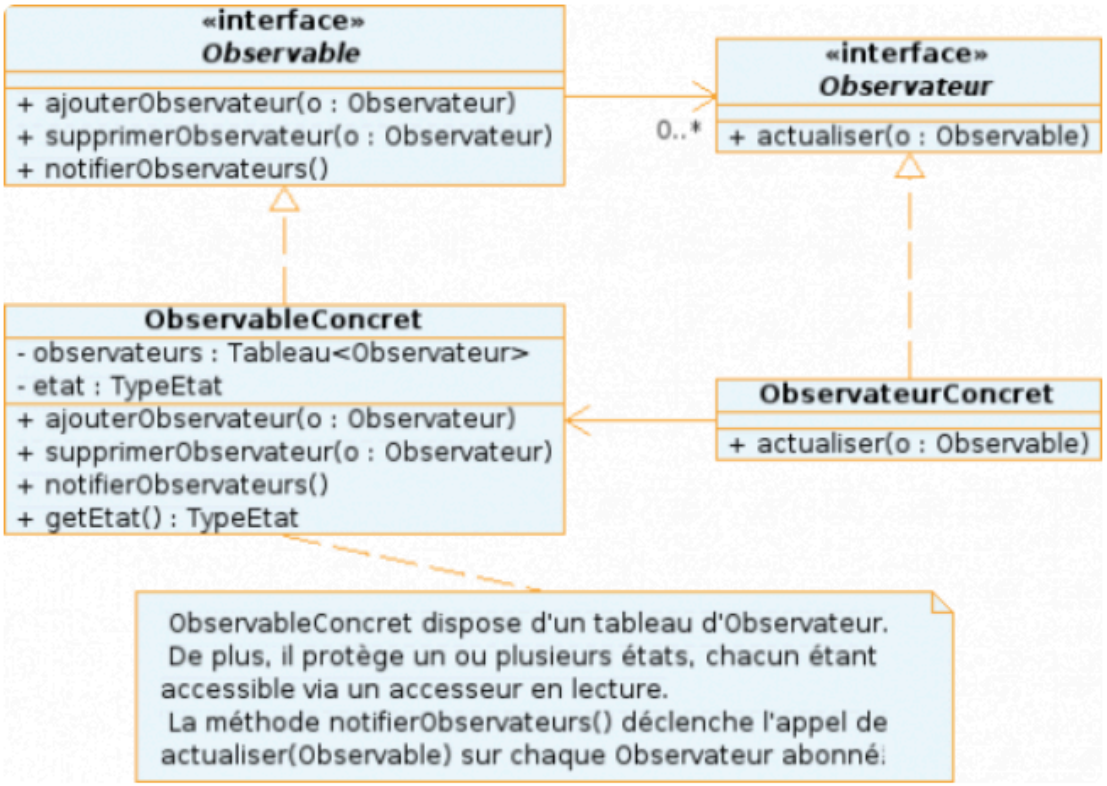
Fiche Résumé

Intention métier :

Le pattern Observateur a été créé et formalisé par le GoF parce qu'ils ont voulu répondre à ce besoin spécifique que nous avons pu comprendre dans le 1er exemple et que nous illustrerons encore par la suite. C'est-à-dire, le besoin de définir une dépendance un-à-plusieurs entre des objets pour que, lorsque l'un change d'état, tous les autres soient automatiquement notifiés et mis à jour, sans couplage fort.

Diagramme des classes :



Classes participantes :

Donc il définit deux interfaces et deux classes participantes. L'interface Observateur sera implémenté par toutes classes qui souhaitent avoir le rôle d'observateur. C'est le cas de la classe ObservateurConcret qui implémente la méthode actualiser(Observable).
Cette méthode sera appelée automatiquement lors d'un changement d'état de la classe observée. On trouve également une interface Observable qui devra être implémentée par les classes désireuses de posséder des observateurs. La classe ObservableConcret implémente cette interface, ce qui lui permet de tenir informer ses observateurs. Celle-ci possède en attribut un état (ou plusieurs) que les observateurs vont vouloir suivre, et un tableau d'observateurs qui correspond à la liste des observateurs qui sont à l'écoute. En effet, il ne suffit pas à une classe d'implémenter l'interface Observateur pour être à l'écoute, il faut qu'elle s'abonne à un Observable via la méthode ajouterObservateur(Observateur).
La classe ObservableConcret peut donc **ajouterObservateur(Observateur), supprimerObservateur(Observateur), notifierObservateurs()** et possède une méthode getEtat() qui permet d'obtenir la valeur de l'état.

Exemple contexte métier :

La mise en situation est que sur un site e-commerce, un `Produit` a un stock qui change. Lorsqu'une vente est effectuée, plusieurs parties du système doivent réagir :

1. Le site web doit afficher "Rupture de stock" si le stock tombe à zéro.
2. Le service logistique doit être notifié si le stock passe sous un seuil critique pour commander à nouveau.
3. Le service marketing veut envoyer une alerte "De retour en stock" aux clients en liste d'attente.

La modélisation bancale sans le pattern, la classe `Produit` devrait connaître directement les classes `ServiceLogistique`, `AfficheurSiteWeb` et `ServiceMarketing`. La méthode `effectuerVente()` deviendrait un "bloc de couplage fort", illisible et impossible à maintenir.

La Solution avec le pattern Observer :

- Le Sujet : La classe `Produit`.
    - Elle possède l'état
    - Elle implémente `Sujet` et gère une liste d'`Observateurs`.
    - Sa méthode `effectuerVente()` modifie son stock et appelle `notifierObservateurs()`.
- Les Observateurs :

    - `AfficheurSiteWeb` : Implémente `Observateur`. Son `actualiser()` met à jour l'affichage.
    - `ServiceLogistique` : Implémente `Observateur`. Son `actualiser()` vérifie si le stock est sous le seuil critique.
    - `ServiceNotificationClient` : Implémente `Observateur`. Son `actualiser()` vérifie si le produit est de retour et envoie les e-mails.

Le bénéfice de la classe `Produit` est totalement découplée des systèmes qui l'observent. On peut ajouter un `ServiceAnalytics` demain (qui analyse les ventes en temps réel) sans jamais avoir à modifier une seule ligne de code de la classe `Produit`.

La SOLIDité du pattern :

S - Responsabilité Unique

- Avant : Le Canard avait deux jobs : être un canard ET gérer les statistiques (il connaissait le Comptable).
- Après : Chacun son métier. Le Canard (Sujet) s'occupe de sa vie de canard. Le Comptable (Observateur) s'occupe de compter. Les responsabilités sont parfaitement séparées.

O - Ouvert/Fermé

- Fermé : La classe Canard est terminée. On n'a plus jamais besoin de la modifier.
- Ouvert : On peut étendre le système à l'infini. On peut ajouter un chasser, etc… On crée une nouvelle classe Observateur, on l'abonne, et ça marche.

D - Inversion de Dépendance

- Avant : Le Canard (classe "métier") dépendait directement du Comptable (détail technique).
- Après : On inverse tout. Le Canard ne dépend plus du Comptable. À la place, les deux dépendent d'une abstraction (l'interface Observateur). Le lien direct est brisé, le code est découplé.