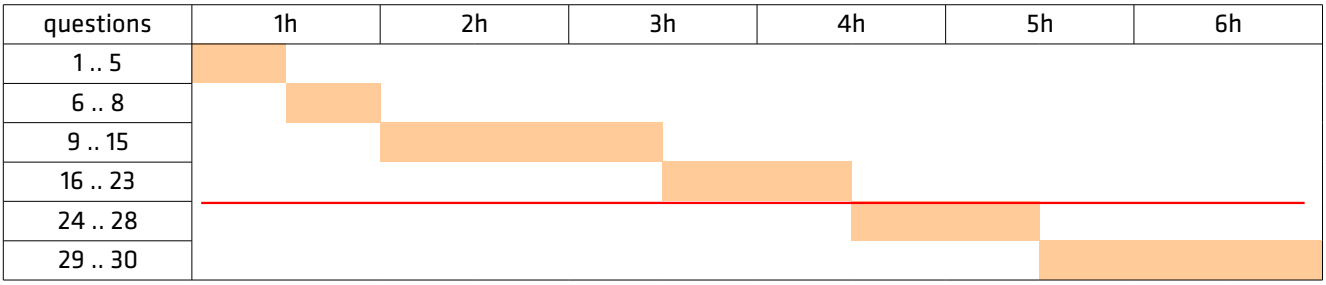
Responsable pédagogique	AF	**AM**	OP	PM
Période	Sem1	Sem2	**Sem3**	Sem4
Volume horaire	Cours/TD		TP	
			6	

[CPP]
UN JOLI PETIT DESSIN !

Indicateur temporel (hors rédaction du compte-rendu) :



Documents à rendre : Compte-rendu minimal accompagné des listings des classes développées.

Ressources : archive tp_CF**igure-v0.6-src.tar.gz** (sources du cours C++ Part. 3)
bibliothèque **XamGraph** version SDL-0.27 ou supérieure

Cahier des Charges

L'objectif de ces travaux est la matérialisation des classes dérivées de **CFigure** exploitées à titre d'exemples dans le cours. La hiérarchie de classes issues de **CFigure** sera ensuite enrichie afin de permettre le dessin d'autres types de figure.

Puisqu'il s'agit de dessin 2D, nous allons utiliser la bibliothèque libre XamGraph qui permet de disposer d'un ensemble ressources graphiques simples...
Pas de panique ! Cette utilisation sera complètement transparente, toutes les fonctionnalités liées au dessin étant encapsulées dans une classe fournie à cet effet.
Travailler de préférence sous Linux... Vérifier la disponibilité de XamGraph.

Mise en œuvre

1. Récupérer l'archive tp_CF**igure-v0.6-src.tar.gz** et la décompresser dans un répertoire de travail. Ce paquetage contient l'ensemble de la hiérarchie de classes vue en cours, et un fichier **Makefile** à renseigner avec le chemin relatif d'accès à **XamGraph**.

2. Générer le programme de test **cfigure** ; il assure le traitement de l'option **-v** (mode *verbose*) transmise éventuellement sur la ligne de commande, le dessin d'un espace de dessin avec un repère orthonormé centré, et l'attente de l'acquittement utilisateur (touche clavier).
   Tester le programme, avec et sans l'option sur la ligne de commande.

3. Étudier le programme principal fourni. Il contient le traitement de l'option de la ligne de commande, puis la création, le paramétrage et la projection d'une instance de classe **CDraft**.

La classe **CDraft** prend en charge l'ensemble des aspects graphiques par héritage de la classe XamGraph. Étudier l'interface de la classe, les méthodes intéressantes pour la suite sont :

```
void line(CPoint p1, CPoint p2, int color = XAM_BLACK ) ;
```

Tracé d'une ligne entre les points **p1** et **p2**, avec la couleur **color**. Matérialisation de l'origine **p1** par le dessin d'une petite ancre (cercle noir).

```
void poly( CPoint p, int nbPts, CPoint* pts,
           int color = XAM_BLACK, bool fill = false ) ;
```

Dessin d'un polygone à partir d'un tableau de **nbPts** points, avec la couleur **color** et un éventuel remplissage de la même couleur. Ancre dessinée au point **p**.

```
void text(CPoint p, std::string s, int color = XAM_BLACK ) ;
```

Dessin du texte **s** à la position **p**, avec la couleur **color**. Ancre dessinée au point **p**.

```
void text(CRectangle r, std::string s ) ;
```

Dessin du texte **s** centré dans le rectangle **r**. Si l'attribut de remplissage du rectangle est actif, le texte est dessiné en noir. Sinon, il est écrit de la couleur de l'objet rectangle. Ancre dessinée à l'origine du rectangle.

La classe **CDraft** va maintenant être spécialisée pour permettre le dessin d'objets de la hiérarchie de classes **CFigure**…

4. Créer une classe **CMyDraft** dérivée de **CDraft**. Placer dans la nouvelle classe les méthodes publiques suivants :

```
CMyDraft(int w = 640, int h = 480 ) : CDraft(w, h ) {}
void draw(bool grad = false ) { CDraft::draw(grad) ; }
```

   À quoi servent-elles ?

5. Mettre à jour le **Makefile** et tester le programme principal en remplaçant la création de l'instance **CDraft** par la création d'une instance **CMyDraft**. Normalement, le résultat est toujours le même…

Ajout d'une propriété

6. Ajouter à la classe **CFigure** une propriété **color** de type **int**.

7. Modifier les constructeurs de **CFigure** de manière à initialiser la propriété à 0 par défaut.

8. Compléter l'implémentation du sélecteur **getColor()** du modificateur **setColor()** déjà présents en *inline* dans la définition de la classe **CFigure**.

Apprendre à CMyDraft comment dessiner des objets de la hiérarchie CFigure

Une première technique permettant de visualiser effectivement les figures consiste à apprendre à un objet **CMyDraft** comment dessiner chacune d'elles ; sachant que la classe hérite des fonctionnalités de dessin offertes par **CDraft**…

Commençons par les objets **CVector**…

9. Ajouter à la classe **CMyDraft** une méthode publique respectant le prototype suivant :

```
void draw(const CVector& v ) ;
```

10. Implémenter cette nouvelle méthode en externe avec le code suivant :

```
void CMyDraft::draw( const CVector& v )
{
      if ( !v.isVisible() ) return ;
      line( v.getPos(), v.getEnd(), v.getColor() ) ;
}
```

Cette méthode n'agit que si la propriété **visible** de l'objet est vraie. Elle fait simplement appel à la primitive **line()** fournie par **CDraft** pour dessiner le vecteur et son ancre.

Pour évaluer le **.h**, le compilateur doit simplement savoir que **CVector** est une classe. Par contre, pour compiler le **.cpp**, il est bien évidemment nécessaire d'inclure le *header* de **CVector**…

11. Ajouter dans l'interface de **CMyDraft** une annonce de classe **CVector**. L'annonce se place avant la déclaration de **CMyDraft** en écrivant :

```
class CVector ;
```

Ajouter l'inclusion idoine dans l'implémentation de la classe.

12. Ajouter au programme de test la même inclusion et les instructions permettant de créer et dessiner un vecteur :

```
CVector v1( CPoint(1.5,2), CPoint(7.2,5) ) ;
v1.setColor( XAM_LIGHT_RED ) ;
v1.setVisible( true ) ;
draft.draw( v1 ) ;
```

Comparer le résultat obtenu avec la copie d'écran fournie en annexe.

13. Enrichir **CMyDraft** avec d'autres méthodes **draw()**, une pour chacun des types **CRectangle**, **CLozenge**, **CLabel** et **CButton** (pour mémoire, un **CSquare** se dessine comme un **CRectangle**, il n'est donc pas utile de créer une méthode pour cette classe d'objets).

14. Implémenter ces 4 nouvelles méthodes en externe. Les rectangles et losanges peuvent être dessinés au moyen de la primitive **poly()** héritée de **CDraft** ; celle-ci attend en argument un tableau de **CPoint** qu'il faut donc fabriquer judicieusement avant l'appel…

15. Ajouter au programme de test les instructions permettant de visualiser les objets suivants

(l'objet **v1** existe déjà, voir question 12) :

objet	v1	r1	s1	z1	l1	b1
classe	CVector	CRectangle	CSquare	CLozenge	CLabel	CButton
visible	true	true	true	true	true	true
pos.x	1.5	-6	2.4	-9	-8	5.5
pos.y	2	4.5	1	-4.5	-2	-5.2
dx	5.7	5	3	3		4
dy	3	2.5	3	1.4		1
color	XAM_LIGHT_RED	XAM_LIGHT_BLUE	XAM_DARK_MAGENTA	XAM_CYAN	XAM_DARK_GREEN	XAM_LAVENDER
fill		true	false	true		true
txt					"MY_CLABEL1"	"MY_CBUTTON1"

Comparer le résultat obtenu avec la copie d'écran fournie en annexe.

Cette technique montre l'intérêt du polymorphisme (les méthodes servant à dessiner se nomment toutes de la même manière), l'utilisation par le programmeur en est d'autant simplifiée…
Mais la technique a ses limites ! Elle oblige notamment à étendre la classe **CMyDraft** à chaque nouvelle création dans la hiérarchie **CFigure**…

Apprendre aux objets CFigure comment s'auto-dessiner sur un CDraft

Une autre solution, certainement plus logique, consiste à doter chaque classe de la hiérarchie des figures d'une méthode de dessin…

16. Compléter le modèle de développement **CFigure** par une nouvelle méthode virtuelle pure :

```
virtual void draw(CDraft& dr ) const = 0 ;
```

Ne pas oublier l'annonce de classe qui va bien…

17. Afin de satisfaire le modèle de développement, implémenter la méthode **draw()** (avec un corps d'instructions vide pour le moment) dans toutes les branches de la hiérarchie nécessitant une primitive de dessin (c'est-à-dire dans les classes **CVector**, **CRectangle**, **CLozenge**, **CLabel** et **CButton**).

18. Le corps de **draw()** de **CVector** peut être renseigné en s'inspirant du travail réalisé à la question 10 :

```
void CVector::draw(CDraft& dr ) const
{
      if ( !isVisible() )   return ;
      dr.line( getPos(), getEnd(), getColor() ) ;
}
```

19. Créer dans le programme de test une collection sous forme d'un tableau **figure** de six pointeurs d'objets **CFigure**. Initialiser le premier de la manière suivante :

```
figure[0] = new CVector( 4, -1.6 ) ;
figure[0]->setPos( CPoint(-5, -4 ) ) ;
figure[0]->setColor( XAM_RED ) ;
```

20. Tester provisoirement l'affichage de la figure en ajoutant :

```
figure[0]->setVisible( true ) ;
figure[0]->draw( draft ) ;
```

Comparer le résultat obtenu avec la copie d'écran fournie en annexe.

21. Implémenter le corps des autres méthodes **draw()**, toujours en s'inspirant des travaux précédents (question 14).

22. Ajouter au programme de test les instructions permettant de créer les objets suivants (le premier objet existe déjà, voir question 19) :

objet	*figure[0]	*figure[1]	*figure[2]	*figure[3]	*figure[4]	*figure[5]
classe	CVector	CRectangle	CSquare	CLozenge	CLabel	CButton
visible	true	true	true	true	true	true
pos.x	-5	-8.5	-3	0	6.5	-8
pos.y	-4	2	-0.5	6	2	6.3
dx	4	1	1.8	4		5
dy	-1.6	3.14	1.8	2.4		0.8
color	XAM_RED	XAM_BLUE	XAM_VIOLET	XAM_DARK_CYAN	XAM_GREEN	XAM_ORANGE
fill		false	true	false		false
txt					"MY_CLABEL2"	"MY_CBUTTON2"

23. Commenter les instructions de la question 20 et mettre en place une boucle d'affichage de la collection complète.
Comparer le résultat obtenu avec la copie d'écran fournie en annexe.

Nouvelles figures

Pour simplifier la suite des travaux, et aussi parce que cela est plus cohérent, nous allons déplacer la propriété **fill** de la classe **CRectangle** vers la classe de base **CFigure**.

24. Ajouter dans **CFigure** la propriété booléenne **fill**, ses accesseurs publics et son initialisation à **false** dans les constructeurs.

25. Commenter ces mêmes éléments dans la classe **CRectangle**.

Pour terminer, nous allons enrichir la hiérarchie avec deux nouvelles classes : la première permettant de représenter un polygone fermé à partir d'une liste de sommets, et la deuxième consacrée au dessin des cercles.

26. Créer les fichiers nécessaires pour une nouvelle classe **`CPolygon`** dérivée de **`CFigure`**.
Étudier les extraits fournis ci-dessous et coder la classe en conformité avec le modèle de développement imposé par la classe de base.

La construction doit accepter un tableau de **`CPoint`** matérialisant les sommets du polygone. Les points sont copiés localement grâce à une méthode protégée nommée **`setPoints()`** ; ils peuvent ainsi être transmis à la primitive de dessin...

```
class CPolygon : public CFigure {

     CPoint*    ppts ;
     int        num ;

  protected :
     void setPoints(int nbPts, CPoint* pts ) ;

  public :
     CPolygon()
     : CFigure(), ppts(NULL), num(0) {}

     CPolygon(CPoint pos, int nbPts = 0, CPoint* pts = NULL )
     : CFigure( pos ) {
          setPoints(nbPts, pts ) ;
     }

     ~CPolygon() { delete [] ppts ; }

     // ... ...

     virtual void draw(CDraft& dr ) const ;
} ;
```

Les méthodes annoncées ci-dessus peuvent être implémentées comme suit :

```
void CPolygon::setPoints(int nbPts, CPoint* pts )
{
     num = nbPts ;
     ppts = NULL ;
     if ( !nbPts ) return ;
     ppts = new CPoint[num] ;
     for (int i = 0 ; i < nbPts ; i++ )    ppts[i] = *( pts + i ) ;
}

void CPolygon::draw(CDraft& dr ) const
{
     if ( ppts == NULL )   return ;
     if ( !isVisible() ) return ;
     dr.poly( getPos(), num, ppts, getColor(), isFill() ) ;
}
```

27. Tester la nouvelle classe en ajoutant les instructions suivantes au programme (dessin d'un triangle) :

```
CPoint     pts1[3] = { CPoint(-1,0), CPoint(1,1), CPoint(1,-1) } ;
CPolygon   poly1( CPoint(0,0), 3, pts1 ) ;
poly1.setVisible( true ) ;
poly1.setColor( XAM_BLACK ) ;
poly1.draw( draft ) ;
```

Comparer le résultat obtenu avec la copie d'écran fournie en annexe.

28. La copie d'écran montre une croix de couleur **XAM_GOLD** localisée en (1,-4) sur le repère. Compléter le programme de test de manière à obtenir le même effet…

La classe **CPolygon** peut être spécialisée pour le tracé de polygones réguliers. Un polygone régulier est facilement définissable par rapport à son centre en spécifiant un nombre de faces ou de sommets et un rayon. En augmentant suffisamment le nombre de faces, on obtient l'illusion d'un cercle…

29. Créer une nouvelle classe **CCircle** dérivée de **CPolygon**. Le constructeur doit accepter un point d'origine et un rayon ; il doit ensuite construire localement un tableau de 12 points avant d'invoquer la méthode **setPoints()** de sa classe mère.

Petit rappel de trigo : Dans un plan, les coordonnées d'un cercle de rayon R sont donnés par
$x = R * \cos(\alpha)$ et $y = R * \sin(\alpha)$ avec $0 \leq \alpha \leq 2\pi$

En C++, les fonctions trigonométriques et la constante **M_PI** sont fournies par :

```
#define _USE_MATH_DEFINES
#include <cmath>
```

La méthode **draw()** de **CCircle** se contentera d'appeler l'homonyme de sa classe de base.

30. Tester la nouvelle classe en ajoutant les instructions suivantes :

```
CCircle circle( CPoint(7.5, -3), 1.2 ) ;
circle.setVisible( true ) ;
circle.setColor( XAM_PINK ) ;
circle.draw( draft ) ;
```

Comparer le résultat obtenu avec la copie d'écran fournie en annexe.

Moralité

Ces petits travaux ont tenté de démontrer l'intérêt de l'héritage. La généralisation regroupe, au sein de la classe **CFigure**, les caractéristiques communes de toute la hiérarchie ; puis chaque classe enfant apporte ses propriétés spécifiques, ou adapte à ses besoins les fonctionnalités dont elle hérite… On parle alors de spécialisation !
Quant au polymorphisme, il simplifie grandement le travail du programmeur utilisant les classes ainsi conçues.

Annexes

Copie d'écran de la fenêtre d'application (version finale)

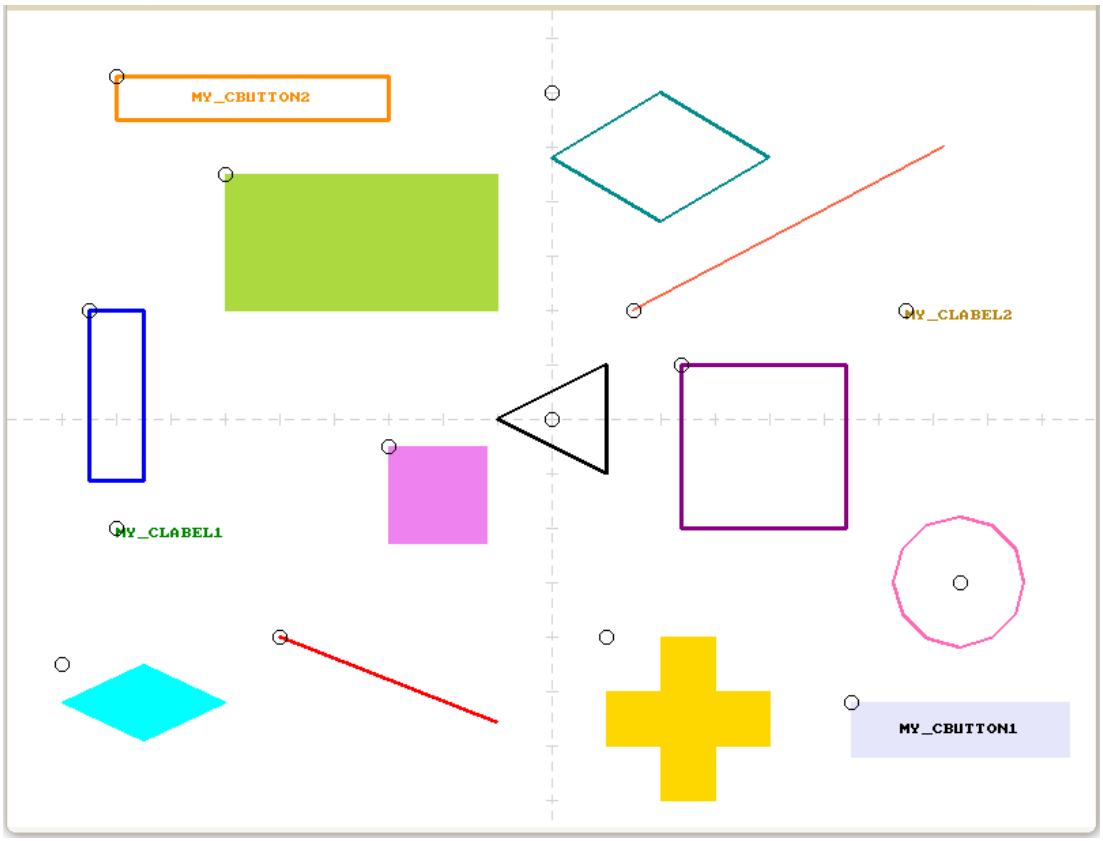


Diagramme de classes (version finale)

BOUML C++ reversing