TD –Principes SOLID

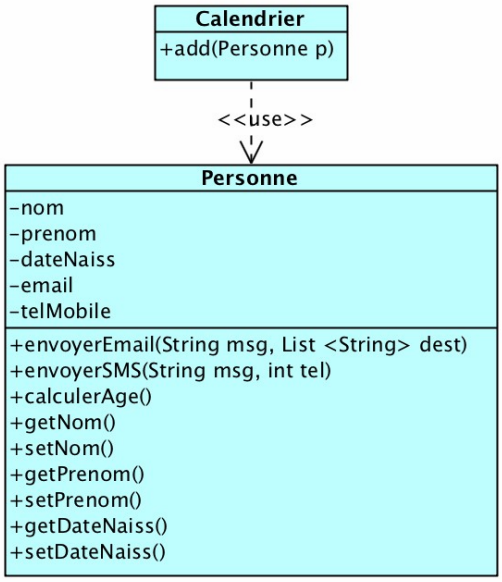
EXERCICE 1 – Gestion d'anniversaires

Imaginons devoir développer une application qui gère les anniversaires : elle stocke les dates d'anniversaires de nos amis et leur envoie un message d'anniversaire automatique.

Nous avons donc une classe qui s'occupe de charger et d'enregistrer les dates d'anniversaires dans un Calendrier partagé. La classe **`Calendrier`** dispose ainsi d'une méthode `add()` qui s'écrit comme ceci :

```
public class Calendrier { public void
      add(Personne personne) {
      /* TODO */
      }
}
```

où Personne est une classe réutilisée de l'équipe de développement :



1.1) On constate donc que **`Calendrier`** est fortement couplée à **`Personne`**. Est-ce bien nécessaire ? Quelle première solution simple vue en cours permettrait de diminuer le couplage ?

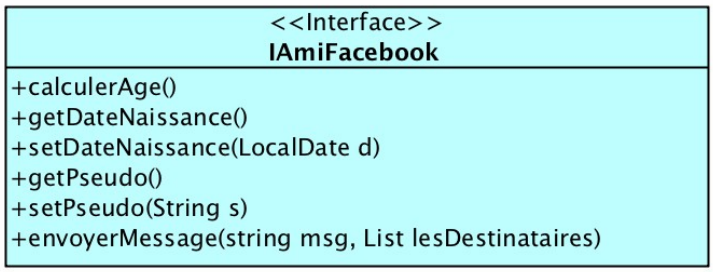
1.2) On décide de construire une interface **`IPersonne`** avec juste les méthodes de **`Personne`** (`extract interface` sous l'IDE).

Décrire l'interface **`IPersonne`** ainsi que la nouvelle signature de `add()` dans **`Calendrier`**.

Lors de la Revue de Sprint, vous apprenez du client que **`Calendrier`** ne va finalement jamais envoyer de SMS et d'emails à une Personne, mais un message pour leur anniversaire. **`Calendrier`** a donc besoin de connaître le nom de la personne et son âge, càd. qu'elle peut affecter une date de naissance, la consulter et envoyer un message.

`IPersonne` est-elle donc adaptée à notre situation ? Si non, décrire ce que serait l'interface souhaitée.

1.3) Votre chef vous apprend finalement que **`Calendrier`** devra aussi gérer les anniversaires des membres de Facebook, avec le contrat suivant :



Trouver les éléments communs entre **`IPersonne`** et **`IAmi Facebook.`**

Proposer ainsi une amélioration de **`Calendrier`** visant à diminuer le couplage entre les 3 classes.

EXERCICE 2 – Appliquer un principe SOLID

3.1- Examiner ce code PHP, donner les responsabilités de classe UserService, et mentionner le principe SOLID qu'elle ne respecte pas.

```
class UserService {
	public function updateFromAPI( User $user): User
	{
		// ...
	}

	public function removeSession( User $user ): void
	{
		// ...
	}

	public function isUserAllowedToAccessAdmin( User $user ): bool
	{
		// ...
	}

	public function serialize( User $user ): string
	{
		// ...
	}
}
```

3.2 – Donner le code corrigé avec respect du principe SOLID.
Indice : arboresence initiale de fichiers PHP :

```
Services/
└── UserService.php
├── ...
```

3.3 – Rappeler les avantages obtenus avec cette modification.

EXERCICE 3 – OCP

Imaginons qu'on dispose d'une classe qui applique des validations en fonction de l'âge d'un utilisateur :

```
public class ValidationAge {
public boolean peutBoireAlcool(int age) {
return age >= 18;
  }
  public boolean peutUtiliserFesseBouk(int age) {
return age >= 13;
  }
  public boolean peutEtreEluMaire(int age) {
return age >= 21;
  }
}
```

Cette classe est utilisée par tous les programmes de l'agence qui exploitent les utilisateurs. Six mois plus tard, on apprend que notre application s'élargit à d'autres coins de la France où les limites d'âge ne sont pas les mêmes…

4.1- Sous IDE, coder la solution proposée par le collègue Julien, où on modifie la classe `ValidationAge` en envoyant la région en paramètre aux fonctions, pour tester si ça concerne la métropole ou les DomTom (limites d'âge : 14, 15 et 19 pour les conditions ci-dessus). Testez si ça fonctionne.

4.2 - Pensez-vous que la classe obtenue respecte les principes SOLID ? Proposez une solution plus élégante.
Indice : encore une petite interface ?

EXERCICE 4 – Principe de Liskov

Voici un code C# représentant un contrat pour l'acquisition de données :

```
public interface IDevice{
      void open();
void read();
       void close();
}
```

Chaque matériel d'acquisition peut être différent selon son type d'interface. On a par exemple des interfaces USB, des interfaces réseaux (TCP ou UDP), des interfaces PCI express ou n'importe quel autre type d'interface d'ordinateur.

6.1- Les composants Client de **`IDevice`** n'ont pas besoin de savoir quel type de matériel il sollicite. Ils passent par ex. par une méthode `public void` **`acquire`**`(IDevice aDevice)` pour accéder aux données. Quel avantage cela procure-t-il ?

6.2 -Supposons qu'il n'y ait dans un premier temps que 2 classes concrètes qui implémentent `IDevice` interface :

```
public class PCIDevice implements IDevice {
public void open(){
   // Device specific opening logic
   }
   public void read(){
   // Reading logic specific to this device
   }
   public void close(){
   // Device specific closing logic.
   }
}
public class NetworkDevice implements IDevice {
public void open(){
   // Device specific opening logic
   }
   public void read(){
   // Reading logic specific to this device
   }
   public void close(){
   // Device specific closing logic.
   }
}
```

Les 3 méthodes (`open`, `read` and `close`) suffisent pour gérer les données sur ces matériels.
Imaginons qu'on introduise **un nouveau dispositif d'acquisition** basé sur une interface USB.

Mais avec ce matériel USB, quand ou ouvre la connexion, les données de la précédente connexion sont **toujours présentes** dans le buffer. Ainsi au premier appel à `read()` de ce matériel, les données de la précédente session sont renvoyées, ce qui fait que l'acquisition des données est corrompue pour la session.

Heureusement, les pilotes des dispositifs USB fournissent une fonction *refresh()* qui efface le buffer du support USB.
On propose donc cette classe **`USBDevice :`**

```
public class USBDevice
implements  IDevice  {
public void open(){
 // Device specific opening logic
  }    public  void
read(){
 // Reading logic specific to
 // this device
 }

 public void close(){
  // Device specific closing logic
 }   public void
refresh(){  // specific
only to USB
 // interface Device
 }
}
```

Et on propose de vérifier le type de matériel au niveau du composant Client : si c'est un matériel USB, on agit en conséquence. Voici le code de la méthode ***acquire()*** d'une classe Client qui exploite le matériel **`IDevice`** :

```
public      void      acquire(IDevice      aDevice){
aDevice.open();
    // Identify if the object passed here is an USBDevice class Object.
       if( aDevice.getType() == typeof(USBDevice) ){
USBDevice anUsbDevice = (USBDevice) aDevice;
       anUsbDevice.refresh();
    }
    // Le reste du code….
}
```

Que penser de cette solution ? Proposer une amélioration *(indice : modifier l'interface ou la classe d'implémentation USBDevice)* **et discuter de ces alternatives.**

EXERCICE 5 – Inversion de Contrôle

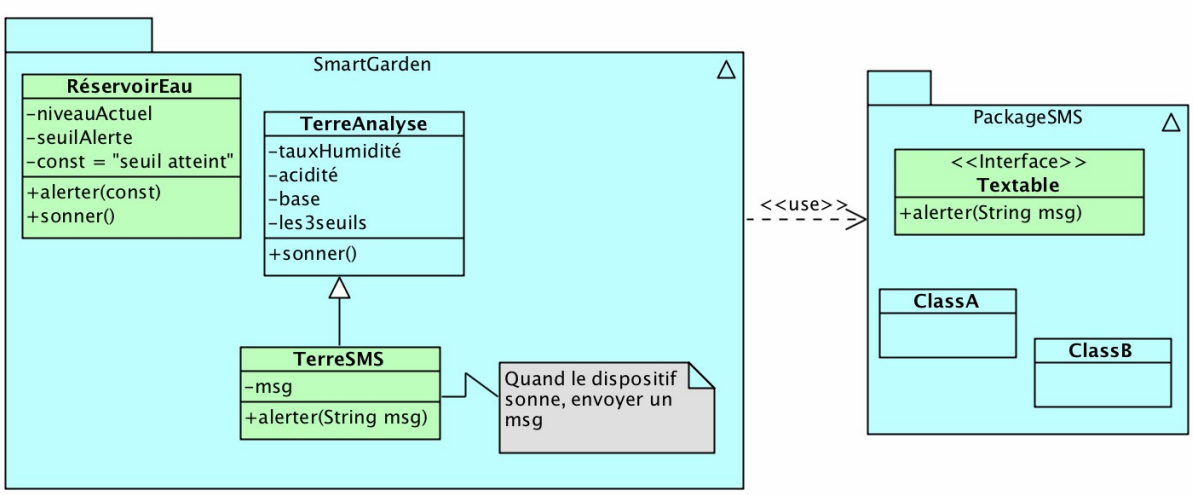
Dans un jardin connecté, différents dispositifs sont installés, certains fabriqués par nous, d'autres sont des composants propriétaire. Un suivi à distance est possible par notifications de SMS. Ainsi :

- Un **réservoir** fait maison (en vert) sonne quand il atteint un seuil d'alerte : on a ajouté une puce qui envoie alors un SMS au jardinier : "Seuil atteint" ;

- Un dispositif propriétaire d'analyse de la terre se met à sonner quand il y a une anomalie sur les mesures (humidité, acidité, base).

- On souhaiterait que le jardinier soit aussi informé à distance : on crée alors, en extension de ce capteur (en vert), un composant qui va **transmettre l'information au jardinier via un SMS.**

On a donc le S.I. suivant :



Mais on aimerait que le déclenchement d'une alerte puisse aussi se faire par une personne : on installe donc un nouveau composant, un **carillon**, qu'un employé du jardin actionne quand il constate un problème.
Et comme pour l'analyse de la terre, on programme le carillon pour qu'il puisse transmettre un SMS aux jardiniers.

7.1. Modifier le **DCL** en conséquence.

7.2. Quel **inconvénient** possède cette architecture ? Imaginez qu'on ajoute un autre composant sonore, puis un autre, etc.

7.3. Notre PKG SmartGarden dépend donc d'une interface **alerter()** par SMS (développée par nos soins). On aimerait que ce soit l'inverse, que le package technique d'alerte dépende du package Métier. Comment procéder ?

Indice : écoutez bien ! La tonalité de sonnerie de chacun des dispositifs est différente. Et si on équipait le jardin d'un micro ?