Conception et Programmation Orientée Objet C++

ECE – INGE2
R. FERCOQ
24/11/2018

Durée : 1H30

Calculatrices non autorisées. Aucun document.

Répondez directement sur l'énoncé dans les cadres prévus.

1. Code C++, héritage, polymorphisme *7 points (environ 30 minutes)*

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!");`
 	
`  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 nom 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.

Le code client ne comportait pas de delete,
ce qui était une erreur d'énoncé (new implique delete !)
Donc pas de pénalité pour un oubli du destructeur virtuel
(indispensable pour une classe utilisée polymorphiquement)

printer.h

```
#include <string>

class Printer
{
    public :
        Printer(std::string name);
        void printName();
        virtual void print(std::string text);
        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()
{
    std::cout << "Printing to " << m_name << " : " << std::endl;
}

void Printer::print(std::string text)
{
    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);

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

printer_spacing.cpp

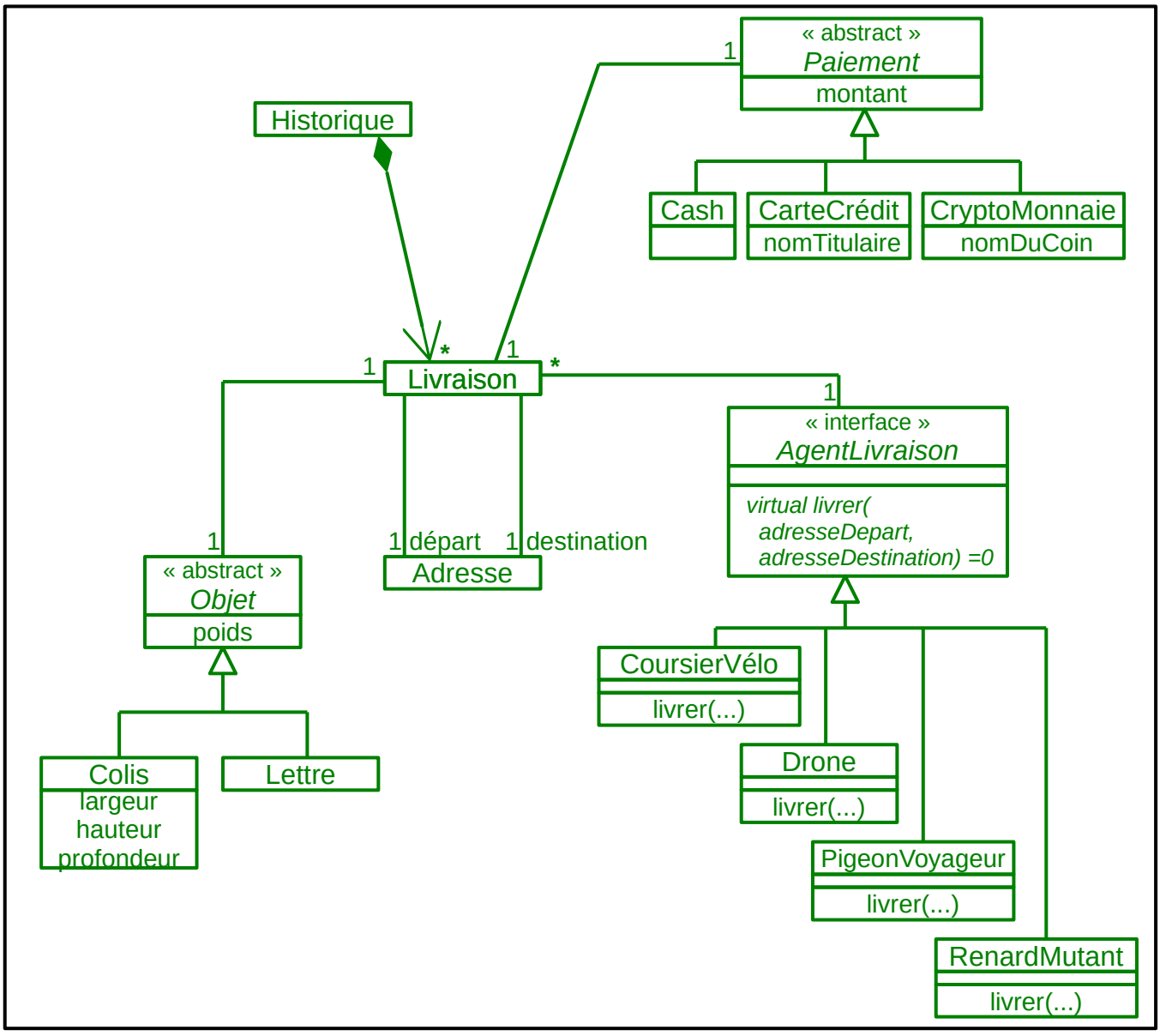
```
#include "printer_spacing.h"
#include <iostream>

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

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

2. Conception orientée objet et UML (pas de code C++) *6 points (environ 25 minutes)*

Faites le **diagramme de classes UML** à partir du **CDC** suivant. Le diagramme de classes demandé devra intégrer toutes les classes, attributs et méthodes qui correspondent à des mots soulignés. Il est demandé de **ne pas ajouter** les attributs et méthodes qui seraient nécessaires pour un projet complet mais **qui ne sont pas soulignés** dans le CDC suivant... Il n'est pas demandé d'identifier ou d'utiliser des « types valeurs » et il n'est pas demandé de préciser les types des paramètres des méthodes ni des attributs (faites au plus simple). Comme il est difficile d'écrire en *italique*, précisez explicitement les classes abstraites avec « abtract », classes interface avec « interface », méthodes virtuelles avec virtual ... et virtuelles pures avec virtual … =0

Cahier Des Charges

Une société futuriste de transport urbain effectue pour ses clients des livraisons. Chaque livraison a un objet qui peut être un colis (poids, largeur, hauteur, profondeur) ou une lettre (poids) qui sera pris à une adresse de départ et livré à une adresse destination. L'agent de livraison pourra être un coursier vélo ou bien un drone mais l'entreprise envisage plus tard d'utiliser aussi des pigeons voyageurs et des renards mutants. Le seul point commun entre tous ces agents de livraison est qu'ils disposent tous d'une méthode livrer qui prend en paramètre une adresse de départ et une adresse de destination, ils sont par ailleurs totalement autonomes en navigation. Chaque livraison donne lieu à un paiement d'un certain montant en euros qui peut être soit en cash (sans détail) soit en carte de crédit (le nom du titulaire est enregistré) soit en crypto-monnaie (le nom du *coin* est enregistré, BTC, XRP, ETH …). Chaque livraison est enregistrée dans un historique, la navigation historique → livraison est la seule qui est à sens unique. L'effacement de l'historique (en cas de contrôle fiscal malvenu) entraîne l'effacement des livraisons.



3. Design pattern, <u>dépend de l'exercice 2 précédent</u> ***1 points (environ 5 minutes)***

Dans le diagramme de classes précédent (exo 2), identifiez (nommez) un design pattern bien connu, précisez les classes impliquées dans ce design pattern et les rôles qu'elles y jouent, indiquez brièvement les avantages de ce pattern.

Clairement la classe Livraison **délègue** (pattern délégation) respectivement le paiement et la livraison effective aux classes Paiement et AgentLivraison.
-> 0.5/1 pour avoir désigné seulement le pattern **délégation** (vers l'un et/ou l'autre).
Dans un cas comme dans l'autre, la délégation se fait vers une classe abstraite ou interface (non instanciée) et on a donc le choix d'une **stratégie** de livraison physique (vélo, drone...) et d'une **stratégie** de paiement (cash, carte de crédit...) :
On a 2 mises en œuvre du pattern stratégie (ou un pattern stratégie avec 2 stratégies).
-> 1 point pour avoir bien désigné et décrit le pattern **stratégie** (vers l'un et/ou l'autre).

4. QCM questions de cours ***6 points (0.5 point par question, environ 30 minutes)***
<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";
```

✓ 3

k) *La façon efficace en C++ de faire marcher le code client suivant implique*

```
Date noel{"25/12/2018"};
noel.serialiser(std::cout); // Affiche la date en console
noel.serialiser(ofs);       // Enregistre la date sur fichier
std::string str;
noel.serialiser(str);       // Écrit la date dans la chaîne str
```

1. De templater la méthode serialiser pour traiter tous les types en entrée
2. De surcharger la méthode serialiser pour traiter tous les types en entrée
3. De polymorphiser (virtual) la méthode serialiser pour recevoir tous les types en entrée
✓ **4.** D'utiliser un type 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...)