ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation Orientée Objet, sections MA et PH
Chappelier J.-C.

Programmation Orientée Objet (SMA/SPH) :
CORRIGÉ DE LA
SÉRIE NOTÉE

11 avril 2019

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre série annulée dans le cas contraire.

1. Vous disposez de 1h45 pour faire cette série notée (9h15 - 11h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.
   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; utilisez aussi le verso des feuilles, **MAIS** n'utilisez *que* le verso de la feuille sur laquelle se trouve la question, et non **pas** celui de la feuille précédente !
   Ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
   Si nécessaire, il y a des pages supplémentaires en fin de copie.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. Cette série notée ne comporte qu'un seul exercice en cinq questions, plus deux questions subsidiaires (bonus).

Exercice 1 – Un séjour au soleil [sur 99 points]

On cherche ici à écrire un programme pour gérer le système de réservation [1] d'un hôtel. Une réservation portera sur trois choses : le type de chambre, la formule de repas (nombre par jour et régime alimentaire [2]) et les extras choisis (sauna, tennis, ...).

Nous allons commencer par nous intéresser aux extras, puis aux types de chambres, aux repas et l'on finira par les réservations proprement dites.

Question 1 – Prestations [sur 9 points]

Mais avant tout cela, toutes ces prestations [3] (chambre, repas, extra, réservation) ayant un prix, on souhaite les modéliser de façon commune : définissez ici une classe `Prestation` qui comprend une méthode `prix()` dont le comportement est propre à chaque prestation et n'a pas de définition générale. Cette méthode retournera un `double` et ne modifiera pas la prestation concernée.

Solution minimale :

```
class Prestation
{
public:
    virtual double prix() const = 0;
    virtual ~Prestation() {}
};
```

Solution avancée :

```
class Prestation
{
public:
    virtual double prix() const = 0;
    virtual ~Prestation() = default;

    // optional (advanced):
    Prestation()                              = default;
    Prestation(Prestation const&)             = default;
    Prestation(Prestation&&)                  = default;
    Prestation& operator=(Prestation const&) = default;
    Prestation& operator=(Prestation&&)       = default;
};
```

Question 2 – Prestations « extra » [sur 10 points]

L'hôtel souhaite offrir à ses clients des prestations « extra » en plus des chambres et des repas : piscine, sauna, tennis, golf, etc. Toutes ces prestations supplémentaires (« extra ») n'ayant pas de comportement particulier, il suffit de les modéliser par un nom et un prix (fixes et connus lors de leur construction).

En haut de la page suivante, définissez **complètement** la classe `Extra` permettant de représenter de telles « prestations supplémentaires ».

```
class Extra : public Prestation
{
public:
    Extra(const string& name_, double price)
        : name(name_), pricePerDay(price)
    {}

    virtual double prix() const override
    {
        return pricePerDay;
    }

private:
    const string name;
    const double pricePerDay;
};
```

Question 3 – Chambres [sur 36 points]

Au niveau des chambres, l'hôtel souhaite offrir trois types de prestations : des chambres classiques, des chambres « confort » et des chambres de luxe.

Toutes ces chambres sont caractérisées par un nombre de lits et un nombre de salles de bains. Elles ont de plus accès à une liste d'« extra » (cf Question 2) qui leur est propre. Les chambres « confort » sont comme des chambres classiques, mais ont en plus un prix forfaitaire pour les prestations « extra » (son utilisation est expliquée ci-dessous). Les chambres « de luxe » sont exactement comme les chambres classiques, sauf que le calcul de leur prix est différent.

Lors de la création d'une chambre (quelle qu'elle soit), on précisera son nombre de lits (2 par défaut) et son nombre de salles de bains (1 par défaut). On ne fournira par contre aucun « extra » lors de la création d'une chambre. Les « extra » seront ajoutés (après construction) au moyen d'une méthode `ajoute_extra()`.

Pour le calcul des prix :

- le prix des chambres classiques vaut 60 fois le nombre de lits plus 40 fois le nombre de salles de bain, plus la somme des prix de toutes les prestations « extra » de la chambre ;
  les valeurs 60 et 40 ci-dessus devront être définies comme attributs de classe ;
- le prix des chambres « confort » est exactement le même que celui des chambres classiques, sauf que l'on rajoute le prix forfaitaire (dont on parlait ci-dessus) et que l'on soustrait le prix du premier « extra » (s'il existe, sinon on ne fait qu'ajouter le prix forfaitaire) ;
- le prix des chambres « de luxe » est simplement de 1000 francs.

Au dos de cette page et sur la suivante, définissez la ou les classe(s) permettant de représenter les chambres telles que décrit ci-dessus. Limitez-vous à ce que vous considérez être le *strict* nécessaire correspondant à la description donnée ci-dessus.

```
class Room : public Prestation
{
public:
    Room(int noOfBeds = 2, int noOfBathRooms = 1)
        : beds(noOfBeds), bathrooms(noOfBathRooms)
    {}

    virtual double prix() const override
    {
        double p(bedPrice * beds + bathroomPrice * bathrooms);
        for (auto e : extras) p += e->prix();
        return p;
    }

    // certainly a pointer as clearly some Extra can be shared (swimingpool)
    void addExtra(Extra* e)
    {
        extras.push_back(e);
    }

protected:
    // subclass Confort needs to access this:
    vector<Extra*> extras;

private:
    int beds; //number of beds/persons
    int bathrooms;

    static constexpr double bedPrice      = 60.0;
    static constexpr double bathroomPrice = 40.0;
};

class Confort : public Room
{
public:
    using Room::Room; // get ctors back in the gamer
    virtual double prix() const override
    {
        double p(Room::prix() + extraPrice);
        // first extra is free
        if (extras.size() != 0) p -= extras[0]->prix();
        return p;
    }
private:
    static constexpr double extraPrice = ...; // une valeur non connue
};

class Luxury : public Room
{
public:
    using Room::Room; // get ctors back in the gamer
    virtual double prix() const override
    {
        return 1000.0;
    }
};
```

Notes :

1. ici c'est la « chambre classique » qui est super-classe des deux autres, mais une solution avec une superclasse abstraite « chambre » ayant trois sous-classes serait aussi possible ; mais **attention à ne pas dupliquer de code !**
2. le prix forfaitaire est ici donné comme attribut de classe ; il pourrait aussi être un attribut d'instance (pas spécifié dans la donnée) si l'on imagine que le prix du forfait dépend de la chambre elle-même

Question 4 – Repas [sur 24 points]

L'hôtel offre également des prestations de restauration. Il y a trois formules de repas selon le régime alimentaire [4] : végétalien [5], végétarien ou standard.

Toutes les formules de repas ont un prix forfaitaire et un nombre de repas par jour. Par défaut, le prix total (les formules de repas sont des prestations) est simplement le produit du prix forfaitaire par le nombre de repas par jour.

[...]

Définissez ci-dessous et en face *tout* ce qui vous semble être nécessaire à la représentation des repas végétariens. Pour simplifier dans cette série notée, on **ne** vous demande **pas** de définir quoique ce soit de *spécifique* au sujet des repas végétaliens ("*vegan*") ni standards, mais ils existent effectivement dans le programme complet envisagé.

```
class Food : public Prestation
{
public:
    Food(double p, int n = 3)
        : unitPrice(p), dishesPerDay(n)
    {
        if (dishesPerDay < 1) dishesPerDay = 1;
        if (dishesPerDay > 3) dishesPerDay = 3;
    }

    virtual double prix() const override
    {
        return unitPrice * dishesPerDay;
    }

private:
    const double unitPrice;
    int dishesPerDay;
};

class Vegetarian : public Food
{
public:
    Vegetarian(int d = 3, int eggs_ = 1)
        : Food(25.0, d), eggs(eggs_)
    {}

    virtual double prix() const override
    {
        return Food::prix() + eggs * eggPricePerDay;
    }

private:
    unsigned int eggs;
    static constexpr double eggPricePerDay =  3.0;
};
```

Question 5 – Réservations [sur 20 points]

On souhaite enfin pouvoir représenter les réservations [6] elles-mêmes. Une réservation comprendra : une chambre, un ensemble de formules de repas et un nombre de jours.

Au niveau de l'initialisation, on ne fournira que la chambre et le nombre de jours ; l'ensemble des formules repas sera fourni ensuite, après initialisation, en utilisant une méthode que nous ne vous demandons pas de coder ici.

Le prix total d'une réservation (qui est aussi une prestation), est simplement la somme du prix de la chambre et du total des prix des formules repas, le tout multiplié au final par le nombre de jours de la réservation.

On vous demande enfin de faire le nécessaire de sorte que le code « `r -= i` » pour une réservation `r` et un entier `i`, diminue de la valeur de `i` le nombre de jours de réservation de `r` sans toutefois descendre en dessous de 1. Par exemple, si le nombre de jours de `r` est 3 et la valeur de `i` est `1`, alors le nombre de jours de `r` après « `r -= i` » sera 2 ; si par contre `i` vaut 4, le nombre de jours de `r` après « `r -= i` » vaudra 1.

Définissez ici et sur la page suivante, la ou les classe(s) permettant de représenter les réservations telles que décrites ci-dessus. Limitez-vous à ce que vous considérez être le *strict* nécessaire correspondant à la description donnée ci-dessus.

```
class Booking : public Prestation
{
public:
    Booking(const Room& r, int d = 1) // or Room*
        : room(r), totalDays(d)
    {}

    Booking& operator -=(int days)
    {
        totalDays -= days;
        if (totalDays < 1) totalDays = 1;
    }

    virtual double prix() const override
    {
        double total(room->prix()) ;
        for (auto const& food : meals) {
            total += food->prix();
        }
        return total * totalDays;
    }

private:
    const Room& room; // or Room*
    int totalDays;
    vector<Food *> meals; // or unique_ptr, see text
};
```

Notes :

1. suivant comment on voit la chose, la chambre peut soit être un pointeur à la C, soit une référence (ou à la rigueur un `shared_ptr`, mais c'est hors du niveau du cours) ;
2. de même, suivant le sens que l'on donne à la liste des repas (non clairement spécifié dans la donnée), on peut soit avoir un pointeur à la C, soit un `unique_ptr` (mais en aucun cas une référence !).

Question 6 – Questions subsidiaires [sur 6 points bonus]

Pour finir, deux questions subsidiaires (= bonus), plus avancées.

Ne modifiez pas votre code donné dans les réponses précédentes, mais répondez simplement ici aux questions posées.

Question 6.1 – Restriction des formules de repas [sur 3 points bonus]

Dans la conception que vous avez proposée en Question 4, peut-on créer des *instances* d'autres formules de repas que végétariens ?
(Nous ne parlons pas ici ni de repas végétaliens, ni de repas standards, qui ne sont de toutes façons pas l'objet de cette série notée. Nous ne parlons ici que du code que vous avez écrit en réponse à la Question 4.)

Si oui : comment pourriez-vous l'éviter ?
Si non : justifiez pourquoi.

On vise ici bien sûr la classe `Food`. La réponse dépend bien sûr du code donné, mais *a priori* la réponse la plus probable devrait être oui (sauf pour des copies avancées qui y auraient déjà pensé).

Pour éviter la copie il faudrait soit :
- en faire une classe abstraite par ajout d'une méthode virtuelle pure, mais je ne vois pas trop laquelle ici, sauf le destructeur : `virtual  Food() = 0;`
- rendre tous ses constructeurs `protected` (en tout cas le constructeur demandé, et ce serait mieux de penser aussi au constructeur de copie (le remettre comme `default` en `protected`).

Question 6.2 – Copie de réservations [sur 3 points bonus]

Dans la conception que vous avez proposée en Question 5 :

1. peut-on créer des copies de réservation ?
2. Est-ce problématique ? Pourquoi ?
3. Si oui, comment y remédier ?

1. La aussi réponse dépend bien sûr du code donnée, mais *a priori* la réponse la plus probable devrait être oui (sauf pour des copies avancées qui y auraient déjà pensé ou celles qui ont mis des références (chambre)).
2. C'est problématique si l'on autorise des copies de pointeurs :
   - en aucun cas on ne devrait pouvoir copier 2 réservations sur la même chambre
   - et le sens de la copie de surface de listes de repas peut laisser à désirer...
     (cela dépend du sens que l'on donne à ces repas : instance individuelles rééellement servis ou description générique d'une commande de repas)
3. en évitant la copie serait le plus simple (suppression du constructeur de copie et de l'opérateur d'affectation)
   ou alors, sinon, en redéfinissant proprement ces deux méthodes pour ce que l'on veut vraiment faire.