Centrale-Lyon - POO (INFO 2A)
Année 2005-2006

Travaux dirigés de 2ème année
Sujet TD n°4

(Version 3.2 – 8/11/2005)

Objectifs

On aborde dans ce TD, comme nouveaux concepts, les fonctions virtuelles et le polymorphisme.

Introduction

Dans ce TD on cherche à modéliser une entreprise et ses employés avec la possibilité d'afficher le nom et la paye de tous les employés quel que soit leur statut.

1) modélisation avec UML d'une entreprise

On veut modéliser une entreprise et ses employés.

L'entreprise est simplement définie par son nom. On peut bien sûr rajouter d'autres infos comme adresse, téléphone, fax, n° SIRET, code NAF, etc…

Pour chaque employé on stocke son nom et son salaire mensuel. De la même manière on pourrait ajouter l'adresse, le n° INSEE, la date de naissance, etc…

Donnez une première modélisation avec un diagramme de classe UML (sans les méthodes).

Les employés peuvent avoir 2 statuts possibles : cadre ou ouvrier. Les cadres disposent, en plus du salaire, d'une prime mensuelle. De leur côté, en plus du salaire, les ouvriers peuvent avoir des heures supplémentaires.

Pour une entreprise, on souhaite :
- ajouter des employés,
- retirer des employés,
- chercher un employé par son nom,
- afficher le nom, le statut et le salaire de tous les employés de l'entreprise.

Pour un employé, on souhaite :
- modifier son salaire (y compris prime ou heures supplémentaires),
- obtenir son statut et son nom,
- calculer son salaire (en tenant compte de la prime ou des heures supplémentaires)

NB : pour le calcul du salaire des ouvriers on comptera les heures supplémentaires à un taux horaire calculé sur la base du salaire mensuel divisé par 169.

Représentez avec un diagramme de classe, la classe Entreprise, la classe Employe et les deux classes dérivées Cadre et Ouvrier. Définissez complètement les attributs et les méthodes.

2) Mise en œuvre de la relation d'agrégation

Dans le diagramme UML ci-dessus on voit apparaître une relation d'agrégation. Il n'y a pas de mécanisme intégré au C++ permettant de mettre en œuvre directement cette relation.

Nous avons les choix suivants :
- Mettre en œuvre un mécanisme *ad hoc* parfaitement adapté à notre problème. L'inconvénient majeur est le temps de développement (et de maintenance !) qui obère cette solution.
- Utiliser des structures de données génériques puisées dans des bibliothèques de composants proposées par des fournisseurs. Le coût éventuel de la bibliothèque et le temps d'apprentissage est généralement largement compensé par les avantages en terme de qualité.

Dans notre cas, nous choisissons la deuxième solution d'autant que nous disposons de la STL (Standard Template Library), bibliothèque standard du C++ qui nous est fournie gratuitement avec tout compilateur C++.

La STL fournit de nombreuses structures de données standard (basé sur le type Cont – Container) :
- *vector* : tableau dynamique unidimensionnel
- *deque* (≈ tableau dynamique extensible à chaque extrémité) qui peut être spécialisé en *Queue* (file) et *Stack* (pile)
- *list* : liste doublement chaînée
- *set, multiset* : ensemble de données ordonnées (avec doublons possibles avec *multiset*)

- *map, multimap* : tableaux associatifs (avec doublons possibles avec *multimap*)
- Etc…

Toutes ces structures permettent l'insertion, la suppression, l'accès direct à des éléments donnés ainsi que la recherche d'un élément par valeur. La principale différence tient dans la **complexité algorithmique** de ces différentes opérations. Le tableau ci-dessous résume les principales complexités, en faisant ressortir en gras la meilleure performance :

	Insertion			**Suppression**			**Accès**			**Recherche d'un élément par valeur**
	Tête	Qcq	Queue	Tête	Qcq	Queue	Tête	Qcq	Queue	
vector	O(N)	O(N)	O(1)	O(N)	O(N)	O(1)	**O(1)**			O(N)
deque	**O(1)**	O(N)	**O(1)**	**O(1)**	O(N)	**O(1)**	**O(1)**			O(N)
list	**O(1)**			**O(1)**			O(1)	O(N)	O(1)	O(N)
set, map	-	O(log N)	-	-	O(log N)	-	O(1)	-	O(1)	**O(log N)**

NB1 : par défaut *stack* et *queue* sont basés sur *deque*. Il est aussi possible de baser *stack* et *queue* sur *list* et *stack* sur *vector*. (grâce au 2ème paramètre du template) En pratique si on se limite aux opérations « empiler/enfiler » et « dépiler/défiler », ces structures sont équivalentes ; la différence intervient si on veut **en plus** :

- des accès direct aux éléments (*deque* et *vector* sont en O(1) comme un tableau)
- insérer et supprimer n'importe où (*list* est toujours en O(1) pour ces opérations)

NB2 : *set* et *map* (et donc *multiset* et *multimap*) proposent une recherche d'élément en O(log N) car les éléments sont rangés dans un arbre binaire ordonné ce qui permet une recherche dichotomique. L'insertion et la suppression sont aussi en O(log N) car il faut rechercher la position avec de ranger le nouvel élément. Ces structures imposent comme **pré requis** que les éléments stockés soient ordonnables :

- Soit parce que ce sont des types de base comme int, double, ou simples comme string qui sont totalement ordonnés,
- Soit parce que l'opérateur < est surchargé pour le type de l'élément ou encore une classe *ad hoc* permettant le tri est fournie comme paramètre du template (voir §8 sur ce sujet)

D'après les critères donnés ci-dessus, choisissez la structure de données de la STL qui vous semble la plus adaptée à notre cas, en pesant le pour et le contre de votre solution.

3) Classe Employe

Passez en C++. Ecrivez toutes les méthodes associées (fichiers employe.h et employe.cc).

4) Classes Cadre et Ouvrier

Développez les classes Cadre et Ouvrier (fichiers employe.h et employe.cc).

5) Classe Entreprise

Développer la classe Entreprise (fichiers entreprise.h et entreprise.cc).

On souhaite, en particulier, développer une méthode « affiche_tous » qui parcourt la liste de tous les employés et les affiche en utilisant l'opérateur "<<". On ajoutera donc la surcharge de l'opérateur << (fonction globale qu'on rendra "friend" de la classe Employe) pour pouvoir afficher le nom, le statut et le salaire d'un employé.

6) Programme principal

Faire un programme principal dans td4.cc qui crée une Entreprise et y ajoute quelques employés. Invoquer la méthode « affiche_tous » qui doit afficher les statuts et les salaires corrects… Ensuite essayez les méthodes de recherche et de suppression d'un employé.

7) Devoir à rendre

Les élèves doivent finir le travail commencé en séance.

Il leur est demander **d'ajouter une nouvelle catégorie d'employés** "Commercial" qui, comme un cadre, possède un salaire mensuel et une prime fixe, mais qui en plus a une prime de 3% du montant des ventes qu'il a réalisé dans le mois.

Délai : 14 jours francs à partir de la date du TD.

8) Travail optionnel

Pour les élèves qui souhaitent s'investir d'avantage en informatique (par exemple suivre l'option Informatique ou l'option TIC en 3ème année), nous leur proposons de réaliser un travail complémentaire. Attention, il s'agit bien d'un travail supplémentaire, donc ne vous y lancez que si vous êtes motivés et à l'aise !

Il s'agit de traiter le problème en utilisant *set* à la place de *list*. La principale difficulté tient au fait qu'il faut fournir un ordre total pour les éléments du *set*. Ici les éléments sont du type « Employe * » ; par défaut le C++ est capable d'ordonner les pointeurs ce qui semblerait pouvoir fonctionner mais en fait ferait systématiquement échouer la recherche et la suppression d'éléments : en effet, pour rechercher un employé on est amené à créer une instance temporaire de la classe Employe et le pointeur sur cette instance est bien sûr différent du pointeur sur l'objet stocké dans le *set* et donc la recherche (ou la suppression) échoue toujours.

La bonne solution est d'établir l'ordre sur le nom des employés. Pour cela on pourrait être tenté de simplement surcharger l'opérateur '<' pour des éléments de type « Employe * », mais le C++ l'interdit ! Une solution est de créer une instance explicite de la classe less<T> existante (Cf. § 9.2.2.5 du polycopié). En effet, la classe *set* utilise la classe less<T> comme prédicat par défaut pour effectuer le tri des éléments. Techniquement, l'écriture de la classe less<T> contient la surcharge de l'opérateur () qui fait la comparaison de 2 éléments avec la sémantique du strictement inférieur et renvoie true ou false.

Voici donc un extrait du code correspondant :

```
namespace std
{
    class less<Employe*>
    {
    public:
        bool operator() (const Employe * e1, const Employe * e2) const
            {
                // --- insérer ici le code faisant la comparaison :
                // --- nom de employe1 < nom de employe2
            }
    };
}
```

Ensuite il faut modifier dans la classe Entreprise les méthodes d'insertion, de suppression et de recherche (voir la documentation sur la STL pour savoir quels sont les bonnes méthodes à utiliser…)

NB : Le fait de trier sur les noms seulement empêche d'avoir des homonymes. On pourrait donc rajouter d'autres champs secondaire pour le tri comme la date de naissance ou un numéro d'employé, etc… Il serait aussi possible d'accepter les homonymes en utilisant le template *multiset* au lieu de *set* qui permet d'avoir plusieurs occurrences d'objets égaux (au sens du prédicat de tri).

Bon courage !

Documentations sur la STL :

- Chez Microsoft : http://msdn.microsoft.com/library/en-us/vcstdlib/html/vcoriStandardCLibraryReference.asp
- chez Silicon Graphics : http://www.sgi.com/tech/stl/

Attention : Ces documentations comportent des extensions à la norme ISO ; elles sont normalement clairement signalées (par exemple hash_set, hash_table, etc…)