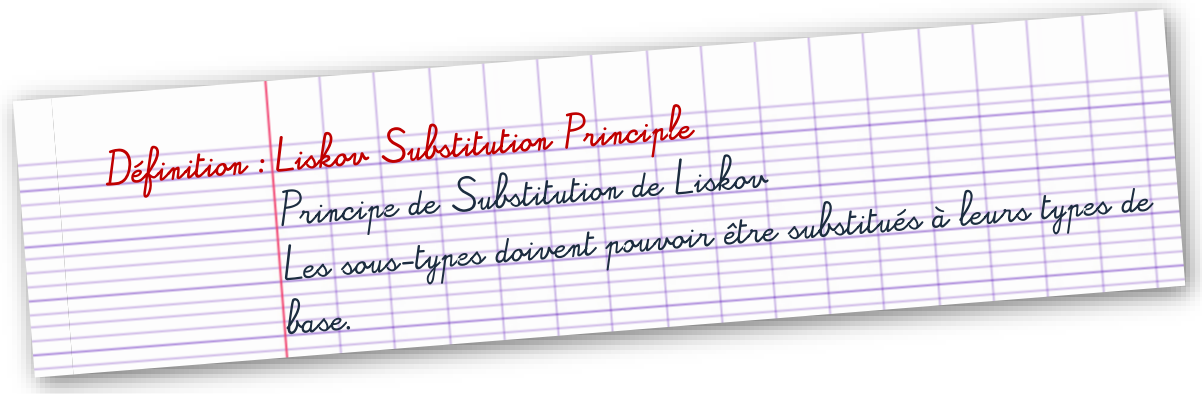
# TD 11 – Premiers Principes de Conceptions Orientée Objet

## Cours 1 Liskov Substitution Principle

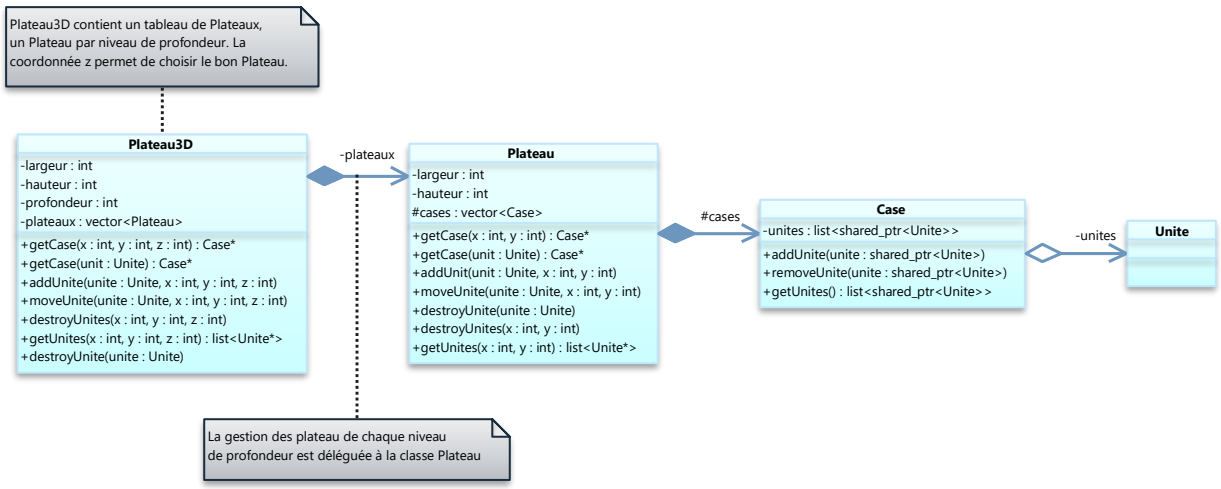
### 1.1 Définition




- Le but de ce principe est de faire une bonne utilisation de l'héritage. A la relation ***EST-UN*** qui est souvent associée à l'héritage, ce principe ajoute la notion ***SE COMPORTE COMME***.
- Dans le cas de l'exercice 1, le **`Plateau3D`** ne peut pas être substitué au **`Plateau`** car aucune des méthodes de **`Plateau`** ne fonctionne correctement dans un contexte 3D. Le principe de substitution de Liskov est ici enfreint.
- Ce principe est étroitement lié à la programmation par contrat, selon laquelle une classe ou fonction doit répondre à un certain nombre de règles, fixées par son développeur.
  - Il y a trois types de règles :
    (1) Les ***préconditions*** : liste des conditions à respecter par le client du composant logiciel avant son utilisation.
    (2) Les ***postconditions*** : liste des conditions qui sont garanties par le composant logiciel après son utilisation.
    (3) Les ***invariants*** : il s'agit des conditions qui sont toujours respectées, certifiant l'état interne du composant logiciel.

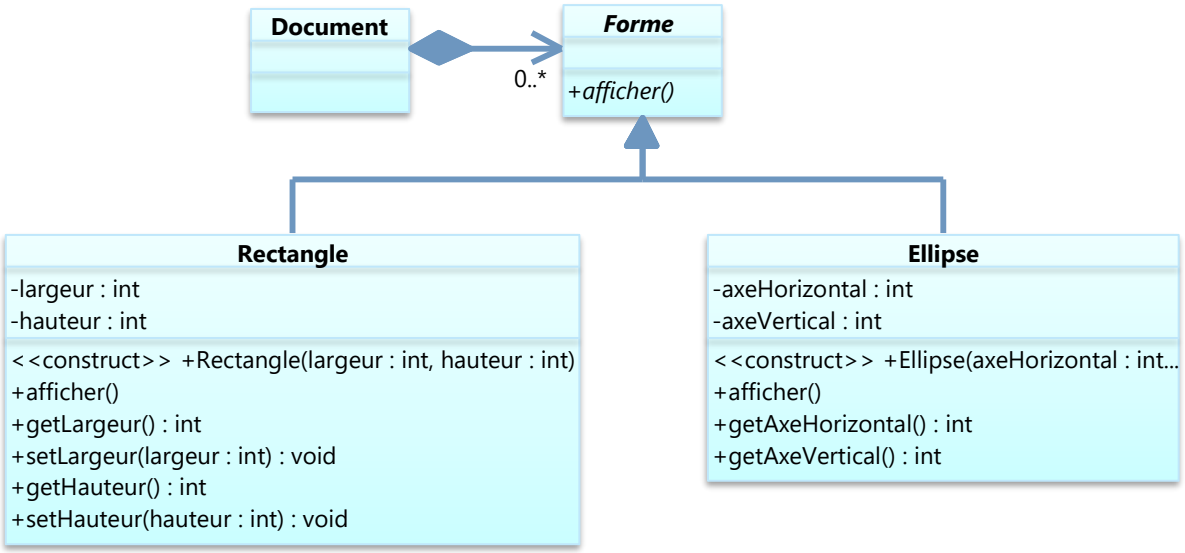
## Exercice 2 BAKWar 3D en mieux

- Appliquez ce principe au passage à la 3D de ***BAKWar***.
  - Rappelez-vous de réutiliser le maximum de code sans le modifier.




## Exercice 3 Le logiciel BAKAD

On ne présente plus le logiciel ***BAKAD*** qui permet de tracer toutes sortes de rectangles et d'ellipses à l'écran ! Pour la version 2, l'équipe de développement doit ajouter la possibilité de créer des carrés et des cercles. Voici un diagramme de classe sommaire du logiciel avant le changement, ainsi qu'un extrait du code de ***BAKAD*** qui permet de modifier la largeur d'un rectangle de façon à ce qu'il respecte un périmètre précis passé en paramètre.

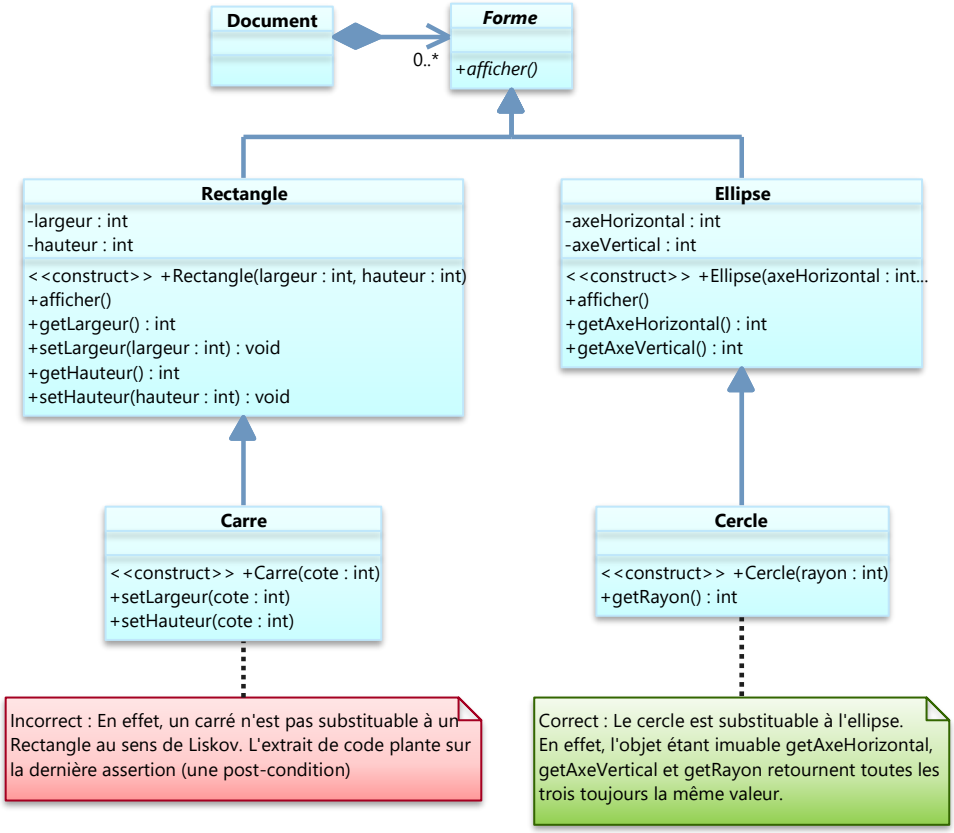


```cpp
class CRectangle
  : public CForme
{
  int m_nLargeur;
  int m_nHauteur;
public:
  CRectangle(int nLargeur, int nHauteur)
    : m_nLargeur(nLargeur)
    , m_nHauteur(nHauteur)
  {}
  void afficher() const override;
  int getLargeur() const {return m_nLargeur;}
  int getHauteur() const {return m_nHauteur;}
  void setLargeur(int nLargeur) {m_nLargeur = nLargeur;}
  void setHauteur(int nHauteur) {m_nHauteur = nHauteur;}
};
```

```cpp
class CEllipse
  : public CForme
{
  int m_nAxeHorizontal;
  int m_nAxeVertical;
public:
  CEllipse(int nAxeHorizontal, int nAxeVertical)
    : m_nAxeHorizontal(nAxeHorizontal)
    , m_nAxeVertical(nAxeVertical)
  {}
  void afficher() const override;
  int getAxeHorizontal() const {return m_nAxeHorizontal;}
  int getAxeVertical() const {return m_nAxeVertical;}
};
```

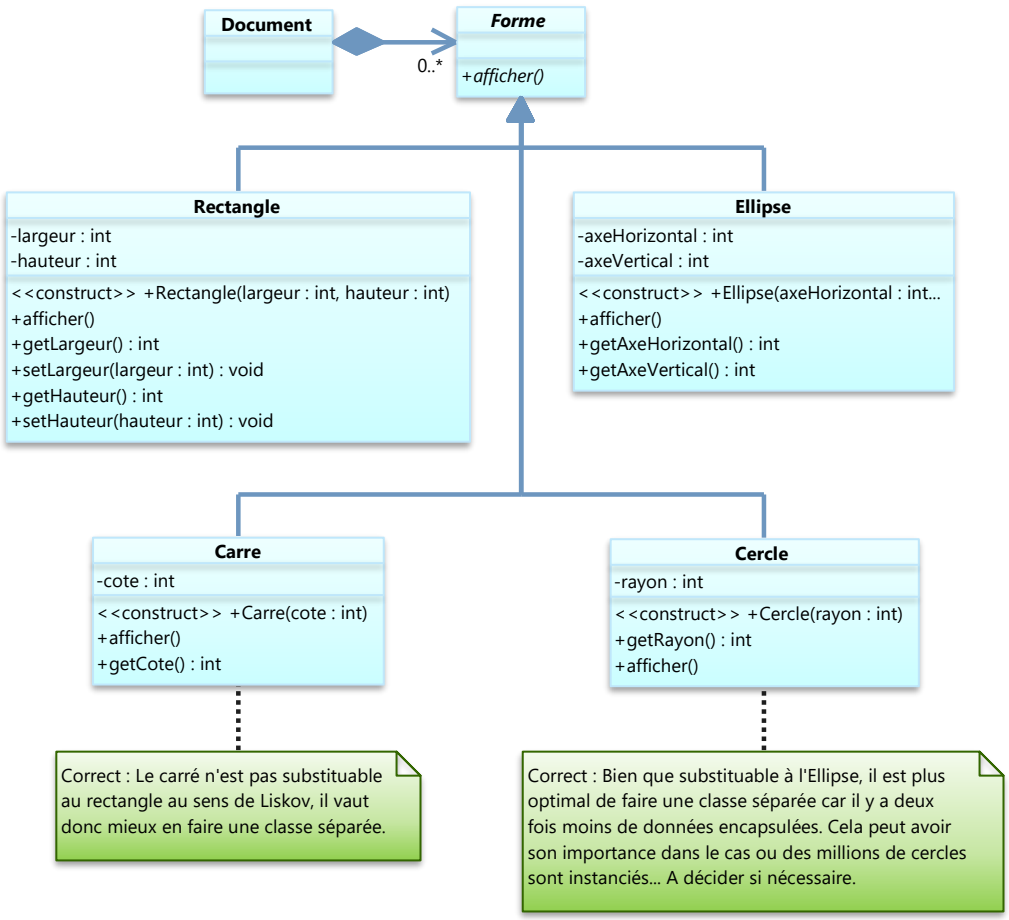
```cpp
/*!
 * \brief
 * Définit le périmètre d'un rectangle en modifiant sa largeur.
 *
 * \param pRect
 * Le rectangle à modifier.
 *
 * \param perim
 * Nouveau périmètre du rectangle.
 *
 * \pre perim doit être pair
 * \pre pRect doit être non nul
 * \post le nouveau périmètre de pRect est perim
 * \invariant la hauteur de pRect est inchangée
 */
void CDocument::setPerimetre(CRectangle* pRect, int perim) const
{
  assert(perim % 2 == 0);
  assert(pRect);

  int nLargeur = perim/2 - pRect->getHauteur();
  if(nLargeur <= 0)
    throw std::invalid_argument("Le périmètre ne peut pas être inférieur "
                                "à deux fois la hauteur du rectangle");
  pRect->setLargeur(perim/2 - pRect->getHauteur());

  assert((pRect->getHauteur()+pRect->getLargeur())*2 == perim);
}
```

- Modifiez la structure de ce programme de façon à pouvoir tracer des carrés de tailles modifiables et des cercles de tailles fixes connues à la construction.
  - Assurez-vous que le programme reste fonctionnel après vos modifications
  - Mauvaise solution :




Il faut noter également qu'avec cette solution, les méthodes **`setLargeur()`** et **`setHauteur()`** doivent devenir virtuelles, sinon le polymorphisme ne fonctionnera pas ! Cela implique une modification du code existant, ce qu'il faut éviter. On peut se dire que c'était une erreur de la première version que nous corrigeons, mais comment aurait-il été possible d'anticiper qu'un jour quelque chose hériterait de **`Rectangle`** au moment de l'écriture de la première version ? Cette question doit éveiller le soupçon sur la solution proposée. Notons enfin que cette problématique est spécifique au C++, car en effet, la plupart des autres langages objets – comme le Java – sont automatiquement polymorphes (toutes les fonctions sont par défaut virtuelles)

- Bonne solution :


Document
Forme
0..*
+afficher()
Rectangle
-largeur : int
-hauteur : int
<<construct>> +Rectangle(largeur : int, hauteur : int)
+afficher()
+getLargeur() : int
+setLargeur(largeur : int) : void
+getHauteur() : int
+setHauteur(hauteur : int) : void
Ellipse
-axeHorizontal : int
-axeVertical : int
<<construct>> +Ellipse(axeHorizontal : int...
+afficher()
+getAxeHorizontal() : int
+getAxeVertical() : int
Carre
-cote : int
<<construct>> +Carre(cote : int)
+afficher()
+getCote() : int
Cercle
-rayon : int
<<construct>> +Cercle(rayon : int)
+getRayon() : int
+afficher()
Correct : Le carré n'est pas substituable au rectangle au sens de Liskov, il vaut donc mieux en faire une classe séparée.
Correct : Bien que substituable à l'Ellipse, il est plus optimal de faire une classe séparée car il y a deux fois moins de données encapsulées. Cela peut avoir son importance dans le cas ou des millions de cercles sont instanciés... A décider si nécessaire.