Réaliser en 2024/2025

DOCUMENTATION PERSONNEL

ioahn
RAMIANDRASOA

Table des matières

Présentation du projet

La Maison des Ligues de Lorraine (M2L) met en place une application permettant de gérer les employés des différentes ligues sportives. Actuellement en ligne de commande et mono-utilisateur, cette application doit être améliorée pour :

- **Devenir multi-utilisateurs** grâce à l'intégration d'une base de données.
- **Mettre en place des niveaux d'habilitation** pour assurer la gestion sécurisée des employés et des administrateurs.
- **Respecter l'architecture 3-tiers**, garantissant modularité et évolutivité.

L'application est conçue selon une architecture **3-tiers** :

1. **Tier Présentation** : Interface en ligne de commande (CLI) et application bureau pour l'administration.
2. **Tier Métier** : Gestion des règles métier (habilitation, droits d'accès, gestion des employés).
3. **Tier Données** : Base de données stockant les informations des employés et des administrateurs.

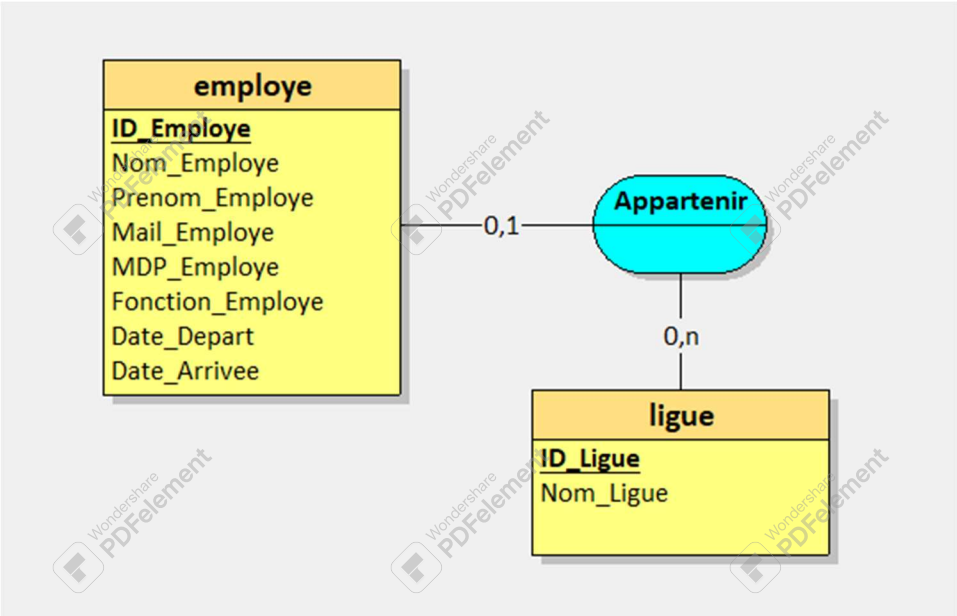
Les rôles et habilitations :

 Employé de ligue : Accès en lecture à l'annuaire des employés de sa ligue.

 Administrateur de ligue : Gestion des employés de sa ligue via une application bureau.

 Super-administrateur : Gestion globale des employés et des comptes administrateurs via la ligne de commande.

Base de données



Ce Modèle Conceptuel de Données (MCD) représente la structure d'une base de données destinée à gérer des employés et des ligues, ainsi que leur relation. Il est composé d'entités, d'une association et de cardinalités qui définissent les liens entre ces éléments.

Il comporte deux entités principales : **Employé** (avec des informations comme l'ID, le nom, le mail, la fonction, etc.) et **Ligue** (avec un ID et un nom).

L'association **"Appartenir"** relie ces deux entités avec des cardinalités :

- Un **employé** peut être rattaché à **zéro ou une seule** ligue *(0,1)*.
- Une **ligue** peut regrouper **zéro ou plusieurs** employés *(0,n)*.

Ce modèle montre qu'un employé peut exister sans être affecté à une ligue et qu'une ligue peut exister sans employés.

```sql
CREATE TABLE `ligue` (
    `ID_Ligue` int NOT NULL AUTO_INCREMENT,
    `Nom_Ligue` varchar(50) DEFAULT NULL,
    PRIMARY KEY (`ID_Ligue`)
);

CREATE TABLE `employe` (
    `ID_Employe` int NOT NULL AUTO_INCREMENT,
    `Nom_Employe` varchar(20) DEFAULT NULL,
    `Prenom_Employe` varchar(20) DEFAULT NULL,
    `Mail_Employe` varchar(50) DEFAULT NULL,
    `MDP_Employe` varchar(50) DEFAULT NULL,
    `Fonction_Employe` varchar(50) DEFAULT NULL,
    `Date_Depart` date DEFAULT NULL,
    `Date_Arrivee` date DEFAULT NULL,
    `ID_Ligue` int DEFAULT NULL,
    PRIMARY KEY (`ID_Employe`),
    KEY `fk_ligue` (`ID_Ligue`),
    CONSTRAINT `fk_ligue` FOREIGN KEY (`ID_Ligue`) REFERENCES `ligue` (`ID_Ligue`) ON DELETE CASCADE
);
```

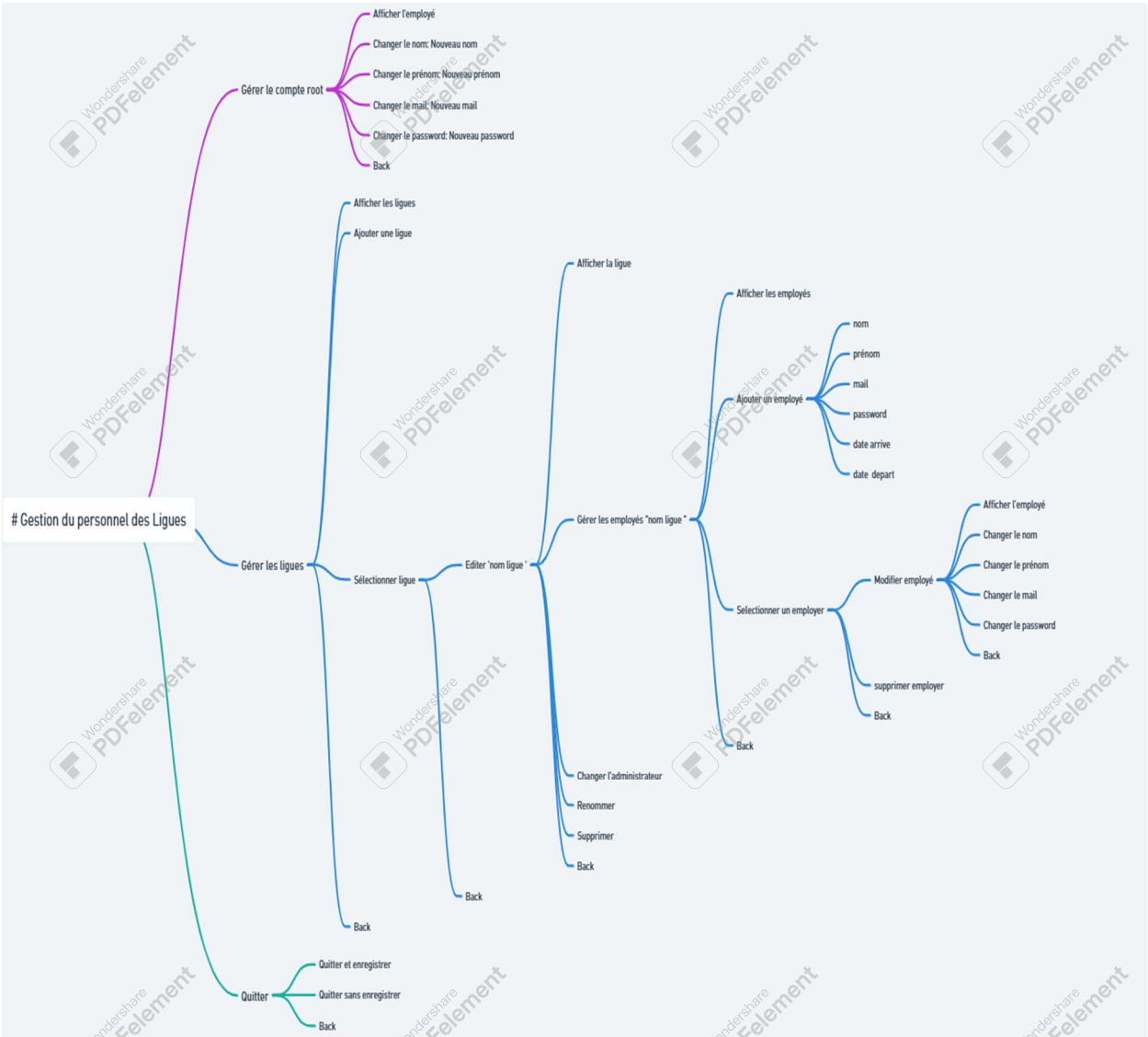
Ce script SQL permet de créer et structurer une base de données pour gérer des employés et des ligues, avec une relation entre eux. Il définit deux tables : **ligue** et **employe**, ainsi qu'une contrainte d'intégrité référentielle.

Ce script crée deux tables, **ligue** et **employe**, avec une relation entre elles.

- **Table ligue** :
    - ID_Ligue (*clé primaire, auto-incrément*) : Identifiant unique.
    - Nom_Ligue : Nom de la ligue (*varchar(50)*).
- **Table employe** :
    - ID_Employe (*clé primaire, auto-incrément*) : Identifiant unique.
    - Nom_Employe, Prenom_Employe, Mail_Employe, MDP_Employe, Fonction_Employe : Informations sur l'employé.
    - Date_Arrivee, Date_Depart : Dates d'arrivée et de départ.
    - ID_Ligue (*clé étrangère*) : Associe l'employé à une ligue.
- **Relation** : ID_Ligue est une **clé étrangère** référencée dans employe, avec **suppression en cascade** (ON DELETE CASCADE). Cela signifie que si une ligue est supprimée, les employés qui y sont associés le seront aussi.

Arbre heuristique

Voici l’arbre heuristique de l’application. Il permet à l'administrateur d'avoir une vue d'ensemble sur la gestion des employés et des ligues. Par exemple, il aide à savoir rapidement quels employés appartiennent à quelle ligue, qui a quels droits, et quelles actions sont possibles selon le rôle de l'utilisateur (employé, administrateur, super-administrateur). Cela permet à l'administrateur de prendre des décisions claires sur qui peut voir, modifier ou gérer quelles informations, tout en gardant une organisation logique et fluide.



📌 1. Architecture Générale

Le projet est organisé en **trois principaux packages**, chacun ayant un rôle bien défini :

1️⃣ commandLine - Interface en Ligne de Commande (CLI)

- Fournit une interface textuelle interactive pour gérer les ligues et les employés.
- S'appuie sur commandLineMenus pour structurer les menus et les options utilisateur.
- Gère la navigation entre différentes fonctionnalités (ajout, modification, suppression).
- Classes principales :
    - **PersonnelConsole** : Menu principal et point d'entrée du programme.
    - **LigueConsole** : Gère les interactions liées aux ligues.
    - **EmployeConsole** : Permet la gestion des employés.

2️⃣ personnel - Modèle Métier (Gestion des Données)

- Contient toutes les classes représentant les entités métier (Ligue, Employe, etc.).
- Définit les **exceptions personnalisées** (ExceptionD, DroitsInsuffisants, ImpossibleDeSupprimerRoot).
- Implémente un système de **gestion centralisée** via GestionPersonnel.
- Interface Passerelle pour la persistance des données.
- Classes principales :
    - **Employe** : Représente un employé, avec gestion des dates et de l'administration.
    - **Ligue** : Structure principale regroupant les employés sous une même entité.
    - **GestionPersonnel** : Singleton permettant l'accès centralisé aux données.

3️⃣ jdbc et serialisation - Gestion de la Persistance

- Le projet supporte **deux modes de stockage** :
    - **Base de données MySQL** (JDBC) → Requêtes SQL pour stocker les employés et ligues.
    - **Fichier de sérialisation Java** (Serialization) → Sauvegarde locale dans un fichier .srz.
- GestionPersonnel sélectionne l'implémentation à utiliser via Passerelle

📌 2. Développement et Implémentation des Fonctionnalités

2.1 Gestion des Dates dans le Modèle Métier (*Personnel 1*)

- Ajout de la Gestion des Dates

- Introduction des attributs dateArrivee et dateDepart pour les employés.
- Mise à jour du **Modèle Conceptuel de Données (MCD)** en conséquence.
- Ajout des variables d'instance LocalDate dateArrivee et LocalDate dateDepart dans Employe.
- Modification du constructeur, ajout des **getters et setters**.

```
private LocalDate dateArrivee;
private LocalDate dateDepart;
```

```
public LocalDate getDateArrivee() {
    return dateArrivee;
}

public LocalDate getDateDepart() {
    return dateDepart;
}
```

```java
public void setDateArrivee(LocalDate dateArrivee) throws SauvegardeImpossible
        {
            this.dateArrivee = dateArrivee;
            gestionPersonnel.update(this);
        }

 public void setDateDepart(LocalDate dateDepart) throws SauvegardeImpossible
        {
            this.dateDepart = dateDepart;
            gestionPersonnel.update(this);
        }
```

- Gestion des Erreurs et Tests

- Déclenchement d'une exception DateIncoherenteException si dateDepart est antérieure à dateArrivee.
- Ajout de **tests unitaires** pour :
    - Vérifier la cohérence des dates.
    - Tester les setters et leur comportement.
    - Vérifier l'impact de la suppression ou modification d'un administrateur.

```java
@Test
void testInvalidDates() throws SauvegardeImpossible {
    Ligue ligue = gestionPersonnel.addLigue("Fléchettes");
    Exception exception = assertThrows(ExceptionD.class, () ->
        ligue.addEmploye("Bouchard", "Gérard", "g.bouchard@gmail.com", "azerty",
        LocalDate.of(2023, 12, 31), LocalDate.of(2023, 1, 1))
    );
    assertEquals("La date de départ ne peut pas être avant la date d'arrivée.", exception.getMessage());
}
```

```java
@Test
void testValidDates() throws SauvegardeImpossible {
    Ligue ligue = gestionPersonnel.addLigue("Fléchettes");
    assertDoesNotThrow(() -> ligue.addEmploye("Wassim", "El Arche", "wsmsevran", "mdp",
        LocalDate.of(2023, 1, 1), LocalDate.of(2023, 12, 31)));
}
```

```java
@Test
void testNullDates() throws SauvegardeImpossible {
    Ligue ligue = gestionPersonnel.addLigue("Football");

    // Test avec date de départ nulle
    Exception exception = assertThrows(ExceptionD.class, () ->
        ligue.addEmploye("Bouchard", "Gérard", "g.bouchard@gmail.com", "azerty", null,
        LocalDate.of(2023, 1, 1))
    );
    assertEquals("La date de départ ne peut pas être avant la date d'arrivée.", exception.getMessage());

    // Test avec date d'arrivée nulle
    exception = assertThrows(ExceptionD.class, () ->
        ligue.addEmploye("Bouchard", "Gérard", "g.bouchard@gmail.com", "azerty",
        LocalDate.of(2023, 1, 1), null)
    );
    assertEquals("La date de départ ne peut pas être avant la date d'arrivée.", exception.getMessage());
}
```

```java
@Test
void testSetDateInvalid() throws SauvegardeImpossible, ExceptionD {
    Ligue ligue = gestionPersonnel.addLigue("Fléchettes");
    Employe employe = ligue.addEmploye("Bouchard", "Gérard", "g.bouchard@gmail.com", "azerty",
        LocalDate.of(2023, 1, 1), LocalDate.of(2023, 12, 31));

    // Test date départ invalide
    Exception exception = assertThrows(ExceptionD.class, () ->
        employe.setDateDepart(LocalDate.of(2022, 1, 1))
    );
    assertEquals("La date de départ ne peut pas être avant la date d'arrivée.", exception.getMessage());

    // Test date arrivée invalide
    exception = assertThrows(ExceptionD.class, () ->
        employe.setDateArrivee(LocalDate.of(2024, 12, 2))
    );
    assertEquals("La date de départ ne peut pas être avant la date d'arrivée.", exception.getMessage());
}
```

2.2 Améliorations de l'Interface en Ligne de Commande (*Personnel 2*)

-Sélection des Employés Avant Modification ou Suppression

- Ajout d'une étape où l'utilisateur doit sélectionner un employé avant d'effectuer une action.
- Menu interactif affichant la liste des employés disponibles dans une ligue.

  Dans ligueConsole.java

```java
private Option selectEmploye(Ligue ligue)
{
    return new List<>("Selectionner un employe", "e",
            () -> new ArrayList<>(ligue.getEmployes()),
            (nb) -> editerEmployer(nb));}
}
```

```
private Menu editerEmployer(Employe employe) {
    Menu menu = new Menu("Gerer : " + employe.getNom());
    menu.add(modifierEmploye(employe));
    menu.add(supprimerEmploye(employe));
    menu.add(changerAdmin(employe));

    menu.addBack("q");
    return menu;
}
```

```
private Menu gererEmployes(Ligue ligue)
{
    Menu menu = new Menu("Gérer les employés de " + ligue.getNom(), "e");
    menu.add(afficherEmployes(ligue));
    menu.add(ajouterEmploye(ligue));
    menu.add(selectEmploye(ligue));
    menu.addBack("q");
    return menu;
}
```

-Modification de l'Administrateur d'une Ligue via la CLI

- Ajout d'une fonctionnalité permettant de changer l'administrateur d'une ligue.
- Affichage des employés éligibles à la nomination.

Dans ligueConsole.Java

```
private Option changerAdmin(final Employe employe) {
    return new Option("Nommer l'administrateur", "a", () -> {
        try{
            employe.getLigue().setAdministrateur(employe);
        } catch (SauvegardeImpossible e) {
            System.err.println("Impossible de sauvegarder le changement d'administrateur : " + e.getMessage());
        }
    });
}
```

*

- Saisie des Dates via la CLI

- Possibilité d'entrer les **dates d'arrivée et de départ** d'un employé.
- Vérification des **erreurs de saisie** et affichage de messages clairs.

Dans ligueConsole.java :

```
private Option ajouterEmploye(final Ligue ligue)
{
    return new Option("ajouter un employé", "a",
            () ->
```

```
try {
ligue.addEmploye(getString("nom : "),
    getString("prenom : "), getString("mail : "),
    getString("password : "), LocalDate.parse(getString("Saisir date arriver : ")) , LocalDate.parse(getString("Saisir date depart
}catch(ExceptionD e) {
    System.out.println("La date d'arrivée est avant la date de départ");

}
catch(DateTimeParseException e) {
    System.out.println("Format de date invalide. Veuillez saisir un bon format de date ex: AAAA-MM-JJ");
} catch (SauvegardeImpossible e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

- Mise à Jour de l'Arbre Heuristique

- Adaptation pour inclure les nouvelles fonctionnalités :
    - Sélection obligatoire d'un employé.
    - Gestion des administrateurs.
    - Saisie des dates.

2.3 Connexion à la Base de Données et Gestion des Données (*Personnel 3*)

- Création et Configuration de la Base de Données

- Création d'une **base de données MySQL** dédiée.
- Exécution du **script SQL** de création des tables.
- Mise en place d'un **compte utilisateur restreint**.
- Ajout d'un fichier Credentials.java (non versionné) pour stocker les identifiants.

```
public final static int SERIALIZATION = 1, JDBC = 2,
        TYPE_PASSERELLE = JDBC;
private static Passerelle passerelle = TYPE_PASSERELLE == JDBC ? new jdbc.JDBC() : new serialisation.Serialization();
```

- Maquettes de l'Interface Graphique

- Conception des **fenêtres** et des interactions principales.
- Objectif : remplacer progressivement la CLI par une interface graphique plus intuitive.

- Gestion du Root dans la Base de Données

- **Insertion automatique du root** au premier lancement.
- Ajout d'une méthode addRoot(...) pour éviter la création en double.
- Lecture du root au démarrage pour vérifier son existence.

```
@Override
public void update(Employe employe) throws SauvegardeImpossible
{
```

```
private int id;
```

```
//creation
public Employe addRoot(String nom, String prenom, String mail, String password,
    LocalDate dateArrivee, LocalDate dateDepart) throws SauvegardeImpossible, ExceptionD
{
    Employe employe = new Employe(this, null, nom, prenom, mail, password, dateArrivee, dateDepart);
    this.root = employe;
    return employe;
}
```

```java
//Lecture
public Employe addRoot(int id, String nom, String prenom, String mail, String password,
    LocalDate dateArrivee, LocalDate dateDepart) throws SauvegardeImpossible, ExceptionD
{
    Employe employe = new Employe(id, this, null, nom, prenom, mail, password, dateArrivee, dateDepart, true);
    this.root = employe;
    return employe;
}
```

```java
PreparedStatement instructionRoot;
instructionRoot = connection.prepareStatement("select ID_Employe, Prenom_Employe, Nom_Employe, ID_Ligue, MDP_Employe
ResultSet root = instructionRoot.executeQuery();

if (!root.next()) {
    gestionPersonnel.addRoot("root", null, null, "toor",
        LocalDate.parse("2020-12-12"), LocalDate.parse("2020-12-13"));
} else {
    gestionPersonnel.addRoot(root.getInt("ID_Employe"), root.getString("Nom_Employe"),
        null, null, root.getString("MDP_Employe"),
        LocalDate.parse("2020-12-12"), LocalDate.parse("2020-12-13"));
}
```

- Chargement des Ligues et Employés

- Ajout de requêtes SQL dans getGestionPersonnel() pour :
    - Charger toutes les ligues.
    - Charger tous les employés.
    - Associer les administrateurs aux ligues.

```java
public GestionPersonnel getGestionPersonnel() throws SauvegardeImpossible {
    GestionPersonnel gestionPersonnel = GestionPersonnel.getGestionPersonnel();
    try {
        Statement instruction = connexion.createStatement();
```

```java
// Charger toutes les ligues
ResultSet resultats = instruction.executeQuery("SELECT * FROM Ligue");
while (resultats.next()) {
    Ligue ligue = gestionPersonnel.addLigue(resultats.getString("nom"));
    ligue.setId(resultats.getInt("id"));
}
```

```
// Charger tous les employés
resultats = instruction.executeQuery("SELECT * FROM Employe");
while (resultats.next()) {
    Ligue ligueActuelle = gestionPersonnel.getLigue(resultats.getInt("idLigue"));
    Employe employe = ligueActuelle.addEmploye(
        resultats.getString("nom"),
        resultats.getString("prenom"),
        resultats.getString("mail"),
        resultats.getString("password"),
        resultats.getDate("dateArrivee").toLocalDate(),
        resultats.getDate("dateDepart") != null ? resultats.getDate("dateDepart").toLocalDate()
    );
    employe.setId(resultats.getInt("id"));
```

```
            // Associer les administrateurs aux ligues
            Integer idAdmin = resultats.getInt("idAdministrateur");
            if (idAdmin != null && idAdmin == employe.getId()) {
                ligueActuelle.setAdministrateur(employe);
            }
        }
    } catch (SQLException e) {
        throw new SauvegardeImpossible(e);
    }
    return gestionPersonnel;
```

- Opérations CRUD sur Ligues et Employés

- **Insertion d'un employé** : Ajout de insert(Employe employe).

  \\

- **Modification d'un employé** : Ajout de update(Employe employe), mise à jour automatique dans les setters.

  ****

- **Suppression d'un employé** : Implémentation de delete(Employe employe), avec vérifications. (JDBC, GestionPersonnel, Passerelle)

```java
    @Override
public void delete(Employe employe) throws SauvegardeImpossible
{
    try
    {
        PreparedStatement instruction = connection.prepareStatement(
            "DELETE FROM employe WHERE id = ?"
        );
        instruction.setInt(1, employe.getId());
        instruction.executeUpdate();
    }
    catch (SQLException exception)
    {
        throw new SauvegardeImpossible(exception);
    }
}
```

```java
public void delete(Employe employe) throws SauvegardeImpossible
{
    if (passerelle != null)
        passerelle.delete(employe);
}
```

```java
//Ajout delete
public void delete(Employe employe) throws SauvegardeImpossible;
```

Modification et suppression de ligues : Adaptations similaires avec gestion des dépendances. (JDBC, GestionPersonnel, Passerelle)

```java
@Override
public void update(Ligue ligue) throws SauvegardeImpossible
{
    try
    {
        PreparedStatement instruction;
        instruction = connection.prepareStatement(
            "UPDATE ligue SET Nom_Ligue = ? WHERE ID_Ligue = ?"
        );
        instruction.setString(1, ligue.getNom());
        instruction.setInt(2, ligue.getIdLigue());

        int lignesModifiees = instruction.executeUpdate();
        if (lignesModifiees == 0)
            throw new SauvegardeImpossible(new Exception("Aucune ligue n'a été modifiée"));
    }
    catch (SQLException exception)
    {
        exception.printStackTrace();
        throw new SauvegardeImpossible(exception);
    }
}
```

```java
@Override
public void delete(Ligue ligue) throws SauvegardeImpossible
{
    try
    {
        PreparedStatement deleteLigue = connection.prepareStatement(
            "DELETE FROM ligue WHERE ID_Ligue = ?"
        );
        deleteLigue.setInt(1, ligue.getIdLigue());
        deleteLigue.executeUpdate();
    }
    catch (SQLException exception)
    {
        throw new SauvegardeImpossible(exception);
    }
}
```

```java
public void delete(Ligue ligue) throws SauvegardeImpossible;
```

```java
public void delete(Ligue ligue) throws SauvegardeImpossible
{
    if (passerelle != null)
        passerelle.delete(ligue);
}
```

- Gestion de l'Administrateur dans la Base

- **Lecture de l'administrateur** : Adaptation de getGestionPersonnel().
- **Modification de l'administrateur** : Mise à jour en base selon la modélisation choisie

```java
Integer idAdmin = resultats.getInt("idAdministrateur");
if (idAdmin != null && idAdmin == idEmploye) {
    ligueActuelle.setAdministrateur(employe);
}
```

```java
if (idAdmin != null && idAdmin == idEmploye) {
    ligueActuelle.setAdministrateur(employe);
}
```