont indiqués
- que certains liens ont été désactivés
- qu'un message de succès est affiché en bas de la fenêtre

A quel moment vont se faire ces initialisations ? Elles ne peuvent se faire qu'après initialisation du champ [electionsMetier] de la classe [ElectionsSwing]. En effet, les noms des listes vont être demandés à la couche [metier]. L'initialisation de ce champ, on l'a dit, va être faite par Spring dans l'ordre suivant :

- utilisation du constructeur sans paramètres de la classe [ElectionsSwing]
- utilisation de la méthode [setElectionsMetier] pour initialiser le champ [electionsMetier].

Lorsque ceci sera fait, l'objet [ElectionsSwing] sera construit en mémoire mais pas encore affiché. C'est la classe [MainElections] qui demandera son affichage en appelant [ElectionsSwing].run().

Une première solution pour initialiser les composants de la fenêtre (combo, labels) est de placer celles-ci dans la méthode [run]. Une autre solution est d'utiliser une possibilité offerte par Spring. Le fichier de configuration instanciant [ElectionsSwing] devient le suivant :

```
1.  ...
2.    <!-- la couche [ui] -->
3.    <bean id="electionsUI" class="istia.st.elections.ui.ElectionsSwing" init-method="init">
4.      <property name="electionsMetier">
5.        <ref local="electionsMetier"/>
6.      </property>
7.    </bean>
8.  ...
```

- ligne 8 : l'attribut [init-method] indique le nom d'une méthode à exécuter après que l'objet [ElectionsSwing] ait été créé et initialisé.

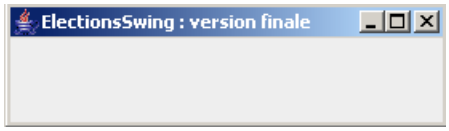
La structure finale de la classe [ElectionsSwing] sera donc la suivante :

```
1.  package istia.st.elections.ui;
2.
3.  import java.awt.BorderLayout;
4.  import java.awt.Dimension;
5.
6.  import javax.swing.JFrame;
7.  import javax.swing.JPanel;
8.
9.  import istia.st.elections.metier.IElectionsMetier;
10.
11. public class ElectionsSwing extends JFrame implements IElectionsUI {
12.
13.     // couche [metier]
14.     private IElectionsMetier electionsMetier;
15.     public void setElectionsMetier(IElectionsMetier electionsMetier) {
16.         this.electionsMetier = electionsMetier;
```

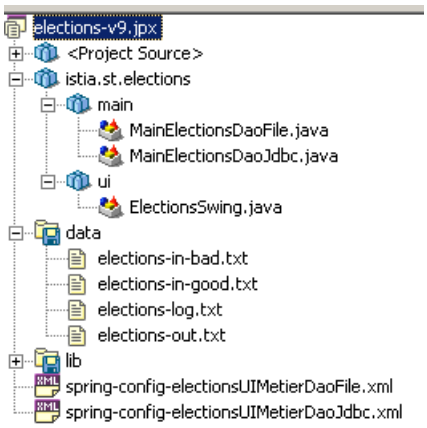
```
17.     }
18.
19.     // composants visuels
20.     JPanel contentPane;
21.     BorderLayout borderLayout1 = new BorderLayout();
22.
23.     public ElectionsSwing() {
24.         try {
25.             setDefaultCloseOperation(EXIT_ON_CLOSE);
26.             jbInit();
27.         } catch (Exception exception) {
28.             exception.printStackTrace();
29.         }
30.     }
31.
32.     /**
33.      * Component initialization.
34.      *
35.      * @throws java.lang.Exception
36.      */
37.     private void jbInit() throws Exception {
38.         contentPane = (JPanel) getContentPane();
39.         contentPane.setLayout(borderLayout1);
40.         setSize(new Dimension(400, 300));
41.         setTitle("Elections");
42.     }
43.
44.     /**
45.      * affichage interface graphqiue
46.      */
47.     public void run() {
48.         // on s'affiche
49.         this.setVisible(true);
50.     }
51.
52.     /**
53.      * initialisation fenêtre
54.      */
55.     public void init() {
56.         // un exemple d'initialisation - on change le titre
57.         setTitle("ElectionsSwing : version finale");
58.     }
59. }
```

- lignes 55-58 : la méthode [init] avec un exemple d'initialisation de la fenêtre.

Lorsque nous exécutons la classe [MainElections] nous obtenons alors la fenêtre suivante :



Le projet Jbuilder aura la forme finale suivante :



Nous testerons la couche [ui] avec deux configurations de la couche [dao] : celle implémentée à l'aide de fichiers texte et celle implémentée avec une base de données Firebird.

- [MainElectionsDaoFile.java] est la classe de lancement de l'application lorsque la couche [dao] est implémentée avec des fichiers texte.

- [MainElectionsDaoJdbc.java] est la classe de lancement de l'application lorsque la couche [dao] est implémentée avec une base de données.

Ces deux classes sont une copie de la classe [MainElections] utilisée avec l'implémentation [ElectionsConsole] de la couche [ui]. Celui de la classe [MainElectionsDaoFile] est le suivant :

```
1.  package istia.st.elections.main;
2.
3.  import istia.st.elections.ui.IElectionsUI;
4.  import org.springframework.beans.factory.xml.XmlBeanFactory;
5.  import org.springframework.core.io.ClassPathResource;
6.  import javax.swing.UIManager;
7.
8.  public class MainElectionsDaoFile {
9.      public static void main(String[] arguments) throws Exception{
10.         // le look and feel des fenêtres
11.         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
12.         // instanciation couche [ui]
13.         IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
14.                 ClassPathResource("spring-config-electionsUIMetierDaoFile.xml"))).
15.                                     getBean("electionsUI");
16.         // exécution
17.         electionsUI.run();
18.     }
19. }
```

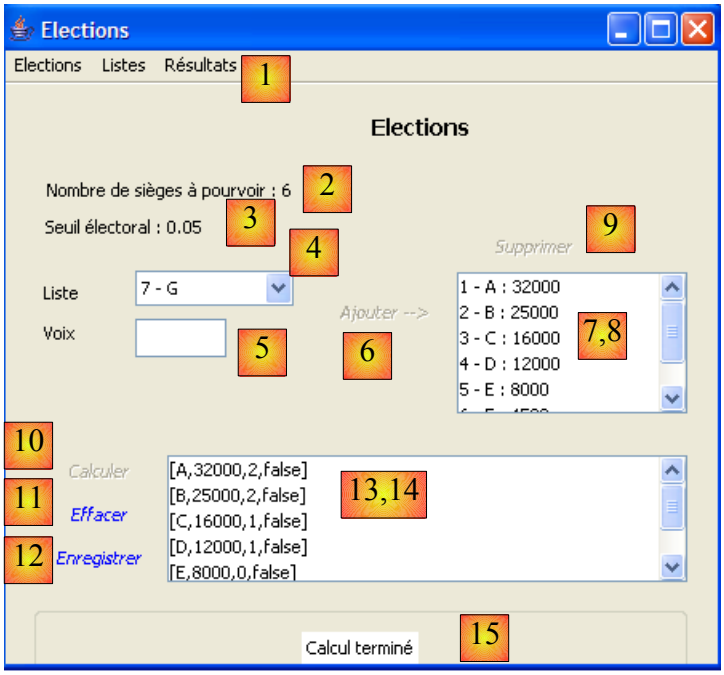
Une seule chose nouvelle apparaît : en ligne 11, on demande à la classe [UIManager] qui gère l'aspect des interfaces graphiques de donner à la nôtre celui des interfaces graphiques du système d'exploitation sous-jacent.

Le code de la classe [MainElectionsDaoJdbc] est identique au code précédent, aux lignes 12-15 près qui utilisent un fichier de configuration différent :

```
12.         // instanciation couche [ui]
13.         IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
14.                 ClassPathResource("spring-config-electionsUIMetierDaoJdbc.xml"))).
15. getBean("electionsUI");
```

4.4.4 Les gestionnaires d'événements et méthodes de [ElectionsSwing]

Nous allons maintenant examiner les différentes actions possibles pour l'utilisateur et écrire les méthodes de la couche [ui] chargées de les exécuter.



4.4.4.1 La méthode init

La méthode [init] a pour objectifs :

- de remplir le combo [4] avec les identifiants et noms des listes sous la forme [id - nom]
- d'afficher un message de succès dans [14]
- d'initialiser les labels [2] et [3]
- de désactiver certains liens

Question 3 : écrire la méthode [init]

4.4.4.2 Gérer l'état du lien [Ajouter]

Le lien [Ajouter] n'est actif que lorsque le champ (5) des voix est non vide.

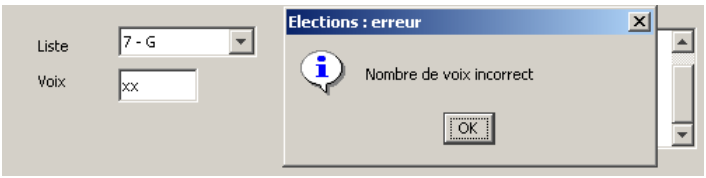
Question 4 : écrire la ou les méthodes permettant de gérer l'état du lien [Ajouter].

4.4.4.3 Affecter les voix à chaque liste

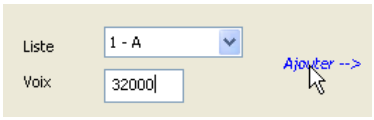
Pour chaque liste candidate de (4), on procède ainsi :

- choix d'une liste dans (4)
- saisie du nombre de voix dans (5)
- validation par un clic sur le lien [Ajouter]

Les erreurs de saisie sont signalées comme le montre l'exemple suivant :



Si le nombre de voix est correct, la liste est ajoutée dans le composants (8), le nombre de voix effacé et le lien [Ajouter] éteint :



Question 5 : Écrire la procédure gérant l'événement [Click] sur le lien [Ajouter].

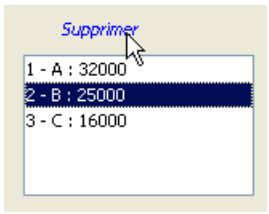
4.4.4.4 Gérer l'état du lien [Supprimer]

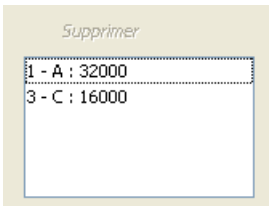
Le lien [Supprimer] n'est actif que lorsqu'un élément est sélectionné dans [7].

Question 6 : écrire la ou les méthodes permettant de gérer l'état du lien [Supprimer].

4.4.4.5 Supprimer une liste candidate

Le lien [Supprimer] permet de supprimer le couple (nom,voix) sélectionné dans (8). Une fois la suppression opérée, le lien [Supprimer] est éteint. Il ne sera rallumé que sur sélection d'une nouvelle liste dans (8).





Question 7 : Écrire la procédure gérant l'événement [Click] sur le lien [Supprimer].

4.4.4.6 Gérer l'état du lien [Calculer]

Le lien [Calculer] n'est actif que lorsqu'il existe au moins un élément dans [8].

Question 8 : écrire la ou les méthodes permettant de gérer l'état du lien [Calculer].

4.4.4.7 Calculer les sièges

Le lien [Calculer] permet de lancer le calcul des sièges et d'afficher les résultats dans (14). Une fois le calcul opéré avec succès, le lien [Enregistrer] doit s'allumer afin de permettre l'enregistrement des résultats. En cas d'échec (toutes les listes ont été éliminées), un message d'erreur est affiché dans [15]. Dans tous les cas, après calcul, le lien [Calculer] est désactivé.

Question 9 : Écrire la procédure gérant l'événement [Click] sur le lien [Calculer].

4.4.4.8 Enregistrer les résultats dans la source de données

Le lien [Enregistrer] permet d'enregistrer les résultats du calcul des sièges dans la source de données. Une fois l'enregistrement opéré avec succès, le lien [Enregistrer] est désactivé. En cas d'échec, un message d'erreur est affiché dans [15]. Dans tous les cas, le lien [Enregistrer] est ensuite désactivé.

Question 10 : Écrire la procédure gérant l'événement [Click] sur le lien [Enregistrer].

4.4.4.9 Effacer les résultats

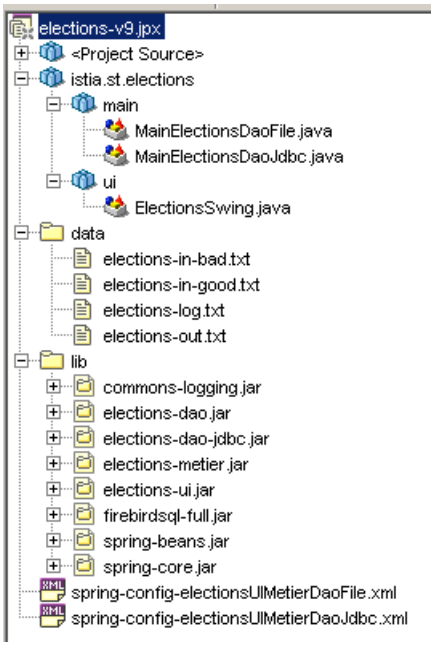
Le lien [Effacer] permet d'effacer les résultats affichés dans (14)

Question 11 : Écrire la procédure gérant l'événement [Click] sur le lien [Effacer].

4.4.5 Tests de la couche [ui]

4.4.5.1 Structure

Rappelons la structure du projet Jbuilder :



Nous testerons la couche [ui] avec deux configurations de la couche [dao] : celle implémentée à l'aide de fichiers texte et celle implémentée avec une base de données Firebird.

- [MainElectionsDaoFile.java] est la classe de lancement de l'application lorsque la couche [dao] est implémentée avec des fichiers texte.
- [MainElectionsDaoKdbc.java] est la classe de lancement de l'application lorsque la couche [dao] est implémentée avec une base de données..
- [ElectionsSwing.java] est la fenêtre graphique dérivée de [JFrame] présentée à l'utilisateur.
- [lib] est le dossier contenant les archives nécessaires à l'application :
  - [elections-dao.jar, elections-dao-jdbc] : pour la couche [dao]
  - [elections-metier.jar] : pour la couche [metier]
  - [elections-ui.jar] : pour l'interface [IElectionsUi] qui sera implémentée par la classe [ElectionsSwing.java]
  - [spring-core.jar, spring-beans.jar, commons-logging.jar] : pour Spring
  - [firebirdsql-full.jar] : pour le pilote JDBC du SGBD Firebird
- toutes les archives de [lib] sont placées dans le " ClassPath " du projet.
- [spring-config-electionsUIMetierDaoFile.xml] : le fichier de configuration Spring utilisé par [MainElectionsDaoFile] pour instancier les couches [ui, metier, dao]. Il sera placé dans le dossier [classes] du projet.
- [spring-config-electionsUIMetierDaoJdbc.xml] : le fichier de configuration Spring utilisé par [MainElectionsDaoJdbc] pour instancier les couches [ui, metier, dao]. Il sera placé dans le dossier [classes] du projet.

4.4.5.2 La classe [MainElectionsDaoFile]

Cette classe est chargée de lancer l'application lorsque la couche [dao] est implémentée par des fichiers texte. Son code est le suivant :

```
1.  package istia.st.elections.main;
2.
3.  import istia.st.elections.ui.IElectionsUI;
4.  import org.springframework.beans.factory.xml.XmlBeanFactory;
5.  import org.springframework.core.io.ClassPathResource;
6.  import javax.swing.UIManager;
7.
```

```
8.  public class MainElections {
9.      public static void main(String[] arguments) throws Exception{
10.         // le look and feel des fenêtres
11.         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
12.         // instanciation couche [ui]
13.         IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
14.                 ClassPathResource("spring-config-electionsUIMetierDaoFile.xml"))).
15.                                 getBean("electionsUI");
16.         // exécution
17.         electionsUI.run();
18.     }
19. }
```

4.4.5.3 Le fichier de configuration [spring-config-electionsUIMetierDaoFile.xml]

Ce fichier est utilisé ci-dessus, ligne 14 pour instancier les trois couches [ui, metier, dao] de l'application. Son contenu est le 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 couche [dao] -->
5.    <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoFile">
6.      <constructor-arg index="0">
7.        <value>data\elections-in-good.txt</value>
8.      </constructor-arg>
9.      <constructor-arg index="1">
10.       <value>data\elections-out.txt</value>
11.     </constructor-arg>
12.     <constructor-arg index="2">
13.       <value>data\elections-log.txt</value>
14.     </constructor-arg>
15.   </bean>
16.   <!-- la couche [metier] -->
17.   <bean id="electionsMetier" class="istia.st.elections.metier.ElectionsMetier">
18.     <property name="electionsDao">
19.       <ref local="electionsDao"/>
20.     </property>
21.   </bean>
22.   <!-- la couche [ui] -->
23.   <bean id="electionsUI" class="istia.st.elections.ui.ElectionsSwing" init-method="init">
24.     <property name="electionsMetier">
25.       <ref local="electionsMetier"/>
26.     </property>
27.   </bean>
28. </beans>
```

- lignes 5-15 : définissent l'implémentation de la couche [dao] - déjà vu. On notera que l'implémentation demandée ici est celle réalisée à l'aide de fichiers texte.
- lignes 17-21 : définissent l'implémentation de la couche [metier] - déjà vu
- lignes 23-27 : définissent l'implémentation de la couche [ui]
- ligne 23 : définit le bean " electionsUI " par instanciation de la classe [ElectionsSwing].
- lignes 24 - 26 : on initialise un champ [electionsMetier] de la classe [ElectionsSwing] avec la référence à l'implémentation [metier] faite lignes 17-20. Ainsi la couche [ui] saura-t-elle comment accéder à la couche [metier].

Travail pratique : tester l'application avec une couche [dao] implémentée avec des fichiers texte.

4.4.5.4 La classe [MainElectionsDaoJdbc]

Cette classe est chargée de lancer l'application lorsque la couche [dao] est implémentée par une base de données. Son code est identique au précédent au fichier de configuration près (ligne 14 dans MainElectionsDaoFile, ligne 4 ci-dessous) :

```
1.  ...
2.          // instanciation couche [ui]
3.          IElectionsUI electionsUI = (IElectionsUI) (new XmlBeanFactory(new
4.                  ClassPathResource("spring-config-electionsUIMetierDaoJdbc.xml"))).
5.                                  getBean("electionsUI");
6.  ...
```

4.4.5.5 Le fichier de configuration [spring-config-electionsUIMetierDaoJdbc.xml]

Ce fichier est utilisé ci-dessus, ligne 4 pour instancier les trois couches [ui, metier, dao] de l'application. Son contenu est le 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 données -->
5.    <bean id="electionsDao" class="istia.st.elections.dao.ElectionsDaoJDBC">
6.      <constructor-arg index="0">
7.        <value>org.firebirdsql.jdbc.FBDriver</value>
8.      </constructor-arg>
9.      <constructor-arg index="1">
10.       <value>jdbc:firebirdsql:localhost/3050:d:/data/databases/firebird/dbelections.gdb</value>
11.     </constructor-arg>
12.     <constructor-arg index="2">
13.       <value>SYSDBA</value>
14.     </constructor-arg>
15.     <constructor-arg index="3">
16.       <value>masterkey</value>
17.     </constructor-arg>
18.   </bean>
19.   <!-- la couche [metier] -->
20.   <bean id="electionsMetier" class="istia.st.elections.metier.ElectionsMetier">
21.     <property name="electionsDao">
22.       <ref local="electionsDao"/>
23.     </property>
24.   </bean>
25.   <!-- la couche [ui] -->
26.   <bean id="electionsUI" class="istia.st.elections.ui.ElectionsSwing" init-method="init">
27.     <property name="electionsMetier">
28.       <ref local="electionsMetier"/>
29.     </property>
30.   </bean>
31. </beans>
```

Ce fichier de configuration est identique au précédent si ce n'est que maintenant la couche [dao] est implémentée avec une base de données (lignes 4-18).

Travail pratique : tester l'application avec une couche [dao] implémentée avec une base de données Firebird. Ne pas oublier de lancer le SGBD auparavant.

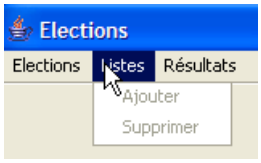
4.4.6 Améliorations

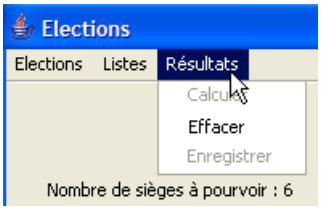
L'interface graphique précédente peut être améliorée de diverses façons.

Tout d'abord, l'utilisateur peut, par oubli, ne pas saisir les voix de toutes les listes présentes dans le combo et par ailleurs, il peut, par erreur, saisir plusieurs fois les voix d'une même liste.

Question 12 : Améliorez l'algorithme afin que ces deux cas ne puissent pas se produire. Une solution simple est de gérer un dictionnaire des listes saisies dont les clés seraient les éléments du combo. On fera également en sorte que le lien [Calculer] ne soit allumé que lorsque la totalité des listes ont été saisies.

On peut également introduire un menu dans l'interface qui pourrait avoir la structure suivante :



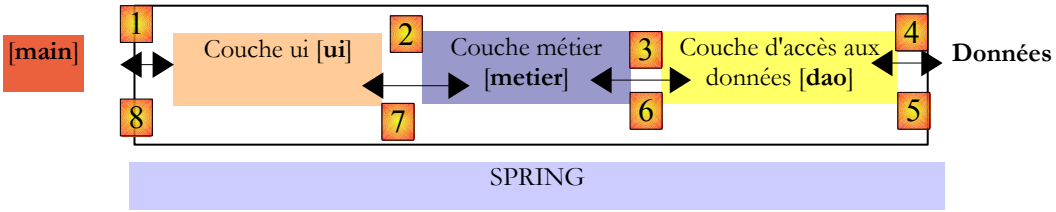


Ces options de menu correspondent une à une aux divers liens déjà présents dans le formulaire.

Question 13 : Introduisez ce menu dans l'interface graphique et écrivez les gestionnaires d'événements associés. Comme le rôle de ces options est identique à celui des liens déjà présents dans le formulaire, leurs gestionnaires d'événements feront appel systématiquement aux gestionnaires d'événements associés aux liens.

5 Conclusion

Nous avons construit une application ayant l'architecture suivante :



- chaque couche a été rendue indépendante des autres par l'utilisation d' interfaces Java
- l'instanciation des couches a été faite à l'aide de fichiers de configuration Spring. Ceci permet de changer une couche sans changer les deux autres.
- nous avons utilisé ces concepts en donnant deux implémentations différentes à chacune des deux couches [ui] et [dao].

Cette architecture dite **trois couches** apporte une grande souplesse dans la construction et la maintenance d'une application Java.

Table des matières

1 PARTIE 1 1

2 PARTIE 2 4

2.1 La classe [ListeElectorale] 4
2.2 Création d'une classe d'exception 6
2.3 Une classe de test unitaire 8
2.4 MainElections : version 2 12

3 PARTIE 3 14

3.1 Introduction 14
3.2 Les interfaces de l'application [Elections] 15
3.3 La couche [dao] 19
3.3.1 La classe de test JUnit 19
3.3.2 Configuration des tests unitaires 21
3.3.3 Mise en oeuvre des tests unitaires 25
3.3.4 Implémentation [ElectionsDaoFile] de la couche [dao] 26
3.3.5 Création de l'archive [elections-dao.jar] de la couche [dao] 26
3.3.6 Conclusion 26
3.4 La couche [metier] 27
3.4.1 L'interface [IElectionsMetier] 27
3.4.2 La classe de test 27
3.4.3 La classe d'implémentation [ElectionsMetier] 29
3.4.4 Mise en oeuvre des tests unitaires de la couche [metier] 30
3.4.5 Création de l'archive [elections-metier.jar] de la couche [metier] 32
3.4.6 Conclusion 32
3.5 La couche [ui] 32
3.5.1 L'interface [IElectionsUI] 32
3.5.2 Le lanceur de l'application 33
3.5.3 La classe d'implémentation [ElectionsConsole] 34
3.5.4 Tests de l'application [Elections] 35
3.5.5 Création de l'archive [elections-ui.jar] de la couche [ui] 36
3.6 Création de l'archive [elections-main.jar] du programme principal 37
3.7 Déploiement de l'application [Elections] 37

4 PARTIE 4 39

4.1 Introduction 39
4.2 La base de données [dbelections] 39
4.3 Implémentation de la couche [dao] avec une base de données 40
4.3.1 L'interface [IElectionsDao] 40
4.3.2 Les tests unitaires 41
4.3.3 La classe [ElectionsDaoJDBC] 42
4.3.4 Configuration des tests de la couche [dao] 44
4.3.5 Création de l'archive [elections-dao-jdbc.jar] 45
4.4 Implémentation de la couche [ui] avec une interface Swing 46
4.4.1 Description de l'interface swing 46
4.4.2 Fonctionnement de l'application 47
4.4.3 La classe [ElectionsSwing] d'implémentation de la couche [ui] 47
4.4.4 Les gestionnaires d'événements et méthodes de [ElectionsSwing] 57
4.4.4.1 La méthode init 57
4.4.4.2 Gérer l'état du lien [Ajouter] 57
4.4.4.3 Affecter les voix à chaque liste 57
4.4.4.4 Gérer l'état du lien [Supprimer] 58
4.4.4.5 Supprimer une liste candidate 58
4.4.4.6 Gérer l'état du lien [Calculer] 58
4.4.4.7 Calculer les sièges 58
4.4.4.8 Enregistrer les résultats dans la source de données 58
4.4.4.9 Effacer les résultats 59
4.4.5 Tests de la couche [ui] 59
4.4.5.1 Structure 59
4.4.5.2 La classe [MainElectionsDaoFile] 60

4.4.5.3Le fichier de configuration [spring-config-electionsUIMetierDaoFile.xml] 60
4.4.5.4La classe [MainElectionsDaoJdbc] 60
4.4.5.5Le fichier de configuration [spring-config-electionsUIMetierDaoJdbc.xml] 61
4.4.6AMÉLIORATIONS 61
5CONCLUSION 62