CAS 1 (AVEC UNE AGREGRATION)

```
// Figure.h
#pragma once
#include <vector>
using namespace std;

class Point;

class Figure
{
public:
	vector<Point*> lesPoints;
	virtual double getPerimetre()=0;
};
```

Ici, je déclare que la classe **Point** existe. A faire si on ne gère que des pointeurs sur cette classe (sinon, il faut inclure le fichier .h)

Figure est un classe abstraite car elle possède au moins une méthode virtuelle pure (ici, c'est **getPerimetre**). En UML, la classe et les méthodes virtuelles pures sont en italique

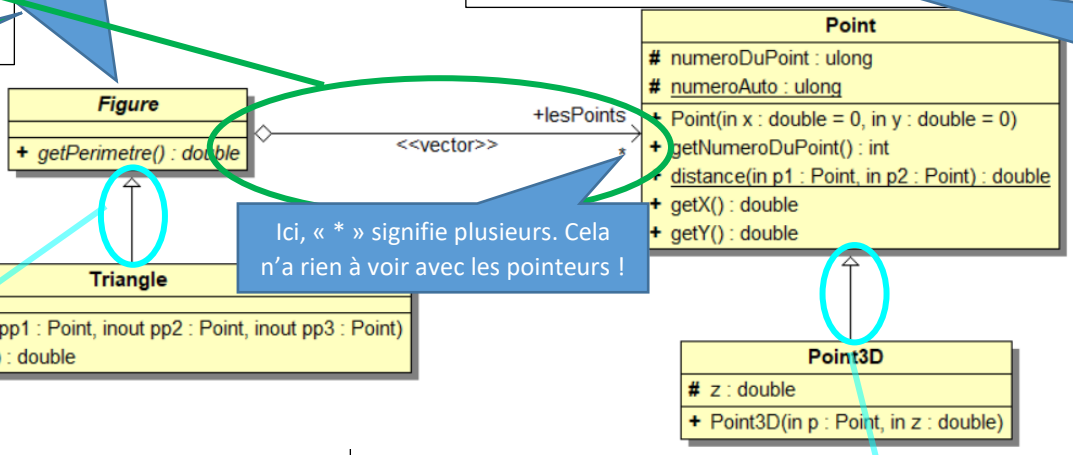
C'est une agrégation car les durées de vie des points et de la figure ne sont pas liées. On peut détruire la figure en laissant les points. Ils sont créés séparément et associés dans le constructeur des figures concrètes.

```
#pragma once
class Point
{
protected :
	double x, y;
	unsigned long numeroDuPoint;
	static unsigned long numeroAuto;

public:
	Point(double x=0 , double y=0);
	int getNumeroDuPoint() { return numeroDuPoint; }
	static double distance(const Point &p1, const Point & p2);
	double getX() const { return x; }
	double getY() const { return y; }
};
```

Les attributs ou méthodes static sont des attributs de classe (soulignés en UML)

Le mot clé « const » à la fin de la méthode signifie que cette méthode ne modifie pas l'objet courant



Ici, « * » signifie plusieurs. Cela n'a rien à voir avec les pointeurs !

```
// Triangle.h
#pragma once
#include "Point.h"
#include "Figure.h"

class Triangle :public Figure
{
	public:
		Triangle(Point *pp1, Point  *pp2 ,  Point *pp3);
		double getPerimetre();
};
```

La méthode getPerimetre() a dû être redéfinie pour que cette classe devienne « concrète »

```
// Point3D.h
#pragma once
#include "Point.h"
class Point3D :      public Point
{
protected:
	double z;

public:
	Point3D(Point p , double z);
};
```

```cpp
//Point.cpp
#include <math.h>
#include "Point.h"

unsigned long Point::numeroAuto = 0;

Point::Point(double x , double y )
{
	numeroAuto++;
	this->numeroDuPoint = numeroAuto;
	this->x = x;
	this->y = y;
}

double Point::distance(const Point &p1, const Point & p2)
{
	double dist = (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y);
	return sqrt(dist);
}
```

Les attributs « static » doivent obligatoirement être initialisés !

Quand le paramètre et l'attribut portent le même nom, on accède à l'attribut par **this->nomAttribut**

On ne réécrit pas le mot clé « static » ici !

« p1 » et « p2 » sont passés par référence pour éviter de dupliquer inutilement des éléments en mémoire. Le mot clé « const » devant indique qu'on ne pourra pas modifier les paramètres dans la méthode.

```cpp
//Point3D.cpp
#include "Point3D.h"

Point3D::Point3D(Point p, double z) : Point(p.getX() , p.getY())
{
	this->z = z;
}
```

Derrière le symbole « : », on appelle le constructeur de la classe « père » en lui transmettant des informations qui dépendent des paramètres du nouveau cconstructeur

```cpp
//Triangle.cpp
#include "Triangle.h"

Triangle::Triangle(Point  *pp1,  Point  * pp2,  Point * pp3)
{
	lesPoints.push_back(pp1);
	lesPoints.push_back(pp2);
	lesPoints.push_back(pp3);
}

double Triangle::getPerimetre()
{
	double dist = Point::distance(*lesPoints[0], *lesPoints[1]);
	dist += Point::distance(*lesPoints[1], *lesPoints[2]);
	dist+=Point::distance(*lesPoints[0], *lesPoints[2]);
	return dist;
}
```

On ajoute les pointeurs des points au vecteur, mais les points ne sont pas créés ici (ils ont été créés dans la fonction principale). C'est donc bien une agrégation, pas une composition

On appelle une méthode « static » (méthode de classe) en écrivant « NOM_DE_CLASSE ::methodeStatic(…) »

```cpp
int main()    // Fonction principale
{
	Point p1(2, 4);
	Point *p2 = new Point(6, 7);

	Triangle t1(&p1, p2, &Point(8, 3));
	cout << "Perimetre = " << t1.getPerimetre() << endl;

	cout << "Le numéro auto de p2 est " << p2->getNumeroDuPoint() << endl;

	delete p2;

	_getch();     return 0;     // fin du programme. Le code 0 est envoyé
}
```

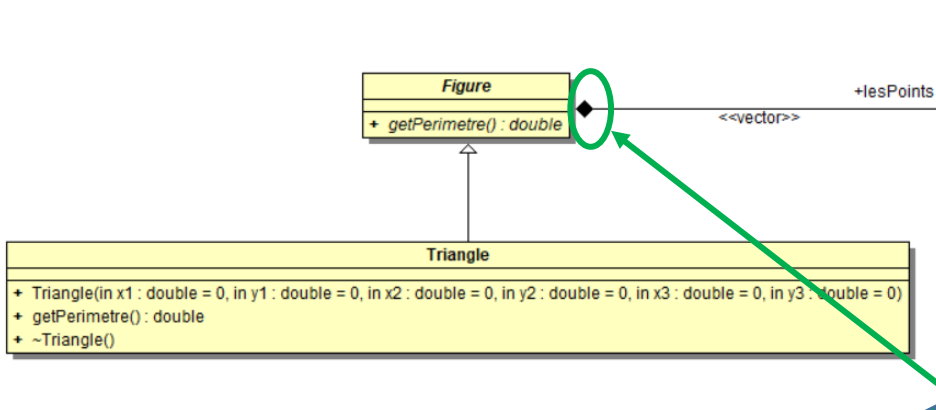
« p1 » est créé en allocation de mémoire statique
« p2 » est créé en allocation de mémoire dynamique (à privilégier)

p2 est effacé de la mémoire

p1 est effacé de la mémoire ainsi que le triangle « t1 »

CAS 2 (AVEC UNE COMPOSITION)

SEULES LES DIFFERENCES AVEC LE CAS 1 SONT REPRESENTEES.




```cpp
int main()    // Fonction principale
{

        Triangle t1(2,4,6,7,8,3);

        cout << "Perimetre = "
             << t1.getPerimetre() << endl;

        _getch();
        return 0;
}
```

Le triangle avec ses points est créé ici !

```cpp
// Triangle.h
#pragma once
#include "Point.h"
#include "Figure.h"

class Point;

class Triangle :public Figure
{
       public:
              Triangle(double x1=0, double y1=0, double x2=0,
                     double y2=0, double x3=0, double y3=0);
              double getPerimetre();
              ~Triangle();
};
```

C'est une composition applicative (la plus fréquente). Les éléments associés (ici, les points) sont créés lors de la construction du triangle.

```cpp
//Triangle.cpp
#include "Triangle.h"


Triangle::Triangle(double x1 , double y1, double x2, double y2 ,
double x3 , double y3 )
{
       lesPoints.push_back(new Point(x1, y1));
       lesPoints.push_back(new Point(x2, y2));
       lesPoints.push_back(new Point(x3, y3));
}

Triangle::~Triangle()
{
       for (unsigned i = 0; i < 3; i++)
              delete lesPoints[i];
}

double Triangle::getPerimetre()
{ …
```

Destructeur obligatoire quand des allocations mémoires sont faites dans la classe afin de libérer la mémoire.

COMMENT CREER LES OBJETS ?

```
int main()    // Fonction principale
{
       unsigned i, n;
```

1 seul objet

```
       Triangle t1;
       cout << t1.getPerimetre() << endl; // affiche 0

       Triangle t2(1,1,2,4,3,2);
       cout << t2.getPerimetre() << endl; // affiche 7,63

       Triangle *pt3;
       pt3 = new Triangle(2, 7, 5.4, 2.98);
       cout << pt3->getPerimetre() << endl; // affiche 18.71
```

Allocation de mémoire statique et paramètres par défaut

Allocation de mémoire statique

Définition d'un pointeur nommé pt3

Allocation dynamique de mémoire

Tableau de plusieurs objets tous initialisés par défaut

```
       Triangle tab1[4];
       for (i = 0; i < 4; i++)
              cout << tab1[i].getPerimetre() << "   "; // affiche 0 0 0 0
       cout << endl;

       Triangle *tab2;
       cout << "Combien de triangle veux-tu ? ";
       cin >> n;
       tab2 = new Triangle[n];
       for (i = 0; i < n; i++)
              cout << tab2[i].getPerimetre() << "   "; // affiche 0 0 0 0....0 (n fois)
       cout << endl;
```

Tableau de plusieurs objets pouvant être initialisés différemment.

```
       Triangle *tab3[4];
       tab3[0] = new Triangle(1, 5, 3, 1, 4, 2);
       tab3[1] = new Triangle(1, 8.4, 6, 2, 4, 7);
       tab3[2] = new Triangle(1, 5, 3.35, 1, 4, 9);
       tab3[3] = new Triangle(1, 4.5, 3, 1, 3, 2);
       for (i = 0; i < 4; i++)
              cout << tab3[i]->getPerimetre() << "   "; // affiche 10.13  16.82 17.66 8.23
       cout << endl;

       vector<Triangle * > tab4;
       tab4.push_back(new Triangle(1.2, 6.8, 2.4, 9.3));
       tab4.push_back(pt3);
       tab4.push_back(&t1);

       for (i = 0; i < tab4.size(); i++)
              cout << tab4[i]->getPerimetre() << "   "; // affiche 19.28 18.71 0
```

Les vecteurs permettent d'ajouter ou de supprimer des éléments facilement. **Utiliser de préférence un vecteur de pointeurs.** Ainsi, si vous modifiez l'objet en dehors du vecteur, sa modification sera effective aussi dans le vecteur.

Libération mémoire

```
      // Libération mémoire
       tab4.clear();

       for (i = 0; i < 4; i++)
              delete tab3[i];

       delete[] tab2;
```

```
       _getch();     // attente d'appui sur une touche
       return 0;     // fin du programme. Le code 0 est envoyé
}
```