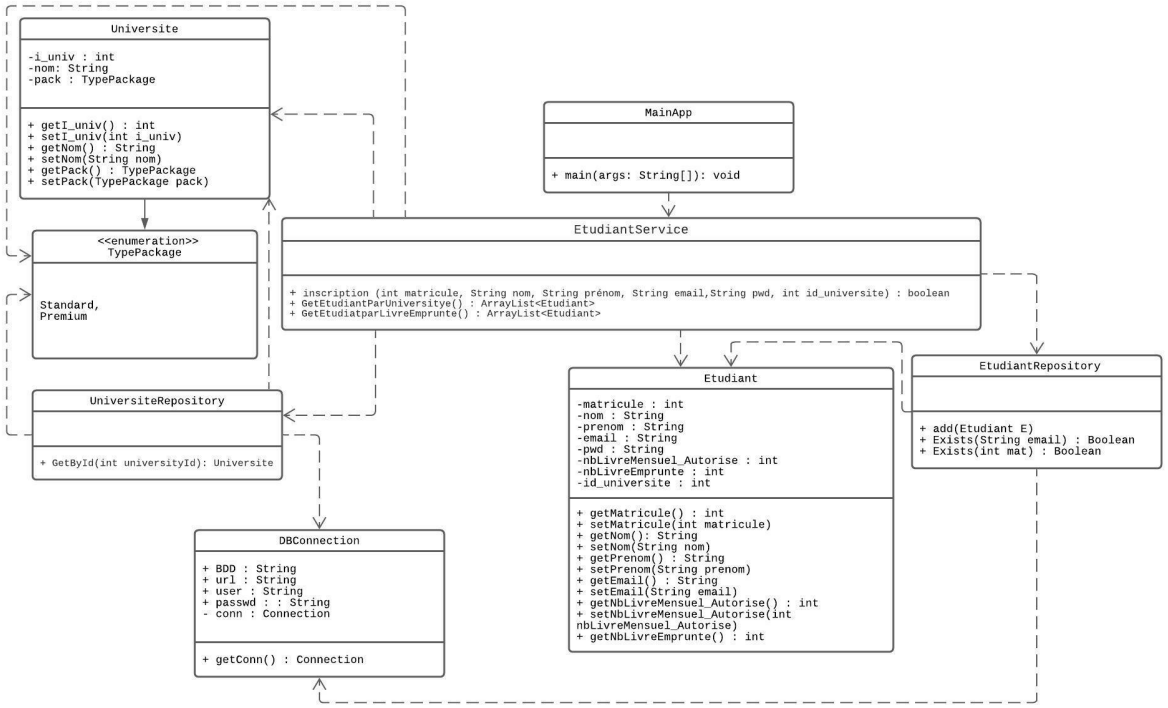
1 - Donnez le diagramme de dépendance entre ces classes :



10 - Analysez chacune de ses responsabilités, puis décidez pour chacune si vous la gardez dans la méthode « inscription » ou l'affectez à une autre classe.

Il convient de noter que la fonction inscription possède de nombreux attributs d'entrée. Elle vérifie ces attributs (leur existence et leur format). De plus, elle initialise le nombre de livres mensuels autorisés. Ainsi la fonction ne respecte pas le principe « S » de responsabilité unique SOLIDE.

À titre de suggestion, au lieu que les paramètres de fonction soient des attributs d'une classe(La classe Etudiant), nous les remplaçons par un objet de cette classe, avec des paramètres supplémentaires de type **EtudiantRepository, UniersiteRepository et Universite**. Suite à cela pour vérifier ce dernier, nous utilisons une autre classe, qui est **EtudiantRepository**.

De plus afin d'initialiser le nombre de livre mensuel autorisé nous utilisons la fonction **setNbLivreMensuelAutorise** de la classe **UniversiteRepository** en passant l'objet étudiant et son package

15 - A l'initialisation du nombre de livre mensuel autorisé ou à l'ajout du bonus, les traitements dans les deux cas dépendent du forfait de l'université à laquelle appartient l'étudiant.

- **Analysez le code de ces deux fonctionnalités et expliquez le problème qui se trouve dans ce code.**

Le problème qui se pose est que la fonction n'a pas respecté le principe "Open for extension, Close for modification" car elle est condamnée par plusieurs des conditions qui peuvent être modifiées après.

```
if (univ.getPack() == TypePackage.Standard)
          {
               stud.setNbLivreMensuel_Autorise(10);
          }
          else if (univ.getPack() == TypePackage.Premium)
          {
           stud.setNbLivreMensuel_Autorise(10*2);
          }
```

Comme solution, nous avons utilisé le patron abstract factory afin de fournir une interface **Package** pour créer des familles d'objets liés, sans spécifier de classes concrètes.

```
AbsractFactory AB = new AbsractFactory();
Package P = AB.getPackage(universite.getPack());
P.setNbLivreMensuelAutorise(etudiant);
```

Remarquons qu'on ne s'intéresse pas sur l'implémentation de la fonction setNbLivreMensuelAutorise(etudiant) dans la class etudiantService car son celle ci dépend de la classe concrète comme on voit ci-dessous :

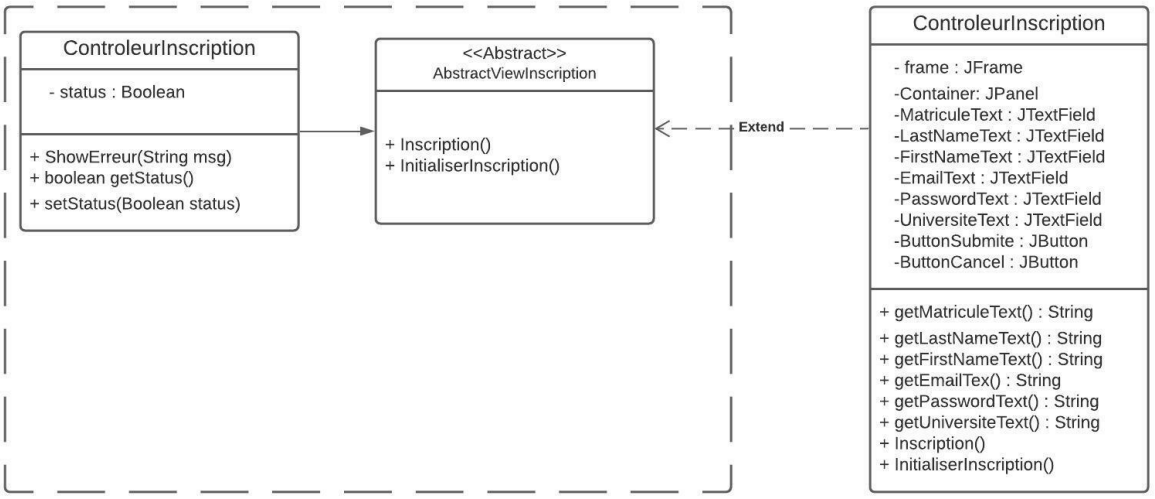
```java
public class Illimité implements Package {
    @Override
    public void setBonus(int bonus) {}
    @Override
    public int getBonus() {
        return 0;
    }
    @Override
    public void setNbLivreMensuelAutorise(Etudiant etudiant) {
        // TODO Auto-generated method stub
    }
}
```

```java
public class Premuim implements Package {
    static int bonus;
    @Override
    public void setBonus(int B) {
        bonus = B;
    }
    @Override
    public int getBonus() {
        return bonus;
    }
    @Override
    public void setNbLivreMensuelAutorise(Etudiant etudiant) {
        // TODO Auto-generated method stub

etudiant.setNbLivreMensuel_Autorise(20);
    }
}
```

```java
public class Standard implements  Package{
    static int bonus;
    public Standard(){}
    @Override
    public void setBonus(int B) {
        bonus = B;
    }
    @Override
    public int getBonus() {
        return bonus;
    }
    @Override
    public void setNbLivreMensuelAutorise(Etudiant etudiant) {
        // TODO Auto-generated method stub

etudiant.setNbLivreMensuel_Autorise(10);
    }
}
```

23- On souhaite que le contrôleur « ControleurInscription » dépend de l'abstraction de la présentation et non pas de son implémentation. Comment peut-on réaliser ça ?

Afin de réaliser ça, nous avons ajouter une classe abstract **AbstractViewInscription** qui va être héritée par la classe **ViewInscription** et appliquer l'injection de dépendance dans la classe **ControleurInscription**.



27. Donnez le diagramme de dépendance entre les packages.