Conception et Programmation Orientée Objet C++

ECE – INGE2
R. FERCOQ
17/12/2019

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:1 b:6)*

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 ni les rôles des associations.
Seuls les termes en gras seront modélisés. Lisez attentivement le CDC...

Cahier Des Charges

Pour optimiser les dépenses du système de santé on étudie une application gouvernementale visant à estimer l'efficacité de divers **traitements** médicaux en fonction des pathologies déclarées dans les **diagnostiques** que des praticiens font sur la base de **symptômes** constatés chez leurs **patients**. Les données sont utilisées de façon statistique : aucune donnée nominative n'est enregistrée, les praticiens ne sont pas intégrés au modèle.

Une **consultation** est associée de façon unique à :

- un **patient** : son **age** (en années) et le fait qu'il soit **fumeur** ou non fumeur.
- un **profil clinique** : 1 à plusieurs **symptômes**
- un **diagnostique** : indication sous forme textuelle d'une **pathologie**.
- une **action thérapeutique** : ensemble de 0, 1 ou plusieurs **traitements**
  d'autre part à chaque action thérapeutique est associée de façon facultative une indication d'**évolution** de la pathologie avec une **description** textuelle et un nombre de **jours**

Chaque **symptôme** peut être concrètement :

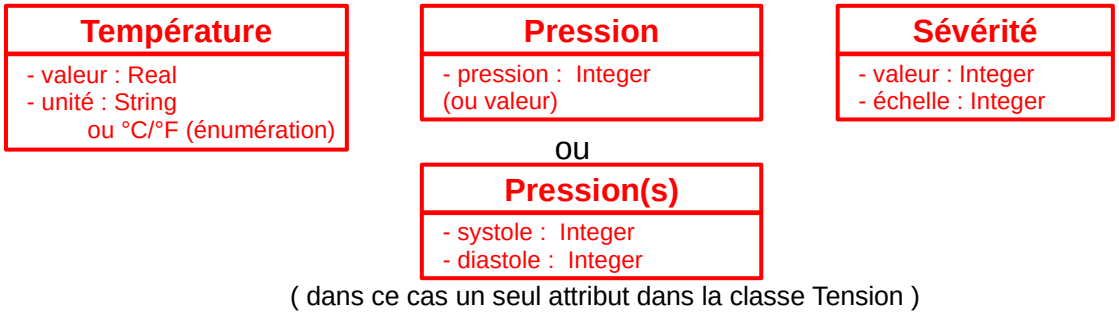
- soit une **fièvre** : **température** enregistrée avec sa **valeur** et son **unité** qui est "°C" ou "°F"
- soit une anomalie de **tension** artérielle : 2 **pressions** : **systole** et **diastole** en mmHg (entiers)
- soit une **douleur** : **zone** douloureuse sous forme textuelle et **sévérité** sous forme d'un couple d'entiers **valeur** et **échelle** rapportés par le patient

Chaque **traitements** peut être concrètement :

- soit une **thermothérapie** : **température** de traitement
- soit un **repos** : prescription d'un **nombre de jours** d'arrêt de travail
- soit un **médicament** : **nom** du médicament et **posologie** sous formes textuelles

Au niveau du système on voudra pouvoir connaître et accéder directement aux 3 ensembles de **tous les profils cliniques** et **tous les diagnostiques** et **toutes les actions thérapeutiques** et pouvoir circuler d'un ensemble à l'autre. Les autres navigations devront être restreintes autant que possible.

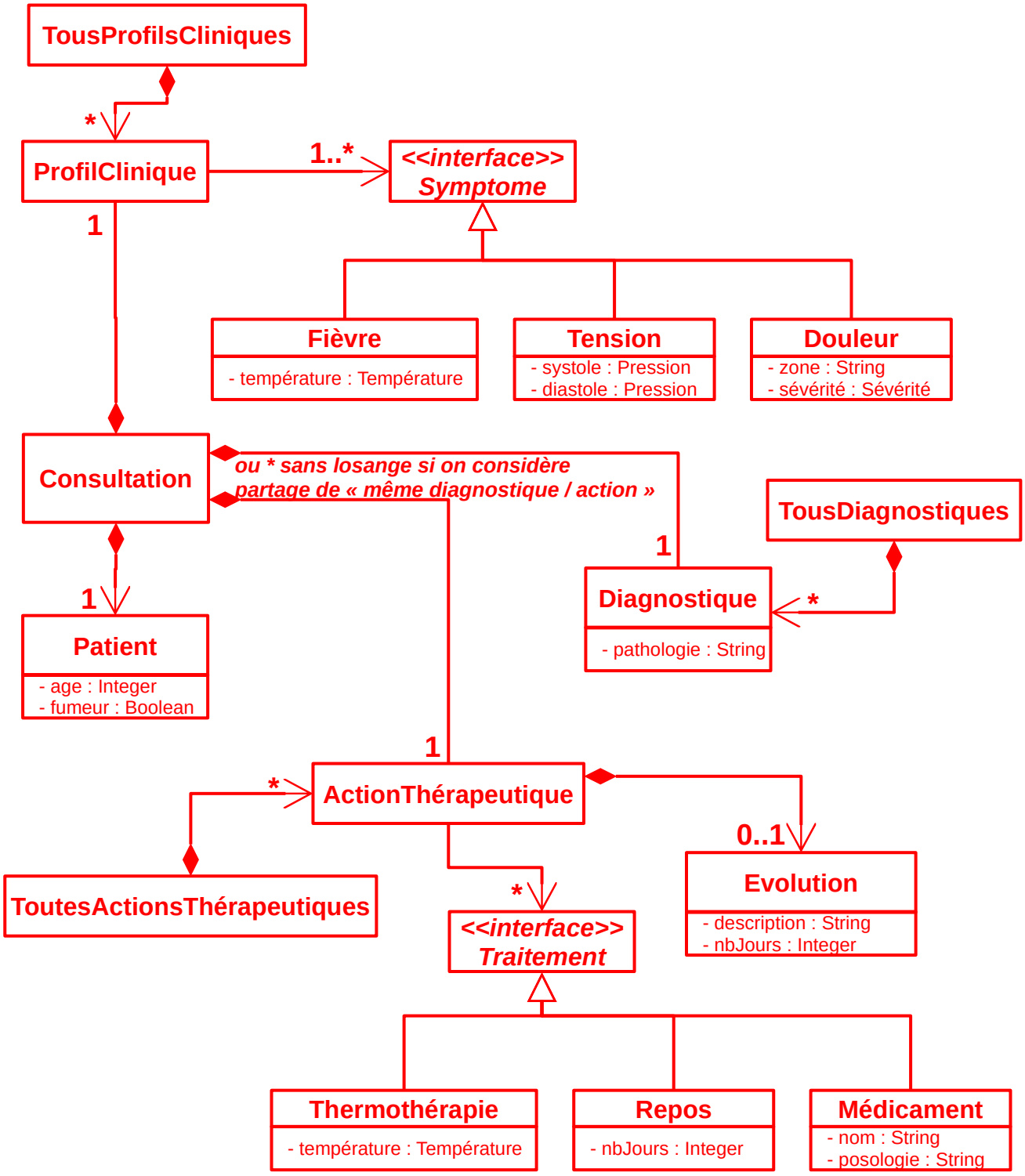
a) *Une analyse préliminaire a conduit à identifier des **types valeurs**. Faire ci dessous le diagramme des types valeurs pour les **températures**, les **pressions** et les **sévérités** de douleurs*



(dans ce cas un seul attribut dans la classe Tension)

b) *Faites sur cette page le diagrammes de classes des types entité.*
Indiquer classes abstraites avec « abstract », classes interface avec « interface ».
*Il n'est **pas** demandé de faire apparaître **ni les méthodes ni les rôles des associations**. **Toutes** les autres infos du diagramme de classes doivent êtres présentes en respect de la norme de notation UML. Pour rappel types de base en notation UML : Integer, Real, Boolean, String.*

Diagramme de classes des types entités



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

<u>Entourer</u> 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) Dans quel cas on a une association avec navigation à double sens entre 2 classes :

1. Si chaque classe hérite de l'autre
2. Si les 2 classes héritent d'une classe abstraite commune
✓ **3.** Si chaque classe possède un(des) pointeur(s) vers l'autre
4. Si le .h de la 1ère inclut celui de la 2ème et le .h de la 2ème a une *forward declaration* de la 1ère

b) Si on dessine un diagramme UML à partir de l'observation d'un code C++, on met un symbole de composition (losange noirci) de la classe A vers la classe B si :

1. La classe A a un attribut de type valeur B, par adresse ce serait forcément une agrégation
✓ **2.** La classe A a un attribut de type valeur B, ou bien une adresse vers B mais dans ce dernier cas on voit un delete de l'objet B pointé dans le destructeur de A
3. La classe A hérite de la classe B (composition par héritage)
4. La classe A a une méthode virtuelle (pure), ou un constructeur (non pur) qui prend en paramètre un objet de type B et qui le détruit

c) Un conteneur de la bibliothèque STL (Standard Template Library) comme vector peut

1. Contenir un mix de données de type quelconque car il est générique
2. Contenir exactement un seul type à la fois dans tous les cas
✓ **3.** Contenir un mix de pointeurs sur des types qui sont soit vers un type de base soit vers des types dérivés car il est compatible avec le polymorphisme
4. Contenir exactement un seul type dérivé d'un type de base à la fois si le type dérivé est polymorphe

d) On souhaite afficher avec un itérateur les valeur stockées dans `std::list<int> maListe;`

1. Impossible d'utiliser un itérateur car c'est une liste ! Il faudrait un vector ou une map.

2.
```
for (size_t it=0; it<maListe.size(); ++it)
    std::cout << maListe[it] << std::endl;
```

3.
```
for (size_t it=maListe.cbegin(); it!=maListe.cend(); ++it)
    std::cout << it << std::endl;
```

✓ **4.**
```
for (auto it=maListe.cbegin(); it!=maListe.cend(); ++it)
    std::cout << *it << std::endl;
```

e) Dans la bibliothèque standard, la complexité O(N) caractérise std::list ...

✓ **1.** La complexité parle de temps d'exécution d'un traitement, de quel traitement parle-t-on ?
2. Faux : la liste est aussi efficace que le vector, elle est en O(1) pour stocker N éléments
3. Oui, la liste est « N » fois plus complexe puisque pour chaque élément stocké on a un pointeur vers l'élément suivant
4. Dans une file la liste est en O(N) (il faut retirer tous les éléments), mais pas dans une pile où elle est en O(1) (le sommet)

f) L'inversion du contrôle c'est

1. Quand on réalise une « programmation à l'envers » à travers une exception qui remonte depuis le framework vers l'appelant
2. Quand les instances d'une classe abstraite décident ce que font les attributs concrets
3. Quand la composition est utilisée à la place de l'héritage et qu'on renverse toutes les flèches (B héritait de A, maintenant A est un composant de B)
✓ **4.** Quand un code de framework (écrit avant) appel un code utilisateur (écrit après)

g) *Le début d'une déclaration de classe contient*

```
class Marteau : public Outil, public Pesant { … };
```

✓ **1.** C'est un héritage multiple (mêmes attributs et méthodes que les classes mères)
2. C'est une liste d'initialisation (mêmes constructeurs que les classes mères)
3. C'est une composition facultative par valeur (on désigne l'un ou l'autre)
4. C'est un polymorphisme de composition par valeur (on désigne l'un et l'autre)

h) Le design pattern délégation est utile quand

1. On veut déléguer à la classe de base, càd qu'on appelle une méthode de la classe de base qui a le même nom que la méthode de la classe fille (on préfixe donc par base::)
✓ **2.** Une classe B veut ré-utiliser les méthodes d'une classe A mais pour une raison de cohérence du modèle B ne peut pas hériter de A : à la place on composera A dans B.
3. Une classe B veut utiliser les attributs privés d'une classe sœur A mais on doit passer par la classe mère commune pour accéder finalement aux attributs (accesseurs en virtual)
4. On fait de la composition au lieu de l'héritage d'une classe A interface parente de B par la classe B elle même (interface de récursion, impossible en héritage car circulaire)

i) En C++ la gestion des exceptions passe par une paire try/catch avec les contraintes suivantes

✓ **1.** Le bloc catch suit immédiatement le bloc try dans un même sous-programme
2. Le bloc catch est dans un code appelé directement pas le bloc try
3. Le bloc try est dans un code appelé directement pas le bloc catch
4. Aucune contrainte au niveau des appels, par contre le type du try doit être le même type ou un type plus spécifique (dérivé) que le type du catch

j) En C++ l'ouverture vers un fichier outfile.txt en écriture se fait avec

```
1. std::ofstream& ofs = fopen("outfile.txt");
2. std::ofstream ofs.fopen("outfile.txt");
3. std::ofstream ofs{"outfile.txt"};
4. std::ofstream* ofs = "outfile.txt";
```

✓

k) *Indiquer l'approche **qui ne marche pas** pour faire fonctionner le code client suivant :*

```
Date noel{25, 12, 2019};

noel.serialiser(std::cout); // Affiche la date en console

noel.serialiser(ofs);       // Enregistre la date sur fichier ofs

std::ostringstream oss;     // Écrit la date dans la chaîne str
noel.serialiser(oss);
std::string str = oss.str();
```

1. Templater la méthode serialiser pour traiter tous les types en entrée
2. Surcharger la méthode serialiser pour traiter tous les types en entrée
✓ **3.** Rendre polymorphe (virtual) la méthode serialiser pour recevoir tous les types en entrée
4. Utiliser un flot polymorphe (ostream&) pour recevoir tous les types en entrée

l) *Dans la définition d'un template de classe (classe « templatée ») dans un .h, le paramètre de type T peut être utilisé pour*

✓ **1.** N'importe quelle déclaration qui nécessite un type ! Déclaration des attributs, déclaration des paramètres des méthodes, déclaration des valeurs de retour.
2. Presque n'importe quelle déclaration qui nécessite un type ! Déclaration des attributs, déclaration des valeurs de retour, déclaration des paramètres des méthodes, sauf du constructeur qui lui nécessite un type spécifique « en dur » comme float ou int.
3. Seulement pour les attributs : les types des méthodes doivent toujours être « en dur » (int...)
4. Seulement pour les paramètres et valeurs de retours des méthodes : les types des attributs doivent toujours être « en dur » (int...)

3. Code C++ *7 points (1,5 point par fichier + 1 point printName non dupliqué)*

Le but de cet exercice est la rédaction d'un code **C++ complet** correspondant à **2 classes** : Printer et PrinterSpacing. Le code client (appelant, utilisateur) de ces 2 classes est déjà fourni. À vous de rédiger les classes de telle sorte que ce code appelant donne le résultat demandé :

Code client	Résultat (affichage console)

`int main()`
`{`
`  std::vector<Printer*> xeroxParc;`
 	
`  xeroxParc.push_back(new Printer{"Jane"});`
`  xeroxParc.push_back(new PrinterSpacing{"Tarzan"});`
`  xeroxParc.push_back(new PrinterSpacing{"Cheeta",2});`
 	
`  for (size_t i=0; i<xeroxParc.size(); ++i)`
`  xeroxParc[i]->print("Hello!");`
 	
`  for (size_t i=0; i<xeroxParc.size(); ++i)`
`  delete xeroxParc[i];`
 	
`  return 0;`
`}`
	`Printing to Jane :`
	`Hello!`
	`Printing to Tarzan :`
	`H e l l o !`
	`Printing to Cheeta :`
	`H  e  l  l  o  !`

Pour la classe Printer, la méthode print affiche le nom de l'objet (Printing to …) puis se contente d'afficher la chaîne envoyée en paramètre.

Pour la classe PrinterSpacing, la méthode print affiche le nom de l'objet (Printing to …) puis affiche la chaîne envoyée en paramètre en séparant chaque caractère du suivant par un certain nombre d'espaces. Par défaut ce nombre d'espaces est 1 (cas de l'objet Tarzan). On peut mettre une autre valeur d'espacement en 2ème paramètre du constructeur d'un objet PrinterSpacing (cas de l'objet Cheeta, 2 espaces entre les caractères de la chaîne affichée).

En plus de la méthode `print` indispensable aux appels directs du code client, vous doterez la classe de base `Printer` d'une méthode `printName` de telle sorte que le code "Printing to " n'apparaisse qu'à un seul endroit de l'application (pas de duplication du code : 1 point).

Pour rappel : une std::string peut s'utiliser comme un std::vector<char>, en particulier on peut accéder au caractère n°i d'une chaîne `std::string text` avec `text[i]` …
D'autre part on peut créer là où on en a besoin une chaîne de nb fois le même caractère `'c'` avec la construction suivante : `std::string(nb, 'c')`

Codez les 2 classes dans les cadres correspondants aux noms de fichiers page suivante.
Il n'est pas demandé de commenter le code. Il est demandé de respecter les consignes générales concernant le code des classes. Utilisez des identifiants clairs et sans ambiguïté.

printer.h

```
#include <string>

class Printer
{
    public :
        Printer(std::string name);
        void printName() const;
        virtual void print(std::string text) const;
        virtual ~Printer() = default;

    private : // protected ok
        std::string m_name;
};
```

printer.cpp

```
#include "printer.h"
#include <iostream>

Printer::Printer(std::string name)
    : m_name{name}
{ }

void Printer::printName() const
{
    std::cout << "Printing to " << m_name << " : " << std::endl;
}

void Printer::print(std::string text) const
{
    printName();
    std::cout << text << std::endl;
}
```

printer_spacing.h

```
#include "printer.h"

class PrinterSpacing : public Printer
{
    public :
        PrinterSpacing(std::string name, int spacing=1);
        void print(std::string text) const;

    private : // protected ok
        int m_spacing;
};
```

printer_spacing.cpp

```
#include "printer_spacing.h"
#include <iostream>

PrinterSpacing::PrinterSpacing(std::string name, int spacing)
    : Printer{name}, m_spacing{spacing}
{ }

void PrinterSpacing::print(std::string text) const
{
    printName();
    for (size_t i=0; i<text.size(); ++i)
        std::cout << text[i] << std::string(m_spacing, ' ');
    std::cout << std::endl;
}
```