TD/TP #4 - Patron de conception Observateur

Guillaume Santini

16 novembre 2023

1 Exercice Principal : Actualisation des prix affichés d'une liste de produits lorsque le prix d'un produit change

1.1 Description du sujet

L'objectif de cet exercice est d'implémenter une classe `ListeProduits` proposant udotée d'une fonctionnalité permettant d'afficher la liste des `Produit`s et leurs prix.

La liste d'`Produit`s est déterminée lors de l'exécution et peut être modifiée dynamiquement (ajout ou suppression d'produits).

L'affichage proposé par la `ListeProduits` doit être mise jour automatiquement lors de la modification du prix d'un des `Produit`s contenu dans la liste.

1.2 Modélisation UML

Exercice 1 : Modélisation des entités de base

Les deux entités de base de cette modélisation sont les `Produit`s et les `ListeProduits`. Ces entités doivent suivre les spécifications suivantes :

- Une instance d' `Produit` est décrite :
  - par une dénomination (*i.e.* `"Velo"` ou `"Chaussures"`) qui est non modifiable après l'instanciation et
  - par un prix (modifiable).
- Une instance de `ListeProduits` est :
  - décrite par une dénomination (*i.e.* `Stock` ou `Panier`) qui est non modifiable après l'instanciation et est
  - composée d'un listing de produits.
- Une dénomination est décrite par une chaîne de caractère dont la longueur ne doit pas dépasser 30 caractères. Si la dénomination dépasse cette taille seuil alors une `DenominationException` est levée.
- Un prix ne peut être négatif. Si l'on cherche à fixer un prix négatif alors une `PrixException` est levée.
- Chaque classe est dotée des méthodes classiques (setters et getter) et d'une redéfinition de la méthode `toString()` de façon à proposer une chaîne de caractère formatée décrivant l'état de l'instance (d'`Produit` ou de `ListeProduits`).
- La classe `Produit` est dotée d'un seul constructeur champ à champ `Produit(String denom, Double prix)`,
- La classe `ListeProduits` est dotée de deux constructeurs :
  - le premier, `ListeProduits( String denom)`, prenant pour seul paramètre la dénomination et initialisant sa liste de produits comme une liste vide,
  - le second, `ListeProduits( String denom, ArrayList<Produit> listing)`, prenant en paramètre la dénomination et une liste de produits préformée permettant d'initialiser le contenu de la liste.

Exercice 2 : Modélisation des mécanismes de mise à jour de l'affichage

Dotez la classe `ListeProduits` d'une méthode `affichage()` permettant d'afficher le contenu courant des produits (dénomination et prix).
Augmentez le modèle des entités de base en ajoutant les éléments nécessaires pour que lorsque le prix d'un `Produit` change alors l'affichage de la liste des produits soit rafraichie (l'affichage et provoqué par tout changement de prix).

1.3 Implémentation Java

Exercice 3 : Implémentation des classes des entités du modèle

Dans un premier temps implémentez le modèle correspondant à l'exercice (1).
Vous implémenterez les classes d'entité et les Exceptions et testerez votre implémentation avec une classe `Main`.

Exercice 4 : Implémentation des classes des entités du modèle

Dans un second temps vous ajouterez à votre implémentation les éléments nécessaires à la mise à jour automatique de l'affichage des listes lorsque le prix d'un produit change.

Exercice 5 : Exécution du modèle

Proposez un programme `Main.main()` principal qui :

- intialise une variable `stock` de type `ListeProduits` contenant :
  - un "Velo" à 200,00€
  - un "Velo" à 40,00€
  - des "Chaussures running" à 40,00€
  - des "Chaussures escalade" à 40,00€
  - des "Complement alim." à 90,00€
  - une "Tente" à 120,00€
- affiche la variable `stock`,
- intialise une variable `panier` de type `ListeProduits` contenant :
  - l'élément en position d'indice 3 du stock (les "Chaussures escalade"),
  - l'élément en position d'indice 5 du stock (la "Tente"),
- affiche la variable `panier`,
- *Modifie le prix des "Chaussures escalade" à 25,00€.*

Les affichage produits devraient correspondre à la trace suivante :

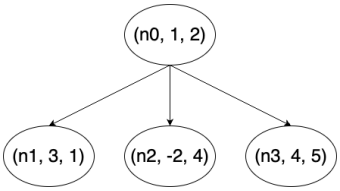
```
Stock :
                        Velo      200,00
                      Ballon       40,00
           Chaussures running       40,00
          Chaussures escalade       40,00
             Complement alim.       90,00
                       Tente      120,00

Panier perso :
          Chaussures escalade       40,00
                       Tente      120,00

-------------------------------------------------
Modification d'un prix
-------------------------------------------------
Stock :
                        Velo      200,00
                      Ballon       40,00
           Chaussures running       40,00
          Chaussures escalade       25,00
             Complement alim.       90,00
                       Tente      120,00

Panier perso :
          Chaussures escalade       25,00
                       Tente      120,00
```

2 Exercice Optionnel : Propagation de mise à jour dans un arbre

2.1 Description du sujet

On souhaite gérer un arbre orienté dont les valeurs des nœuds sont automatiquement mises à jour lorsqu'on modifie la valeur du nœud racine.

Chaque nœud est représenté par un triplet `(nom,valeur,coefficient)`.



Chaque fois qu'un nœud reçoit une valeur `v` transmise par son nœud père, sa valeur devient $(valeur + v * coefficient)$. Le nœud transmet alors sa nouvelle `valeur` à ses nœuds fils.

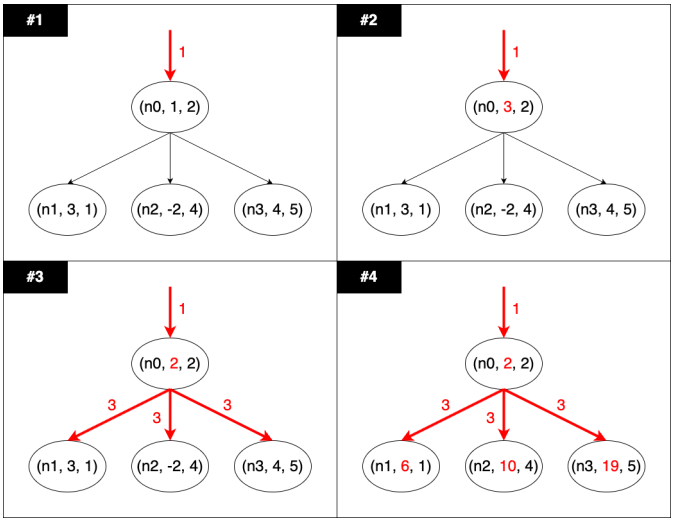
Par exemple, si le nœud racine `n0` de l'arbre reçoit (de l'extérieur) la valeur 1, sa valeur devient $1 + 1 * 2 = 3$.

La nouvelle valeur de `n0` (=2)est propagée à ses fils :

— la valeur de `n1` devient $3 + 3 * 1 = 6$,

— la valeur de `n2` devient $-2 + 3 * 4 = 10$,

— la valeur de `n3` devient $4 + 3 * 5 = 19$.

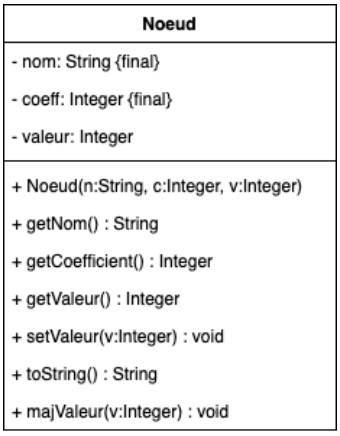


La propagation se poursuit de la même façon (à partir des fils `n1`, `n2` et `n3`) jusqu'aux nœuds feuilles de l'arbre.

2.2 Modélisation UML

Exercice 6 : Modélisation d'un arbre de profondeur 1

Soit la classe `Noeud` suivante :



Utilisez le patron de conception Observateur pour compléter ce diagramme de façon à pouvoir modéliser les arbres de profondeur 1 (*i.e.* uniquement constitués d'un unique nœud racine et d'un nombre variable de nœuds feuilles).

Exercice 7 : Modélisation d'un arbre de profondeur arbitraire

Adaptez le modèle pour pouvoir instancier des arbres de profondeur arbitraire. Dans cette situation il est à noter que les nœuds intermédaires doivent pouvoir être mis à jour comme les nœuds feuilles et doivent pouvoir propager à leurs fils leur nouvelle valeur comme le nœud racine.

2.3 Implémentation Java

Exercice 8 : Implémentation des arbres de profondeur 1

Traduisez le diagramme de classe UML (arbre de profondeur 1) en java en commençant par créer et tester la classe `Noeud`.

Exercice 9 : Implémentation des arbres de profondeur arbitraire

Traduisez le diagramme de classe UML (arbre de profondeur arbitraire). Vous proposerez un programme `Main` :main() : permettant d'implémenter le cas d'utilisation suivant :