Conception et Programmation Orientée Objet C++

ECE – INGE2
R. FERCOQ
19/10/2019

Durée : 1H30

Calculatrices non autorisées. Aucun document.

Répondez directement sur l'énoncé dans les cadres prévus.

1. Conception orientée objet et UML (pas de code C++) *7 points (a:2 b:5)*

Le but de cet exercice est la conception d'un **diagramme de classes UML** à partir d'un **CDC**. Le diagramme de classes demandé devra couvrir toutes les classes et tous les attributs mais il n'est **pas** demandé de remplir les parties méthodes. Lisez attentivement le CDC...

Cahier Des Charges

On étudie une **application** de suivi de **patients** comme **monsieur Martin** atteints d'affections chroniques nécessitant des **hospitalisations** dans des **hôpitaux** comme **Cochin** ou **Necker**. Chaque **patient** a un **dossier médical** qui permet de connaître chaque **hospitalisation** avec des **descriptions** textuelles des **médecins**. Chaque **médecin** peut avoir une ou plusieurs **spécialités** tel que **cardiologie** ou **rhumatologie** ou **dermatologie**… (il y en a des dizaines). L'**application** connaît directement tous les **patients**, tous les **hôpitaux**, tous les **médecins**, toutes les **spécialités**. Il est possible de connaître tous les **médecin** d'une certaine **spécialité** ou travaillant dans un certain **hôpital**. A un moment donné un **patient** est dans un **hôpital** ou n'est pas hospitalisé. On peut connaître les **patients** d'un certain **hôpital**. Chaque **médecin** a en charge des **dossiers médicaux courants** (pas d'accès direct aux **patients**). Chaque **patient** et chaque **médecin** a un **nom** et un **prénom**, ainsi qu'une **adresse** (**numéro** et **rue** et **code postal**). Chaque **hôpital** a une **adresse**. Chaque **patient** a une **date de naissance**. Une **hospitalisation** a une **date de début** et une **date de fin** (connue ou pas) ainsi qu'une référence à un **hôpital**.

a) *Ranger **chaque texte en gras** du CDC ci-dessus dans une/des case(s) du tableau ci-dessous. Un même texte peut éventuellement apparaître plusieurs fois avec des rôles différents. Grouper horizontalement une classe avec ses attributs et objets qui peuvent y correspondre.*

Classes	Attributs	Objets
Application	*Attributs entre crochets : **facultatifs** (associations)* [médedins] [hopitaux] [patients] [spécialités]	
Hôpital	[patients] [médecins] adresse	Cochin, Necker
Patient	[dossier médical] nom, prénom date_de_naissance, adresse	monsieur Martin
Médecin	[hôpital] [spécialité(s)] [dossiers médicaux courants] nom, prénom, adresse	
Spécialité	*nom_de_la_spécialité (facultatif)*	cardiologie, rhumatologie dermatologie
Hospitalisation	[hôpital] date_de_début, date_de_fin descriptions	
Dossier médical	[hospitalisations] [patient]	
Date	jour, mois, année	
Adresse	numéro, rue, code_postal	

b) *Faites sur cette page 2 diagrammes de classes, en bas le diagramme de classes des types valeurs, en haut le diagramme de classe des types entités. On ne demande* ***pas*** *de justification pour déterminer quels sont les « types valeurs » et quels sont les « types entités ».*
Il n'est ***pas*** *demandé de faire apparaître les méthodes.* ***Toutes*** *les autres infos du diagramme de classes doivent êtres présentes en respect de la norme de notation UML. Pour rappel les types de base en notation UML sont Integer, Real, Boolean, String. Prévoir des boites* ***larges****.*

Diagramme de classes des types entités

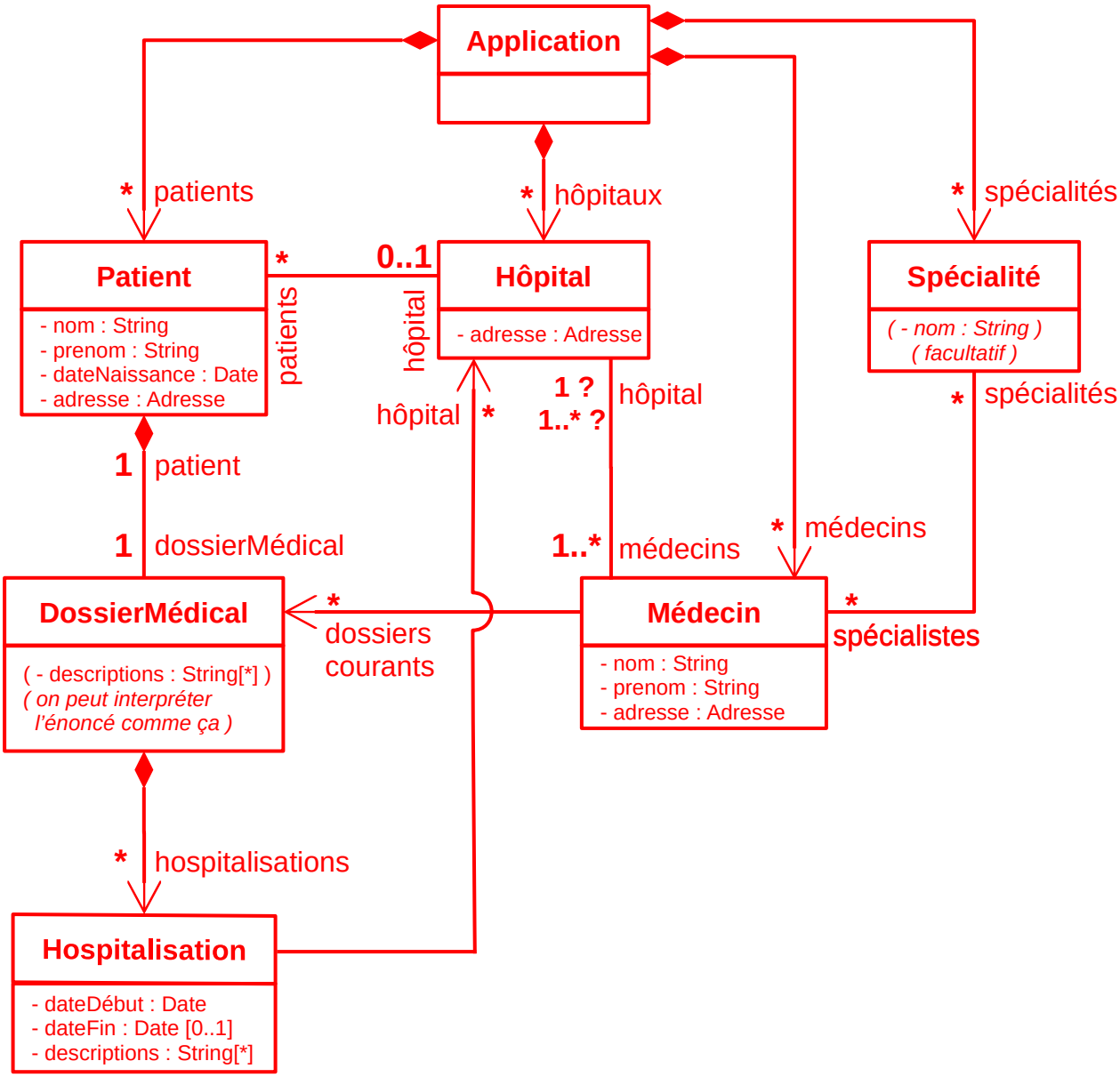
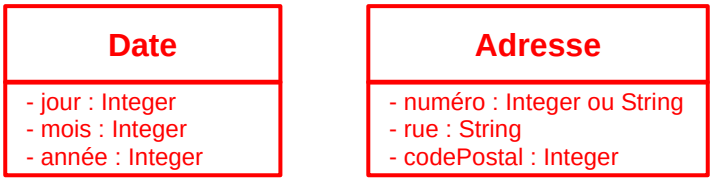


Diagramme de classes des types valeurs



On peut à la rigueur accepter ***Hospitalisation*** *ici car il n'a pas (strictement) de "lien entrant partagé"*

Cependant la classe Hospitalisation référence des types entités et participe à la "circulation" entre entités du modèle il est donc ***maladroit*** *de ne pas la considérer comme un type entité*

2. QCM questions de cours *6 points (0.5 point par question)*

***Entourer** une seule bonne réponse (entourer le 1. 2. 3. ou 4.) : 0.5 point*
Plusieurs réponses ou mauvaise réponse ou baratin : 0 point
Pas de stylo rouge SVP. Pas de justification demandée.

a) En programmation orientée objet, quand on parle de l'interface d'un objet :
1. C'est l'accès publique aux méthodes de saisie et d'affichage pour l'utilisateur du logiciel
✓ **2.** C'est l'ensemble des méthodes publiques qui permettent au code client de piloter l'objet
3. C'est l'ensemble des données de l'objet visibles par l'utilisateur du logiciel
4. C'est l'ensemble des données privées qui ne sont utilisables que par le code objet

b) Dans un affichage complexe avec plusieurs << … << … << chaînés, le bloc **`std::endl`**
1. Indique la fin de la chaîne d'affichage std::endl (end of linked), obligatoire à la fin
2. Indique la fin de la chaîne d'affichage std::endl (end of linked), facultatif
✓ **3.** Indique un retour à la ligne dans la console (end of line), facultatif
4. Indique un retour à la ligne dans la console (end of line), obligatoire à la fin

c) Dans un code C++ on voit une méthode de classe A qui prend un paramètre de type A :
1. C'est forcément une anomalie : le seul objet de type A dans une méthode de A est *this
2. C'est possible mais dans ce cas il faut utiliser ce paramètre à la place de l'objet cible this
✓ **3.** C'est possible, l'objet cible this peut coexister avec un autre objet de type A en paramètre, et le code pourra accéder directement aux attributs privés de ce paramètre
4. C'est possible, l'objet cible this peut coexister avec un autre objet de type A en paramètre, mais le code ne pourra pas accéder directement aux attributs privés de ce paramètre

d) L'encapsulation des données privées derrière des méthodes publiques permet d'**éviter** :
✓ **1.** D'avoir un code client qui dépend de l'organisation des données privées qui peut varier et qui pourrait envoyer de mauvaises valeurs à l'objet sans détection du problème
2. D'avoir un code client qui dépend de l'organisation des méthodes publiques qui peut varier et qui pourrait utiliser l'objet dans de mauvais algorithmes sans détection du problème
3. D'avoir un code client qui dépend de l'organisation des données privées qui peut varier et qui pourrait utiliser l'objet dans de mauvais algorithmes sans détection du problème
4. D'avoir un code client qui dépend de l'organisation des méthodes publiques qui peut varier et qui pourrait envoyer de mauvaises valeurs à l'objet sans détection du problème

e) Utilisation des chaînes de caractères de type std::string à la place de char chaine[…] du C :
1. Une std::string autorise l'accès au caractère n°i avec [i]. Une std::string ne peut contenir qu'un seul mot (sans espace), pour avoir une phrase il faut std::vector<std::string> phrase;
2. Une std::string n'autorise pas l'accès au caractère n°i. Une std::string peut contenir une séquence quelconque de caractères, espaces et retours lignes inclus.
✓ **3.** Une std::string autorise l'accès au caractère n°i avec [i]. Une std::string peut contenir une séquence quelconque de caractères, espaces et retours lignes inclus.
4. Une std::string n'autorise pas l'accès au caractère n°i. Une std::string ne peut contenir qu'un seul mot (sans espace), pour avoir une phrase il faut std::vector<std::string> phrase;

f) Le qualificatif **`const`** est utilisable :
1. uniquement pour un paramètre de méthode
2. uniquement pour un paramètre de méthode, ou une méthode (l'objet cible est constant)
✓ **3.** uniquement pour un paramètre de méthode, ou une méthode (l'objet cible est constant), ou un paramètre de sous-programme (qui n'est pas une méthode)
4. pour un paramètre de méthode, ou une méthode, ou un paramètre de sous-programme, ou une classe (classe constante)

g) Dans un programme qui gère des relations entre des personnes employées (classe Personne) et les entreprises qui les emploient (classe Entreprise), on a dans la classe Personne un attribut de type pointeur sur entreprise, et un mutateur déclaré de la sorte :
```
void setEmployeur(Entreprise* employeur); // init. ou changer d'employeur
```

Le code client suivant est proposé, les 3 premières lignes semblent fonctionner mais il y a un problème avec les 2 dernières :

```
std::vector<Entreprise> entreprises;
Personne p{"Fercoq"};
entreprises.push_back( Entreprise{"ECE"} );
p.setEmployeur( &entreprises[0] );
entreprises.push_back( Entreprise{"ESCE"} );
```

1. Ça ne compile pas, l'appel passe une référence à Entreprise, le mutateur attend un pointeur
2. Ça ne compile pas, le vecteur "par valeurs" est bloqué en taille dès l'utilisation d'un pointeur sur un élément, le push_back est alors impossible
3. ✓ Ça n'est pas correct, le vecteur "par valeurs" ne conserve pas ses éléments à des adresses stables quand il change de taille avec push_back
4. Ça n'est pas correct, on a indiqué l'employeur de la Personne en utilisant une référence, une référence ne peut plus être modifiée, la personne restera bloquée sur l'employeur initial

h) Dans un programme graphique qui utilise une sortie fichier SVG on veut pouvoir positionner un carré soit en coordonnées cartésiennes (x,y) soit en coordonnées polaires (r, theta). On a les 2 méthodes suivantes dans la classe Svgfile :

```
void Svgfile::addSquare(double x, double y, double size);
void Svgfile::addSquare(double r, double theta, double size);
```

1. Impossible à moins de mettre les paramètres surchargés en dernier (après size)
2. ✓ Impossible parce que les 2 versions surchargées ont les mêmes types de paramètres
3. C'est possible avec la surcharge mais il faut faire l'appel avec les bons noms en paramètres
4. C'est possible avec la surcharge mais il faut se placer dans le bon scope avec ::

i) Un itérateur permet :

1. D'accéder efficacement au rang i avec la notation [i] indifféremment pour un vecteur ou pour une listes chaînées
2. ✓ De parcourir indifféremment un vecteur ou une liste chaînée avec un même code
3. D'accéder efficacement à un rang de type quelconque : ["chaine"] [3,14159] [elephant] ...
4. De parcourir indifféremment un vecteur 1D 2D 3D etc... avec un même code

j) Ce code écrit par un expert :

```
void test(int &x, int &y)
{
    x=y;
}
```

```
int main()
{
    int x = 4;
    int y = 3;
    test(&x, &y);
    y = 5;
    std::cout << x << " " << y;
```

1. ✓ Ne compile pas : les paramètres attendus sont des (références)int et l'appel envoie des int*
2. Compile mais l'appel ne sert à rien parce que les références sont copiées, affiche 4 5
3. Compile et marche par référence : x appelant prend bien la valeur 3 de y, affiche 3 5
4. Compile et marche par référence : la référence x devient la même que y , affiche 5 5

k) La classe A ne contient aucun attribut pointeur et ne fait pas d'allocation. Elle ne déclare pas et ne code pas de destructeur. Quand un objet de type A est déclaré localement :

```
A monObjet{ paramètres éventuels… };
```

1. Sa mémoire n'est pas libérée à la sortie du scope car il n'a pas de destructeur, elle sera libérée automatiquement quand l'exécutable se termine
2. Il a un destructeur implicite, mais qui ne fonctionne à la sortie du scope que si le constructeur par défaut implicite est utilisé à la construction de l'objet
3. Si on a déclaré et codé un/des constructeur(s) explicite(s) il faut coder le destructeur
4. ✓ Le destructeur implicite existe et libère la mémoire automatiquement à la sortie du scope

l) Un paramètre de méthode déclaré référence constante (… `const Type& parametre` …)

1. Oblige l'appelant à fournir un objet constant et l'appelé à ne pas modifier l'objet et éventuellement à donner ce paramètre à des paramètres d'appels eux mêmes déclarés const
2. Oblige l'appelant à fournir un objet constant
3. ✓ Oblige l'appelé à ne pas modifier l'objet et éventuellement à donner ce paramètre à des paramètres d'appels eux mêmes déclarés const
4. Est purement indicatif : en lisant la déclaration et en voyant const l'appelant sait que le passage par référence a été choisi pour éviter une copie, pas pour modifier le paramètre

3. Code C++ *7 points (1 point par question)*

L'objectif est de gérer une collection d'inscrits à une compétition sportive. Il faut coder 2 classes :

→ la classe **Inscrit** avec

3 attributs	2 méthodes
- **email** de type chaîne de caractères - **date de naissance** de type Date - **date d'inscription** de type Date	+ **afficher** sans paramètre, affiche les attributs + **saisir** sans paramètre, saisie les attributs

→ la classe **Competition** avec attribut

1 attribut	3 méthodes
- **inscrits** de type vecteur de pointeurs sur Inscrit	+ **ajouterInscrit** sans paramètre : créer un nouvel inscrit, le saisir, l'ajouter + **afficherInscrits** sans paramètre, affiche tous les inscrits + **destructeur** qui fait son job ...

La classe Date est déjà codée, **il n'est pas demandé de coder/compléter la classe Date** ! Voici les 3 seules méthodes publiques dont vous disposez pour l'utiliser :

```
Date();        void afficher() const;        void saisir();
```

Il n'est demandé **aucun blindage sur les saisies ni aucune fioriture sur les affichage** : saisies et affichages bruts, au plus simple. Ce n'est pas sur ces aspects que vous serez évalué.
Les 2 classes demandées ayant des attributs avec constructeurs par défaut, le constructeur par défaut implicite conviendra : vous déclarerez le constructeur par défaut `=default;` dans la classe, et ainsi vous n'avez **pas besoin de coder ces constructeurs par défaut**.

a) **inscrit.h** écrire ici le code pour la classe Inscrit

```
class Inscrit
{
    private :
        std::string m_email;
        Date m_dateNaissance;
        Date m_dateInscription;

    public :
        Inscrit() = default;
        void afficher() const;
        void saisir();
};
```

b) c) **inscrit.cpp** écrire ici le code d'implémentation des 2 méthodes de la classe Inscrit

```
void Inscrit::afficher() const
{
    std::cout << m_email;          std::cout << std::endl;
    m_dateNaissance.afficher();    std::cout << std::endl;
    m_dateInscription.afficher();  std::cout << std::endl;
}
void Inscrit::saisir()
{
    std::cout << "Email SVP : ";
    std::cin >> m_email; // Pas d'espaces dans un email, pas besoin de getline
    //getline(std::cin, m_email); // ADL, ici std:: est facultatif !
    std::cout << "Date naissance SVP : ";
    m_dateNaissance.saisir();
    std::cout << "Date inscription SVP : ";
    m_dateInscription.saisir();
}
```

Ne pas pénaliser l'absence des mises en formes et messages: codes en gris considérés ici comme facultatifs

d) **competition.h** écrire ici le code pour la classe Competition

```
class Competition
{
    private :
        std::vector<Inscrit*> m_inscrits;

    public :
        Competition() = default;
        ~Competition();
        void ajouterInscrit();
        void afficherInscrits() const;
};
```

e) f) g) **competition.cpp** écrire ici code d'implémentation des 3 méthodes de la classe Competition

```
Competition::~Competition()
{
    for (size_t i=0; i<m_inscrits.size(); ++i)
        delete m_inscrits[i];
}

void Competition::ajouterInscrit()
{
    Inscrit* nouveau = new Inscrit;
    nouveau->saisir();
    m_inscrits.push_back(nouveau);
}

void Competition::afficherInscrits() const
{
    std::cout << std::endl;
    for (size_t i=0; i<m_inscrits.size(); ++i)
        m_inscrits[i]->afficher();
}
```