INF1010

Programmation Orientée-Objet

Travail pratique #3

Classe abstraite, Héritage, Polymorphisme, Classes génériques, Méthodes génériques

---

À remettre avant le mardi 25 février 2025, 23:30

À la fin de ce laboratoire, vous devrez être capable de :

- Comprendre le concept de classes abstraites et d'héritage.
- Appliquer le polymorphisme.
- Utiliser des classes génériques et des méthodes génériques.

Directives :

- Les travaux s'effectuent obligatoirement en équipe de deux personnes faisant partie du même groupe.
- Ne touchez pas aux entêtes des fichiers fournis sauf si on vous le demande.
- Les fonctions que vous décidez d'ajouter au programme doivent être documentées.

Conseils :

- Lisez les conseils, l'aperçu et les spécifications avant de commencer!
- Ayez lu vos notes de cours!
- Si vous avez des difficultés au cours du TP, rappelez-vous que de nombreux problèmes demandés sont assez couramment rencontrés en C++.
- Consulter la documentation sur cppreference, les notes de cours, et les forums sur Google peuvent vous donner de bonnes pistes de résolution!
- Si votre programme ne compile pas, veuillez mettre en commentaires les instructions qui ne compilent pas.
- Relisez votre code après l'avoir écrit pour éviter les erreurs de syntaxe.

Spécifications générales

- Tout warning à la compilation sera pénalisé. Si vous utilisez Visual Studio de Microsoft, vous devez activer l'option /W4. Sur Visual Studio, assurez-vous de compiler avec x64, certains warnings pourraient ne pas s'afficher sinon.
- Des TODOs sont présents dans le code pour vous guider. Vous devez les compléter.
- Toutes les méthodes doivent être définies dans le fichier d'implémentation (.cpp) dans le même ordre que leur déclaration dans le fichier d'en-tête (.h). Le non-respect de cette règle entraînera une pénalité au niveau du style.
- Utilisez le plus possible la liste d'initialisation des constructeurs. L'utilisation du corps des constructeurs à la place de la liste d'initialisation entraînera une pénalité de style.
- L'ordre des variables (attributs) dans la liste d'initialisation doit être la même que celle dans la liste des attributs de la classe dans le fichier d'en-tête
- Suivez le guide de codage sur Moodle.

Mise en contexte

En tant que développeur stagiaire dans un hôpital, votre tâche est de mettre en place un système de gestion du personnel et des patients. Ce système doit permettre de gérer les médecins, les infirmiers, les patients et leurs interactions. Les médecins peuvent avoir des spécialités spécifiques (Cardiologie, Chirurgie, Pédiatrie), et les infirmiers s'occupent des patients dans des chambres spécifiques.

Travail à réaliser

Les fichiers Employe.h, Medecin.h, Infirmier.h, Pediatre.h, Cardiologue.h, Chirurgien.h, Patient.h, Liste.h, HopitalPoly.h vous sont fournis, vous devez compléter les implémentations des classes (suivez les TODO pour les instructions et ensuite les supprimer).

Classe Liste(Classe Générique)

Cette classe est un conteneur générique qui permet de gérer une liste d'objets de type T. Elle est conçue pour être réutilisable et peut être utilisée pour stocker des médecins, des infirmiers, ou d'autres types d'objets.

Attributs :

- **`listes_`**: Un **`vector<T>`** qui stocke les objets de type `T`.

Méthodes :

- **`Constructeur`** par défaut
- **`Destructeur`** par défaut : ~Liste() = default
- **`operator+=:`**
    - Ajoute un objet **`T`** à la liste s'il n'existe pas déjà
    - Ajoute tous les éléments d'une autre liste **`Liste<T>`**
- **`operator-=:`**
    - Supprime un objet **`T`** de la liste s'il existe.
    - Supprime un objet de la liste en fonction de son nom (si **`T`** a une méthode get-Nom()).
    - Supprime tous les éléments d'une autre liste **`Liste<T>`** de la liste actuelle.
- **`chercher():`** Recherche un objet dans la liste ou recherche un objet par son nom.
- **`operator[] :`** permet d'accéder aux éléments de la liste par index.

- **`size()`:** Retourne la taille de la liste.

Classe Employe (Classe abstraite)

La classe **`Employe`** est une classe abstraite qui représente un employé dans l'hôpital. Elle contient un nom, un salaire et une méthode virtuelle **`calculerSalaire()`,** qui doit être redéfinie dans les classes dérivées, et une méthode **`afficher()`** pour afficher les informations de l'employé.

Attributs :

- **`nom_`:** Le nom de l'employé.

Méthodes :

- **`Constructeur`** prenant un nom en paramètre.
- **`getNom()` :** Retourne le nom de l'employé.
- **`setNom(const string& nom)` :** Modifie le nom de l'employé.
- **`calculerSalaire()`:** Méthode virtuelle pure.
- **`afficher()`:** Affiche les attributs de l'employé.
- **`operator <<` :** pour l'affichage.

Classe Infirmier (Dérivée de Employe)

La classe **`Infirmier`** représente un infirmier, qui a un prénom, un taux horaire et est affecté à un patient. Il peut aussi gérer plusieurs chambres dans l'hôpital.

Attributs :

- **`prenom_`:** Le prénom de l'infirmier.
- **`tauxHoraire_`:** Le taux horaire de l'infirmier.
- **`patient_(shared_ptr<Patient>)`:** Un objet `Patient` auquel l'infirmier est assigné.
- **`heuresTravaillees_ (unsigned) : Le`** nombre d'heures travaillées.
- **`listeSalle_ (vector<string>) :`** Liste des chambres gérées par l'infirmier.

Méthodes :

- **`Constructeur`** prenant un nom, un prénom et un taux horaire.

- **`Constructeur de copie.`**
- **`getPrenom(), getTauxHoraire(), getPatient():`** Accesseurs**.**
- **`setPrenom(), setTauxHoraire(), setPatient():`** Mutateurs**.**

- **`calculerSalaire()`:** Calcule le salaire en fonction des heures travaillées et des chambres gérées.
- **`ajouterHeuresTravaillees()` :** ajoute des heures travaillées.
- **`examinerPatient()`:** Assigne un patient à l'infirmier et met à jour ses informations.
- **`afficher()`:** Affiche les informations de l'infirmier.
- **`operator+=`:** Ajout d'une chambre.

Classe Medecin (Dérivée de Employe)

La classe **`Medecin`** représente un médecin, qui a une spécialité et est assigné à un patient. La spécialité peut être nulle.

Attributs :

- **`specialite_`:** Un objet **`Specialite`** qui définit la spécialité du médecin.
- **`patient_(shared_ptr<Patient>)`:** Un objet Patient auquel le médecin est assigné.

Méthodes :

- **`Constructeur`** prenant un nom et une spécialité.
- **`getSpecialite(), getPatient()` :** accesseurs.
- **`getSpecialite(), setPatient()` :** mutateurs
- **`calculerSalaire()`:** Calcule le salaire du médecin en fonction de sa spécialité. (Formule indiquée dans les TODO)
- **`afficher(ostream& out)` :** Affiche les informations du médecin, y compris sa spécialité.
- **`operator==` :** Compare deux médecins en fonction de leur nom et spécialité.
- **`Operator<<` :** pour l'affichage

Classe Cardiologue (hérite de Medecin)

Attributs :

- **`nbPatients_ (unsigned)`** : Nombre de patients traités.
- **`nbConferences_ (unsigned)` :** Nombre de conférences données.
- **`nbPublications_ (unsigned)` :** Nombre de publications.
- **`salaireBase (const float)` :** Salaire de base du cardiologue.

Méthodes :

- **`Constructeur`** prenant un nom, un nombre de patients, un nombre de conférences et un niveau de spécialité.
- **`getNbPatients(), getNbConferences(), getNbPublications() :`** Accesseurs.
- **`setNbPatients(), setNbConferences(), ajouterPublications() :`** Mutateurs.
- **`calculerSalaire() :`** Calcule le salaire en fonction des patients, conférences et publications. (Formule indiquée dans les TODO)
- **`opererCoeur() :`** Opère un patient et met à jour les informations.
- Surcharge de **`l'opérateur <<`** pour l'affichage.

Classe Chirurgien (hérite de Medecin)

Attributs :

- **`nbHeuresDeGarde_ (unsigned) :`** Nombre d'heures de garde.
- **`nbOperations_ (unsigned) :`** Nombre d'opérations effectuées.
- **`salaireBase (const float) :`** Salaire de base du chirurgien.

Méthodes :

- **`Constructeur`** prenant un nom, un nombre d'heures de garde et un niveau de spécialité.
- **`getNbHeuresDeGarde(), getNbOperations() :`** Accesseurs.
- **`setNbHeuresDeGarde(), ajouterNbOperations()`** : Mutateurs.
- **`calculerSalaire() :`** Calcule le salaire en fonction des heures de garde et des opérations. (Formule indiquer dans les TODO)
- **`opererPatient() :`** Opère un patient et met à jour les informations.
- **`operator<<`** pour l'affichage.

Classe Pediatre (hérite de Medecin)

Attributs :

- **`nbEnfantsSoignes_ (unsigned) :`** Nombre d'enfants soignés.
- **`certifications_ (vector<string>) :`** Liste des certifications du pédiatre.
- **`salaireBase (const float) :`** Salaire de base du pédiatre.

Méthodes :

- **`Constructeur`** prenant un nom, un nombre d'enfants soignés et un niveau de spécialité.
- **`getNbEnfantsSoignes(), getCertifications()` :** Accesseurs.
- **`setNbEnfantsSoignes()` :** Mutateur.
- **`operator+=` :** Ajoute une certification.
- **`calculerSalaire()` :** Calcule le salaire en fonction des enfants soignés et des certifications.
- **`examinerPatient()` :** Examine un patient et met à jour les informations.
- **`operator<<` :** pour l'affichage.

Classe Patient

Attributs :

- **`nom_ (string)` :** Nom du patient.
- **`age_ (int)` :** Âge du patient.
- **`numeroSalle_ (int)` :** Numéro de la salle du patient.
- **`typeSoins_ (TypeSoins)` :** Type de soins requis.
- **`antecedentsMedicaux_ (vector<string>)`** : Liste des antécédents médicaux.
- **`infirmier_ (shared_ptr<Infirmier>)`** : Infirmier assigné au patient.
- **`medecin_ (shared_ptr<Medecin>)` :** Médecin assigné au patient.

Méthodes :

- **`Constructeur`** prenant un nom et un âge.
- **`getNom(), getSalle(), getTypeSoins()`** : Accesseurs.
- **`setNom(), misAjourSalle(), misAjourTypeSoin()`** : Mutateurs.
- **`ajouterAntecedent()`** : Ajoute un antécédent médical.
- **`assignerInfirmier(), assignerMedecin()`** : Assignent un infirmier ou un médecin au patient.
- **`afficher(ostream& out)`** : Affiche les informations du patient.
- **`operator<<` :** pour l'affichage.

Classe HopitalPoly

La classe **`HopitalPoly`** représente un hôpital. Elle gère une liste de médecins et d'infirmiers. Elle permet d'ajouter, de supprimer des médecins et des infirmiers, et de calculer le coût salarial total de l'hôpital.

Attributs :

- **`nom_`:** Le nom de l'hôpital.
- **`medecins_`:** Une liste de médecins de l'hôpital.
- **`infirmiers_`:** Une liste d'infirmiers de l'hôpital.

Méthodes :

- **`operator+=`:** Ajoute un médecin ou un infirmier à l'hôpital.
- **`operator-=`:** Supprime un médecin ou un infirmier de l'hôpital.
- **`operator==`:** Compare deux hôpitaux selon leur nom.
- **`coutSalarialTotal()`:** Calcule le coût salarial total de l'hôpital.

Main.cpp

Les tests unitaires pourront être exécutés quand vous aurez des implémentations pour les classes. Pour les exécuter, vous devrez compiler le projet et exécuter le fichier main.cpp.

Fonctionnalités et Objectifs

- La classe **`Liste<T>`** permet de gérer une collection d'objets de type générique, avec des méthodes pour ajouter, supprimer et rechercher des éléments.
- La classe **`Infirmier`** et **`Medecin`** sont des classes dérivées de **`Employe`** et sont capables de calculer le salaire, d'afficher leurs informations et de gérer leurs patients.
- Les médecins ont des spécialités, qui peuvent être utilisées pour déterminer leur salaire et la nature de leurs soins.
- Implémenter les méthodes pour gérer les interactions entre les médecins, les infirmiers et les patients (ex. **`examinerPatient()`, `opererCoeur()`, `opererPatient()`.**

Correction

Veuillez remettre tous les fichiers .cpp SAUF le main.cpp, testSuite.cpp et remettez aussi les autres fichiers dans un fichier .zip sous format Matricule1_Matricule2_TP3.zip

Également, SVP ne mettez pas de commentaires dans le code. Enlevez les TODO après les avoir faits.
La correction du TP3 se fera sur 20 points.

Voici les détails de la correction :

- (4 points) Compilation du programme ;
- (4 points) Exécution du programme ;
- (2 points) Qualité du code;
- (10 points) Comportement exact de l'application