TP Western C++ n°5

Sommaire

Les objectifs de ce tp sont de découvrir la programmation orientée objet en C++.
On désire réaliser un programme C++ permettant d'écrire facilement des histoires de Western.
Dans nos histoires, nous aurons des brigands, des cowboys, des shérifs, des barmen et des dames en détresses ... (à partir d'une idée de Laurent Provot)

Histoire n°5 : le duel !

Objectifs

On désire réaliser un programme orienté objet en C++ qui **racontera une histoire où les personnages sont armés et tirent**.

```
** Le vaillant Lucky Luke tire sur Joe
** PAN !
(Lucky Luke) -- Prends ça, rascal !
```

La programmation orientée objet en action

Rappels

La programmation orientée objet consiste à **définir des objets logiciels et à les faire interagir entre eux**.

Le fait que les objets interagissent entre eux implique qu'ils ont des **relations**.

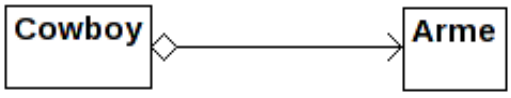
La relation d'agrégation

L'**agrégation d'objets** signifie implicitement « contient un ou plusieurs » (ou « est composée d'un ou plusieurs » ou « possède un ou plusieurs ») ou tout simplement **« a un ou plusieurs »**.

On distingue deux types d'agrégation :
- **Agrégation interne** : l'objet agrégé est créé par l'objet agrégateur
- **Agrégation externe** : l'objet agrégé a été créé extérieurement à l'objet agrégateur

Il y a 3 possibilités de mise en oeuvre :
- agrégation par valeur (appelée aussi **composition**)
- agrégation par référence
- agrégation par pointeur

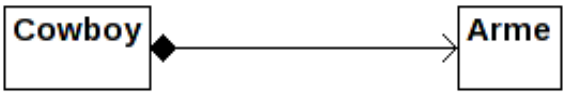
L'**agrégation** se représente de la manière suivante en UML :



La relation de composition

Une **composition** est une agrégation plus forte signifiant **« est composée d'un »** et impliquant :
– une partie ne peut appartenir qu'à un seul composite (agrégation non partagée)
– la destruction du composite entraîne la destruction de toutes ses parties (responsable du cycle de vie de ses parties).

La **composition** se représente de la manière suivante en UML :



Exemple détaillé : un cowboy armé

Dans nos histoires, un **cowboy**, un **brigand** et un **sherif** peuvent posséder une **arme**.

Une **arme** est caractérisée par son **nom** (exemple "Smith & Wesson"), la **capacité** de son barillet (c'est le magasin ou chargeur d'une arme), le **nombre de balles** présentes dans le barillet et son **prix** (par défaut 100 dollars). Evidemment, on peut **tirer** (cela fait « PAN ! ») avec une arme si elle est chargée. Lorsqu'il n'y a plus de balles dans le barillet, il est possible de la **recharger**. Par défaut, une arme a une capacité de 6 balles mais elle est vide.

Arme
string nom int capacite int nbBalles int prix
Arme(const string nom = "", int prix = 100, int capacite = TAILLE_MAX_CHARGEUR, int nbBalles = 0) string getNom() int getPrix() int getNbBalles() void tire() void recharge(int nbBalles)

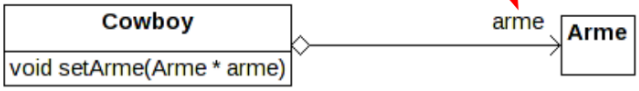
Un **cowboy** peut posséder une **arme**. Il y a donc une relation entre la classe `Cowboy` et la classe `Arme`. Comme on va supposer qu'un cowboy peut être dépossédé de son arme, on va donc établir une **relation d'agrégation**.

C++

```
class Cowboy : public Humain
{
   public:
      // ...
      void setArme(Arme *arme);

   private:
      // ...
      Arme *arme; // l'agrégation
};
```

UML




```
Cowboy lucky("Lucky Luke", "coca-cola");
Brigand joe("Joe");

// Un Smith & Wesson de 100 $ chargé de 5 balles
Arme *arme = new Arme("Smith & Wesson", 100, 5, 5);

// Lucky se procure une arme
lucky.setArme(arme);

// Et tire sur Joe
lucky.tire(joe);
```

PAN !

On obtient ceci :

```
** Le vaillant Lucky Luke tire sur Joe
** PAN !
(Lucky Luke) -- Prends ça, rascal !
```

Il a fallu apporter quelques modifications à notre classe `Cowboy` :

```
// La méthode setArme() permettra à un Cowboy de posséder une arme
void Cowboy::setArme(Arme *arme)
{
   this->arme = arme;
}

// On modifie la méthode tire() car maintenant on tire pour de vrai !
void Cowboy::tire(const Brigand &brigand) const
{
   // ai-je une arme ?
   if(arme != NULL)
   {
      // est-elle chargée ?
      if(arme->getNbBalles() != 0)
      {
         cout << "** Le " << getQualite() << " " << getNom() << " tire sur " << brigand.
            getNom() << endl;
         arme->tire();
         cout << "(" << nom << ") -- " << "Prends ça, rascal ! " << endl;
      }
   }
}
```

cowboy.cpp

Pour éviter des fuites de mémoire, il faut s'assurer que l'objet `arme` est bien détruit. On va établir que lorsqu'un cowboy meure, son arme sera détruite. Il faut donc ajouter un **destructeur** à notre classe `Cowboy` qui aura comme responsabilité de détruire l'arme.

```
class Cowboy : public Humain
{
   public:
      // Constructeur
      Cowboy(const string nom="", const string boissonFavorite="whisky", const string
         qualite="vaillant");
```



```
      // et Destructeur
      ~Cowboy();
      ...
};

// Constructeur : un Cowboy nait
Cowboy::Cowboy(const string nom/*=""*/, const string boissonFavorite/*="whisky"*/, const
    string qualite/*="vaillant"*/) : Humain(nom, boissonFavorite), qualite(qualite),
    popularite(0), arme(NULL)
{
}

// et Destructeur : un Cowboy meure
Cowboy::~Cowboy()
{
   // ai-je une arme à détruire ?
   if(arme != NULL)
      delete arme;
}

...
```

La classe Cowboy

Travail demandé

Question 1. Modifier la classe `Brigand` pour que les objets de cette classe puissent eux aussi posséder une **arme**. Faut-il modifier aussi la classe `Sherif` ?

Question 2. Écrire une histoire de duel entre un **cowboy** et un **brigand**.

