Design Pattern Observateur

# Présentation

## Introduction

Le développement de logiciels orientés objet repose sur la communication entre différents objets. Cependant, il peut être difficile de maintenir cette communication de manière efficace et flexible. C'est là qu'intervient le design pattern Observateur.

## Définition

Le design Pattern Observateur (Observer) est un patron de conception qui permet à un objet d’envoyer des notifications à plusieurs objets qui l’observent lors de la création d’évènements. Il fait partie des patrons comportementaux qui mettent en place une communication efficace et répartissent les responsabilités entre les objets. Son objectif est d’éviter des communications inutiles entre objets, et aussi de bien cibler les objets qui ont besoin de recevoir l’information.

## Problème

Le client d’un magasin attend la sortie d’un nouveau produit. Il pourrait faire des allers-retours tous les jours pour savoir si le produit en question est déjà disponible à la vente. Mais cela pourrait rapidement devenir fastidieux si le produit met du temps à être disponible et couteux en temps.

Le magasin pourrait, quant à lui, envoyer des notifications à tous ses clients pour les informer de la disponibilité de ce nouveau produit. Cette solution économiserait bien des voyages à leurs clients. En contrepartie, le magasin risque de se mettre à dos ceux qui ne sont pas intéressés par le nouveau produit.

Une image contenant capture d’écran, texte, graphisme, Graphique

Description générée automatiquement

La situation devient compliquée : soit les clients perdent leur temps, soit le magasin gâche des ressources pour prévenir des clients qui ne sont pas concernés.

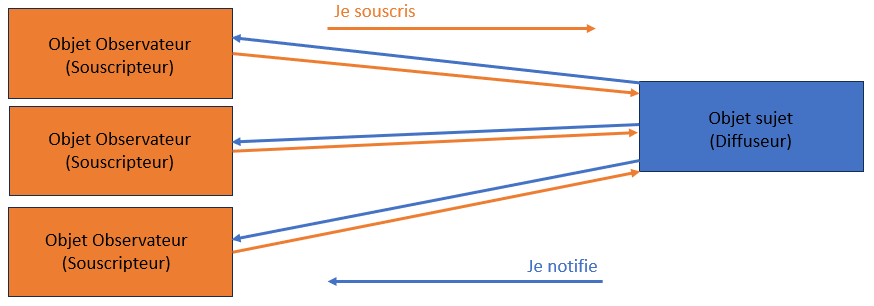
# Composants et fonctionnement

## Composants

* Le sujet (diffuseur) : l’objet que l’on veut suivre et qui va envoyer des notifications aux autres objets.
* L’observateur (souscripteur) : l’objet qui souhaite être notifié des changements d’état du sujet.
* La souscription : la méthode qui permet aux observateurs de s’inscrire ou de se désinscrire auprès du sujet.
* La notification : la méthode par lequel le sujet diffuse l’information d'un changement d’état auprès des observateurs.

## Principe de fonctionnement

Quand un changement d’état intervient chez l’objet sujet (diffuseur), il fait le tour de ses objets observateurs (souscripteurs) et appelle la méthode de notification pour diffuser l’information auprès de ces derniers.



# Mise en œuvre

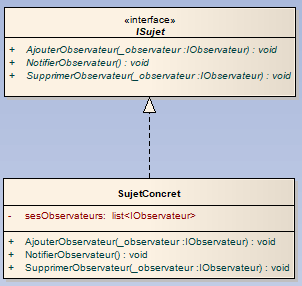
## Les sujets concrets et leur interface

Pour ne pas coupler l’objet sujet (diffuseur) aux dizaines d’objets observateurs (souscripteurs) qui veulent être tenus au courant d’un changement d’état de l’objet sujet (diffuseur), on va implémenter une interface qui va être le seul moyen utilisé par l’objet sujet (diffuseur) pour communiquer avec eux.

De plus, les objets sujet (diffuseurs) doivent tous suivre la même interface pour que tous les objets observateurs (souscripteur) soient tous compatibles avec eux. Cette interface doit contenir quelques méthodes de souscription et elle doit permettre aux objets observateurs (souscripteurs) d’observer les états du diffuseur sans le coupler avec leurs classes concrètes.

L’interface Sujet déclare les méthodes d’ajout de nouveaux observateurs, de suppression d’observateurs actuels et de notification aux observateurs ajoutés. Elle envoie des évènements à d’autres objets quand l’objet sujet change d’état.

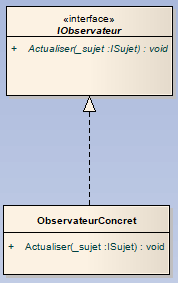
L’objet Sujet Concret implémente les méthodes héritées de l’interface Sujet, et y ajoute la liste des Observateurs.



## Les observateurs concrets et leur interface

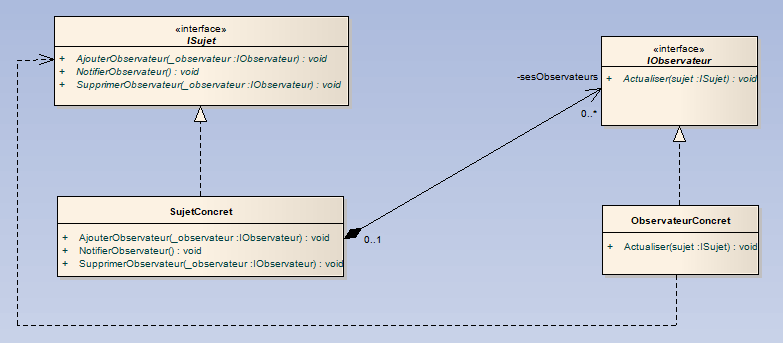
L’interface Observateur déclare la méthode d’actualisation qui va permettre de se mettre à jour en fonction de la notification reçue.

L’objet Observateur Concret implémente les méthodes héritées de l’interface Observateur.



## Enregistrement des Observateurs auprès des Sujets

Il faut maintenant mettre en relation le groupe Sujet et le groupe Observateur. Diagramme de classe patron de conception Observateur.



# Avantages, inconvénients

## Avantages

Séparation des préoccupations : Le Design Pattern Observateur facilite la séparation entre le sujet observable et les observateurs, permettant ainsi une meilleure modularité et une plus grande flexibilité.

Réutilisabilité du code : De nouveaux observateurs peuvent être ajoutés facilement sans modifier le code existant du sujet.

Mise à jour dynamique : Les observateurs sont notifiés automatiquement lorsqu'un changement d'état se produit, ce qui garantit une mise à jour dynamique et cohérente des objets dépendants.

## Inconvénients

Les souscripteurs sont avertis dans un ordre aléatoire.

# Exemple d’utilisation

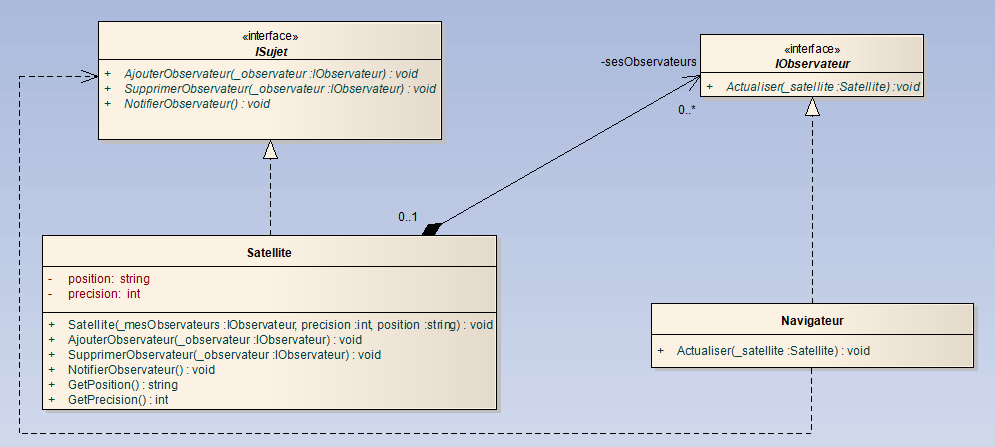
## Contexte

Une personne souhaitant connaître sa position utilise un récepteur GPS. Ce récepteur reçoit des informations d’un satellite et peut alors calculer sa position.

Nous pouvons déjà identifier deux acteurs :

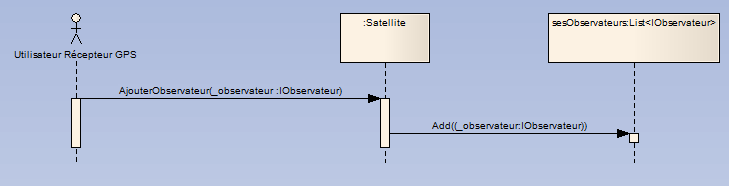
* Le satellite qui va être l’objet Sujet (Diffuseur) qui va hériter de l’interface ISujet
* Le récepteur GPS qui va être l’objet Observateur (Souscripteur) qui va hériter de l’interface IObservateur

## Diagramme UML

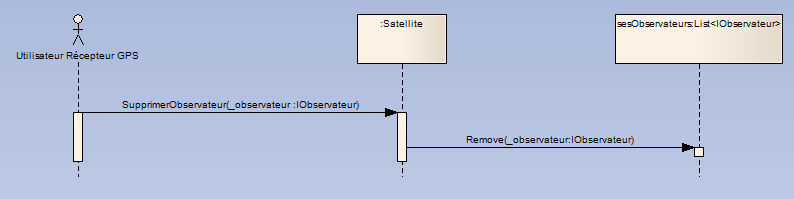


## Implémentation

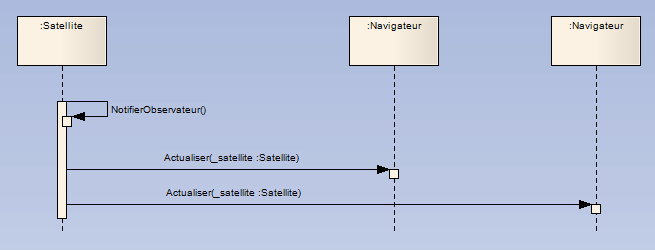
### Diagramme de séquence pour l’ajout d’un Observateur



### Diagramme de séquence pour la suppression d’un Observateur



### Diagramme de séquence pour la notification d’un Observateur



### Implémentation interface ISujet

|  |  |
| --- | --- |
|  |  |

### Implémentation classe Satellite

|  |  |
| --- | --- |
|  |  |

### Implémentation interface IObservateur

|  |  |
| --- | --- |
|  |  |

### Implémentation classe Navigateur

|  |  |
| --- | --- |
|  |  |

# Conclusion

Le design pattern Observer est un outil essentiel pour faciliter la communication entre objets dans les applications orientées objet. En permettant aux objets de s'observer mutuellement et de réagir aux changements d'état, ce modèle de conception favorise la modularité, l'extensibilité et la réactivité des systèmes logiciels. En comprenant les concepts fondamentaux du design pattern Observer et en l'appliquant de manière appropriée, les développeurs peuvent améliorer la qualité de leur code et simplifier la maintenance de leurs applications.