Architecture et fonctionnement du mini moteur géométrique

Projet WS3-OOP

16 novembre 2025

1 Introduction

Ce document décrit la base de code C++ développée pour simuler un ensemble simple de formes géométriques. L'objectif est de démontrer l'héritage et le polymorphisme en programmation orientée objet, conformément au diagramme UML proposé. On présente ici la hiérarchie des classes, les choix d'implémentation et un aperçu du flux d'exécution.

2 Vue d'ensemble

Le code est composé d'une classe abstraite `Forme` qui définit l'API commune pour toutes les formes (calcul d'aire, de périmètre et représentation textuelle). `Polygone` dérive de `Forme` pour représenter des formes à côtés multiples. Les classes concrètes `Cercle`, `Rectangle`, `Carre` et `Triangle` héritent respectivement de `Forme` ou `Polygone`. Les fichiers sont séparés en en-têtes (`include/`) et implémentations (`src/`).

3 Classe racine : Forme

```
class Forme {
public:
    virtual ~Forme() = default;

    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual std::string afficherInfo() const = 0;
};

class Polygone : public Forme {
public:
    ~Polygone() override = default;
    double area() const override = 0;
```

`Forme` est une classe abstraite pure contenant trois méthodes virtual pures. Chaque méthode est marquée `const` car elles n'altèrent pas l'état interne ; elles servent uniquement à interroger l'objet. `Polygone` réutilise la même interface, offrant un point d'ancrage pour les formes à plusieurs côtés (utile si l'on veut à l'avenir stocker des sommets communs).

4 Cercle

```
class Cercle : public Forme {
```

```
public:
    explicit Cercle(double rayon);

    double area() const override;
    double perimeter() const override;
    std::string afficherInfo() const override;

private:
    double rayon_;
};

#endif // CERCLE_HPP
```

```
#include "Cercle.hpp"

#include <sstream>

namespace {
constexpr double kPi = 3.14159265358979323846;
}

Cercle::Cercle(double rayon) : rayon_(rayon) {}

double Cercle::area() const {
    return kPi * rayon_ * rayon_;
}

double Cercle::perimeter() const {
    return 2.0 * kPi * rayon_;
}

std::string Cercle::afficherInfo() const {
    std::ostringstream oss;
    oss << "Cercle(rayon=" << rayon_ << ")";
    oss << "␣aire=" << area();
    oss << "␣perimetre=" << perimeter();
    return oss.str();
}
```

Le cercle stocke un seul attribut, son rayon. Les méthodes `area` et `perimeter` s'appuient sur les formules classiques $A = \pi r^2$ et $P = 2\pi r$. La constante `kPi` est définie dans un espace de noms anonyme, ce qui limite sa portée au fichier.

5 Polygones concrets

5.1 Rectangle

```
class Rectangle : public Polygone {
public:
    Rectangle(double largeur, double longueur);

    double area() const override;
    double perimeter() const override;
    std::string afficherInfo() const override;

protected:
```

```
    double largeur_;
    double longueur_;
};

#endif // RECTANGLE_HPP
```

```
#include "Rectangle.hpp"

#include <sstream>

Rectangle::Rectangle(double largeur, double longueur)
    : largeur_(largeur), longueur_(longueur) {}

double Rectangle::area() const {
    return largeur_ * longueur_;
}

double Rectangle::perimeter() const {
    return 2.0 * (largeur_ + longueur_);
}

std::string Rectangle::afficherInfo() const {
    std::ostringstream oss;
    oss << "Rectangle(largeur=" << largeur_ << ",␣longueur=" <<
       longueur_ << ")";
    oss << "␣aire=" << area();
    oss << "␣perimetre=" << perimeter();
    return oss.str();
}
```

`Rectangle` gère deux attributs (largeur et longueur). Il dérive de `Polygone` pour marquer cette forme comme un polygone. Les méthodes redéfinies multiplient ou additionnent ces dimensions suivant les formules classiques.

5.2 Carré

```

class Carre : public Polygone {
public:
    explicit Carre(double cote);

    double area() const override;
    double perimeter() const override;
    std::string afficherInfo() const override;

private:
    double cote_;
};

#endif // CARRE_HPP
```

```
#include "Carre.hpp"

#include <sstream>

Carre::Carre(double cote) : cote_(cote) {}
```

```cpp
double Carre::area() const {
    return cote_ * cote_;
}

double Carre::perimeter() const {
    return 4.0 * cote_;
}

std::string Carre::afficherInfo() const {
    std::ostringstream oss;
    oss << "Carre(cote=" << cote_ << ")";
    oss << "␣aire=" << area();
    oss << "␣perimetre=" << perimeter();
    return oss.str();
}
```

`Carre` est une spécialisation de polygone avec un seul paramètre `cote`. Les méthodes utilisent les formules $A = c^2$ et $P = 4c$.

5.3 Triangle

```cpp
class Triangle : public Polygone {
public:
    Triangle(double a, double b, double c);

    double area() const override;
    double perimeter() const override;
    std::string afficherInfo() const override;

private:
    double coteA_;
    double coteB_;
    double coteC_;
};
```

```cpp
#include "Triangle.hpp"

#include <cmath>
#include <sstream>

Triangle::Triangle(double a, double b, double c)
    : coteA_(a), coteB_(b), coteC_(c) {}

double Triangle::area() const {
    const double s = perimeter() / 2.0;
    return std::sqrt(s * (s - coteA_) * (s - coteB_) * (s - coteC_));
}

double Triangle::perimeter() const {
    return coteA_ + coteB_ + coteC_;
}

std::string Triangle::afficherInfo() const {
    std::ostringstream oss;
    oss << "Triangle(a=" << coteA_ << ",␣b=" << coteB_ << ",␣c=" <<
       coteC_ << ")";
```

```
    oss << "␣aire=" << area();
    oss << "␣perimetre=" << perimeter();
    return oss.str();
}
```

Le triangle stocke trois longueurs de côté. L'aire est calculée selon la formule de Héron : $A = \sqrt{s(s-a)(s-b)(s-c)}$ avec s demi-périmètre. Cette approche évite de manipuler explicitement les coordonnées des sommets.

6 Programme principal

```
#include <iostream>
#include <memory>
#include <vector>

#include "Carre.hpp"
#include "Cercle.hpp"
#include "Rectangle.hpp"
#include "Triangle.hpp"

int main() {
    std::vector<std::unique_ptr<Forme>> formes;
    formes.push_back(std::make_unique<Cercle>(5.0));
    formes.push_back(std::make_unique<Rectangle>(4.0, 6.0));
    formes.push_back(std::make_unique<Carre>(3.0));
    formes.push_back(std::make_unique<Triangle>(3.0, 4.0, 5.0));

    for (const auto &forme : formes) {
        std::cout << forme->afficherInfo() << '\n';
    }
    return 0;
}
```

Le fichier `main.cpp` illustre l'utilisation de la hiérarchie. Des `unique extunderscore ptr<Forme>` sont stockés dans un vecteur pour démontrer le polymorphisme ; l'appel à `afficherInfo` déclenche automatiquement la version correcte pour chaque instance.

7 Compilation

Un `Makefile` minimal compile la bibliothèque et l'exécutable de démo :

```
CXX := g++
CXXFLAGS := -std=c++17 -Wall -Wextra -pedantic -Iinclude
SRCS := $(wildcard src/*.cpp)
OBJS := $(SRCS:.cpp=.o)
TARGET := demo

all: $(TARGET)

$(TARGET): $(OBJS)
        $(CXX) $(CXXFLAGS) $^ -o $@

clean:
        rm -f $(OBJS) $(TARGET)

.PHONY: all clean
```

`make` produit l'exécutable `demo`, et `./demo` affiche les informations sur les formes créées.

8 Extensions possibles

— Ajouter des validations (rayon strictement positif, inégalité triangulaire, etc.).
— Enrichir `Polygone` avec une liste de sommets pour obtenir des transformations géométriques.
— Intégrer une interface graphique pour visualiser les formes.

9 Conclusion

La structure actuelle démontre clairement l'héritage et le polymorphisme en C++. Chaque classe encapsule les calculs propres à sa forme, et l'interface `Forme` garantit une API cohérente. Le projet constitue une base solide pour des fonctionnalités graphiques futures ou des validations plus poussées.