

# Premier Pas vers l'Ingénierie du logiciel

# Projet de synthèse

## Sommaire :

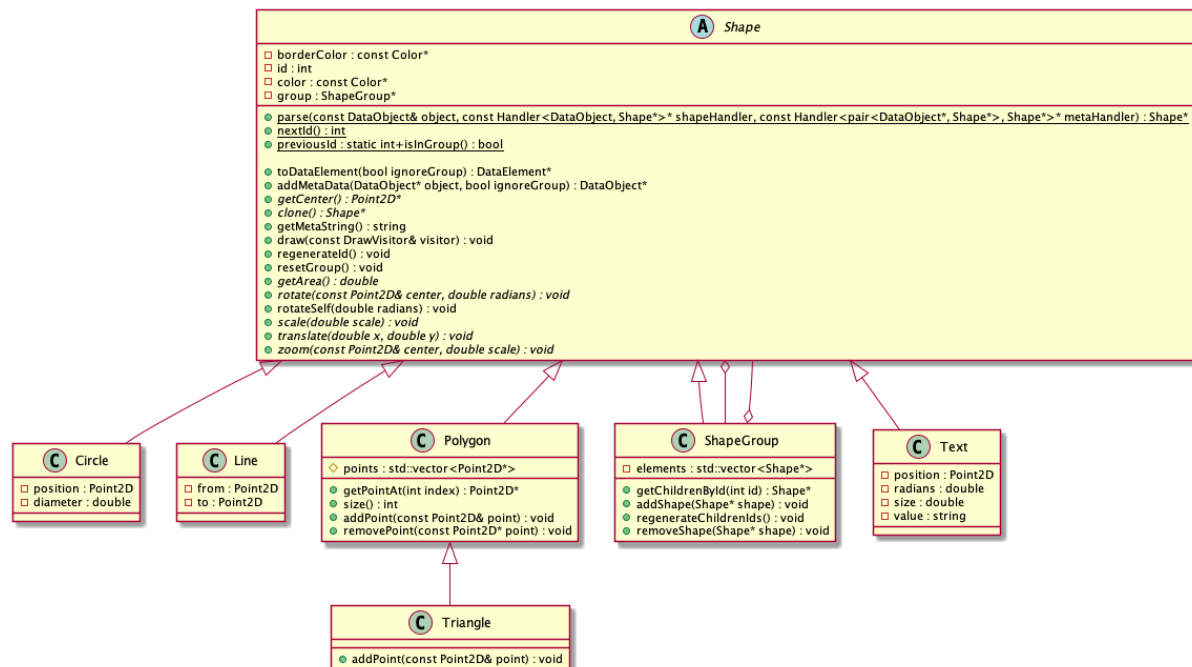
1. Créations des formes	2
2. Communication avec le serveur	3
1. Quel protocole de communication adapter	4
1. Une chaîne de caractère ?	4
2. Format JSON ?	5
3. Adaptabilité au futur	6
2. Réception de l'information sur le serveur de dessin	7
3. Sauvegarde et chargement des scènes	9
4. Implémentation d'un serveur de dessin en Java	10
1. Implémentation à l'aide de Swing	12
5. Fonctionnalités supplémentaires	13
1. Swing	13
2. Interface en ligne de commande	13

## 1. Créations des formes

Chaque forme géométrique est caractérisée par une couleur, parfois une couleur de bordure; de plus cette forme peut être composée pour former un groupe.

Toutes ces formes doivent avoir les mêmes capacités, nous devons être capables d'exécuter plusieurs transformations : translation, rotation, agrandissement, zoom.

Nous nous retrouvons donc avec cette hiérarchie.



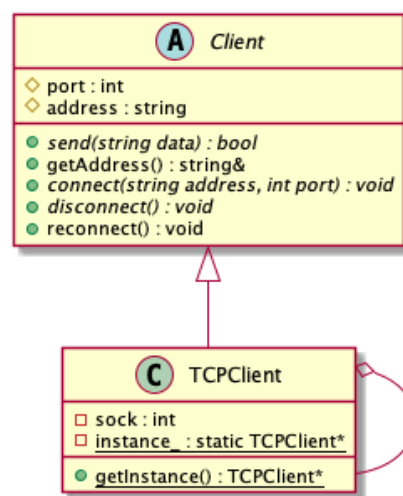
Graphes d'héritage de la classe Shape

Toutes les formes héritent de la classe abstraite `Shape` qui contient les définitions des méthodes communes, ainsi lorsqu'une propriété est ajoutée sur cette classe ( exemple: un axe z de positionnement ), toutes les formes seront directement à jour.

## 2. Communication avec le serveur

La communication entre le client et le serveur se fait à l'aide de socket, sur le protocole TCP.

Cependant il est aussi possible de modifier ce fonctionnement grâce à la classe abstraite `Client` dont `TCPClient` en est une implémentation.



Graphe d'héritage de la classe Client

L'objectif de cette communication est de créer une forme sur l'application en C++ et de dessiner cette forme sur une application externe ( ici en Java ).

Par exemple, il est possible de demander :

- « Ajouter un cercle de rayon 5 en (0,0) et en rouge »

## 1. Quel protocole de communication adapter

Le protocole de communication doit être connu des deux parties et doit permettre le dialogue ( Dans notre cas, l'échange ne se fera que dans une direction ).

### 1. Une chaine de caractère ?

Cette chaine de caractères séparée par des caractères spéciaux était mon premier choix pour sa simplicité.

En effet, il suffit de concaténer des chaines de caractères et de les séparer par un symbole unique. On se retrouve avec une chaine complexe, contenant toutes les informations utiles à la communication.

Cependant, on se retrouvait avec des chaines du type :

```
D_CIRCLE=0#0##5##0#255#0
```

Ici on dessine un cercle en rouge en (0,0)

Cette ligne est toujours facile à interpréter, on comprend qu'il s'agit d'un cercle et que le message est composé de plusieurs parties.

Mais sans savoir ce qui a été envoyé, il m'est impossible de savoir le contenu. Est-ce que le 0##0 correspond à une position, est-ce que c'est x##y ou y##x ? Est-ce que 0##255##0 est un code couleur, et si c'est le cas, est-ce que c'est la couleur de bordure ou la couleur de fond ?

Le seul moyen d'interpréter ce message est de toujours envoyer les informations et dans le même ordre, même si une partie de l'information n'est pas utile : Une forme qui n'a pas de couleur de bordure, devra quand même envoyer l'information.

```
D_CIRCLE=0#0##5##0#255#0##0#0#0
```

Exemple précédent avec information de bordure inutile

Cette approche est fonctionnelle mais elle n'est pas adaptable ou ouverte vers le futur. En effet, si on décide d'ajouter une propriété sur une forme, toutes les implémentations de dessin ne fonctionneront plus. Le principe est identique si une propriété disparaît, il n'est pas raisonnable d'envoyer une information uniquement pour que certains interpréteurs continuent de fonctionner.  
(Il faudra bien évidemment implémenter les modifications à un moment.)

## 2. Format JSON ?

Ce format est très utile pour l'envoi de données car chaque propriété correspond à une clé dans l'arbre.

En reprenant l'exemple précédent, on se retrouve avec:

```
{
  "x": 0,
  "y": 0,
  "diameter": 5,
  "color": {
    "r": 0,
    "g": 255,
    "b": 255
  }
}
```

Les problèmes précédents sur la compréhension du message sont résolus, le message contient une partie sur la position, la couleur et le diamètre, et ces parties sont correctement délimitées.

Ajouter une nouvelle propriété ou en retirer une n'est plus un souci car chaque paramètre a un identifiant ce qui permet d'isoler chaque partie d'information.

Imaginons que la forme n'ait pas de couleur de fond, il est tout à fait possible de retirer cette propriété, cela n'impactera pas la compréhension du message.

De plus, le code est lisible pour le développeur, voire modifiable par l'utilisateur ( Voir le point sur la sauvegarde ).

Le souci est maintenant dans la complexité d'interprétation, séparer la chaîne sur un caractère ne fonctionne plus, il faut donc créer un interpréteur et un sérialiseur pour le json. Ce qui rend l'opération de lecture/écriture plus longue.

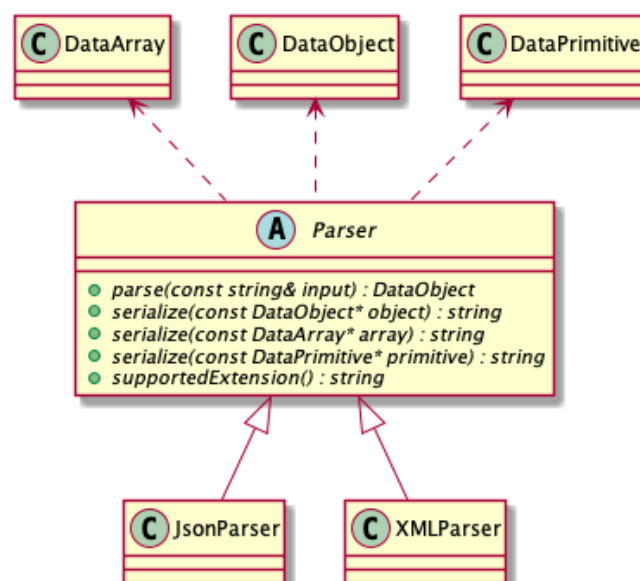
*Remarque : Le JSON créé dans notre code n'est pas formaté, il ne contient aucun saut de ligne ni espaces hors des guillemets*

Pour les raisons énoncées plus haut, j'ai fait le choix d'utiliser le format JSON.

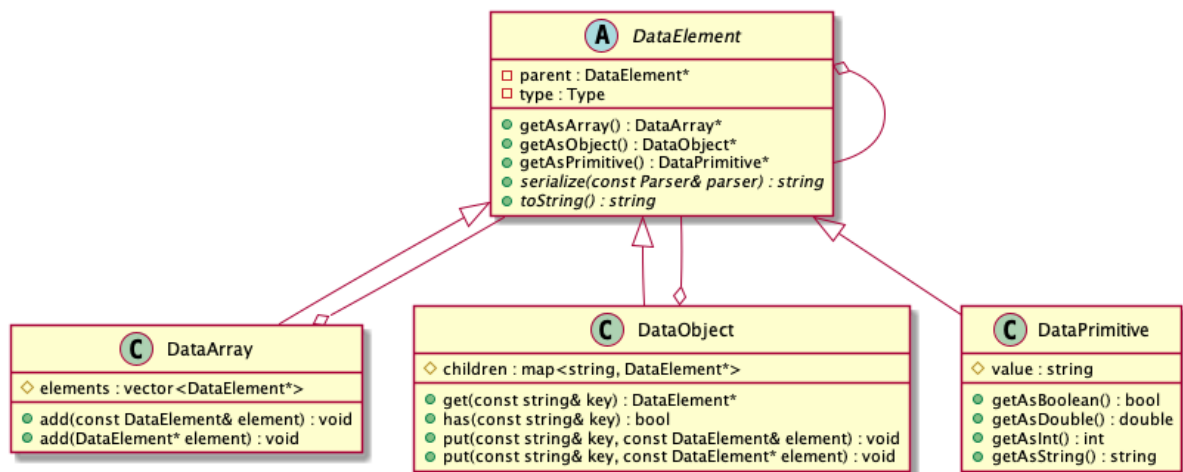
### 3. Adaptabilité au futur

Même si le format JSON propose une adaptabilité au futur grâce à la possibilité de modifier les paramètres, se limiter qu'à un seul format n'est pas viable, on peut envisager que dans le futur une application ait besoin de lire les données dans un autre format.

Ainsi pour résoudre ce problème j'utilise le pattern visitor.



Graphe d'héritage de la classe Parser



Graphe d'héritage de la classe DataElement

La classe abstraite DataElement contient des méthodes pour sérialiser / parser l'objet concerné à l'aide d'une classe `Parser` passée en paramètre.

À l'inverse, la classe la classe Parser contient une méthode pour transformer une chaine de caractère en un DataObject.

Il sera donc possible d'utiliser d'autres formats comme le xml, ou la chaine de caractère pour la sauvegarde, le chargement ou l'envoi d'information ( Il est possible d'utiliser un format différent pour chaque actions ).

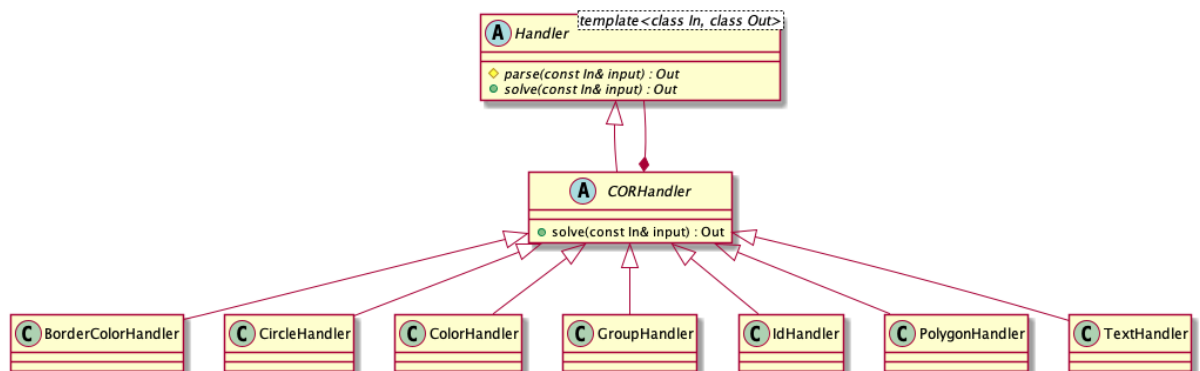
*Remarque : L'utilisation de librairie comme gson côté Java aurait pu être faite en créant un adaptateur qui traduit les méthodes de notre parser vers celles de ces libraires, mais pour le côté éducatif, j'ai préféré le refaire entièrement moi-même. ( Pour cette raison, nous allons retrouver des noms de méthodes similaire à gson )*

## 2. Réception de l'information sur le serveur de dessin

Dans notre application il est possible de dessiner plusieurs type de formes et de manières différentes (couleurs, bordure, etc)

En suivant la logique JSON, nous pouvons envoyer un dessin sous cette forme.

Une chaine de responsabilité est donc adaptée ici.



Graphe d'héritage de Handler

Le principe d'une la chaine de responsabilité est de résoudre un problème en testant une par une des solutions, et tant qu'une solution n'est pas trouvée ( et qu'il reste des solutions à tester ), on passe à l'expert suivant.

Ici l'expert est une implémentation de `Handler`, il est donc possible d'utiliser une chaine de responsabilité mais aussi n'importe qu'elle autre implémentation.

Dans notre cas, il nous suffit donc de vérifier si la clé qui nous intéresse existe dans l'object. ( Ici la clé intéressante est « CIRCLE » )  
Et d'appliquer la logique correspondante à l'interprétation de la forme.

```

{
  "CIRCLE": {
    "diameter": "50.000000",
    "position": {
      "x": "200.000000",
      "y": "200.000000"
    }
  },
  "meta": {
    "color": {
      "a": "255",
      "b": "0",
      "g": "0",
      "r": "255"
    },
    "id": "1"
  }
}

```



Une logique similaire est appliquée pour l'interprétation des métadonnées de la forme ( couleur, bordure, etc ).

Dans le cas où l'information est absente, il est aussi possible d'y ajouter une valeur par défaut ou de lancer une exception. Par exemple, dans l'interprétation faite avec Java Swing, lorsque la propriété `color` est manquante, on assigne une couleur transparente.

*Remarque : Il était aussi possible d'utiliser d'autres méthodes, un simple tableau pouvait même être plus performant.*

*Par exemple: la clé « Circle » peut avoir comme valeur son interpréteur, ce qui évite la boucle réalisée dans la chaîne de responsabilité.*

### 3. Sauvegarde et chargement des scènes

```
{
  "items": [
    {
      "CIRCLE": {
        "diameter": "50.000000",
        "position": {
          "x": "200.000000",
          "y": "200.000000"
        }
      },
      "meta": {
        "color": {
          "a": "255",
          "b": "0",
          "g": "0",
          "r": "255"
        },
        "id": "1"
      }
    }
  ],
  "window": {
    "height": "500",
    "width": "1000",
    "name": "Exemple"
  }
}
```

Une scène est composée de formes, d'un titre, et d'une dimension.

Pour la sérialisation et l'interprétation, nous utilisons le même principe que pour les scènes avec le Parser. Il est donc possible de charger un fichier dans n'importe quel format, tant qu'il est implémenté.

Le chargement est réalisé en exécutant la chaîne de responsabilité sur chaque élément dans la liste « items ».

Puis les informations de scènes sont lues dans l'objet « window ».

A l'inverse, la sauvegarde est réalisée en utilisant la méthode de sérialisation sur chaque forme et de les ajouter à une liste.

























L'avantage du Json ici est sa lisibilité pour l'utilisateur, comme ce format est très populaire et simple d'utilisation, modifier ce fichier est à la portée de tout le monde : on remarque directement où modifier ce que l'on veut grâce à la structure en cascade et aux clés.

## 4. Implémentation d'un serveur de dessin en Java

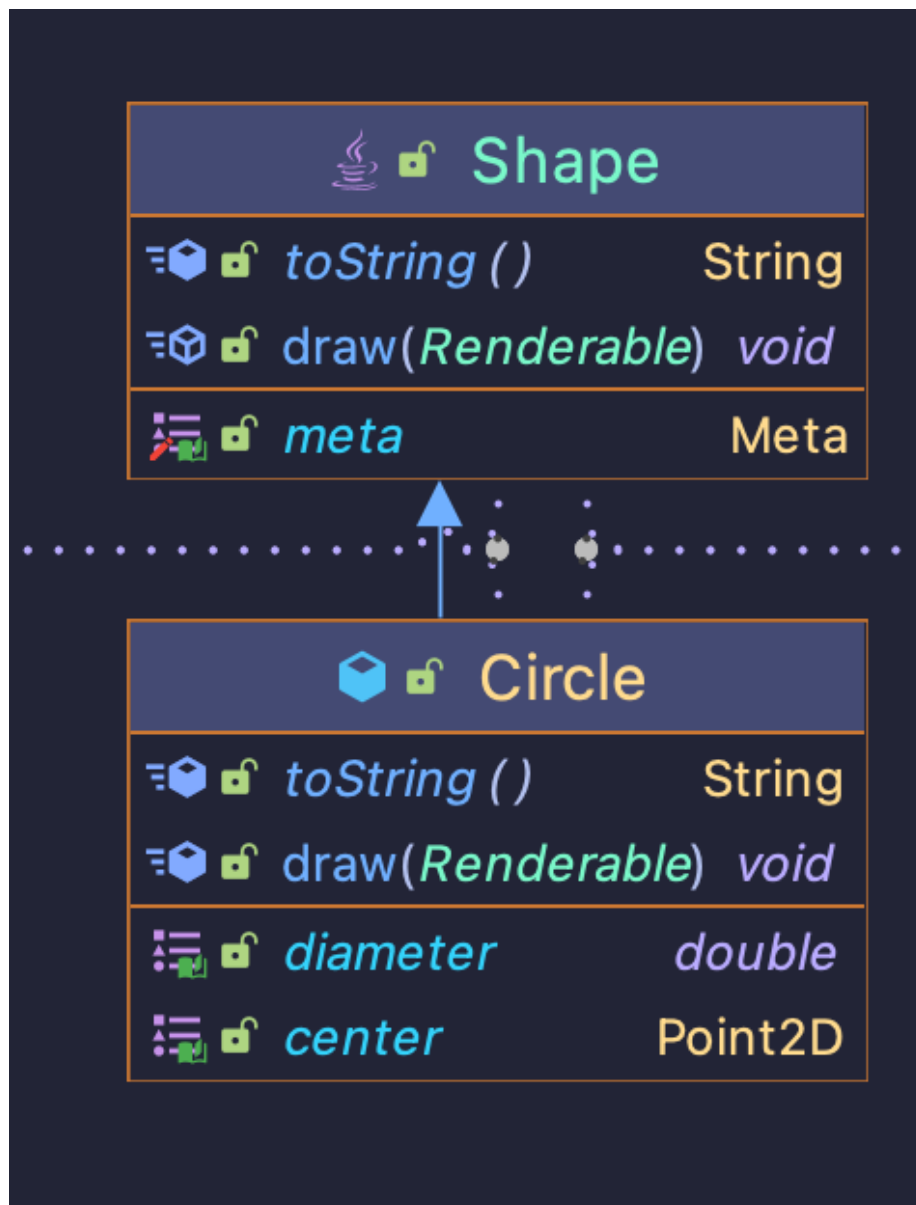
La communication fonctionne à l'exacte opposée du client, nous nous retrouvons avec un Thread qui attend qu'un client lui envoie un message pour le décomposer et dessiner une forme.

Comme le protocole doit être identique entre le serveur et le client, nous retrouvons les mêmes classes pour le Parser et les chaînes de responsabilités liées à la sauvegarde / chargement de fichier.

Il existe aussi plusieurs moyen de réaliser un dessin en java, pour cela nous utilisons une classe abstraite `Renderable` regroupant toutes les méthodes nécessaire au fonctionnement de l'application.

 <i>Renderable</i>		
 	<code>drawLine(Line)</code>	<i>void</i>
 	<code>resetGraphics()</code>	<i>void</i>
 	<code>drawShape(Shape)</code>	<i>void</i>
 	<code>resetScene()</code>	<i>void</i>
 	<code>disposeBuffer()</code>	<i>void</i>
 	<code>setSize(int, int)</code>	<i>void</i>
 	<code>drawText(Text)</code>	<i>void</i>
 	<code>redraw()</code>	<i>void</i>
 	<code>drawPolygon(Polygon)</code>	<i>void</i>
 	<code>setBackgroundColor(Color)</code>	<i>void</i>
 	<code>drawCircle(Circle)</code>	<i>void</i>
 	<code>setTitle(String)</code>	<i>void</i>

Chaque forme ne contient que les informations nécessaires à la réalisation du dessin ainsi qu'une référence à une classe Meta qui contient les informations sur la couleur de fond, de bordure, etc.



A chaque fois que la forme doit être dessinée (agrandissement, nouvel ajout etc.) la méthode `Shape.draw` est appelée.

Ajouter une implémentation en Java revient donc à créer une classe qui implémente ces méthodes. Toutes les formes sont indépendantes de l'implémentation réalisée.

En résumé, pour qu'une forme s'affiche sur le serveur Java, toutes ces étapes sont réalisées:

1. Création de la forme sur le client C++
2. Sérialisation de la forme en utilisant un Parser
3. Envoie de la chaîne de caractère en utilisant un Client
4. Le message est reçu côté serveur
5. Un objet `Shape` est créé à la suite de l'interprétation
6. L'objet est ajouté au `Renderable`

## 1. Implémentation à l'aide de Swing

Par défaut, lorsqu'on dessine sur une fenêtre Swing, chaque mouvement de fenêtre, redimensionnement ou autre provoque une actualisation de la zone de dessin.

Pour éviter cela, j'ai utilisé une technique dite d' « Active Rendering » à l'aide d'un double buffer.

*Le principe est que chaque forme est dessinée sur un écran fantôme, puis lorsque tout est prêt, l'écran fantôme remplace l'écran affiché, puis un nouveau cycle commence.*

Pour gérer le redimensionnement de la fenêtre, chaque forme est enregistrée dans une liste lorsqu'on l'ajoute à la scène puis à la fin du redimensionnement, toutes les formes de la liste sont re-dessinées.

- La zone de dessin est zoomée en conséquence, à l'aide d'un ratio entre la taille désirée initialement et la taille actuelle de la fenêtre.

Initialement avec swing, la position (0,0) correspond au coin haut gauche. Pour corriger cela toutes les formes sont décalée de  $W/2$  en  $x$  et  $H/2$  en  $y$ .

## 5. Fonctionnalités supplémentaires

### 1. Swing

Swing est une librairie graphique très complète, il est donc possible d'y ajouter des fonctionnalités en plus du dessin.

Par exemple, j'y ai ajouté une fonction de zoom à l'aide de la molette. Aucun calcul n'est nécessaire, swing propose nativement une fonction ``scale``.

Une autre fonctionnalité est la possibilité de déplacer la caméra, cette fonctionnalité est réalisée en déplaçant la zone de dessin sur le même principe que le décalage du (0,0).

### 2. Interface en ligne de commande

Actuellement toutes les actions de gestions de formes, sauvegardes et d'envoi sont gérées à l'avance par le développeur.

Une extension possible est d'ouvrir ces fonctionnalités à l'utilisateur à l'aide de commandes sur le terminal.

```
cli > load scene1.json
Scene SolarSystem has been loaded
cli > draw swing
SolarSystem has been drawn
cli > █
```

Chargement puis dessin d'une scène