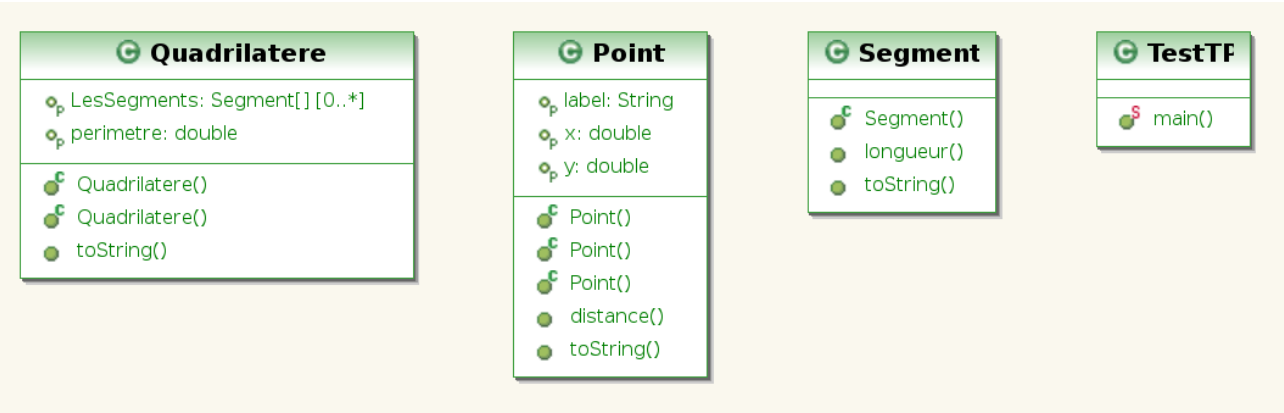
Thanh LUU – 3A STI

Exercice 1 : Une première série de classe

a) On distingue 4 classes dans le projet initial représentées par les diagrammes UML suivants



b) TestTP.java possède déjà 4 initialisations d'objets de la classe Point.
Chaque initialisation fait appel à un constructeur différent de la classe Point. L'existence de plusieurs constructeurs est visible dans le diagrammes UML.

La première initialisation fait appel au constructeur à trois arguments : deux doubles pour les coordonnées et un string pour le nom, dont le prototype est :

```
public Point(double unX, double unY, String unLabel)
```

La deuxième initialisation fait appel au constructeur sans arguments, qui définit un point de coordonnées nulles : l'origine.
Ce constructeur est considéré comme le constructeur par défaut.

```
public Point()
{
   this(0.0,0.0,"O");
}
```

Enfin un dernier constructeur non utilisé utilise deux arguments : les coordonées en double tandis que le nom est initialisé à Null, de prototype :

```
public Point(double unX, double unY)
```

Exercice 2 : Héritage : classe Rectangle

c) Dans la console nous obtenons :

```
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Quadrilatère

Constructeur de Rectangle
Perimetre du rectangle : 0.0
```

On comprend qu'à cause de l'héritage, lors de l'instruction

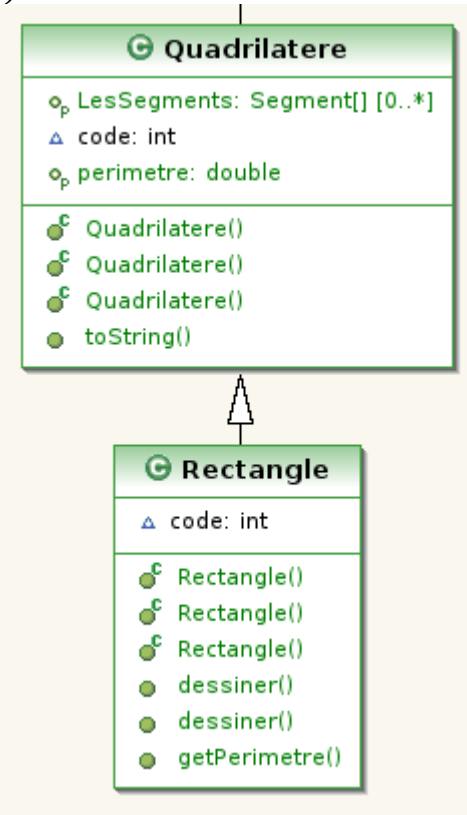
```
Rectangle rect = new Rectangle();
```

on fait d'abord appel au constructeur de la classe Quadrilatère avant d'afficher notre message (issue de la première instruction) du constructeur Rectangle.

Enfin, l'héritage est immédiat grâce au resultat du perimetre, obtenu via

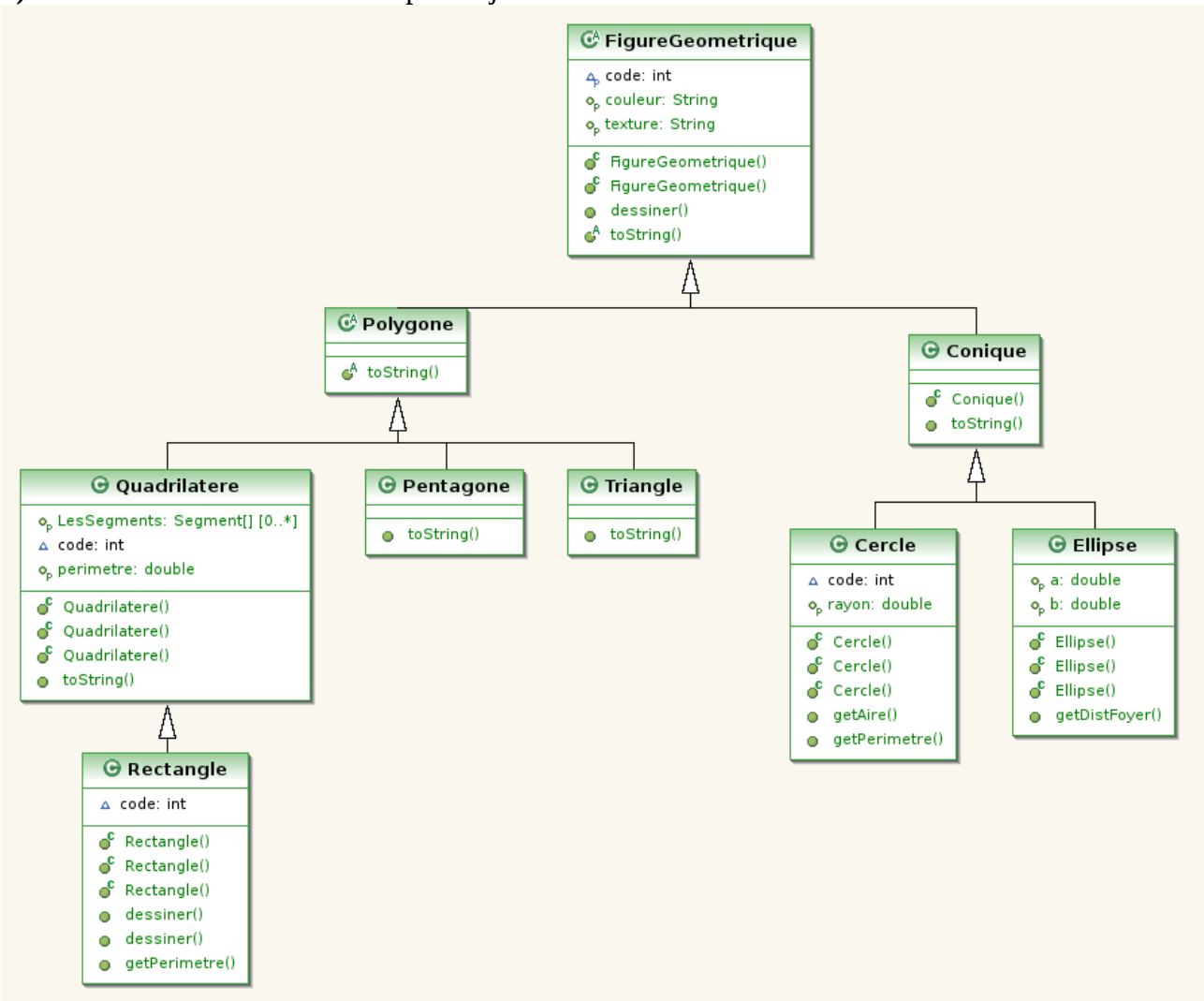
```
System.out.println("Perimetre du rectangle : "+rect.getPerimetre());
```

e) Le schéma UML obtenu est le suivant :
(*je n'avais pas le plugin eUML2 lorsque j'ai fait cette question chez moi, donc la capture a été faite après toutes les autres questions)*



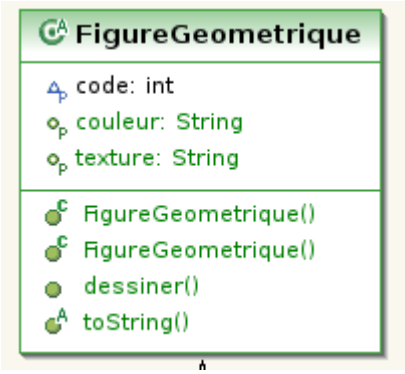
Exercice 3 : Héritage : quelques classes supplémentaires

a) Voici le schéma UML obtenu après l'ajout des différentes classes :



Je tiens à préciser que je n'ai pas modifié les classes filles en profondeur puisque j'ai décidé de créer un **constructeur par défaut** dans la classe *FigureGeometrique*.
À la place, j'ai juste rajouté des constructeurs supplémentaires pour y inclure l'ajout de la texture et de la couleur dès la création d'un nouvel objet.

c) On obtient le schéma suivant :



On voit deux constructeurs parce que j'ai aussi créé un constructeur par défaut, en plus de celui demandé.

Exercice 4 : Polymorphisme

b) À l'exécution on obtient le résultat suivant :

```
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Quadrilatère

Constructeur de Rectangle
Dessin de la zone 4 d'une figure géométrique
Dessin d'une figure géométrique
Dessin d'un rectangle
```

Ce comportement est normal puisque l'on fait d'abord appelle aux méthodes mères qui sont au fur et à mesure redéfinies par les classes héritées.

c) On constate bien l'utilisation de deux méthodes **getPerimetre()** différentes

```
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Quadrilatère
Calcul du périmètre d'un quadrilatère

Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Un segment de droite vient d'être créé
Quadrilatère

Constructeur de Rectangle
Calcul du périmètre d'un rectangle
```

e) L'exécution de la commande nous provoque une erreur (comme l'indiquait l'IDE)

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The method getPerimetre() is undefined for the type FigureGeometrique

	at geometrie.TestTP.main(TestTP.java:94)
```

L'IDE evalue le type selon celui du « contenaire ». Ici, on a stocké un objet de type *Rectangle*, *Quadrilatere* ou *Cercle* dans une liste de type *FigureGeometrique*.
Or nous n'avons pas défini de méthode **getPerimetre()** pour la classe *FigureGeometrique*.

Il est cependant possible de déclencher la méthode **getPerimetre()** de *Rectangle* grâce à un cast lors de l'appel.

```
((Rectangle)listFig[0]).getPerimetre();
```

Il y a deux moyens de déclencher la méthode **getPerimetre()** de *FigureGeometrique.*
On peut bêtement définir une méthode **getPerimetre()** qui sera redéfinie à chaque fois dans les classes héritées, mais ceci n'est pas élégant au niveau du code car on crée une méthode inutile dans la classe mère.
La deuxième option serait de créer une **méthode abstraite** afin d'expliquer à l'IDE que la méthode existe et qu'elle sera redéfinie par les classes héritées. On devra cependant modifier le code en conséquence. Je pense que c'est la solution la plus appropriée.

f) On essaie ici de faire du polymorphisme. Cependant, c'est **rectangle qui hérite** de Quadrilatère. On ne peut donc pas créer une dérivée de Quadrilatère *depuis* Rectangle. L'inverse est cependant possible.

h) On obtient :

```
0
0
0
```

Encore une fois, l'attribut *code* demandé à l'instruction renvoie la variable propre au type du « conteneur », donc l'objet FigureGeometrique.
Avec l'instruction ci-dessous, j'obtiens cependant bien 2. Il faut donc encore caster pour obtenir le typage souhaité.

```
System.out.println(((Rectangle)listFig[0]).code);
```

Exercice 5 : Visibilité des méthodes et attributs

a) Après l'ajout de la méthode, l'IDE affiche une erreur à la compilation :

```
Multiple markers at this line
	- overrides geometrie.FigureGeometrique.dessiner
	- Cannot override the final method from Quadrilatere
	- Method breakpoint:Rectangle [entry] - dessiner()
```

Ceci s'explique par l'ajout du mot clé **final** qui empêche les classes héritées de *Quadrilatere* de redéfinir la méthode **dessiner()**. J'ai annulé la mise en place du « final » après cette question.

b) Nous obtenons les messages d'erreurs suivants :

```
The type Ellipse cannot subclass the final class Conique
```

```
The type Cercle cannot subclass the final class Conique
```

Ces messages étaient prévisibles puisque l'on essaie de faire hériter depuis une classe **final**. Ce mot clé rend justement cette action impossible.
De même, plusieurs méthodes utilisant les **setters** et **getters** de cette classe finale ne sont plus visibles dans les classes héritées.

Il y a restriction totale à la visibilité de la classe final (pour les classes héritées et même à travers le package) d'une part, et d'autre part, ce mot-clé empêche les classes filles de redéfinir des méthodes finales.

c) On obtient le message d'erreur suivant :

```
Multiple markers at this line
	- Cannot reduce the visibility of the inherited method from FigureGeometrique
	- overrides geometrie.FigureGeometrique.dessiner
```

J'ai proposé de restreindre la méthode **FigureGeometrique.dessiner()** à protected aussi pour corriger l'erreur.

d) On obtient l'erreur

```
The type Conique must implement the inherited abstract method  FigureGeometrique.toString()
```

Elle était prévisible puisque l'on doit obligatoirement rendre les classes héritées concrètes en vue de leur utilisation.