

**CER U.E UN PROBLEME
D'HERITAGE
"your name "**

Rôle	Nom Prénom
Animateur	nouara
Scribe	abdelghani
Gestionnaire	mehdi
Secrétaire	sarah

1. Mots clés:

- Application graphique
- Objets géométriques
- Réutilisabilité du code

2. Mots à définir:

- Arbre d'héritage

3. Analyse du contexte:

Jean-Eude, étudiant en informatique, doit concevoir en C++ une application pour créer et gérer des formes géométriques afin de prouver ses compétences en programmation orientée objet lors d'un entretien chez AutoDesk, en veillant à rendre certaines parties de son code réutilisables.

4. Définition de la problématique:

Comment concevoir une hiérarchie de classes en C++ permettant de modéliser des formes géométriques variées tout en respectant les principes d'héritage, de substitution de Liskov ?

5. Contraintes:

- Respecter le principe de Liskov (LSP)
- Ne pas faire d'interface graphique

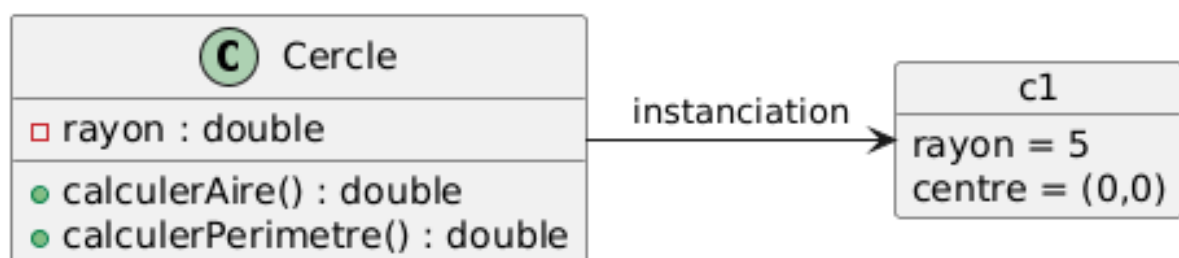
6. Plan d'actions :

6.1 Découvrir la notion d'héritage en POO

a. Qualification : **Abouti** / **Difficile à concrétiser** / **Non abouti**

b. Démonstration :

6.1 – Étudier la notion d'héritage en Programmation Orientée Objet (POO)



(ex. un rectangle avec "Classe" → flèche vers un rectangle "Objet")

L'héritage est l'un des quatre piliers de la programmation orientée objet, aux côtés de :

- l'encapsulation,
- l'abstraction,
- le polymorphisme.

Dans ce sujet, il est central car Jean-Eude doit organiser plusieurs formes géométriques partageant des comportements communs (position, aire, périmètre), tout en garantissant un code propre, réutilisable et bien structuré.

6.1.1 — Rappel : classes et objets

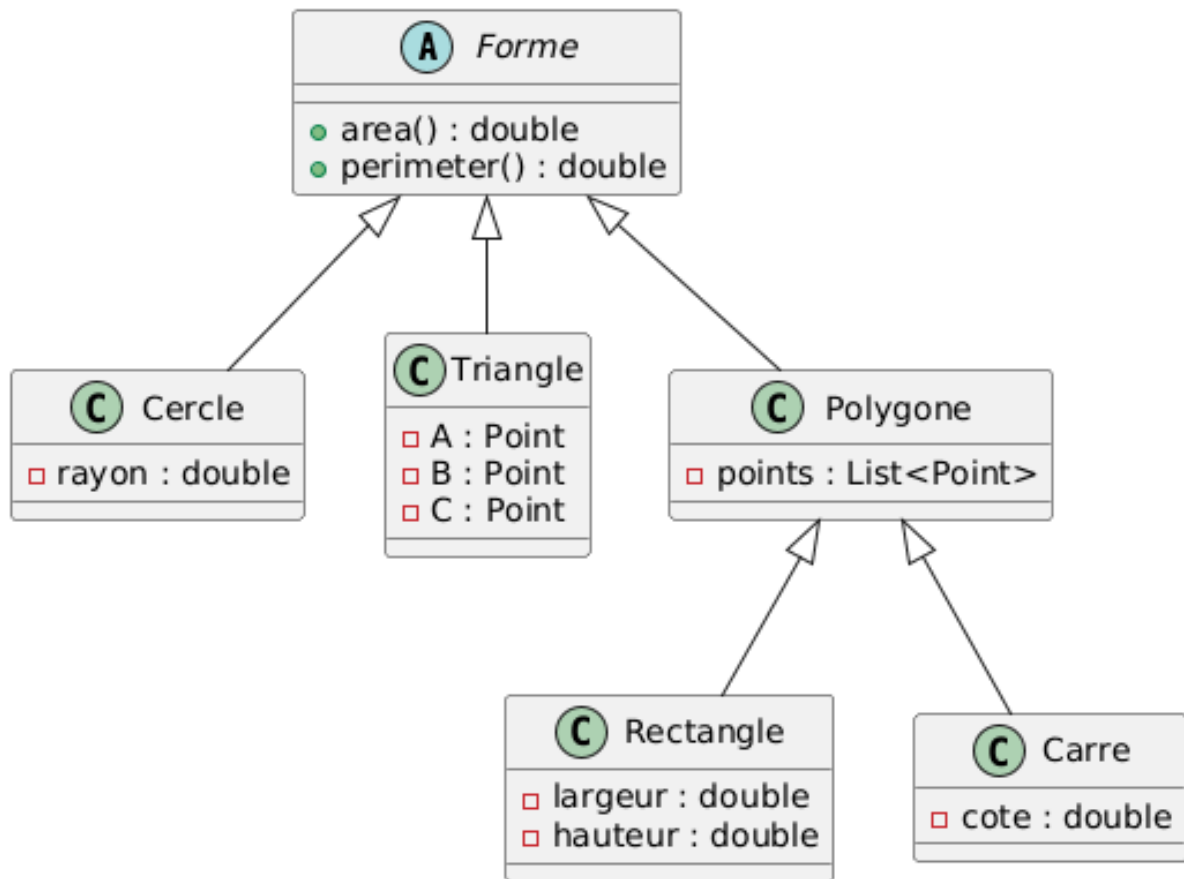
Avant d'aborder l'héritage, il faut comprendre deux notions :

- Une **classe** est un modèle qui définit :
 - des **attributs** (état),
 - des **méthodes** (actions).
- Un **objet** est une instance concrète de cette classe.

Exemple :

- La classe Cercle décrit ce qu'est un cercle en général.
- L'objet c1 est un cercle précis : rayon 5, centre (0,0).

6.1.2 — Définition de l'héritage



L'héritage permet à une **classe fille** d'hériter des attributs et méthodes d'une **classe mère**.

La classe fille :

- réutilise ce qui existe déjà,
- ajoute ce qui lui est propre,
- peut redéfinir certains comportements (override).

On part donc du **général** (classe mère) vers le **spécifique** (classe fille).

Exemples dans le sujet :

- Classe mère : **Forme**
- Classes filles : **Cercle**, **Triangle**, **Rectangle**, **Carre**

6.1.3 — Relation “est-un” (is-a) et choix de l’héritage

On utilise l’héritage lorsqu’il existe une relation naturelle :

X est un Y

→ *Un Cercle est une Forme*

Exemples valides dans le sujet :

- Un **Cercle** est une **Forme**
- Un **Triangle** est une **Forme**
- Un **Rectangle** est une **Forme**
- Un **Carré** est une **Forme**

 Mais attention :

“Un Carré est un Rectangle” **en mathématiques** ≠ **en POO**.

Cela violerait le principe de Liskov (voir section 6.3).

6.1.4 — Pourquoi utiliser l'héritage ?

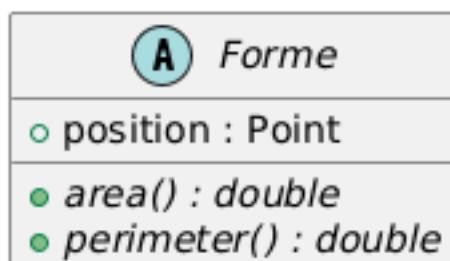
a) Réutilisation du code

Toutes les formes partagent :

- une position,
- une aire,
- un périmètre.

Ces éléments doivent être définis une seule fois dans *Forme*.

b) Factorisation des comportements communs



La factorisation permet :

- d'éviter les duplications,
- de centraliser les comportements génériques,
- de rendre le code plus maintenable.

c) Spécialisation : du général vers le particulier

Grâce à l'héritage, on crée une hiérarchie cohérente :

- *Forme* (général)

- Cercle (a un rayon)
- Triangle (a trois sommets)
- Rectangle (a longueur / largeur)
- Carre (a un côté identique sur les quatre côtés)

d) Cohérence conceptuelle et lisibilité du code

Une hiérarchie bien pensée rend le code :

- logique,
- facile à maintenir,
- proche du domaine métier (ici : la géométrie).

C'est un atout majeur pour un stage chez AutoDesk.

6.1.5 — Héritage et encapsulation

L'héritage **n'annule pas** l'encapsulation :

- les attributs restent protégés (private ou protected),
- les classes filles utilisent les méthodes publiques de la classe mère,
- la cohérence interne est préservée.

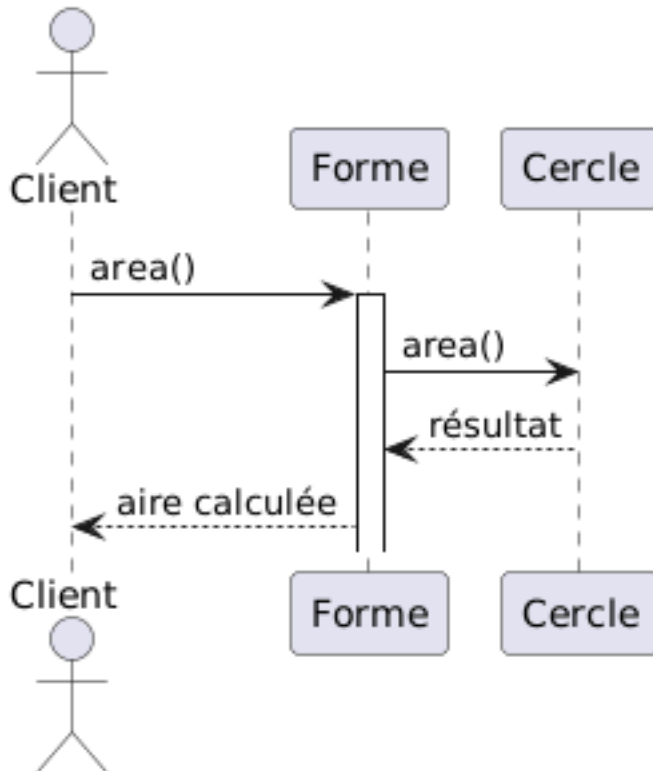
Exemple :

Forme peut contenir :

- un attribut position (protégé),
- une méthode pour déplacer la forme.

Les classes filles n'ont qu'à spécialiser les calculs d'aire et de périmètre.

6.1.6 — Héritage et polymorphisme (transition vers 6.4)



Grâce à l'héritage, on peut :

- mettre des Cercles, Triangles et Rectangles dans une même liste de Forme,
- appeler `area()` sur chacun d'eux,
- obtenir automatiquement le calcul spécifique à chaque forme.

Ce mécanisme repose sur :

- les **méthodes virtuelles**,
- le **type déclaré** (`Forme*`),
- le **type réel** (`Cercle`, `Triangle...`).

6.1.7 — Application directe au sujet

Jean-Eude doit gérer plusieurs formes et repère des comportements communs.

L'héritage lui permet de :

1. Créer une classe abstraite *Forme*.
2. Décliner : *Cercle*, *Triangle*, *Rectangle*, *Carre*.
3. Réutiliser la base commune pour d'autres fonctionnalités (interface graphique, sélection, etc.).

Il passe d'un groupe de formes indépendantes à une **architecture propre et évolutive**.

6.1.8 — Limites et pièges de l'héritage

- Carré **ne doit pas** hériter de *Rectangle* → viole Liskov.
- Modifications dans *Forme* peuvent casser les classes filles si la hiérarchie est mal pensée.
- L'héritage doit rester **logique** et **stable**.

6.2 Mise en œuvre de l'héritage en C++

a. Qualification :

b. Démonstration :

6.2 – Mise en œuvre de l'héritage en C++

Dans cette section, on passe de la théorie à la pratique : comment écrire l'héritage en C++ sur le plan syntaxique, comment choisir le bon mode d'héritage (public, protégé, privé) et quelles sont les conséquences sur l'accès aux membres.

6.2.1 — Déclarer une classe dérivée en C++

En C++, une classe dérivée se déclare en utilisant le symbole : après le nom de la classe, suivi d'un **modificateur d'accès** et du nom de la classe de base :

```
class ClasseFille : modeHeritage ClasseMere {  
    // membres spécifiques  
};
```

```
// Exemple de classe base et classe dérivée  
class Forme {  
public:  
    void afficherPosition(); // méthode commune  
};
```

```
class Cercle : public Forme {  
public:  
    double rayon;  
};
```

Selon W3Schools, « *l'héritage permet à une classe de réutiliser les attributs et méthodes d'une autre classe, afin d'éviter la duplication de code et d'améliorer la clarté* » 【

6.2.2 — Modes d'héritage : public, protégé et privé

C++ offre trois modes d'héritage qui influencent l'accessibilité des membres de la classe de base dans la classe dérivée :

Héritage public : les membres publics restent publics et les membres protégés restent protégés.

Héritage protégé : les membres publics et protégés de la base deviennent protégés dans la dérivée.

Héritage privé : les membres publics et protégés de la base deviennent privés dans la dérivée.

Les membres privés de la base sont toujours inaccessibles dans la classe dérivée, quel que soit le mode d'héritage.

6.2.3 — Exemple complet : héritage public

On crée une classe `Forme` avec un attribut et une méthode. On dérive ensuite `Cercle` en mode public :

```
#include <iostream>
using namespace std;

class Forme {
public:
    string nom = "Forme";
    void afficherNom() {
        cout << "Nom : " << nom << endl;
    }
};

class Cercle : public Forme {
public:
    double rayon;
    Cercle(double r) : rayon(r) {
        nom = "Cercle"; // on peut modifier l'attribut hérité
    }
};

int main() {
    Cercle c(5.0);
    c.afficherNom();      // Affiche "Nom : Cercle"
    cout << "Rayon : " << c.rayon << endl;
    return 0;
}
```

Dans cet exemple, on voit que la méthode de la classe `Forme` est récupérée et peut être utilisée par un objet de type `Cercle`.

6.2.4 — Mode d'héritage et accès aux membres

Le tableau ci-dessous récapitule la visibilité finale de chaque catégorie de membres selon le mode d'héritage. Les membres privés de la classe de base sont toujours inaccessibles dans la dérivée.

Mode d'héritage	Membres publics de la base	Membres protégés de la base	Membres privés de la base
public	public	protégé	inaccessible
protégé	protégé	protégé	inaccessible
privé	privé	privé	inaccessible

6.2.5 — Conseil de bonne pratique

Dans le contexte d'un projet comme celui de Jean-Eude, l'héritage public est généralement préférable : les méthodes et attributs communs restent accessibles, ce qui facilite la réutilisation et le polymorphisme.

6.3 Étudier le principe de substitution de Liskov

a. Qualification :

b. Démonstration :

6.3 – Étudier le principe de substitution de Liskov (LSP)

Le principe de substitution de Liskov (LSP) est l'un des principes SOLID. Formulé par Barbara Liskov et Jeannette Wing, il stipule que **les classes dérivées doivent pouvoir remplacer leurs classes de base sans altérer la correction du programme**. Autrement dit, si S est un sous-type de T, tout objet de type T doit pouvoir être remplacé par un objet de type S sans que le programme ne change de comportement.

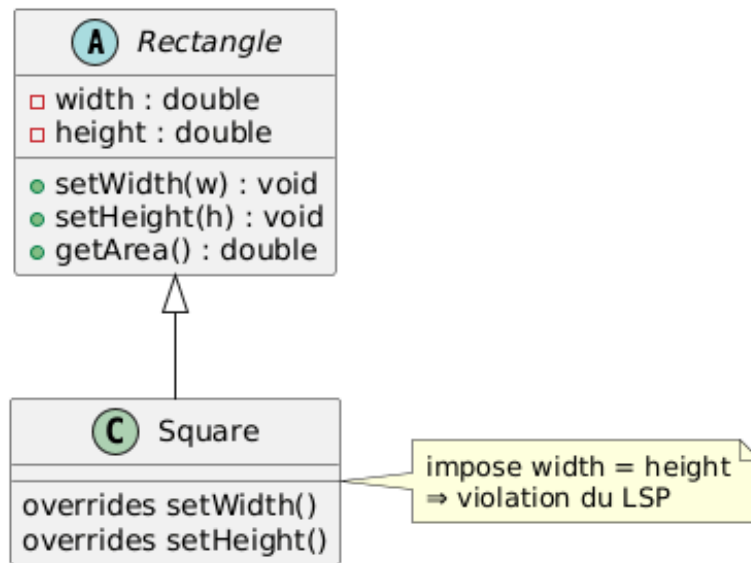
6.3.1 — Définition formelle et conditions

Formellement, Liskov et Wing énoncent que **si un objet x satisfait une propriété ϕ en tant que type T, alors tout objet y de type S (sous-type de T) doit aussi satisfaire la propriété ϕ** . Pour que la substituabilité soit préservée, le sous-type doit respecter certains engagements :

- Ne pas renforcer les préconditions.
- Ne pas affaiblir les postconditions.
- Ne pas briser les invariants.
- Respecter la « history rule », c'est-à-dire ne pas introduire des changements d'état inattendus.

6.3.2 — Exemple de violation : carré dérivé de rectangle

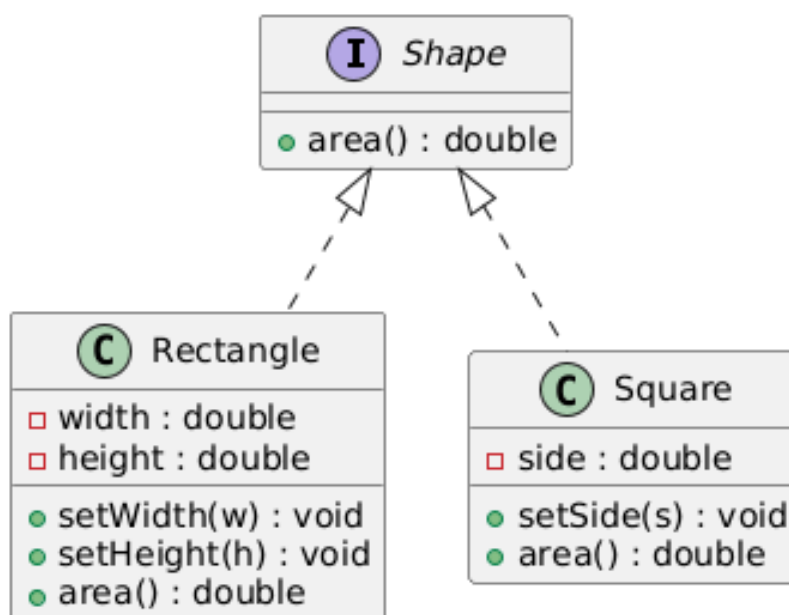
Considérons une classe Rectangle avec deux méthodes virtuelles `setWidth()` et `setHeight()`, et une classe Square qui en hérite. Si Square impose que la largeur et la hauteur restent égales à chaque modification, un programme qui s'attend à pouvoir modifier largeur et hauteur indépendamment via une référence à Rectangle se comportera mal. Cette situation viole donc le LSP.



Ce diagramme montre qu'en héritant et en redéfinissant `setWidth()` et `setHeight()`, `Square` impose une contrainte (largeur = hauteur) qui n'existe pas dans `Rectangle`. Si une fonction client appelle `setWidth()` puis `setHeight()` sur un objet supposé être un `Rectangle`, cette contrainte supplémentaire compromet la substitution attendue.

6.3.3 — Solutions pour respecter le LSP

Pour remédier à cette violation, il est préférable de **ne pas dériver `Square` de `Rectangle`**. On introduit plutôt une interface ou classe abstraite `Shape` (ou `Forme`) qui définit l'opération commune `area()`. Les classes `Rectangle` et `Square` implémentent indépendamment cette interface, évitant ainsi toute contrainte supplémentaire.



6.4 S'initier au polymorphisme

a. Qualification :

b. Démonstration :

6.4 – S'initier au polymorphisme

Le polymorphisme est un pilier de la programmation orientée objet. Littéralement, le mot signifie « prendre plusieurs formes ». En C++, il permet à une même interface (par exemple une méthode `area()`) d'être utilisée pour différentes classes dérivées, tout en appelant la bonne implémentation selon l'objet réel. Les fonctions virtuelles, déclarées dans la classe de base et redéfinies dans les classes dérivées, sont au cœur du polymorphisme dynamique .

6.4.1 — Principe du polymorphisme dynamique

Une **fonction virtuelle** est une fonction membre déclarée avec le mot-clé `virtual` dans la classe de base et redéfinie dans la classe dérivée . C++ effectue alors **un appel tardif** : le choix de la fonction exécutée se fait à l'exécution, en fonction du type réel de l'objet pointé, et non du type statique du pointeur .

Selon le C++ FAQ, lorsqu'on a un pointeur vers un objet, il existe deux types : **le type statique** (le type du pointeur, connu à la compilation) et **le type dynamique** (le type réel de l'objet, connu à l'exécution) . Le compilateur vérifie que l'appel est légal en se basant sur le type statique, mais l'implémentation appelée est choisie selon le type dynamique grâce aux fonctions virtuelles .

Exemple de base

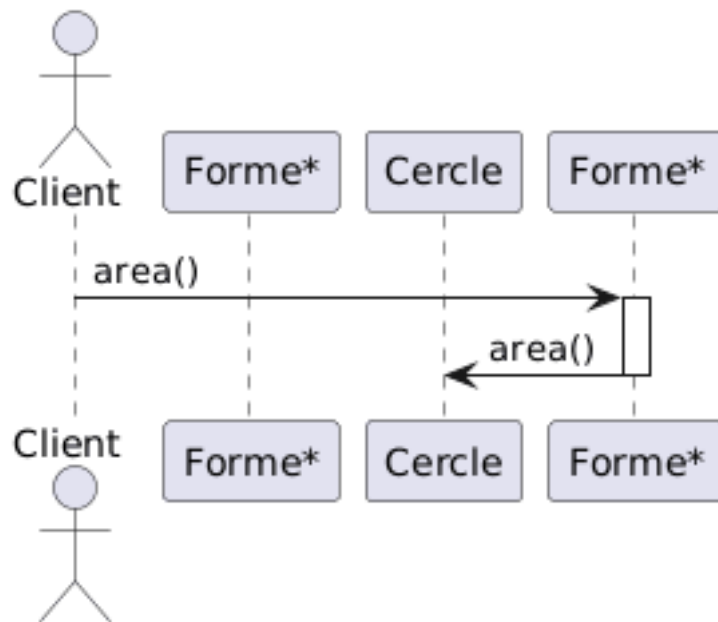
Considérons une classe abstraite `Forme` avec une méthode virtuelle `area()` redéfinie dans ses sous-classes (`Cercle`, `Triangle`, etc.). Un pointeur de type `Forme*` peut alors pointer vers n'importe quelle forme concrète ; appeler `area()` exécutera la version appropriée selon l'objet pointé.

```
class Forme {  
  
public:  
  
    virtual double area() const = 0; // méthode virtuelle pure  
  
    virtual ~Forme() {}              // destructeur virtuel  
  
};  
  
class Cercle : public Forme {  
  
public:  
  
    double rayon;  
  
    Cercle(double r) : rayon(r) {}  
  
    double area() const override { return 3.14159 * rayon * rayon; }  
  
};  
  
class Rectangle : public Forme {  
  
public:  
  
    double largeur, hauteur;  
  
    Rectangle(double l, double h) : largeur(l), hauteur(h) {}  
  
    double area() const override { return largeur * hauteur; }  
  
};
```

Cette hiérarchie permet de manipuler des objets différents via un même type de base.

6.4.2 — Illustration de l'appel dynamique

Le comportement polymorphe peut être représenté par un diagramme de séquence : un client appelle `area()` sur un pointeur de type `Forme*` qui pointe vers un `Cercle`. L'appel est résolu dynamiquement vers `Cercle::area()`.

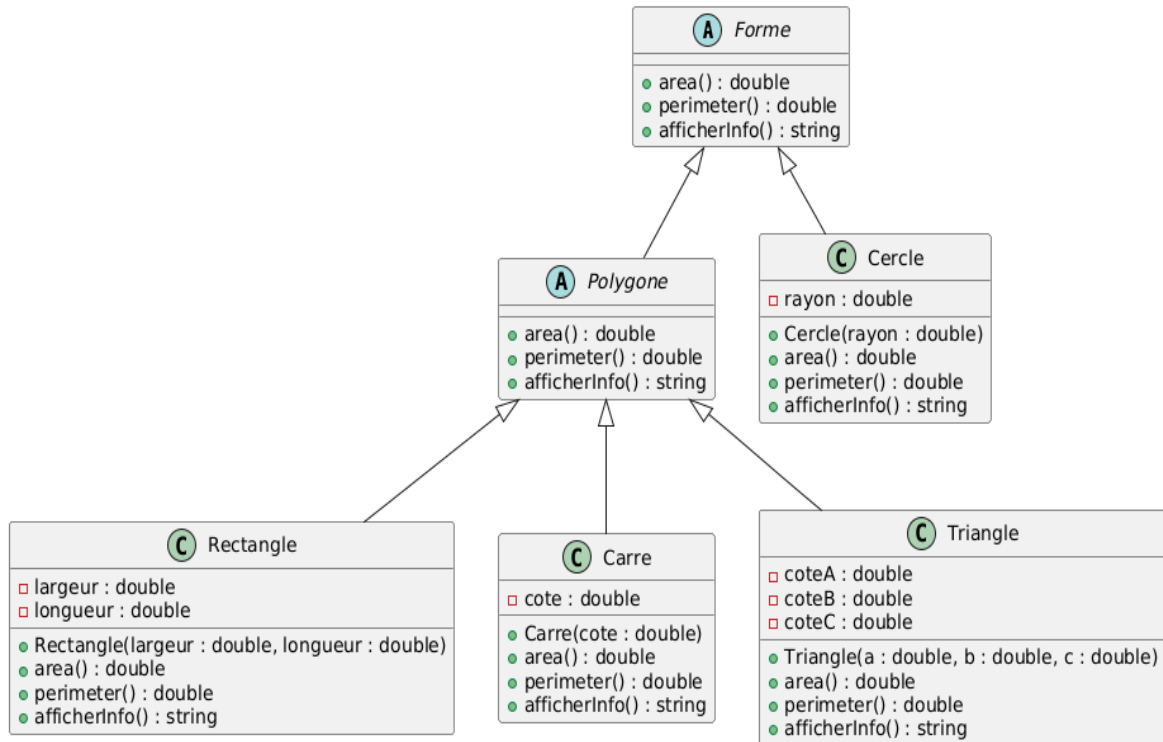


Ce diagramme montre que l'appel `area()` passe du type de base vers la classe dérivée réelle, illustrant la liaison tardive (dynamic binding).

6.5 Réaliser un diagramme de classes

a. Qualification :

b. Démonstration :



6.6 Faire le code en utilisant les principes de l'héritage

a. Qualification : **Abouti**

b. Démonstration :

<https://github.com/FrigaaAbdou/CER-3>