Grands Moulins

13/06/2023 - 9h30-11h30

Langage Objet Avancé - C++

Consignes :

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention : il vaut mieux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles on peut ajouter une barre de titre (classe Titlebar) et/ou un ascenseur (classe Slider). Sur chacun de ces objets on peut invoquer une méthode void draw() le dessinant. Dans l'exercice, ces méthodes se contentent d'afficher un texte : la méthode draw d'une fenêtre affiche un texte et ses dimensions, celles des objets Titlebar et Slider affichent un texte puis délèguent le reste à un objet référencé en interne.

Pour que vos réponses soient lisibles sur papier :

- commencez au brouillon afin de faire vos ajustements sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp : écrivez tout dans un .cpp par classe.
- ne vous préoccupez pas des copies, ni des destructions, ce qui nous intéresse est la structure que vous concevez pour que le code fasse ce qui est indiqué.

Question : écrivez les classes Window, Titlebar et Slider en ajoutant éventuellement d'autres classes pour que le code ci-dessous fonctionne et donne les affichages mis en commentaire. Vous pouvez également donner quelques indications sur le raisonnement qui à conduit à vos choix.

```
int main() {
 Window* win0 = new Window(10, 10);
 win0->draw();  // Je dessine une fenetre 10x10

 Slider* win1 = new Slider(new Window(10, 20));
 win1->draw();
 // Je dessine l'ascenseur
 // Je dessine une fenetre 10x20

 Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
 win2->draw();
 // Je dessine la barre de titre Win2
 // Je dessine une fenetre 20x10

 Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
 win3->draw();
 // Je dessine la barre de titre Win3
 // Je dessine l'ascenseur
 // Je dessine une fenetre 20x20


 Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
 win4->draw();
 // Je dessine l'ascenseur
 // Je dessine la barre de titre Win4
 // Je dessine une fenetre 20x20
}
```

Exercice 2 (6 points)

- Le code suivant modélise des listes simplement chaînées.

```
class Noeud {
 public :
  int valeur;
  Noeud* suivant;
  Noeud (int v, Noeud *n):valeur{v}, suivant{n} {}
  ~Noeud() {}
};
class ListeChainee {
 private:
  Noeud* tete;
 public:
  ListeChainee() : tete{nullptr} {}
  void ajouter(int valeur) { tete = new Noeud (valeur, tete); }
  // Destructeur
  ~ListeChainee() { ....  // a completer }
};
```

1. Ecrivez le destructeur de listes.
2. En réfléchissant un peu à l'utilisation habituelle qu'on fait des listes (ajout, suppression, opérations de parcours ou autre) j'aimerais que vous trouviez une justification (en quelques lignes) du fait que le destructeur de noeud est bien celui qui est écrit dans ce code.

- Ecrivez le destructeur qui convient pour les vecteurs d'entiers :

```
class VecteurEntier {
 private:
  int* elements; // Pointeur vers le tableau d'entiers
  int taille;    // Taille du tableau
 public:
  VecteurEntier(int taille) :taille{taille}, elements {new int[taille]} {}
  // Destructeur
  ~VecteurEntier() {  .... // a completer }
}
```

- Quels sont les affichages produits lors de l'exécution du main suivant ? (Expliquez un peu mais clairement votre réponse)

```
class A {
 public :
  virtual ~A() { cout << "del A" << endl;}
};

class B : public A {
 public :
  virtual ~B(){ cout << "del B" << endl;}
};

class C {
 public :
  ~C() { cout << "del C" << endl;}
};
```

```
class D : public C {
 public :
  virtual ~D(){ cout << "del D" << endl;}
};

int main() {
 A * a = new B();
 C * c = new D();
 delete a;
 delete c;
}
```

Exercice 3 (4 points) On cherche à caractériser les animaux en relevant leur couleur et leur poids dans des attributs private. On établi ensuite une classification principalement en les séparant entre les sous classes des oiseaux et des mammifères. Puis on complètera éventuellement (ne le faites pas) cette hiérarchie par des races (Chien, Poule, etc ...). Ce qui nous intéresse vraiment est de traiter un cas particulier : celui de la chauve-souris qui a la particularité de voler et d'être un mammifère ...

```
class Animal {
 public:
  void presentation() {
   whatIam();  // a faire ...
   cout << "de couleur " << couleur << " et de poids " << poids << endl;
  }
  Animal(string c, int p): couleur{c}, poids{p}{}

 private :
  string couleur;
  int poids;
}
```

1. Ecrivez la hiérarchie de classes la plus adaptée qui permet de modéliser les chauve-souris. (On veut voir apparaître le code des classes en se focalisant sur l'héritage, et que vous écriviez un constructeur simple par classe)

2. La méthode whatIam() **n'est pas publique**. Ecrivez là, de sorte que la présentation d'un exemple trivial composé d'un animal, d'un mammifère, d'un oiseau et d'une chauve souris donne :

```
je suis un animal inconnu de couleur blanc et de poids 100
je suis un mammifere de couleur rose et de poids 200
je suis un oiseau de couleur vert et de poids 300
je suis un mammifere je suis un oiseau de couleur noire et de poids 400
```

   Merci de respecter cette consigne : en particulier une chauve souris ne dit pas qu'elle est une chauve souris, et chaque animal ne donne son poids et sa couleur qu'une fois.

Exercice 4 (6 points) On rappelle que le patron de conception "Observateur" est utilisé lorsque des objets, appelés sujets, maintiennent une liste d'objets dépendants, appelés observateurs, et les informent automatiquement de tout changement d'état. Vous trouverez son diagramme ci-dessous. Notez bien que les signatures sont indicatives, dans la mesure où la communication entre sujet et observateur peut être enrichie en transmettant des données utiles.

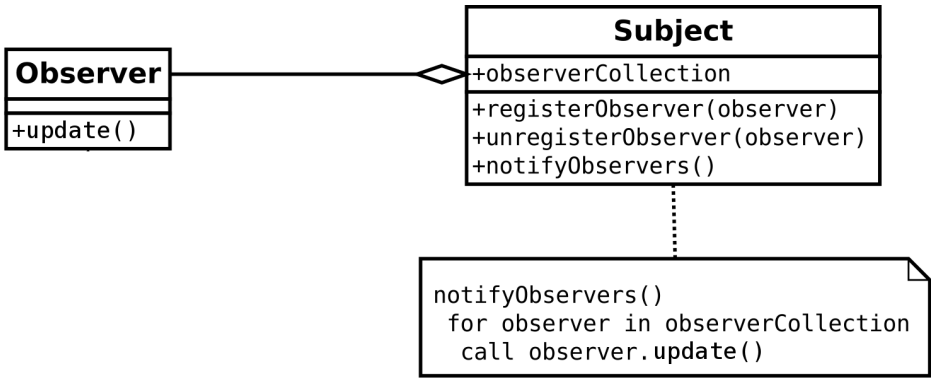

Figure 1: UML du patron observateur.

Nous allons nous intéresser à un contexte qui est celui de la diffusion de vidéos en streaming entre ce que nous appellerons des Serveur (canal plus, disney, netflix), et des postes de visionnage qui seront leurs Client.

1. Dans ce cas, qui joue le rôle de l'Observer ? et qui joue celui du Subject ?
2. Voici quelques éléments préalables à l'écriture des classes :
   (a) Seule l'image du direct sera stockée coté serveur (nous supposons pour cela qu'une classe Image est disponible, cela n'a pas d'autre importance). L'observateur, lui, enregistrera tout ce qui l'intéresse dans une liste pour permettre sont visionnage [1]. Cela devrait vous donner une première indication sur les attributs à introduire, et sur les signatures les plus adaptées pour les notifications.
   (b) Il faudra prévoir un setter de l'image du direct (...)
   (c) L'observateur devra être construit avec en paramètre une référence vers son sujet, cela aura une conséquence **à prendre en compte lors de la destruction de l'observateur** ; ainsi que sur **l'ordre d'écriture de votre code**.
   (d) L'enregistrement, et la désinscription sont purement déclaratives, elles se feront sans contraintes (en réalité il faudrait appliquer la politique commerciale).
   (e) ne vous préoccupez pas de constructeurs de copies, concentrez vous sur la modélisation du problème faite dans le cadre de ce pattern.

   A présent vous devriez être en mesure d'écrire les deux classes Serveur et Client. Un conseil : faites le d'abord au brouillon. Ici, séparez .hpp et .cpp et **dites bien dans quel ordre** vous les écririez dans le compilateur, si vous faites des déclarations préalables etc ... Pour info, ma correction fait une trentaine de lignes en tout, pas plus.

[1] Vous pouvez utiliser la librairie standard et "inventer" les opérations si vous ne vous rappelez pas bien de la syntaxe