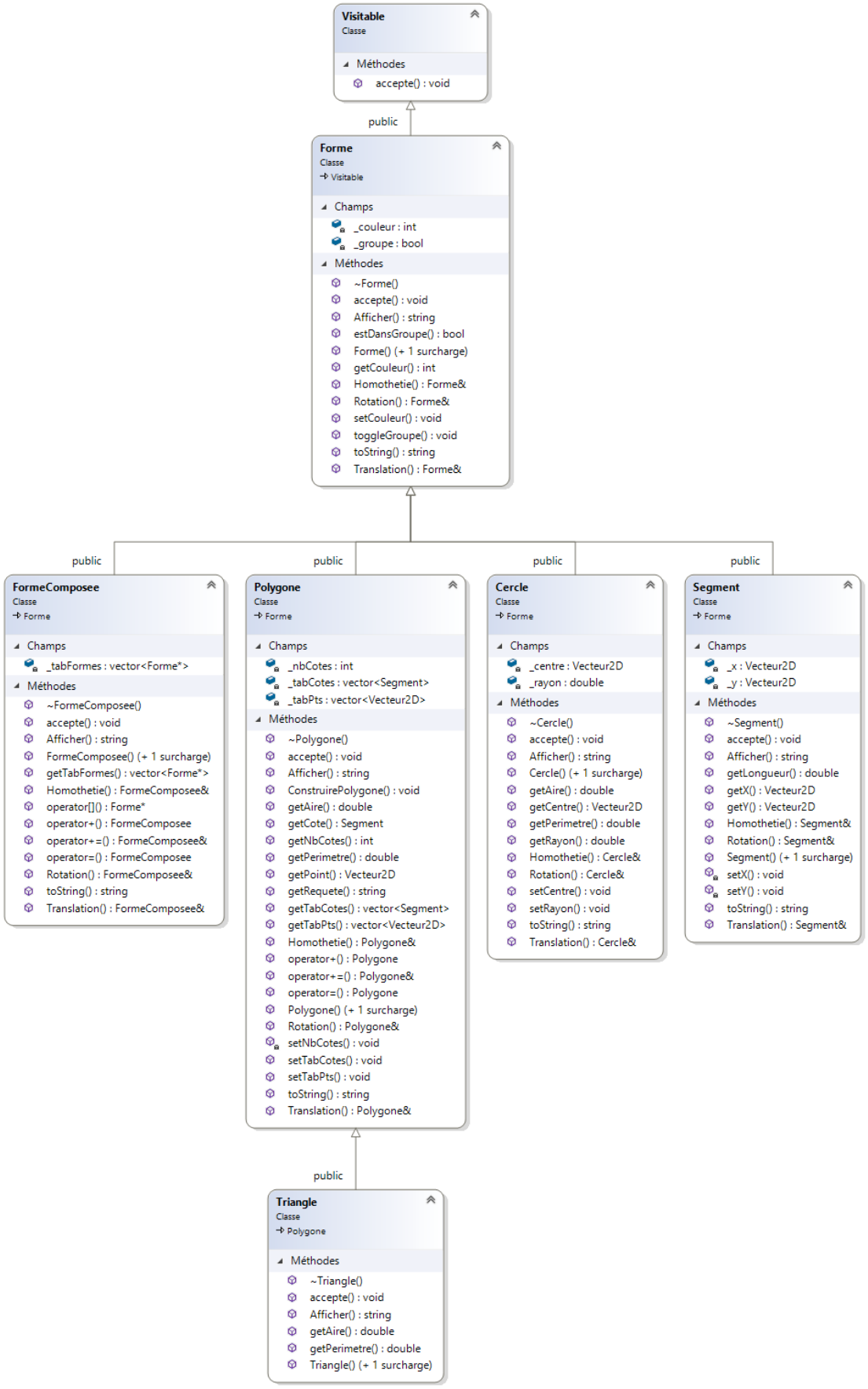
Rapport – Projet de Synthèse

Table des matières

1. Forme Géométrique



Une forme est caractérisée par une couleur qui est une constante entière. Il existe différentes formes comme les segments (2 points), les cercles (un centre et un rayon), et les polygones.

Les polygones sont des formes à plusieurs cotés. Le triangle est un polygone, il hérite donc de la classe polygone. Ce dernier est caractérisé par un tableau de points et un tableau de segments qui le composent.

Une forme composée est une forme qui en comporte plusieurs. C'est donc un tableau de formes.

2. Transformations géométriques

L'application offre la possibilité d'appliquer trois sortes de transformations géométriques aux formes :

- Une translation : c'est un déplacement qui correspond à un vecteur (direction) donné

  On ajoute simplement les coordonnées du vecteur donné aux coordonnées de la forme

```
Vecteur2D& Vecteur2D::Translation(const Vecteur2D &v)
{
    *this += v;
    return *this;
}
```

- Une homothétie : c'est une sorte de zoom obtenue à partir d'un point et d'un vecteur (direction) qui indique le rapport d'homothétie.

  On multiplie les coordonnées de la forme avec le rapport d'homothétie et on ajoute à ce résultat les coordonnées du point.

```
Vecteur2D& Vecteur2D::Homothetie(const Vecteur2D &v, double rapport)
{
    _x = (_x - v._x)*rapport + v._x;
    _y = (_y - v._y)*rapport + v._y;
return *this;
}
```

- Une rotation atour d'un point p avec un angle de rotation

  Exemple : Rotation d'un point

```
Vecteur2D& Vecteur2D::Rotation(const Vecteur2D &v, double angle){
    _x = v._x + _x * cos(angle) - v._y * sin(angle);
    _y = v._x + _x * sin(angle) + v._y * cos(angle);
    return *this;
}
```

3. Client TCP/IP

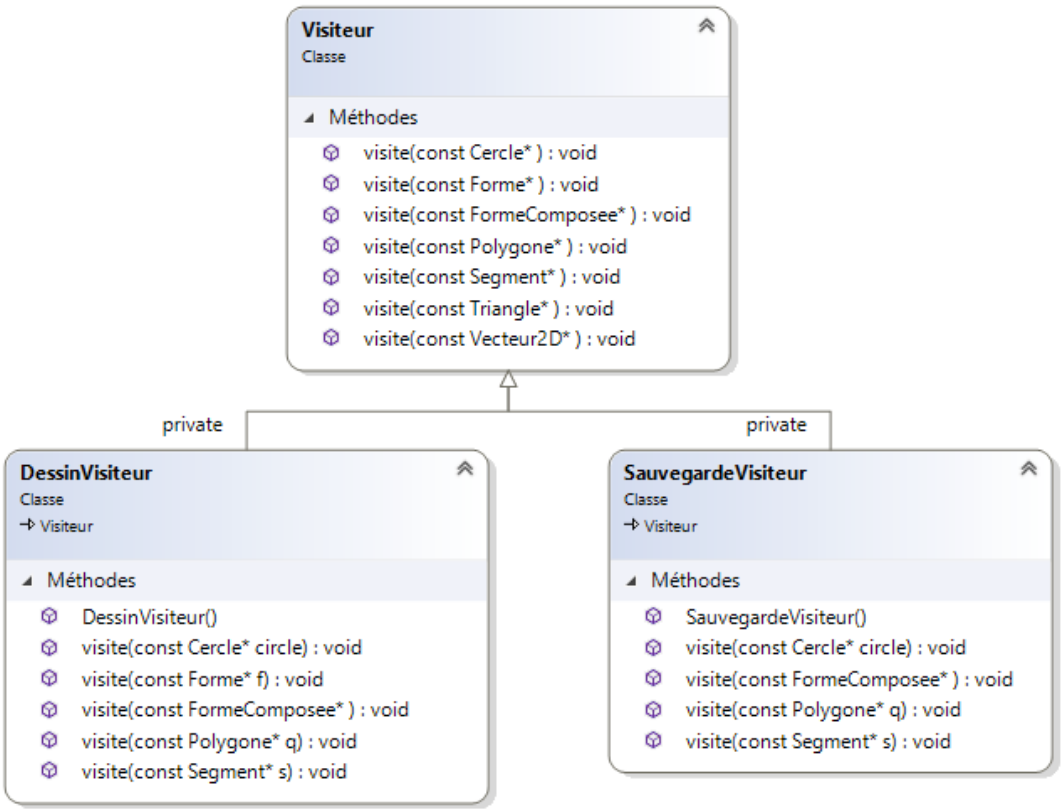
Le client c++ utilise la libraire winsock, et envoie des requêtes contenant plusieurs paramètres, séparés par un « ; » :

- Le type de Forme : Segment, composée, cercle ou polygone
- La couleur de la forme (sous forme d'entier)
- Les coordonnées des points de la forme (sous la forme x1 ;y1 ;x2 ;y2 ;…)
- Pour les formes composées, la requete sera sous forme « Composee #Forme1#Forme2#... »

Le design pattern Singleton a été utilisé afin de garantir l'unicité de l'instance du Client :

```
#pragma once
class SingletonWinsock
{
public:
    static SingletonWinsock& Instance();
private:
    static SingletonWinsock m_instance;
    SingletonWinsock();
    ~SingletonWinsock();
};
```

Les requêtes sont créées et envoyées via le design pattern visiteur, qui permet la séparation de la partie données et de la partie traitement



Exemple du visiteur pour Cercle :

```
void DessinVisiteur::visite(const Cercle * circle) const
{
    // Envoie l'ordre de créer une nouvelle fenêtre
    if (!ExistingFrame)
        send("newFrame");

    //Création de la requête et envoi
    cout << "je Dessin un cercle ";
    double x = circle->getCentre().getX() - circle->getRayon();
    double y = circle->getCentre().getY() + circle->getRayon();
    double diam = circle->getRayon() * 2;
    string typ = "Cercle;";

       string requete = typ + to_string(circle->getCouleur()) + ";"
                    + to_string(x) + ";"
                    + to_string(y) + ";"
                    + to_string(diam);
    cout << "Requête à envoyer : " << requete;

    send(requete);
}
```

Exemple du visiteur pour une forme composée :

```
void DessinVisiteur::visite(const FormeComposee *fc) const
{
    bool newFrame = false;
    if (!ExistingFrame)
    {
        send("newFrame");
        ExistingFrame = true;
        newFrame = true;
    }

    cout << "je Dessin une forme compose";
    string requete = "Composee#";
    int taille = fc->getTabFormes().size();
    for (int i = 0; i < taille; i++)
    {
        (*fc)[i]->accepte(this);

    }

    if (newFrame) ExistingFrame = false;
}
```

4. Dessin

Le serveur de dessin se connecte, reçoit les requêtes et les traite de façon à pouvoir dessiner les formes demandées.

Connexion :

```
try
{

    int portServeur = 16000;
    ServerSocket serveur = new ServerSocket(portServeur);
    System.out.println("Serveur démarré :" + serveur);
    int noClient = 1;
    while(true)
    {
        Socket client = serveur.accept();
        System.out.println("Connexion reussie numero " + noClient);
        ++noClient;
        Receveur r = new Receveur(client, noClient);
        r.start();
    }
}
catch(Exception e)
{
    System.out.println(e.getMessage());
}
```

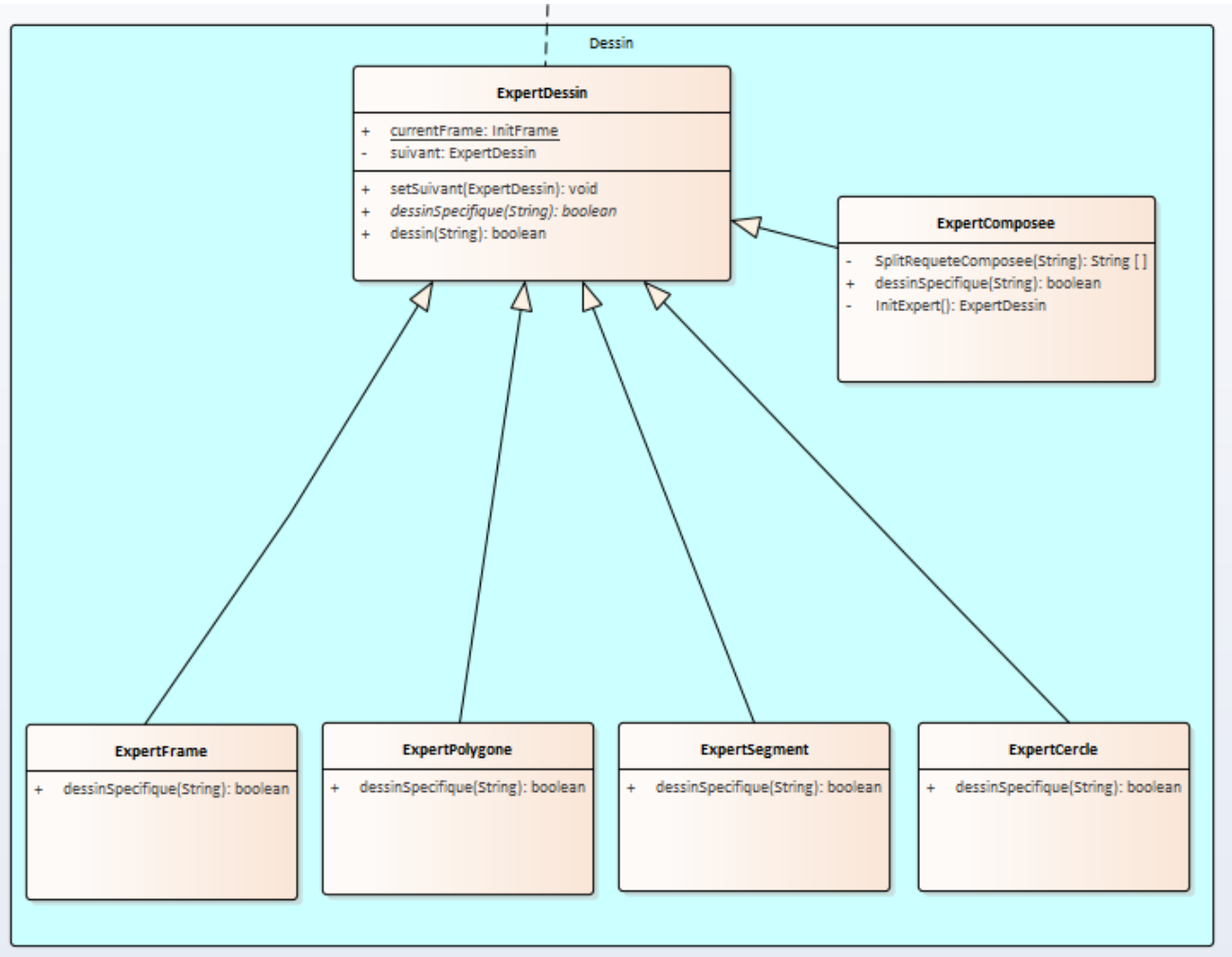
Reception des données :

```
public class Receveur extends Thread
{
    private BufferedReader fluxEntrant;
    int noClient;
    public Receveur(Socket client, int noClient) throws IOException
    {
        fluxEntrant = new BufferedReader(
            new InputStreamReader(client.getInputStream()));
        this.noClient = noClient;

    }
```

```
    private void expert(String requete)
    {
       //Initialisation des experts
        System.out.println("Le client a envoyé" + requete);
        ExpertDessin expertComposee = new ExpertComposee();
        ExpertDessin expertFrame = new ExpertFrame();
        ExpertDessin expertCercle = new ExpertCercle();
        ExpertDessin expertPolygone = new ExpertPolygone();
        ExpertDessin expertSegment = new ExpertSegment();
        expertComposee.setSuivant(expertFrame);
        expertFrame.setSuivant(expertCercle);
        expertCercle.setSuivant(expertPolygone);
        expertPolygone.setSuivant(expertSegment);

        expertFrame.dessin(requete);

    }
    public void run()
    {
        try
        {
            while (!this.isInterrupted())
            {
                String requete = fluxEntrant.readLine();
                expert(requete);
            }
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Dessin :

Pour dessiner, le serveur utilise le design pattern Chain of Responsability, ce qui permet de faciliter la maintenance du code, et de ne pas se soucier de la forme de la requête.



Il y a un expert par forme, en plus de ExpertFrame qui va enclencher l'action d'initialiser la fenetre de dessin.

Chaque expert a comme objectif d'éclater la requête et de convertir les champs en entier ou double si besoin.

Exemple pour ExpertSegment :

```
import java.awt.Graphics2D;
import java.awt.geom.*;

public class ExpertSegment extends ExpertDessin
{
    public boolean dessinSpecifique(String req)
    {
        if (!(req.contains("Segment"))) return false;
        else
        {
            System.out.println("Je dessine un Segment");
            //Eclatement de la requete
            String [] requeteSplitee = req.split(";");
            //conversion et initialisation des ^paramètres de la forme
            double x1 = Double.parseDouble(requeteSplitee[2]);
            System.out.println("x1 =" + x1);
            double y1 = Double.parseDouble(requeteSplitee[3]);
            System.out.println("y1 =" + y1);
            double x2 = Double.parseDouble(requeteSplitee[4]);
            System.out.println("x2 =" + x2);
            double y2 = Double.parseDouble(requeteSplitee[5]);
            System.out.println("y2 =" + y2);
            //Création de la forme
            Line2D.Double ligne = new Line2D.Double(x1, y1, x2, y2);
            //Dessin de la forme, avec comme paramètre la forme
            //et le champ correspondant à sa couleur
            currentFrame.dessinerForme(ligne,requeteSplitee[1]);

        }
        return true;
    }
}
```

L'expert de la forme composée est un peu différent. Il éclate la chaine avec les #, et fait un appel récursif sur les formes qui la compose :

```
    public boolean dessinSpecifique(String request)
    {

        if (!(request.contains("Composee"))) return false;
        else
        {
            String[] req = SplitRequeteComposee(request);
            for (String forme : req)
            {
                ExpertDessin expert = initExpert();
                expert.dessin(forme);
            }
            return true;
        }

     }
```

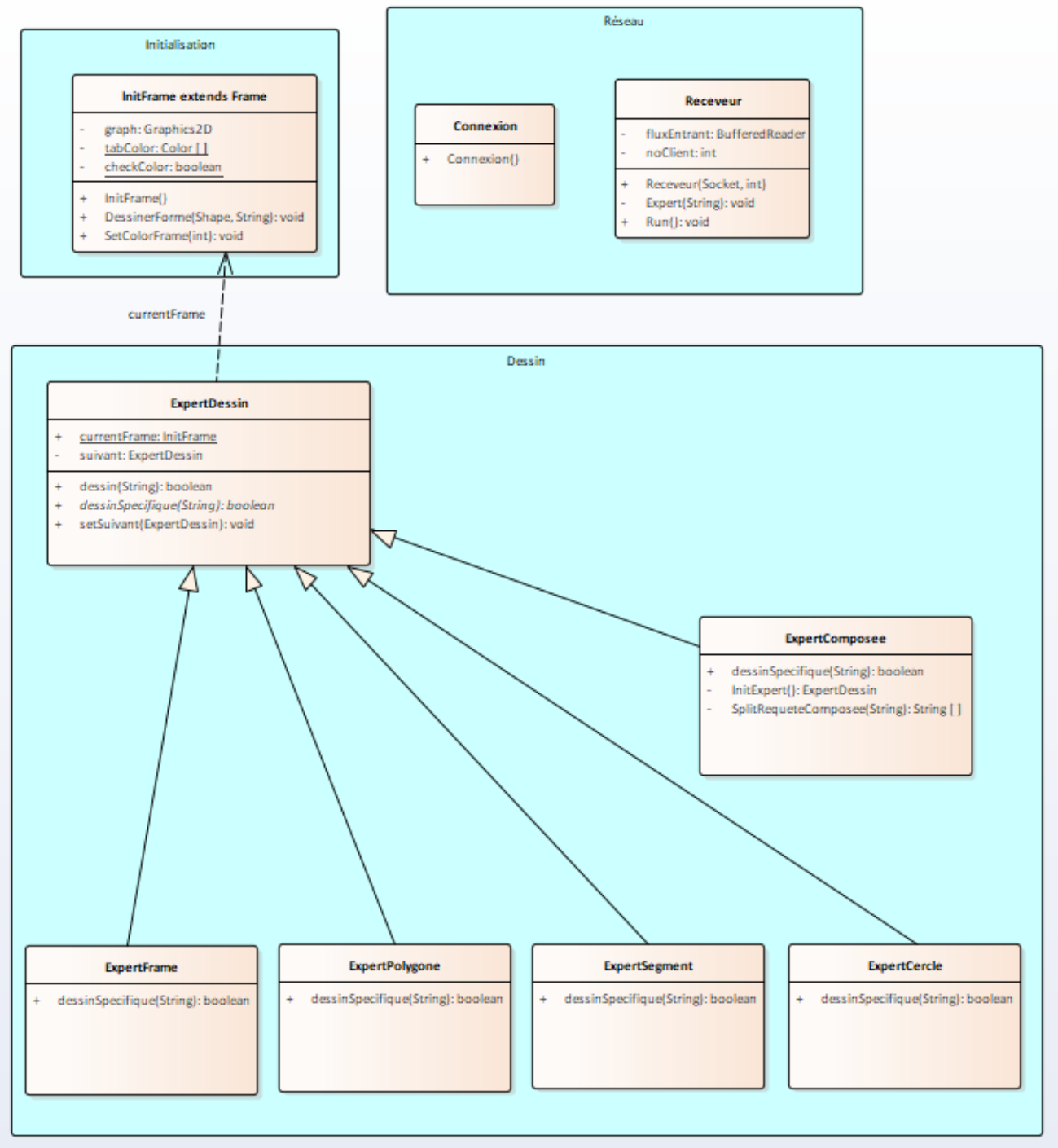
La fenêtre, quant à elle, est initialisée grâce à Expert Frame, puis dessine les formes demandées :

```
public class InitFrame extends Frame{

    private Graphics2D graph;
    private final Color tabColor[] = {RED, GREEN, BLUE, BLACK, YELLOW, CYAN};
    private static boolean checkColor = false;
    public InitFrame()
    {
        //Constructeur lancé par l'ExpertFrame
        super(" Frame tout simple en Active Rendering");

            setUndecorated(true); // empêche de réduire la fenêtre, ce qui provoque des bugs
            setBounds(30, 60, 1000, 1000);
            setIgnoreRepaint(true);
            setVisible(true);
            checkColor = false;


    }
```

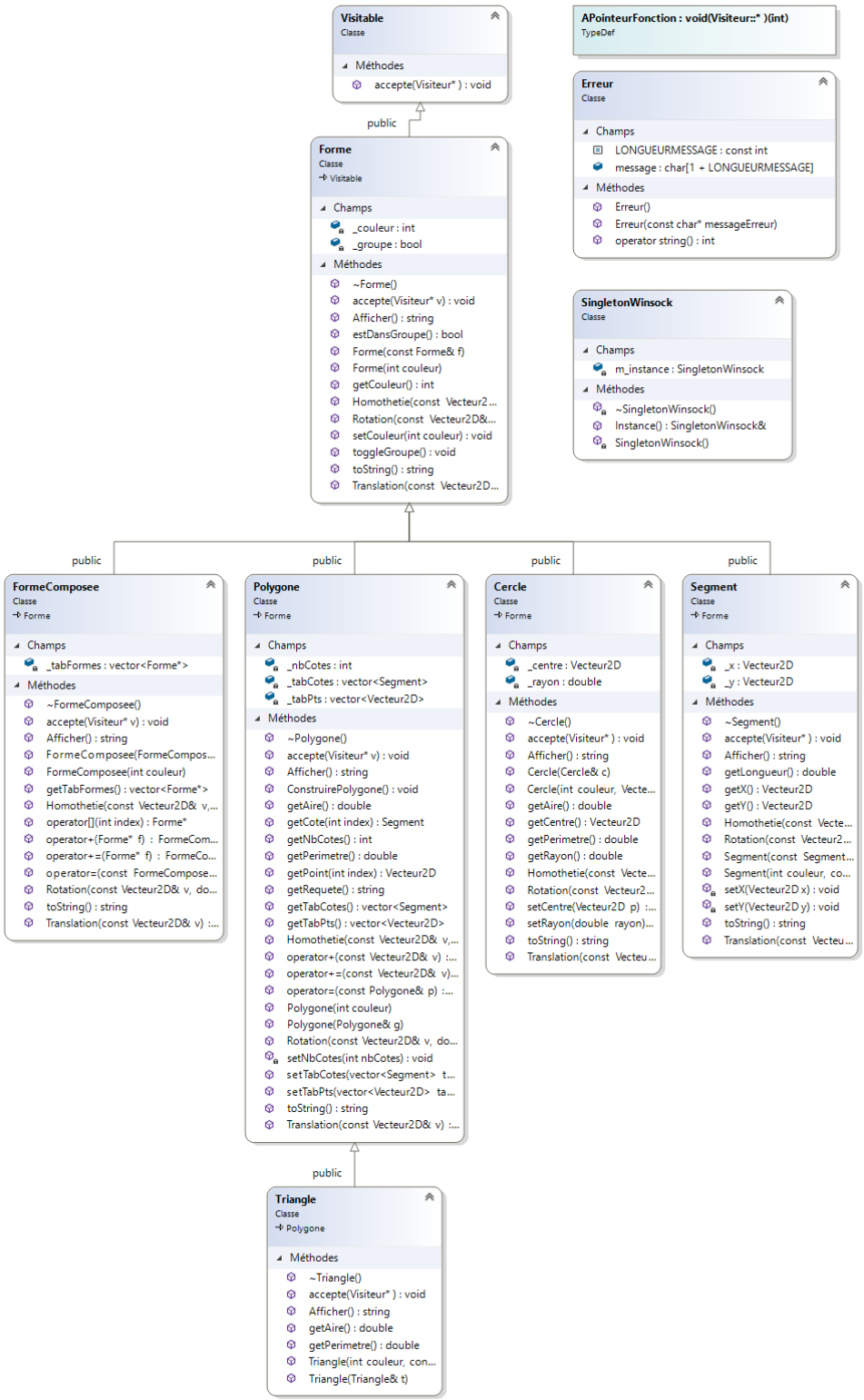
```
    public void dessinerForme(Shape s, String couleur)
    {
        //Dessine la forme demandée
        createBufferStrategy(1);
        try {
            Thread.sleep(150);
        }
        catch(Exception e)
        {
            System.out.println("Ca marche pas");
        }
        BufferStrategy strategie = getBufferStrategy();
        graph = (Graphics2D) strategie.getDrawGraphics();
        if(!checkColor)
        {
            int couleurint = Integer.parseInt(couleur);
            setColorFrame(couleurint);
        }
        graph.draw(s);
        graph.fill(s);
        strategie.show();
        graph.dispose();

    }

    public void setColorFrame(int i)
    {
        //Initialise la couleur, et la fixe pour toutes les autres formes de cette fenêtre
        graph.setColor(tabColor[i]);
        checkColor = true;
    }
}
```

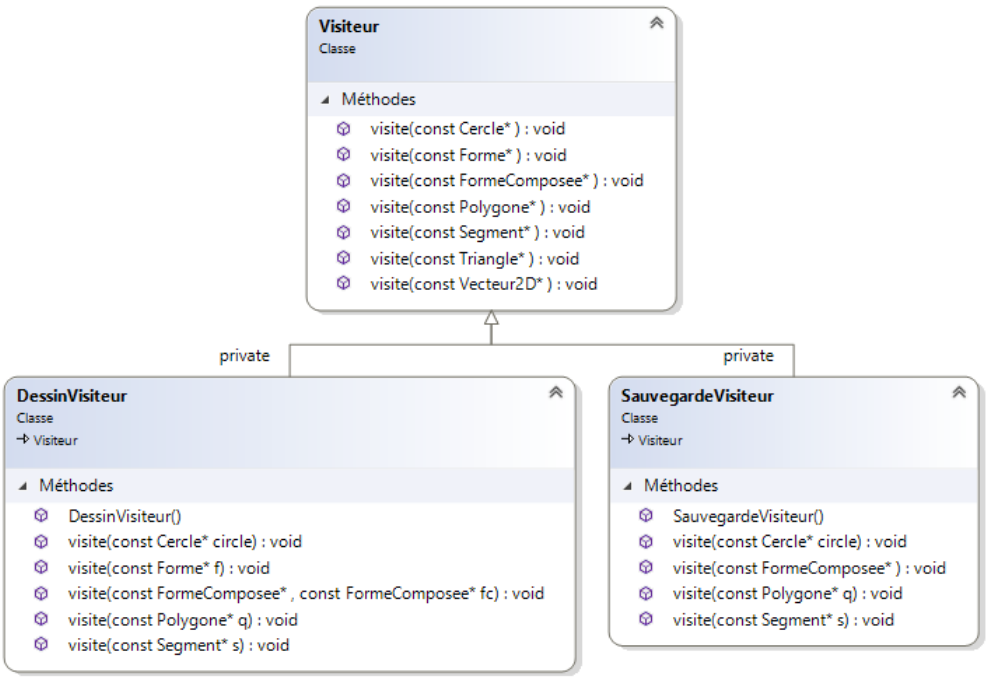
5. UML complet

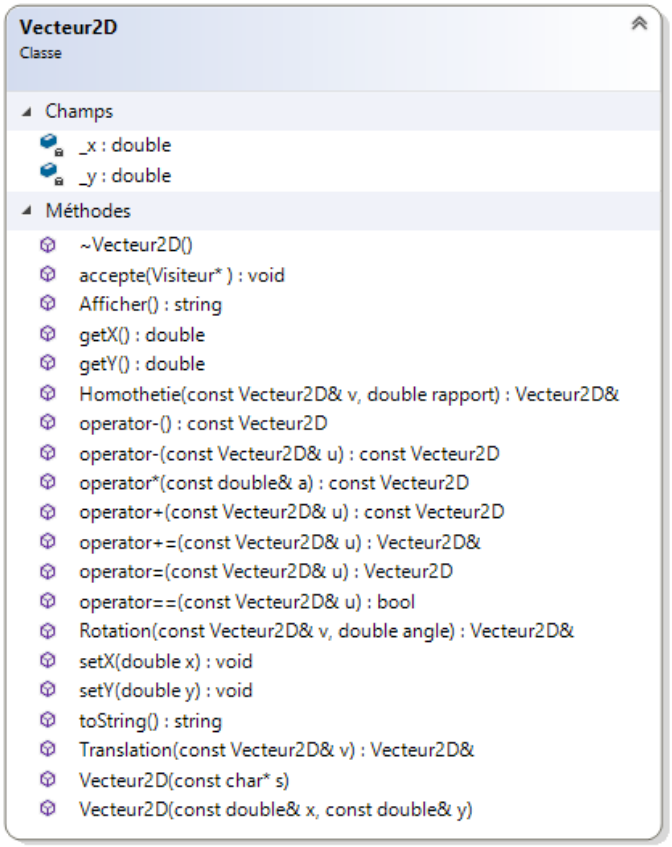
Serveur Java :

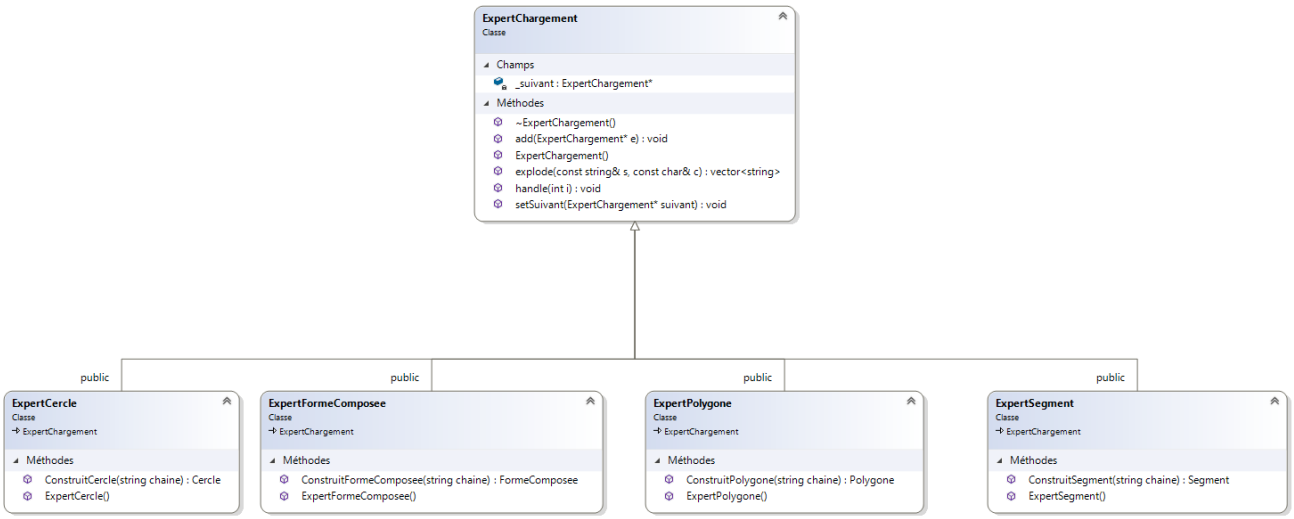


Client c++ :


Visitable
Classe
Méthodes
accepte(Visiteur*) : void
APointeurFonction : void(Visiteur::*)(int)
TypeDef
Erreur
Classe
Champs
LONGUEURMESSAGE : const int
message : char[1 + LONGUEURMESSAGE]
Méthodes
Erreur()
Erreur(const char* messageErreur)
operator string() : int
public
Forme
Classe
Visitable
Champs
_couleur : int
_groupe : bool
Méthodes
~Forme()
accepte(Visiteur* v) : void
Afficher() : string
estDansGroupe() : bool
Forme(const Forme& f)
Forme(int couleur)
getCouleur() : int
Homothetie(const Vecteur2...
Rotation(const Vecteur2D&...
setCouleur(int couleur) : void
toggleGroupe() : void
toString() : string
Translation(const Vecteur2D...
SingletonWinsock
Classe
Champs
m_instance : SingletonWinsock
Méthodes
~SingletonWinsock()
Instance() : SingletonWinsock&
SingletonWinsock()
public
public
public
public
FormeComposee
Classe
Forme
Champs
_tabFormes : vector<Forme*>
Méthodes
~FormeComposee()
accepte(Visiteur* v) : void
Afficher() : string
FormeComposee(FormeCompos...
FormeComposee(int couleur)
getTabFormes() : vector<Forme*>
Homothetie(const Vecteur2D& v,...
operator[](int index) : Forme*
operator+(Forme* f) : FormeCom...
operator+=(Forme* f) : FormeCo...
operator=(const FormeCompose...
Rotation(const Vecteur2D& v, do...
toString() : string
Translation(const Vecteur2D& v) :...
Polygone
Classe
Forme
Champs
_nbCotes : int
_tabCotes : vector
_tabPts : vector<Vecteur2D>
Méthodes
~Polygone()
accepte(Visiteur* v) : void
Afficher() : string
ConstruirePolygone() : void
getAire() : double
getCote(int index) : Segment
getNbCotes() : int
getPerimetre() : double
getPoint(int index) : Vecteur2D
getRequete() : string
getTabCotes() : vector
getTabPts() : vector<Vecteur2D>
Homothetie(const Vecteur2D& v,...
operator+(const Vecteur2D& v) :...
operator+=(const Vecteur2D& v)...
operator=(const Polygone& p) :...
Polygone(int couleur)
Polygone(Polygone& g)
Rotation(const Vecteur2D& v, do...
setNbCotes(int nbCotes) : void
setTabCotes(vector t...
setTabPts(vector<Vecteur2D> ta...
toString() : string
Translation(const Vecteur2D& v) :...
Cercle
Classe
Forme
Champs
_centre : Vecteur2D
_rayon : double
Méthodes
~Cercle()
accepte(Visiteur*) : void
Afficher() : string
Cercle(Cercle& c)
Cercle(int couleur, Vecte...
getAire() : double
getCentre() : Vecteur2D
getPerimetre() : double
getRayon() : double
Homothetie(const Vecte...
Rotation(const Vecteur2...
setCentre(Vecteur2D p) :...
setRayon(double rayon)...
toString() : string
Translation(const Vecteu...
Segment
Classe
Forme
Champs
_x : Vecteur2D
_y : Vecteur2D
Méthodes
~Segment()
accepte(Visiteur*) : void
Afficher() : string
getLongueur() : double
getX() : Vecteur2D
getY() : Vecteur2D
Homothetie(const Vecte...
Rotation(const Vecteur2...
Segment(const Segment...
Segment(int couleur, co...
setX(Vecteur2D x) : void
setY(Vecteur2D y) : void
toString() : string
Translation(const Vecteu...
public
Triangle
Classe
Polygone
Méthodes
~Triangle()
accepte(Visiteur*) : void
Afficher() : string
getAire() : double
getPerimetre() : double
Triangle(int couleur, con...
Triangle(Triangle& t)


Visiteur
Classe
Méthodes
visite(const Cercle*) : void
visite(const Forme*) : void
visite(const FormeComposee*) : void
visite(const Polygone*) : void
visite(const Segment*) : void
visite(const Triangle*) : void
visite(const Vecteur2D*) : void
private
private
DessinVisiteur
Classe
Visiteur
Méthodes
DessinVisiteur()
visite(const Cercle* circle) : void
visite(const Forme* f) : void
visite(const FormeComposee* , const FormeComposee* fc) : void
visite(const Polygone* q) : void
visite(const Segment* s) : void
SauvegardeVisiteur
Classe
Visiteur
Méthodes
SauvegardeVisiteur()
visite(const Cercle* circle) : void
visite(const FormeComposee*) : void
visite(const Polygone* q) : void
visite(const Segment* s) : void


Vecteur2D
Classe
Champs
_x : double
_y : double
Méthodes
~Vecteur2D()
accepte(Visiteur*) : void
Afficher() : string
getX() : double
getY() : double
Homothetie(const Vecteur2D& v, double rapport) : Vecteur2D&
operator-() : const Vecteur2D
operator-(const Vecteur2D& u) : const Vecteur2D
operator*(const double& a) : const Vecteur2D
operator+(const Vecteur2D& u) : const Vecteur2D
operator+=(const Vecteur2D& u) : Vecteur2D&
operator=(const Vecteur2D& u) : Vecteur2D
operator==(const Vecteur2D& u) : bool
Rotation(const Vecteur2D& v, double angle) : Vecteur2D&
setX(double x) : void
setY(double y) : void
toString() : string
Translation(const Vecteur2D& v) : Vecteur2D&
Vecteur2D(const char* s)
Vecteur2D(const double& x, const double& y)



6. Sauvegarde/Chargement

La sauvegarde et le chargement d'un fichier se font sur un fichier texte. Les différents composant d'une forme sont séparés par un « ; » et les formes elles-mêmes sont séparés par des « # ».

La sauvegarde s'effectue grâce au Design Pattern Visiteur et le chargement grâce à une chaine de responsabilité.