

LES NIVEAUX D'IMPLÉMENTATION D'ARCHITECTURE LOGICIEL AVEC LES MODÈLES DE CONCEPTION (DESIGN PATTERNS)

Par Metra Phirielle VANDJY BOUNDZANGA

DEFINITION

- Les **modèles de conception (Design Patterns)** en architecture logicielle sont des solutions réutilisables pour résoudre des problèmes courants de conception. Ils sont classés en trois grandes catégories selon leur objectif tels que **Modèles de Création (Creational Patterns)**, **Modèles de Structure (Structural Patterns)** et les **Modèles de Comportement (Behavioral Patterns)**
- Les **Modèles de Création (Creational Patterns)** traitent de la **création d'objets** en offrant des mécanismes flexibles et réutilisables. Ce modèle est composé de **Singleton** (Garantit qu'une classe n'a qu'une seule instance), **Factory Method** (Délègue l'instanciation à des sous-classes), **Abstract Factory** (Fournit une interface pour créer des familles d'objets liés), **Builder** (Sépare la construction d'un objet complexe de sa représentation), **Prototype** (Permet de cloner des objets existants sans dépendre de leur classe)
- Les **Modèles de Structure (Structural Patterns)** concernent la **composition des classes et objets** pour former des structures plus grandes. Ce modèle est composé d' **Adapter** (Permet à des interfaces incompatibles de travailler ensemble), **Decorator** (Ajoute dynamiquement des responsabilités à un objet), **Proxy** (Fournit un substitut ou un espace réservé pour un autre objet), **Composite** (Traite des objets individuels et des compositions de manière uniforme), **Bridge** (Sépare une abstraction de son implémentation), **Facade** (Fournit une interface simplifiée à un système complexe) et **Flyweight** (Réduit l'utilisation de la mémoire en partageant des objets similaires)

- **Les Modèles de Comportement (Behavioral Patterns)** gèrent les **interactions et la communication entre objets**. Ce modèle est composé d'**Observer** (Notifie les objets des changements d'état), **Strategy** (Permet de changer d'algorithme à l'exécution), **Command** (Encapsule une requête sous forme d'objet), **State** (Permet à un objet de changer de comportement selon son état), **Chain of Responsibility** (Passe une requête à travers une chaîne de handlers), **Iterator** (Fournit un moyen d'accéder séquentiellement aux éléments d'une collection), **Mediator** (Réduit les dépendances entre objets en centralisant la communication), **Memento** : Capture et restaure l'état interne d'un objet), **Template Method** (Définit le squelette d'un algorithme dans une méthode) et **Visitor** (Sépare un algorithme de la structure d'objets sur laquelle il opère)

Pour ce devoir j'ai choisi trois composant de chacun des design patterns tels que:

- **Singleton (Création)**
- **Adapter (Structure)**
- **Observer (Comportement)**

Et j'ai choisi de les adapter à un mini projet de **Système de Notification Intelligent** pour montrer les solutions.

MINI-PROJET : SYSTÈME DE NOTIFICATION INTELLIGENT

- **Contexte** : Une application qui envoie des notifications (SMS, Email, Push) avec des règles dynamiques.

Problème du système:

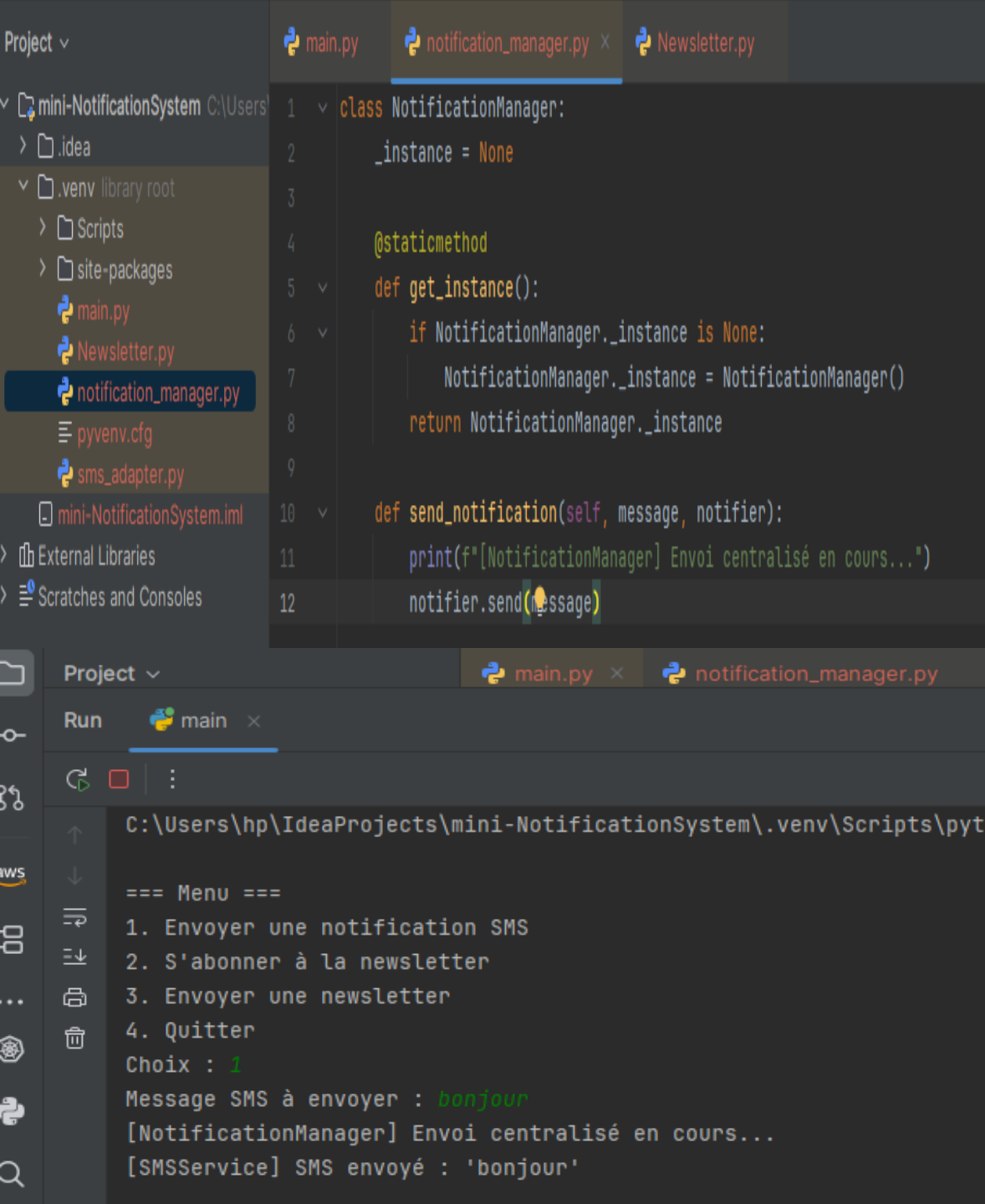
- *Comment éviter d'avoir plusieurs instances du gestionnaire de notifications, ce qui pourrait causer des incohérences (ex: notifications en double, conflits de ressources) ?*
- *Comment intégrer un service externe (ex: SMSService) qui a une interface incompatible avec le reste du système ?*
- *Comment notifier automatiquement des utilisateurs quand un événement se produit, sans créer un couplage fort entre l'émetteur et les récepteurs ?*

SOLUTION AUX PROBLEMES DU SYSTEME

• 1. Singleton

Pourquoi?

- **Garantit une seule instance** du gestionnaire :
- Toutes les notifications passent par un **point unique**, ce qui :
 - Évite les duplications (risque de spam).
 - Centralise le contrôle (ex: logs, taux limite d'envoi).

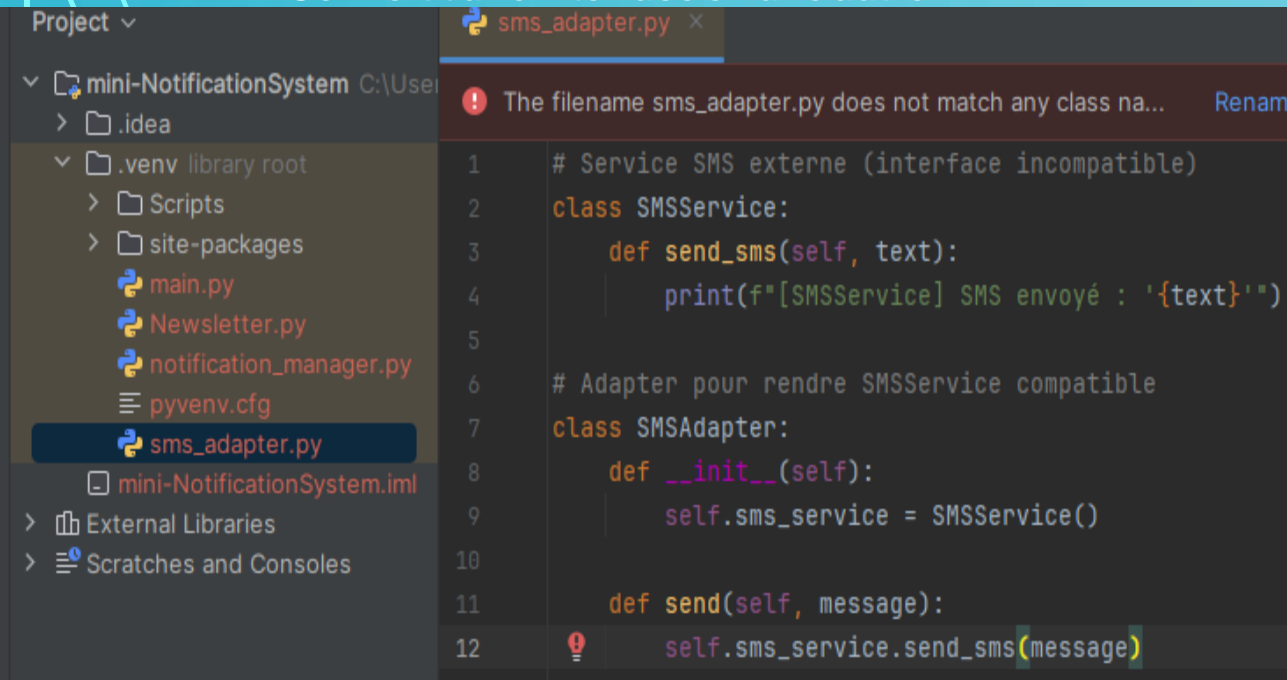


- La class NotificationManager comme sur la capture implémente le **design pattern Singleton** :
- On a une variable **statique** partagée par toutes les instances de la classe qui est initialisé à None
- On a une méthode statique qu'on peut l'appeler sans avoir besoin d'une instance et Vérifie si _instance est None (premier appel)
 - Si oui : crée une nouvelle instance (NotificationManager())
 - Sinon : retourne l'instance existante
- Garantit qu'il n'y a qu'une seule instance de la classe
- La méthode send_notification() permet d'affiche un message de log, de délègue l'envoi à l'objet notifier (qui doit avoir une méthode send()) et le self fait référence à l'instance unique
- Le résultat du code en exécution est démontré dans la capture d'écran du bas

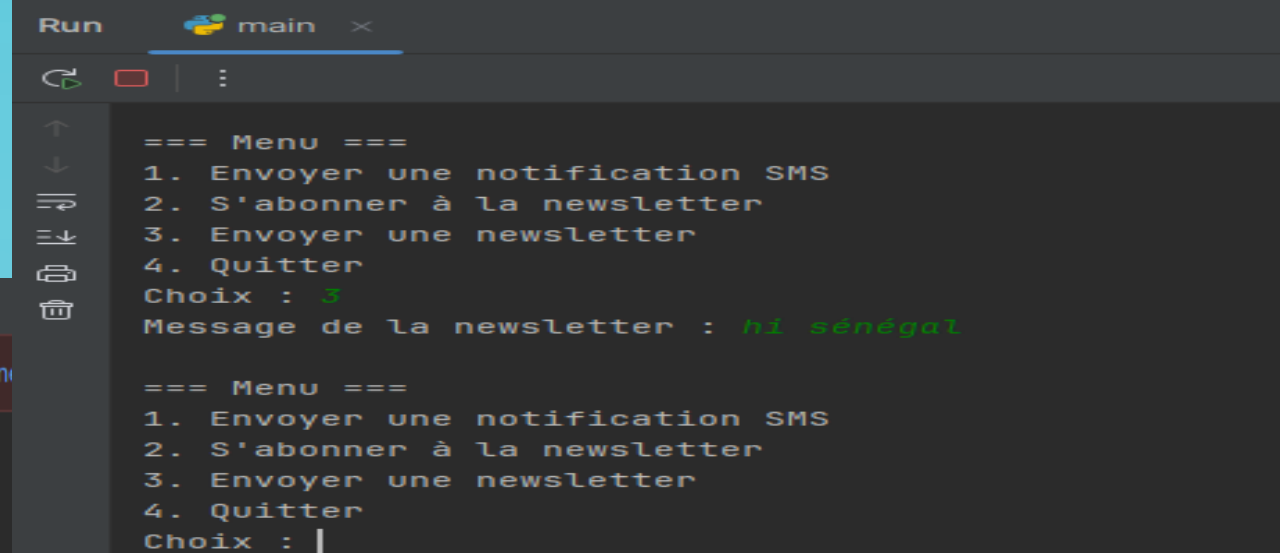
• 2. Adapter

Pourquoi?

- Convertit une interface en une autre



```
1 # Service SMS externe (interface incompatible)
2 class SMSService:
3     def send_sms(self, text):
4         print(f"[SMSService] SMS envoyé : '{text}'")
5
6 # Adapter pour rendre SMSService compatible
7 class SMSAdapter:
8     def __init__(self):
9         self.sms_service = SMSService()
10
11     def send(self, message):
12         self.sms_service.send_sms(message)
```



```
Run main x
=== Menu ===
1. Envoyer une notification SMS
2. S'abonner à la newsletter
3. Envoyer une newsletter
4. Quitter
Choix : 3
Message de la newsletter : hi sénégal

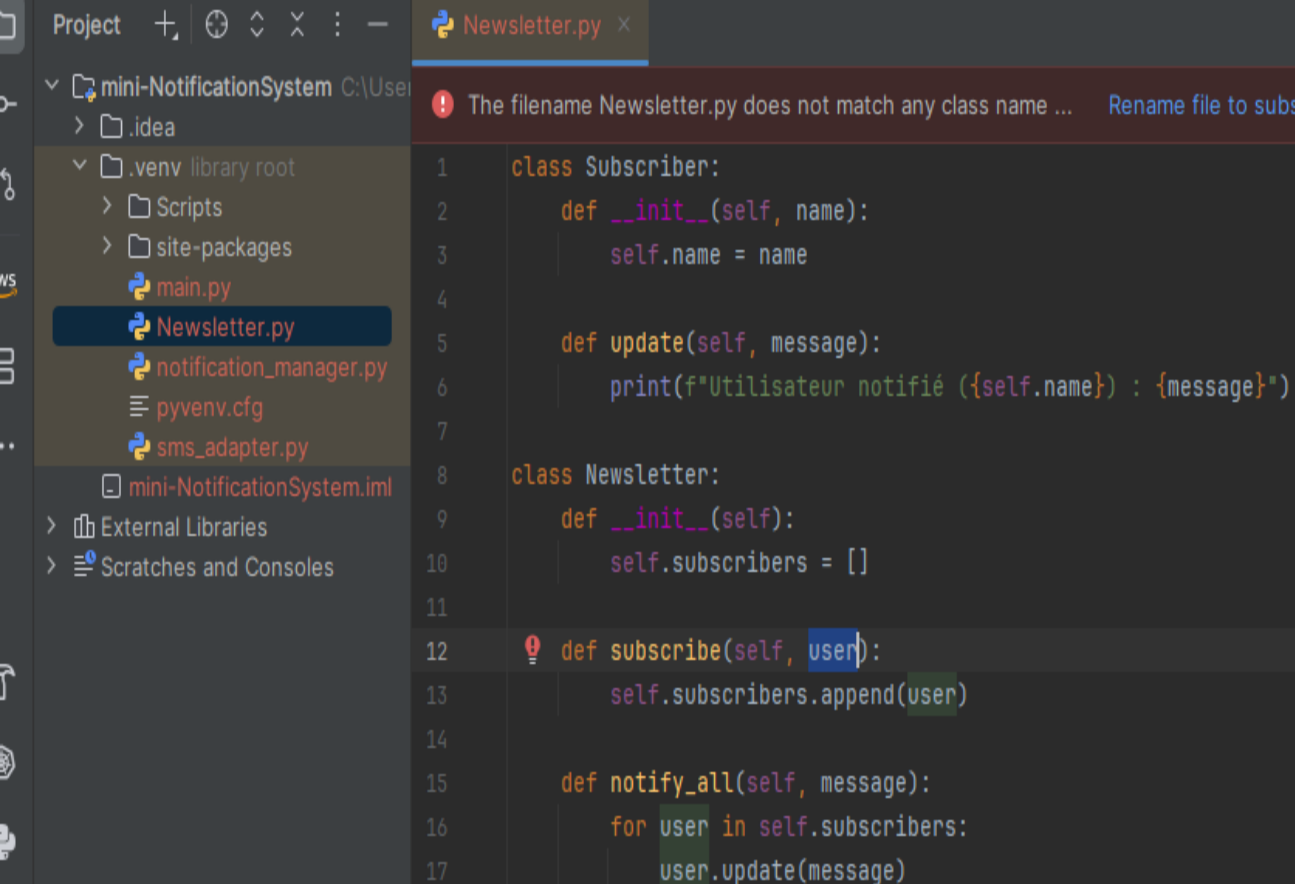
=== Menu ===
1. Envoyer une notification SMS
2. S'abonner à la newsletter
3. Envoyer une newsletter
4. Quitter
Choix : |
```

- La class `SMSService` comme sur la capture implémente le **design pattern Adapter**:
- Le Problème on est que cette classe a une interface spécifique (`send_sms()`) qui ne correspond pas à l'interface standard attendue par le système (`send()`) mais elle sera initialisée par l'adaptateur `SMSAdapter` qui va créer une instance du service SMS qu'il va adapter
- Et le Mécanisme d'adaptation va expose une méthode `send()` standard va Traduire l'appel vers la méthode spécifique `send_sms()`
- Le résultat est visible sur la capture d'écran ci-dessus par ce que le message une fois envoyé, on revient sur le menu principal

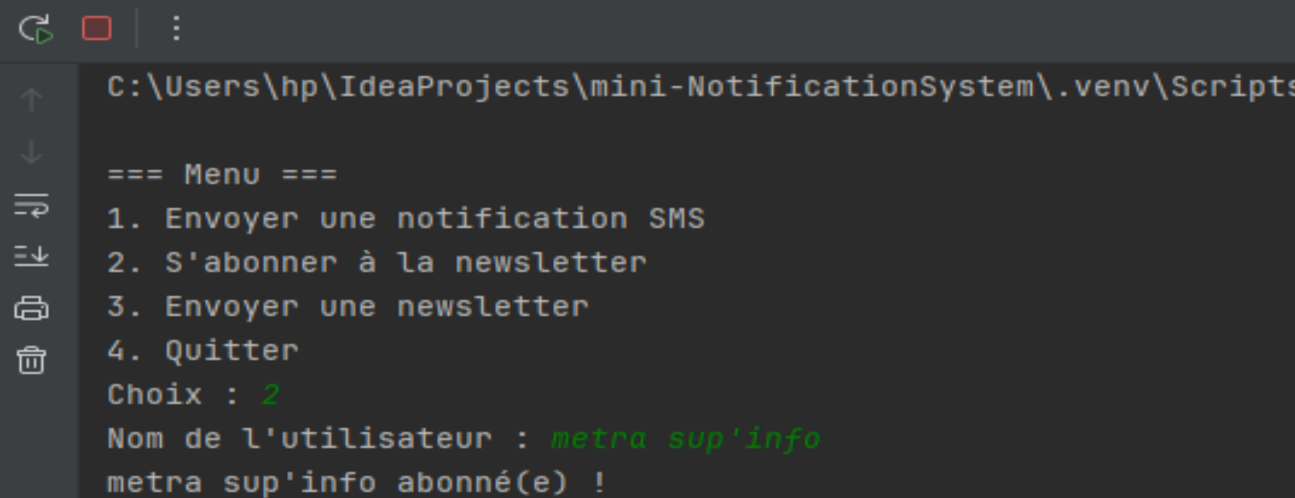
• 3. Observer

Pourquoi?

- Les utilisateurs reçoivent automatiquement les messages sans polluer le gestionnaire
- Permet des notifications **temps réel**.

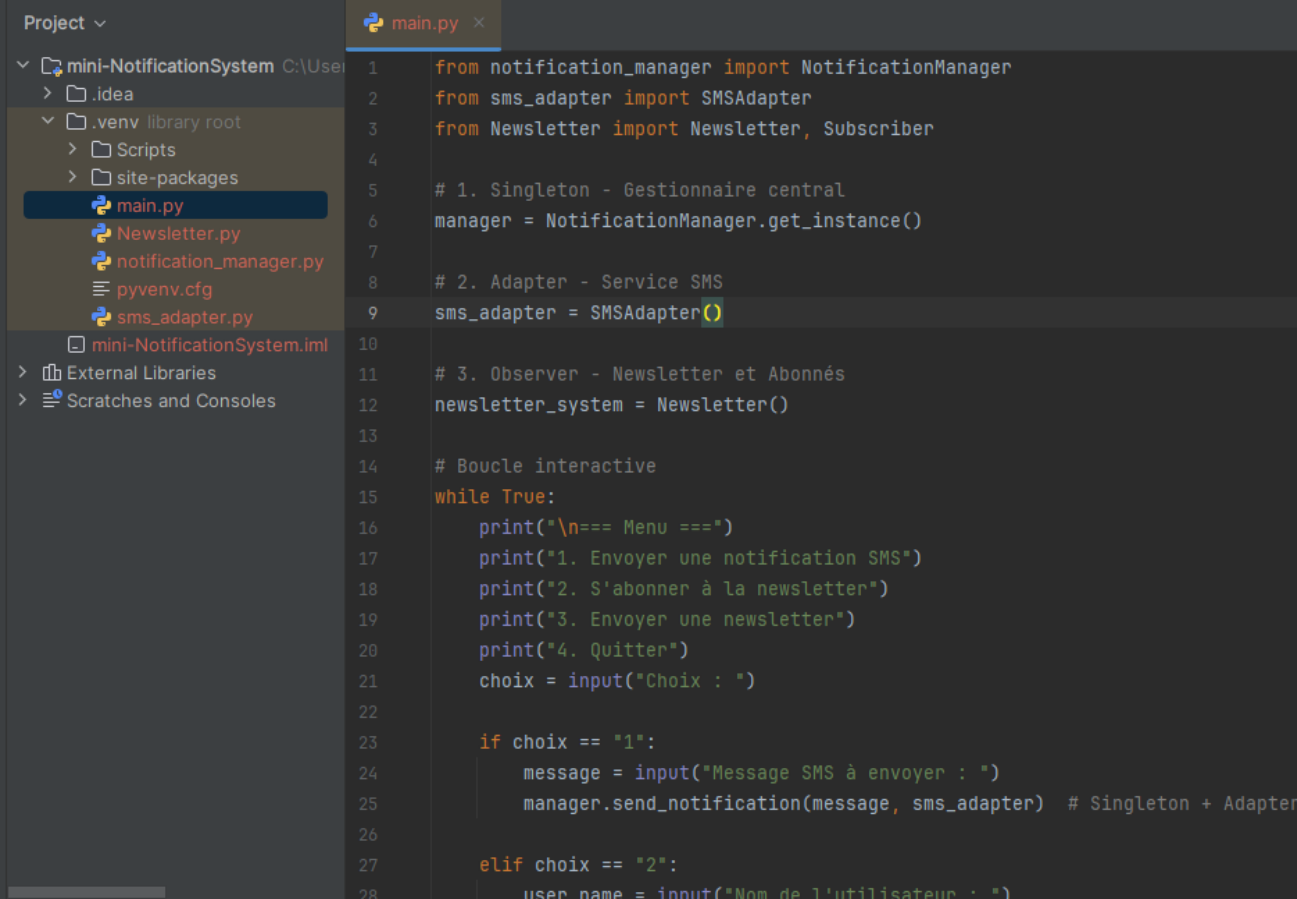


```
1 class Subscriber:
2     def __init__(self, name):
3         self.name = name
4
5     def update(self, message):
6         print(f"Utilisateur notifié ({self.name}) : {message}")
7
8 class Newsletter:
9     def __init__(self):
10         self.subscribers = []
11
12     def subscribe(self, user):
13         self.subscribers.append(user)
14
15     def notify_all(self, message):
16         for user in self.subscribers:
17             user.update(message)
```

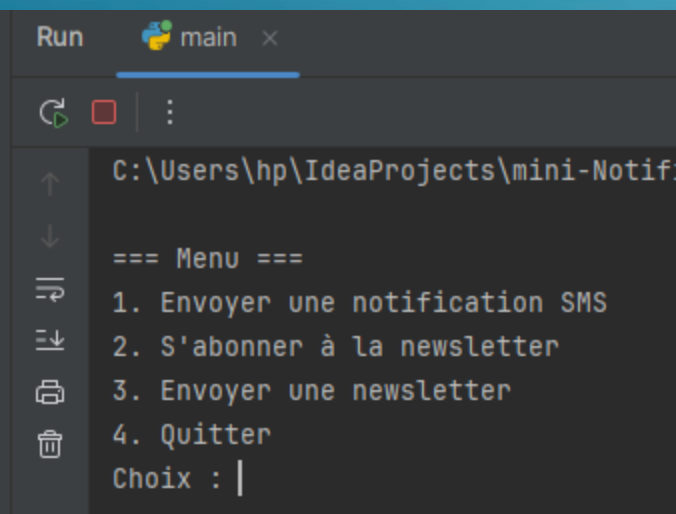


```
=== Menu ===
1. Envoyer une notification SMS
2. S'abonner à la newsletter
3. Envoyer une newsletter
4. Quitter
Choix : 2
Nom de l'utilisateur : metra sup'info
metra sup'info abonné(e) !
```

- La class Subscriber comme sur la capture implémente le **design pattern Observer**:
- Cette class représente un abonné qui reçoit des notifications
- le `__init__` est un constructeur qui initialise l'abonné avec un nom
 - `self.name` : Variable d'instance pour stocker le nom
- La `update` est une méthode appelée quand une notification arrive et permet d'afficher le message avec le nom de l'abonné
- La class Newsletter (Le Sujet Observable) permet d'initialiser `self.subscribers` : Liste qui contiendra tous les abonnés
- Le `subscribe` ajoute un nouvel abonné (Subscriber) à la liste et Paramètre `user` en une instance de Subscriber à ajouter
- Le `notify_all` parcourt tous les abonnés, appelle leur méthode `update()` avec le message et chaque abonné traite le message à sa façon
- Le résultat est montré sur la capture d'écran en dessous



```
1 from notification_manager import NotificationManager
2 from sms_adapter import SMSAdapter
3 from Newsletter import Newsletter, Subscriber
4
5 # 1. Singleton - Gestionnaire central
6 manager = NotificationManager.get_instance()
7
8 # 2. Adapter - Service SMS
9 sms_adapter = SMSAdapter()
10
11 # 3. Observer - Newsletter et Abonnés
12 newsletter_system = Newsletter()
13
14 # Boucle interactive
15 while True:
16     print("\n=== Menu ===")
17     print("1. Envoyer une notification SMS")
18     print("2. S'abonner à la newsletter")
19     print("3. Envoyer une newsletter")
20     print("4. Quitter")
21     choix = input("Choix : ")
22
23     if choix == "1":
24         message = input("Message SMS à envoyer : ")
25         manager.send_notification(message, sms_adapter) # Singleton + Adapter
26
27     elif choix == "2":
28         user_name = input("Nom de l'utilisateur : ")
```



```
Run main
C:\Users\hp\IdeaProjects\mini-Notif:
=== Menu ===
1. Envoyer une notification SMS
2. S'abonner à la newsletter
3. Envoyer une newsletter
4. Quitter
Choix : 1
```

Le main.py est la struture du menu d'affichage visuel, il comporte:

- NotificationManager : Classe Singleton pour gérer les notifications
- SMSAdapter : Adapteur pour le service SMS
- Newsletter/Subscriber : Implémentation du pattern Observer
- **L'initialisation des Composants:**
 - Singleton : Une seule instance du gestionnaire
 - Adapter : Prêt à convertir les messages pour le service SMS
 - Observer : Système de newsletter initialisé
- **Boucle Principale (Menu Interactif):**
 - Option 1 : Envoyer un SMS
 - Option 2 : S'abonner
 - Option 3 : Envoyer une Newsletter
 - Option 4 : Quitter

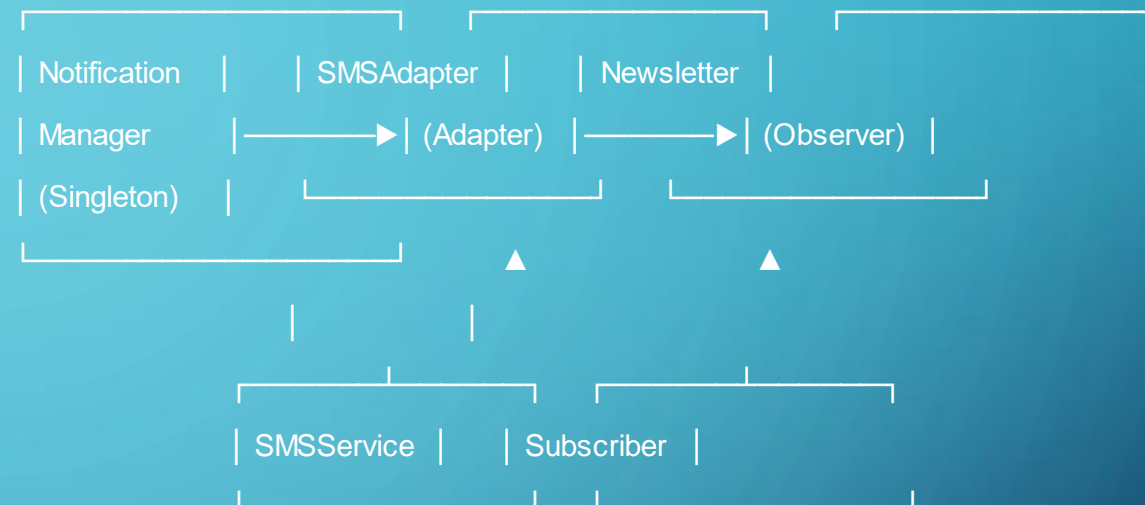
• Le résultat est démontré sur la capture d'écran du bas

CONCLUSION

Avantages de Ce Projet

- ✓ Simple mais couvre 3 Design Patterns.
- ✓ Montre l'utilité de chaque pattern :
 - Singleton → Contrôle centralisé.
 - Adapter → Intégration de services externes.
 - Observer → Communication événementielle.

Schéma d'Architecture



Résumé du Projet

Design Pattern	Rôle	Fichier
Singleton	Gestion centralisée	`notification_manager.py`
Adapter	Convertit SMS → Interface commune	`sms_adapter.py`
Observer	Notifications événementielles	`Newsletter.py`