Mini Rapport TP2

Question 1:

Les principes non respectés dans l'implémentation de la classe **Contact** et de l'interface **ContactService** sont les suivants :

- **Single Responsibility Principle** : La classe **Contact** devrait uniquement gérer les informations relatives à un contact, et non la sauvegarde en BDD, l'envoi de mails ou d'autres types de messages.
- **Interface Segregation Principle**: L'interface **ContactService** regroupe des comportements différents (envoi de messages, gestion de la liste des contacts, gestion de la base de données). Il serait préférable de diviser cette interface en plusieurs interfaces spécialisées, telles que **ServiceEnvoi** et **GestionBdd**, chacune traitant un seul type de comportement.
- **Don't Repeat Yourself (DRY)** : Le principe **DRY** n'est pas respecté, car on retrouve des lignes de code similaires dans deux méthodes différentes pour afficher le nom et le numéro de téléphone d'un contact. Il serait plus efficace d'appeler **toString()** dans **getInfoContact()**.

Question 2:

Pour gérer tout type de message pouvant être envoyé, on ajoute une interface **EnvoiService** qui gère les envois. Ensuite, il y a une classe pour chaque type de message (par exemple, **ServiceEnvoiSMS**, **ServiceEnvoiMail**, etc.). Ces classes implémentent l'interface **EnvoiService** et définissent la méthode **envoi(Contact c, String m)** spécifiquement pour chaque type de message. Ainsi, il suffit que le service correspondant au type de message à traiter appelle la méthode **envoi** pour appliquer le processus approprié.

Si un nouveau type de message doit être ajouté, il suffit de créer une nouvelle classe **ServiceEnvoi...** qui implémente l'interface **EnvoiService** et définit la méthode **envoi(Contact c, String m)** avec le comportement spécifique au type de message.

Question3:

Puisque la classe ServiceEnvoiPhoto ne peut pas être modifiée, nous allons utiliser le **pattern Adapter**.

Solution 1 :
On crée la classe **ServiceEnvoiPhotoAdapterObj**, qui adaptera l'interface **EnvoiService** à la méthode **envoiPhoto()** de **ServiceEnvoiPhoto** (Adaptateur d'objet).

Solution 2 :
On peut également utiliser un adaptateur de classe. Dans ce cas, on crée la classe **ServiceEnvoiPhotoAdapterClss**, qui hérite de **ServiceEnvoiPhoto** et implémente

l’interface **EnvoiService** pour l’adapter à la méthode **envoiPhoto()** de **ServiceEnvoiPhoto**.

Question 4:

Puisqu'on a plusieurs formats possibles dans lesquels on doit récupérer les infos d’un contact, on doit passer par le **Pattern Strategy**. Il va gérer plusieurs formats de présentation des infos d'un contact (texte, JSON, XML) de manière flexible et modulaire. En créant une interface **FormatStrategy** et en définissant une classe pour chaque format, on sépare la logique de formatage du reste du code de la classe **Contact**. Cela permet à **Contact** de déléguer le choix du format à une stratégie, rendant le système extensible et facile à maintenir sans modifier le code de **Contact.**

Solution 1 :

Nous ajoutons un attribut -format- dans la classe **Contact**, de type **FormatStrategy**, qui va définir le format des informations du contact. Ce format peut être modifié dynamiquement grâce à un **setter** (setFormatStrategy). La méthode **getInfoContact()** appelle ensuite la méthode **formatter()** sur cet attribut pour obtenir les informations du contact dans le format souhaité (XML, JSON ou texte).

Solution 2 :

Nous n'ajoutons pas d'attribut formatStrategy dans **Contact**. Au lieu de cela, le format est **spécifié à chaque appel de la méthode getInfoContact()** en passant directement un objet **FormatStrategy** en paramètre. Ainsi, **Contact** ne stocke pas l'information sur le format, mais on l'indique explicitement au moment où on souhaite obtenir les informations du contact.