TP Supplémentaire 03 – Implémentation de BAKAD

L'objectif de ce TP est d'implémenter le logiciel BAKAD du TD 11 – Liskov Substitution Principle

Exercice 0 Initialisation

Pour ce TP, il n'y a pas de code initial à télécharger, vous allez TOUT faire.

Créez un nouveau projet C++ de type Console.

Exercice 1 Remise en forme

Pour rappel, la figure suivante représente le diagramme de classes établi en fin de TD et est suivi des extraits de codes fournis également durant le TD.

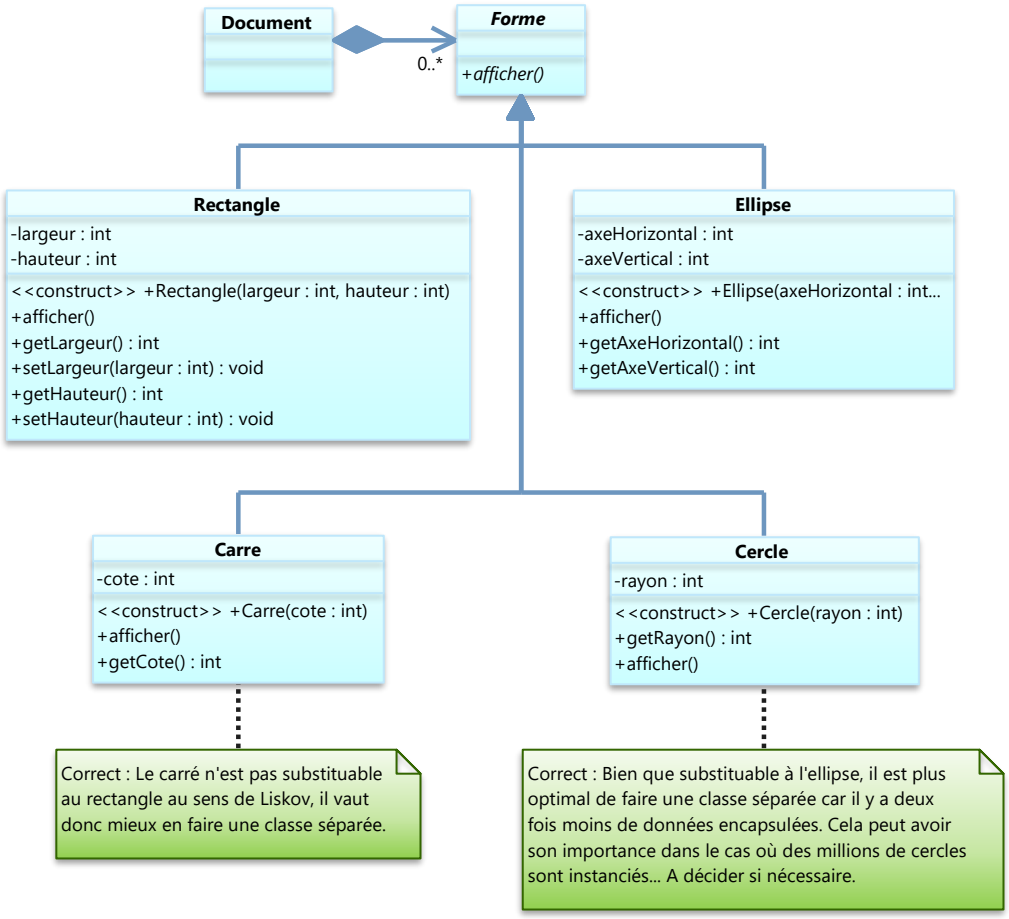


Figure 1 : Diagramme de classe du programme BAKAD en fin de TD

```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);
}
```

- Commencez par implémenter la hiérarchie des classes **`CForme`**.
  - Pour l'instant, laissez vide la fonction **`afficher()`**.

> Nous nous passerons dans ce TP d'implémenter la classe **`CDocument`**. A la place, la liste de formes sera directement implémentée dans la fonction **`main`**, afin de tester le principe.

Exercice 2 Le doc'

- Dans le programme principal, instanciez une liste de formes composée de :
  - Un rectangle de dimension 10 x 20
  - Une ellipse d'axes 30 x 40
  - Un carré de côté 30
  - Un cercle de rayon 42
- Dans le programme principal, parcourez cette liste afin de vous assurer que vous retrouvez bien les bons objets avec les bons attributs.
  - Vous ***pouvez*** utiliser le framework de test LittleTestFramework™
    - https://gitlab-lepuy.iut-clermont.uca.fr/bealbouy/littletestframework

Exercice 3 L'affichage…

Comme vous le savez, mélanger l'affichage et les données, c'est mal. Or, notre architecture semble conçue pour que les objets puissent être affichés depuis leur fonction **`afficher`**. Il ne doit pas être de leur responsabilité de faire l'affichage, mais il est de la leur de le demander. Les formes vont alors ***déléguer*** l'affichage à un objet supplémentaire, dont la responsabilité sera d'afficher la forme.

Selon cette idée, au niveau de la classe **`CCarre`**, la délégation d'affichage pourrait se représenter sous la forme suivante :

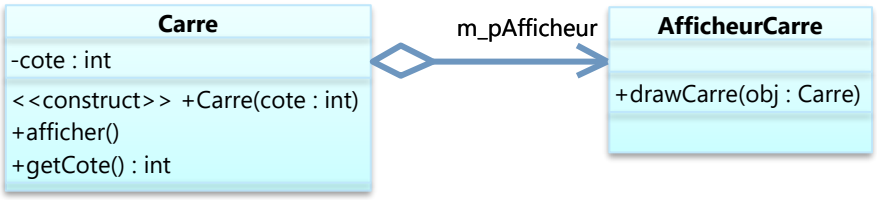


Figure 2 : Délégation d'affichage au niveau de la classe `CCarre`

Le code de la fonction **`CCarre::afficher()`** serait donc :

```
void CCarre::afficher()
{
  m_pAfficheur->drawCarre(*this);
}
```

La fonction **`CAfficheurCarre::drawCarre`** s'occuperait de faire les appels nécessaires pour faire le rendu du carré à l'écran selon la technologie choisie.

Avec cette architecture de classe et de code, on se rend compte que plusieurs objets **CCarre** peuvent partager le même objet **CAfficheurCarre**, puisque le carré à afficher est passé en paramètre de la fonction **CAfficheurCarre::drawCarre**. On peut même aller jusqu'à dire que tous les **CCarre** peuvent partager le même objet délégué. Dans ce cas, on peut passer le membre **m_pAfficheur** en **static** de façon qu'il soit associé à la classe plutôt qu'aux objets. Cela apporte plusieurs avantages. Tout d'abord, il suffira de spécifier à la classe **CCarre** quel est l'objet délégué à l'affichage pour que tous les **CCarre** s'affichent avec la même technologie. Cela allègera le code, car il n'y aura pas besoin d'associer chaque objet **CCarre** avec le délégué, ce sera fait une seule fois pour tous. Ensuite, cela limite l'usage de la mémoire car il n'y a plus un pointeur vers délégué par **CCarre** mais un seul pour tous.

Afin d'accélérer l'implémentation, sachant qu'il n'y a que quatre formes différentes, on peut faire un seul délégué pour les 4 formes en le munissant de 4 fonctions différentes. Ainsi, on peut également l'associer directement à la classe **CForme**. Le diagramme de classe devient donc celui de la figure suivante.

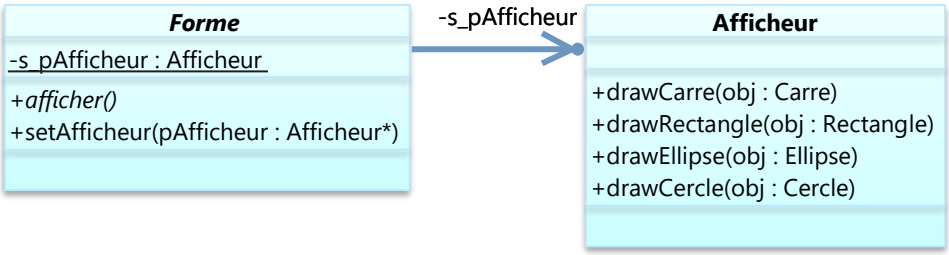


Figure 3 : Délégation d'affichage au niveau de la Forme

3.1 …en morse

- Implémentez ce lien statique entre la classe **CForme** et votre nouvelle classe **CAfficheur**.
  - La classe **CAfficheur** affichera les informations des objets dans la console.
- Dans le programme principal, associez un objet **CAfficheur** avec la classe **CForme**.
- Dans le programme principal, faites une boucle pour afficher toutes les formes de la liste et assurez vous d'obtenir le résultat suivant :

```
Rectangle de largeur x hauteur = 10 x 20
Ellipse d'axes horizontal x vertical = 30 x 40
Carré de côté = 30
Cercle de rayon 42
```

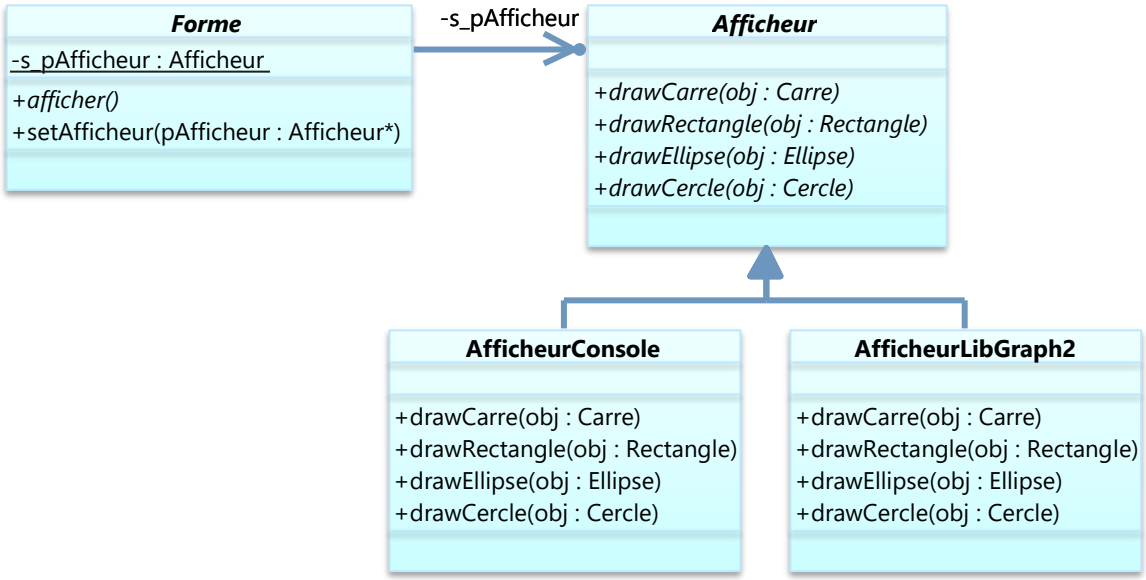
- Dans le programme principal, parcourez la liste des formes pour modifier tous les rectangles afin de leur imposer un périmètre de 100
  - Utilisez la fonction **setPerimetre**
  - Assurez-vous d'obtenir l'affichage suivant :

```
Rectangle de largeur x hauteur = 30 x 20
Ellipse d'axes horizontal x vertical = 30 x 40
Carré de côté = 30
Cercle de rayon 42
```

3.2 …graphiquement

L'affichage console, c'est bien, mais graphiquement c'est mieux. Pour implémenter l'affichage graphique, il faut reprendre la classe **`CAfficheur`**. Oui mais modifier du code, c'est mal, cela ne respecte pas le principe ***Ouvert/Fermé***. En effet, notre architecture n'est pas résiliente au changement. Nous allons donc la reprendre de façon que le changement de technologie d'affichage ne casse pas tout.

Nous aurions dû considérer l'affichage comme quelque chose que l'on sait qui va se réaliser, mais que l'on veut modulaire car on ne sait pas comment il se réalisera. Une solution possible est alors d'utiliser une classe abstraite à la place de **`CAfficheur`**, ce qui permettra par héritage et polymorphisme d'affecter un afficheur concret au moment de l'exécution. Le diagramme de classe pour l'affichage devient alors celui de la figure suivante.



- Implémentez la hiérarchie **`CAfficheur`** ← **`CAfficheurConsole`** et mettez-la en œuvre.
  - Vérifiez que le comportement du programme n'est pas changé.

3.2.1 LibGraph2…

- Vous pouvez vous amuser à implémenter l'affichage graphique avec l'excellentissime bibliothèque LibGraph2™
  - git@gitlab-lepuy.iut.uca.fr:bealbouy/libGraph.git
  - Avec un peu d'astuce, vous pourriez obtenir le résultat suivant :

3.2.2 … ou Qt

- Vous pouvez vous amuser à implémenter l'affichage graphique avec Qt
  - Avec un peu d'astuce, vous pourriez obtenir le résultat suivant :

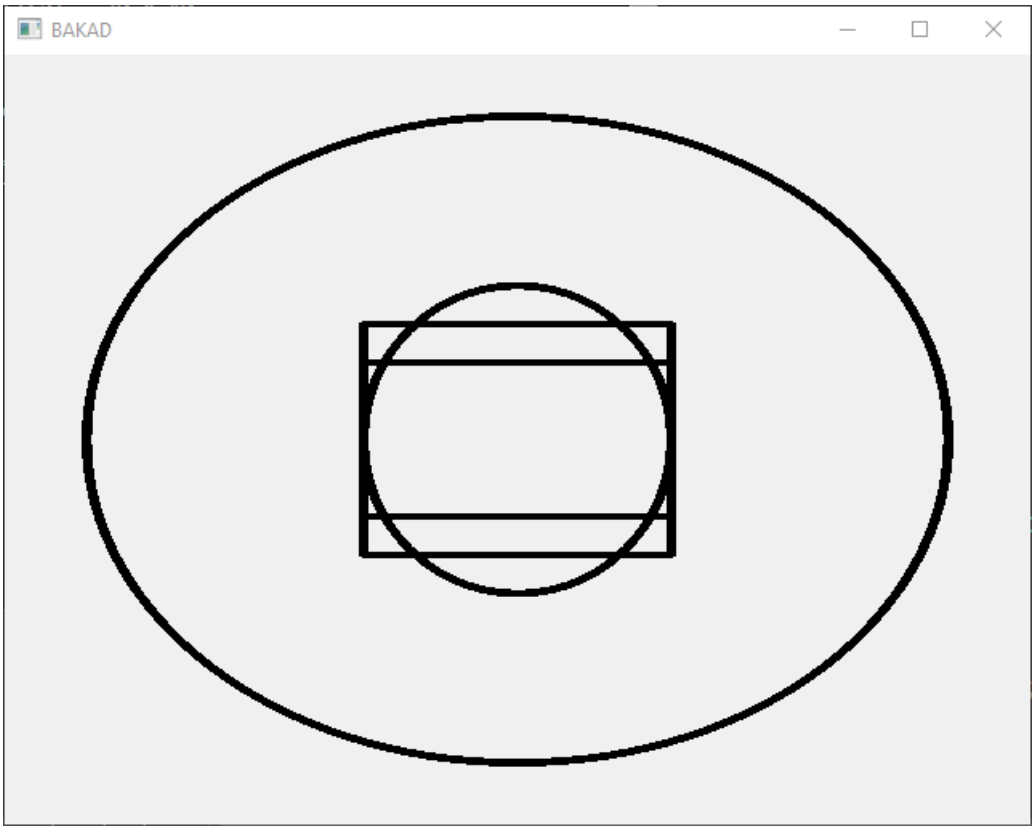


Figure 5 : Capture d'écran du rendu avec Qt dans un repère 100x100 avec l'origine au centre