Interfaces - comprendre utilité

🧩 EXERCICE UML — “Du code rigide à l’interface flexible”

🎯 Objectif

> Comprendre **pourquoi** le couplage fort entre classes rend ton code rigide,
>
> et **comment** une interface permet d’obtenir du polymorphisme et de la flexibilité.

📦 ÉTAPE 1 — Version rigide (pas d’interface)

Diagramme UML

```
classDiagram
    class User {
        -String name
        -SmsNotifier notifier
        +User(String name)
        +notify(String message)
    }

    class SmsNotifier {
        +send(String message)
    }

    User --> SmsNotifier : dépendance directe
```

🔍 Analyse UML

- User **connaît directement** la classe SmsNotifier. → Si tu veux un autre moyen de notification (Email, Push…), il faut **modifier** la classe User.
- C’est ce qu’on appelle un **couplage fort** : l’une **dépend** d’une classe concrète.

💬 Questions à te poser avant de coder

1. Que se passe-t-il si le client te demande d'ajouter un EmailNotifier ?

   → Doit-on modifier User ?
2. Si un jour tu veux tester User sans vraiment envoyer de SMS,

   → peux-tu le faire facilement ?

⚙️ ÉTAPE 2 — Version flexible (avec interface)

Diagramme UML

```
classDiagram
    class Notifier {
        <<interface>>
        +send(String message)
    }

    class SmsNotifier {
        +send(String message)
    }

    class EmailNotifier {
        +send(String message)
    }

    class User {
        -String name
        -Notifier notifier
        +User(String name, Notifier notifier)
        +notify(String message)
    }

    Notifier <|.. SmsNotifier
    Notifier <|.. EmailNotifier
    User --> Notifier : dépendance abstraite
```

🔍 Analyse UML

- User ne dépend **plus d'une classe concrète**, mais d'une **abstraction** (Notifier).
- Notifier agit comme une **prise universelle** : tant que la classe "implémente" ce contrat, elle est compatible.

💬 Questions à te poser avant de coder

1. Comment User choisit-il maintenant le type de notifier à utiliser ?

   (indice : on lui **injecte** via le constructeur)
2. Si tu veux créer un PushNotifier, dois-tu modifier User ?

   (indice : non, car User ne connaît que Notifier)
3. Que se passe-t-il si tu veux tester User sans rien envoyer ?

   (indice : tu peux créer un MockNotifier qui simule un envoi)

🧠 COMPARAISON DES DEUX MODÈLES

Aspect	Version rigide	Version flexible
Couplage	Fort (User → SmsNotifier)	Faible (User → Notifier)
Extensibilité	Doit modifier User	Juste créer une nouvelle classe
Polymorphisme	Aucun	Oui (un User fonctionne avec tout Notifier)
Testabilité	Difficile	Facile avec MockNotifier
Principe SOLID respecté	❌ Non (viol du OCP)	✅ Oui (Open/Closed Principle)

💬 Résumé conceptuel à retenir

> 🔶 Une **classe concrète** est rigide : elle "sait faire une seule chose".

> 🔶 Une **interface** définit un **comportement générique** : plusieurs classes peuvent “savoir faire” de manières différentes.

> 🔶 Le polymorphisme par interface permet à un objet d’agir **sans savoir qui** exécute l’action.

🎯 Mini challenge

En t’aidant uniquement du diagramme UML **de la version flexible**,

essaie de :

1. recréer les classes Notifier, SmsNotifier, EmailNotifier, User, et Main,
2. faire en sorte que User puisse envoyer une notification quel que soit le type de Notifier,
3. tester l’ajout d’un PushNotifier **sans toucher à User**.