TP C++

UML, STL, lambda, flux et template

Eric Ramat
eric.ramat@univ-littoral.fr

8 mars 2024

Durée : 9 heures

---

1 Objectifs

Le cinquième TP a pour objectif d'utiliser la STL (pointeurs intelligents, vector, array, ...) pour la traduction d'un diagramme de classes UML. Il est impératif de tenir compte de tous les éléments exprimés dans le diagramme : relation d'héritage, association, agrégation, composition, sens de navigation, ...

De plus, le code des méthodes doit être le plus court possible et pour cela, il faut utiliser les méthodes disponibles dans la bibliothèque *algorithm*, par exemple. Vous serez alors amener à écrire des lambdas.

2 Travail demandé

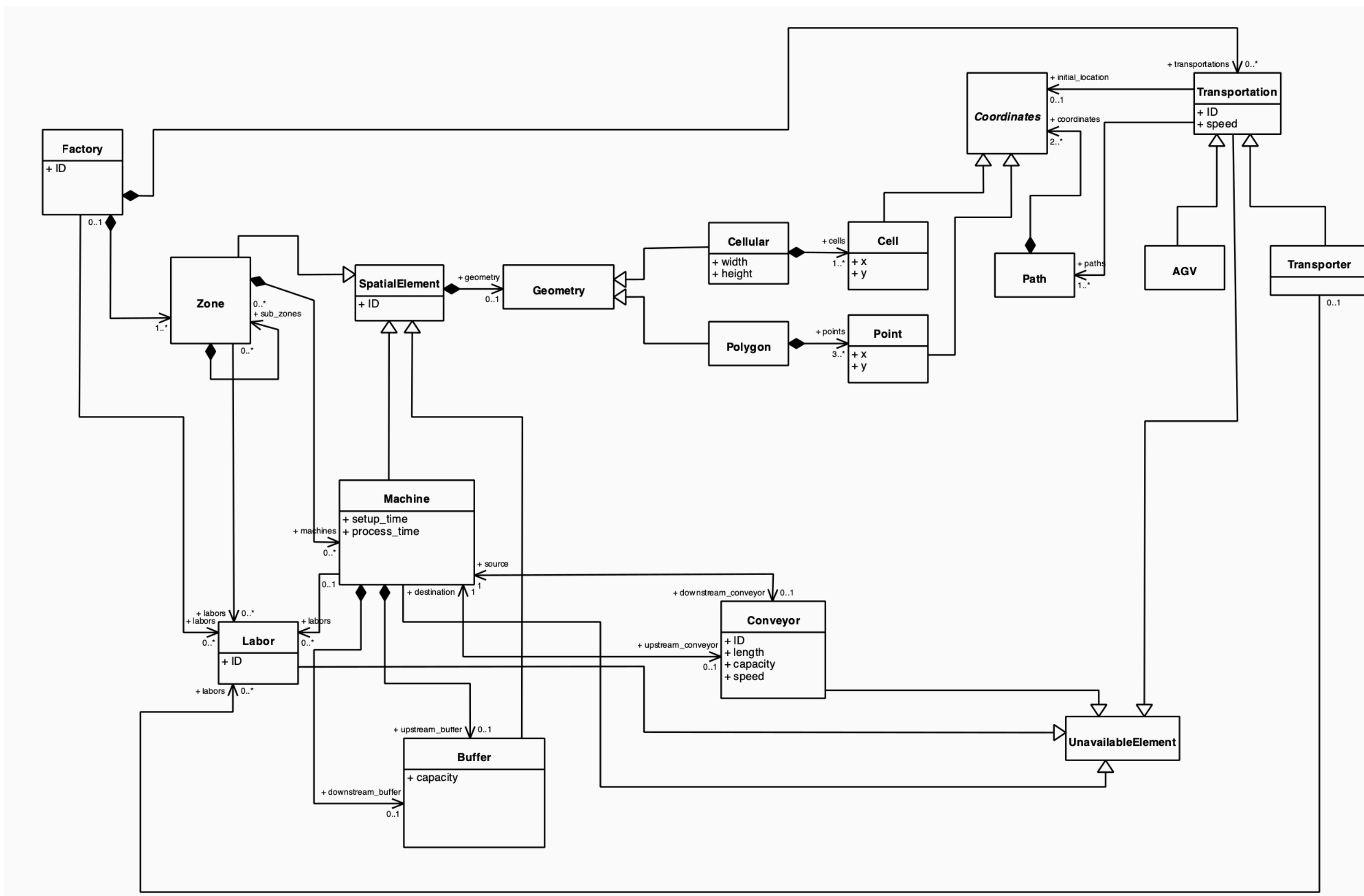
Avant de commencer le TP, il est nécessaire de forker un nouveau dépôt `git` contenant le projet avec un début de code très minimaliste. Vous devez voir apparaître dans votre compte gitlab (gitlab.dpt-info.univ-littoral.fr) le projet "factory-l3". Comme d'habitude, réalisez un fork dans votre compte et associez moi à votre dépôt privé avec les droits Reporter.

Le diagramme de classes à traduire modélise les éléments constituants des ateliers de fabrication dans une entreprise manufacturière. Chaque élément fera l'objet d'une petite définition afin que vous compreniez le concept représenté.

Pour chaque question, une ou plusieurs classes sont à développer. Si ces classes possèdent des relations (héritage, association, agrégation ou composition) avec des classes à développer dans la même question ou avec des classes déjà développées alors il faut aussi traduire les relations.

Comme pour les TP précédents, il ne faut pas utiliser les pointeurs mais uniquement les pointeurs intelligents, les références et les instances.

Il est fortement conseillé d'écrire le code de la question 8 en même temps que les questions 1 à 7. Vous pourrez vérifier que votre code est bien conforme aux attendus.



Question 1. Une entreprise manufacturière (*Factory*) se compose d'une ou plusieurs zones. Une zone peut se décomposer en sous-zones. Une sous-zone qui possède des sous-zones ne contient pas de machines. Une zone sans sous-zone représente un lieu où sont les machines de production (*Machine*). L'entreprise possède au moins une zone.

CLASSES A CODER : Factory, Zone et Machine

Question 2. Une zone peut être représentée sous forme d'un polygone ou d'un ensemble de celulles. Une zone est une sous-classe de *SpatialElement* et un élément spatial est défini par un ID et une géométrie (*Geometry*). La géométrie d'un élément spatial peut être de deux types : polygonal ou cellulaire.
Attention, il faut introduire le polymorphisme couplé à un pointeur unique.

CLASSES A CODER : SpatialElement, Geometry, Cellular et Polygon

Question 3. Les géométries cellulaires et polygonales possèdent des coordonnées (*Coordinates*) de deux types (*Cell* et *Point*). Les coordonnées x et y d'une cellule sont en réalité des indices (par exemple, la cellule $(3, 5)$). Pour les points, ce sont des réels dans un repère (x, y). Attention, le concept de coordonnées est abstrait.

CLASSES A CODER : Coordinates, Cell et Point

Question 4. Lorsqu'un produit est réalisé par une machine, il est stocké temporairement dans un stock (*Buffer*)

à côté de la machine. De même, lorsqu'un produit est acheminé vers une machine, ce dernier est stocké avant d'être sélectionné pour que la machine effectue une opération sur le produit. On parle donc de buffer d'entrée et buffer de sortie. Une machine peut disposer ou pas de l'un de ces buffers.

CLASSES A CODER : Buffer

Question 5. L'acheminement des produits d'une machine à une autre est réalisé à l'aide soit des convoyeurs (des sortes de tapis roulant) soit à l'aide d'engins de transport. Un convoyeur (*Conveyor*) relie deux machines. S'il y a un convoyeur entre 2 machines alors il n'y a pas de buffer de sortie pour la machine "source" et pas de buffer d'entrée pour la machine "destination". Attention à bien définir tous les constructeurs et les méthodes nécessaires à la définition. Les machines doivent connaître leur convoyeur et les convoyeurs doivent connaître les machines "source" et "destination".

Voici un exemple pour construire la relation bi-directionnelle :

```
const auto& M3 = factory.get_machine("M3");
const auto& M4 = factory.get_machine("M4");
auto C1 = std::make_shared<Conveyor>("C1", nullptr, 10, 100, 1, M3, M4);
factory.attach_downstream_conveyor_to_machine("M3", C1);
factory.attach_upstream_conveyor_to_machine("M4", C1);
```

CLASSES A CODER : Conveyor

Question 6. BONUS - Le second type d'acheminement sont les engins de transport (*Transportation*) : les véhicules autonomes autoguidés (*AGV*) et les chariots (*Transporter*). Les engins de transport suivent des chemins (*Path*) prédéfinis. Un engin peut emprunter plusieurs chemins selon la demande. Un chemin peut être composé d'un ensemble de points ou de cellules.

CLASSES A CODER : Transportation, AGV, Path et Transporter

Question 7. BONUS - Pour faire fonctionner l'ensemble du système, il faut des employés (*Labor*). Les employés sont tous rattachés à l'entreprise et à une ou plusieurs zones. Une partie d'entre eux travaille sur des machines, l'autre partie conduit des chariots. L'instance initiale d'employé est rattachée à une instance de Factory puis des méthodes permettent le rattachement "secondaire" aux zones, machines ou moyens de transport.

CLASSES A CODER : Labor

Question 8. Toutes les classes et toutes les relations sont maintenant définies. On va instantier les classes afin de représenter l'exemple suivant :

- l'entreprise "Inoxetum" fabrique des barbecues ;
- elle est divisée en plusieurs zones que l'on notera : A, B et C. La zone B se subdivise en deux zones : B1 et B2 ;
- les zones sont représentées par des polygones :
  - A : (0,0) → (100, 0) → (100, 200) → (0,200)
  - B1 : (100,200) → (400, 200) → (400, 600) → (100,600)
  - B2 : (100,600) → (400, 600) → (400, 1000) → (100,1000)
  - C : (400,200) → (1000, 200) → (1000, 600) → (400,600)
- la zone A dispose de 2 machines (M1 et M2) sans buffer d'entrée et de sortie ;

- la zone B1 dispose de 2 machines (M3 et M4) reliées par le convoyeur C1 ; C1 a une capacité de 10, une longueur 100m et une vitesse de 1 m/s ;
- la zone B2 dispose de 3 machines (M5, M6 et M7). M5 et M6 sont reliées par le convoyeur C2. M7 possède un buffer d'entrée B1 (d'une capacité de 10) et un buffer de sortie B2 (d'une capacité de 10) ; le convoyeur C2 a une capacité de 20, une longueur 50m et une vitesse de 0.5 m/s ;
- la zone C dispose d'une seule machine (M8) qui possède un buffer d'entrée B3 d'une capacité de 10 ;
- les temps de démarrage et d'opération sont respectivement :
  - M1 : 30s et 90s
  - M2 : 10s et 120s
  - M3 : 0s et 45s
  - M4 : 15s et 30s
  - M5 : 20s et 20s
  - M6 : 12s et 150s
  - M7 : 5s et 100s
  - M8 : 8s et 75s
- l'entreprise dispose de 10 travailleurs que l'on nommera de L1 à L10 ;
- L1 et L2 sont respectivement associés aux machines M1 et M2 ;
- L3 est conducteur de l'unique chariot (T1) de l'entreprise ; le chariot se déplace à la vitesse de 2 m/s ;
- de L4 à L9 sont associés aux machines M3 à M8 ;
- L10 est rattaché à M8 ; M8 est une emballeuse et a besoin de 2 travailleurs pour fonctionner ;
- le chariot T1 suivi l'un des trois chemins suivants :
  - (80,190) → (80, 220) → (350, 220) → (350, 580)
  - (320,190) → (150, 190) → (150, 975) → (375, 975) → (375, 840)
  - (375,720) → (530, 720) → (530, 510)
- il n'y a pas d'AGV ;
- aucune géométrie pour les machines et les buffers ne sont pas définies

Compléter la fonction main avec l'instantiation des classes conforme à la description ci-dessus.

Question 9. Implémenter les opérateurs de flux afin que les instructions ci-dessous soient possibles.

```
Factory factory("Inoxetum");

// définition des éléments de l'entreprise

std::cout << factory << std::endl;
```

On doit obtenir la sortie console suivante :

```
Factory < Inoxetum ; Zones { Zone < A ; Polygon < (0, 0) -> (100, 0) -> (100, 200) -> (0,
    200) > ; { Machine < M1 ; 30 ; 90 ; Labors { Labor < L1 > } > Machine < M2 ; 10 ;
    120 ; Labors { Labor < L2 > } > } ; Labors { } > Zone < B ; { Zone < B1 ; Polygon <
    (100, 200) -> (400, 200) -> (400, 600) -> (100, 600) > ; { Machine < M3 ; 0 ; 45 ;
    down: Conveyor < C1 ; 10 ; 100 ; 1 > ; Labors { Labor < L4 > } > Machine < M4 ; 15 ;
    30 ; up: Conveyor < C1 ; 10 ; 100 ; 1 > ; Labors { Labor < L5 > } > } ; Labors { } >
    Zone < B2 ; Polygon < (100, 600) -> (400, 600) -> (400, 1000) -> (100, 1000) > ; {
    Machine < M5 ; 20 ; 20 ; down: Conveyor < C2 ; 50 ; 20 ; 0.5 > ; Labors { Labor < L6
    > } > Machine < M6 ; 12 ; 150 ; up: Conveyor < C2 ; 50 ; 20 ; 0.5 > ; Labors { Labor
    < L7 > } > Machine < M7 ; 5 ; 100 ; up: Buffer < B1 ; 10 > ; down: Buffer < B2 ; 20 >
     ; Labors { Labor < L8 > } > } ; Labors { } > } ; Labors { } > Zone < C ; Polygon <
```

```
(400, 200) -> (1000, 200) -> (1000, 600) -> (400, 600) > ; { Machine < M8 ; 8 ; 75 ;
up: Buffer < B3 ; 10 > ; Labors { Labor < L9 > Labor < L10 > } > } ; Labors { } > } ;
 Transportations { Transporter < T1 ; 2 ; { Path < (80, 190) (80, 220) (350, 220)
(350, 580) > Path < (320, 190) (150, 190) (150, 975) (375, 975) (375, 840) > Path <
(375, 720) (530, 720) (530, 510) >  } > } ; Labors { Labor < L1 > Labor < L2 > Labor
< L3 > Labor < L4 > Labor < L5 > Labor < L6 > Labor < L7 > Labor < L8 > Labor < L9 >
Labor < L10 > } >
```

Question 10. Un employé (*Labor*) possède une compétence (*Skill*). Au lieu d'utiliser la notion de relation, on se propose de définir les compétences via la notion de template. La classe *Labor* devient un template dont le paramètre est un type (le type de la compétence). Le type est utilisé pour déclarer un attribut *skill*. La déclaration d'un employé sera du type :

```
Labor<Driver> L1;
Labor<Welder> L2;
Labor<Pipefitter> L3;
```

Les classes *Driver*, *Welder* et *Pipefitter* ne possèdent aucun attribut mais dispose d'une méthode *get_skill_type* qui retourne un string contenant le nom de la compétence (par exemple "driver" pour la classe *Driver*. Si on envoie un instance de *Labor* dans un flux de sortie, on affichera sa compétence ($Labor < L1, driver >$).

CLASSES A CODER : Driver, Welder, Pipefitter CLASSES A MODIFIER : Labor

Question 11. On aimerait se débarrasser de la classe *Geometry*. Pour cela, on va la supprimer et la remplacer par le paramétrage de la classe *SpatialElement* via un template. La déclaration de la classe *SpatialElement* devient :

```
SpatialElement<Polygon> e1("e1");
```

Les sous-classes de *SpatialElement* sont aussi impactées par ce changement. Une machine est maintenant déclarée de la manière suivante :

```
Machine<Polygon> m1("M1", std::make_unique_ptr<Polygon>({Point(0, 0), Point(50, 0),
 Point(50, 50), Point(0, 50)}), 30, 90);
```

En revanche, on aimerait pouvoir déclarer une machine (ou un buffer ou une zone) sans géométrie. Comment peut-on faire sans utiliser un pointeur nul ?

```
Machine m1("M1", 30, 90);
```

CLASSES A MODIFIER : SpatialElement, Machine, Buffer et Zone

Question 12. En utilisant l'instantiation obtenue dans la question 9, développer les méthodes suivantes dans la classe Factory :

— retourner une référence constante sur une machine à partir de son ID ;

— donner le nombre de machines d'une zone identifiée par son ID ;
— donner le nombre total de machines de l'entreprise ;
— retourner la surface totale de l'entreprise (somme des surfaces de chaque zone) ;
— calculer la longueur des différents chemins d'un chariot désigné par son ID ;

Il est impératif d'utiliser les algorithmes proposés par la STL et les fonctions lambda. De plus, il faut que le nombre de lignes des méthodes soit limité (une ligne serait idéal).