Exercice Java – Gestion de Rv

Lectures conseillées :
- ✗ [1] : Spring IoC [http://tahe.developpez.com/java/springioc]
- ✗ [2] : Chapitre 17 du document [http://tahe.developpez.com/java/baseswebmvc] qui décrit l'utilisation de l'outil **iBatis** et son utilisation avec une base de données Firebird.
- ✗ [3] : la classe [SimpleDateFormat] : [http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html]
- ✗ [4] : Introduction au langage SQL avec le SGBD Firebird [http://tahe.developpez.com/divers/sql-firebird/]

Thèmes : Accès aux données, Architectures 2 couches, Spring IoC, iBatis, Classes et Interfaces, SQL
Niveau : intermédiaire
Durée : 8 / 15 h

Les exemples de code présentés utilisent les collections génériques, disponibles qu'à partir du JDK 1.5.

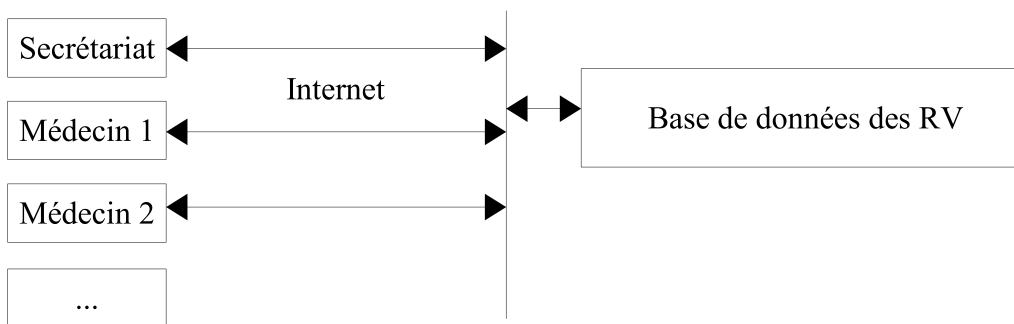
Le texte est long parce qu'il détaille l'environnement dans lequel prennent place les questions. Ces questions ne sont pas particulièrement difficiles. Le sujet mesure donc aussi bien la capacité à comprendre un problème qu'à le traiter.

1 Le problème

Une société de services en informatique [ISTIA-IAIE] désire proposer un service de prise de rendez-vous. Le premier marché visé est celui des médecins travaillant seuls. Ceux-ci n'ont en général pas de secrétariat. Les clients désirant prendre rendez-vous téléphonent alors directement au médecin. Celui-ci est donc dérangé fréquemment au cours d'une journée ce qui diminue sa disponibilité à ses patients. La société [ISTIA-IAIE] souhaite leur proposer un service de prise de rendez-vous fonctionnant sur le principe suivant :

- un service secrétariat assure les prises de RV pour un grand nombre de médecins. Ce service peut être réduit à une unique personne. Le salaire de celle-ci est mutualisé entre tous les médecins utilisant le service de RV.
- le service secrétariat et tous les médecins sont reliés à Internet
- les RV sont enregistrés dans une base de données centralisée, accessible par Internet, par le secrétariat et les médecins
- la prise de RV est normalement faite par le secrétariat. Elle peut être faite également par les médecins eux-mêmes. C'est le cas notamment lorsqu'à la fin d'une consultation, le médecin fixe lui-même un nouveau RV à son patient.

L'architecture du service de prise de RV est le suivant :

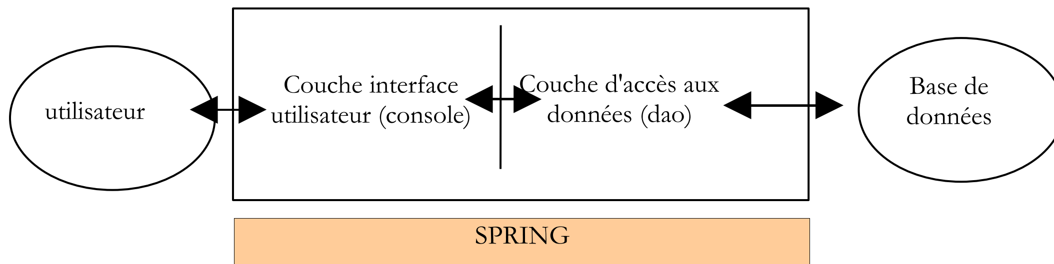


Les médecins gagnent en efficacité s'ils n'ont plus à gérer les RV. S'ils sont suffisamment nombreux, leur contribution aux frais de fonctionnement du secrétariat sera faible.

La société [ISTIA-IAIE] décide de confier à un stagiaire l'élaboration d'une première maquette du service, une application console basique. C'est cette maquette que nous nous proposons de réaliser ici.

2 Les éléments de l'application

L'application [RdvMedecins] aura une architecture à deux couches :

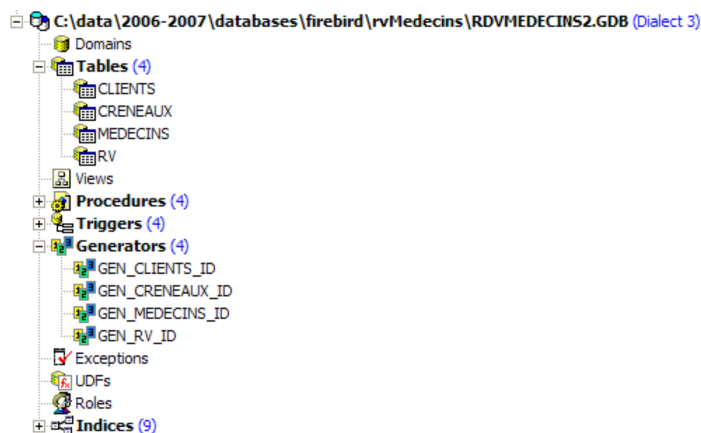


- les deux couches sont rendues indépendantes grâce à l'utilisation d'interfaces Java,
- l'intégration des couches est réalisée par Spring.

La couche [console] est destinée à être remplacée ultérieurement par une couche [web].

2.1 La base de données

La base de données qu'on appellera [RDVMEDECINS] est une base de données Firebird avec quatre tables :



2.1.1 La table [MEDECINS]

Elle contient des informations sur les médecins gérés par l'application [RdvMedecins].

CP	Nom du Ch...	Type du C...	Dom...	Longueur
1	ID	INTEGER		
	VERSION	INTEGER		
	NOM	VARCHAR		30
	PRENOM	VARCHAR		30
	TITRE	VARCHAR		5

ID	VERSION	TITRE	NOM	PRENOM
1	1	Mme	PELISSIER	Marie
2	1	Mr	BROMARD	Jacques
3	1	Mr	JANDOT	Philippe
4	1	Melle	JACQUEMOT	Justine

- ID : n° identifiant le médecin - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du médecin
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

2.1.2 La table [CLIENTS]

Les clients des différents médecins sont enregistrés dans la table [CLIENTS] :

CP	Nom du Ch...	Type du Champ	Domaine	Longueur
PK1	ID	INTEGER		
	VERSION	INTEGER		
	NOM	VARCHAR		30
	PRENOM	VARCHAR		30
	TITRE	VARCHAR		5

ID	VERSION	TITRE	NOM	PRENOM
1	1	Mr	MARTIN	Jules
2	1	Mme	GERMAN	Christine
3	1	Mr	JACQUARD	Jules
4	1	Melle	BISTROU	Brigitte

- ID : n° identifiant le client - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du client
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

2.1.3 La table [CRENEAUX]

Elle liste les créneaux horaires où les RV sont possibles :

FK	CP	Nom du Ch...	Type du Champ
	PK1	ID	INTEGER
		VERSION	INTEGER
FK		ID_MEDECIN	INTEGER
		HDEBUT	INTEGER
		MDEBUT	INTEGER
		HFIN	INTEGER
		MFIN	INTEGER

ID	VERSION	ID_MEDECIN	HDEBUT	MDEBUT	HFIN	MFIN
1	1	1	8	0	8	20
2	1	1	8	20	8	40
3	1	1	8	40	9	0
4	1	1	9	0	9	20
5	1	1	9	20	9	40
6	1	1	9	40	10	0
7	1	1	10	0	10	20
8	1	1	10	20	10	40
9	1	1	10	40	11	0
10	1	1	11	0	11	20
11	1	1	11	20	11	40
12	1	1	11	40	12	0
13	1	1	14	0	14	20
14	1	1	14	20	14	40
15	1	1	14	40	15	0
16	1	1	15	0	15	20
17	1	1	15	20	15	40
18	1	1	15	40	16	0
19	1	1	16	0	16	20
20	1	1	16	20	16	40
21	1	1	16	40	17	0
22	1	1	17	0	17	20
23	1	1	17	20	17	40
24	1	1	17	40	18	0
25	1	2	8	0	8	20
26	1	2	8	20	8	40
27	1	2	8	40	9	0
28	1	2	9	0	9	20
29	1	2	9	20	9	40
30	1	2	9	40	10	0
31	1	2	10	0	10	20
32	1	2	10	20	10	40
33	1	2	10	40	12	0
34	1	2	12	0	12	20
35	1	2	12	20	12	40
36	1	2	12	40	12	0
37	1	3	8	0	8	20
38	1	3	8	20	8	40
39	1	3	8	40	9	0
40	1	3	9	0	9	20
41	1	3	9	20	9	40
42	1	3	9	40	10	0
43	1	3	10	0	10	20
44	1	3	10	20	10	40
45	1	3	10	40	12	0
46	1	3	12	0	12	20

1

- ID : n° identifiant le créneau horaire - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- ID_MEDECIN : n° identifiant le médecin auquel appartient ce créneau – clé étrangère sur la colonne ID de la table [MEDECINS].
- HDEBUT : heure début créneau
- MDEBUT : minutes début créneau
- HFIN : heure fin créneau
- MFIN : minutes fin créneau

La seconde ligne de la table [CRENEAUX] (cf [1] ci-dessus) indique, par exemple, que le créneau n° 2 commence à 8 h 20 et se termine à 8 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER).

2.1.4 La table [RV]

Elle liste les RV pris pour chaque médecin :

FK	PK	Field Name	Field Type
	PK1	ID	INTEGER
		JOUR	DATE
FK		ID_CRENEAU	INTEGER
FK		ID_CLIENT	INTEGER

ID	JOUR	ID_CRENEAU	ID_CLIENT
3	22.08.2006	1	2
4	23.08.2006	4	3
5	23.08.2006	20	4
6	23.08.2006	12	1
7	10.09.2006	10	2
8	23.08.2006	6	1
9	23.08.2006	7	3

1

- ID : n° identifiant le RV de façon unique – clé primaire
- JOUR : jour du RV
- ID_CRENEAU : créneau horaire du RV - clé étrangère sur le champ [ID] de la table [CRENEAUX] – fixe à la fois le créneau horaire et le médecin concerné.
- ID_CLIENT : n° du client pour qui est faite la réservation – clé étrangère sur le champ [ID] de la table [CLIENTS]

Cette table a une contrainte d'unicité sur les valeurs jointes des colonnes (JOUR, ID_CRENEAU) :

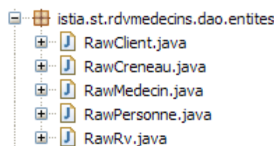
```
ALTER TABLE RV ADD CONSTRAINT UNQ1_RV UNIQUE (JOUR, ID_CRENEAU);
```

Si une ligne de la table[RV] a la valeur (JOUR1, ID_CRENEAU1) pour les colonnes (JOUR, ID_CRENEAU), cette valeur ne peut se retrouver nulle part ailleurs. Sinon, cela signifierait que deux RV ont été pris au même moment pour le même médecin. D'un point de vue programmation Java, le pilote JDBC de la base lance une *SQLException* lorsque ce cas se produit.

La ligne 3 (cf [1] ci-dessus) signifie qu'un RV a été pris pour le créneau n° 20 et le client n° 4 le 23/08/2006. La table [CRENEAUX] nous apprend que le créneau n° 20 correspond au créneau horaire 16 h 20 - 16 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER). La table [CLIENTS] nous apprend que le client n° 4 est Melle Brigitte BISTROU.

2.2 Les objets de l'application

Les objets manipulés par la couche [dao] de l'application [RdvMedecins] reflètent les lignes des tables de la base de données. Ce sont les suivants :



2.2.1 Classe [RawPersonne]

La classe [RawPersonne] représente une ligne de la table [MEDECINS] ou de la table [CLIENTS]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RawPersonne {
4.          // caractéristiques d'une personne
5.          private int id;
6.          private int version;
7.          private String titre;
8.          private String nom;
9.          private String prenom;
10.
11.         // constructeur par défaut
12.         public RawPersonne() {
13.
14.         }
15.
16.         // constructeur avec paramètres
17.         public RawPersonne(int id, int version, String titre, String nom,
18.                         String prenom) {
19. ...
20.         }
21.
22.         // constructeur par recopie
23.         public RawPersonne(RawPersonne personne) {
24. ...
25.         }
26.
27.         // toString
28.         public String toString() {
29.                 return "[" + id + "," + version + "," + titre + "," + prenom + ","
30.                                 + nom + "]";
31.         }
32.
33.         // setters and getters
34. ...
35. }
```

- ligne 5 : le n° de la personne
- ligne 6 : la version de l'objet
- ligne 7 : son titre (Mr, Mme, Melle)
- ligne 8 : son nom
- ligne 9 : son prénom
- lignes 17-19 : le constructeur permettant d'initialiser une instance [RawPersonne] avec les cinq informations nécessaires
- lignes 23-25 : le constructeur permettant d'initialiser une instance [RawPersonne] avec les valeurs (id, version, titre, nom, prenom) d'une autre personne.
- lignes 28-31 : la méthode [toString] de la classe.
- lignes 33 et au-delà : les méthodes get / set associées à chacun des champs privés de la classe.

2.2.2 Classe [RawMedecin]

La classe [RawMedecin] représente une ligne de la table [MEDECINS]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RawMedecin extends RawPersonne{
4.          // constructeur par défaut
5.          public RawMedecin(){
6.          }
7.
8.          // constructeur avec paramètres
9.          public RawMedecin(int id, int version, String titre, String nom, String prenom){
10.                 // parent
11.                 super(id,version,titre,nom,prenom);
12.         }
13.
14.         // constructeur par recopie
15.         public RawMedecin(RawMedecin medecin){
16.                 // parent
17.                 super(medecin);
18.         }
19.
20. }
```

- ligne 3 : la classe dérive de la classe [RawPersonne] et en a donc toutes les caractéristiques.

2.2.3 Classe [RawClient]

La classe [RawClient] représente une ligne de la table [CLIENTS]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RawClient extends RawPersonne{
4.          // constructeur par défaut
5.          public RawClient(){
6.          }
7.
8.          // constructeur avec paramètres
9.          public RawClient(int id, int version, String titre, String nom, String prenom){
10.                 // parent
11.                 super(id,version,titre,nom,prenom);
12.         }
13.
14.         // constructeur par recopie
15.         public RawClient(RawClient client){
16.                 // parent
17.                 super(client);
18.         }
19. }
```

- ligne 3 : la classe dérive de la classe [RawPersonne] et en a donc toutes les caractéristiques.

2.2.4 Classe [RawCreneau]

La classe [RawCreneau] représente une ligne de la table [CRENEAUX]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
```

```
2.
3.  public class RawCreneau {
4.
5.          // caractéristiques d'un créneau de RV
6.          private int id;
7.          private int version;
8.          private int idMedecin;
9.          private int hdebut;
10.         private int mdebut;
11.         private int hfin;
12.         private int mfin;
13.
14.         // constructeur par défaut
15.         public RawCreneau() {
16.
17.         }
18.
19.         // constructeur avec paramètres
20.         public RawCreneau(int id, int version, int idMedecin, int hDebut,
21.                         int mDebut, int hFin, int mFin) {
22. ...
23.         }
24.
25.         // constructeur par recopie
26.         public RawCreneau(RawCreneau creneau) {
27. ...
28.         }
29.
30.         // toString
31.         public String toString() {
32.                 return "[" + id + "," + version + "," + idMedecin + "," + hdebut + ":"
33.                                 + mdebut + "," + hfin + ":" + mfin + "]";
34.         }
35.
36.         // setters - getters
37. ...
38. }
```

- ligne 6 : le n° du créneau
- ligne 7 : le n° de version de l'objet
- ligne 8 : le n° du médecin auquel appartient le créneau
- ligne 9 : l'heure de début
- ligne 10 : les minutes de début de créneau
- ligne 11 : l'heure de fin
- ligne 12 : les minutes de fin de créneau
- lignes 20-23 : le constructeur permettant d'initialiser une instance [RawCreneau] avec les sept informations nécessaires
- lignes 26-28 : le constructeur permettant d'initialiser une instance [RawCreneau] avec les informations d'un autre objet [RawCreneau]
- lignes 31-34 : la méthode [toString] de la classe.
- lignes 36 et au-delà : les méthodes get / set associées à chacun des champs privés de la classe.

2.2.5 Classe [RawRv]

La classe [RawRv] représente une ligne de la table [RV]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.
6.  public class RawRv {
7.          // caractéristiques
8.          private int id;
9.          private Date jour;
10.         private int idClient;
11.         private int idCreneau;
12.
13.         // constructeur par défaut
14.         public RawRv() {
15.
16.         }
17.
18.         // constructeur avec paramètres
19.         public RawRv(int id, Date jour, int idClient, int idCreneau) {
20. ...
21.         }
22.
23.         // constructeur par recopie
```

```
24.        public RawRv(RawRv rv) {
25. ...
26.        }
27.
28.        // toString
29.        public String toString() {
30.                return "[" + id + ","
31.                                + new SimpleDateFormat("dd/MM/yyyy").format(jour)
32.                                + "," + idClient + "," + idCreneau + "]";
33.        }
34.
35.        // getters - settters
36. ...
37. }
```

- ligne 8 : le n° du RV
- ligne 9 : le jour du RV de type [java.util.Date]
- ligne 10 : le n° du client
- ligne 11 : le n° du créneau
- lignes 16-23 : le constructeur permettant d'initialiser une instance [RawRv] avec les quatre informations nécessaires
- lignes 24-26 : le constructeur permettant d'initialiser une instance [RawRv] avec les quatre informations d'une autre instance de type [RawRv].
- lignes 29-33 : la méthode [toString] de la classe. L'instruction :

```
new SimpleDateFormat("dd/MM/yyyy").format(dateJour)
```

fait la conversion de la valeur *dateJour* de type [java.util.Date] vers un type String au format " dd/MM/yyyy " où dd (day) désigne les deux chiffres du jour dans le mois, MM (month) les deux chiffres du mois dans l'année, yyyy (year) les quatre chiffres de l'année. L'instruction :

```
new SimpleDateFormat("yyyy:MM:dd").parse(stringJour)
```

fait l'opération inverse : elle fait la conversion de la valeur *stringJour* de type [String] ayant le format "yyyy:MM:dd" où dd, MM, yyyy ont la même signification que précédemment vers un type [java.util.Date]. Ainsi l'opération :

```
java.util.Date dateJour = new SimpleDateFormat("yyyy:MM:dd").parse("2006:08:23)
```

affecte à *dateJour* de type [java.util.Date] la date du 23 août 2006. L'expression

```
String stringJour=new SimpleDateFormat("dd/MM/yyyy").format(dateJour))
```

affecte à *stringJour* de type [String] la chaîne "23/08/2006".

L'utilisation de la classe [SimpleDateFormat] nécessite l'import de la ligne 3.

- lignes 35 et au-delà : les méthodes get / set associées à chacun des champs privés de la classe.

2.2.6 Classe [RdvMedecinsException]

La classe [RdvMedecinsException] est la suivante :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RdvMedecinsException extends RuntimeException {
4.
5.         private static final long serialVersionUID = 1L;
6.
7.         // champs privés
8.         private int code = 0;
9.         private Exception exception;
10.
11.        // constructeurs
12.        public RdvMedecinsException() {
13.                super();
14.        }
15.
16.        public RdvMedecinsException(String message) {
17.                super(message);
18.        }
19.
20.        public RdvMedecinsException(String message, Throwable cause) {
21.                super(message, cause);
22.        }
23.
24.        public RdvMedecinsException(Throwable cause) {
25.                super(cause);
```

```
26.         }
27.
28.         public RdvMedecinsException(String message, int code) {
29.                 super(message);
30.                 setCode(code);
31.         }
32.
33.         public RdvMedecinsException(String message, Exception exception, int code) {
34.                 super(message);
35.                 setException(exception);
36.                 setCode(code);
37.         }
38.
39.         // getters - setters
40.         public int getCode() {
41.                 return code;
42.         }
43.
44.         public void setCode(int code) {
45.                 this.code = code;
46.         }
47.
48.         public Exception getException() {
49.                 return exception;
50.         }
51.
52.         public void setException(Exception exception) {
53.                 this.exception = exception;
54.         }
55. }
```

- ligne 3 : la classe dérive de [RuntimeException] et est donc un type d'exception non contrôlée.
- la classe [RdvMedecinsException] va servir à encapsuler une exception, contrôlée ou non, dans un type d'exception non contrôlée.
- pour différencier les exceptions de type [RdvMedecinsException] entre-elles, on pourra leur affecter un code qui sera mémorisé par le champ privé de la ligne 8. Un code Java interceptant une exception de type [RdvMedecinsException] aura accès à ce code d'erreur grâce à la méthode [getCode] (lignes 40-42).
- l'exception encapsulée le sera dans le champ privé de la ligne 9. Un code Java interceptant une exception de type [RdvMedecinsException] aura accès à cette exception grâce à la méthode [getException] (lignes 48-50).
- lignes 11-37 : les différentes façons de construire un objet de type [RdvMedecinsException].

Un exemple d'utilisation de la classe [RdvMedecinsException] pourrait être le suivant :

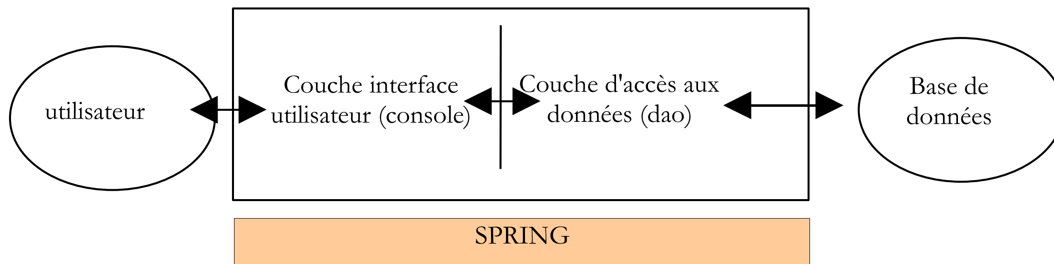
```
1.       ...
2.       try {
3.          paramètres.put("jour", new SimpleDateFormat("yyyy:MM:dd")
4.                         .parse(jour));
5.       } catch (ParseException e) {
6.          throw new RdvMedecinsException("Jour incorrect", e, 1);
7.       }
8.  ...
```

- lignes 2-3 : la méthode *[SimpleDateFormat].parse* essaie de transformer une chaîne telle que " 2006:08:23 " en un type [java.util.Date] correspondant au jour 23 août 2006. Si la chaîne ne correspond pas à une date valide, la méthode lance une exception de type [ParseException] dérivée de [Exception].
- lignes 5-7 : parce qu'on ne veut remonter que des exceptions non contrôlées (c.a.d. dérivées de RuntimeException), on encapsule l'exception dans une exception de type [RdvMedecinsException]. On utilise ici, le constructeur des lignes 33-37.

3 Application console de gestion des Rv

3.1 L'interface de la couche [dao]

Revenons à l'architecture de l'application :



L'interface Java de la couche [dao] sera la suivante :

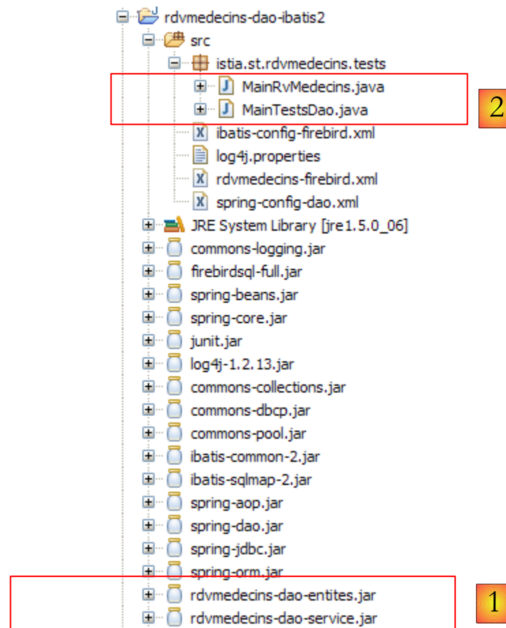
```
1.  package istia.st.rdvmedecins.dao.service;
2.  
3.  import istia.st.rdvmedecins.dao.entites.RawClient;
4.  import istia.st.rdvmedecins.dao.entites.RawCreneau;
5.  import istia.st.rdvmedecins.dao.entites.RawMedecin;
6.  import istia.st.rdvmedecins.dao.entites.RawRv;
7.  
8.  import java.util.List;
9.  
10. public interface IDao {
11. 
12.         // liste des clients
13.         public List<RawClient> getAllClients();
14.         // liste des Médecins
15.         public List<RawMedecin> getAllMedecins();
16.         // liste des créneaux horaires d'un médecin
17.         public List<RawCreneau> getAllCreneaux(int idMedecin);
18.         // liste des Rv d'un médecin, un jour donné
19.         public List<RawRv> getRvMedecinJour(int idMedecin, String jour);
20.         // ajouter un RV
21.         public int ajouterRv(String jour, int idCreneau, int idClient);
22.         // supprimer un RV
23.         public int supprimerRv(int idRv);
24. 
25. }
```

- ligne 13 : la méthode [getAllClients] met les lignes de la table [CLIENTS] dans une liste d'objets de type [RawClient]
- ligne 15 : la méthode [getAllMedecins] met les lignes de la table [MEDECINS] dans une liste d'objets de type [RawMedecin]
- ligne 17 : la méthode [getAllCreneaux] met les lignes de la table [CRENEAUX] pour lesquelles la colonne ID_MEDECIN a la valeur du paramètre **idMedecin** dans une liste d'objets de type [RawCreneau]
- ligne 19 : la méthode [getRvMedecinJour] met dans une liste d'objets de type [RawRv] les lignes de la table [RV] pour lesquelles :
  - la colonne ID_CRENEAU correspond au médecin dont le n° a pour valeur le paramètre **idMedecin**
  - la colonne JOUR (de type Date) correspond au paramètre **jour** (de type String).
- ligne 21 : la méthode [ajouterRv] ajoute dans la table [RV] un RV pour :
  - le client **idClient**,
  - le créneau horaire **idCreneau** (donc pour un médecin précis) du jour **jour**.

  Elle rend comme résultat, la clé primaire ID de la ligne ajoutée. La valeur de celle-ci est générée par le SGBD lui-même comme nous le verrons ultérieurement.
- ligne 23 : la méthode [supprimerRv] supprime la ligne de la table [RV] identifiée par le n° **idRV**. Elle rend comme résultat le nombre de lignes supprimées, 0 ou 1.

3.2 Application de test

Un projet Eclipse de test basique de la couche [dao] pourrait être le suivant :



- l'implémentation de la couche [dao] a été encapsulée dans les archives [1]. Elle est utilisée par le programme de test.

Le programme de test **MainTestsDao** [2] est le suivant :

```
1.  package istia.st.rdvmedecins.tests;
2.
3.  import java.util.List;
4.
5.  import istia.st.rdvmedecins.dao.entites.RdvMedecinsException;
6.  import istia.st.rdvmedecins.dao.service.IDao;
7.
8.  import org.springframework.beans.factory.xml.XmlBeanFactory;
9.  import org.springframework.core.io.ClassPathResource;
10.
11. public class MainTestsDao {
12.
13.        public static void main(String[] args) {
14.                // instanciation couche [dao]
15.                IDao dao = (IDao) (new XmlBeanFactory(new ClassPathResource(
16.                               "spring-config-dao.xml"))).getBean("dao");
17.                // affichage clients
18.                try {
19.                       display("Liste des clients :", dao.getAllClients());
20.                } catch (RdvMedecinsException ex) {
21.                       System.out.println(ex.getMessage()+ " : "+ex.getException());
22.                }
23.                // affichage médecins
24.                try {
25.                       display("Liste des médecins :", dao.getAllMedecins());
26.                } catch (RdvMedecinsException ex) {
27.                       System.out.println(ex.getMessage()+ " : "+ex.getException());
28.                }
29.                // affichage créneaux d'un médecin
30.                try {
31.                       display("Liste des créneaux du médecin n° 1 :", dao
32.                                      .getAllCreneaux(1));
33.                } catch (RdvMedecinsException ex) {
34.                       System.out.println(ex.getMessage()+ " : "+ex.getException());
35.                }
36.                // liste des Rv d'un médecin, un jour donné
37.                try {
38.                       display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
39.                                      .getRvMedecinJour(1, "2006:08:23"));
40.                } catch (RdvMedecinsException ex) {
41.                       System.out.println(ex.getMessage()+ " : "+ex.getException());
42.                }
43.                // ajouter un RV
44.                System.out.println("Ajout d'un Rv au médecin n° 1");
45.                int idRv = 0;
46.                try {
47.                       idRv = dao.ajouterRv("2006:08:23", 2, 2);
48.                       display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
49.                                      .getRvMedecinJour(1, "2006:08:23"));
50.                } catch (RdvMedecinsException ex) {
```

```
51.                         System.out.println(ex.getMessage()+ " : "+ex.getException());
52.                 }
53.                 // ajouter un RV dans le même créneau du même jour
54.                 System.out.println("Ajouter un RV dans le même créneau du même jour");
55.                 try {
56.                         idRv = dao.ajouterRv("2006:08:23", 2, 4);
57.                         display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
58.                                         .getRvMedecinJour(1, "2006:08:23"));
59.                 } catch (RdvMedecinsException ex) {
60.                         System.out.println(ex.getMessage()+ " : "+ex.getException());
61.                 }
62.                 // supprimer un RV
63.                 System.out.println("Suppression du Rv ajouté");
64.                 try {
65.                         dao.supprimerRv(idRv);
66.                         display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
67.                                         .getRvMedecinJour(1, "2006:08:23"));
68.                 } catch (RdvMedecinsException ex) {
69.                         System.out.println(ex.getMessage()+ " : "+ex.getException());
70.                 }
71.         }
72.
73.         // méthode utilitaire - affiche les lignes résultats d'un SELECT
74.         private static void display(String message, List elements) {
75.                 System.out.println(message);
76.                 for (Object element : elements) {
77.                         System.out.println(element);
78.                 }
79.         }
80.
81. }
```

On notera les points suivants :

- lignes 15-16 : instanciation d'une classe implémentant l'interface [IDao]. Le fichier de configuration [spring-config-dao.xml] fournit à cette classe les informations sur le SGBD (pilote JDBC, url de la base, utilisateur de la connexion et son mot de passe). C'est pourquoi ces informations n'apparaissent nulle part dans le programme de test. Il n'a pas à les connaître. Nous n'avons pas, non plus, à les connaître pour comprendre le programme de test.
- les exceptions lancées par la couche [dao] seront de type [RdvMedecinsException]. Ici chaque appel de méthode de la couche [dao] a été entouré d'un try / catch, mais ce n'était pas obligatoire puisque [RdvMedecinsException] est un type d'exception non contrôlée.

A l'exécution, l'affichage écran obtenu est le suivant :

```
1.  ...
2.  Liste des clients :
3.  [1,1,Mr,Jules,MARTIN]
4.  [2,1,Mme,Christine,GERMAN]
5.  [3,1,Mr,Jules,JACQUARD]
6.  [4,1,Melle,Brigitte,BISTROU]
7.  Liste des médecins :
8.  [1,1,Mme,Marie,PELISSIER]
9.  [2,1,Mr,Jacques,BROMARD]
10. [3,1,Mr,Philippe,JANDOT]
11. [4,1,Melle,Justine,JACQUEMOT]
12. Liste des créneaux du médecin n° 1 :
13. [1,1,1,8:0,8:20]
14. [2,1,1,8:20,8:40]
15. [3,1,1,8:40,9:0]
16. [4,1,1,9:0,9:20]
17. [5,1,1,9:20,9:40]
18. [6,1,1,9:40,10:0]
19. [7,1,1,10:0,10:20]
20. [8,1,1,10:20,10:40]
21. [9,1,1,10:40,11:0]
22. [10,1,1,11:0,11:20]
23. [11,1,1,11:20,11:40]
24. [12,1,1,11:40,12:0]
25. [13,1,1,14:0,14:20]
26. [14,1,1,14:20,14:40]
27. [15,1,1,14:40,15:0]
28. [16,1,1,15:0,15:20]
29. [17,1,1,15:20,15:40]
30. [18,1,1,15:40,16:0]
31. [19,1,1,16:0,16:20]
32. [20,1,1,16:20,16:40]
33. [21,1,1,16:40,17:0]
34. [22,1,1,17:0,17:20]
35. [23,1,1,17:20,17:40]
36. [24,1,1,17:40,18:0]
37. Liste des Rv du médecin n° 1, le 23/08/2006 :
38. [4,23/08/2006,3,4]
39. [8,23/08/2006,1,6]
40. [9,23/08/2006,3,7]
```

```
41. [5,23/08/2006,4,20]
42. Ajout d'un Rv au médecin n° 1
43. Liste des Rv du médecin n° 1, le 23/08/2006 :
44. [95,23/08/2006,2,2]
45. [4,23/08/2006,3,4]
46. [8,23/08/2006,1,6]
47. [9,23/08/2006,3,7]
48. [5,23/08/2006,4,20]
49. Ajout d'un Rv au médecin n° 1
50. Erreur d'accès à la base de données :  : org.springframework.jdbc.UncategorizedSQLException:
    SqlMapClient operation; uncategorized SQLException for SQL []; SQL state [HY000]; error code
    [335544665];
51. --- The error occurred in rdvmedecins-firebird.xml.
52. --- The error occurred while applying a parameter map.
53. --- Check the Rv.insertOne-InlineParameterMap.
54. --- Check the statement (update failed).
55. --- Cause: org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544665. violation of PRIMARY or
    UNIQUE KEY constraint "UNQ1_RV" on table "RV"; nested exception is
    com.ibatis.common.jdbc.exception.NestedSQLException:
56. .....................................................................................
57. Suppression du Rv ajouté
58. Liste des Rv du médecin n° 1, le 23/08/2006 :
59. [4,23/08/2006,3,4]
60. [8,23/08/2006,1,6]
61. [9,23/08/2006,3,7]
62. [5,23/08/2006,4,20]
```

Nous laissons au lecteur le soin de faire le lien entre le code exécuté et les résultats obtenus.

3.3 L'application console de gestion des Rv

Nous souhaitons écrire une application console basique de gestion des Rv. C'est l'application [MainRvMedecins] présentée en [2] dans le projet Eclipse précédent, au paragraphe 3.2, page 10. Nous présentons ci-dessous des copies d'écran de son fonctionnement.

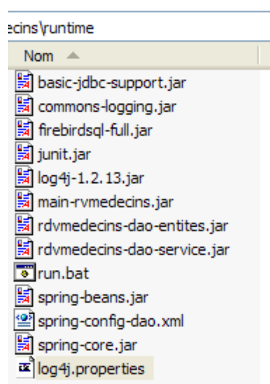
Le programme console est interactif : il accepte d'exécuter des commandes tapées au clavier. Celles-ci sont présentées au démarrage de l'application et également après chaque exécution d'une commande de l'utilisateur :

```
Commandes (* pour arrêter) :
1 : (getMedecins)
2 : (getCreneauxMedecin) idMedecin
3 : (getClients)
4 : (getRvMedecinJour) idMedecin aaaa:mm:jj
5 : (ajouterRvJourCreneauClient) aaaa:mm:jj idCreneau idClient
6 : (supprimerRv) idRv
```

Pour taper une commande, l'utilisateur tape son n° et éventuellement les informations complémentaires dont elle a besoin. L'argument entre parenthèses n'est pas à donner.

Dans la démonstration qui suit, nous supposons :

- que tous les exécutables nécessaires à l'application ont été placés dans des archives
- que celles-ci ont été rassemblées dans un dossier avec les fichiers de configuration nécessaires :



La classe [istia.st.rdvmedecins.tests.MainRvMedecins] testée a été placée ci-dessus dans l'archive [main-rvmedecins.jar]. L'application est lancée dans une fenêtre Dos, avec le fichier [run.bat] suivant :

```
1.  @ECHO OFF
2.  SET JAVA_HOME="C:\Program Files\Java\jre1.5.0_10"
3.  %JAVA_HOME%\bin\java.exe -classpath .;rdvmedecins-dao-entites.jar;rdvmedecins-dao-
    service.jar;basic-jdbc-support.jar;spring-core.jar;spring-beans.jar;log4j-
```

```
    1.2.13.jar;junit.jar;firebirdsql-full.jar;commons-logging.jar;main-rvmedecins.jar
    istia.st.rdvmedecins.tests.MainRvMedecins
```

- ligne 2 : fixe l'emplacement de la machine virtuelle Java (JVM)
- ligne 3 : lance cette JVM pour exécuter la classe [istia.st.rdvmedecins.tests.MainRvMedecins].

On lance le programme :

```
dos>run

Commandes (* pour arrêter) :
1 : (getMedecins)
2 : (getCreneauxMedecin) idMedecin
3 : (getClients)
4 : (getRvMedecinJour) idMedecin aaaa:mm:jj
5 : (ajouterRvJourCreneauClient) aaaa:mm:jj idCreneau idClient
6 : (supprimerRv) idRv
```

Dans les copies d'écran ci-dessous, **D->** désigne la **demande** de l'utilisateur, **R<-** la **réponse** que lui fait l'application. Celle-ci comprend toujours la liste des commandes ci-dessus. Par facilité, celle-ci n'est pas reproduite dans les copies d'écran ci-dessous.

On demande la liste des médecins :

D->

```
1
```

R<-

```
Liste des médecins :
[1,Mme,Marie,PELISSIER]
[2,Mr,Jacques,BROMARD]
[3,Mr,Philippe,JANDOT]
[4,Melle,Justine,JACQUEMOT]
```

On demande la liste des clients :

D->

```
3
```

R<-

```
Liste des clients :
[1,Mr,Jules,MARTIN]
[2,Mme,Christine,GERMAN]
[3,Mr,Jules,JACQUARD]
[4,Melle,Brigitte,BISTROU]
```

On demande la liste des créneaux du médecin n° 1 (Mme PELISSIER) :

D->

```
2 1
```

R<-

```
Liste des créneaux du médecin n° 1 :
[1,1,8:0,8:20]
[2,1,8:20,8:40]
[3,1,8:40,9:0]
[4,1,9:0,9:20]
[5,1,9:20,9:40]
[6,1,9:40,10:0]
[7,1,10:0,10:20]
[8,1,10:20,10:40]
[9,1,10:40,11:0]
[10,1,11:0,11:20]
[11,1,11:20,11:40]
[12,1,11:40,12:0]
[13,1,14:0,14:20]
[14,1,14:20,14:40]
[15,1,14:40,15:0]
[16,1,15:0,15:20]
[17,1,15:20,15:40]
[18,1,15:40,16:0]
[19,1,16:0,16:20]
[20,1,16:20,16:40]
[21,1,16:40,17:0]
[22,1,17:0,17:20]
[23,1,17:20,17:40]
[24,1,17:40,18:0]
```

Mme PELISSIER a 24 créneaux.

Mr BROMARD (client n° 2) téléphone pour avoir un RV avec Mme PELISSIER le 23/08/2006. On regarde les RV de Mme PELISSIER pour ce jour :

D->
```
4 1 2006:08:23
```
R<-
```
Liste des Rv du médecin n° [1], le 2006:08:23
[4,23/08/2006,3,4]
[8,23/08/2006,1,6]
[9,23/08/2006,3,7]
[6,23/08/2006,1,12]
[5,23/08/2006,4,20]
```

Après discussion, Mr BROMARD prend le créneau n° 10 (11 h – 11 h 20). On le lui réserve :

D->
```
5 2006:08:23 10 2
```
R<-
```
Rv ajouté
```

On vérifie :

D->
```
4 1 2006:08:23
```
R<-
```
Liste des Rv du médecin n° [1], le 2006:08:23
[4,23/08/2006,3,4]
[8,23/08/2006,1,6]
[9,23/08/2006,3,7]
[77,23/08/2006,2,10]
[6,23/08/2006,1,12]
[5,23/08/2006,4,20]
```

La réservation a bien été faite (n° 77 ci-dessus).

Mr MARTIN (client n° 1) qui, parce qu'il hésitait, avait fait 2 réservations pour le 23/08/2006, téléphone pour annuler celle de 11 h 40 (créneau n° 12). On supprime le RV n° 6 :

D->
```
6 6
```
R<-
```
Rv n° 6 supprimé.
```

On vérifie :

D->
```
4 1 2006:08:23
```
R<-
```
Liste des Rv du médecin n° [1], le 2006:08:23
[4,23/08/2006,3,4]
[8,23/08/2006,1,6]
[9,23/08/2006,3,7]
[77,23/08/2006,2,10]
[5,23/08/2006,4,20]
```

On arrête l'application :

D->
```
*
```

Le programme teste la validité des commandes tapées au clavier, comme le montre la nouvelle exécution suivante :

```
dos>run

Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

On tape une commande vide :

D->
```

```
R<-

```
Tapez une commande...

Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

On tape une commande inexistante :

D->

```
7
```

R<-

```
Commande non reconnue ...

Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

On tape la commande 1 avec des arguments incorrects :

D->

```
1 xx
```

R<-

```
Syntaxe incorrecte
```

```


Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

etc...

Voici les cas à considérer selon la valeur de la commande :

1. la commande 1 ne doit pas avoir de paramètres
2. la commande 2 doit avoir un paramètre de type entier
3. la commande 3 ne doit pas avoir de paramètres
4. la commande 4 doit avoir un paramètre de type entier et un paramètre de type chaîne.
5. la commande 5 doit avoir un paramètre de type chaîne et deux paramètres de type entier.
6. la commande 6 doit avoir un paramètre de type entier

Note : il n'y a pas lieu de vérifier que les dates saisies sont valides ni que les nombres entiers saisis sont dans un intervalle donné. En effet, dans ces deux cas d'erreur, le SGBD réagit correctement :

- dans le cas d'un SELECT, il ne rend aucune ligne
- dans le cas d'un INSERT dans la table RV, il ne fait pas l'insertion et lance une exception soit à cause de contraintes de clés étrangères non respectées, soit à cause d'une date invalide.
- dans le cas d'un DELETE sur la table RV, il ne fait pas la suppression parce qu'aucune ligne n'a la clé primaire recherchée

---

Question 1 : Ecrire l'application console [MainRvMedecins].

- la syntaxe des différentes commandes tapées par l'utilisateur sera vérifiée et les erreurs éventuelles signalées.
- les exceptions seront interceptées et donneront lieu à des messages d'erreur à l'écran
- on s'inspirera de l'exemple présenté au paragraphe 3.2, page 11. Notamment, l'instanciation de la couche [dao] sera faite de la même façon.

Conseils : Pour récupérer les éléments de la commande tapée au clavier par l'utilisateur, on pourra utiliser la méthode **[String].split("expression régulière")**. Voici un exemple d'utilisation de cette méthode :

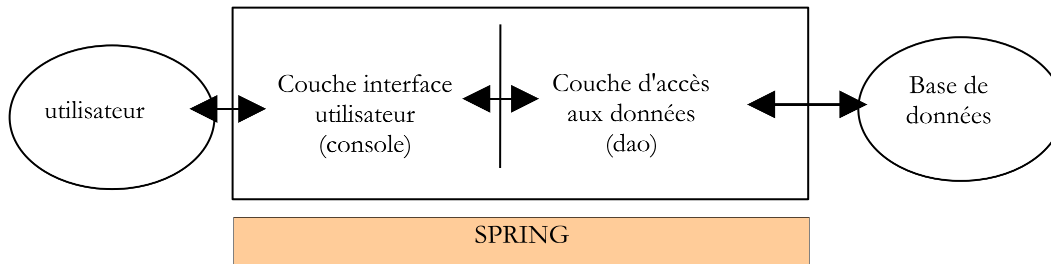
```
String ligne="champ0 champ1 champ2";
String[] champs=ligne.split("\\s+");
```

La chaîne "**\\s+**" indique que les champs de la chaîne **ligne** sont séparés par un ou plusieurs " espaces ". Après les instructions précédentes, **champs** sera un tableau de 3 objets de type String avec :

```
champ[0]="champ0"
champ[1]="champ1"
champ[2]="champ2"
```

4 Implémentation de la couche [dao]

Nous nous proposons d'étudier maintenant l'implémentation de l'interface [IDao] de la couche [dao] :



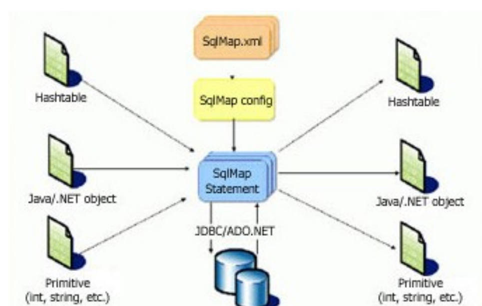
L'interface Java [IDao] de la couche [dao] a été présentée au paragraphe 3.1, page 10. Le projet Eclipse de son implémentation pourrait être le suivant :

- les objets de l'application, décrits au paragraphe 2.2, page 4, sont encapsulés dans l'archive [rdvmedecins-dao-entites.jar] (cf [1])
- [2] montre le fichier source [DaoIbatis] de l'implémentation particulière de l'interface [IDao] que nous allons écrire.

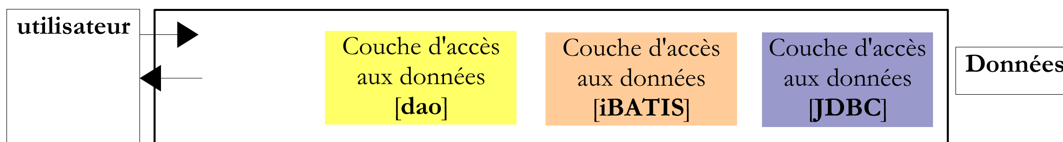
4.1 Les fichiers de configuration de la couche [dao]

Note : les pages 17 à 21 rappellent des notions vues en cours. Elles sont là par souci pédagogique. Il est possible de passer directement à la page 21 si ces informations vous sont déjà connues.

Nous nous proposons d'implémenter la couche [dao] à l'aide des classes et interfaces de l'outil iBatis/SqlMap [http://ibatis.apache.org/] :



L'architecture de l'accès aux données est alors la suivante :



Spring [http://www.springframework.org] offre un certain nombre de classes et interfaces facilitant l'usage d'iBatis. Parmi celles-ci, la classe abstraite [SqlMapClientDaoSupport] encapsule la partie générique de l'utilisation du framework [iBATIS], c.a.d. des parties de code qu'on retrouve dans toutes les couche [dao] utilisant l'outil [iBATIS]. Pour écrire la partie non générique du code, c'est à dire ce qui est spécifique à la couche [dao] que l'on écrit, il suffit de dériver la classe [SqlMapClientDaoSupport]. C'est ce que nous ferons ici.

La classe [SqlMapClientDaoSupport] est définie comme suit :

org.springframework.orm.ibatis.support

Class SqlMapClientDaoSupport

```
java.lang.Object
  └ org.springframework.dao.support.DaoSupport
      └ org.springframework.orm.ibatis.support.SqlMapClientDaoSupport
```

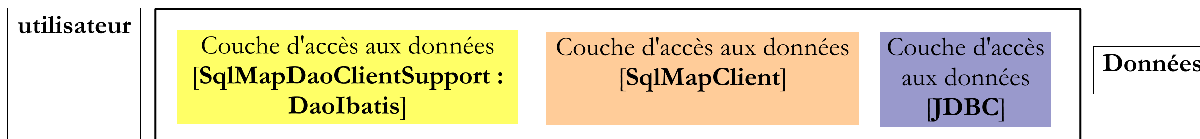
All Implemented Interfaces:

InitializingBean

Parmi les méthodes de cette classe, l'une d'elles permet de configurer le client [iBATIS] avec lequel on va exploiter la base de données :

void	`setSqlMapClient(com.ibatis.sqlmap.client.SqlMapClient sqlMapClient)` Set the iBATIS Database Layer SqlMapClient to work with.

L'objet [SqlMapClient sqlMapClient] est l'objet [IBATIS] utilisé pour accéder à une base de données. A lui tout seul, il implémente la couche [iBATIS] de notre architecture :



Une séquence typique d'actions avec l'objet [SqlMapClient] est la suivante :

1. demander une connexion à un pool de connexions
2. ouvrir une transaction
3. exécuter une série d'ordres SQL mémorisée dans un fichier de configuration
4. fermer la transaction
5. rendre la connexion au pool

Si notre implémentation [DaoIbatis] travaillait directement avec [iBATIS], elle devrait faire cette séquence de façon répétée. Seule l'opération 3 est spécifique à une couche [dao], les autres opérations étant génériques. La classe Spring [SqlMapClientDaoSupport] assurera elle-même les opérations 1, 2, 4 et 5, déléguant l'opération 3 à sa classe dérivée, ici la classe [DaoIbatis].

Pour pouvoir fonctionner, la classe [SqlMapClientDaoSupport] a besoin d'une référence sur l'objet iBATIS [SqlMapClient sqlMapClient] qui va assurer le dialogue avec la base de données. Cet objet a besoin de deux choses pour fonctionner :

- un objet [DataSource] connecté à la base de données auprès duquel il va demander des connexions
- un (ou des) fichier de configuration où sont externalisés les ordres SQL à exécuter. En effet, ceux-ci ne sont pas dans le code Java. Ils sont identifiés par un code dans un fichier de configuration et l'objet [SqlMapClient sqlMapClient] utilise ce code pour faire exécuter un ordre SQL particulier.

Un embryon de configuration de notre couche [dao] qui reflèterait l'architecture ci-dessus serait le suivant :

```
1.         <!-- la classes d'accè à la couche [dao] -->
2.         <bean id="dao" class="istia.st.rdvmedecins.dao.service.DaoIbatis">
3.                 <property name="sqlMapClient" ref="sqlMapClient" />
4.     </bean>
```

Ici la propriété [sqlMapClient] (ligne 3) de la classe [DaoIbatis] (ligne 2) est initialisée. Elle l'est par la méthode [setSqlMapClient] de la classe [DaoIbatis]. Cette classe n'a pas cette méthode. C'est sa classe parent [SqlMapClientDaoSupport] qui l'a. C'est donc elle qui est en réalité initialisée ici.

Maintenant ligne 3, on fait référence à un objet nommé " sqlMapClient " qui reste à construire. Celui-ci, on l'a dit, est de type [SqlMapClient], un type [iBATIS] :

com.ibatis.sqlmap.client

Interface SqlMapClient

[SqlMapClient] est une interface. Spring offre la classe [SqlMapClientFactoryBean] pour obtenir un objet implémentant cette interface :

org.springframework.orm.ibatis

Class SqlMapClientFactoryBean

```
java.lang.Object
  └org.springframework.orm.ibatis.SqlMapClientFactoryBean
```

All Implemented Interfaces:
FactoryBean, InitializingBean

Rappelons que nous cherchons à instancier un objet implémentant l'interface [SqlMapClient]. Ce n'est apparemment pas le cas de la classe [SqlMapClientFactoryBean]. Celle-ci implémente l'interface [FactoryBean] (cf ci-dessus). Celle-ci a la méthode [getObject()] suivante :

Object	getObject() Return an instance (possibly shared or independent) of the object managed by this factory.

Lorsqu'on demande à Spring une instance d'un objet implémentant l'interface [FactoryBean], il :

- crée une instance [I] de la classe - ici il crée une instance de type [SqlMapClientFactoryBean].
- rend à la méthode appelante, le résultat de la méthode [I].getObject() - la méthode [SqlMapClientFactoryBean].getObject()] va rendre ici un objet implémentant l'interface [SqlMapClient].

Pour pouvoir rendre un objet implémentant l'interface [SqlMapClient], la classe [SqlMapClientFactoryBean] a besoin de deux informations nécessaires à cet objet :

- un objet [DataSource] connecté à la base de données auprès duquel il va demander des connexions
- un (ou des) fichier de configuration où sont externalisés les ordres SQL à exécuter

La classe [SqlMapClientFactoryBean] possède les méthodes **set** pour initialiser ces deux propriétés :

void	setConfigLocation(Resource configLocation) Set the location of the iBATIS SqlMapClient config file.
void	setDataSource(DataSource dataSource) Set the DataSource to be used by iBATIS SQL Maps.

Notre fichier de configuration se précise et devient :

```
1.    <!-- SqlMapCllient -->
2.    <bean id="sqlMapClient"
3.       class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
4.       <property name="dataSource" ref="dataSource" />
5.       <property name="configLocation"
6.          value="classpath:ibatis-config-firebird.xml" />
7.    </bean>
8.    <!-- la classes d'accè à la couche [dao] -->
9.    <bean id="dao" class="istia.st.rdvmedecins.dao.service.DaoIbatis">
10.      <property name="sqlMapClient" ref="sqlMapClient" />
```

```
11. </bean>
```

- lignes 2-3 : le bean " sqlMapClient " est de type [SqlMapClientFactoryBean]. De ce qui vient d'être expliqué, nous savons que lorsque nous demandons à Spring une instance de ce bean, nous obtenons un objet implémentant l'interface iBATIS [SqlMapClient]. C'est ce dernier objet qui sera donc obtenu en ligne 10.
- lignes 7-9 : nous indiquons que le fichier de configuration nécessaire à l'objet iBATIS [SqlMapClient] s'appelle " ibatis-config-firebird.xml " et qu'il doit être cherché dans le *ClassPath* de l'application. La méthode [SqlMapClientFactoryBean].setConfigLocation est ici utilisée.
- lignes 4-6 : nous initialisons la propriété [dataSource] de [SqlMapClientFactoryBean] avec sa méthode [setDataSource].

Ligne 4, nous faisons référence à un bean appelé " dataSource " qui reste à construire. Si on regarde le paramètre attendu par la méthode [setDataSource] de [SqlMapClientFactoryBean], on voit qu'il est de type [DataSource] :

javax.sql
Interface DataSource

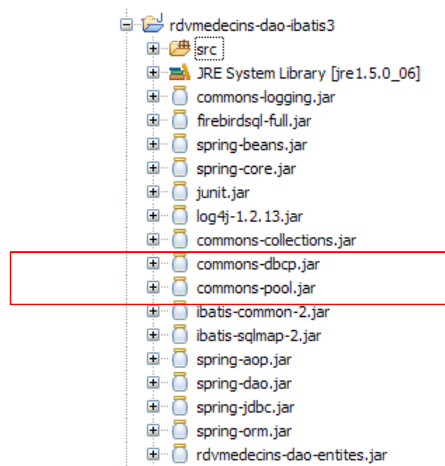
On a de nouveau affaire à une interface dont il nous faut trouver une classe d'implémentation. Le rôle d'une telle classe est de fournir à une application, de façon efficace, des connexions à une base de données particulière. Un SGBD ne peut maintenir ouvertes simultanément un grand nombre de connexions. Pour diminuer le nombre de connexions ouvertes à un moment donné, on est amenés, pour chaque échange avec la base, à :

- ouvrir une connexion
- commencer une transaction
- émettre des ordres SQL
- fermer la transaction
- fermer la connexion

Ouvrir et fermer des connexions de façon répétée est coûteux en temps. Pour résoudre ces deux problèmes (limiter à la fois le nombre de connexions ouvertes à un moment donné, et limiter le coût d'ouverture / fermeture de celles-ci, les classes implémentant l'interface [DataSource] procèdent souvent de la façon suivante :

- elles ouvrent dès leur instanciation, N connexions avec la base de données visée. N a en général une valeur par défaut et peut le plus souvent être défini dans un fichier de configuration. Ces N connexions vont rester tout le temps ouvertes et forment un pool de connexions disponibles pour les threads de l'application.
- lorsqu'un thread de l'application demande une ouverture de connexion, l'objet [DataSource] lui donne l'une des N connexions ouvertes au démarrage, s'il en reste de disponibles. Lorsque l'application ferme la connexion, cette dernière n'est en réalité pas fermée mais simplement remise dans le pool des connexions disponibles.

Il existe diverses implémentations de l'interface [DataSource] disponibles librement. Nous allons utiliser ici l'implémentation [commons DBCP] disponible à l'url [http://jakarta.apache.org/commons/dbcp/]. L'utilisation de l'outil [commons DBCP] nécessite deux archives [commons-dbcp, commons-pool] qui ont été toutes deux placées dans le *Classpath* du projet :



La classe [BasicDataSource] de [commons DBCP] fournit l'implémentation [DataSource] dont nous avons besoin :

org.apache.commons.dbcp

Class BasicDataSource

java.lang.Object
└org.apache.commons.dbcp.BasicDataSource

All Implemented Interfaces:

DataSource

Cette classe va nous fournir un pool de connexions pour accéder à la base Firebird [rdvmedecins.gdb] de notre application. Pour cela, il faut lui donner les informations dont elle a besoin pour créer les connexions du pool :

1. le nom du pilote JDBC à utiliser – initialisé avec [setDriverClassName]
2. le nom de l'url de la base de données à exploiter - initialisé avec [setUrl]
3. l'identifiant de l'utilisateur propriétaire de la connexion – initialisé avec [setUser**n**ame] (et non pas setUser**N**ame comme on aurait pu s'y attendre)
4. son mot de passe - initialisé avec [setPassword]

Le fichier de configuration [spring-config-dao.xml] de notre couche [dao] pourra 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 source de donnéees DBCP -->
5.   <bean id="dataSource"
6.      class="org.apache.commons.dbcp.BasicDataSource"
7.      destroy-method="close">
8.      <property name="driverClassName"
9.         value="org.firebirdsql.jdbc.FBDriver" />
10.     <property name="url"
11.        value="jdbc:firebirdsql:localhost/3050:C:\...\rdvmedecins2.gdb" />
12.     <property name="username" value="sysdba" />
13.     <property name="password" value="masterkey" />
14.  </bean>
15.  <!-- SqlMapCllient -->
16.  <bean id="sqlMapClient"
17.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
18.     <property name="dataSource" ref="dataSource" />
19.     <property name="configLocation"
20.        value="classpath:ibatis-config-firebird.xml" />
21.  </bean>
22.  <!-- la classes d'accè à la couche [dao] -->
23.  <bean id="dao" class="istia.st.rdvmedecins.dao.service.DaoIbatis">
24.     <property name="sqlMapClient" ref="sqlMapClient" />
25.  </bean>
26. </beans>
```

La ligne 20 référence le fichier [ibatis-config-firebird.xml] qui configure le client [SqlMapClient] d'iBATIS. Ce fichier est le suivant :

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

- ce fichier doit avoir <sqlMapConfig> comme balise racine (lignes 6 et 8)
- ligne 7 : la balise <sqlMap> sert à désigner les fichiers qui contiennent les ordres SQL à exécuter.

Le fichier [rdvmedecins-firebird.xml] décrit les ordres SQL qui vont être émis sur les tables de la base de données Firebird [RDVMEDECINS]. Revenons sur l'interface [IDao] à implémenter :

```
1.  package istia.st.rdvmedecins.dao.service;
2.
3.  import istia.st.rdvmedecins.dao.entites.RawClient;
4.  import istia.st.rdvmedecins.dao.entites.RawCreneau;
5.  import istia.st.rdvmedecins.dao.entites.RawMedecin;
6.  import istia.st.rdvmedecins.dao.entites.RawRv;
7.
8.  import java.util.List;
9.
10. public interface IDao {
11.
12.        // liste des clients
13.        public List<RawClient> getAllClients();
```

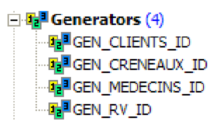
```
14.         // liste des Médecins
15.         public List<RawMedecin> getAllMedecins();
16.         // liste des créneaux horaires d'un médecin
17.         public List<RawCreneau> getAllCreneaux(int idMedecin);
18.         // liste des Rv d'un médecin, un jour donné
19.         public List<RawRv> getRvMedecinJour(int idMedecin, String jour);
20.         // ajouter un RV
21.         public int ajouterRv(String jour, int idCreneau, int idClient);
22.         // supprimer un RV
23.         public int supprimerRv(int idRv);
24.
25. }
```

Chacune des méthodes de l'interface fait un accès à la base de données [RDVMEDECINS].

Question 2 : indiquez pour chacune des méthodes de l'interface [IDao] le ou les ordres SQL qui devront être émis sur la base.

Conseils :

La méthode [ajouterRv] insère une ligne dans la table RV de la base. Cette table a un générateur de valeurs primaires qui peut être utilisé pour obtenir une valeur pour le champ ID de la ligne à insérer. L'exemple suivant montre comment procéder :

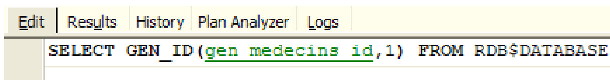


La base a 4 générateurs de valeurs entières, 1 par table.

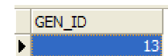
Generators | Dependencies | D

Name	Value
GEN_MEDECINS_ID	12

Celui de la table [MEDECINS] s'appelle GEN_MEDECINS_ID et a pour l'instant la valeur 12.



On demande la valeur suivante du générateur



On obtient 13.

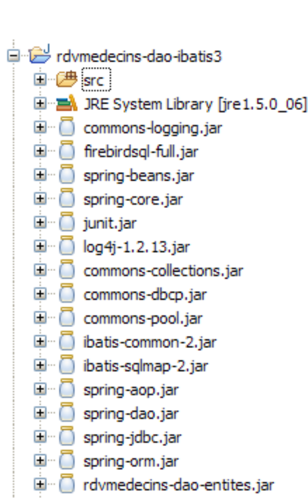
Pour la table [RV], on utilisera le générateur [GEN_RV_ID].

Question 3 : écrire le fichier de configuration [rdvmedecins-firebird.xml]. On s'inspirera des exemples traités en cours.

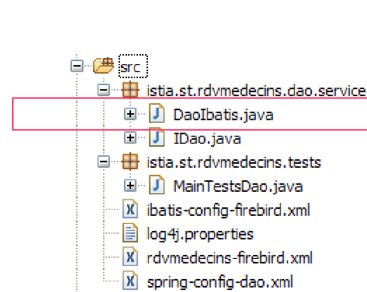
4.2 Implémentation 1 de l'interface [IDao]

Redonnons une vue d'ensemble du projet Eclipse de l'implémentation de l'interface [IDao] :

1



2



En [2], la classe [DaoIbatis] implémentant l'interface [IDao] pourrait être la suivante :

```
1.  package istia.st.rdvmedecins.dao.service;
2.  
3.  import istia.st.rdvmedecins.dao.entites.RawClient;
4.  import istia.st.rdvmedecins.dao.entites.RawCreneau;
5.  import istia.st.rdvmedecins.dao.entites.RawMedecin;
6.  import istia.st.rdvmedecins.dao.entites.RawRv;
7.  import istia.st.rdvmedecins.dao.entites.RdvMedecinsException;
8.  
9.  import java.text.ParseException;
10. import java.text.SimpleDateFormat;
11. import java.util.Hashtable;
12. import java.util.List;
13. 
14. import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;
15. 
16. public class DaoIbatis extends SqlMapClientDaoSupport implements IDao {
17. 
18.        // ajout d'un rv
19.        public int ajouterRv(String jour, int idCreneau, int idClient) {
20. ...
21.        }
22. 
23.        // liste des clients
24.        @SuppressWarnings("unchecked")
25.        public List<RawClient> getAllClients() {
26. ...
27.        }
28. 
29.        // liste des créneaux d'un médecin
30.        @SuppressWarnings("unchecked")
31.        public List<RawCreneau> getAllCreneaux(int idMedecin) {
32. ...
33.        }
34. 
35.        // liste des médecins
36.        @SuppressWarnings("unchecked")
37.        public List<RawMedecin> getAllMedecins() {
38. ...
39.        }
40. 
41.        // liste des Rv pour un médecin et un jour donnés
42.        @SuppressWarnings("unchecked")
43.        public List<RawRv> getRvMedecinJour(int idMedecin, String jour) {
44. ...
45.        }
46. 
47.        // suppression d'un Rv
48.        public int supprimerRv(int idRv) {
49. ...
50.        }
51. 
52. }
```

Grâce à l'usage de la classe [SqlMapClientDaoSupport] (ligne 16), la totalité de la classe ne dépasse pas 70 lignes. Il ne manque donc, ci-dessus, qu'une vingtaine de lignes de code. La classe sera instanciée par le fichier [spring-config-dao.xml] décrit au paragraphe 4.1, page 21.

Question 4 : écrire la classe [DaoIbatis] en faisant l'hypothèse que chaque méthode ayant besoin de données présentes dans la base fait un accès à cette base.

Conseils : Un programme utilisant la classe [DaoIbatis] a été décrit au paragraphe 3.2, page 11. On pourra y revenir si nécessaire pour éclaircir le rôle de cette classe.

4.3 Implémentation 2 de l'interface [IDao]

Certaines des données de la base sont en lecture seule : la liste des médecins, la liste des clients, la liste des tous les créneaux horaires des différents médecins. D'un point de vue performances, il serait intéressant de mémoriser ces données en mémoire lors de l'instanciation de la couche [dao] puis ensuite de servir ces données aux clients depuis la mémoire afin d'éviter des accès, toujours plus lents, à la base de données.

On souhaite paramétrer ce fonctionnement à l'aide du fichier [spring-config-dao.xml] suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
```

```
4.   <!-- la source de donnéees DBCP -->
5.   <bean id="dataSource"
6.      class="org.apache.commons.dbcp.BasicDataSource"
7.      destroy-method="close">
8.      <property name="driverClassName"
9.         value="org.firebirdsql.jdbc.FBDriver" />
10.     <property name="url"
11.        value="jdbc:firebirdsql:localhost/3050:C:\data\2006-
   2007\databases\firebird\rvMedecins\rdvmedecins2.gdb" />
12.     <property name="username" value="sysdba" />
13.     <property name="password" value="masterkey" />
14.  </bean>
15.  <!-- SqlMapCllient -->
16.  <bean id="sqlMapClient"
17.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
18.     <property name="dataSource" ref="dataSource" />
19.     <property name="configLocation"
20.        value="classpath:ibatis-config-firebird.xml" />
21.  </bean>
22.  <!-- la classes d'accè à la couche [dao] -->
23.  <bean id="dao" class="istia.st.rdvmedecins.dao.service.DaoIbatis" init-method="init">
24.     <property name="sqlMapClient" ref="sqlMapClient" />
25.     <property name="clientsEnCache" value="true" />
26.     <property name="medecinsEnCache" value="true" />
27.     <property name="creneauxEnCache" value="true" />
28.  </bean>
29. </beans>
```

- ligne 25 : indique si les clients doivent être mis ou non en cache
- ligne 26 : idem pour les médecins
- ligne 27 : idem pour les créneaux
- ligne 23 : l'attribut " init-method " indique la méthode de la classe [DaoIbatis] à exécuter, une fois la classe instanciée et ses champs [clientsEnCache, medecinsEnCache, creneauxEnCache] initialisés.

L'administrateur de l'application peut ainsi se livrer à des réglages afin de rechercher la configuration la plus performante. Il s'agit de trouver un équilibre entre deux contraintes : occupation mémoire des données mises en cache et délai de réponse au client.

Avec cette nouvelle configuration, la classe [DaoIbatis] devient la suivante :

```
1.  package istia.st.rdvmedecins.dao.service;
2.
3.  ...
4.
5.  public class DaoIbatis extends SqlMapClientDaoSupport implements IDao {
6.
7.    // cache des données en lecture seule de la BD
8.    private List<RawClient> clients = null;
9.    private Hashtable<Integer, List<RawCreneau>> creneaux = new Hashtable<Integer, List<RawCreneau>>();
10.   private List<RawMedecin> medecins = null;
11.
12.   // configuration de la couche [dao]
13.   private boolean clientsEnCache = false;
14.   private boolean creneauxEnCache = false
15.   private boolean medecinsEnCache = false;
16.
17.   // constructeur
18.   public DaoIbatis() {
19.
20.   }
21.
22.   // init service
23.   @SuppressWarnings("unchecked")
24.   public void init() {
25. ...
26.   }
27.
28.   // ajout d'un rv
29.   public int ajouterRv(String jour, int idCreneau, int idClient) {
30. ...
31.   }
32.
33.   // liste des clients
34.   @SuppressWarnings("unchecked")
35. ...
36.   }
37.
38.   // liste des créneaux d'un médecin
39.   @SuppressWarnings("unchecked")
40.   public List<RawCreneau> getAllCreneaux(int idMedecin) {
41.      // créneaux en cache ?
42.      if (creneaux == null)
43.         return getSqlMapClientTemplate().queryForList("CreneauxForMedecin",
44.                 idMedecin);
45.      else {
46.         return creneaux.get(idMedecin);
47.      }
48.   }
49.
50.   // liste des médecins
```

```
51.   @SuppressWarnings("unchecked")
52. ...
53.   }
54.
55.   // liste des Rv pour un médecin et un jour donnés
56.   @SuppressWarnings("unchecked")
57.   public List<RawRv> getRvMedecinJour(int idMedecin, String jour) {
58. ...
59.   }
60.
61.   // suppression d'un Rv
62.   public int supprimerRv(int idRv) {
63. ...
64.   }
65.
66.   // getters - setters (clientsEnCache, medecinsEnCache, creneauxEnCache)
67. ...
68. }
```

- lignes 13-15 : les champs initialisés par le fichier de configuration Spring
- ligne 8 : la liste des clients mis en cache
- ligne 10 : la liste des médecins mis en cache
- ligne 9 : la liste des créneaux horaires mis en cache dans un dictionnaire indexé par les n°s des médecins. Ainsi l'instruction [creneaux.get(4)] donne un objet de type List<RawCreneau> qui est la liste des créneaux horaires du médecin n° 4.

Question 5 : écrire la nouvelle classe [DaoIbatis]. On prêtera attention au fait que :
– selon la configuration faite par l'administrateur de l'application, certaines données ont pu être mises en cache et d'autres pas,
– le fichier [rdvmedecins-firebird.xml] qui configure l'outil iBatis a pu se voir ajouter un nouvel ordre SQL.

Table des matières

1LE PROBLÈME....1
2LES ÉLÉMENTS DE L'APPLICATION....1
2.1LA BASE DE DONNÉES....2
2.1.1LA TABLE [MEDECINS]....2
2.1.2LA TABLE [CLIENTS]....2
2.1.3LA TABLE [CRENEAUX]....3
2.1.4LA TABLE [RV]....3
2.2LES OBJETS DE L'APPLICATION....4
2.2.1CLASSE [RAWPERSONNE]....4
2.2.2CLASSE [RAWMEDECIN]....5
2.2.3CLASSE [RAWCLIENT]....5
2.2.4CLASSE [RAWCRENEAU]....5
2.2.5CLASSE [RAWRV]....6
2.2.6CLASSE [RDVMEDECINSEXCEPTION]....7
3APPLICATION CONSOLE DE GESTION DES RV....9
3.1L'INTERFACE DE LA COUCHE [DAO]....9
3.2APPLICATION DE TEST....10
3.3L'APPLICATION CONSOLE DE GESTION DES RV....13
4IMPLÉMENTATION DE LA COUCHE [DAO]....17
4.1LES FICHIERS DE CONFIGURATION DE LA COUCHE [DAO]....17
4.2IMPLÉMENTATION 1 DE L'INTERFACE [IDAO]....22
4.3IMPLÉMENTATION 2 DE L'INTERFACE [IDAO]....23