Compte-rendu TP4 CPP

Bastien Felix, Tommy Royer, Christine Ye Jin

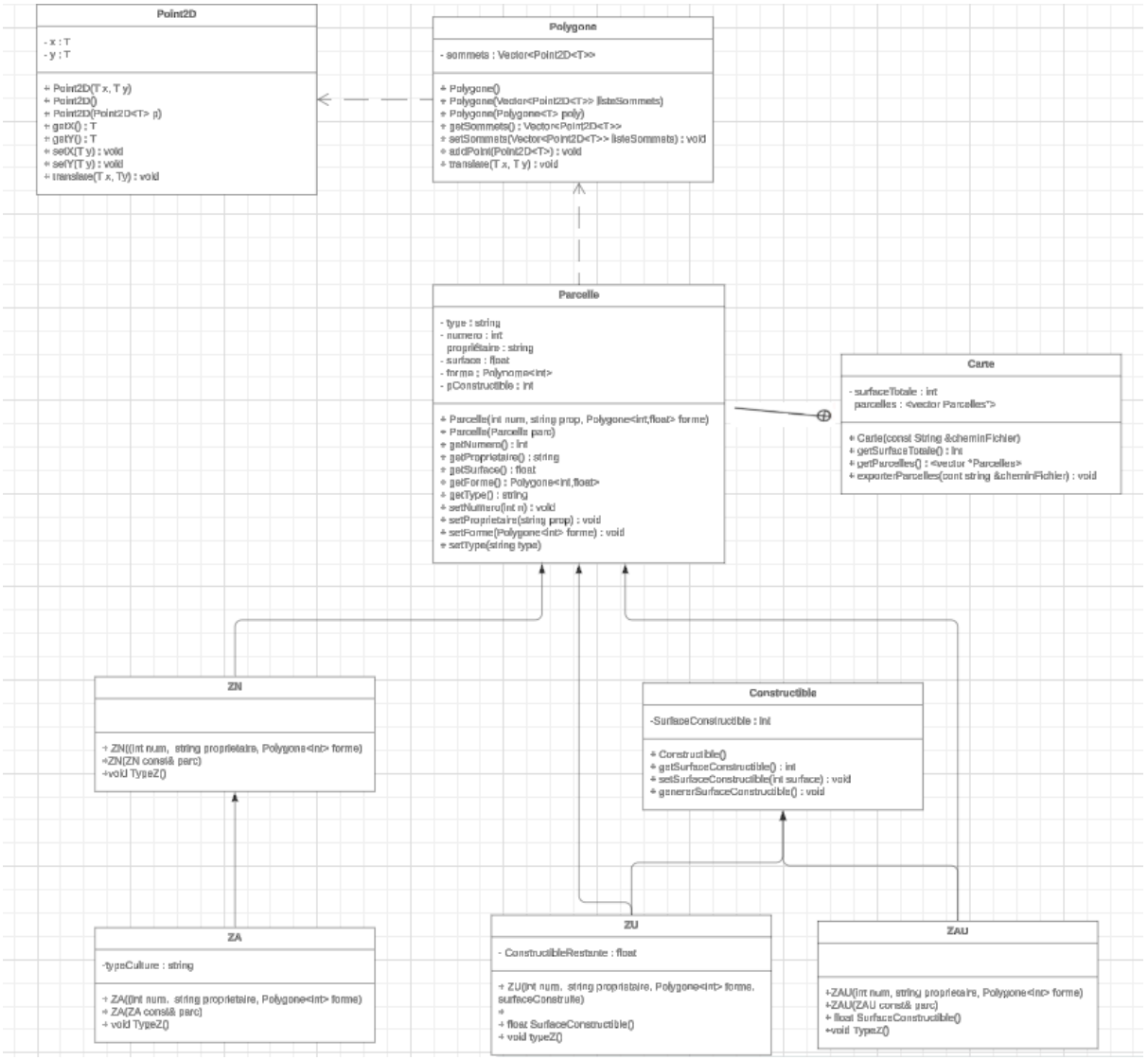
Lien Git :

https://github.com/ECN-SEC-SMP/tp4-note-tommy_bastienf_christine

Objectif du TP : Le but de ce TP est de créer un ensemble de classes en C++ permettant à une communauté de communes de gérer un PLU (Plan Local d'Urbanisme) simplifié et plus particulièrement lui permettant d'implanter un nouveau SI pour gérer son cadastre de manière adéquate en incluant les nouvelles directives du PLU.

I) Conception

Afin d'avoir une idée plus claire et globale dans notre projet, nous avons dessiné un diagramme des exigences. Parcelle qui est la classe principale ici, dérive d'un polygone qui elle-même dérive de Point2D. Les classes dérivées (ZN, ZA, ZU, ZAU) ajoutent des spécificités selon le type de parcelle. De plus, Parcelle compose la classe Carte car elle centralise la gestion et le stockage des parcelles.



L'image représente le diagramme de classes UML pour le TP sur la gestion simplifiée d'un PLU. Voici une explication des différents éléments et relations du diagramme :

1. **Classe Point2D** :
    - Représente un point dans un plan cartésien.
    - Contient les attributs x et y de type générique T.
    - Propose des méthodes pour obtenir (getX, getY), modifier (setX, setY), et translater (translate) les coordonnées d'un point.
2. **Classe Polygone** :
    - Contient une liste de sommets, chacun étant un Point2D.
    - Permet la création de polygones via plusieurs constructeurs.
    - Possède des méthodes pour obtenir les sommets (getSommets), modifier la liste des sommets, ajouter un point, ou translater le polygone.
3. **Classe Parcelle** (Abstraite) :
    - Représente une parcelle de terrain avec :
        - Un type (type).
        - Un numéro (numero).

        - Un propriétaire (proprietaire).
        - Une surface (surface) calculée à partir de la forme (un Polygone).
        - Un pourcentage de surface constructible (pConstructible).
    - Méthodes pour accéder et modifier les attributs (e.g., getSurface, setType).
4. **Sous-classes de Parcelle** :
    - **ZN** : Zones Naturelles, uniquement définies par leurs attributs généraux.
    - **ZA** : Zones Agricoles, avec un attribut supplémentaire (typeCulture) pour indiquer le type de culture cultivée.
    - **Constructible** (Abstraite) :
        - Interface pour les zones constructibles (ZU et ZAU).
        - Définit une méthode virtuelle pure surfaceConstructible pour calculer les surfaces constructibles.
    - **ZU** : Zones Urbaines, avec une surface construite et une méthode pour calculer la surface constructible restante.
    - **ZAU** : Zones à Urbaniser, avec une méthode calculant la surface constructible totale.
5. **Classe Carte** :
    - Représente l'ensemble des parcelles d'un PLU.
    - Attributs principaux :
        - surfaceTotale : surface totale des parcelles.
        - parcelles : un vecteur contenant les différentes parcelles.
    - Méthodes pour charger une carte depuis un fichier, obtenir des parcelles, ou exporter les données dans un fichier de sauvegarde.

Relations :

- **Héritage** :
    - Parcelle est la classe mère abstraite de ZN, ZA, ZU, et ZAU.
    - Constructible est une interface implémentée par ZU et ZAU.
- **Composition** :
    - Une Parcelle contient un Polygone pour représenter sa forme.
    - Un Polygone est composé de plusieurs Point2D.
    -

II) Définition des classes

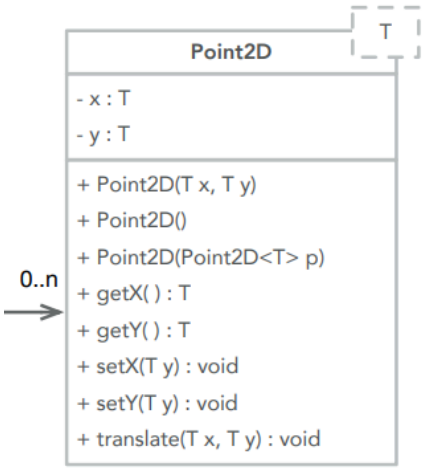
Dans cette première partie, nous créons les classes Point2D, Polygone et Parcelle dont la classe est donnée dans l'énoncé.

1) Classe Point2D

La classe Point2D est une classe template qui permet de caractériser un point dans un plan cartésien en 2D. Elle est composée de :

- Un type (chaîne de caractère) qui sera défini dans la partie suivante.
- Un numéro (entier)

- Un propriétaire (chaîne de caractères)
- Une surface en m2 (réel)
- Un pourcentage de surface constructible de la parcelle (entier)
- Une forme définie par un Polygone



Nous l'avons traduit ainsi en C++ :

```cpp
template <typename T>
class Point2D {
private:
    T x;
    T y;

public:
    // Constructeurs
    Point2D(T x, T y);
    Point2D();
    Point2D(const Point2D<T>& p);

    // Constructeur par conversion
    template <typename U>
    Point2D(const Point2D<U>& p);

    // Accesseurs
    T getX() const;
    T getY() const;

    // Mutateurs
    void setX(T x);
    void setY(T y);

    // Méthode de translation
    void translate(T dx, T dy);
```

```
    // Opérateur << en tant qu'ami
    template <typename U>
    friend ostream& operator<<(ostream& os, const Point2D<U>& point);
};
```

Nous avons également surcharger l'opérateur d'affichage afin d'afficher les coordonnées d'un point.

```
// Opérateur << (fonction externe)
template <typename T>
ostream& operator<<(ostream& os, const Point2D<T>& point) {
    os << "(" << point.getX() << ", " << point.getY() << ")";
    return os;
}
```

Dans le main, nous testons avec des jeux d'essais. Nous créons 3 points de coordonnées :

- p1 (0,0)
- p2 (1.5, 1.5)
- p3 (1.5, 1.5), qui est une copie du point p2

Nous procédons ensuite à une modification du point p1 en affectant à X, Y respectivement les valeurs 10 et 20. De plus, nous faisons translater les points p2 de (3.5, -1.5) et p3 de (-1, 2)

```
int main() {
    // Création de points avec différents types
    Point2D<int> p1; // Constructeur par défaut
    Point2D<float> p2(1.5f, 2.5f); // Constructeur avec paramètres
    Point2D<double> p3(p2); // Constructeur par copie

    // Affichage initial
    std::cout << "p1 (int): " << p1 << std::endl;
    std::cout << "p2 (float): " << p2 << std::endl;
    std::cout << "p3 (double, copie de p2): " << p3 << std::endl;

    // Modification des coordonnées
    p1.setX(10);
    p1.setY(20);
    std::cout << "Après modification, p1: " << p1 << std::endl;

    // Utilisation des accesseurs
    std::cout << "Coordonnées de p1: (" << p1.getX() << ", " << p1.getY() << ")"
<< std::endl;

    // Translation des points
    p2.translate(3.5f, -1.5f);
    std::cout << "Après translation, p2: " << p2 << std::endl;
```

```
    p3.translate(-1.0, 2.0);
    std::cout << "Après translation, p3: " << p3 << std::endl;

    return 0;
}
```

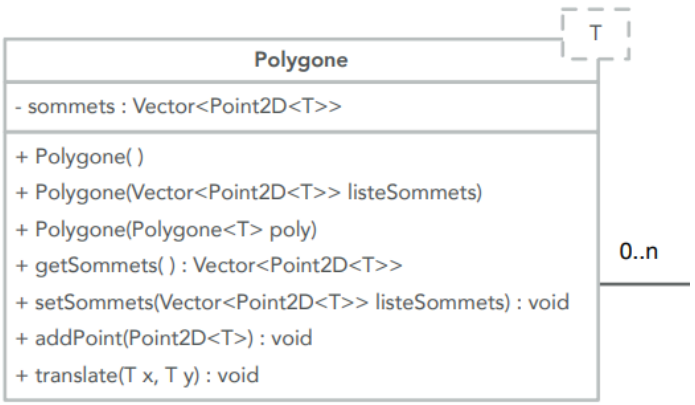
Résultat : les points s'affichent avec les bonnes valeurs.

```
@BastienFelix →/workspaces/CPP-TP4 (main) $ ./main
p1 (int): (0, 0)
p2 (float): (1.5, 2.5)
p3 (double, copie de p2): (1.5, 2.5)
Après modification, p1: (10, 20)
Coordonnées de p1: (10, 20)
Après translation, p2: (5, 1)
Après translation, p3: (0.5, 4.5)
```

2) Classe Polygone

Plusieurs points en 2D peuvent construire un polygone. La classe template polygone est caractérisée,en attribut privé, par un sommet.
En méthode, elle possède :

- Deux constructeurs
- Un accesseur : *getSommets( ) : Vector<Point2D<T>>*
- Un mutateur : *setSommets(Vector<Point2D<T>> listeSommets) : void*
- Une fonction qui ajoute un point au polygone : *addPoint(Point2D<T>) : void*
- Une fonction permettant de translater un point d'une valeur x et y sur un plan cartésien : *translate(T x, T y) : void*



Traduction en C++ :

```cpp
template <typename T>
class Polygone {

private:
    vector<Point2D<T>> sommets;

public:
    Polygone();
    Polygone(const vector<Point2D<T>>& listeSommets);
    Polygone(const Polygone<T>& poly);
    vector<Point2D<T>> getSommets() const;
    void setSommets(const vector<Point2D<T>>& listeSommets);
    void addPoint(const Point2D<T>& point);
    void translate(T x, T y);
    template <typename U>
    friend ostream& operator<<(ostream& os, const Polygone<U>& poly);
};
```

Afin de tester le bon fonctionnement de notre code, nous créons un triangle composé des points p1, p2 et p3 avec comme coordonnées (1, 1), (2, 3) et (4, 5) respectivement.

```cpp
int main() {
    // Création de points
    Point2D<int> p1(1, 1);
    Point2D<int> p2(2, 3);
    Point2D<int> p3(4, 5);

    // Création d'un polygone avec ces points
    vector<Point2D<int>> points = { p1, p2, p3 };
    Polygone<int> poly(points);

    // Affichage du polygone initial
    cout << "Polygone initial: " << poly << endl;

    // Ajout d'un point
    Point2D<int> p4(6, 7);
    poly.addPoint(p4);

    // Affichage après ajout du point
    cout << "Polygone après ajout du point: " << poly << endl;

    // Translation du polygone
    poly.translate(1, -1);

    // Affichage après translation
    cout << "Polygone après translation: " << poly << endl;

    return 0;
```

```
}
```

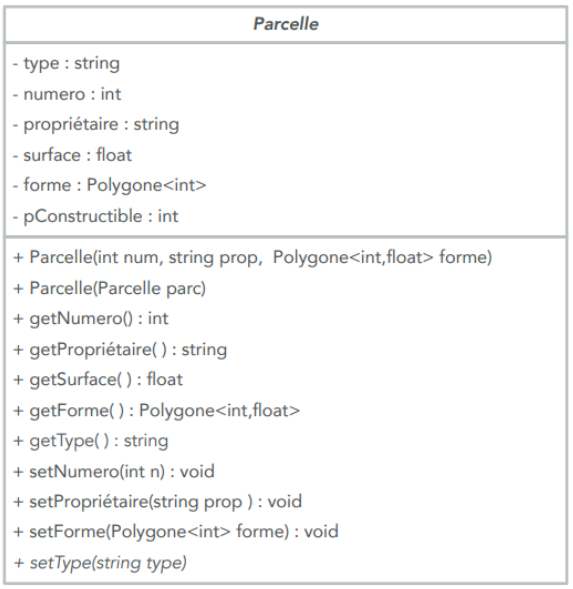
La fonction addPoint permet d'ajouter un point au polygone. Ici, p4 s'ajoute pour former un quadrilatère. Ainsi, à l'affichage nous avons les coordonnées des 4 sommets.

```
@BastienFelix →/workspaces/CPP-TP4 (main) $ ./main
Polygone initial: Polygone: (1, 1) (2, 3) (4, 5)
Polygone après ajout du point: Polygone: (1, 1) (2, 3) (4, 5) (6, 7)
Polygone après translation: Polygone: (2, 0) (3, 2) (5, 4) (7, 6)
```

3) Classe Parcelle

Une Parcelle est caractérisée par :

- Son type (ZU, ZAU, ZA, ZN)
- Un numéro
- Un propriétaire
- Une surface
- Sa forme



La classe a comme méthode des constructeurs pour construire une parcelle, des accesseurs et mutateurs pour définir et accéder au nom du propriétaire, sa surface, sa forme, son numéro et son type.

```cpp
int main() {
    // Création de points pour définir un polygone
    Point2D<int> p1(0, 0);
    Point2D<int> p2(4, 0);
    Point2D<int> p3(4, 3);
    Point2D<int> p4(0, 3);
```

```
    // Création du polygone avec ces points
    vector<Point2D<int>> points = { p1, p2, p3, p4 };
    Polygone<int> poly(points);

    // Création d'une parcelle avec un numéro, un propriétaire et la forme
(polygone)
    Parcelle<int> parcelle(1, "John Doe", poly);

    // Affichage des informations de la parcelle
    cout << parcelle << endl;

    // Test de modification de la surface (changement de forme)
    Point2D<int> p5(1, 1);
    poly.addPoint(p5);
    parcelle.setForme(poly); // La surface sera recalculée après le changement de
forme

    // Affichage après modification de la forme et recalcul de la surface
    cout << "Après ajout d'un point:" << endl;
    cout << parcelle << endl;

    // Test d'exception avec un polygone à moins de 3 sommets
    try {
        vector<Point2D<int>> points_invalides = { p1, p2 };
        Polygone<int> poly_invalid(points_invalides);
        Parcelle<int> parcelle_invalid(2, "Jane Doe", poly_invalid);
    } catch (const invalid_argument& e) {
        cout << "Erreur: " << e.what() << endl;
    }

    // Test d'exception avec une surface négative
    try {
        vector<Point2D<int>> points_invalid_surface = { p1, p2, p3 };
        Polygone<int> poly_invalid_surface(points_invalid_surface);
        Parcelle<int> parcelle_invalid_surface(3, "Alice", poly_invalid_surface);
        parcelle_invalid_surface.setType("Invalid");
        cout << parcelle_invalid_surface << endl;
    } catch (const invalid_argument& e) {
        cout << "Erreur: " << e.what() << endl;
    }

    return 0;
}
```

Des exceptions permettent de tester si un polygone a moins de 3 sommets. Dans le cas où elle possède 1 ou 2 sommets, le programme renvoie une erreur avec sa description. De même, si une parcelle possède une surface négative, le programme retourne une erreur.

Voici le résultat de l'exécution du bout de code précédent dans le cas où la parcelle de Jane Doe possède 2 sommets, et Alice est propriétaire d'une par :

```
Parcelle 1 (John Doe) : Surface = 12 m², Type =
Après ajout d'un point:
Parcelle 1 (John Doe) : Surface = 10.5 m², Type =
Erreur: Un polygone doit avoir au moins 3 sommets pour calculer une surface.
Parcelle 3 (Alice) : Surface = 6 m², Type = Invalid
@BastienFelix →/workspaces/CPP-TP4 (main) $
```

III) Définition des classes Zones

Le PLU prévoit de découper les parcelles de son cadastre selon 4 zones : zones urbaines, zones à urbaniser, zones agricoles et zones naturelles et forestières.

1) Zones à Urbaniser

La classe des zones à urbaniser (ZAU) hérite de Parcelle et de Constructible. Et comme les zones sont constructibles, on implémente une classe abstraite constructible avec une fonction virtuelle pure.
Le constructeur initialise la parcelle avec le type spécifique "Zone Agricole Urbaine" et calcule sa surface constructible. L'accesseur setType redéfinit son type et enfin, on surcharge l'opérateur << pour afficher les informations d'une parcelle ZAU.

```hpp
#ifndef ZAU_HPP
#define ZAU_HPP

#include "Constructible.hpp"
#include "Parcelle.hpp"

using namespace std;

class ZAU : public Constructible, public Parcelle {
  public:
    ZAU(int num, const string& prop, Polygone<int> polygone);
    void setType(const string& type) override;
    friend ostream& operator<<(ostream& os, const ZAU& zau);
};

ZAU::ZAU(int num, const string& prop, Polygone<int> polygone) : Parcelle(num,
prop, polygone) {
    setType("Zone Agricole Urbaine");
    Constructible::genererSurfaceConstructible();
}
```

```cpp
void ZAU::setType(const string& type) {
    Parcelle::setType(type);
}

ostream& operator<<(ostream& os, const ZAU& zau) {
    os << "Parcelle n°" << zau.getNumero() << " :" << endl;
    os << "     Type : " << zau.getType() << endl;
    os << "     Propriétaire : " << zau.getProprietaire() << endl;
    os << "     Surface : " << zau.getSurface() << " m²" << endl;
    os << "     Surface constructible : " << zau.getSurfaceConstructible() << "%"
<< endl;
    os << "     " << zau.getForme() << endl;

    return os;
}

#endif
```

Nous avons créé une parcelle ZAU en définissant d'abord un polygone à partir de quatre sommets. Une instance de ZAU est ensuite initialisée avec un numéro, un propriétaire et ce polygone comme forme. Enfin, les informations détaillées de la parcelle sont affichées à l'écran grâce à la surcharge de l'opérateur <<.

```cpp
int main() {
    // Création de points pour définir un polygone
    Point2D<int> p1(0, 0);
    Point2D<int> p2(4, 0);
    Point2D<int> p3(4, 3);
    Point2D<int> p4(0, 3);

    // Création du polygone avec ces points
    vector<Point2D<int>> points = { p1, p2, p3, p4 };
    Polygone<int> poly(points);

    // Création d'une instance de ZAU
    ZAU zau(1, "Tommy Royer", poly);

    // Affichage des informations de ZAU
    cout << zau << endl;

    return 0;
}
```

Résultat :

La parcelle de Tommy Royer est de type Zone Agricole Urbaine, sa surface est de 12m² avec 87% de surface constructible.

```
@BastienFelix →/workspaces/CPP-TP4 (main) $ ./main
Parcelle n°1 :
      Type : Zone Agricole Urbaine
      Propriétaire : Tommy Royer
      Surface : 12 m²
      Surface constructible : 87%
      Polygone: (0, 0) (4, 0) (4, 3) (0, 3)
```

2) Zones Naturelles et Forestières

Les Zones Naturelles et Forestières (ZN) héritent également de Parcelle. Cependant comme elles ne sont pas constructibles, nous n'avons pas accès à leurs surfaces.
Dans le code ci-dessous, une zone a été créée à partir de 4 sommets. Elle appartient au domaine public.

```cpp
int main() {
    // Création de points pour définir un polygone pour ZN
    Point2D<int> p5(1, 1);
    Point2D<int> p6(5, 1);
    Point2D<int> p7(5, 4);
    Point2D<int> p8(1, 5);

    // Création du polygone avec ces points
    vector<Point2D<int>> pointsZN = { p5, p6, p7, p8 };
    Polygone<int> polyZN(pointsZN);

    // Création d'une instance de ZN
    ZN zn(2, "Publique", polyZN);

    // Affichage des informations de ZN
    cout << zn << endl;

    return 0;
}
```

Résultat :

La surface calculée est de 14m² et nous avons bien 0% de surface constructible à l'exécution du code.

```
@BastienFelix →/workspaces/CPP-TP4 (main) $ ./main
Parcelle n°2 :
      Type : Zone Naturelle et forestière
      Propriétaire : Publique
      Surface : 14 m²
      Surface constructible : 0%
      Polygone: (1, 1) (5, 1) (5, 4) (1, 5)
```

3) Zones Urbaines

4 sommets définissent la forme d'une parcelle en Zone Urbaine. De même que les autres classes, une instance de la classe ZU est créée avec un numéro, un propriétaire, et la forme (polygone). Nous affichons toutes les informations toujours en surchargeant l'opérateur <<. La classe ZU inclut également des calculs pour gérer la surface déjà construite et la surface restante constructible.

```cpp
int main()
{
    // Création de points pour définir un polygone pour ZU
    Point2D<int> p1(0, 0);
    Point2D<int> p2(7, 0);
    Point2D<int> p3(10, 5);
    Point2D<int> p4(0, 9);

    // Création du polygone avec ces points
    vector<Point2D<int>> pointsZU = {p1, p2, p3, p4};
    Polygone<int> polyZU(pointsZU);

    // Création d'une instance de ZU
    ZU zu(1, "Christine Yejin", polyZU);

    // Affichage des informations de ZU
    cout << zu << endl;

    return 0;
}
```

Résultat :

Pour un polygone de coordonnées (0,0), (7,0), (10, 5) et (0,9), la surface de la parcelle en zone urbaine de Christine est de 62,5 m² avec 8% de surface constructible.

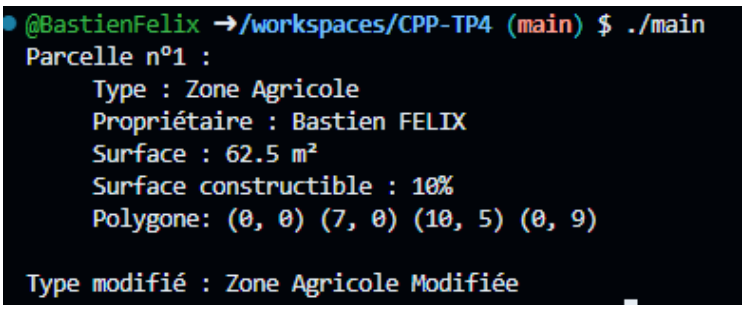
```
@BastienFelix →/workspaces/CPP-TP4 (main) $ ./main
Parcelle n°1 :
     Type : Zone Urbaine
     Propriétaire : Christine Yejin
     Surface : 62.5 m²
     Surface constructible restante : 8%
     Polygone: (0, 0) (7, 0) (10, 5) (0, 9)
```

4) Zones Agricoles

Les zones agricoles, comme l'indique l'énoncé, sont des zones naturelles et forestières mais qui ont un type de culture cultivée en plus dans la classe. La surface construite ne doit pas dépasser 10% de la surface de la parcelle et doit être inférieure ou égale à 200m².

```cpp
int main() {
    // Création de points pour définir un polygone
    Point2D<int> p1(0, 0);
    Point2D<int> p2(7, 0);
    Point2D<int> p3(10, 5);
    Point2D<int> p4(0, 9);

    // Création du polygone avec ces points
    vector<Point2D<int>> points = { p1, p2, p3, p4 };
    Polygone<int> poly(points);

    // Création d'une instance de ZA
    ZA za(1, "Bastien FELIX", poly, "Blé");

    // Affichage des informations de ZA
    cout << za << endl;

    // Test de modification du type
    za.setType("Zone Agricole Modifiée");
    cout << "Type modifié : " << za.getType() << endl;

    return 0;
}
```

Résultat :



Pour permettre la sauvegarde d'une **Carte** dans un fichier de sauvegarde, nous avons ajouté une méthode dédiée qui enregistre les détails de toutes les parcelles dans un fichier. Cela peut être accompli en adaptant la méthode exporterParcelles pour qu'elle sauvegarde dans un format standardisé.

La méthode sauvegarder enrichit la classe Carte en permettant une gestion complète des données, facilitant leur exportation et leur réutilisation. La classe Carte peut ainsi évoluer pour inclure d'autres détails comme des métadonnées sur les parcelles ou des statistiques supplémentaires.

Objectif de la méthode exporterParcelles

Cette méthode sauvegarde :
Les informations des parcelles (type, numéro, propriétaire).
Les coordonnées des sommets de chaque parcelle sous forme de points.

Les données sont écrites dans un fichier texte de manière structurée.

Début de la méthode

```
ofstream fichier(cheminFichier);
```

- **Action** : Ouvre un fichier en mode écriture à l'emplacement donné par cheminFichier.
- Si le fichier n'existe pas, il est créé. Si le fichier existe déjà, il sera écrasé.

```
if (!fichier.is_open()) {
        throw runtime_error("Impossible d'ouvrir le fichier " +
cheminFichier);
    }
```

- **Vérification** : Si le fichier n'a pas pu être ouvert pour une raison quelconque (droits d'accès, emplacement inexistant, etc.), une exception est levée pour signaler l'erreur.

Parcours des parcelles et écriture des données

```
 for (const auto &parcelle : parcelles) {
        fichier << parcelle->getType() << " " << parcelle->getNumero()
<< " " << parcelle->getProprietaire() << endl;
```

- **Action** : Parcourt chaque parcelle dans le vecteur parcelles.
    - Utilise les méthodes getType(), getNumero(), et getProprietaire() pour obtenir les informations de base sur la parcelle (type, numéro, propriétaire).

Ces informations sont écrites dans le fichier sous la forme :
[Type] [Numéro] [Propriétaire]

```
 const auto &points = parcelle->getForme().getSommets();
        for (const auto &point : points) {
            fichier << "[" << point.getX() << ";" << point.getY() << "]
";
        }
        fichier << endl;
```

- **Action** : Récupère les sommets de la parcelle sous forme d'un vecteur de points via getForme().getSommets().

Parcourt les sommets de chaque parcelle et les écrit dans le fichier sous la forme :
[x;y] [x;y] [x;y]

    - Chaque point représente un sommet du polygone décrivant la forme de la parcelle.

Fermeture du fichier

```
    fichier.close();
```

- **Action** : Ferme le fichier pour s'assurer que toutes les données sont correctement sauvegardées.

Voici les fichiers que nous obtenons :

Parcelles_short.txt

```
ZU 17 Martin 55 1003
[0;30] [60;100] [0;100]
ZAU 25 Robert 16
[0;30] [80;30] [80;100] [60;100]
ZA 42 Babin blé
[0;0] [80;0] [80;30] [0;30]
ZN 54 Savard
[80;0] [100;0] [100;100] [80;100]
```

Parcelles_export.txt

```
Zone Agripcole 24 SAMSON
[-47;-158] [-72;-239] [0;-275] [25;-208]
Zone Agripcole 101 ROUGE
[0;125] [-38;146] [-60;115] [0;50]
Zone Agripcole 111 REVIEUX
[-135;75] [-113;50] [-69;115]
Zone Agripcole 121 AMPLOI
[128;100] [146;171] [102;194] [55;100] [111;50]
Zone Agripcole 134 MANAC'H
[150;-100] [195;-60] [150;-5] [104;-5] [104;-100]
Zone Agripcole 152 AMPLOI
[96;5] [111;50] [55;100] [30;50] [50;20] [50;5]
Zone Agripcole 153 LEPOT
[50;-30] [0;-47] [-10;-47] [96;-94] [96;-5] [50;-5]
Zone Agripcole 187 AMPLOI
[210;147] [250;250] [200;257] [152;178]
Zone Agripcole 211 REVIEUX
[-47;300] [3;203] [50;203] [50;282]
Zone Agripcole 239 LEPOT
[-44;-147] [22;-194] [56;-150]
Zone Agripcole 261 LEPOT
[-150;-13] [-113;50] [-135;75] [-229;23]
Zone Agripcole 277 DUPUIS
[-109;198] [-41;158] [0;196]
Zone Agripcole 287 MANAC'H
[-121;203] [-6;203] [-53;300] [-100;287]
```

Conclusion :

Ce TP nous a permis de concevoir et de développer un système complet pour gérer un PLU simplifié, en créant une carte contenant diverses parcelles avec leurs caractéristiques propres.
Nous avons d'abord modélisé les points et polygones, pour représenter les formes géométriques des parcelles, avant de définir une hiérarchie de classes pour gérer les différents types de zones (ZU, ZAU, ZA, ZN).
La classe Carte a centralisé la gestion des parcelles, avec des fonctionnalités d'import/export pour la persistance des données.
Ce TP a renforcé nos compétences en POO, gestion des exceptions, structuration de projet avec Git, tests unitaires (Google Test) et manipulation de fichiers.