Compte Rendu – TP4

Saouti Rayan El Khayder Zakaria Montino Omar Lebihan Nicolas

December 1, 2024

Contents

1 Introduction

Le but de ce TP est de créer un ensemble de classes permettant de gérer un PLU simplifié. Ce document détaille les étapes de conception ainsi que les résultats obtenus.

2 Création du Repository

2.1 Instructions de départ

Les groupes sont constitués et un fichier `README.md` a été créé avec les noms des membres. Le projet est hébergé sur `GitHub`. Voici le lien du repo : `https://github.com/ECN-SEC-SMP/tp4-note-rayan-nicolas-omar-zakaria.git`

3 Diagramme UML

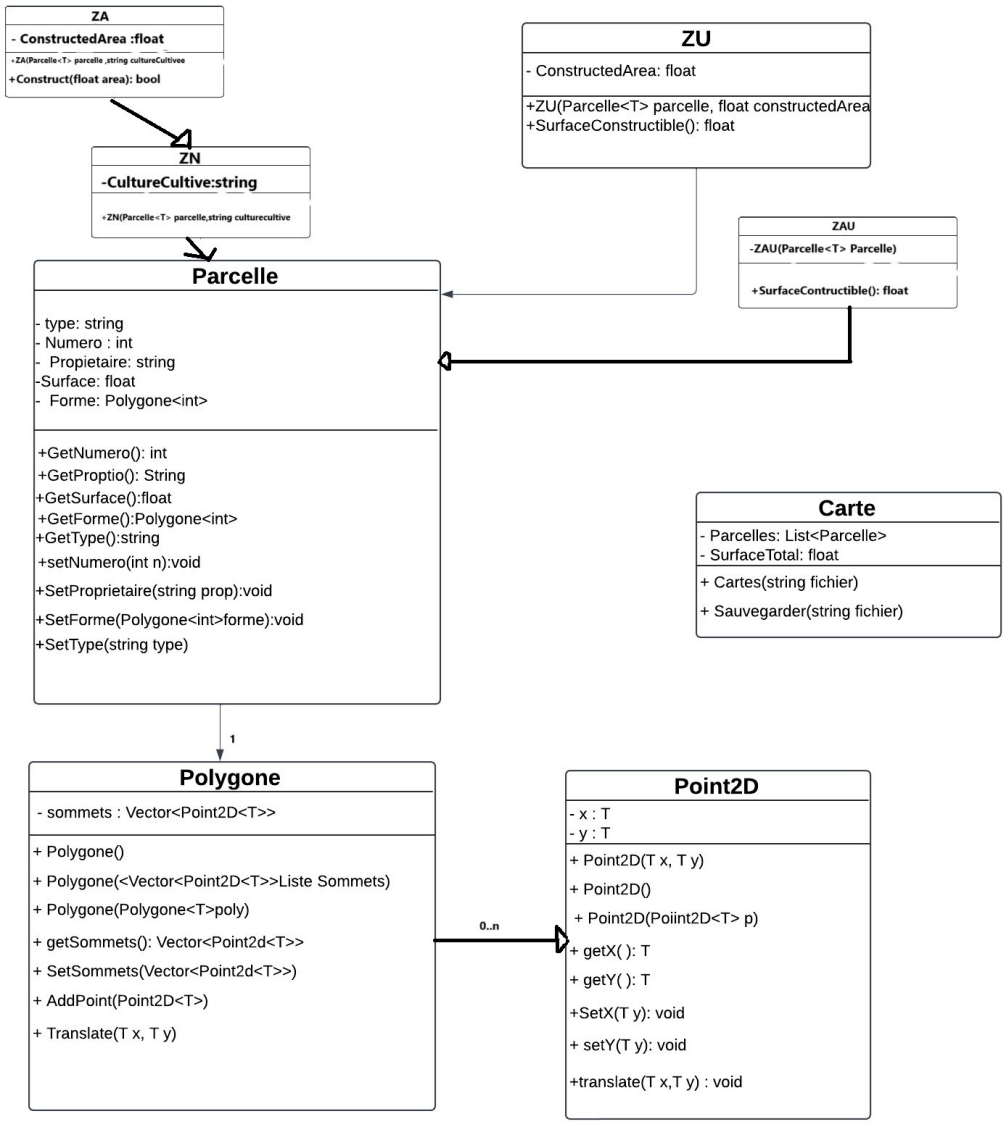

Figure 1: Diagramme UML

4 Classe Point2D

4.1 Code et Explications

Listing 1: Classe Point2D

```
#pragma once
#include <iostream>

template <typename T>
class Point2D
{
public:
    Point2D(T x, T y) : _x(x), _y(y) {}
    Point2D() : _x(0), _y(0) {}
    Point2D(const Point2D<T> &original) : _x(original._x), _y(original.
       _y) {}

    T getX() const { return _x; }
    T getY() const { return _y; }

    void setX(T x) { _x = x; }
    void setY(T y) { _y = y; }
    void translate(T x, T y)
    {
        _x += x;
        _y += y;
    }
    friend std::ostream &operator<<(std::ostream &os, const Point2D &p)
    {
        os << '[' << p._x << ";" << p._y << ']';
        return os;
    }

private:
    T _x, _y;
};
```

4.2 Explication

La classe Point2D représente un point dans un plan 2D avec des coordonnées T . Elle permet de manipuler les coordonnées et de les afficher.

4.3 Tests

Les tests ont été réalisés pour vérifier les opérations de translation et l'affichage.

5 Classe Polygone

Listing 2: Classe Polygone

```
#pragma once
#include "point2d.hpp"
#include <vector>

template <typename T>
class Polygon
{
public:
    Polygon() {}
    Polygon(const Polygon<T> &other) : _summits(other._summits), _area(
       other._area) {}
    Polygon(const std::vector<Point2D<T>> &summits) : _summits(summits)
    {
        _recalculateArea();
    }

    const std::vector<Point2D<T>> &getSummits() { return _summits; };

    void setSummits(const std::vector<Point2D<T>> &summits)
    {
        _summits = summits;
        _recalculateArea();
    }
    void addSummit(const Point2D<T> &p)
    {
        _summits.push_back(p);
        _recalculateArea();
    }
    void translate(T x, T y)
    {
        for (auto &p : _summits)
            p.translate(x, y);
    }

    double getArea() const { return _area; }

    friend std::ostream &operator<<(std::ostream &os, const Polygon &p)
    {
        for (size_t i = 0; i < p._summits.size(); ++i)
        {
            os << p._summits[i];
            if (i != p._summits.size() - 1)
                os << ' ';
        }
        return os;
    }
private:
    void _recalculateArea()
    {
        double s = 0;
```

```
        for (size_t i = 0; i < _summits.size() - 1; ++i)
        {
            T x1 = _summits[i].getX();
            T y1 = _summits[i].getY();
            T x2 = _summits[i + 1].getX();
            T y2 = _summits[i + 1].getY();
            s += x1 * y2 - x2 * y1;
        }
        _area = std::abs(s) / 2.0;
    }
private:
    std::vector<Point2D<T>> _summits;
    double _area = 0;
};
```

5.1 Explication

La classe peut manipuler des sommets définis par des coordonnées génériques T (exemple : int, float, etc.). Elle gère aussi la translation des sommets, le calcul de l'aire, ou l'ajout de nouveaux sommets.

6 Classe Parcelle

6.1 Caractéristiques

Description des attributs et des exceptions levées pour des surfaces invalides ou des polygones croisés.

6.2 Code

Listing 3: Classe Parcelle

```
#pragma once
#include <iostream>
#include <string>
#include "polygon.hpp"

template <typename T>
class Parcelle
{
public:
    Parcelle(int number, std::string owner, Polygon<T> shape) : number_
       (number),
                                                                owner_(
                                                                   owner
                                                                   ),
                                                                shape_(
                                                                   shape
                                                                   )

    {
    }
```

```
    Parcelle(const Parcelle<T> &other) : number_(other.number_),
                                         constructableAreaPercentage_(
                                            other.
                                            constructableAreaPercentage_
                                            ),
                                         type_(other.type_),
                                         owner_(other.owner_),
                                         shape_(other.shape_) {}

    int getNumber() const { return number_; }
    void setNumber(int number) { number_ = number; }

    int getConstructableAreaPercentage() const { return
       constructableAreaPercentage_; }
    void setConstructableAreaPercentage(int value) {
       constructableAreaPercentage_ = value; }

    std::string getOwner() const { return owner_; }
    void setOwner(const std::string &owner) { owner_ = owner; }

    Polygon<T> getShape() const { return shape_; }
    void setShape(Polygon<T> shape) { shape_ = shape; }

    std::string getType() const { return type_; }
    void setType(const std::string &type) { type_ = type; }

    float getArea() const { return shape_.getArea(); }
    friend std::ostream &operator<<(std::ostream &os, const Parcelle &p
       )
    {
        os << "Parcelle n  " << p.number_ << std::endl
           << " Type : " << p.type_ << std::endl
           << " Polygone : " << p.shape_ << std::endl
           << " Propri taire : " << p.owner_ << std::endl
           << " Surface : " << p.getArea() << std::endl;
        return os;
    }
protected:
    int number_,
        constructableAreaPercentage_ = 0;
    std::string type_, owner_;
    Polygon<T> shape_;
};
```

6.3 Explication

La classe Parcelle représente une parcelle de terrain avec des informations comme son numéro, son propriétaire, son type, et sa forme (décrite par un polygone). Elle permet de manipuler ces attributs et afficher les détails de la parcelle. Les types de parcelles sont détaillés en commentaire dans la classe carte.

7 Classe Carte

Listing 4: Classe Carte

```
#pragma once
#include <vector>
#include <fstream>
#include <sstream>
#include <string>
#include <exception>
#include <regex>
#include "constructible.hpp"

template <typename T>
class Carte
{
public:
    Carte(std::vector<std::shared_ptr<Parcelle<T>>> parcelles) :
       _parcelles(parcelles) {}

    static Carte FromFile(const std::string &filename)
    {
        std::fstream file;
        file.open(filename);

        if (!file.is_open())
            throw std::invalid_argument("Error opening file");

        std::vector<std::shared_ptr<Parcelle<T>>> parcelles;
        std::string line;
        while (std::getline(file, line))
        {
            std::istringstream lineStream(line);

            // Parse the first line
            std::string type, num, owner;
            lineStream >> type >> num >> owner;

            // Parse the second line (list of points)
            std::getline(file, line);
            Polygon shape = Polygon(Carte::ParsePoint(line));

            Parcelle p(std::stoi(num), owner, shape);

            //     Une Parcelle est d finie par deux lignes :
            //     La premi re contient au moins : typeParcelle
               num ro propri taire
            //     La deuxi me contient la liste des points
               d finissant la forme de la Parcelle
            //     Pour une ZU : typeParcelle num ro propri taire
               pConstructible surfaceConstruite
            //     Pour une ZAU : typeParcelle num ro propri taire
               pConstructible
```

```
            //     Pour une ZA : typeParcelle num ro propri taire
               typeCulture
            //     Pour une ZN : typeParcelle num ro propri taire

            std::shared_ptr<Parcelle<T>> zone;
            if (type == "ZU")
            {
                std::string pConstructible, surfaceConstruite;
                lineStream >> pConstructible >> surfaceConstruite;
                p.setConstructableAreaPercentage(std::stoi(
                   pConstructible));
                zone = std::make_shared<ZU<T>>(p, std::stoi(
                   surfaceConstruite));
            }
            else if (type == "ZAU")
            {
                std::string pConstructible;
                lineStream >> pConstructible;
                p.setConstructableAreaPercentage(std::stoi(
                   pConstructible));
                zone = std::make_shared<ZAU<T>>(p);
            }
            else if (type == "ZA")
            {
                std::string typeCulture;
                lineStream >> typeCulture;
                zone = std::make_shared<ZA<T>>(p, typeCulture);
            }
            else if (type == "ZN")
            {
                zone = std::make_shared<ZN<T>>(p);
            }
            parcelles.push_back(zone);
        }
        file.close();
        return Carte(parcelles);
    }
    // template <typename T>
    static std::vector<Point2D<T>> ParsePoint(const std::string &line)
    {
        std::vector<Point2D<T>> points;
        std::regex pointRegex(R"(\[(-?\d+);(-?\d+)\])");
        std::string::const_iterator searchStart(line.cbegin());
        std::smatch match;

        while (std::regex_search(searchStart, line.cend(), match,
           pointRegex))
        {
            int x = std::stoi(match[1]);
            int y = std::stoi(match[2]);
            // std::cout << match[1] << ' ' << x << ' ' << match[2] <<
               ' ' << y << std::endl;
```

```
            points.emplace_back(x, y);
            searchStart = match.suffix().first;
        }
        return points;
    }
    void ToFile(const std::string &filename) const
    {
        std::ofstream file;
        file.open(filename);

        if (!file.is_open())
            throw std::invalid_argument("Error opening file");

        //              Une Parcelle est definie par deux lignes :
        //      La premiere contient au moins : typeParcelle numero
           proprietaire
        //      La deuxieme contient la liste des points definissant la
           forme de la Parcelle
        //      Pour une ZU : typeParcelle numero proprietaire
           pConstructible surfaceConstruite
        //      Pour une ZAU : typeParcelle numero proprietaire
           pConstructible
        //      Pour une ZA : typeParcelle numero proprietaire
           typeCulture
        //      Pour une ZN : typeParcelle numero proprietaire

        for (size_t i = 0; i < _parcelles.size(); ++i)
        {
            auto parcelle = _parcelles[i];
            file << parcelle->getType() << ' ' << parcelle->getNumber()
                << ' ' << parcelle->getOwner() << ' ';

            auto type = parcelle->getType();

            if (type == "ZU")
            {
                auto zone = std::static_pointer_cast<ZU<T>>(parcelle);
                file << parcelle->getConstructableAreaPercentage() << '
                    ' << zone->getConstructedArea();
            }
            else if (type == "ZAU")
            {
                file << parcelle->getConstructableAreaPercentage();
            }
            else if (type == "ZA")
            {
                auto zone = std::static_pointer_cast<ZA<T>>(parcelle);
                file << zone->getCultureCultivee();
            } // ZN got no extra fields

            file << std::endl;
            // print coordinates
            const auto summits = parcelle->getShape().getSummits();
```

```
            for (size_t i = 0; i < summits.size(); ++i)
                file << '[' << summits[i].getX() << ';' << summits[i].
                    getY() << "] ";

            file << std::endl;
        }
        file.close();
    }
private:
    std::vector<std::shared_ptr<Parcelle<T>>> _parcelles;
};
```

7.1 Explication

La classe Carte représente un groupe de parcelles organisées. Elle permet de charger, manipuler et sauvegarder ces parcelles à partir de fichiers. Elle permet aussi de traiter les différents types de zones urbaines et agricoles en fonction de leurs spécificités.

8 Conclusion

Voici le travail que nous avons pu réaliser.