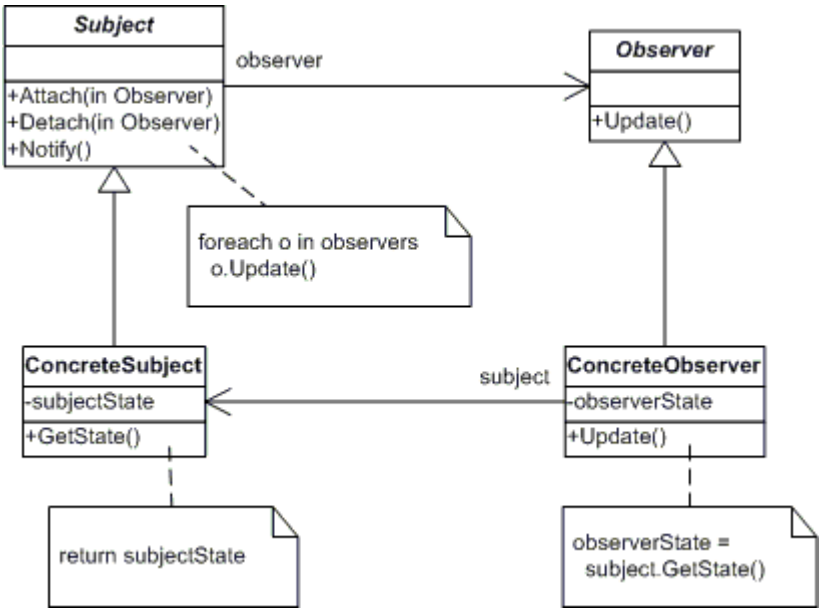
TD n°2 Design Pattern

I – Le Design Pattern Observer



Intention :

- Définir une dépendance 1-plusieurs entre un objet **o** et d'autres objets : si l'objet **o** change d'état, tous les objets qui en dépendent en sont informés.

Motivation :

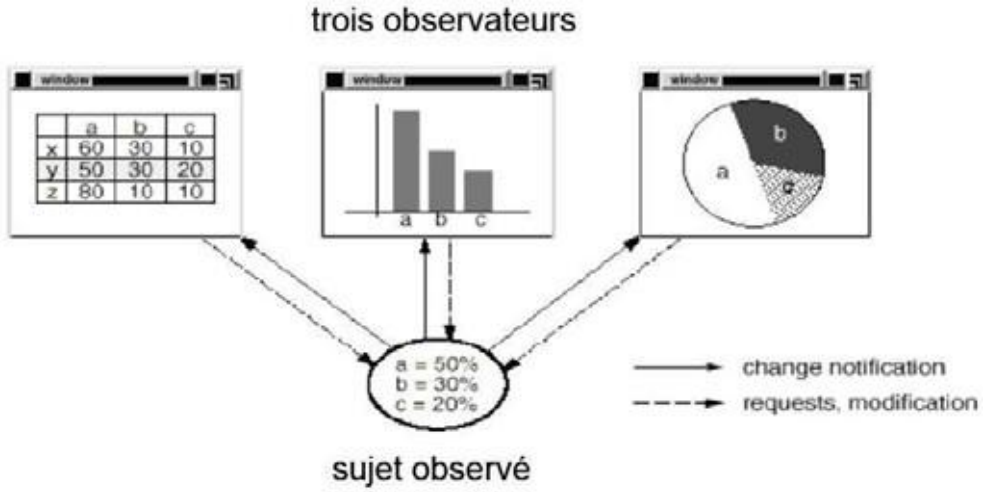
- La nécessité de maintenir la consistance entre des objets reliés sans pour autant trop coupler les classes.

Participants :

- **Le Sujet (Subject)**
  - garde une trace de ses observateurs
  - propose une interface pour **ajouter** ou **enlever** des observateurs
- **L'Observateur (Observer)**
  - définit une interface pour la notification de mises à jour
- **Le sujet concret (ConcreteSubject)**
  - C'est l'objet observé
  - Enregistre son état intéressant les observateurs
  - Envoie une notification à ces observateurs quand il change d'état
- **L'Observateur concret (ConcreteObserver)**
  - Il observe l'objet concret
  - Il stocke l'état qui doit rester consistent avec celui de l'objet observé
  - Il implémente l'interface de mise à jour permettant à son état de rester consistent avec celui de l'objet observé

Question 1 : Notions de vue - modèle

L'interface utilisateur est chargée de représenter, sous une forme interprétable par un humain, les informations internes de l'application. Ces informations constituent le **modèle** de l'application (son **état** actuel).
On peut créer plusieurs **vues** distinctes d'un même modèle.



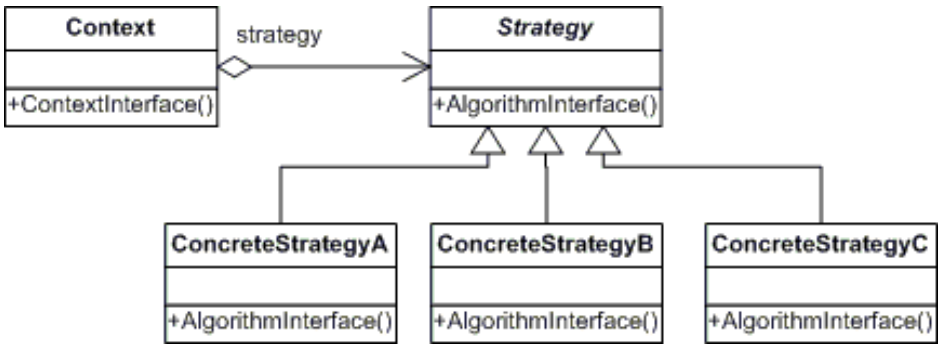
Les observables (ou objets observés) envoient des notifications à leurs observateurs en cas de changement. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis le module qu'ils observent (l'observable).

Proposez une modélisation UML de la relation vue-modèle, en considérant les vues comme des observateurs du modèle.

II – Le Design Pattern Strategy

Le modèle de conception stratégie est un modèle de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours du temps d'exécution selon certaines conditions.

- Le patron « Strategy » cherche principalement à séparer un objet de ses comportements/algorithmes en encapsulant ces derniers dans des classes à part.
- Pour ce faire, on doit alors définir une famille de comportements ou d'algorithmes encapsulés et interchangeables.



- **Strategy:**
  - L'interface commune de toutes les stratégies.
  - Utilisé par Context pour appelé une stratégie concrète.
- **ConcreteStrategy:**
  - Utilise l'interface Strategy pour implémenter un algorithme.
- **Context:**
  - Maintient une référence à un objet Strategy.
  - Est configuré avec une ConcreteStrategy.

Question 1 : Notion de vue-contrôleur

Pour disposer de moyens d'interaction avec le modèle, on se dote de contrôleurs associés aux vues. Les contrôleurs réagissent aux événements extérieurs interceptés par la vue (clics sur des boutons, frappes clavier...) et modifient le modèle en réponse à ces événements. **La manière dont un contrôleur agit sur le modèle dépend de la vue à laquelle il est associé.** Ainsi, les contrôleurs sont semblables à des algorithmes qui, en fonction du contexte défini par la vue, déclencheront des opérations différentes.

Utilisez le patron « strategy » pour proposer une modélisation du lien entre vues et contrôleurs.

Question 2 : Algorithmes de tri

Concevez une application en UML permettant de trier les éléments d'un tableau en choisissant l'algorithme de tri utilisé : tri rapide, tri à bulle ou tri par sélection. Il s'agit ici de laisser ouvert le choix de la méthode de tri à utiliser...

Modélisez cette situation en utilisant le pattern **Strategy**.