Projet INFO732 : Application pour les étudiants

Lien GitHub : https://github.com/SytHamMer/Student_Dashboard.git

Sujet choisi : Problème 1

Partie Conception :

1. Use Case

Pour commencer, nous avons déterminé les acteurs qui entraient en jeu dans ce problème. Nous sommes partis du principe que l'étudiant était le seul acteur. Pour ce tableau de bord, nous avons considéré que les informations affichées (les notes, l'emploi du temps…) étaient des données récupérées sur le ou les sites de l'université, tels que moodle. En effet, ce sont des données qui ne sont pas modifiables par l'étudiant. Les récupérer sur des sites parallèles fait en sorte que seul l'étudiant est impliqué dans cette application, et non pas un administrateur supérieur.

Pour le Use Case ci-dessous, nous n'avons donc que l'acteur "Student", dans 2 états différents. En effet, toutes les fonctionnalités sont accessibles uniquement une fois l'étudiant connecté. On a donc commencé par l'acteur "Student", avec comme seule action possible "se connecter". Ensuite, on a l'acteur "Connected Student", avec les 3 fonctionnalités principales que nous avons jugées importantes. Pour chacune de ses fonctionnalités, des modifications d'affichage sont possibles pour obtenir plus de détails.

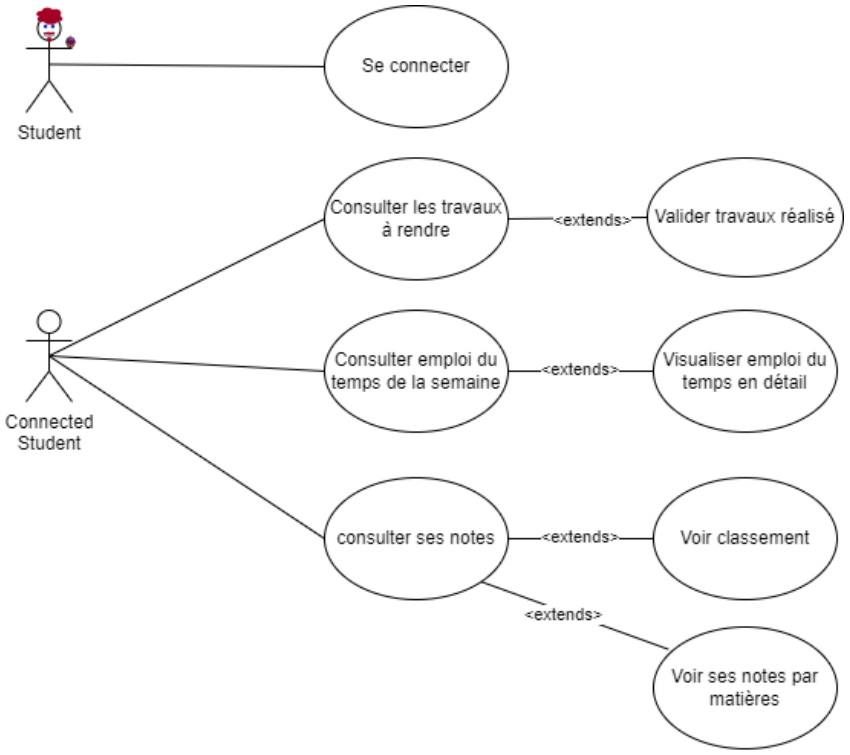

Figure 1: Use Case

2. Diagramme des classes

Ensuite nous nous sommes concentrés sur la réalisation du diagramme de classe. Ce dernier à une importance capitale pour la suite de la conception. En effet, il contient les informations relatives aux différents patrons utilisés, mais aussi l'ensemble des classes, c'est-à-dire la structure même de notre application.

Il fallait dans un premier temps définir toutes les classes nécessaires. Trois classes principales ressortent: Etudiant,Note et Matière. Ces dernières comportent toutes les méthodes principales permettant le bon fonctionnement de notre dashboard. Elles sont reliées entre elles par différentes relations, en effet un étudiant peut suivre plusieurs matières et une matière est suivie par plusieurs étudiants. Pour les notes, une note est reliée à un seul étudiant cependant ce dernier peut en posséder plusieurs, inversement une matière peut avoir plusieurs notes mais une note n'est reliée qu'à une matière.

Autour de ces classes principales nous avons ajouté différentes classes correspondantes aux choix de patrons que nous avons faits. Le premier est le patron Singleton, ce dernier appliqué à la classe BDD, (qui correspond à la base de données) permet d'utiliser une seule et même instance de toutes les autres classes lorsqu'un appel à la base de données est nécessaire.
La classe BDD contient un constructeur, ainsi qu'un attribut "instance" qui est static et qui est de type BDD.
Enfin elle contient une méthode static "getInstance()" qui permet de renvoyer l'accès à la BDD.
Grâce à cette structure la base de données peut être accessible par toutes les classes avec une simple et unique instance.
Ensuite, afin de pouvoir recevoir des notifications lorsqu'un nouveau travail est à réaliser, nous avons fait le choix d'utiliser un patron observateur. Pour ce dernier l'ajout de nouvelles classes est nécessaire:

Tout d'abord, la classe TravailEvent qui correspond à un travail à réaliser. Cette dernière est reliée à une seule matière mais une matière peut être composée de plusieurs travaux ce qui explique ces cardinalités.

Une interface "TravailListener" implémentée par la classe Etudiant, une classe EventObject héritant de TravailListener.
Le fonctionnement de ce patron, sera expliqué dans un second temps lors de la partie implémentation.

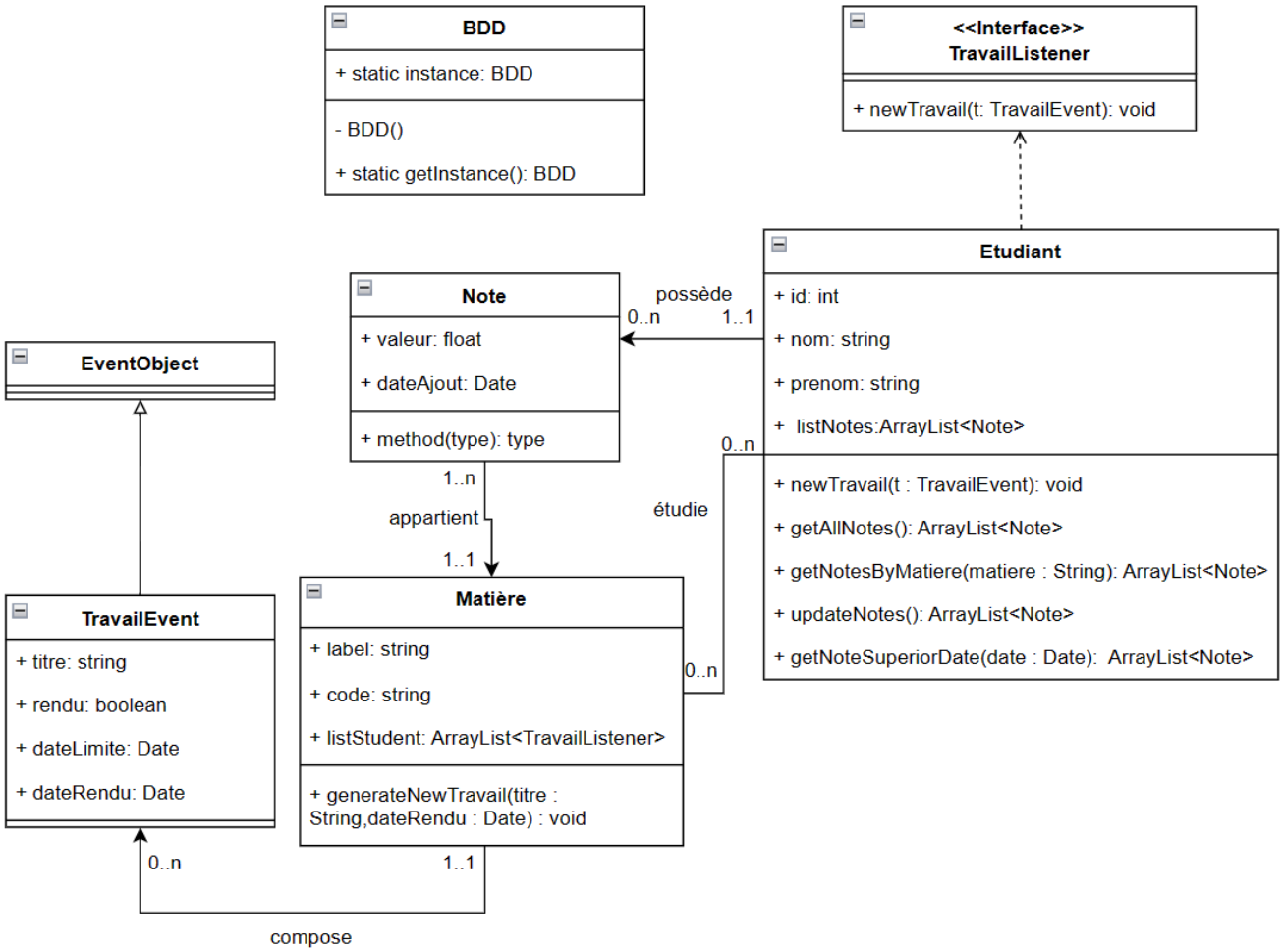

Figure 2: Diagramme de classe

3. Diagramme de séquence

Nous nous sommes ensuite concentrés sur le diagramme de séquence. Nous avons accès à ce dernier sur l'un des cas du Use Case (*Figure 1)* la consultation de ses notes par un étudiant.

Pour cette action trois classes sont sollicitées, la classe Etudiant, qui réalise l'appel de la fonction "updateNotes()". Cette fonction entraîne l'appel de la fonction getInstance() de la classe BDD, et permet ensuite d'interagir dans la base de données.
Nous avons donc dans un premier temps opté pour un seul diagramme séquence mais nous nous sommes rendu compte qu'il fallait mieux réaliser deux diagrammes de séquence pour mieux représenter notre vision des choses.
Pour ce faire nous avons réalisé un diagramme sur l'appel de la fonction "getNotesSuperiorDate()". Deux cas se distinguent comme le montre la condition dans le diagramme. Le cas où l'étudiant à déjà récupérer des notes et donc en possède déjà dans sa liste, et le cas inverse. La condition se base sur l'état de "dateLastNote".
Si "*date == null*" ceci sous-entend qu'il n'y a aucune note. Dans ce cas, on réalise une boucle sur l'ensemble des notes correspondant à l'étudiant dans la base de données. On récupère ces notes pour en créer un objet et on l'ajoute à la liste personnelle de ce dernier.
L'autre cas se distingue car si des notes existent déjà on récupère la date la plus récente. Pour récupérer dans la base de données uniquement les notes plus récentes que cette dernière. Ensuite la boucle est la même afin de créer et ajouter ces notes à la liste.

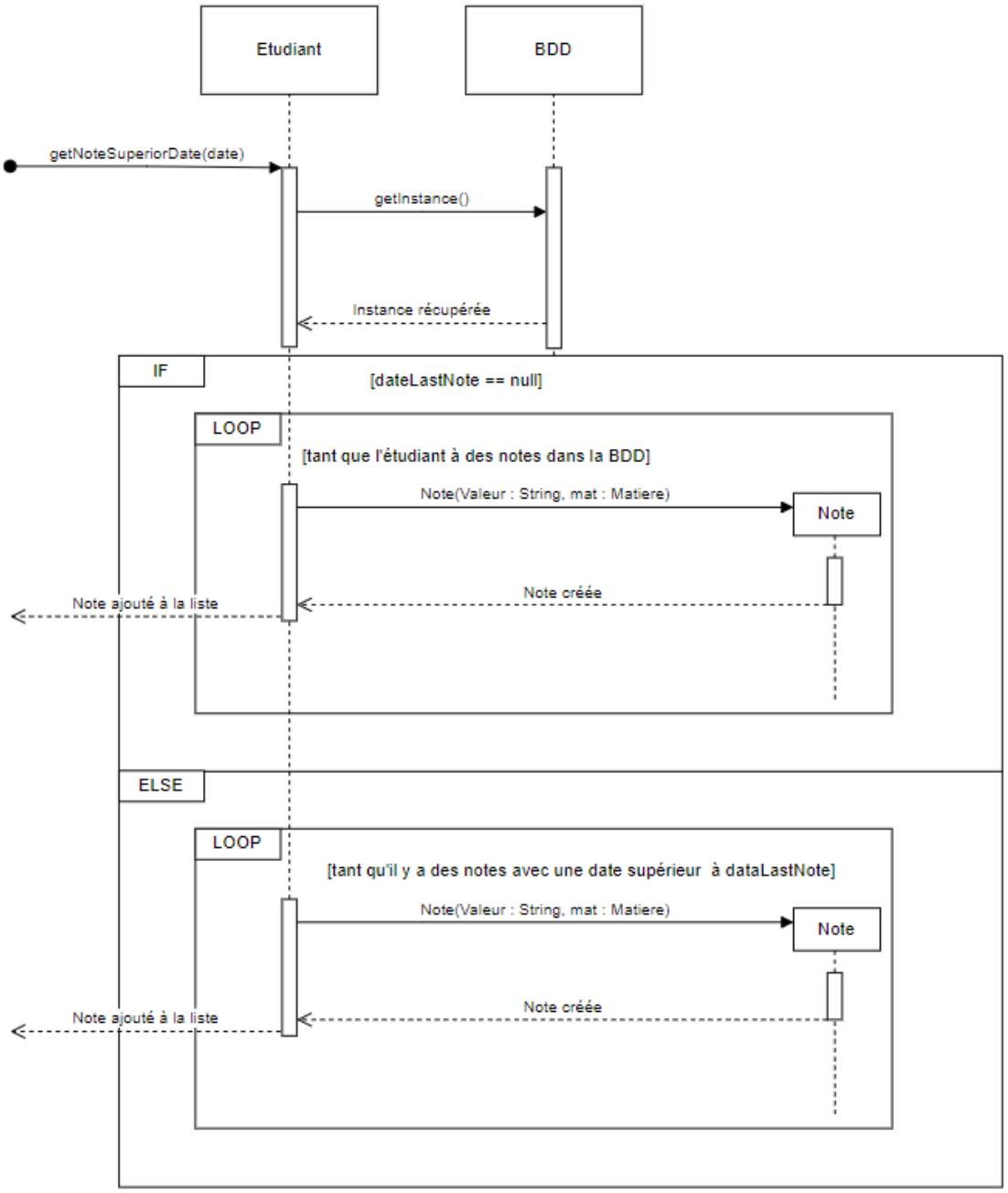

Figure 3: Diagramme de séquence de *getNoteSuperiorDate()*

Le deuxième diagramme de séquence lui se concentre sur la fonction principale “updateNotes()” cette fonction utilise la fonction “getNoteSuperiorDate()”, c’est pourquoi décrire le diagramme de séquence de cette dernière est nécessaire.

Ici, de même que précédemment une connexion est réalisée à la base de données. Puis on distingue le cas où l’étudiant n’a pas de notes comme évoqué précédemment alors l’appel de la fonction

“getNoteSuperiorDate()” a en attribue “null” et l’autre cas qui possède en attribue la date la plus récente présente dans la liste de notes.

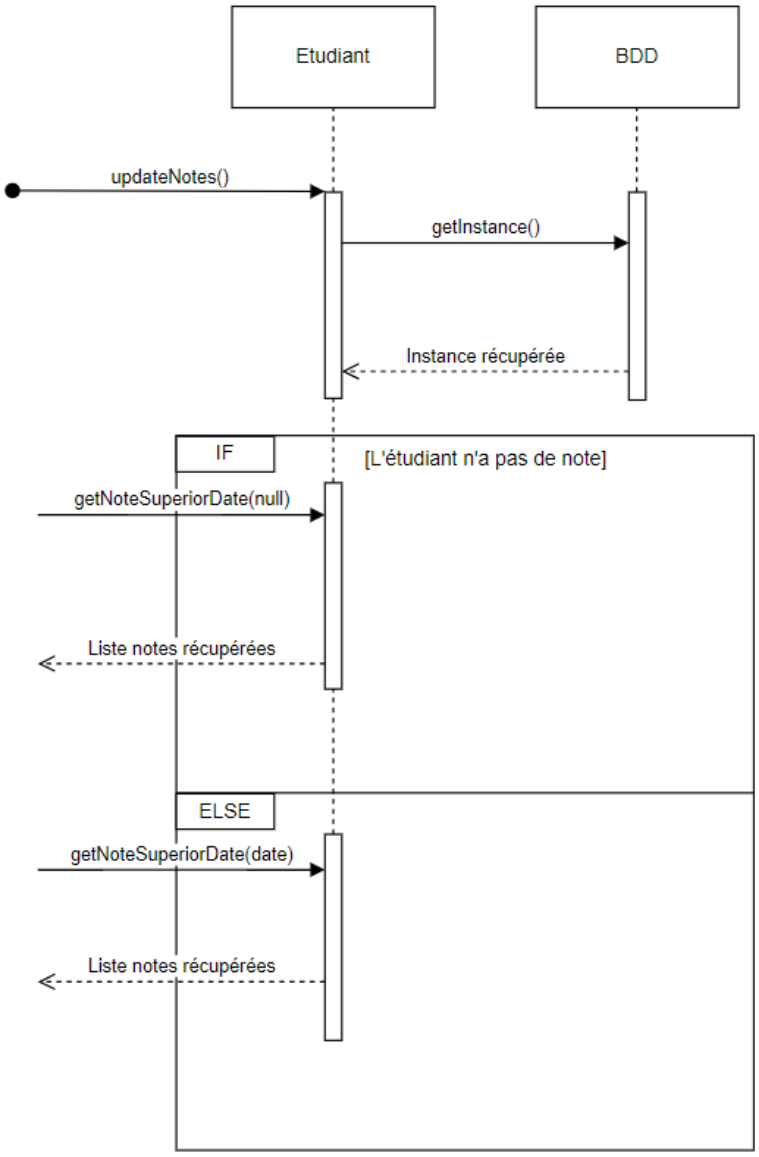

Figure 4: Diagramme de séquence d’*updateNotes()*

4. Diagramme d'état transition

Enfin nous avons réalisé le diagramme d'état transition sur le même cas que les notes. L'idée est d'étudier l'évolution de l'état de la liste de notes réalisé lors de l'appel de la fonction updateNotes().
Un point important de ce diagramme est la transition entre l'état "empty" et "not empty". En effet, nous ajoutons à cette liste de nouvelles notes. L'ensemble des notes de la base de données possédant une date plus récente que la dernière date que possède l'étudiant dans sa liste de notes personnelles.

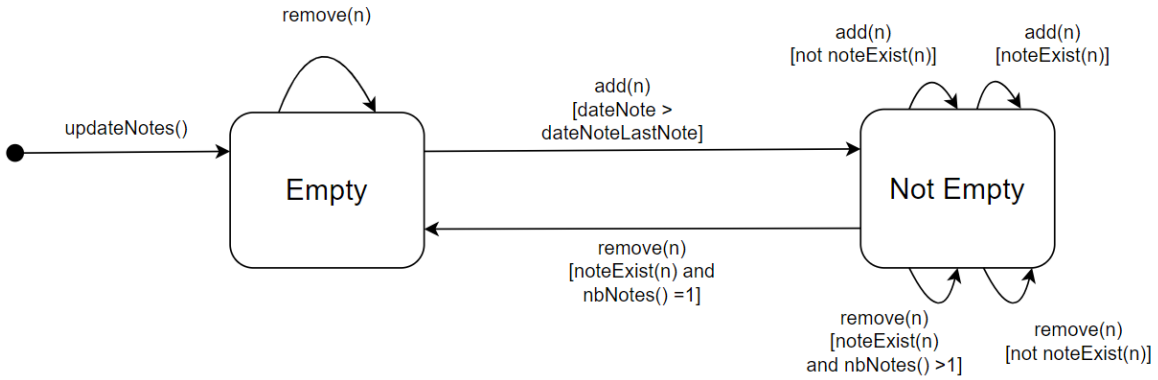

Figure 5: Diagramme d'état transition

Partie Implémentation :

Pour la partie implémentation, nous avons décidé de réaliser la fonctionnalité “Visualiser les travaux à rendre” et surtout le fait de recevoir une notification lorsque nous avons un nouveau travail à faire. Pour cela, nous avons suivi la conception faite lors du diagramme de classe avec le pattern ***Observateur***.

TravailEvent :

Nous avons créé une classe *TravailEvent* qui contient tous les attributs d’un travail et qui hérite de la classe EventObject.

```
4 usages  Trikzy7
public class TravailEvent extends EventObject {
    3 usages
    private int idTravail;
    3 usages
    private String titre;
    3 usages
    private boolean rendu;
```

TravailListener :

Ensuite, nous avons créé l’interface *TravailListener* qui contient la méthode newTravail(). Cette méthode va avertir les étudiants concernés lorsqu’ils ont un nouveau travail à faire.

```
8 usages  1 implementation  Trikzy7
public interface TravailListener {
    1 usage  1 implementation  Trikzy7
    public void newTravail(TravailEvent te);
}
```

Etudiant :

Par la suite, nous avons créé la classe *Etudiant* qui contient les attributs d'un étudiant et qui implémente l'interface TravailListener créé précédemment.

Voici les attributs de l'étudiant :

```
6 usages  Trikzy7
public class Etudiant implements TravailListener{
    3 usages
    private int id;
    3 usages
    private String nom;
    3 usages
    private String prenom;
```

Cette classe va donc implémenter la méthode newTravail() de l'interface *TravailListener*.

```
1 usage  Trikzy7
@Override
public void newTravail(TravailEvent te) {
    System.out.println("NAME/SURNAME : " + this.getNom() + "/" + this.getPrenom());
    System.out.println("[NEW TRAVAIL]");
    System.out.println("\t Matiere : " + te.getSource().toString());
    System.out.println("\t\t " + te.getTitre());
}
```

Matiere :

La dernière classe que nous avons créé pour ce patron c'est la classe *Matiere*. Cette classe contient les attributs de la classe *Matiere* et surtout la méthode generateNewTravail(). La classe *Matiere* contient un attribut listStudent qui est la liste d'étudiant qui suivent cette matière.

```
8 usages  Trikzy7
public class Matiere {
    3 usages
    private int idMatiere;
    3 usages
    private String label;
    3 usages
    private String code;
    4 usages
    private ArrayList<TravailListener> listStudent;
```

La méthode generateNewTravail() permet de de notifier tous les students qu'un nouveau travail a été mis en ligne. C'est-à dire, créer un nouveau travail et exécuter la méthode newTravail() sur tous les students qui sont présents dans la liste.

```
1 usage  Trikzy7
public void generateNewTravail(int idTravail, String titre, Date dateRendu) {
    /*
    GOAL : Envoyer une notification à chaque student dans la listStudent
     */

    TravailEvent te = new TravailEvent( source: this, idTravail, titre,  dateRendu);

    for (TravailListener tl : listStudent) {
        tl.newTravail(te);
        System.out.println("\n");
    }
}
```

Main :

Enfin pour tester ce pattern, dans le *Main* nous avons réalisé les étapes suivantes :

- Créer une matière :

```
// -- Creattion matiere
Matiere math = new Matiere( idMatiere: 1,  label: "PROBA STATS",  code: "MATH741");
```

- Créer des étudiants et les ajouter à la matière :

```
// -- Creation de 3 etudiants
Etudiant arthur = new Etudiant( id: 1,  nom: "RATA",  prenom: "Arthur");
Etudiant mathys = new Etudiant( id: 2,  nom: "LEBON",  prenom: "Mathys");
Etudiant mathieu = new Etudiant( id: 3,  nom: "FERREIRA",  prenom: "Mathieu");

// -- Ajouter les étudiants à la matière
math.addStudent(arthur);
math.addStudent(mathys);
math.addStudent(mathieu);
```

- Générer un nouveau travail pour cette matière :

```
// -- Créer un nouveau travail en math
System.out.println("----------------------------- Pattern OBSERVATEUR \n");
math.generateNewTravail( idTravail: 1,  titre: "TD2 : Exercice 1", new Date( year: 2023,  month: 10,  date: 28));
```

Nous avons donc en output le résultat suivant avec tous les élèves présents dans la liste qui ont été notifié du nouveau travail :

```
----------------------------- Pattern OBSERVATEUR

NAME/SURNAME : RATA/Arthur
[NEW TRAVAIL]
     Matiere : MATH741 [PROBA STATS]
         TD2 : Exercice 1


NAME/SURNAME : LEBON/Mathys
[NEW TRAVAIL]
     Matiere : MATH741 [PROBA STATS]
         TD2 : Exercice 1


NAME/SURNAME : FERREIRA/Mathieu
[NEW TRAVAIL]
     Matiere : MATH741 [PROBA STATS]
         TD2 : Exercice 1
```

Ensuite le deuxième pattern que nous avons implémenté, c'est le ***Singleton.*** Pour ce faire, nous l'avons appliqué sur notre classe qui interagit avec la base de données. L'objectif de ce pattern est de ne pas établir une connexion avec la BDD à chaque requête et donc de ne pas créer une nouvelle instance à chaque fois également mais de prendre l'instance qui est active.

Pour réaliser cela, nous avons mis un attribut 'instance" en static.

```
5 usages
private static BDD instance;
```

Ensuite, nous avons implémenter la méthode static getInstance() qui permet de créer une nouvelle instance si elle n'existe pas ou de reprendre l'instance qui existe déjà.

```
4 usages   Trikzy7
public static BDD getInstance() {
    if (instance == null) {
        instance = new BDD();
        System.out.println("Une nouvelle instance est crée :" + instance );
    }
    else {
        System.out.println("Une instance existe deja : " + instance);
    }

    return instance;
}
```

Nous établissons la connexion avec la base de données lorsque l'on crée une nouvelle instance. Le constructeur est en private car il ne faut pas qu'il soit possible de créer une nouvelle instance partout dans le code. Il est seulement possible d'en créer une dans la méthode getInstance().

```
1 usage   Trikzy7
private BDD() {
    // -- Faire la connexion dans le conctructeur
    // A chaque fois que l'on instancie un objet BDD, on etabli directement la
    // connexion

    this.conn = this.connexion();
}
```

Conclusion :

Nous avons donc réussi à partir d'une consigne écrite à réaliser l'entièreté de la conception de ce Students_Dashboard.
Pour ce faire nous sommes passés par plusieurs étapes lors de la conception, en passant par les différents diagrammes importants ainsi que l'implémentation.
Ce projet nous a permis de comprendre à quel point la réflexion et l'utilisation des diagrammes permet de mieux appréhender la partie implémentation du code de manière plus efficace. Ce sont des outils primordiaux pour la réalisation d'une application.
Ces connaissances emmagasinées pendant ce projet nous seront sans aucun doute utiles pour nos futurs métiers.