** **

**Filière : Génie informatique**

TD0 : Principes SOLID

Binôme :

**Hajar Elmechhouri** et **Mouna Ettalbi**

**Année universitaire : 2024/2025**

1. Refacetoring du code

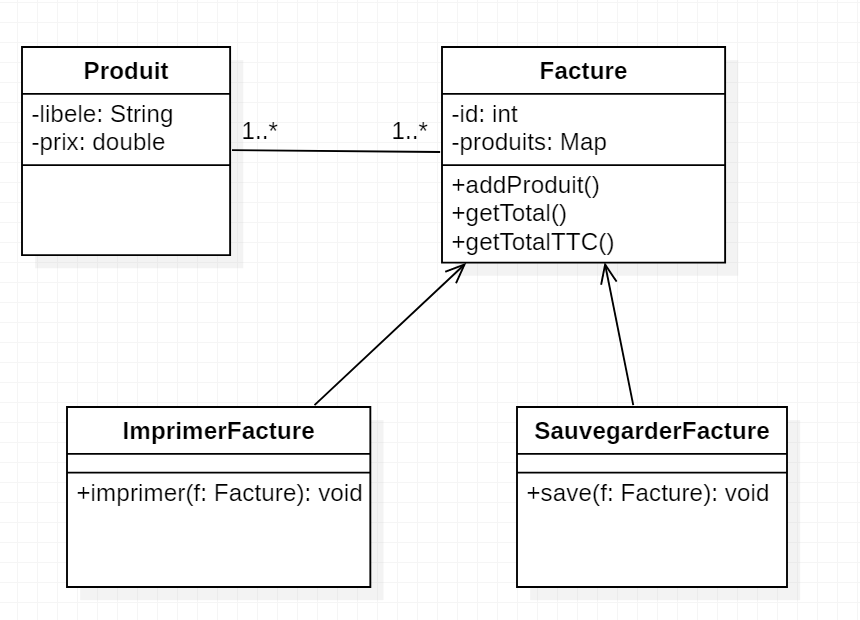
**Exercice 1 :**

1. Le principe SOLID qui n'est pas respecté ici est le **principe de responsabilité unique .**

En effet, la classe Facture a plusieurs responsabilités :

* Gestion des produits et du calcul du total.
* Affichage des informations de la facture (imprimer).
* Sauvegarde de la facture dans un fichier (saveFacture).

1. Nouveau diagramme de classe :

****

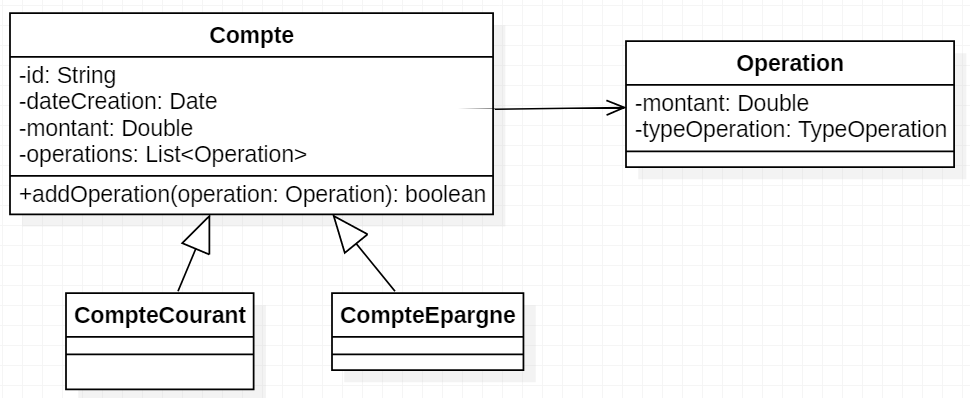
Ce nouveau diagramme de classe offre une conception plus claire et modulaire du système de facturation. En séparant les responsabilités et en respectant les principes SOLID, le code devient plus maintenable, extensible et testable.

**Exercice 2 :**

1. Principe SOLID non respecté :

* Le principe de **Responsabilité Unique** n'est pas respecté. La classe Compte gère à la fois les informations du compte (id, date de création, montant, type) et la logique des opérations bancaires (addOperation).
* De plus, le **principe Ouverture/Fermeture** est également violé. Le traitement des types de compte (Courant et un autre type) est intégré dans la méthode addOperation. Si on souhaite ajouter un nouveau type de compte, il serait nécessaire de modifier cette méthode, alors qu'elle devrait être **ouverte à l'extension et fermée à la modification**.

1. Nouveau diagramme de classe :

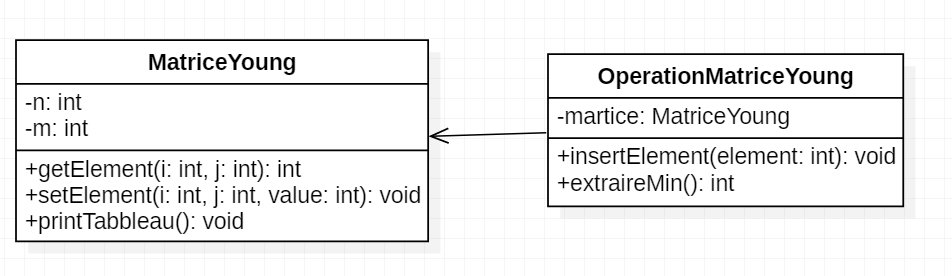


Respect du SRP : Chaque classe gère sa propre logique métier, évitant ainsi que la classe Compte ait plusieurs responsabilités.

Respect du OCP : Pour ajouter un nouveau type de compte, il suffit de créer une nouvelle sous-classe sans modifier le code existant dans Compte.

**Exercice 3 :**

1. Le principe de "Responsabilité unique" n'est pas respecté. La classe MatriceYoung a plusieurs responsabilités : elle gère la logique de manipulation du tableau de Young (extraction du minimum, insertion et rétablissement de la propriété du tableau) ainsi que la représentation de la structure elle-même (les opérations sur les éléments du tableau). Cela viole le SRP, car une classe devrait avoir une seule responsabilité.
2. Nouveau diagramme de classe :

****

La séparation des opérations permet de mieux comprendre et tester chaque partie de l'implémentation.

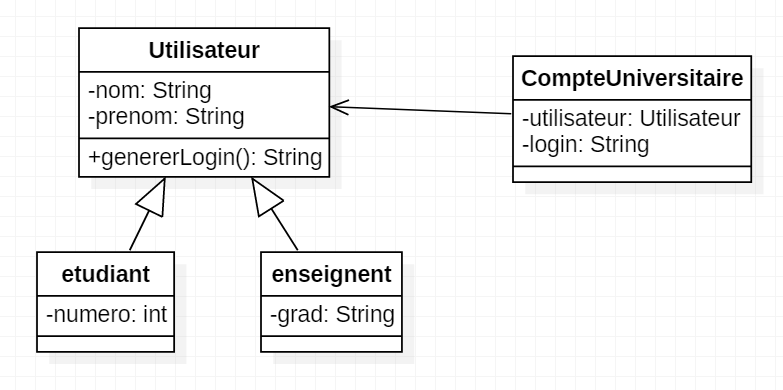
En séparant les opérations, il devient facile d’ajouter d’autres fonctionnalités (comme la recherche) sans modifier la classe de base.

**Exercice 4 :**

1. Principe SOLID non respecté :

* Le principe de **Responsabilité Unique** n'est pas respecté.La classe CompteUniversitaireEnseignent et la classe CompteUniversitaireEtudiant sont responsables à la fois de la gestion des comptes et de la génération des logins. Cela signifie que chaque classe a deux responsabilités différentes :
* Gérer les informations des enseignants ou des étudiants.
* Gérer la création des logins.
* Le principe d’**Inversion des dépendances** n'est pas respecté. La classe CompteUniversitaireEnseignent dépend directement d'une classe concrète Enseignent. Pour respecter ce principe, il serait préférable d'utiliser une interface ou une classe abstraite pour permettre une plus grande flexibilité et faciliter les tests.

1. Nouveau diagramme de classe :



1. Conception avancée

**Exercice 5 :**

1. Non, il n'est pas nécessaire que la classe Calendrier soit fortement couplée à la classe Personne. Un couplage fort signifie que Calendrier dépend directement de Personne, ce qui limite la flexibilité et la réutilisabilité du code.

La première solution simple vue en cours pour diminuer ce couplage est d'introduire une **interface**. En définissant une interface, par exemple IPersonne, qui contient les méthodes nécessaires, la classe Calendrier ne dépendrait plus directement de Personne, mais d'une abstraction.

1. L'interface actuelle IPersonne n'est pas totalement adaptée, car elle inclut des méthodes inutiles comme celles pour envoyer des SMS ou des emails.

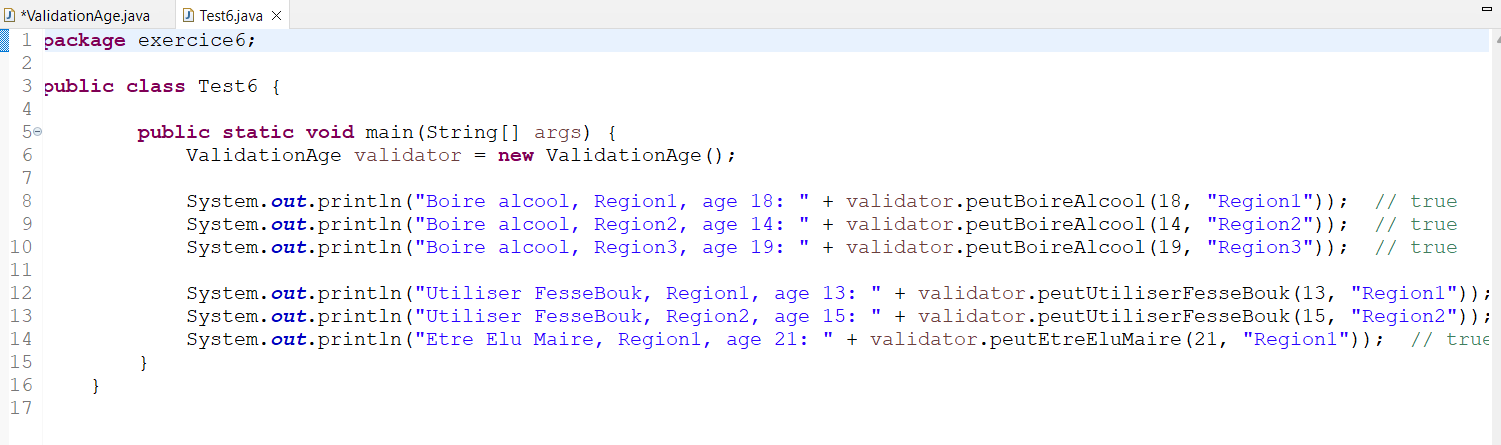
**Interface mise à jour IPersonne :**

L'interface doit se concentrer sur les informations pertinentes pour un anniversaire : le nom et la date de naissance (pour calculer l'âge). Les méthodes pour envoyer des messages (SMS ou emails) ne sont plus nécessaires.

1. .
2. Le principe **SOLID** qui n'était pas respecté lorsque l'on voulait utiliser directement IPersonne dans Calendrier est le **principe de l'inversion de dépendance.**

**Exercice 6 :**

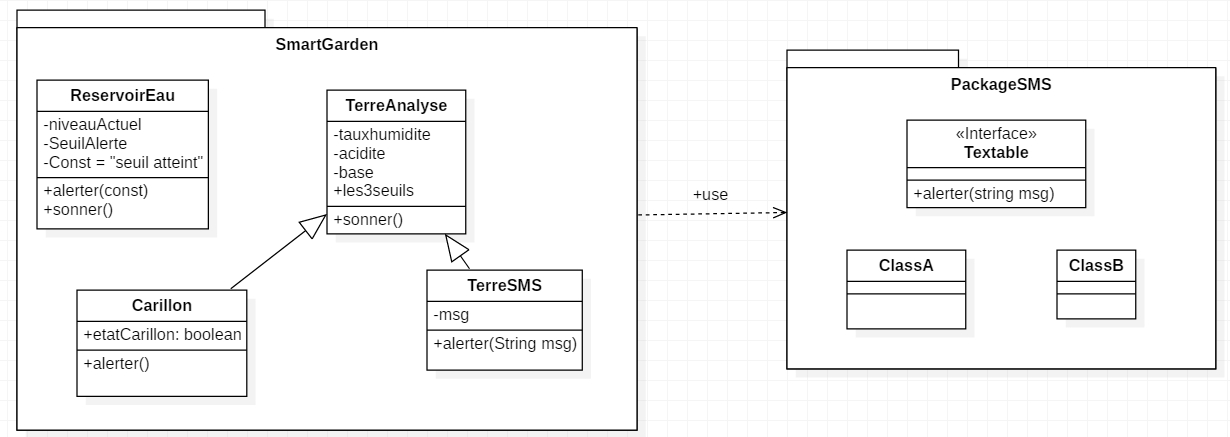
****

****

1. **:** Cette solution ne respecte pas complètement les principes SOLID, notamment :

* **Responsabilité unique (SRP)** : La classe ValidationAge fait trop de choses. Elle gère la validation des âges pour différentes régions et différentes règles.
* **Ouverture/fermeture (OCP)** : La classe n'est pas facilement extensible. Si une nouvelle région avec de nouvelles règles devait être ajoutée, il faudrait modifier la classe ValidationAge, ce qui va à l'encontre du principe OCP.

Exercice 7 :



* 1. L'inconvénient principal de cette architecture est que chaque nouveau composant sonore ou de surveillance doit être relié à la méthode d'envoi de SMS, ce qui complexifie l'ajout de nouveaux dispositifs. Si vous voulez ajouter un nouveau dispositif (par exemple, un autre capteur ou une alarme), vous devrez modifier le code pour chaque nouveau composant afin de le connecter à l'interface d'alerte.

Cela va à l'encontre des principes SOLID, notamment le principe de responsabilité unique et le principe d'extension ouverte, modification fermée .