Développement Orienté Objet

Cours 5 : Classe abstraite et interface

Albrecht Zimmermann

albrecht.zimmermann@unicaen.fr

Objectifs

- La programmation objet :
  - Les classes abstraites
  - Les interfaces

Classes abstraites (1)

- **Super-classe** : **prototype commune** à plusieurs classes dérivées : permet de définir les **signatures des méthodes partagées** par l'ensemble de ces classes.
- Classe **abstraite** : une classe qui ne **permet pas d'instancier** directement d'objets :
  - On peut avoir un constructeur
  - On ne peut pas l'appeler directement : faut que ce soit une classe enfant qui l'appelle via `super`
- Sert de **base** à des **classes dérivées** → garantie que toutes les classes dérivées accèdent **toutes** à l'ensemble des méthodes définies dans la classe abstraite :
  - Soit avec des méthodes définies dans la classe
  - Soit avec des méthodes par héritage

Classes abstraites (3)

- Les classes abstraites facilitent la conception orientée objet car elle permet de **garantir** que **toutes** les classes descendantes posséderont les fonctionnalités (méthodes) requises.
- Cette certitude permet une **exploitation poussée** du polymorphisme

```
public abstract class Sequence {
   public abstract void debut(); // placement au début de la séquence
   public abstract boolean suivant(); // passe à l'élément suivant
                                      // le résultat est faux s'il n'y a pas de
   suivant
… }
public class Algo {
   …
   public void recherche(Sequence s){
         s.debut();
         do { … }
         while(s.suivant());
         …
```

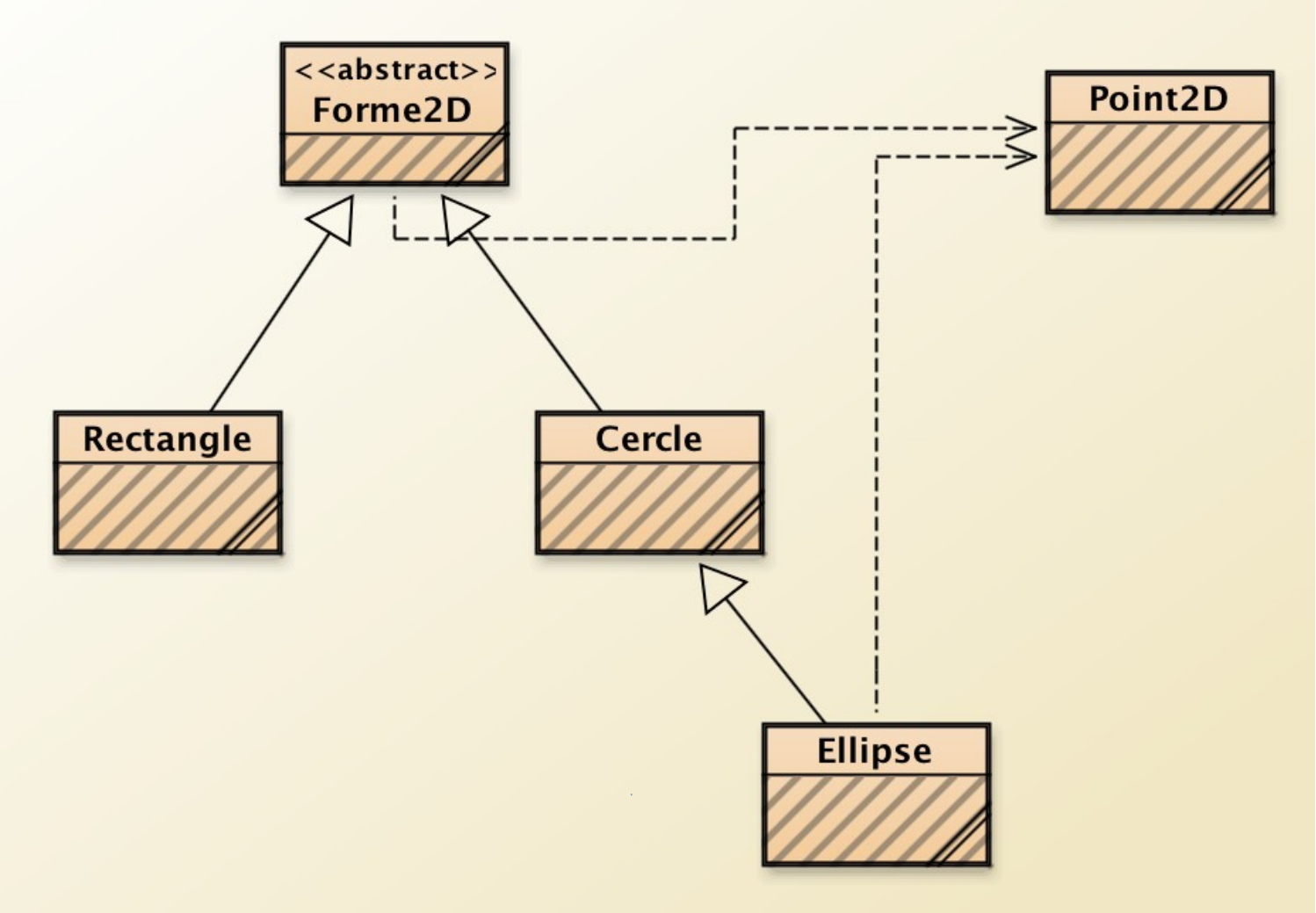
On peut passer n'importe quelle instance d'une classe enfant de la classe Sequence Et être sûr que la méthode `suivant()` existe

Un exemple

- Construire des classes pour gérer des rectangles et des cercles.
- Caractéristiques :
  - Un point **origine** pour chaque figure géométrique
  - Pouvoir **afficher** les caractéristiques de la figure
  - Pouvoir **calculer** son aire

- La figure géométrique :
  - Notion englobante
  - Pas de représentation,
  - Rassemble des caractéristiques communes aux rectangles et aux cercles

Schéma UML d'organisation des classes



Un exemple (2) – la classe abstraite

```
public abstract class Forme2D{
   // le point de référence de la forme
   Point2D origine;

   public Forme2D(Point2D p){
         origine=p;
   }

   // 2 méthodes abstraites
   public abstract double aire();
   public abstract String toString();

   // méthode d'instance
   public  void deplace (double  dx, double dy){
      origine.move(dx,dy);
   }
}
```

Impossibilité de faire :
`Forme2D f=new Forme2D(p);`

Un exemple (3)

```
class Rectangle extends Forme2D {
     protected double largeur = 0;
     protected double hauteur = 0;

     // constructeur
     public Rectangle(Point2D p, double largeur, double hauteur){
        super(p);
        this.largeur=largeur;
        this.hauteur=hauteur;
     }
     @Override
     public double aire(){
        return largeur*hauteur;
     }
     @Override
     public String toString(){
        return "Rectangle : "+ origine + " largeur="+largeur+
                  ", hauteur="+hauteur;
     }
}
```

3 attributs !

3 méthodes !

Un exemple (3)

```
class Rectangle extends Forme2D {
     protected double largeur = 0;
     protected double hauteur = 0;

     // constructeur
     public Rectangle(Point2D p, double largeur, double hauteur){
        super(p);
        this.largeur=largeur;
        this.hauteur=hauteur;
     }
    @Override
     public double aire(){
        return largeur*hauteur;
     }
     @Override
     public String toString(){
        return "Rectangle : "+ origine + " largeur="+largeur+
                  ", hauteur="+hauteur;
     }
}
```

3 attributs !

Une sous-classe non-abstraite qui dérive directement d'une classe abstraite doit redéfinir toutes les méthodes abstraites pour pouvoir être instanciée

3 méthodes !

Un exemple (3)

```
class Rectangle extends Forme2D {
      protected double largeur = 0;
      protected double hauteur = 0;

      // constructeur
      public Rectangle(Point2D p, double largeur, double hauteur){
         super(p);
         this.largeur=largeur;
         this.hauteur=hauteur;
      }
     @Override
      public double aire(){
         return largeur*hauteur;
      }
      @Override
      public String toString(){
         return "Rectangle : "+ origine + " largeur="+largeur+
                  ", hauteur="+hauteur;
      }
}
```

3 attributs !

Une sous-classe non-abstraite qui dérive directement d'une classe abstraite doit redéfinir toutes les méthodes abstraites pour pouvoir être instanciée

Sinon → `abstract`

3 méthodes !

Un exemple (4)

```
class Cercle extends Forme2D {
    double rayon;
    static final protected double PI=3.1415;

    public Cercle(Point2D p, double rayon){
        super(p);
        this.rayon=rayon;
    }
    @Override
    public double aire(){
       return PI*rayon*rayon;
    }
    @Override
    public String toString(){
       return "Cercle : "+ origine + " rayon="+rayon;
    }
}
```

2 attributs !

Exécution du programme

```
class ExForme2D{
    public static void main(String[] args){
       Rectangle r1= new Rectangle(new Point2D(1,2),3,4);
       Cercle c1= new Cercle(new Point2D(),4);

       Forme2D f [] = {c1,r1};
       for(Forme2D fi:f){
           System.out.println(fi);
       }
    }
}
```

Affichage :

```
Cercle : Point2D x=0.0, y=0.0 rayon=4.0
Rectangle : Point2D x=1.0, y=2.0 larg=3.0, hauteur=4.0
```

Exécution du programme

```
class ExForme2D{
    public static void main(String[] args){
       Rectangle r1= new Rectangle(new Point2D(1,2),3,4);
       Cercle c1= new Cercle(new Point2D(),4);

       Forme2D f [] = {c1,r1};
       for(Forme2D fi:f){
           System.out.println(fi);
       }
    }
}
```

Déclaration du tableau exploite le polymorphisme en étant d'un type qui ne peut pas directement instancié !

Affichage :

```
Cercle : Point2D x=0.0, y=0.0 rayon=4.0
Rectangle : Point2D x=1.0, y=2.0 larg=3.0, hauteur=4.0
```

Exécution du programme

```
class ExForme2D{
    public static void main(String[] args){
       Rectangle r1= new Rectangle(new Point2D(1,2),3,4);
       Cercle c1= new Cercle(new Point2D(),4);

       Forme2D f [] = {c1,r1};
       for(Forme2D fi:f){
           System.out.println(fi);
       }
    }
}
```

Déclaration du tableau exploite le polymorphisme en étant d'un type qui ne peut pas directement instancié !

Appel à .toString()

Affichage :

```
Cercle : Point2D x=0.0, y=0.0 rayon=4.0
Rectangle : Point2D x=1.0, y=2.0 larg=3.0, hauteur=4.0
```

Un exemple (5) - **Attention**

```
class Ellipse extends Cercle{
    Point2D origine2;
    double rayon2;

   public Ellipse(Point2D p1, Point2D p2, double rayon1,
                                          double rayon2){
          super(p1,rayon1);
          origine2 = p2;
          this.rayon2=rayon2;
   }
   public double aire(){
          return PI*rayon/2*rayon2/2;// A revoir
   }
}
```

Pas nécessaire de redéfinir PI

Un exemple (5) - **Attention**

```
class Ellipse extends Cercle{
    Point2D origine2;
    double rayon2;

   public Ellipse(Point2D p1, Point2D p2, double rayon1,
                                          double rayon2){
          super(p1,rayon1);
          origine2 = p2;
          this.rayon2=rayon2;
   }
   public double aire(){
          return PI*rayon/2*rayon2/2;// A revoir
   }
}
```

Pas nécessaire de redéfinir PI

La méthode `toString` n'est pas redéfinie dans la classe Ellipse. La méthode `toString` appelée sera celle de la classe `Cercle` dont la classe `Ellipse` dérive.

Exemple suite

```
class ExForme2D{
  public static void main(String[] args){
   Rectangle r1= new Rectangle(new Point2D(1,2),3,4);
   Cercle c1= new Cercle(new Point2D(),4);
   Ellipse e1= new Ellipse(new Point2D(),
                           new Point2D(4,0),1,1);
   Forme2D f [] = {c1,r1,e1};
   for(Forme2D fi:f){
       System.out.println(fi);
   }
  }
}
```

Affichage :

```
Cercle : Point2D x=0.0, y=0.0 rayon=4.0
Rectangle : Point2D x=1.0, y=2.0 larg=3.0,hauteur=4.0
Cercle : Point2D x=0.0, y=0.0 rayon=1.0
```

Interface (1)

- Java **ne permet pas** l'héritage multiple !

- Une **interface** est une liste de méthodes (prototype) qu'une classe **doit** implémenter

- Une classe abstraite peut cependant ne posséder aucun attributs → candidate à devenir une interface
- Une classe Java **peut** implémenter plusieurs interfaces
  - Mot clé **`implements`**
  - Les interfaces permettent une programmation objet exploitant massivement le polymorphisme.
  - Contourne la difficulté de l'héritage multiple des attributs avec C++

Interface (1)

- Java **ne permet pas** l'héritage multiple !

Afin d'éviter des conflits entre attributs ayant le même nom

- Une **interface** est une liste de méthodes (prototype) qu'une classe **doit** implémenter

- Une classe abstraite peut cependant ne posséder aucun attributs → candidate à devenir une interface
- Une classe Java **peut** implémenter plusieurs interfaces
  - Mot clé **`implements`**
  - Les interfaces permettent une programmation objet exploitant massivement le polymorphisme.
  - Contourne la difficulté de l'héritage multiple des attributs avec C++

Interface (1)

- Java **ne permet pas** l'héritage multiple !

Afin d'éviter des conflits entre attributs ayant le même nom

- Une **interface** est une liste de méthodes (prototype) qu'une classe **doit** implémenter

Pas d'attributs (d'instance), pas de constructeur

- Une classe abstraite peut cependant ne posséder aucun attributs → candidate à devenir une interface
- Une classe Java **peut** implémenter plusieurs interfaces
  - Mot clé **`implements`**
  - Les interfaces permettent une programmation objet exploitant massivement le polymorphisme.
  - Contourne la difficulté de l'héritage multiple des attributs avec C++

Interface (2) - Exemples d'interfaces

```
public interface Compressible {
   public void compresse();
   public void decompresse();
}

public interface Enregistrable {
   public boolean enregistre_dans(Fichier f);
   public boolean charge_a_partir(Fichier f);
}

public interface Affichable {
   public void affiche(Ecran e);
}
```

Aspects de comportement différents

Interface (3) - Implémentation

Méthodes venant de trois ! interfaces différentes

```
public class Texte
implements Compressible, Enregistrable, Affichable  {
   String message;
   public Texte() {…}
   public void ajouter_phase(String p) {…}
   public void compresse() {…}
   public void decompresse() {…}
   public boolean enregistre_dans(Fichier f) {…}
   public boolean charge_a_partir(Fichier f) {…}
   public void affiche(Ecran e){…}
}
```

Interface (4) - Exploitation

```
…
Texte texte = new Texte();
texte. ajouter_phase("Le début du texte.");
texte. ajouter_phase("…");
…
texte.compresse();
texte.enregistre_dans("nomf");
…
texte.charge_a_partir("nomf");
texte.decompresse();
texte.affiche(un_ecran);
…
```

Interface (5)

- Interface ↔ classe abstraite où **toutes** les méthodes sont abstraites
  - On ne peut pas instancier une interface (pas de new)
  - Par contre, une variable peut être de **type** interface

A partir de cette variable, seul les méthodes de l'interface considérée peuvent être utilisées.

```
Texte texte = new texte();
…
texte.affiche(un_ecran);
texte.compresse();
Compressible c = texte;
c.compresse();
c.decompresse();
c.affiche(un_ecran); // impossible
Affichable a =texte;
a.affiche(un_ecran); // ok, pour ici par-contre
```

Analogue à l'utilisation d'une variable type classe ancêtre

Interface (6)

- Une classe peut **implémenter** une ou plusieurs interfaces tout en **héritant** (`extends`) d'une seule classe (qui peut également être abstraite)
  - **S'engage** à fournir une implémentation pour **toutes** les méthodes définies dans les interfaces (et dans la classe abstraite)

javax.swing

Class AbstractButton

java.lang.Object
  └ java.awt.Component
    └ java.awt.Container
      └ javax.swing.JComponent
        └ **javax.swing.AbstractButton**

All Implemented Interfaces:
ImageObserver, ItemSelectable, MenuContainer, Serializable, SwingConstants

Direct Known Subclasses:
JButton, JMenuItem, JToggleButton

```
public abstract class AbstractButton
extends JComponent
implements ItemSelectable, SwingConstants
```

Defines common behaviors for buttons and menu items.

Buttons can be configured, and to some degree controlled, by `Actions`. Using an `Action` with a button has many benefits beyond directly configuring a button. Refer to Swing Components Supporting `Action` for more details, and you can find more information in How to Use Actions, a section in *The Java Tutorial*.

For further information see How to Use Buttons, Check Boxes, and Radio Buttons, a section in *The Java Tutorial*.

Pour information

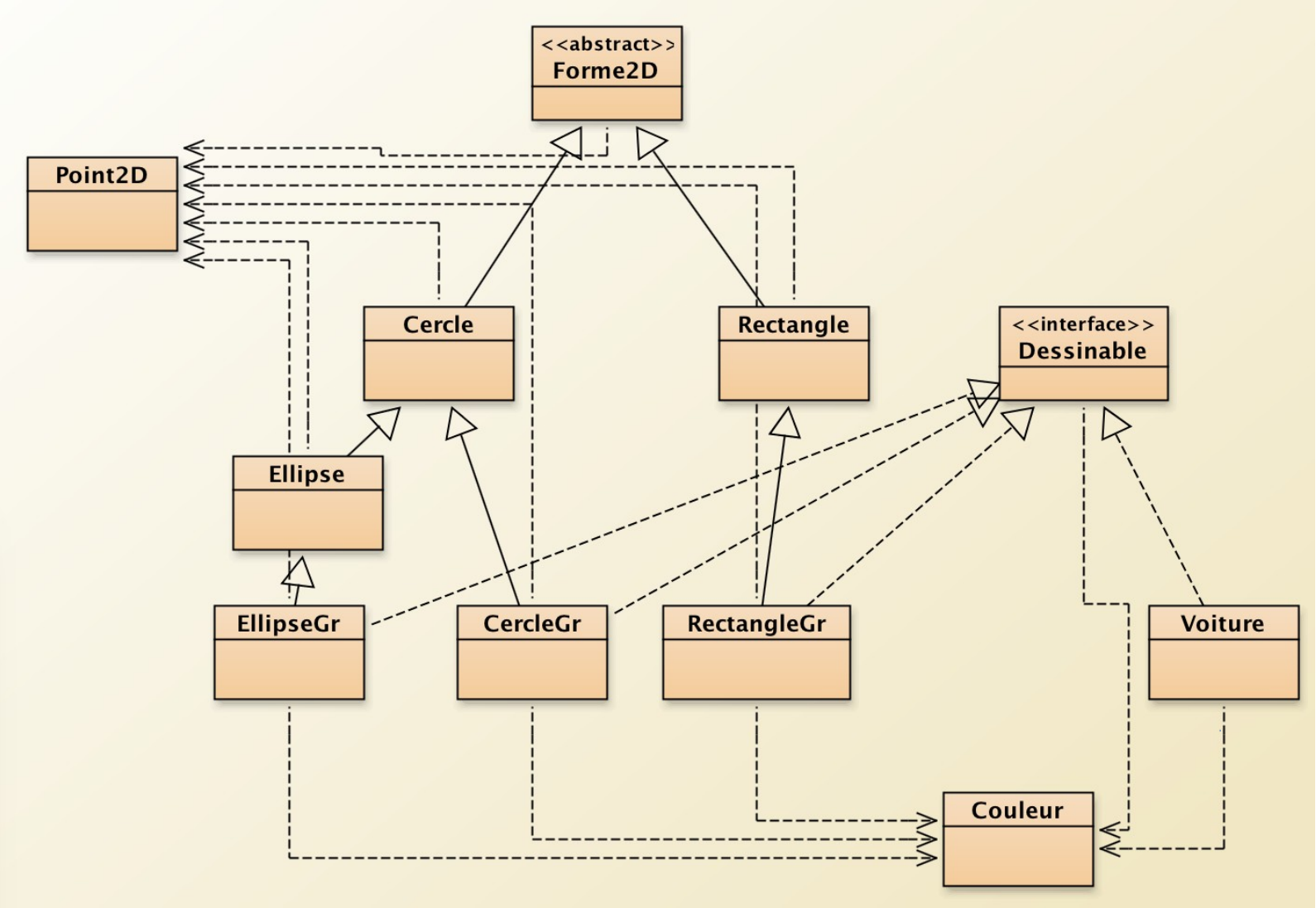
```
public abstract class JComponent
extends Container
implements Serializable
```

L'exemple des figures géométriques

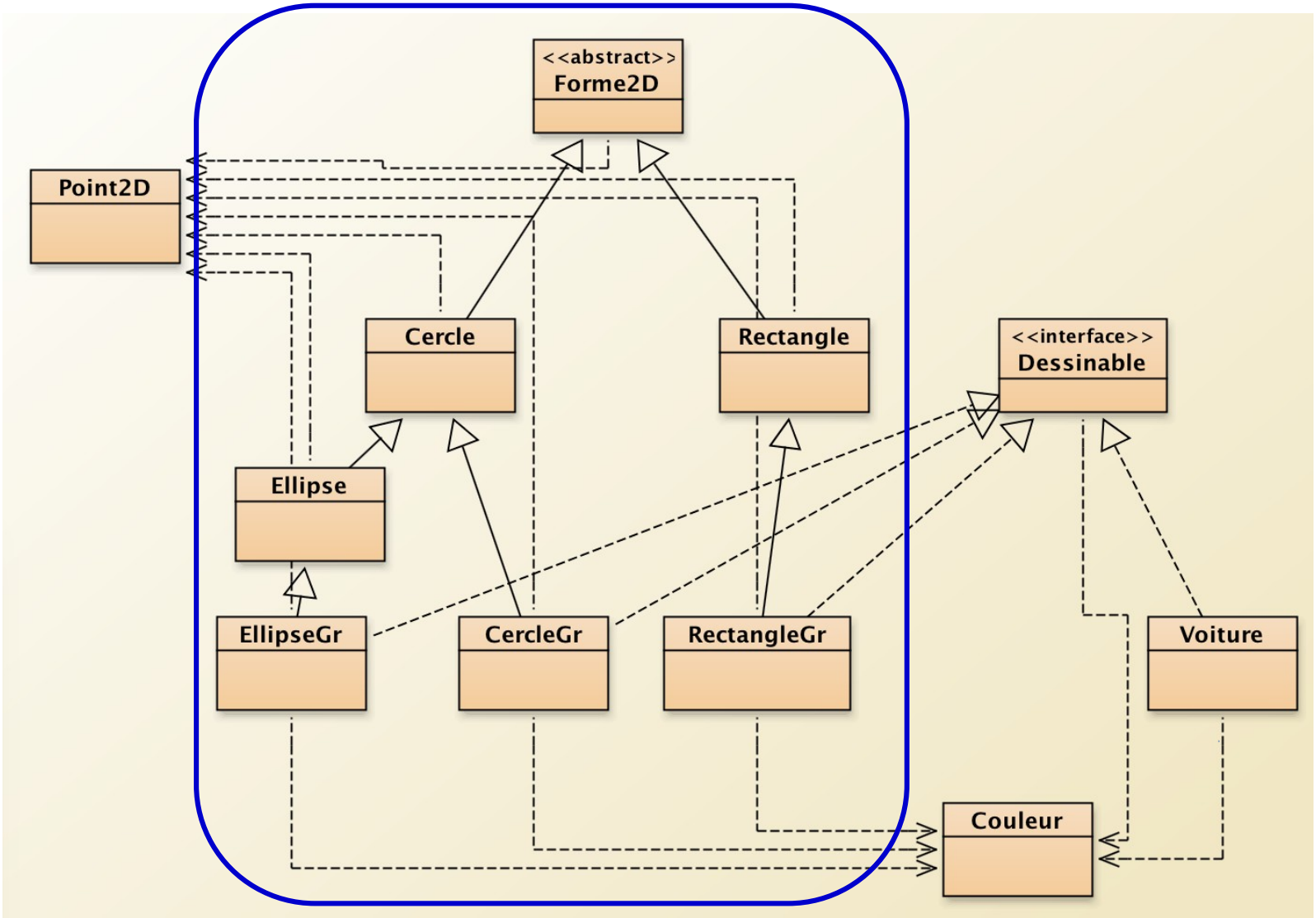
- On souhaite maintenant dessiner nos figures géométriques. Deux méthodes :
  - setCouleur
  - dessiner
- Approche :
  - Dériver les figures en créant des sous classes implantant une interface
- Avantages :
  - **Spécialisation** des classes de base qui ne changent pas
  - Garantie que **toutes** les classes implanteront **toutes** les méthodes de l'interface

L'interface peut aussi être implantée dans une autre classe. On pourra alors utiliser le polymorphisme avec toutes les classes implantant l'interface.

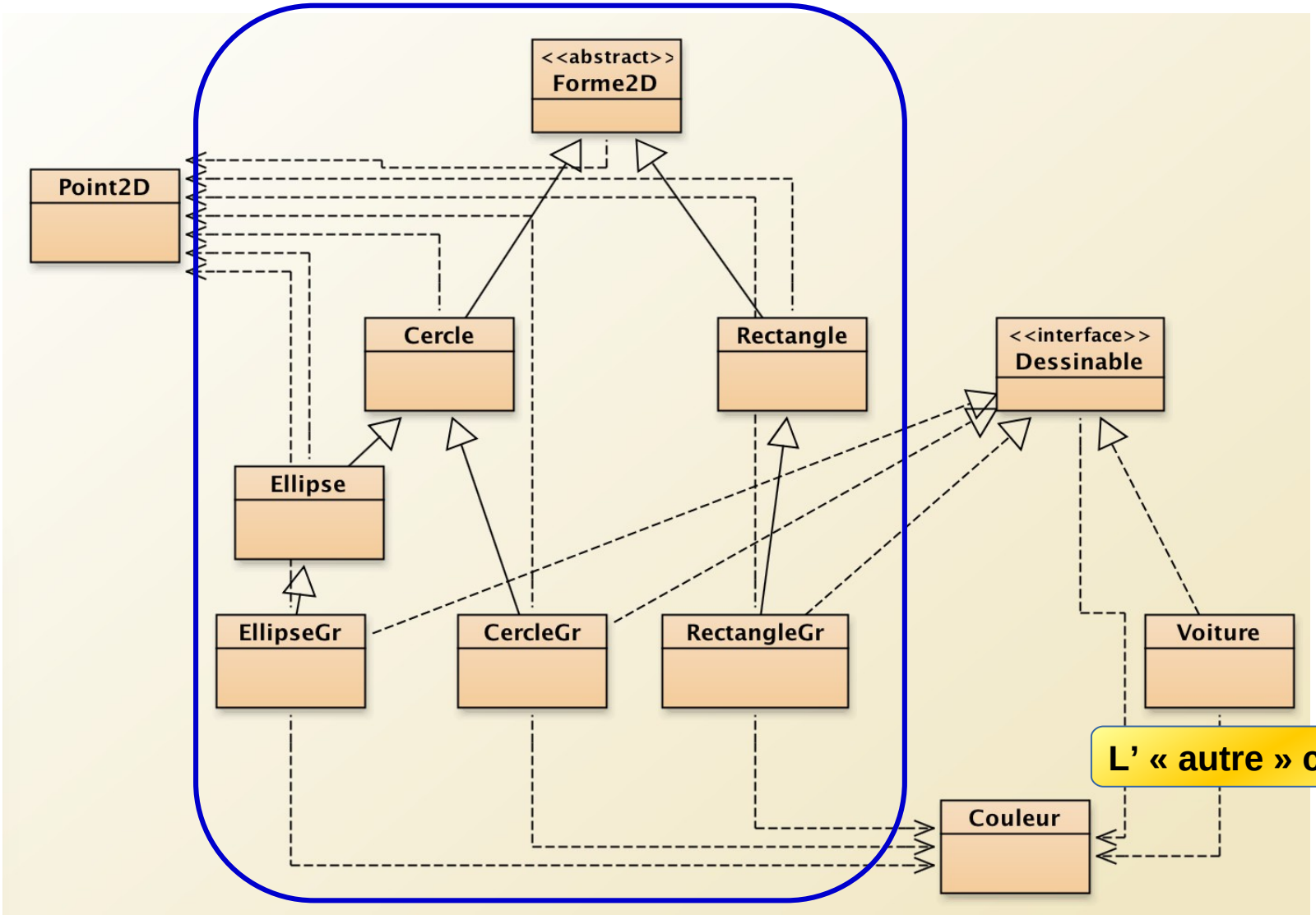
Nouveau Diagramme UML



Nouveau Diagramme UML



Nouveau Diagramme UML



Interface (7) - Dessinable.java

```
public interface Dessinable {
  public void setCouleur(Couleur c);
  public void dessiner();
}

class CercleGr extends Cercle implements Dessinable{
    Couleur c;

    public CercleGr(Point2D p, double rayon){
           super(p,rayon);
    }

    @Override
    public void setCouleur(Couleur c){
           this.c=c;
    }
    @Override
    public void dessiner(){
           System.out.println("Je dessine avec la fonction de dessin
   CERCLE \n"+this);
    }
}
```

Appel à toString()

Interface (8)

```
class RectangleGr extends Rectangle implements Dessinable{
    Couleur c=null;

    public RectangleGr(Point2D p, double largeur, double
   hauteur){
           super(p,largeur, hauteur);
    }

    @Override
    public void setCouleur(Couleur c){
          this.c=c;
    }
    @Override
    public void dessiner(){
          System.out.println("Je dessine avec la fonction"+
                             "de dessin RECTANGLE \n"+this);
    }
}
```

Interface (9) - Exploitation

```
RectangleGr r1= new RectangleGr(new Point2D(1,2),3,4);
CercleGr c1= new CercleGr(new Point2D(),4);
RectangleGr r2= new RectangleGr(new Point2D(2,4),5,6);
Dessinable[] des={r1, r2, c1};

System.out.println("\nAppel dessiner");
for(Dessinable d:des)  d.dessiner();
System.out.println("\nAppel toString");
for(Dessinable d:des)  System.out.println(d);
```

```
Appel dessiner
Je dessine avec la fonction de dessin RECTANGLE
Rectangle : Point2D x=1.0, y=2.0 largeur=3.0,   hauteur=4.0
Je dessine avec la fonction de dessin RECTANGLE
Rectangle : Point2D x=2.0, y=4.0 largeur=5.0,   hauteur=6.0
Je dessine avec la fonction de dessin CERCLE
Cercle : Point2D x=0.0, y=0.0 rayon=4.0

Appel toString
Rectangle : Point2D x=1.0, y=2.0 largeur=3.0,   hauteur=4.0
Rectangle : Point2D x=2.0, y=4.0 largeur=5.0,   hauteur=6.0
Cercle : Point2D x=0.0, y=0.0 rayon=4.0
```

Interface (9) - Exploitation

```
RectangleGr r1= new RectangleGr(new Point2D(1,2),3,4);
CercleGr c1= new CercleGr(new Point2D(),4);
RectangleGr r2= new RectangleGr(new Point2D(2,4),5,6);
Dessinable[] des={r1, r2, c1};

System.out.println("\nAppel dessiner");
for(Dessinable d:des)  d.dessiner();
System.out.println("\nAppel toString");
for(Dessinable d:des)  System.out.println(d);
```

On peut manipuler les `RectangleGr` et les `CercleGr` comme des `Dessinable` (polymorphisme)

```
Appel dessiner
Je dessine avec la fonction de dessin RECTANGLE
Rectangle : Point2D x=1.0, y=2.0 largeur=3.0,   hauteur=4.0
Je dessine avec la fonction de dessin RECTANGLE
Rectangle : Point2D x=2.0, y=4.0 largeur=5.0,   hauteur=6.0
Je dessine avec la fonction de dessin CERCLE
Cercle : Point2D x=0.0, y=0.0 rayon=4.0

Appel toString
Rectangle : Point2D x=1.0, y=2.0 largeur=3.0,   hauteur=4.0
Rectangle : Point2D x=2.0, y=4.0 largeur=5.0,   hauteur=6.0
Cercle : Point2D x=0.0, y=0.0 rayon=4.0
```

Interface (10)

Une interface peut hériter (`extends`) d'une autre interface

- les interfaces forment une hiérarchie séparée de celle des classes

Hiérarchie de comportements

```
Interface Sup {
   boolean limiteSup(int n);
   static final int MAXI = 100;
}

Interface Encadrement extends Sup{
   boolean dans(int n);
   static final int MINI = -100;
}
```

Bilan

- Les classes abstraites et interfaces
  - Les caractéristiques
  - Notion sur leur utilisation
  - Des exemples d'utilisation
  - Leur importance pour le polymorphisme