# - Projet de synthèse -

# Premiers Pas vers l'Ingénierie du logiciel

## *1. Sujet : Gestion de formes géométriques en 2D*

Il est demandé de développer une application distribuée (client C++/serveur JAVA) permettant de gérer des formes géométriques en 2D. L'application doit permettre de construire des formes, d'appliquer quelques transformations géométriques sur celles-ci, de les dessiner et enfin, de les sauvegarder/charger sur un fichier disque.

Sauf en ce qui concerne l'opération de dessin (ce point est détaillé au paragraphe 4), toute notion, toute classe, toute fonctionnalité **DOIT** être écrite en C++.

## *2. Forme géométrique*

Une forme géométrique est caractérisée par une couleur parmi "black", "blue", "red", "green", "yellow" et "cyan". L'application gère deux sortes de formes : les formes géométriques simples et les formes géométriques composées.

- Une forme géométrique simple peut être un soit un segment, soit un cercle, soit un triangle ou encore un polygone quelconque fermé (sans auto-intersection).

- Une forme géométrique composée est appelée groupe. Elle est composée d'une ou plusieurs formes géométriques (simples ou composées) disjointes. Une forme ne peut pas appartenir simultanément à plusieurs groupes.

On suppose qu'il n'y a jamais d'intersection entre les pièces créées.

## *3. Transformations géométriques*

L'application offre la possibilité d'appliquer trois sortes de transformations géométriques aux formes : translation, homothétie et rotation.

- Translation : Une translation est définie par un vecteur de translation

- Homothétie : Une homothétie est définie par un point invariant et par un rapport d'homothétie (nombre réel quelconque). Rappel : une homothétie correspond intuitivement à une opération de zoom.

- Rotation : Une rotation est définie par un point invariant (le centre de la rotation) et par un angle signé donné en radians.

Lorsqu'une transformation géométrique est appliquée à un groupe, toutes les pièces constituant celui-ci subissent la transformation.

Voir annexe pour des conseils de programmation des opérations géométriques.

## *4. Dessiner une forme*

Cette partie du sujet est la plus complexe à réaliser.

### 4.1 Client TCP/IP

Le langage C++ ne fournit pas de librairie graphique standard. Aussi, une application C++, pour effectuer des dessins, doit utiliser d'autres ressources. Par ailleurs, le langage JAVA fournit d'excellentes librairies graphiques (java.awt, javax.swing, FX). Ce projet a donc pour objectif, concernant les opérations de dessin, d'utiliser les librairies graphiques JAVA depuis une application C++. La méthode C++, "dessiner une forme", devient donc un client TCP/IP vers un serveur JAVA de dessin.
La méthode ouvre donc une connexion TCP/IP vers le serveur, puis transmet successivement au serveur les instructions d'ouverture de fenêtre graphique, puis de tracés. Le dessin terminé, elle met fin à la connexion. Le serveur qui est multi-client, se charge d'exécuter les requêtes du client. Pour l'organisation de l'application distribuée, on pourra s'inspirer de la maquette vue en TD (application client-serveur de dessin simplifiée). Le serveur JAVA de dessin doit bien sûr aussi être écrit.

### 4.2 Librairie réseau C++

Sur une plateforme Windows, utilisez la librairie Winsock. Sur les systèmes UNIX/LINUX, utilisez les librairies équivalentes (elles existent et sont très semblables à la winsock).

### 4.3 Protocole :

Vous êtes libre d'inventer le protocole qui vous convient le mieux. Vous pouvez bien sûr utiliser celui vu en TD (cf. maquette), basé sur des chaînes de caractères.

### 4.4 Design Patterns :

**Visitor :** Dans une future extension de l'application, on pourrait envisager d'écrire une seconde version de la méthode "dessiner une forme" utilisant une librairie graphique C++ comme SFML, Qt, wxWidget ou GTK. Le développeur de la future extension devra alors (pour des raisons évidentes de temps de développement) réutiliser les algorithmes mis en oeuvre dans la première version de "dessiner une forme". Dans cette hypothèse, la solution la plus simple consiste à être prévoyant et donc à rendre directement la méthode "dessiner une forme" indépendante de la librairie graphique utilisée.

A cette fin, la méthode "dessiner une forme" doit, par une mise en oeuvre du Design Pattern *Visitor*, **séparer** l'algorithmique du client TCP/IP. De telle sorte que si on change le client TCP/IP par une librairie graphique C++, **on laisse la méthode "dessiner une forme" intacte** (Attention : pas de if-else ou de switch pour distinguer les différentes formes !).

**Chain Of Responsibility :**
Côté serveur, les différentes requêtes d'un client doivent être distinguées par une mise en oeuvre du Design Pattern *Chain Of Responsibility*.

**Singleton :**
L'initialisation de la winsock doit être effectuée par une classe *Singleton* de façon à garantir que l'initialisation soit effectuée exactement une seule fois.

Il est imposé, pour l'application à écrire, de mettre en oeuvre, dans les contextes décrits, ces trois Design Patterns.

### 4.5 Dessin d'un groupe :

Pour le dessin d'un groupe, la couleur du groupe est appliquée à chaque pièce qui en fait partie.

## *5. Sauvegarde/chargement sur un fichier disque*

Une forme peut être sauvegardée ou chargée à partir d'un fichier disque. Ce dernier doit être au format texte. Dans ce fichier, vous êtes libre d'organiser les données comme il vous convient. Cependant, dans une future version de l'application, on pourrait envisager d'autres manières de sauvegarder une forme : base de données, format XML, etc. Dans cette hypothèse, la solution la plus simple consiste à être prévoyant et donc à rendre directement les méthodes "sauvegarder/charger une forme" indépendantes du format de sauvegarde.

Il est donc imposé encore une fois, pour la sauvegarde d'une forme, de se servir du Design Pattern *Visitor*.

Il est aussi imposé, pour le chargement d'une forme, que les différents cas possibles de formes soient distingués par une mise en oeuvre du Design Pattern *Chain Of Responsibility*.

## *6. Fonctionnalités diverses*

- **Calcul d'aire :** L'application offre aussi la possibilité de calculer l'aire d'une forme géométrique. L'aire d'un groupe est la somme des aires des formes qui la composent puisque celles-ci sont toutes disjointes. Voir annexe pour le calcul de l'aire d'un polygone convexe.

- **Conversion en string :** Chaque classe représentant une forme est munie d'un opérateur de conversion d'un objet en string (cf. librairie STL).

## *7. Qualité du code source produit*

Les notions vues en cours telles que classe, héritage, constructeur, destructeur, fonction virtuelle et fonction virtuelle pure doivent être exploitées au maximum afin de simplifier le code produit.
Comme toujours, il faut éviter les redondances de code (copiés-collés) (leur présence signifie qu'on a oublié d'écrire une fonction) et préférer aux maladroites instructions *if*/*else* et *switch,* le mécanisme des fonctions virtuelles.
Les contraintes de cohérence doivent être respectées (exemple : le rayon d'un cercle est toujours strictement positif, une forme ne peut appartenir qu'à un seul groupe à la fois).
La gestion de la mémoire dynamique doit être assurée (constructeurs de copie, destructeurs, destructeurs virtuels).

La notation tiendra compte du strict respect de ces consignes.

Ces consignes sont valables aussi bien en C++ qu'en JAVA.

### *8. Extensibilité de l'application*

On pourrait par la suite ajouter d'autres formes (Ellipse, Rectangle, Etoile, etc…). L'application doit être conçue de telle sorte que le code déjà écrit puisse rester intact. La seule modification à apporter au programme étant l'écriture des classes qui traitent les nouveaux cas particuliers. Une utilisation combinée du DP COR et des notions vues en POO est recommandée pour obtenir ce résultat.
De même, on pourrait par la suite changer de librairie graphique. L'application doit être conçue de telle sorte que ce remplacement affecte le moins possible le code C++ déjà écrit. L'utilisation du DP Visitor permet d'obtenir cette indépendance de l'application vis à vis de la librairie graphique.

La notation tiendra compte du strict respect de ces consignes.

### *9. fonction main(...)*

Le but de la fonction *main(...)* de l'application C++ est de tester les fonctionnalités décrites dans les paragraphes 2 à 6. Elle permet donc de construire/charger des formes, puis de réaliser des opérations sur les formes construites (dessiner, transformer, sauver). Elle utilise directement les méthodes définies dans les classes qui composent l'application. Il n'est pas demandé d'écrire une interface graphique, une application console est suffisante. De même, un menu n'est pas demandé. Lors de la soutenance du projet, il sera demandé d'écrire une fonction *main()* qui réalise de telles opérations.

### *10. Organisation du travail*

Les étudiants peuvent se grouper par binômes pour réaliser le travail.

### *11. Documents à rendre*

#### 11.1 Rapport :

Un rapport (d'une dizaine de pages) expliquant les stratégies utilisées pour résoudre les problèmes posés par le sujet.
Ce rapport contient en particulier les diagrammes UML détaillés des principales hiérarchies de classes définies dans l'application. Il contient également des explications sur le fonctionnement de l'application distribuée ("schémas" de communication, protocoles, etc...)

#### 11.2 Sources + exécutables :

L'exécutable, le byte-code de l'application, les programmes sources C++ et JAVA et la documentation au format HTML sur le code source (cf. outils Doxygen et javadoc) doivent également être rendus.
Les documents du projet doivent être envoyés par courrier électronique à l'adresse suivante : domic62@hotmail.com.

### *12. Date de remise*

Tous les documents doivent être rendus le dimanche 6 mars 2022 à minuit au plus tard.

### *13. Modalités de la soutenance*

Les soutenances seront organisées à partir du mardi 8 mars 2022, elles se dérouleront à l'UFR MIM courant mars 2022 (ou par visio-conférence si les conditions sanitaires ne le permettent pas).
Le planning des soutenances sera communiqué par courrier électronique durant la semaine du lundi 7 février 2022. Il aura la forme d'un fichier excel partagé qui définira les créneaux disponibles pour placer les soutenances. Les étudiants seront invités à s'y inscrire eux-mêmes pour prendre RDV. Tous les membres de l'équipe devront être présents. La soutenance durera environ 45 minutes.

### *14. Notation*

Le travail effectué sera évalué suivant trois critères :

- Rapport
- Qualité de code source, rigueur de programmation et documentation sur les sources
- Fonctionnement de l'application

N'hésitez pas à poser des questions ou à solliciter des RDVs pour discuter de points de programmation.

Cette matière produit deux notes distinctes : une note de rapport et une note de soutenance.
Bon travail

## Annexes

Dans la suite, les grandeurs vectorielles sont notées en **gras**.

### *Annexe 1. Représentation d'un point du plan*

Faites l'hypothèse suivante:

point du plan = vecteur du plan = couple (x,y) de coordonnées réelles.

Donnons quelques rappels de géométrie élémentaire :

Si $\mathbf{v_1}(x_1,y_1)$ et $\mathbf{v_2}(x_2,y_2)$ sont deux vecteurs du plan et $a$ un nombre réel alors

- $\mathbf{v_1} + \mathbf{v_2} = (x_1,y_1) + (x_2,y_2) = (x_1+x_2, y_1+y_2)$
- $a^* \mathbf{v_1} = a^*(x_1,y_1) = (a^*x_1, a^*y_1)$

Si $A(x_A,y_A)$ et $B(x_B,y_B)$ sont deux points du plan alors

$\mathbf{AB} = \mathbf{OB} - \mathbf{OA} = (x_B,y_B) - (x_A,y_A) = (x_B - x_A, y_B - y_A)$

Puis écrivez une classe *Vecteur2D* représentant la notion de couple de coordonnées.

## *Annexe 2. Classe Vecteur2D*

Une esquisse de cette classe peut être la suivante :

```
template <class T>
inline const T operator - (const T & u, const T & v)
{
return u + -v;
}



class Vecteur2D
{
public:
double x,y;

inline explicit Vecteur2D(const double & x = 0, const double & y = 0);

/**
 * DONNEES : s respectant le format "(  nombre réel, nombre réel)"
 *
 * */
inline Vecteur2D(const char * s);
inline const Vecteur2D operator + (const Vecteur2D & u) const;
inline const Vecteur2D operator * (const double & a) const;
/**
 * - unaire (c'est-à- dire opposé d'un vecteur)
 * */
inline const Vecteur2D operator - () const;

..............
autres méthodes
...................
operator string() const;

}; // classe Vecteur2D

inline const Vecteur2D operator *(const double & a, const Vecteur2D & u)
{ return u*a;}

 //------------ implémentation des fonctions inline
----------------------

inline  Vecteur2D::
```

```
Vecteur2D(const double & x, const double & y): x(x),y(y){}

inline const Vecteur2D Vecteur2D::operator + (const Vecteur2D & u) const
{
return Vecteur2D( x+u.x, y+u.y);
}

inline const Vecteur2D Vecteur2D::operator * (const double & a) const
{
return Vecteur2D( x*a, y*a);
}

inline const Vecteur2D Vecteur2D::operator - () const
{
return Vecteur2D(-x,-y);
}

Vecteur2D::operator string() const
{
ostringstream os;
os << "( " << x <<", " << y << ")";
return os.str();
}

ostream & operator << (ostream & os, const Vecteur2D & u)
{
os << (string) u;
return os;
}
.........
```

### *Annexe 3. Test de la classe Vecteur2D (fonction main dans TestVecteur2D.cpp)*

```
int main()
{
cout << "essai des vecteurs 2D \n";

Vecteur2D u1(2,3), u2(2,3), v(5), w, v1(35,-63), u3(3,4), u4(3,-4), v3;

cout << " u1 = " <<  u1 << endl;
cout << " u2 = " <<  u2 << endl;
cout << " u1 - u2 = " <<  u1-u2 << endl;
cout << " 5*u1 = " <<  5*u1 << endl;

.........
```

### *Annexe 4. Calcul de l'aire d'un polygone convexe*

Le polygone doit être décomposé en triangles. L'aire du polygone est alors la somme des aires des triangles qui le composent. L'aire d'un triangle (ABC) est obtenue en calculant la moitié du déterminant des vecteurs **AB** et **AC,** ou plus formellement :
Aire (triangle ABC) = 0.5 * det(**AB**,**AC**).

Idée : rajouter une méthode *déterminant* dans la classe Vecteur2D.