TP Séance 8
Boisson, Friandise et Distributeur

On s'intéresse dans ce TP à la mise en place d'un système d'information de gestion de distributeurs (de boissons et de friandises). Vous trouverez dans le code qui vous est fourni la classe Produit. Un produit est défini par un identifiant, une marque, un prix et un nombre de calories.

Remarque : Au cours du TP, décommentez la partie correspondante de votre `main` afin de pouvoir tester votre implémentation.

1 Classe Friandise

On considère dans ce TP qu'une Friandise est un Produit ayant un poids.

Exercice 8 : Quelle est la relation qui lie les classes Produit et Friandise ?

Exercice 9 : Déterminez le/les attribut(s) propre(s) de la classe Friandise.

Exercice 10 : On considère que la classe Friandise propose les mêmes services que la classe Produit. Donnez la déclaration et l'implémentation de la classe Friandise.

2 Classe Boisson

On considère dans ce TP qu'une Boisson est un Produit ayant un volume.

Exercice 11 : Quelle est la relation qui lie les classes Produit et Boisson ?

Exercice 12 : Déterminez le/les attribut(s) propre(s) de la classe Boisson.

Exercice 13 : On considère que la classe Boisson propose les mêmes services que la classe Produit. Donnez la déclaration et l'implémentation de la classe Boisson.

3 Classe Distributeur

On considère ici qu'un Distributeur possède une caisse permettant de stocker le montant des transactions et un certains nombres de produits (friandises ou boissons). Vous trouverez ci-dessous un diagramme décrivant une ébauche de cette classe :

Distributeur
- m_montant_caisse : float - m_nombre_friandises : unsigned int - m_nombre_boissons : unsigned int - m_contenance_friandises : unsigned int - m_contenance_boissons : unsigned int - m_produits : map<unsigned int, vector<Produit *> >
+ Distributeur(contenance_friandises, contenance_boissons : unsigned int) - ajouterProduit(p : Produit *, nb_type_produit : unsigned int &, contenance_max : unsigned int) + ajouterProduit(b : Boisson *) : void + ajouterProduit(f : Friandise *) : void + retirerProduit(identifiant : unsigned int , montant : float) : Produit* + viderCaisse() : float + description() const : string

`m_montant_caisse`, `m_nombre_friandises` et `m_nombre_boissons` représentent respectivement le montant de la caisse, le nombre de total de friandises et de boissons contenues dans le distributeur. `m_produits` est un tableau associatif qui met en correspondance un identifiant de produit (boisson ou friandise) et les produits ayant cet identifiant contenus dans le distributeur.

Exercice 14 : Implémentez le constructeur de la classe Distributeur.

Exercice 15 : Implémentez la méthode `ajouterProduit(Produit *p, unsigned int nb_type_produit, unsigned int contenance_max)` qui ajoute un produit au distributeur. Avant d'ajouter le produit, cette méthode s'assure que la contenance maximale pour ce produit n'a pas été atteinte (par exemple, si c'est une boisson, que le nombre maximal de boisson dans le distributeur n'a pas été atteint).

Exercice 16 : Implémentez les méthodes `ajouterProduit(Boisson *b)` et `ajouterProduit(Friandise *f)`.

Exercice 17 : Implémentez la méthode `description()` qui renvoie une chaine de caractères contenant l'ensemble des informations d'un distributeur, i.e. le nombre de produits qu'il contient et les informations de chacun de ses produits.

Exercice 18 : Implémentez la méthode
`Produit* retirerProduit(unsigned int identifiant, float montant)`.
Cette méthode doit vérifier qu'un produit d'identifiant `identifiant` est présent et que le montant `montant` y correspond (retourne `nullptr` dans le cas contraire). Si le produit peut effectivement être retiré, alors cette méthode retourne un pointeur sur ce produit, le retire du distributeur et ajoute le montant à la caisse. Pour implémenter cette méthode, il est nécessaire de connaître le type "réel" de l'objet d'identifiant `identifiant`. En effet, cette méthode doit mettre à jour soit `m_nombre_friandises` soit `m_nombre_boissons` en fonction du produit retiré.

Pour cela, vous devrez utiliser le **transtypage**. Or il n'est pas possible de transtyper directement un pointeur de produit en pointeur de boisson :

```
Produit *p;
...
Boisson * b = p; // interdit
```

En effet, le compilateur ne peut pas autoriser une telle instruction car rien ne dit que l'élément pointé par `p` a été effectivement instancié comme un objet `Boisson` (ou dérivé). Lorsque l'on souhaite transtyper un pointeur, il faut donc l'indiquer explicitement. Cela se fait avec l'opérateur `dynamic_cast`. Ainsi :

```
Produit *p;
...
Boisson * b = dynamic_cast<Boisson *>(p);
if ( b == nullptr )
  cerr << "ptr ne pointait pas sur une instance de Boisson." << endl;
else ...
```

L'opérateur `dynamic_cast` analyse le type réel de l'objet pointé et, si le transtypage est valide, retourne un pointeur du bon type, sinon il retourne le pointeur `nullptr`. En quelque sorte, il permet de redescendre da manière sécurisée dans une hiérarchie de types.

Remarque : Si vous n'avez pas d'autre choix que d'utiliser le transtypage, c'est très certainement dû à une **mauvaise conception**. Si vous avez la possibilité de revenir à l'étape de conception (ce n'est pas toujours le cas), cette solution sera surement préférable.

Exercice 19 : Implémentez la méthode `viderCaisse()` qui remet la caisse du distributeur à zéro et retourne la somme correspondante.