DELARUELLE Hugo - DARGERE Lucas
DEPEYRIS Julien - LAVAUX Lucas

SEC A2

CPP - TP4 : Gestion d’un PLU

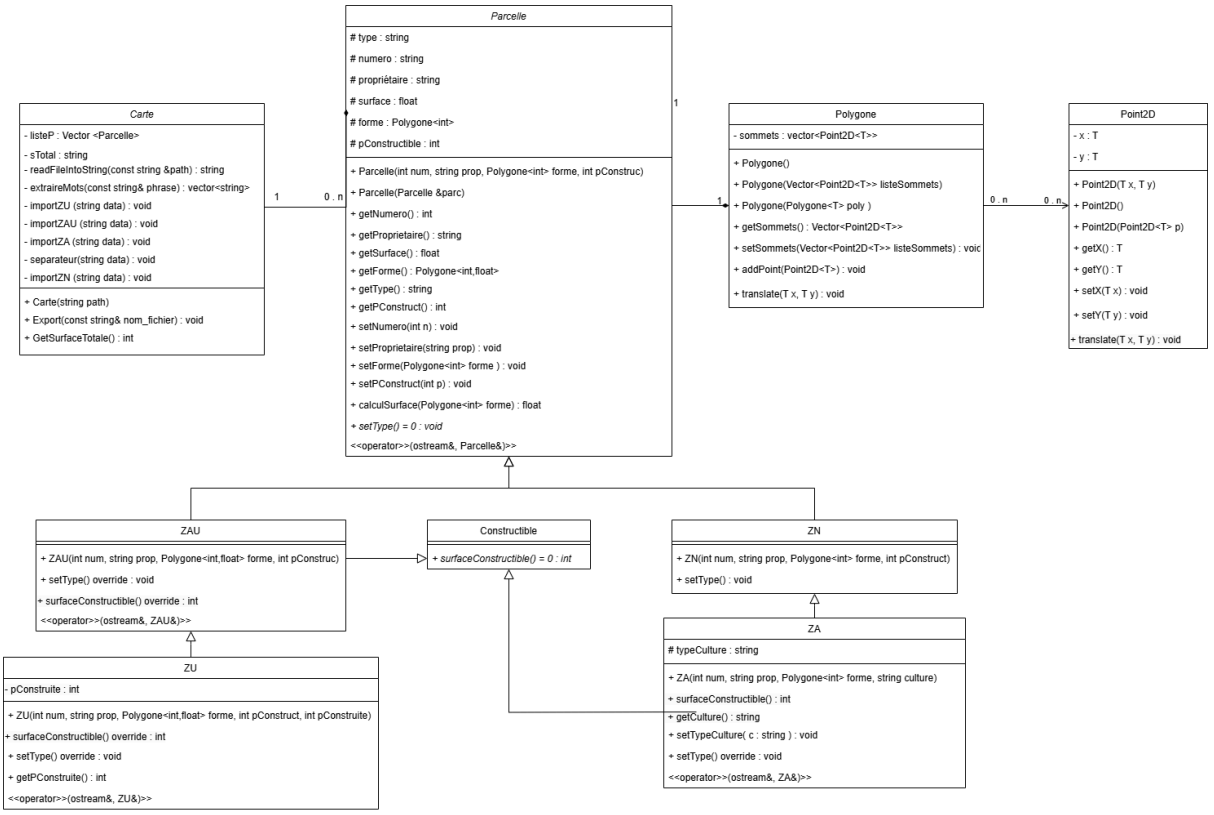
Table des matières

1. Objectif

Le but de ce TP est de créer un ensemble de classes 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.

2. Conception du projet

2.1. Diagramme de classe



2.2. Changements

2.2.1. setType

Pour la méthode virtuel pure setType non avons décidé de retirer le passage de paramètre, nous préférons spécifier dans la redéfinition de celle-ci dans chaque classe de zone.

Exemple : spécification de la classe ZN

```
void ZN::setType() {
    this->type = "ZN" ;
}
```

2.2.2. setPConstruct

Nous avons décidé de rajouter un setter dans la classe Parcelle pour changer la valeur de l'attribut protected. On passe en paramètre la valeur qu'on veut modifier.

```
void Parcelle::setPConstruct(int p) {
    this->pConstructible = p ;
```

3. Conception des classes

3.1. Point2D

La classe Point2D permet de créer des points pour former une structure. La classe permet l'utilisation de plusieurs types pour son utilisation grâce aux templates.

```
template <typename T>
```

Cela permet lors de la définition des attributs, des méthodes, des paramètres de ne pas spécifier les types.

3.1.1. Constructeurs

On peut créer un point grâce aux différents constructeurs de la classe :

```
Point2D(T x, T y) ;
Point2D() ;
Point2D(const Point2D<T> & p) ;
```

- Le premier crée un point à partir des coordonnées passer en paramètre.
- Le second crée un point par défaut de coordonnées (0,0).
- Le dernier permet de créer un point grâce à un autre point, cela permet de le dupliquer et d'effectuer des modifications sans perdre le point avec les changements.

3.1.2. Attribut

La classe possède des attributs qui permettent de stocker les coordonnées en X et en Y du point. La classification "protected" permet aux enfants de la classe d'y avoir accès également.

```
protected:
    T x, y;
```

3.1.3. Méthodes

Les méthodes getters et setters de la classe permettent de récupérer et de modifier les attributs de la classe.

Exemple avec pour l'attribut X:

```
void setX(T x) ;
```

```
template <typename T>
void Point2D<T>::setX(T x) {
    this->x = x;
```

```
T getX() ;
```

```
template <typename T>
T Point2D<T>::getX() {
    return x;
```

Une méthode "translate" permet de déplacer le point :

```
template <typename T>
void Point2D<T>::translate(T x, T y) {
    this->x += x;
    this->y += y;
```

3.2. Polygone

Cette classe utilise la classe Point2D pour créer des formes en stockant les différents points de cette forme. Elle utilise également les templates pour les cas où plusieurs types pourraient être utilisés.

3.2.1. Constructeurs

On peut créer un polygone grâce aux différents constructeurs de la classe :

```
Polygone() ;
Polygone(const vector<Point2D<T>> listeSommets) ;
Polygone(const Polygone<T> & poly) ;
```

Un polygone peut être créé à partir d'une liste de sommets ou d'un polygone déjà existant.

3.2.2. Atribut

La classe possède un attribut qui permet de stocker les sommets de la forme pour pouvoir les utiliser plus tard. Il est notamment appelé dans le constructeur pour renseigner les sommets.

```
protected:
    vector<Point2D<T>> sommets;
```

```
template <typename T>
Polygone<T>::Polygone(const vector<Point2D<T>> listeSommets) : sommets(listeSommets) {}
```

3.2.3. Méthodes

Les méthodes getter et setter de la classe permettent de récupérer et de modifier l'attribut de la classe.

Exemple avec pour l'attribut sommets:

```
vector<Point2D<T>>& getSommets() ;
void setSommets(vector<Point2D<T>> listeSommets) ;
```

```
template <typename T>
vector<Point2D<T>>& Polygone<T>::getSommets() {
    return sommets ;
}
```

```
template <typename T>
void Polygone<T>::setSommets(vector<Point2D<T>> listeSommets) {
    sommets = listeSommets  ;
}
```

Pour ajouter un point supplémentaire on peut passer par la méthode addPoint :

```
void addPoint(Point2D<T> p) ;
```

```
template <typename T>
void Polygone<T>::addPoint(Point2D<T> p) {
    this->sommets.push_back(p);
}
```

Une dernière méthode permet de translater la forme en entière :

```
template <typename T>
void Polygone<T>::translate(T x, T y) {
    for(typename vector<Point2D<T>>::iterator it = this->sommets.begin(); it != this->sommets.end(); ++it) {
        it->translate(x,y);
    }
}
```

Nous avons aussi fait une surcharge d'opérateur sur "<<" :

```cpp
template <typename T>
ostream& operator<<(ostream& os,Polygone<T> & p) {
    for(typename vector<Point2D<T>>::iterator it = p.getSommets().begin(); it != p.getSommets().end(); it++)
    {
        os << "[" << it->getX() << "," << it->getY() << "] ";
    }
    return os;
}
```

Elle permet d'afficher correctement tous les sommets de la forme.

3.3. Parcelle

La classe Parcelle permet de stocker les informations sur la parcelle telle que sa forme, son propriétaire, son numéro, son type, la taille de sa surface et le pourcentage constructible. La classe est abstraite, ceci est dû à sa méthode virtuelle pure setType(). Dû à cela, aucune instance de parcelle ne peut être créée.

3.3.1. Constructeurs

On peut créer un polygone grâce aux différents constructeurs de la classe :

```cpp
Parcelle(int num, string prop, Polygone<int> forme, int pConstruc) ;
Parcelle(const Parcelle & parc) ;
```

Une parcelle peut être créée à partir d'un polygone et des différentes informations nécessaires à sa construction ou d'une parcelle déjà existante.

3.3.2. Attributs

On retrouve les informations sur la parcelle en "protected" pour donner l'accès à ses enfants.

```cpp
protected:
    std::string type ;
    int numero ;
    std::string proprietaire ;
    float surface ;
    Polygone<int> forme ;
    int pConstructible ;
```

3.3.3. Méthodes

Les méthodes getters et setters de la classe permettent de récupérer et de modifier les attributs de la classe.

Exemple avec pour l'attribut pConstructible:

```cpp
int getPConstruct() ;
void setPConstruct(int p) ;
```

```cpp
int Parcelle::getPConstruct() {
    return pConstructible ;
}
```

```cpp
void Parcelle::setPConstruct(int p) {
    this->pConstructible = p ;
}
```

On trouve également une méthode qui permet de calculer la surface de la zone mais nous reviendrons dessus ultérieurement.

La méthode virtuel pure setType() est définie dans la classe Parcelle

On retrouve également une surcharge d'opérateur "<<" :

```
ostream& operator<<(ostream& os, Parcelle& p) {
    os << "Parcelle n° " << p.getNumero() << " :" << endl;
    os << "\tType : " << p.getType() << endl;
    os << "\tPolygone : " << p.getForme() << endl;
    os << "\tPropriétaire : " << p.getProprietaire() << endl;
    os << "\tSurface : " << p.getSurface() << endl;
    return os;
}
```

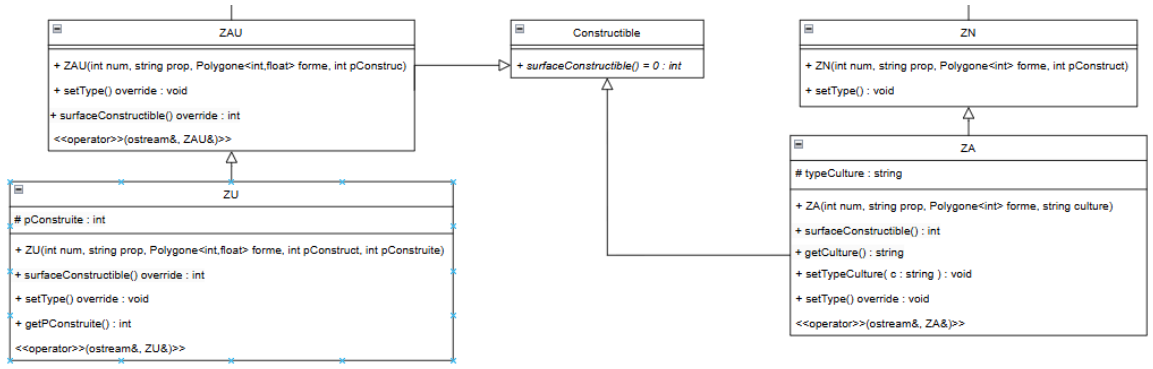
4. Le différentes zones

Nous avons d'abord créé la classe Constructible qui est abstraite. Cela est dû à son unique méthode surfaceConstructible qui est virtuel pure.

```
virtual int surfaceConstructible() = 0 ;
```

Elle sera hérité par les classes ZU, ZAU, ZA.

Voici un diagramme UML qui permet de montrer un peu mieu les liens entre les classes :



4.1. ZAU

La parcelle ZAU possède un constructeur:

```
ZAU(int num, std::string prop, Polygone<int> forme, int pConstruc);
```

Il fait directement appel au constructeur de parcelle,

```
ZAU::ZAU(int num, std::string prop, Polygone<int> forme, int pConstruc)
    : Parcelle(num, prop, forme, pConstruc) { setType() ; }
```

et fait appel à la méthode setType() hérité de la classe Parcelle qui est override.

```
void setType() override ;
```

```
void ZAU::setType() {
    this->type = "ZAU" ;
}
```

De plus la classe ZAU possède une autre méthode override qui est hérité de la classe "Constructible": `int surfaceConstructible() override;`

```
int ZAU::surfaceConstructible() {
    return ((pConstructible/100) * this->getSurface()) ;
}
```

Ainsi qu'une surcharge de l'opérateur "<<" pour l'affichage de la zone et de ses informations:

```
ostream& operator<<(ostream& os, ZAU& z) {
    os << "% constructible : " << z.getPConstruct() << endl ;
    return os;
}
```

4.2. ZU

La parcelle ZU possède une spécificité supplémentaire par rapport à la classe ZU. Il s'agit d'un attribut, le pourcentage déjà construit "pConstruite". `int pConstruite;`

Le constructeur est donc un peu modifié :

```
ZU::ZU(int num, std::string prop, Polygone<int> forme, int pConstruc, int pConstruite)
    : ZAU(num, prop, forme, pConstruc), pConstruite(pConstruite) { setType(); }
```

Et un getter est créé : `int getPContruite() ;`

```
int ZU::getPContruite() {
    return this->pConstruite;
}
```

Ainsi qu'une surcharge de l'opérateur "<<" pour l'affichage de la zone et de ses informations:

```
ostream& operator<<(ostream& os, ZU& z) {
    os << "% constructible : " << z.getPConstruct() << endl ;
    os << "surface construite : " << (z.getSurface() * (z.getPContruite()/100)) ;
    os << "surface à construire restante : " << z.surfaceConstructible() << endl ;
    return os;
}
```

4.3. ZN

La classe ZN possède un constructeur et une méthode setType override:

```
class ZN : public Parcelle
{
    public:
        ZN(int num, std::string prop, Polygone<int> forme, int pConstruct);
        void setType() override ;
} ;
```

```
ZN::ZN(int num, std::string prop, Polygone<int> forme, int pConstruct)
    : Parcelle(num, prop, forme, pConstruct) { setType() ; }
```

```
void ZN::setType() {
    this->type = "ZN" ;
}
```

4.4. ZA

La classe ZA permet de spécifier la classe ZN avec un type de culture. Elle possède donc un setter et un getter en plus.

```
void setTypeCulture(std::string c) ;
void setType() override ;
```

```
void ZA::setTypeCulture(std::string c) {
    this->typeCulture = c ;
}
```

```
std::string ZA::getCulture() {
    return this->typeCulture ;
}
```

De plus la classe ZA hérite de la classe Constructible et possède donc une méthode surfaceConstructible, elle permet de définir si une zone peut être constructible ou non:

```
int ZA::surfaceConstructible() {
    int s = static_cast<int>(0.1 * this->getSurface());
    if(s > 200)
        return 200 ;
    else
        return s ;
}
```

Une surcharge de l'opérateur "<<" permet de montrer la culture qui est cultivé dans la zone:

```
ostream& operator<<(ostream& os, ZA& z) {
    os << "Type culture : " << z.getCulture() << endl ;
    return os ;
}
```

5. La classe Carte

5.1. Calcul de surface

5.1.1. Surface d'une zone

Dans la classe Parcelle nous vous avons parlé d'une méthode "calculSurface", celle-ci permet de calculer la surface de la zone dans le sens trigonométrique mais aussi dans le sens inverse grâce à la valeur absolue:

```
float Parcelle::calculSurface(Polygone<int> forme) {
    float result = 0 ;
    vector<Point2D<int>>::iterator itNext ;
    for(typename vector<Point2D<int>>::iterator it = forme.getSommets().begin(); it != forme.getSommets().end(); it++)
    {
        if(std::next(it,1) != forme.getSommets().end()){
            itNext = std::next(it,1) ;
        } else {
            itNext = forme.getSommets().begin() ;
        }
        result +=  it->getX()*itNext->getY() - itNext->getX()*it->getY() ;
    }
    return (abs(0.5*result)) ;
}
```

Grâce à la formule suivante:

$$surface = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$

5.1.2. Surface Totale des zones

La classe carte contient via ses attributs la liste des parcelles, puis grâce la méthode "GetSurfaceTotale" on fait appel une par une les parcelles en prenant leur surface et en les additionnant:

```
int Carte::GetSurfaceTotale(){
  int SurfaceTotale = 0;
  for(int i = 0 ; i<ListaParcelles.size(); i++){
    SurfaceTotale = SurfaceTotale + ListaParcelles[i]->getSurface();
  }
  return SurfaceTotale;
}
```

5.2. Import

5.2.1. ReadFile

Nous commençons par lire le fichier avec la liste des zones en transformant les données du fichier dans une variable string:

```
string readFileIntoString(const string& path) {
  string  content;
    ifstream input_file(path);
    if (!input_file.is_open()) {
        cerr << "Could not open the file - '"
             << path << "'" << endl;
        exit(EXIT_FAILURE);
    }
    content = string((std::istreambuf_iterator<char>(input_file)), std::istreambuf_iterator<char>())+string("\n")
    input_file.close();

  return content;
}
```

5.2.2. Séparateur

Une fois les données dans la variables on vient séparer les informations avec la méthode séparateur, chaque ligne 1 par 1 est passé et on vient récupérer le type de zone:

```
void separateur(string data){
  int line = 0;
  string phrase = "";
  for (int i = 0 ; i <  data.size(); i++)
    {
      phrase += data[i];
        // check whether parsing character is punctuation or not
      if (data[i] =='\n')
        {
          if(line ==1){
            size_t espaceIndex = phrase.find(' ');
            string Zone = phrase.substr(0, espaceIndex);
            if(Zone =="ZU"){
              ImporZU(phrase);
            } else if (Zone =="ZAU"){
              ImporZAU(phrase);
            }else if (Zone == "ZA"){
              ImporZA(phrase);
            }else if (Zone=="ZN"){
              ImporZN(phrase);
            }

          line = 0 ;
          phrase = "";
          }
          else{
            line++;
          }
        }
    }
}
```

Une fois le type de zone obtenue on vient import le bon type de zone, exemple avec une zone ZA:

```
void ImporZA (string data){
  vector<string> mots = extraireMots(data);
  string Zone = "ZA" ;
  int Numéro = stoi(mots[1]);
  string Propriétaire = mots[2];
  string typeCulture = mots[3];
  vector<Point2D<int>> points ;
  for(int i=4;i<mots.size();i++){
    int x = 0;
    int y = 0;
    mots[i] = mots[i].substr(1, mots[i].size() - 2); // Enlève '[' et ']'
  // Trouver la position du ';'
    size_t posPointVirgule = mots[i].find(';');
      // Extraire les parties avant et après le ';'
    string avantStr = mots[i].substr(0, posPointVirgule);
    string apresStr = mots[i].substr(posPointVirgule + 1);
      // Convertir les chaînes en entiers
    x = stoi(avantStr);
    y = stoi(apresStr);
    points.push_back(Point2D<int>(x,y));
  }
  Polygone<int> poly = Polygone<int>(points);
  ZA* parcelle = new ZA( Numéro, Propriétaire, poly , typeCulture);
  ListaParcelles.push_back(parcelle);
}
```

Dans celle-ci on fait appel à “extraireMots” pour récupérer les différentes informations et les attribuer correctement dans le nouvel objet ZA créer puis push dans la liste de parcelle.

5.2.3. Import depuis le main

Une méthode est créée pour créer une zone supplémentaire directement depuis le main:

```
void Carte::ImportParcelle(string type ,int numero, string proprietaire, int pConstructible , int surfaceConstruite, string typeCulture, string ListeDePoint ){
string phrase = "";
vector<Point2D<int>> points ;
vector<string> mots = extraireMots(ListeDePoint);
for(int i=0;i<mots.size();i++){
    int x = 0;
    int y = 0;
    mots[i] = mots[i].substr(1, mots[i].size() - 2); // Enlève '[' et ']'
  // Trouver la position du ';'
    size_t posPointVirgule = mots[i].find(';');
      // Extraire les parties avant et après le ';'
    string avantStr = mots[i].substr(0, posPointVirgule);
    string apresStr = mots[i].substr(posPointVirgule + 1);
      // Convertir les chaînes en entiers
    x = stoi(avantStr);
    y = stoi(apresStr);
    points.push_back(Point2D<int>(x,y));

  }

  if(type == "ZA"){
    Polygone<int> poly = Polygone<int>(points);
    ZA* parcelle = new ZA( numero, proprietaire, poly , typeCulture);
    ListaParcelles.push_back(parcelle);


  }else if(type == "ZAU"){
    Polygone<int> poly = Polygone<int>(points);
    ZAU* parcelle = new ZAU( numero, proprietaire, poly , pConstructible);
    ListaParcelles.push_back(parcelle);


  }
  else if(type == "ZN"){
    Polygone<int> poly = Polygone<int>(points);
    ZN* parcelle = new ZN( numero, proprietaire, poly,-1);
    ListaParcelles.push_back(parcelle);


  }else if(type == "ZU"){
    Polygone<int> poly = Polygone<int>(points);
    ZU* parcelle = new ZU( numero, proprietaire, poly , pConstructible, surfaceConstruite);
    ListaParcelles.push_back(parcelle);
```

5.3. Export

La méthode Export permet de copier dans un fichier toute les informations contenue dans la liste de parcelle:

```
void Carte::Export(const string& nom_fichier){
   ofstream fichier(nom_fichier);
    // Vérifier si le fichier est ouvert
    if (fichier.is_open()) {
     for(int i = 0 ; i<ListaParcelles.size(); i++){
        // Écrire dans le fichier
         fichier<< ListaParcelles[i]->getType()<<" ";
         fichier<< ListaParcelles[i]->getNumero()<<" ";
         fichier<< ListaParcelles[i]->getProprietaire()<<" ";


        if ((ListaParcelles[i]->getType())=="ZA"){
          auto za = dynamic_cast<ZA*>(ListaParcelles[i]) ;
          fichier << za->getCulture() << "\n";

        }else if((ListaParcelles[i]->getType())=="ZAU"){
          fichier<< ListaParcelles[i]->getPConstruct()<<"\n";

        }else if((ListaParcelles[i]->getType())=="ZN"){
          fichier<<"\n";

        }else if((ListaParcelles[i]->getType())=="ZU"){
          fichier<< ListaParcelles[i]->getPConstruct()<<" ";
          auto zu = dynamic_cast<ZU*>(ListaParcelles[i]) ;
          fichier << zu->getPContruite() << "\n";

        }
        fichier<< ListaParcelles[i]->getForme()<<"\n";
     }
        // Fermer le fichier
        fichier.close();
    } else {
        cerr << "Impossible d'ouvrir le fichier !" << endl;
    }
}
```

6. Jeu de test

7. Conclusion

Ce TP nous a permis de mettre en place beaucoup des cours vue en 1er et 2ème année, tels que la programmation objet, l'héritage, les templates, les classes virtuels, les googles tests, la documentation Doxygen, et d'autres.

Nous avons pu réfléchir en groupe à la façon dont nous voulions construire le projet et après plusieurs propositions, nous avons pris certaines décisions et supprimer d'autres. Ce fut une bonne expérience pour développer notre travail d'équipe et la construction d'un projet.

Nous n'avons pas eu le temps d'aborder certains aspects du projet et amélioration possible comme la vérification de polygone croisé.