De la modélisation objet à la programmation objet

TP n°6 - Hoerdt Mickaël/Quirin Arnaud - Avril 2004

Résumé

Le but de ce TP est de passer de la modélisation objet d'un problème (ici le modèle UML d'un stade de foot) à sa version écrite et compilée dans un langage défini (ici, le programme Stade réalisant les diverses opérations demandées). Bien que n'importe quel langage puisse convenir, le C++ a été choisi pour ce TP. Il devrait normalement pouvoir être adapté à d'autres langages objets.

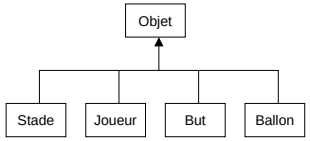

FIG. 1 – Diagramme de généralisation

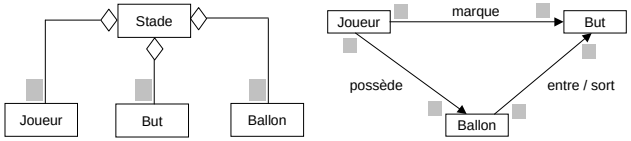

FIG. 2 – Diagramme classe-association (agrégations et associations)

1 Description

L'ensemble des diagrammes UML présentés ci-dessus permet de modéliser les entités (ou objets) intervenant dans un logiciel de gestion d'un stade. Le diagramme de généralisation indique la hiérarchie entre ces entités et les diagrammes de classe-association indiquent les types d'agrégation (de type conteneur-contenant) et les types d'association entre ces entités.

Notes supplémentaires : Un stade peut contenir 0 ou 1 ballon, 2 buts et plusieurs joueurs. Le ballon peut être possédé par plusieurs joueurs, dans le cas d'une mêlée. Un joueur peut marquer entre 0 et plusieurs buts. Un but n'est marqué que par un seul joueur.

En UML, chaque morceau d'arc est valué par un symbole indiquant le nombre d'entités intervenant dans la relation en question. Des symboles possibles sont "5", "2...3" pour [2 ;3] ou "*" pour [0 ;infini].

Rappel : une classe en C++ est la structure permettant de contenir un objet, avec ses attributs et ses fonctions, à la manière d'un type de donnée en C. On peut créer des variables de ce nouveau type. La définition d'une telle variable se nomme instance ou objet de la classe considérée. Veillez toujours à ce que les fonctions ou les attributs demandés se trouvent dans les bonnes classes. Par exemple le prototype `double Legume : :Donner_prix(void)` indique une fonction `Donner_prix()` renvoyant un double et opérant sur une instance de la classe Legume.

2 Mise en place

- Commencez par étudier les diagrammes UML et proposez une description possible en terme de classes (prototype des classes, écrit en C++). Le TP étant relativement court, n'hésitez pas à prendre du temps pour cette première question. Créez autant de classes que d'objets nécessaires et explicitez les notions d'héritage dans ces classes. Complétez dans le schéma les valuations des arcs dans les cadres grisés.

  La définition UML des classes est la suivante :

Joueur
int age ; string nom ;
Joueur(string, int) ; void print(void) ;

Ballon
int identifiant ;
Ballon(int) ; void print(void) ;

But
int local ;
But(int) ; void print(void) ;

- Téléchargez chez vous le code source à l'adresse suivante :
  `/users/prof/quirina/GL/TP6/*`

- Complétez le code source avec les classes définies pour la première question. Les définitions de classes doivent être placées dans les fichiers .hh
- Compilez et testez votre programme.
- Créez une classe 'But' contenant un attribut local valant 1 pour l'équipe locale et 0 pour l'équipe visiteuse.
- Fabriquez pour But une fonction constructeur et une fonction d'affichage.
- Fabriquez une fonction de saisie pour chacune des classes, permettant de saisir à la main les différents objets. Par exemple `Joueur : :saisie()` demandera le nom et l'âge du joueur et stockera les informations dans les attributs de classe. On pourra éventuellement fabriquer un constructeur sans paramètre appelant `saisie()`.

3 Polymorphisme

- Les objets de type 'Joueur' et 'Ballon' héritent leurs fonctions de la classe 'Objet', notamment la fonction `print()`. Vérifiez que l'affichage d'objets déclarés sous forme de pointeurs renvoie à la fonction `print()` générique.
- Que faudrait-il faire pour utiliser à la place la 'bonne' fonction, c'est à dire `Ballon : :print()` à la place de `Objet : :print()` lorsqu'on veut afficher un Ballon ? Le C++ dispose d'un mécanisme appelé le polymorphisme permettant de réaliser cela. Mettez ce principe en oeuvre et re-exécutez `main()` pour voir la différence.

4 Généricité

- La généricité permet à une même fonction de prendre des paramètres de différents types (par exemple soit un entier, soit un réel) et de faire des calculs avec, sans avoir à écrire deux fonctions distinctes. Ce principe utilise la notion de *function template*

du C++. Testez la fonction `sqr()` définie au début de Main.cc après avoir déclaré des objets de différents types (char, short, int, double, ...) et les avoir passé en paramètre à `sqr()`.
- Mettez ce principe en oeuvre dans la définition d'une fonction `static void Objet : :print(Objet*)` dans la classe Objet, affichant l'objet donné en paramètre, qu'il soit de classe 'Ballon', 'Joueur' ou 'But'.

La STL (ou Standard Templates Library) est une bibliothèque de fonction génériques utilisant les templates à outrances, plus précisement les *class templates*, afin de mettre en oeuvre :

1. des classes permettant d'intégrer différents types (par exemple des tableaux d'entiers, de double, de char, d'un type personnalisé, ... implémentés dans la même classe 'tableau'). Ces classes sont plus généralement appelées des 'conteneurs'.
2. des algorithmes s'appliquant à ces conteneurs (par exemple `sort()` permettant de trier un tableau d'objets à partir du moment ou une fonction de comparaison a été définie pour deux objets de cette classe).

- La bibliothèque STL contient le conteneur "vector" (tableau) permettant de stocker des objets du même type dans un tableau. Mettez en oeuvre ce type de conteneur pour créez des tableaux d'entiers. Insérez 5 éléments dans ce tableau et affichez-les à l'aide d'une boucle.

  Note : Cette classe est définie dans <vector> (`#include <vector>`). De plus, pour simplifier le source, il sera peut être utile d'ajouter la ligne `using namespace std;` en haut du source. Des fonctions utiles seront :
  `vector : :push_back(const T &val)` et `int vector : :size(void)`.

- Utilisez le même principe pour créer des tableaux d'objets de type 'Joueur'.

5 Classes et instances

- Créez une classe Stade pouvant contenir des instances de classe 'Ballon', 'Joueur' et 'But'. Ces instances seront stockées dans un tableau générique de type <vector> lorsque UML spécifie une association multiple ou un pointeur pour une association simple.
- Créez une fonction `saisie()` permettant de donner la main à chacune de vos autres fonctions `saisie()`, qui finalement va saisir le contenu du stade entier. Par exemple, la saisie d'un joueur sera confiée à la fonction `Joueur : :saisie()`, tandis que l'introduction du nombre de joueurs sera confiée à la fonction `Stade : :saisie()` qui appellera N fois `Joueur : :saisie()`.
- Créez une fonction d'affichage `void Stade : :print(void)` permettant d'afficher le contenu du stade.

6 Associations

- Créez les fonctions suivantes dans leurs classes respectives :
  ```
  void Ballon::Associe(Joueur*);
  void Ballon::Associe(But*);
  void But::Associe(Joueur*);
  ```
  Elles permettent d'associer respectivement un joueur au ballon qu'il dirige, un ballon à un but et un joueur à un but. Respectez les contraintes définies dans le diagramme 'Associations'. Par exemple à tout moment, un joueur ne peut être associé qu'à un et un seul ballon. Concrètement, cela pourra se réaliser à l'aide un pointeur ou d'un tableau de pointeurs en tant qu'attribut de classe de la classe considérée, selon que l'association soit simple ou multiple. Par exemple :
  ```
  Joueur* MaClasse::monJoueur;
  vector<Joueur*> MaClasse::mesJoueurs;
  ```
- Modifiez la fonction `print()` de chacune des classes pour qu'elle affiche les objets associés avec la classe en question. Par exemple `Ballon : :print()` affichera le ou les joueurs manipulant le ballon.
- Une association dans le modèle UML étant toujours bi-directionnelle, réalisez les associations inverses, c'est-à-dire :
  ```
  void Joueur::Associe(Ballon*);
  void But::Associe(Ballon*);
  void Joueur::Associe(But*);
  ```
- Testez vos fonctions dans la fonction `main()`.