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 Système, sections IN et SC
Chappelier J.-C.

Programmation Orientée Objet (SPH) : CORRIGÉ DU MIDTERM

17 avril 2025

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. Ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.

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, demandez des précisions à l'un(e) des assistant(e)s.

6. Cette série notée ne comporte qu'un seul exercice (en 9 questions ; total : 98 points).

Exercice 1 – Jeu de plateau

Question 1.1 – Les joueurs [sur 14 points]

Commencez par créer la classe `Player` permettant la représentation des joueurs.

Chaque joueur a :

- un nom, qui ne change pas (`"red"`, `"blue"`, `"black"` dans l'exemple fourni) ;
- une certaine quantité d'argent, de type `Currency` que l'on supposera fourni ;
- un certain nombre de points, de type `Points` que l'on supposera également fourni.

Lors de l'initialisation de tout joueur, on doit fournir son nom ; si l'on ne fournit pas sa quantité d'argent, elle sera initialisée à 50 ; et le nombre de points est de toute façon initialisé à 0.

Au moyen d'une méthode `pay()`, chaque joueur devra pouvoir dépenser/perdre [1] un certain montant [2] de son argent :
si le montant à dépenser [3] est plus grand que la somme disponible, on lancera une exception (de votre choix) ;
par ailleurs, pour faciliter le suivi du déroulement de la partie, à chaque fois qu'un joueur dépense de l'argent, on affichera un message au format (voir aussi des exemples dans l'exemple fourni départ) :
`<name> a payé <amount>, reste <total money>`

On doit par ailleurs pouvoir ajouter des points au joueur.

Et n'oubliez pas que les joueurs doivent pouvoir être affichés en utilisant l'opérateur usuel (`<<`), au format donné dans l'exemple de départ.

Réponse :

```
class Player : public Printable
{
public:
  Player(const string& name_, Currency money_ = default_money)
    : name(name_), money(money_), points(0) {}

  void pay(Currency amount) {
    if (amount > money) throw 42; // whatever
    money -= amount;
    cout << name << " a payé " << amount << ", reste : " << money << endl;
  }
  void gain(Points p) { points += p; }

  virtual void display(ostream& out) const override
  { out << name << " (money=" << money << ", points=" << points << ')'; }

private:
  const string name;
  Currency money;
  Points points;

  static constexpr Currency default_money = 50; // optionnel
};
```

1. "*pay*" in English. 2. "*amount*" in English. 3. "*the amount to pay*" in English.

Commentaire : Il n'est pas du tout nécessaire d'avoir des `try`–`catch` à l'endroit des `throw`, bien **au contraire**.
On ne peut pas (sauf dans un bloc `catch()`) utiliser un `throw;` sans valeur derrière : il est nécessaire de `throw` une valeur (de n'importe quel type valide).

Question 1.2 – Les tuiles [sur 71 points]

Ce jeu est composé de différentes tuiles[4] (océans, forêts, villes et « tuiles spéciales ») qui peuvent chacune avoir jusqu'à 6 tuiles voisines (tuiles hexagonales, cf illustration en début de donnée).

Question 1.2.1 – Les tuiles génériques [sur 17 points]

Chaque tuile devra donc avoir un moyen d'accéder à ses six voisins, ainsi qu'une méthode `link()` permettant de connecter un nouveau voisin (voir l'exemple de `main()` fourni au début).

Au départ, toute tuile n'a aucun voisin.

Par ailleurs, chaque tuile aura aussi :

- une méthode `score()` dont le comportement sera spécifique à chaque type de tuiles et que l'on ne sait pas spécifier pour le moment ; cette méthode ne prend pas d'argument, ne modifie pas la tuile et ne retourne aucun argument (il sera expliqué plus tard comment cette méthode fonctionne) ;
- une méthode `is_owner()` qui prend un joueur en argument et retourne vrai ou faux suivant que ce joueur est propriétaire de la tuile ou non ;
  par défaut, pour une tuile quelconque, cette fonction retourne simplement `false` (mais ce comportement pourra être changé pour des tuiles particulières) ;
- une méthode `neighbor_points()` qui prend un joueur en argument et retourne des `Points` ; elle ne modifie pas la tuile en question ; par défaut, pour une tuile quelconque, cette fonction retourne simplement 0 (mais ce comportement pourra être changé pour des tuiles particulières).

Définissez ici la classe `Tile` pour représenter une tuile quelconque.
Pour la méthode `link()` ne mettez ici *que son prototype* ; sa définition fait l'objet de la sous-question suivante (page ci-contre).

Et n'oubliez pas que les tuiles doivent pouvoir être affichées en utilisant l'opérateur usuel (`<<`) ; même si on ne sait pas définir cet affichage pour une tuile quelconque.

Réponse : (vous pouvez répondre en deux colonnes si nécessaire.)

```
class Tile : public Printable {
public:
  Tile() : neighbors{nullptr, nullptr, nullptr, nullptr, nullptr, nullptr} {}

  virtual void score() const = 0;

  virtual bool is_owner(Player const& other) const
  { return false; }

  void link(size_t position, Tile* neighbor);

  virtual Points neighbor_points(Player const & p) const
  { return 0; }

protected:
  array<const Tile*, neighborhood> neighbors;

private:
  void add(size_t position, const Tile* neighbor);
};
```

(voir remarques au dos)

4. "*tiles*" in English.

Cette classe ayant des pointeurs (sur les voisins), il serait bon de penser à la copie et à l'affection (pas besoin de penser à la destruction car on n'est pas propriétaire de ses voisins). On pourrait simplement les interdire, ou, mieux, forcer à `nullptr` les nouveaux voisins.

Par ailleurs, cette classe ayant de nouvelles méthodes virtuelles, il serait bon de définir son destructeur comme virtuel. Cependant, on peut imaginer que cela a déjà été fait au niveau de la classe `Printable` (et donc hérité).

Note : on aurait pu écrire l'initialisation (qui est nécessaire) de la façon plus compacte suivante :
`Tile() : neighbors{} {}`
mais cette syntaxe n'est pas forcément connue au niveau de ce cours.

Commentaire : Comment représenter les voisins ?
On ne peut clairment pas mettre un `Tile` tel quel, cela ne fait aucun sens : on ne « possède » pas ses voisins et comment faire pour une tuile `A` voisine à la fois de `B` et `C` ? Ce seraient des *copies* différentes qui seraient dans `B` et dans `C`, sans aucun lien entre elles.
On cherche donc bien ici à **référencer** des tuiles déjà existantes. C'est le « premier cas » d'utilisation des pointeurs présenté en cours. Dans un tel cas : utilisez des références si vous le pouvez (ici on ne le peut pas, on ne peut pas mettre de références dans un `array`), des pointeurs si vous le devez (c'est le cas ici).
Quels pointeurs ?
Ce **n'**est **pas** un cas d'utilisation pour allocation dynamique et donc les « pointeurs intelligents » n'ont rien à faire ici. La *seule* solution possible est donc bien des pointeurs « à la C ».

Question 1.2.2 – La connexion des tuiles [sur 10 points]

On s'intéresse ici à définir la méthode `link()` permettant de rendre deux tuiles voisines. Cette méthode reçoit une position et « une tuile » (nous mettons ici des guillemets ; libre à vous d'interpréter ce terme comme cela convient au mieux à votre conception ; il y a aussi un exemple dans le `main()`, que vous êtes libre d'adapter).

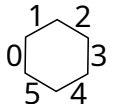
Si la position reçue est supérieure ou égale à 6, lancez une exception (de votre choix).
Si la tuile courante (`*this`) a déjà un voisin à la position reçue en argument, lancez une *autre* exception.
Et sinon, affectez la tuile reçue à la position reçue (dans les voisins de la tuile courante).

Puis faites *exactement* de même dans l'autre sens : ajoutez la tuile courante comme voisine de la tuile reçue, à la position opposée à celle reçue.

Par exemple, si l'on demande d'ajouter la tuile B à la position 3 de la tuile A, on ajoutera *aussi* la tuile A comme voisine à la position 0 de la tuile B (si possible ; sinon, on lancera les *mêmes* exceptions).
Vous supposerez pour cela qu'il existe une fonction

```
size_t opposite(size_t position);
```

qui retourne la position opposée (par exemple qui retourne 3 si elle reçoit 0 et qui retourne 0 si elle reçoit 3).



Un exemple *possible* de numérotation des positions des voisins.
En fait, peu importe : utilisez simplement la fonction `opposite()`.

Réponse : Ce qui importe ici est de ne faire *aucune* copie de code. par exemple :

```
void Tile::add(size_t position, const Tile* neighbor)
{
  if (position >= neighborhood) throw 42; // whatever
  if (neighbors[position] != nullptr) throw 67; // whatever else
  neighbors[position] = neighbor;
}

void Tile::link(size_t position, Tile* neighbor)
{
  add(position, neighbor);
  neighbor->add(opposite(position), this);
}
```

Le C++ étant un langage de portée de classe, on peut aussi faire comme ceci (qui fait l'hypothèse raisonnable que si `x` est valide alors `opposite(x)` l'est aussi ; mais c'est quand même une hypothèse de plus qu'avec la solution ci-dessus) :

```
void Tile::link(size_t position, Tile* neighbor)
{
  if (position >= neighborhood) throw 42; // whatever

  const Tile*& mine   = neighbors[position];
  const Tile*& others = neighbor->neighbors[opposite(position)];
  if ((mine != nullptr) or (others != nullptr))
    throw 67; // whatever else

  mine = neighbor;
  others = this;
}
```

Il y a bien sûr plusieurs façons d'écrire cette dernière version, mais il faut le faire **sans aucun** copié-collé.

Question 1.2.3 – Les océans [sur 4 points]

Les tuiles océans sont les plus simples des tuiles. Elle affichent simplement « océan ». Et au niveau du score, elle ne font rien du tout (mais on peut créer des instances de tuiles océans ; voir le `main()` fourni au début).

Définissez ici votre implémentation des tuiles océans.

Réponse :

```
class Ocean : public Tile
{
public:
  virtual void display(ostream& out) const override
  { out << "océan"; }

  virtual void score() const override
  {} // does nothing: no point to anyone
};
```

Question 1.2.4 – Tuiles possédées par des joueurs [sur 12 points]

Mises à part les tuiles océans, toutes les autres tuiles (forêts, villes et « spéciales ») sont possédées par un joueur lorsqu'elles sont en jeu.

Il est donc nécessaire qu'elles possèdent une référence (non constante) vers un joueur, lequel sera obligatoirement fourni lors de leur initialisation.

Par ailleurs, toutes ces tuiles ont un coût que les joueurs doivent payer à leur construction. Ce coût, dont la valeur par défaut est 0, sera donc passé lors de l'initialisation et de suite prélevé au joueur propriétaire (au moyen de sa méthode `pay()`).

De plus, la méthode `is_owner()` de ces tuiles doit se comporter correctement : répondre « vrai » si son argument correspond au propriétaire de la tuile et « faux » sinon.

Définissez ici une classe `Property` pour représenter de telles tuiles.

Peut-on créer des instances de `Property` ?
Répondez par oui ou par non et **justifiez** *brièvement* votre réponse.

Réponse :

```
class Property : public Tile
{
public:
  Property(Player& p_, Currency cost = 0) : p(p_) { p.pay(cost); }

  bool is_owner(Player const& other) const
  { return &other == &p; }

protected:
  Player& p;
};
```

Commentaire : Il n'est absolument pas nécessaire d'avoir un attribut « coût » ici.

Commentaire : Comment représenter le propriétaire ?
On ne peut clairment pas mettre un `Player` tel quel, cela ne fait aucun sens : la tuile ne « possède » pas son propriétaire et comment faire pour deux tuiles ayant le même propriétaire ? Ces deux tuiles auraient des *copies* différentes de leur propriétaire, sans aucun lien entre elles.
On cherche donc bien ici à **référencer** des `Player` déjà existants. C'est (à nouveau) le « premier cas » d'utilisation des pointeurs présenté en cours. Dans un tel cas : utilisez des références si vous le pouvez, des pointeurs si vous le devez.
Peut-on ici utiliser une référence ?
C'est une question de point de vue, non spécifié dans la donnée : qu'imposent les références ?
Deux choses : (a) de référencer des objets déjà existants au préalable ; et
(b) de ne pas pouvoir référencer autre chose que l'objet référencer au départ.
Est-ce que les propriétaires préexistent aux tuiles ? On peut ici supposer que oui avec tout ce qui a été décrit.
Est-ce qu'une tuile peut changer de propriétaire en cours de jeu (« conquête ») ? Ce n'est pas précisé.
– Si l'on supposer que non, alors on peut utiliser une référence (comme dans le corrigé ci-dessus) ;
– si l'on supposer que oui, alors il faudra utiliser un pointeur « à la C ».

Commentaire : `is_owner()` : qu'est-ce qui identifie *toujours* de façon unique un objet en C++ ?
Son adresse !

Question 1.2.5 – Les forêts [sur 8 points]

Les forêts sont des tuiles ayant un joueur propriétaire et dont le coût est 7.

Au niveau de l'affichage, les forêts affichent simplement : « forêt de » suivi de l'affichage de leur propriétaire.

Au niveau du `score()`, elles apportent un `gain()` de 1 point à leur propriétaire.

Au niveau de `neighbor_points()` enfin, elles retournent 1 si le joueur passé en argument est le propriétaire de la tuile et 0 sinon.

Définissez ici une classe `Forest` pour représenter de telles tuiles.

Réponse :

```
class Forest : public Property
{
public:
  Forest(Player& p_) : Property(p_, 7) {}

  virtual void score() const override
  { p.gain(1); }

  virtual void display(ostream& out) const override
  { out << "forêt de " << p; }

private:
  virtual Points neighbor_points(Player const & other) const override
  { return is_owner(other) ? 1 : 0; }
};
```

Note : il n'est pas du tout nécessaire que `neighbor_points()` soit `private` ; elle peut rester `public` ici ; c'est juste pour montrer qu'elle *pourrait* aussi être mise en `private`.

Question 1.2.6 – Les tuiles spéciales [sur 10 points]

Les tuiles spéciales sont des tuiles ayant un joueur propriétaire et dont le coût doit obligatoirement être spécifié (pas de valeur par défaut). Elles possèdent de plus un nom (qui ne sera pas modifié) et un nombre de points qui doit être fourni lors de l'initialisation et qui sera utilisé dans le `score()`.

Au niveau de l'affichage, les tuiles spéciales affichent leur nom, suivi de « de », puis de l'affichage de leur propriétaire.

Au niveau du `score()`, elles apportent à leur propriétaire un `gain()` de leur nombre de points (celui donné lors de leur initialisation).

Pour tout le reste, elles ne font rien de plus (`is_owner()` comme toute tuile à propriétaire et `neighbor_points()` comme toute tuile générale).

Définissez ici une classe `Special` pour représenter de telles tuiles.

Réponse :

```
class Special : public Property
{
public:
  Special(const string& name_, Player& p_, Currency cost, Points pts_)
    : Property(p_, cost), name(name_), pts(pts_) {}

  virtual void display(ostream& out) const override
  { out << name << " de " << p; }

  virtual void score() const override
  { p.gain(pts); }

private:
  const string name;
  const Points pts;
};
```

Note : l'affichage des « tuiles à propriétaire » commence à avoir un aspect systématique (quoique non souligné par la donnée). On pourrait donc imaginer factoriser cette partie commune (affichage du propriétaire) dans un `Property::display()` que l'on appelerait systèmatiquement dans les `display()` de ses sous-classes.

Question 1.2.7 – Les villes [sur 10 points]

Les villes sont des tuiles ayant un joueur propriétaire et dont le coût est 25.

Au niveau de l'affichage, les villes affichent simplement : « ville de » suivi de l'affichage de leur propriétaire (voir l'exemple de `main()` fourni au début).

L'autre seule spécificité des villes réside dans le calcul du `score()` : elles rapportent tout d'abord 5 points à leur propriétaire, puis, pour chaque tuile voisine, elles rapportent en plus (à leur propriétaire) le nombre de points retourné par la méthode `neighbor_points()` de la tuile voisine, appliquée au propriétaire de la tuile ville.
(Avec tout ce que nous avons défini plus haut, cela signifie qu'elles rapportent en plus 1 point pour chaque forêt adjacente qui appartient à leur propriétaire.)

Définissez ici une classe `City` pour représenter de telles tuiles.

Réponse :

```
class City : public Property
{
public:
  City(Player& p_) : Property(p_, 25) {}

  virtual void display(ostream& out) const override
  { out << "ville de " << p; }

  virtual void score() const override
  {
    p.gain(5);
    for (auto const& n : neighbors) {
      if (n != nullptr)
        p.gain(n->neighbor_points(p));
    }
  }
};
```

Commentaire : Attention à ne pas oublier de tester la non « nullité » d'un pointeur avant de l'utiliser ; p.ex. ici sur les voisins dans le calcul du `score()`.

Question 1.3 – Le plateau [sur 13 points]

Pour finir, on souhaite représenter le plateau[5], qui est simplement un ensemble de tuiles.

On doit pouvoir l'initialiser soit à vide, soit en passant un ensemble initial de tuiles (comme fait par exemple dans le `main()` fourni au début).

On devra aussi pouvoir (cf le `main()` fourni au début pour chacune de ces trois méthodes) :

- ajouter une tuile au plateau (`add()`) ;
- calculer le score du jeu (`score()`) en appelant simplement la méthode `score()` de chaque tuile sur le plateau ;
- pouvoir lier (`link()`) deux tuiles du plateau ; on donnera pour cela : l'index de la première tuile, l'index de la seconde tuile et le numéro de face de la première où la seconde va la toucher ;

  en cas d'erreur, vous afficherez un message d'erreur de votre choix.

Définissez ici une classe `Board` pour représenter le plateau de jeu.

Réponse :

```
class Board // : public Printable // inheritance can be forgotten here
{
public:
  Board(vector<Tile*> init) // or initilizer_list
    : board(init)
  {}
  Board() = default;

  void add(Tile* tile) { board.push_back(tile); }

  void score() const {
    for (auto tile : board) tile->score();
  }

  void link(size_t from, size_t to, size_t position)
  {
    try {
      board[from]->link(position, board[to]);
    } catch(int) {
      cerr << "ERREUR : impossible de mettre la tuile no" << to
           << " à coté de la tuile no" << from << " en position " << position
           << endl;
    }
  }

private:
  vector<Tile*> board;
};
```

Note : avant d'accéder à `from` et à `to`, il serait juticieux de vérifier ne pas déborder (mais on peut aussi imaginer que cette vérification a déjà été faite ailleurs).

Commentaire : Ne pas oublier de remettre le constructeur par défaut qui était explicitement demandé.

5. "*board*" in English.