Projet : Application de Gestion de projets étudiants

Documentation développeur

Structure du code

A travers ce projet nous devions créer une application de gestion de projets étudiants destinée à un enseignant.

Chaque entité (Formation, Etudiant, binôme, Projet) peut être vu comme un objet avec ses propres caractéristiques et méthodes associées. De plus, ces objets peuvent être présents en nombre au sein d'une université et sont liés entre eux par certaines caractéristiques, fonctions ou contraintes.

Ainsi il fallait utiliser un outil de stockage permettant de caractériser ces liens : nous avons choisi **MySQL**.

Nous devions concevoir une application graphique, donc il fallait aussi apporter cette dimension visuelle, nous avons donc choisit le langage **Swing** pour nous permettre de visualiser ces objets et leurs interactions entre elles.

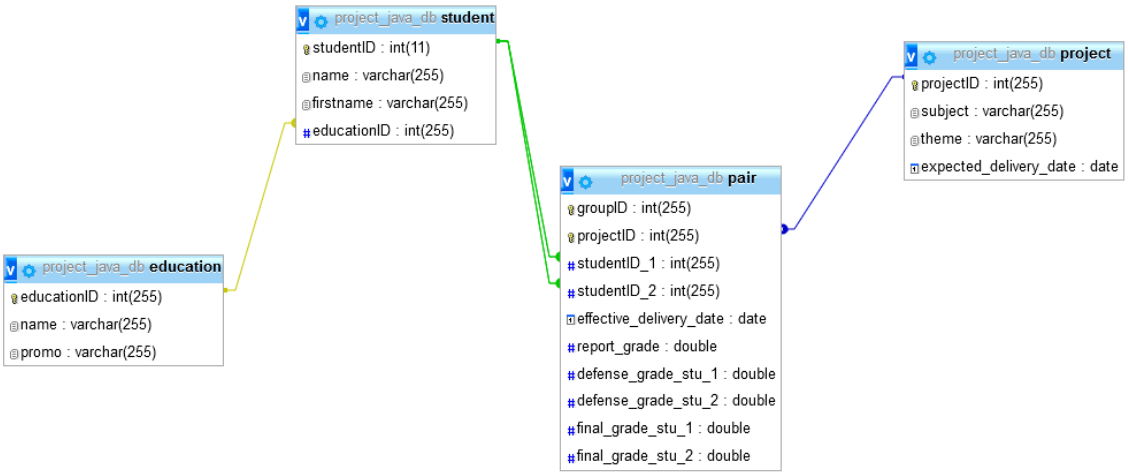
Ainsi, nous devions concevoir une application se situant au croisement de trois aspects différents de la programmation Objet : le langage **java** de base pour la conception et l'interaction entre ces objets, le langage **SQL** pour la gestion, le stockage et la caractérisation des liens / contraintes entre ces objets, et enfin le **langage Swing** pour la conception de l'interface graphique afin de permettre à l'utilisateur de visualiser toutes ces entités.

Nous avons donc décidé d'utiliser le ***Design Pattern* Modèle-Vue-Contrôleur (MVC),** qui divise l'application en trois parties distinctes ayant chacune leurs responsabilités - le Modèle via le ***package basicObjects*** qui se charge de la logique Métier de l'application, la Vue via le ***package managementAppUI*** qui gère l'interface graphique et le Contrôleur via le **package *dataBase*** qui gère les requêtes entre la Base de données et le code java.

Nous avons choisi cette structure car la séparation nette qu'elle propose facilite la gestion des responsabilités et rend le code plus modulaire.

Nous avions plusieurs contraintes à respecter dans le cahier des charges. Nous avons formalisé certaines de ces contraintes en concevant la base de données sur MySQL.

Il fallait concevoir une base de données en respectant toutes ces contraintes (types des colonnes, clés primaires, clés étrangères, l'auto incrémentation des clés, valeur par défaut de certaines colonnes à NULL pour accepter certaines entrées …).



Une fois la base de données crée nous avons décidé de construire la logique métier de notre application, qui est à la base de l'application.

Nous avons codé les classes Education, Pair, Project, Student. Ces objets seront réutilisés dans d'autres parties du code pour permettre certaines fonctionnalités.

Les Constructeurs et Getters / Setters sont ***public*** pour que les autres parties du code puissent instancier la classe et que cet objet soit en interaction avec le reste du code, cela permet une certaine créativité et du dynamisme.

Les ***attributs*** sont déclarés comme ***private*** pour éviter un accès direct non contrôlé aux données de l'extérieur de la classe. Cela permet de protéger les données internes à la classe, minimisant les risques de mauvaise manipulation. C'est un aspect du principe d'***encapsulation***.

Au niveau du ***package dataBase,*** nous avons décidé de créer une classe DBConnection afin que ce soit cette classe qui soit appelée à chaque fois par les autres classes lorsqu'une interaction avec la base de données est nécessaire. Cela permet de simplifier le code et d'éviter la redondance.

Ses ***attributs*** sont ***statiques*** pour avoir une seule référence à l'URL, au nom d'utilisateur et au mot de passe pour toutes les connexions, plutôt que d'avoir des copies distinctes pour chaque instance.

Le ***mot-clé final***, indique que ces valeurs ne doivent pas être modifiées ce qui est très utile puisque ces données sont d'une haute importance et uniques.

Pour les autres classes de ce package nous avons voulu gérer les interactions avec la BDD de manière bidirectionnelle : de la base de données au code java (**DBToProject**, **DBToStudent** ..) et du code java à la base de données en fonction du besoin (***ProjectToDB***, ***StudentToDB*** …)

Quand il s'agit de passer du code à la BDD, on utilise et récupère les informations encapsulées dans un Objet crée via les classes de ***basicObjects*** pour formaliser une requête ***SQL***.

D'où l'importance d'avoir des ***Constructeurs***, ***Getters*** et ***Setters*** en ***public***. Comme ici dans la méthode ***addPair(Pair p)*** de la classe ***PairToDB*** : On récupère les informations du ***binôme p*** via les ***Getters et Setters*** pour qu'ils soient insérés dans la table binôme associée dans la base de données via une requête SQL « INSERT INTO pair … ».

```
public class PairToDB {
        public void addPair(Pair p) {


            try (Connection connection = DBConnection.getConnection()) {
                 String Query = "INSERT INTO pair (groupID, projectID, studentID_1, studentI
                try (PreparedStatement preparedStatement = connection.prepareStatement(Query
                    preparedStatement.setInt(1, p.getGroupID());
                    preparedStatement.setInt(2, p.getProject().getProjectID());
                    preparedStatement.setInt(3, p.getStudent1().getStudentID());
                    preparedStatement.setInt(4, p.getStudent2().getStudentID());




                    java.util.Date utilDate = p.getEffectiveDeliveryDate();
                    java.sql.Date sqlDate = (utilDate != null) ? new java.sql.Date(utilDate.
                    preparedStatement.setDate(5, sqlDate);


                    preparedStatement.setDouble(6, p.getReportGrade());
                    preparedStatement.setDouble(7, p.getDefenseGradeStu1());
                    preparedStatement.setDouble(8, p.getDefenseGradeStu2());
                    preparedStatement.setDouble(9, p.getFinalGradeStu1());
                    preparedStatement.setDouble(10, p.getFinalGradeStu2());
                    preparedStatement.executeUpdate();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
```

Cela nous permet de mettre à jours la BDD quand des actions sont demandés par l'utilisateur via l'application. Il est important de garder la BDD mise à jour car elle est aussi appelée dans l'autre sens souvent pour obtenir ***la liste d'objets***, qu'une ***requête SQL*** rend très pratique.

Par exemple dans cette méthode ***getProjects()*** de la classe ***DBToProject*** où on remplit la liste des projets en itérant grâce à une requête SQL.

```
public static List<Project> getProjects() {
    List<Project> projects = new ArrayList<>();

    try (Connection connection = DBConnection.getConnection()) {
        String query = "SELECT * FROM project";
        try (PreparedStatement preparedStatement = connection.prepareStatement(query);
             ResultSet resultSet = preparedStatement.executeQuery()) {
            while (resultSet.next()) {
                int projectID = resultSet.getInt("projectID");
                String subject = resultSet.getString("subject");
                String theme = resultSet.getString("theme");
                Date expectedDeliveryDate = resultSet.getDate("expected_Delivery_Date");

                Project project = new Project(projectID, subject, theme, expectedDeliveryDate);
                projects.add(project);
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return projects;
}
```

Quand il 'agit de passer de la BDD au code, on utilise une ***requête SQL*** pour récupérer des informations et instancier des objets en conséquence.

Cela nous a été très utile pour récupérer les ID de chaque objets des tables de la BDD, puisqu'on a décidé que les ID s'auto-incrémentent à partir des tables de notre BDD. Et particulièrement pour l'ID d'un binôme qui est intrinsèquement relié à celui d'un projet :

```
public class DBToPair {

    public static int getMaxGroupID(int projectID) {
        int maxID = 0;

        try (Connection connection = DBConnection.getConnection()) {
            String query = "SELECT MAX(groupID) FROM pair WHERE projectID = ?";
            try (PreparedStatement preparedStatement = connection.prepareStatement(query)) {
                preparedStatement.setInt(1, projectID);
                try (ResultSet resultSet = preparedStatement.executeQuery()) {
                    if (resultSet.next()) {
                        maxID = resultSet.getInt(1);
                    }
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return maxID;
    }
```

Ce qui nous a avéré être très utile quand il fallait implémenter des ***méthodes techniques*** comme : ***getEducations(), getProjects, getStudents()*** qui permettent de facilement remplir des listes d'objets à partir d'une ***requête SQL***.

On a aussi les méthodes qui calculent le ***Max, le Min, la moyenne*** d'un projet via des requêtes SQL, le langage SQL rend très pratique cela, comparé à ce qu'on aurait dû faire si on était passé par du code java pure.

En voici un exemple :

```java
public static double getMinFinalGradeForProject(int projectID) {
    double minFinalGrade = 0.0;

    try (Connection connection = DBConnection.getConnection()) {
        String query = "SELECT MIN(final_grade) AS min_final_grade FROM " +
                       "(SELECT LEAST(final_grade_stu_1, final_grade_stu_2) AS final_grade " +
                       "FROM pair WHERE projectID = ?) AS subquery";
        try (PreparedStatement statement = connection.prepareStatement(query)) {
            statement.setInt(1, projectID);

            try (ResultSet resultSet = statement.executeQuery()) {
                if (resultSet.next()) {
                    minFinalGrade = resultSet.getDouble("min_final_grade");
                }
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();

    }

    return minFinalGrade;
}
```

et donc par la suite de remplir les tables de notre interface graphique très facilement.

On a là encore une preuve de la puissance de la ***programmation orientée Objet***.

Respect des contraintes du cahier des charges

Répartition du travail et Difficultés rencontrées

Conclusion

A travers ce projet nous avons pu acquérir une expérience significative au niveau de la programmation orientée objet.

Nous avons compris qu'elle comporte quatre piliers, l'abstraction, l'encapsulation, le polymorphisme et l'héritage.

Nous avons pu identifier certains avantages de la POO par rapport à la programmation classique / fonctionnelle :

- On peut créer des objets proches du monde réel, qu'on a la possibilité de percevoir et classifier de plusieurs manières grâce au polymorphisme, à l'héritage et à l'abstraction.

- On peut réutiliser des parties de codes plusieurs fois et faire interagir les objets entre eux, ce qui apporte une créativité et flexibilité.

- L'encapsulation permet de mieux sécuriser et paramétrer l'accès au code.

Finalement, nous avons réussi à utiliser l'approche orientée objet pour créer une solution, une application de gestion de projets étudiants pour enseignants, adaptée à un enjeu universitaire mais aussi très présent dans d'autres domaines qui font appel à la conception et à la gestion de bases de données respectant certaines contraintes comme l'intégrité, la cohérence et la non redondance.