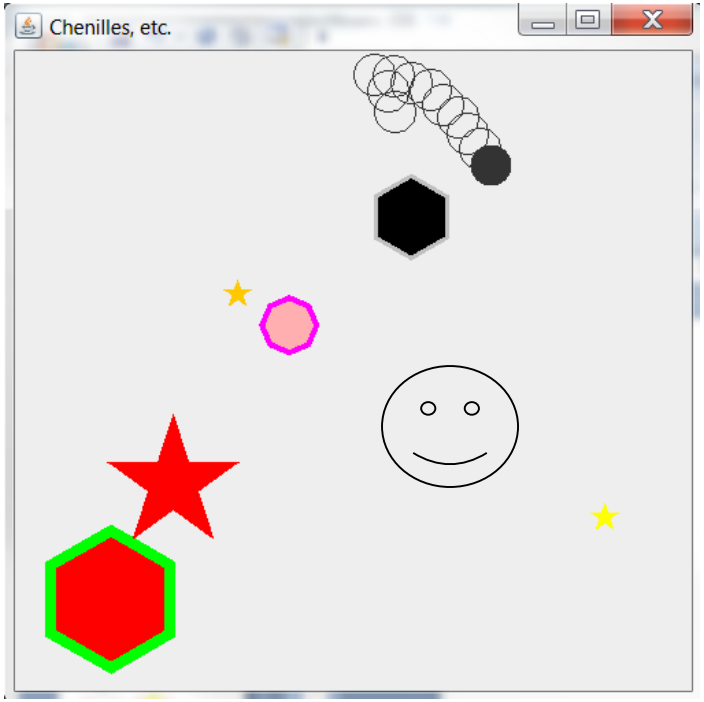
# TP n° 9 : Formes Animées

**Philippe Genoud**
**dernière mise à jour : 13/01/2017**

# Le problème

- Etendre l'application d'animation des chenilles pour afficher de nouveaux type de formes :
  - *chenilles, visages ronds, étoiles, polygone réguliers ….*



# Généralisation adaptation de Dessin

- il faut que la zone de dessin puisse afficher des objets de type autre que **Chenille**.

Il faut remplacer le type **Chenille** par un type le plus général possible.

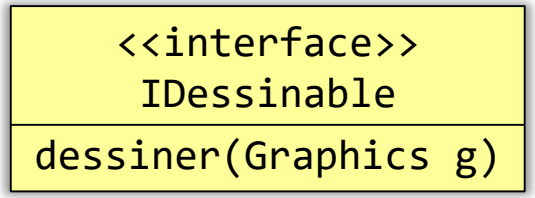
Pour assurer le plus de généralité l'idéal est de définir une **interface** : **IDessinable**

Quelles sont les opérations que doit définir cette interface (quels sont les échanges entre le dessin et un objet **IDessinalbe** ?)

```
import javax.swing.JPanel;

public class Dessin extends JPanel{

    /**
     * stocke la liste des Chenilles dans cette zone de dessin.
     */
    private final List<Chenille> lesChenilles = new ArrayList<>();

   ...
  public void ajouterObjet(Chenille ch) {

        if (!lesChenilles.contains(ch)) {
            // la chenille n'est pas déjà dans la liste
            // on la rajoute a la liste des objets du dessin
            lesChenilles.add(ch);
        }
   }
   ...
   public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // dessiner les Objets que contient le dessin
        for (Chenille ch : lesChenilles) {
            ch.dessiner(g);
        }
    }
}
```

# Généralisation adaptation de Dessin

- il faut que la zone de dessin puisse afficher des objets de type autre que **Chenille**.



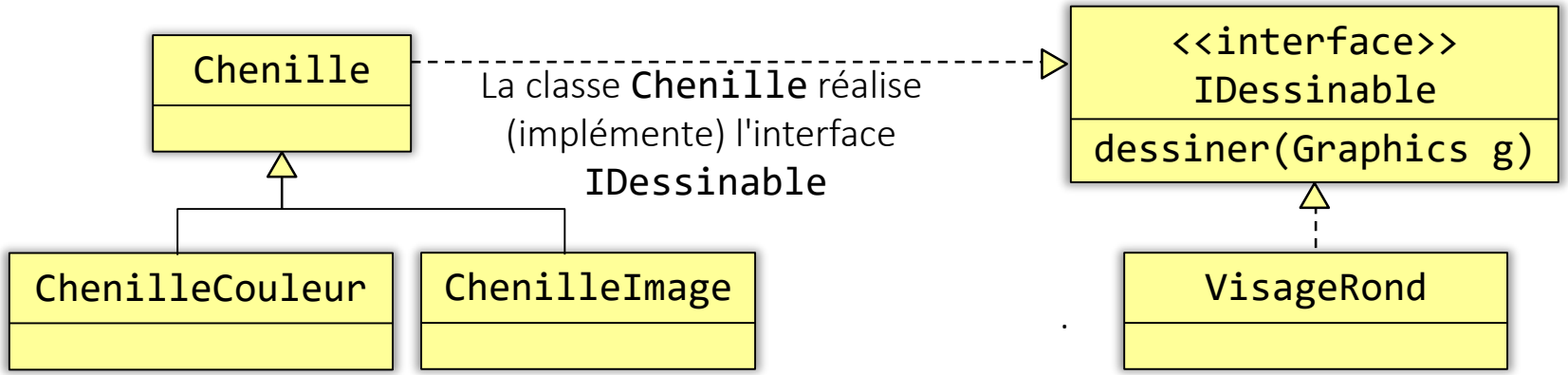

IDessinable.java

```
import java.awt.Graphics;

public interface IDessinable {

    void dessiner(Graphics g);
}
```

```
import javax.swing.JPanel;

public class Dessin extends JPanel{

    /**
     * stocke la liste des objets à dessiner.
     */
    private final List<IDessinable> lesObjets = new ArrayList<>();

   ...
  public void ajouterObjet(IDessinable obj) {

        if (!lesObjets.contains(obj)) {
            // l'objet n'est pas déjà dans la liste
            // on le rajoute a la liste des objets du dessin
            lesObjets.add(obj);
         }
   }
   ...
   public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // dessiner les Objets que contient le dessin
        for (IDessinable obj: lesObjets) {
            obj.dessiner(g);
        }
    }
}
```

# Généralisation adaptation de Chenille

- Refactoring de **Chenille** et **VisageRond** pour que l'application d'animation des chenilles puisse continuer à fonctionner




```
import java.awt.Graphics;

public class Chenille  implements IDessinable {

    @Override
    public void dessiner(Graphics g) {
         for (Anneau a : lesAnneaux) {
            a.dessiner(g);
          }
         laTete.dessiner(g);
    }
}
```

```
import java.awt.Graphics;

public class VisageRond  implements IDessinable {
    ...
    @Override
    public void dessiner(Graphics g) {
         ...
    }
}
```

```
import java.awt.Graphics;

public class ChenilleCouleur extends Chenille {
    ...
}
```

```
import java.awt.Graphics;

public class ChenilleImage extends Chenille {
    ...
}
```

# Le programme principal

```
public class AnimationFormes1 {

    public static void main(String[] args) throws IOException {
        BufferedImage imgVador = ImageIO.read(new File("images/darthVador.png"));
        BufferedImage imgLeila = ImageIO.read(new File("images/leila.png"));
        // création de la fenêtre de l'application
        JFrame laFenetre = new JFrame("Chenilles, etc.");
        laFenetre.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        laFenetre.setSize(512, 512);
        // création de la zône de dessin dans la fenêtre
        Dessin d = new Dessin();
        laFenetre.getContentPane().add(d);
        // affiche la fenêtre
        laFenetre.setVisible(true);

        // les chenilles animées
        Chenille[] lesChenilles = new Chenille[10];
        lesChenilles[0] = new ChenilleImage(d, 10, imgVador);  d.ajouterObjet(lesChenilles[0]);
        lesChenilles[1] = new ChenilleImage(d, 10, imgLeila);  d.ajouterObjet(lesChenilles[1]);
        for (int i = 2; i < 10; i++) {
            lesChenilles[i] = new ChenilleCouleur(new Color((float) Math.random(), (float) Math.random(),
                                        (float) Math.random()), d, 10, 10);
            d.ajouterObjet(lesChenilles[i]);
        }

        while (true) {
            // la zone de dessin se réaffiche
            d.repaint();
            // un temps de pause pour avoir le temps de voir le nouveau dessin
            d.pause(50);
            // fait réaliser aux chenille un déplacement élémentaire
            for (Chenille c : lesChenilles) {
                c.deplacer();
            }

        }
    }
} // AnimationFormes
```

```
VisageRond v1 = new VisageRond(d, 300, 500, 40, 60);
d.ajouterObjet(v1);
```

```
v1.deplacer()
```

# Le programme principal

```
public class AnimationFormes1 {

    public static void main(String[] args) throws IOException {
        BufferedImage imgVador = ImageIO.read(new File("images/darthVador.png"));
        BufferedImage imgLeila = ImageIO.read(new File("images/leila.png"));
        // création de la fenêtre de l'application
        JFrame laFenetre = new JFrame("Chenilles, etc.");
        laFenetre.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        laFenetre.setSize(512, 512);
        // création de la zône de dessin dans la fenêtre
        Dessin d = new Dessin();
        laFenetre.getContentPane().add(d);
        // affiche la fenêtre
        laFenetre.setVisible(true);

        // les chenilles animées
        Chenille[] lesChenilles = new Chenille[10];
        lesChenilles[0] = new ChenilleImage(d, 10, imgVador);  d.ajouterObjet(lesChenilles[0]);
        lesChenilles[1] = new ChenilleImage(d, 10, imgLeila);  d.ajouterObjet(lesChenilles[1]);
        for (int i = 2; i < 10; i++) {
            lesChenilles[i] = new ChenilleCouleur(new Color((float) Math.random(), (float) Math.random(),
                                        (float) Math.random()), d, 10, 10);
            d.ajouterObjet(lesChenilles[i]);
        }

        while (true) {
            // la zone de dessin se réaffiche
            d.repaint();
            // un temps de pause pour avoir le temps de voir le nouveau dessin
            d.pause(50);
            // fait réaliser aux chenille un déplacement élémentaire
            for (
                }
            }
        }
    }
} // AnimationFormes
```

```
VisageRond v1 = new VisageRond(d, 300, 500, 40, 60);
d.ajouterObjet(v1);
```

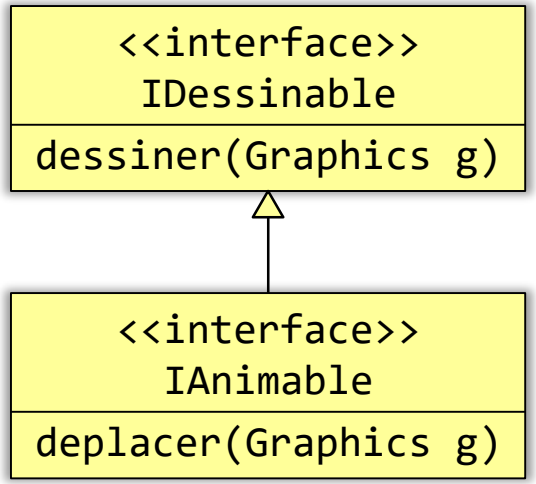
```
// fait réaliser aux objets du dessin
// un déplacement élémentaire
d.animer();
```

De la même manière que le dessin se charge de redessiner tous les objets *dessinables* qu'ils contient, ne pourrait-il pas s'occuper également de déplacer tous les objets *animables* ?

- Définir une nouvelle interface **IAnimable** sous interface de **IDessinable**

```
import javax.swing.JPanel;
import java.util.List;

public class Dessin extends JPanel{
   private final List<IDessinable> lesObjets = new ArrayList<>();

    ...
   public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // dessiner les Objets que contient le dessin
        for (IDessinable obj: lesObjets) {
            obj.dessiner(g);
        }
    }

   public void animer() {
        // deplacer les Objets animables que contient le dessin
        for (IDessinable objDessinable : lesObjets) {

             objDessinable.deplacer();

        }
   }
}
```

ne compile pas... un objet dessinable n'est pas nécessairement un objet animable




IDessinable.java

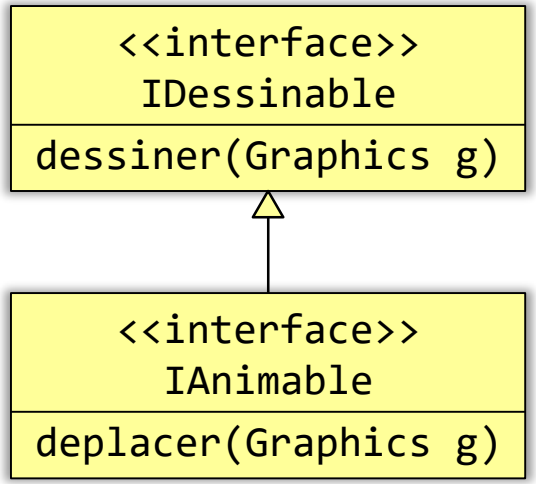
```
import java.awt.Graphics;

public interface IDessinable {
    void dessiner(Graphics g);
}
```

IAninable.java

```
import java.awt.Graphics;

public interface IAnimable
       extends IDessinable {
    void deplacer();
}
```

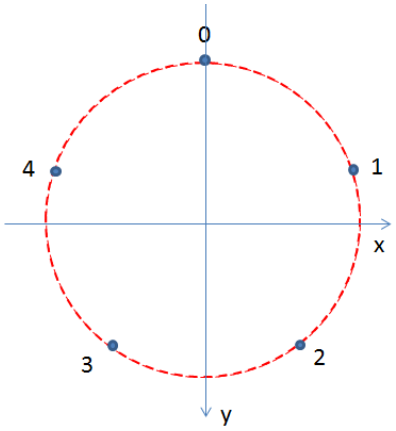
- Définir une nouvelle interface **IAnimable** sous interface de **IDessinable**

```
import javax.swing.JPanel;
import java.util.List;

public class Dessin extends JPanel{
   private final List<IDessinable> lesObjets = new ArrayList<>();

    ...
   public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // dessiner les Objets que contient le dessin
        for (IDessinable obj: lesObjets) {
            obj.dessiner(g);
       }
    }

   public void animer() {
        // deplacer les Objets animables que contient le dessin
        for (IDessinable objDessinable : lesObjets) {
               if (objDessinable instanceof IAnimable) {
                   ((IAnimable) objDessinable).deplacer();
               }
       }
   }
}
```

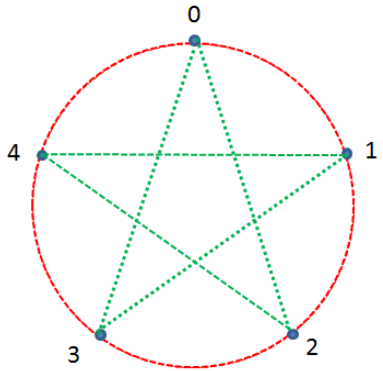



il faut vérifier que l'objet est bien animable et dans ce cas faire un *downcasting* pour pouvoir appeler la méthode **déplacer**.

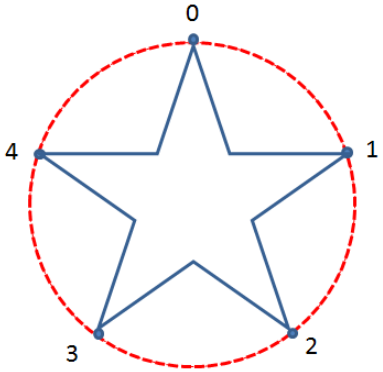
- étoiles à 5 branches



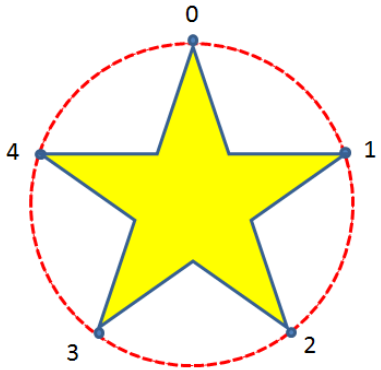

1) calcul des sommets du polygone régulier




2) construction du chemin reliant les points



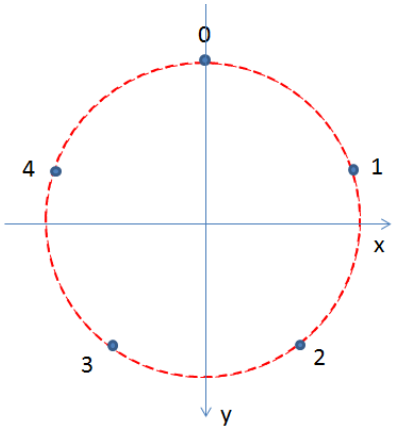

3) Dessin du contour



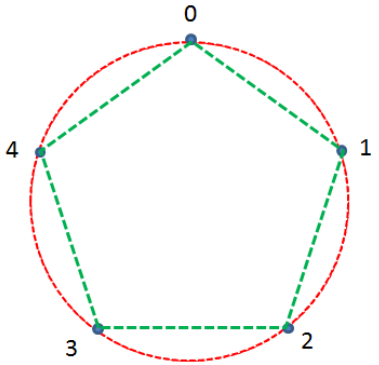

4) Remplissage de la forme

```
int nbSommets = 5;

// calcul des sommets du polygone régulier
float deltaAngle = 360f / nbSommets;
float angle = -90;
Point2D.Float[] sommets = new Point2D.Float[nbSommets];
for (int i = 0; i < nbSommets; i++) {
    sommets[i] = new Point2D.Float(
         (float) Math.cos(Math.toRadians(angle)) * r,
         (float) Math.sin(Math.toRadians(angle)) * r
    );
    angle += deltaAngle;
}

// construction du chemin reliant les points
Path2D star = new Path2D.Float();
star.moveTo(sommets[0].getX(), sommets[0].getY());
star.lineTo(sommets[2].getX(), sommets[2].getY());
star.lineTo(sommets[4].getX(), sommets[4].getY());
star.lineTo(sommets[1].getX(), sommets[1].getY());
star.lineTo(sommets[3].getX(), sommets[3].getY());
star.closePath();


// dessin à l'aide de l'objet Graphics
Graphics2D g2 = (Graphics2D) g.create();

//dessin du contour
g2.setColor(couleurTrait);
g2.setStroke(new BasicStroke(2.0f));
g2.translate(x, y);
g2.draw(star);

// remplissage
g2.setPaint(couleurRemplissage);
g2.fill(star);
```

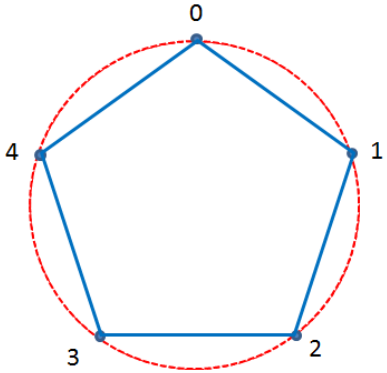
- polygones régulier (ex. pentagone)



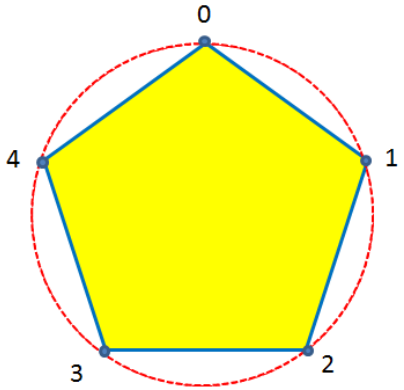

1) calcul des sommets du polygone régulier




2) construction du chemin reliant les points



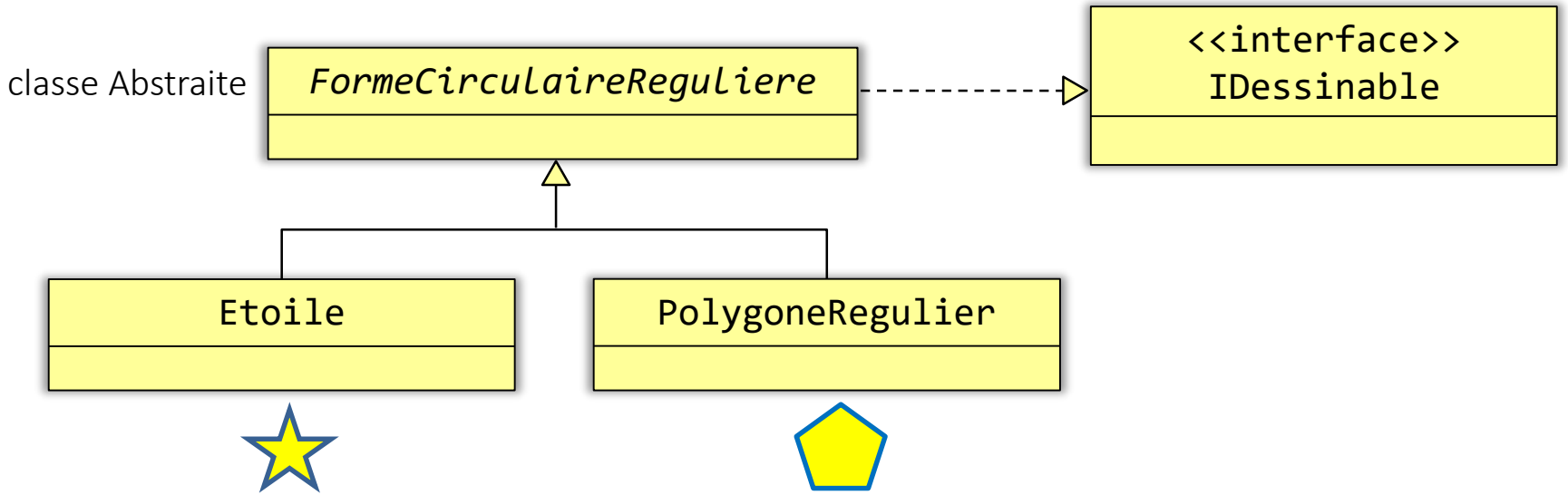

3) Dessin du contour




4) Remplissage de la forme

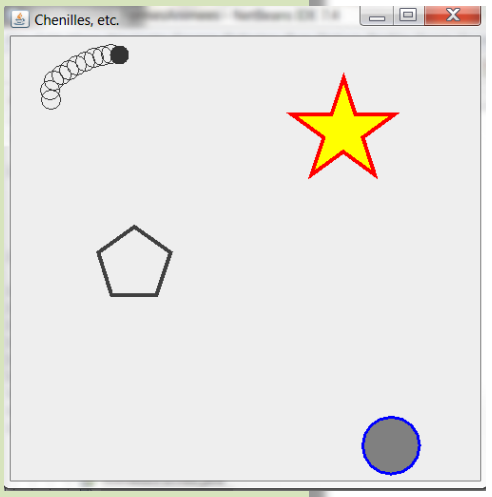
```
int nbSommets = 5;

// calcul des sommets du polygone régulier
float deltaAngle = 360f / nbSommets;
float angle = -90;
Point2D.Float[] sommets = new Point2D.Float[nbSommets];
for (int i = 0; i < nbSommets; i++) {
    sommets[i] = new Point2D.Float(
         (float) Math.cos(Math.toRadians(angle)) * r,
         (float) Math.sin(Math.toRadians(angle)) * r
    );
    angle += deltaAngle;
}

// construction du chemin reliant les points
Path2D  path = new Path2D.Float();
path.moveTo(sommets[0].getX(), sommets[0].getY());
for (int i = 1; i < nbSommets; i++) {
   path.lineTo(sommets[i].getX(), sommets[i].getY());
}
path.closePath();


// dessin à l'aide de l'objet Graphics
Graphics2D g2 = (Graphics2D) g.create();

//dessin du contour
g2.setColor(couleurTrait);
g2.setStroke(new BasicStroke(2.0f));
g2.translate(x, y);
g2.draw(star);

// remplissage
g2.setPaint(couleurRemplissage);
g2.fill(star);
```

- polygones réguliers et étoiles à 5 branches partagent de nombreuses caractéristiques :
  - *ces sont des formes inscrites dans un cercle, dont le contour a les sommets répartis régulièrement sur le cercle circonscrit.*
- Comment partager ces caractéristiques au niveau du code ?
  - *En généralisant les concepts Etoile et Polygone Régulier*

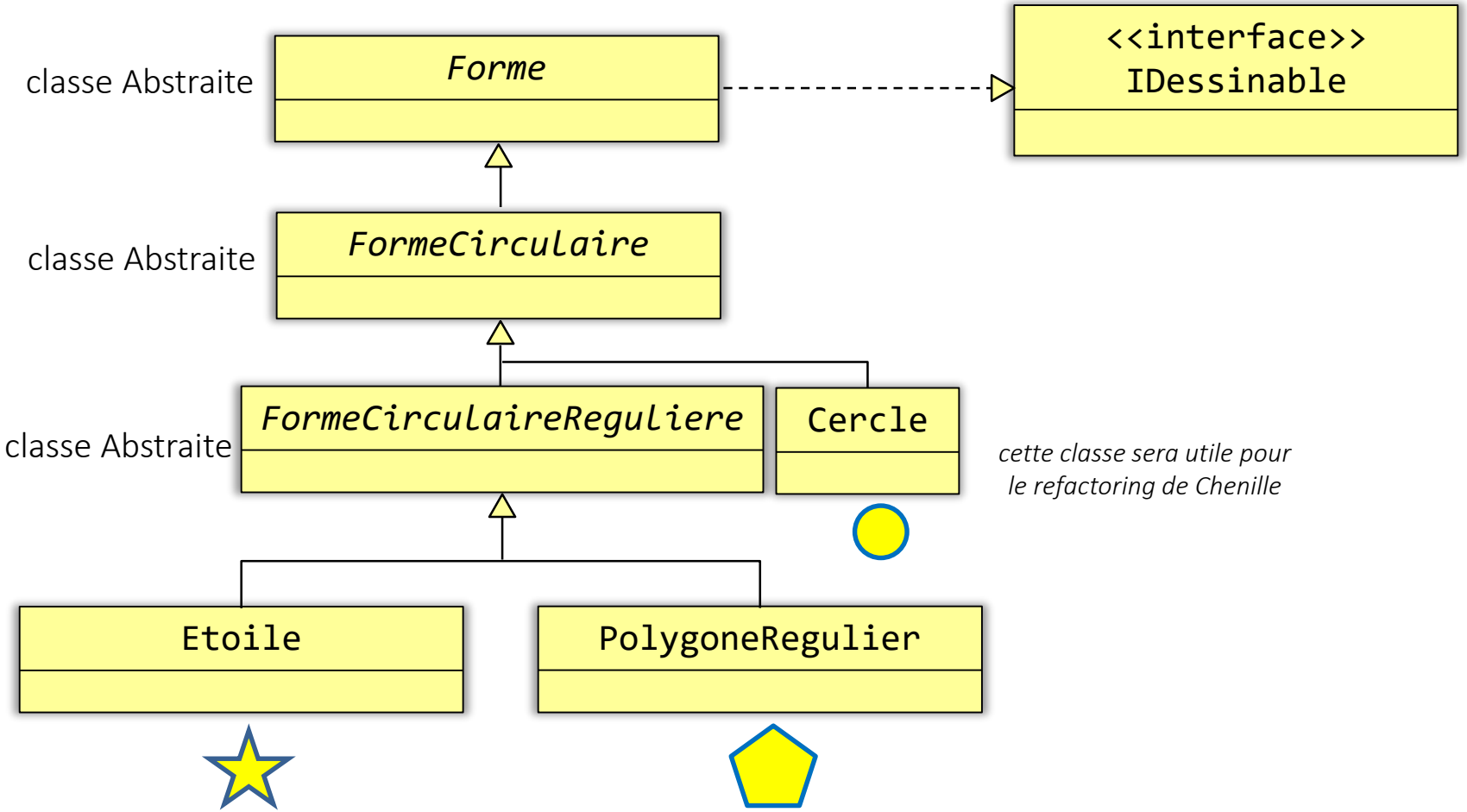


# Le programme principal

```
public class AnimationFormes {

    public static void main(String[] args) {

        // création de la fenêtre de l'application
        JFrame laFenetre = new JFrame("Chenilles, etc.");
        laFenetre.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        laFenetre.setSize(512, 512);
        // création de la zône de dessin dans la fenêtre
        Dessin d = new Dessin();
        laFenetre.getContentPane().add(d);
        // affiche la fenêtre
        laFenetre.setVisible(true);

        // les objets fixes de la zone de dessin
        d.ajouterObjet(new Etoile(350, 100, 50, 8.f, Color.RED, Color.YELLOW));
        d.ajouterObjet(new PolygoneRegulier(130, 240, 40, 5, 4.0f, Color.DARK_GRAY, null));
        d.ajouterObjet(new Cercle(400, 430, 30, 3.0f, Color.BLUE, Color.GRAY));
        // une chenille animée
         d.ajouterObjet(new Chenille(d,10,10));
        // un visage rond
        d.ajouterObjet(new VisageRond(d,10,10));

        while (true) {
            // la zone de dessin se réaffiche
            d.repaint();
            // un temps de pause pour avoir le temps de voir le nouveau dessin
            d.pause(50);
            // fait un déplacement élémentaire aux objets animables
            d.animer();
        }
    }
} // AnimationFormes
```



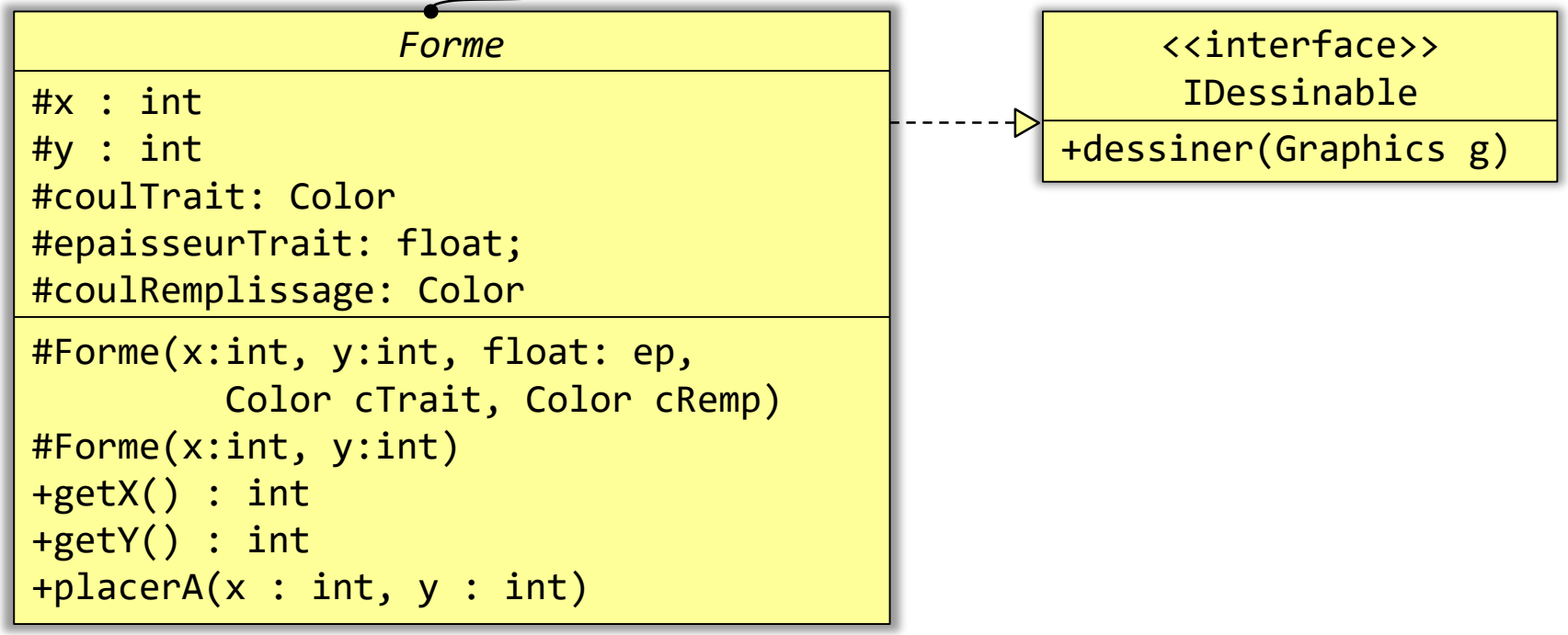
- La généralisation des concepts `Etoile` et `PolygoneRegulier` en `FormeCirculaire` peut être poussée plus loin pour offrir plus de généralité et faciliter des évolutions futures du logiciel



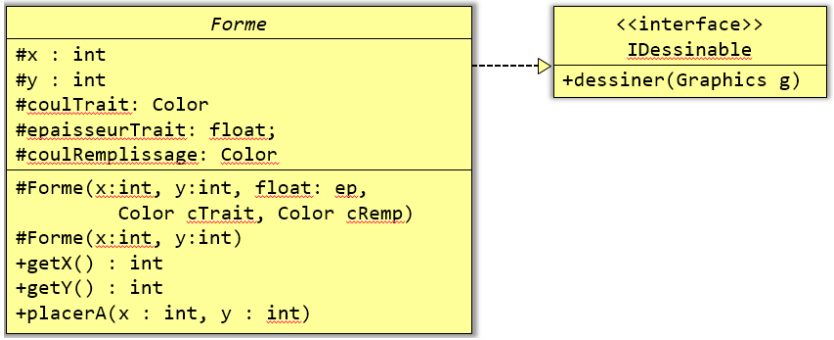
Forme : un objet dessinable défini par un point de référence (qui permet de positionner la forme dans le plan)

Visibilité des membres
- + public
- # protected
- - private
- (rien) : package

La classe est abstraite elle n'implémente pas la méthode `dessiner(Graphics g)`



La classe est abstraite elle n'implémente pas la méthode `dessiner(Graphics g)`

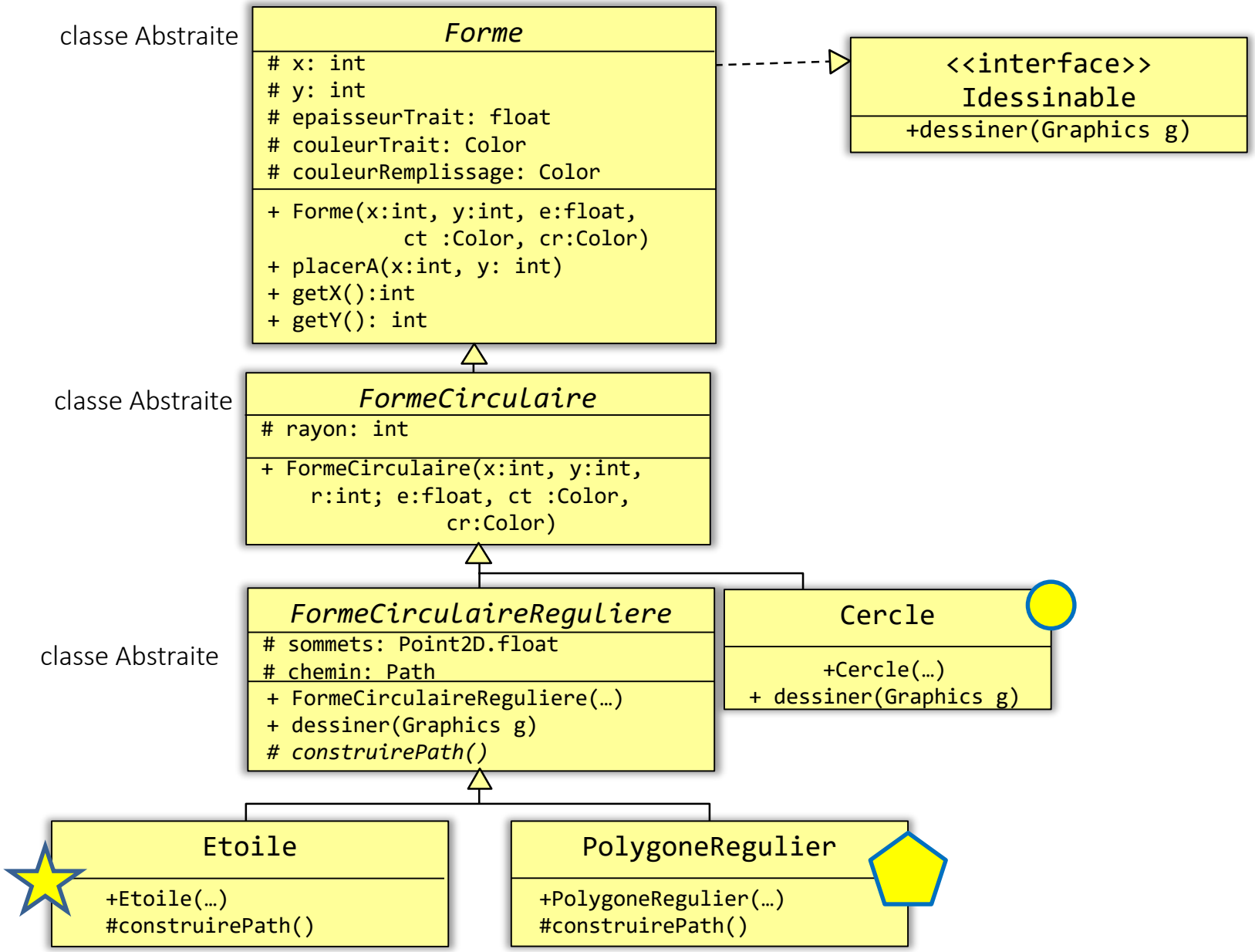



Forme.java

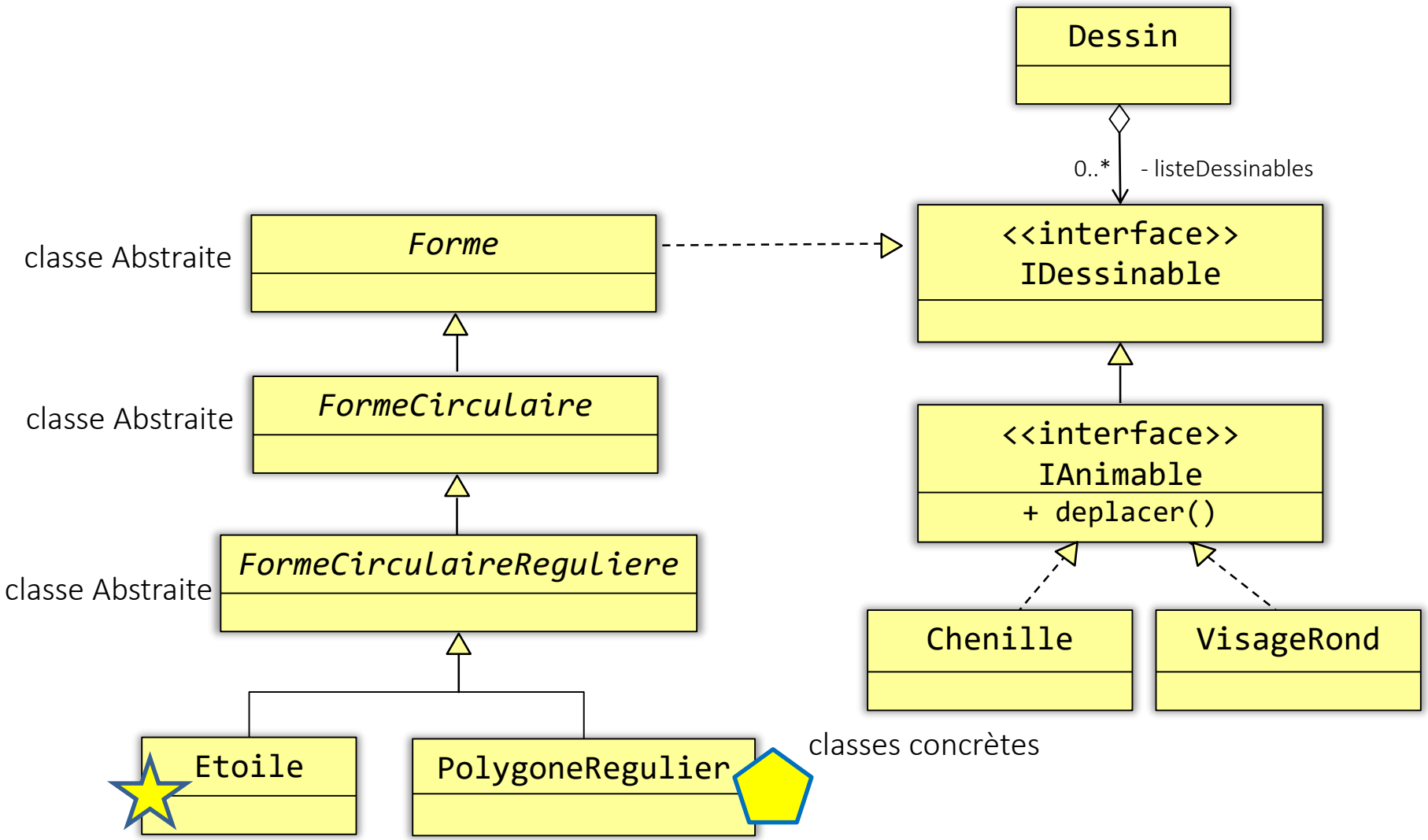
```
import java.awt.Color;
import java.awt.Rectangle

public abstract Forme implements IDessinable {
    protected int x;
    protected int y;

    protected float epaisseur = 1.0f;
    protected Color coulTrait = null;
    protected Color coulRemp = null;

    protected Forme(int x, int y, float ep,
                Color cTrait, Color cRemp) {
        this.x = x;
        this.y = y;
        this.epaisseur = ep;
        this.coulTrait = cTrait;
        this.coulRemp = cRemp;
    }

    protected Forme(int x, int y) {
        this(x,y,1.0f,null,null);
    }
```

```
    @Override
    public void placerA(int px, int py) {
        x = px;
        y = py;
    }

    @Override
    public int getX() {
        return x;
    }

    @Override
    public int getY() {
        return y;
    }

}  // Forme
```

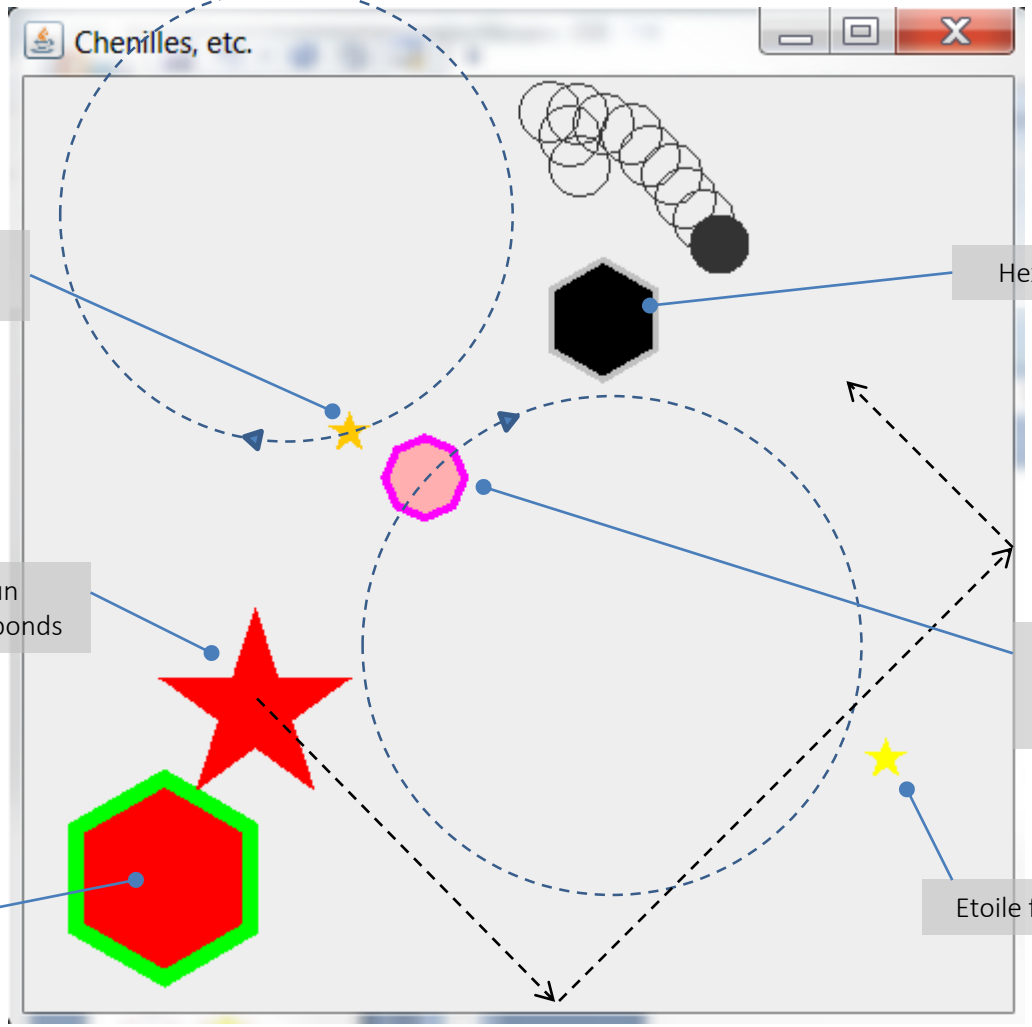
# Généralisation de la notion de forme



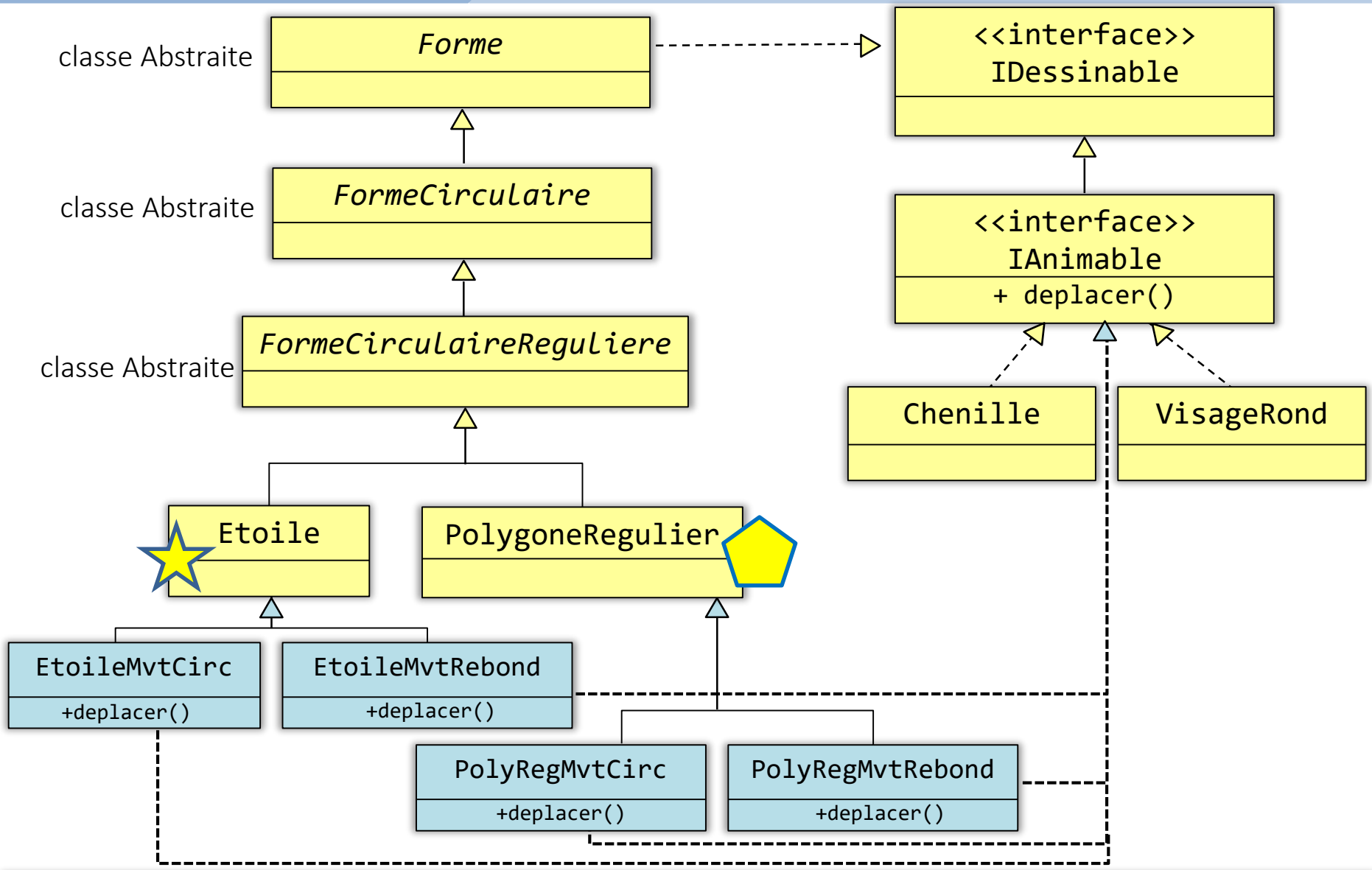
# Animer certaines formes



# Animer certaines formes

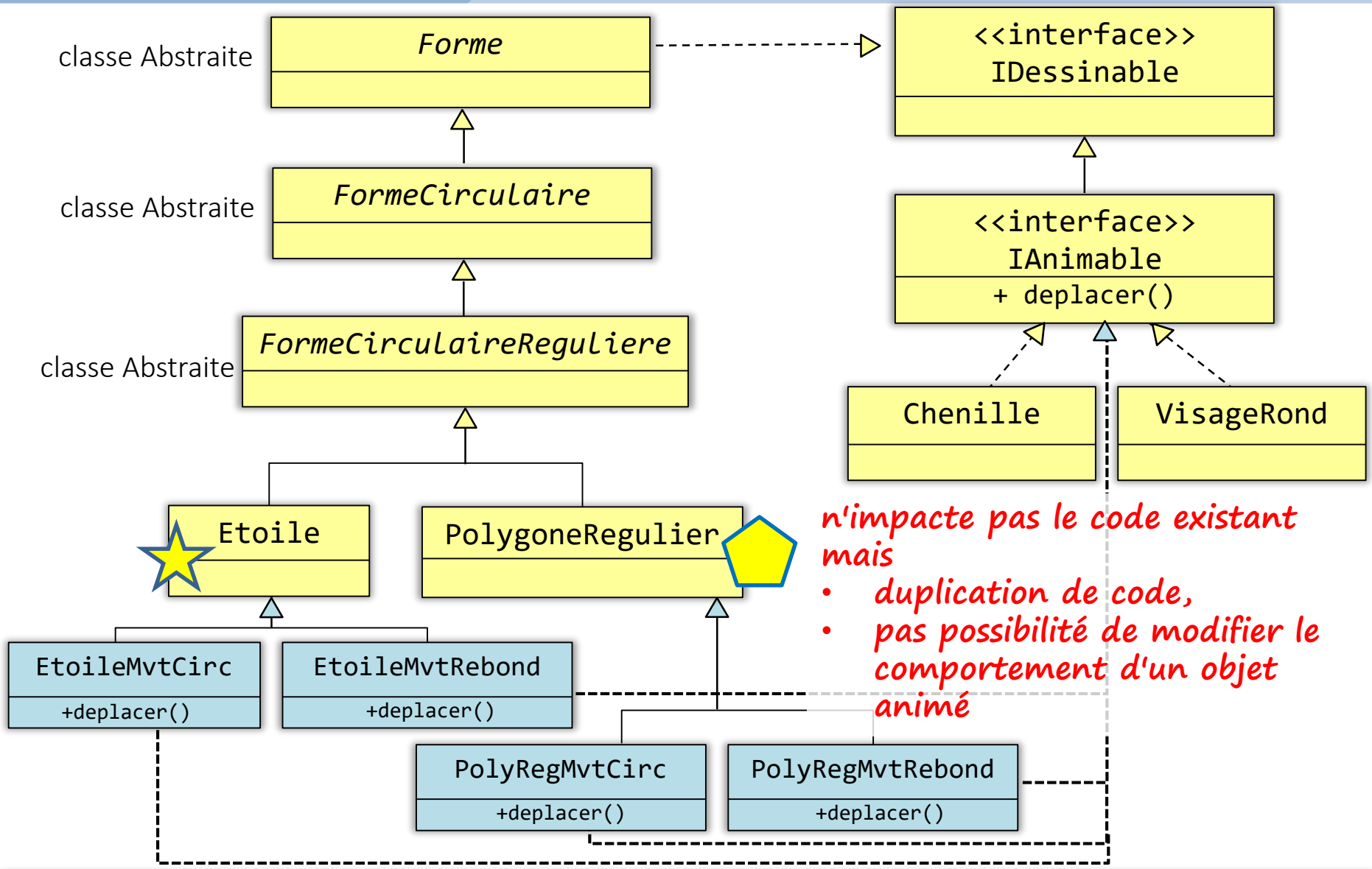


# Animer certaines formes




1ère solution: sous classer les différents types de formes en les dotant d'un comportement d'animation

# Animer certaines formes

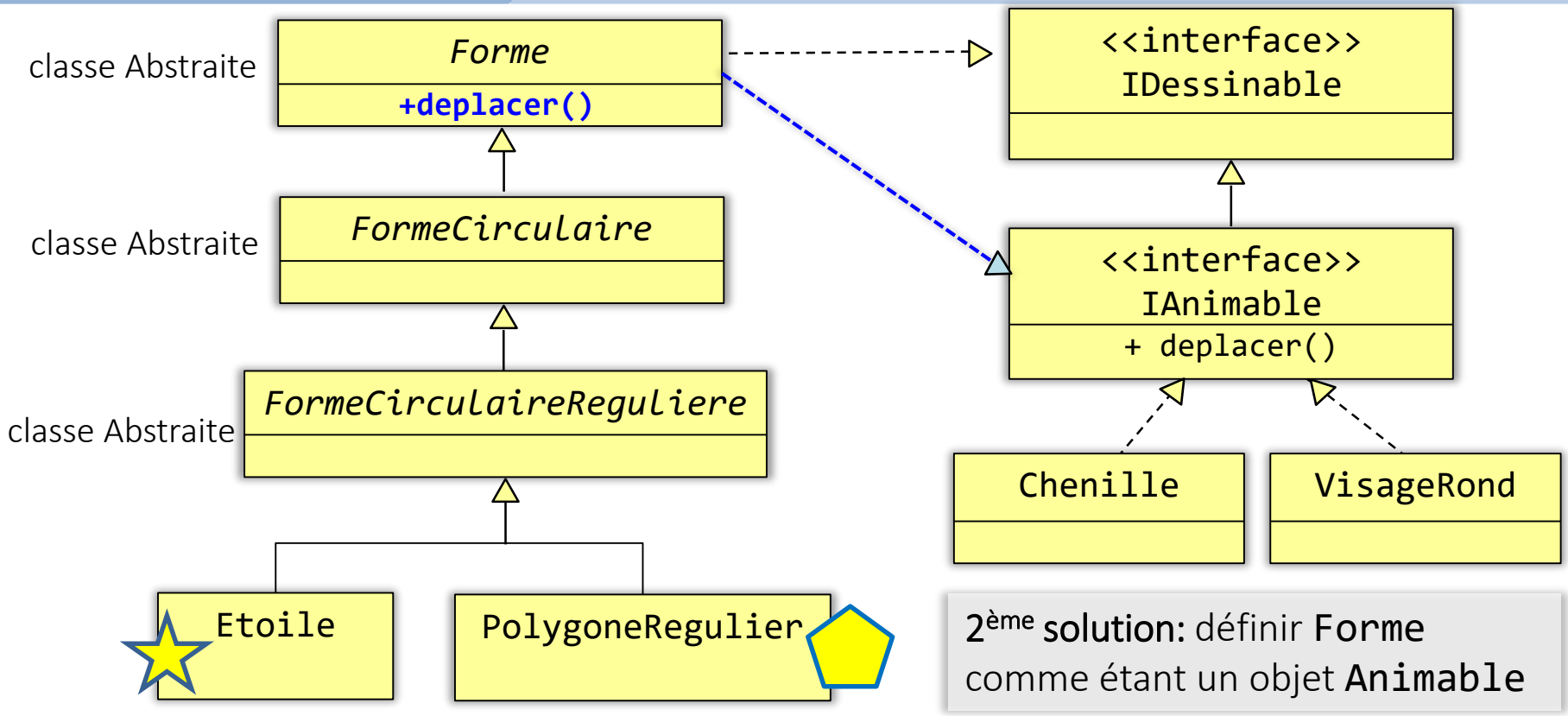



n'impacte pas le code existant mais

- duplication de code,
- pas possibilité de modifier le comportement d'un objet animé

1ère solution: sous classer les différents types de formes en les dotant d'un comportement d'animation

# Animer certaines formes

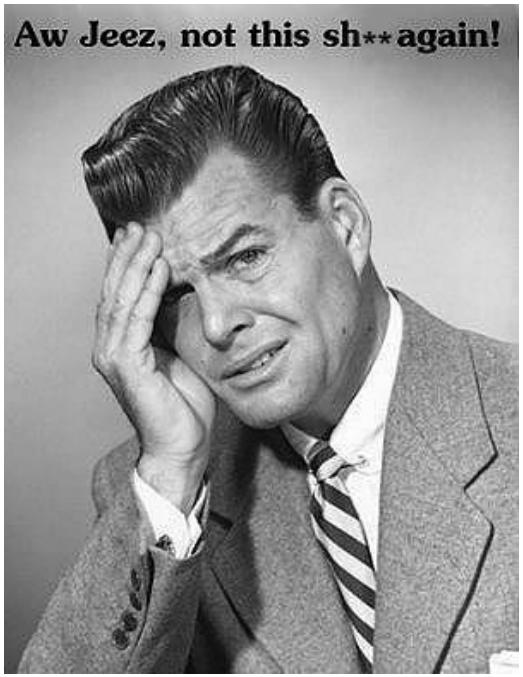



2ème solution: définir **Forme** comme étant un objet **Animable**

- ajouter à **Forme** des attributs correspondant à chaque type d'animation
- pour permettre de définir le type d'animation:
  - ajouter des constructeurs avec les paramètres correspondant à chaque type d'animation
  - ou bien proposer des méthodes permettant de les fixer.
- déplacer doit intégrer les différents cas correspondant aux différentes stratégies d'animation

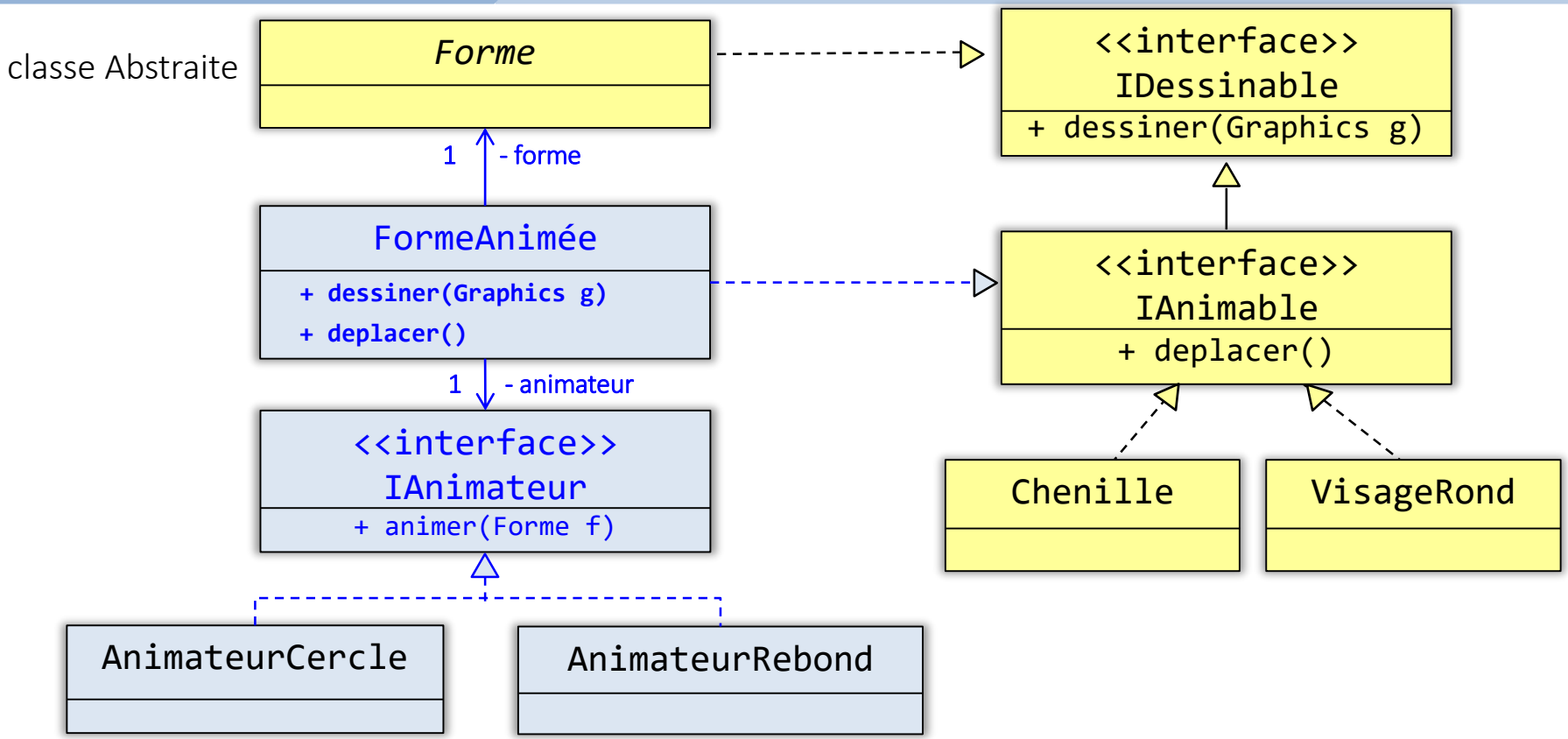
*Solution lourde, peu flexible et extensible, impact possible sur le code existant*

- Java, du fait de l'absence d'héritage multiple n'offre-t-il donc pas de bonne solution ?




*Non, Java permet de réponde de manière élégante à ce problème mais la solution ne passe pas par l'héritage mais par la délégation et par une bonne utilisation des interfaces*

# Animer certaines formes




- Une **FormeAnimée** est **Animable** (et par conséquent **Dessinable**)
  - pour se dessiner elle délègue l'opération dessiner à une instance de la classe **Forme**

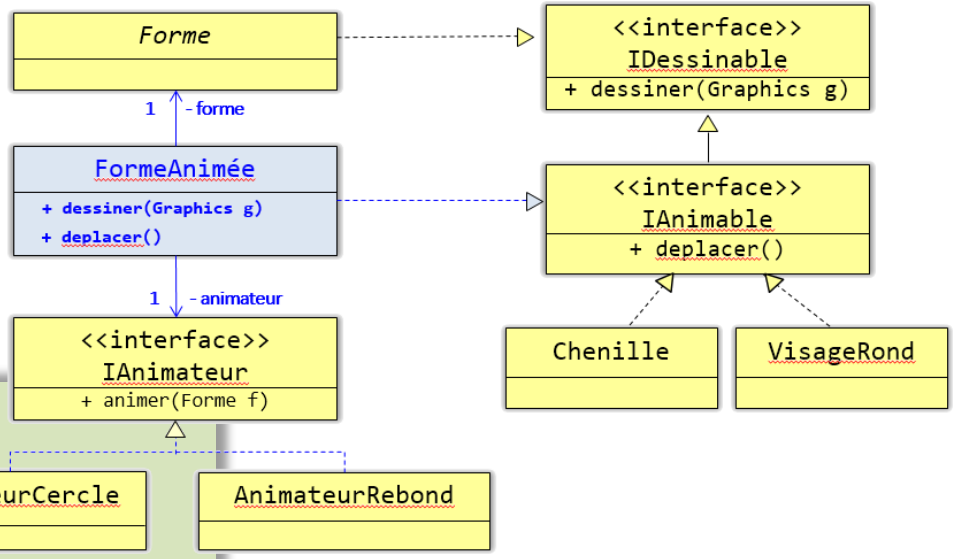
```
forme.dessiner(g);
```

  - pour se déplacer (calculer la nouvelle position x,y de sa forme) elle délègue l'opération à un objet **Animateur**

```
animateur.animer(this.forme);
```

- Les comportements d'animation seront implémentés dans différentes classes d'**Animateur**

# la classe FormeAnimee




FormeAnimee.java

```
public class FormeAnimee implements IAnimable {

    protected Forme forme;
    protected IFormeAnimator animator;

    public FormeAnimee(Forme forme, IFormeAnimator animator) {
        this.forme = forme;
        this.animator = animator;
    }

    @Override
    public void deplacer() {
        animator.deplacer(forme);
    }

    @Override
    public void dessiner(Graphics g) {
        forme.dessiner(g);
    }

}
```

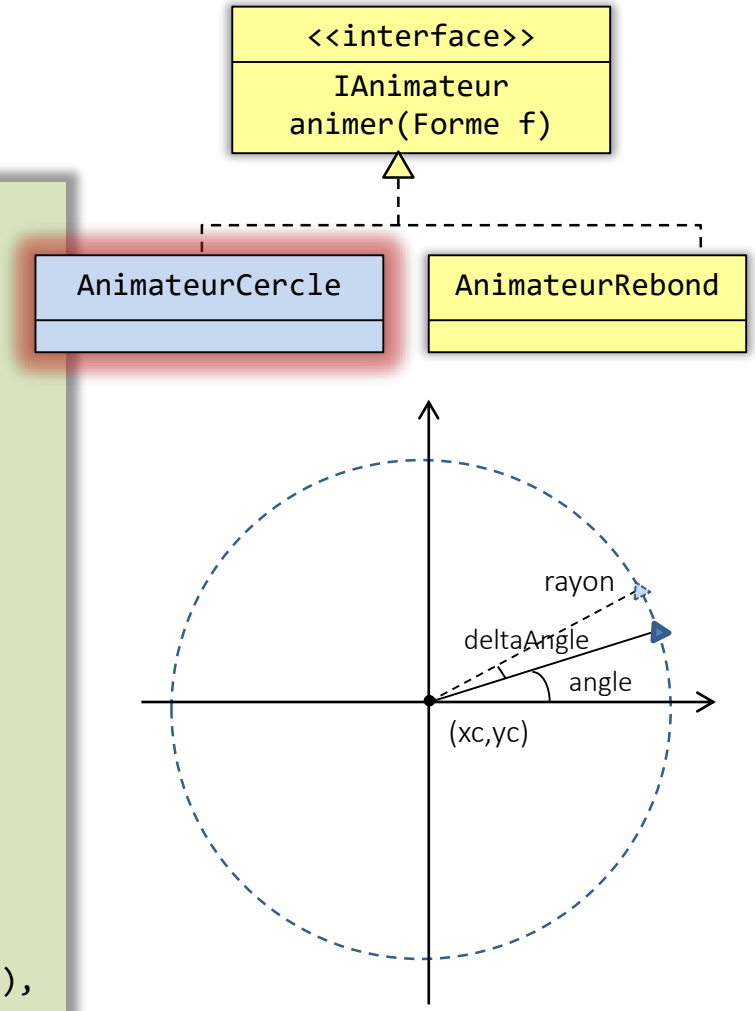
délégation de l'animation à l'animateur

délégation du dessin à la forme

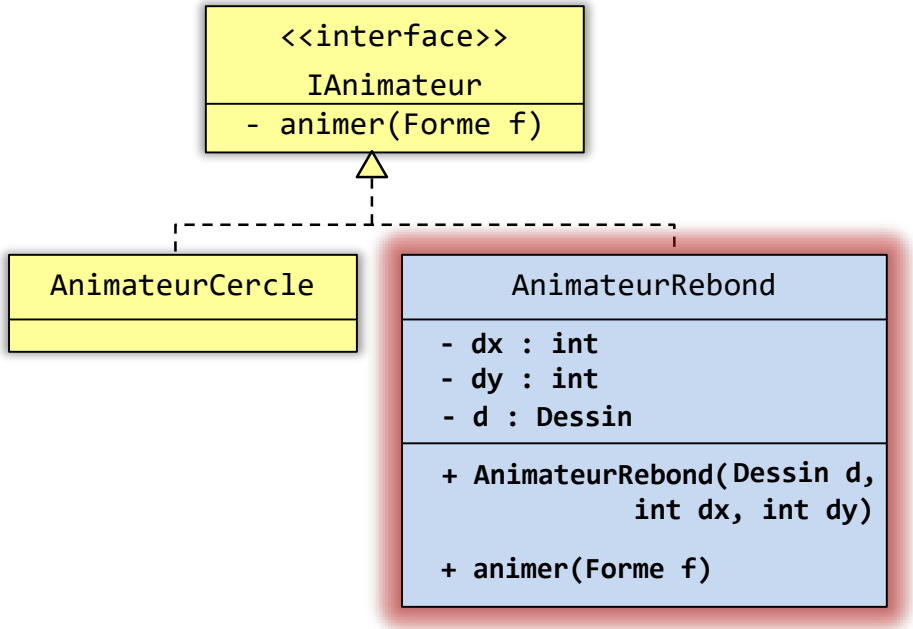
# La classe AnimateurCercle

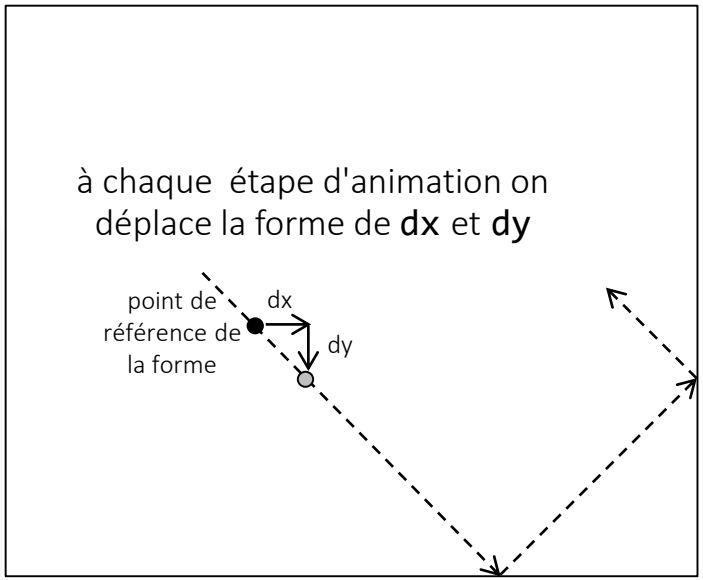
AnimateurCercle.java

```
public class AnimateurCercle implements IAnimateur {
    private int rayon;
    private int xc;
    private int yc;
    private double angle;
    private double deltaAngle;

     public AnimateurCercle(int xc, int yc, int r,
                        double angle,double deltaAngle ) {
        this.deltaAngle = deltaAngle;
        this.angle = angle;
        this.rayon = r;
        this.xc = xc;
        this.yc = yc;
    }

    @Override
    public void animer(Forme f) {
        angle += deltaAngle;
        double angleRadians = Math.toRadians(angle);
        f.placerA((int) (xc + rayon * Math.cos(angleRadians)),
                (int) (yc + rayon * Math.sin(angleRadians)));
    }

}
```



# La classe AnimateurRebond




rebond lorsque la forme sort de la zone de dessin on change le signe de **dx** ou **dy**

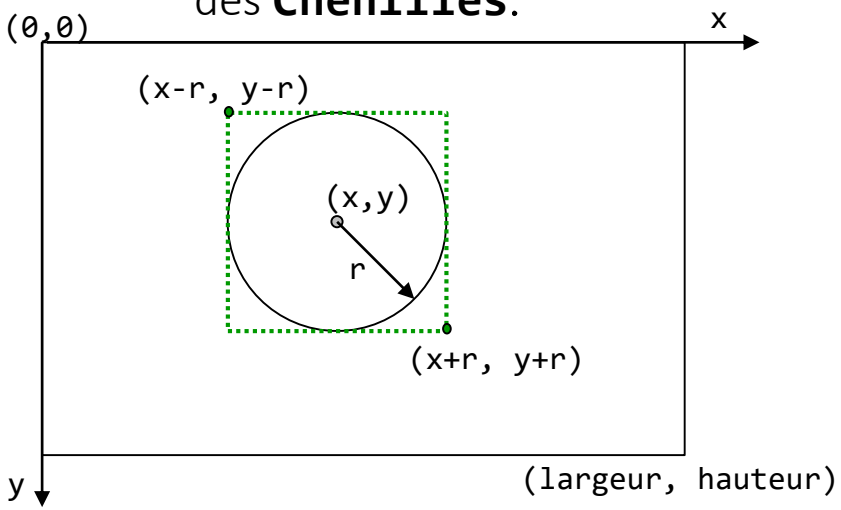
```
void animer(Forme f) {

    if (f sort à gauche de d || f sort à droite de d) {
       dx = -dx;
    }
    if (f sort en haut de d || f sort en bas de d) {
       dy = -dy;
    }
    int newx = f.getX() + dx;
    int newy = f.getY() + dy;

    f.placerA(newX,newY);
}
```

comment savoir que la forme sort de la zone dessin ?

# La classe AnimateurRebond

## comment savoir que la forme sort de la zone dessin ?

regardons ce qui a été fait pour les **VisageRond** ou les **Tête** des **Chenilles**.



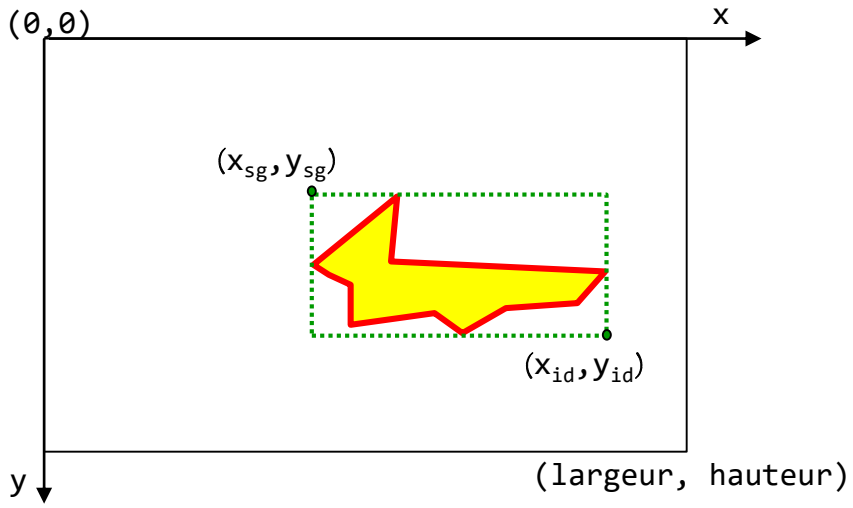

sort à gauche `x - r < 0` || sort à droite `x + r > dessin.getLargeur()`

||

sort en haut `y - r < 0` || sort en bas `y + r > dessin.getHauteur()`

ce test fonctionne pour tout objet inscrit dans un cercle, cela marcherait pour les étoiles ou les polygones réguliers mais quid d'une forme quelconque ?

généralisons cela pour supporter n'importe quel type de **Forme**.



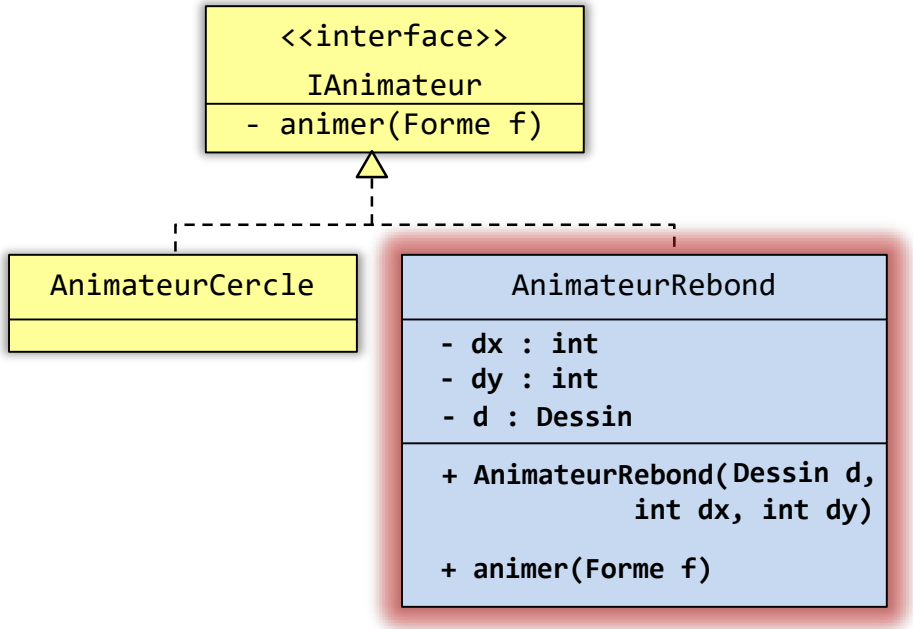

sort à gauche $x_{sg} < 0$ || sort à droite $x_{id} >$ `dessin.getLargeur()`

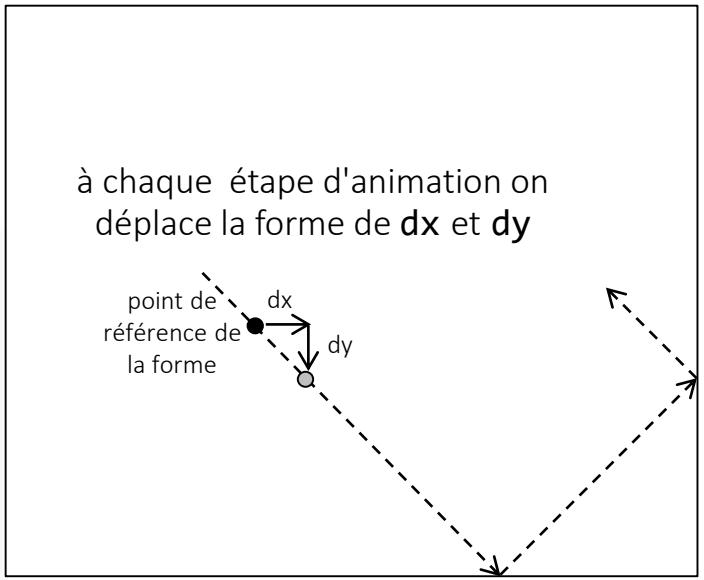
||

sort en haut $y_{sg} < 0$ || sort en bas $y_{id} >$ `dessin.getHauteur()`

comme pour une forme circulaire on se sert des coordonnées du coin supérieur gauche $(x_{sg}, y_{sg})$ et du coin inférieur droit $(x_{id}, y_{id})$ du rectangle englobant la forme

# La classe AnimateurRebond




rebond lorsque la forme sort de la zone de dessin on change le signe de **dx** ou **dy**

*il suffit que les formes puissent donner leur rectangle englobant*

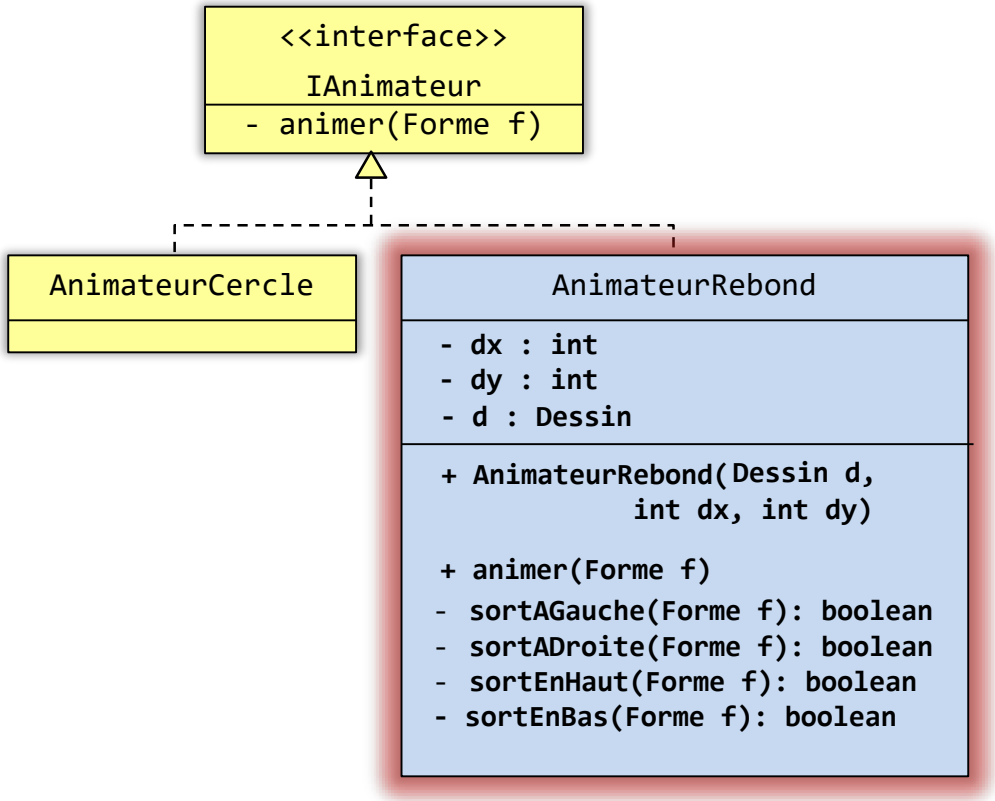
```
void animer(Forme f) {

    if (   sortAGauche(f)    ||    sortADroite(f)    ) {
       dx = -dx;
    }
    if (   sortEnHaut(f)    ||    sortEnBas(f)    ) {
       dy = -dy;
    }
    int newx = f.getX() + dx;
    int newy = f.getY() + dy;

    f.placerA(newX,newY);
}
```

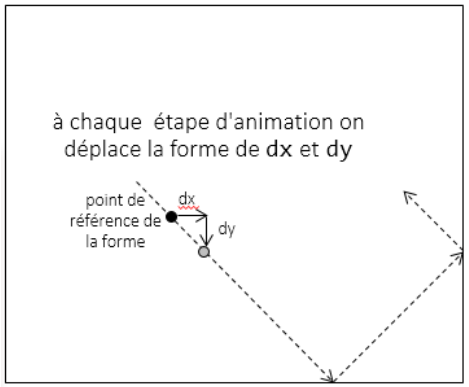
```
private  boolean sortAGauche(Forme f) {
   Rectangle rect = f.getRectEnglobant();
   return rect.getX() < 0;
}
```

```
private  boolean sortADroite(Forme f) {
   Rectangle rect = f.getBounds();
   return rect.getX() + rect.getWidth() > d.getLargeur();
}
```

...

# La classe AnimateurRebond



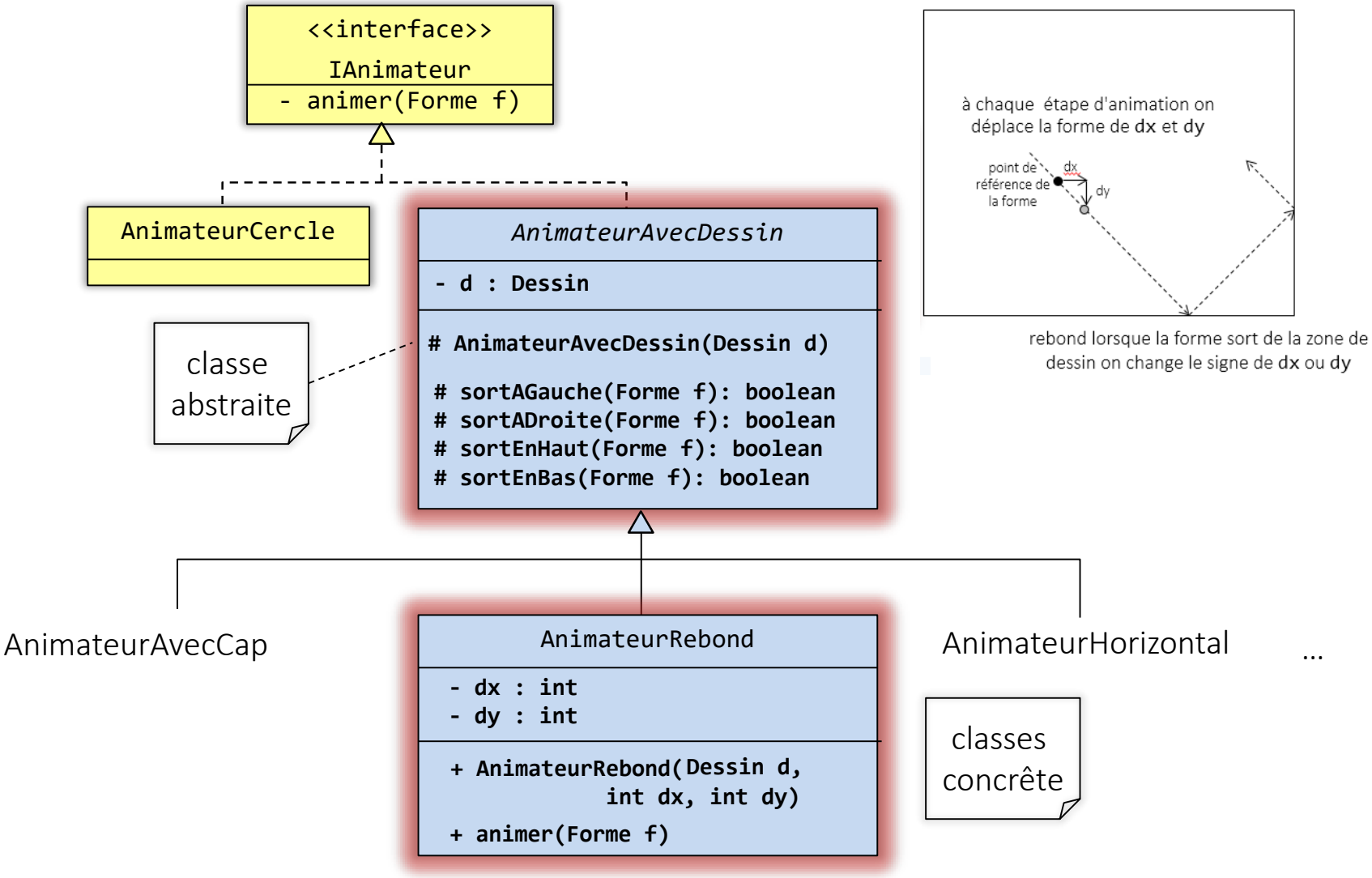




rebond lorsque la forme sort de la zone de dessin on change le signe de dx ou dy

D'autres animateur que l'**AnimateurRebond** peuvent avoir besoin de connaître la position de la forme par rapport à la zone de dessin.

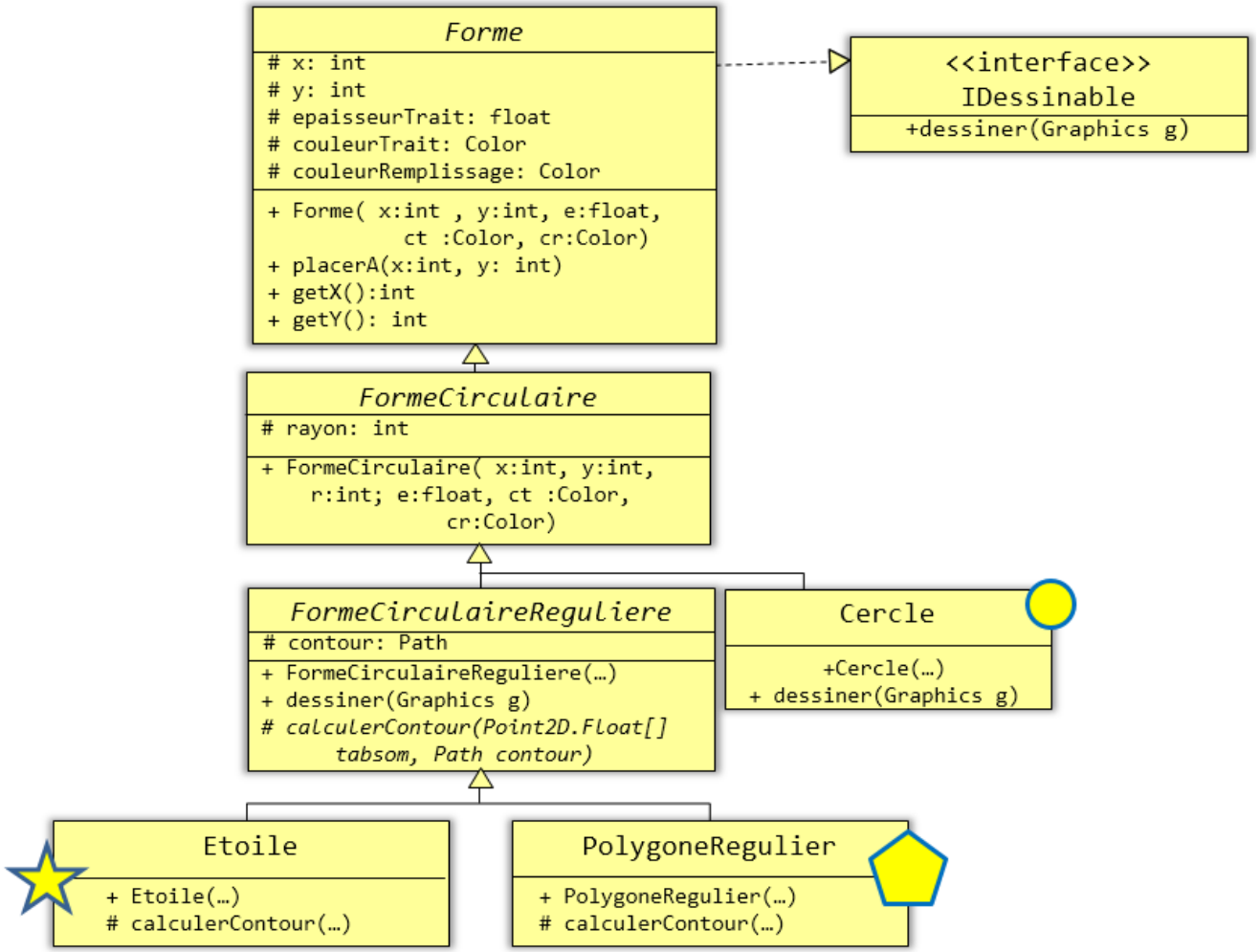
→ factoriser ce code pour éviter de le réécrire pour d'autres types d'animateurs

# La classe AnimateurRebond



# Formes et rectangle englobant

- comment gérer le rectangle englobant au niveau des formes ?



# Formes et rectangle englobant

- comment gérer le rectangle englobant au niveau des formes ?

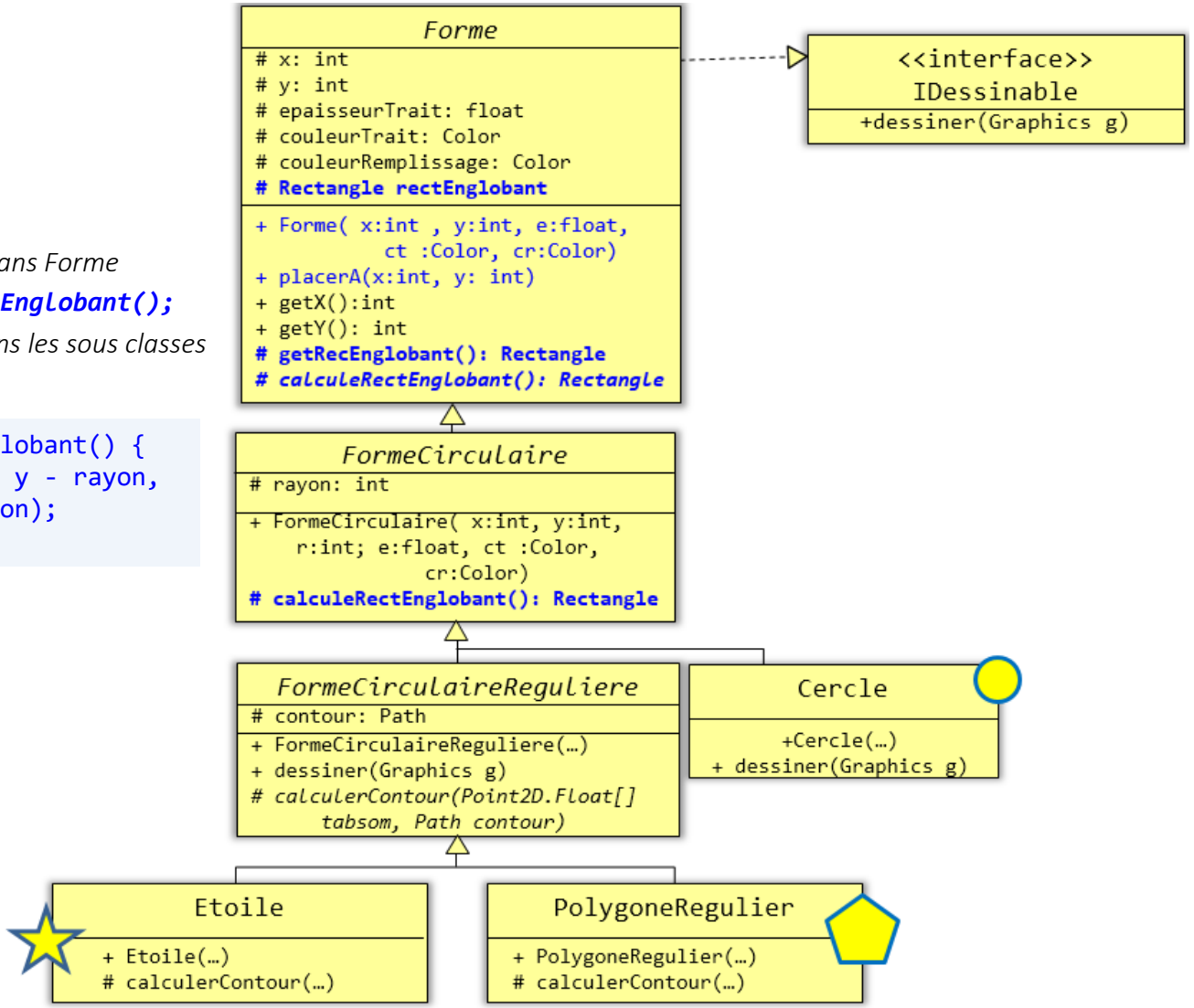
- solution 1:
  - *définir une méthode abstraite dans Forme*

```
public abstract Rectangle getRectEnglobant();
```

  - *implémenter cette méthode dans les sous classes*

```
public final Rectangle getRectEnglobant() {
  return new Rectangle(x - rayon, y - rayon,
                       2*rayon, 2*rayon);
}
```

+ simplicité
- calcul du rectangle englobant à la volée peut être coûteux



# Formes et rectangle englobant

- comment gérer le rectangle englobant au niveau des formes ?

- solution 2:
  - *définir le rectangle englobant comme attribut de Forme*

```
protected Rectangle rectEnglobant;
```

*- le rectangle englobant doit être initialisé dans le constructeur*

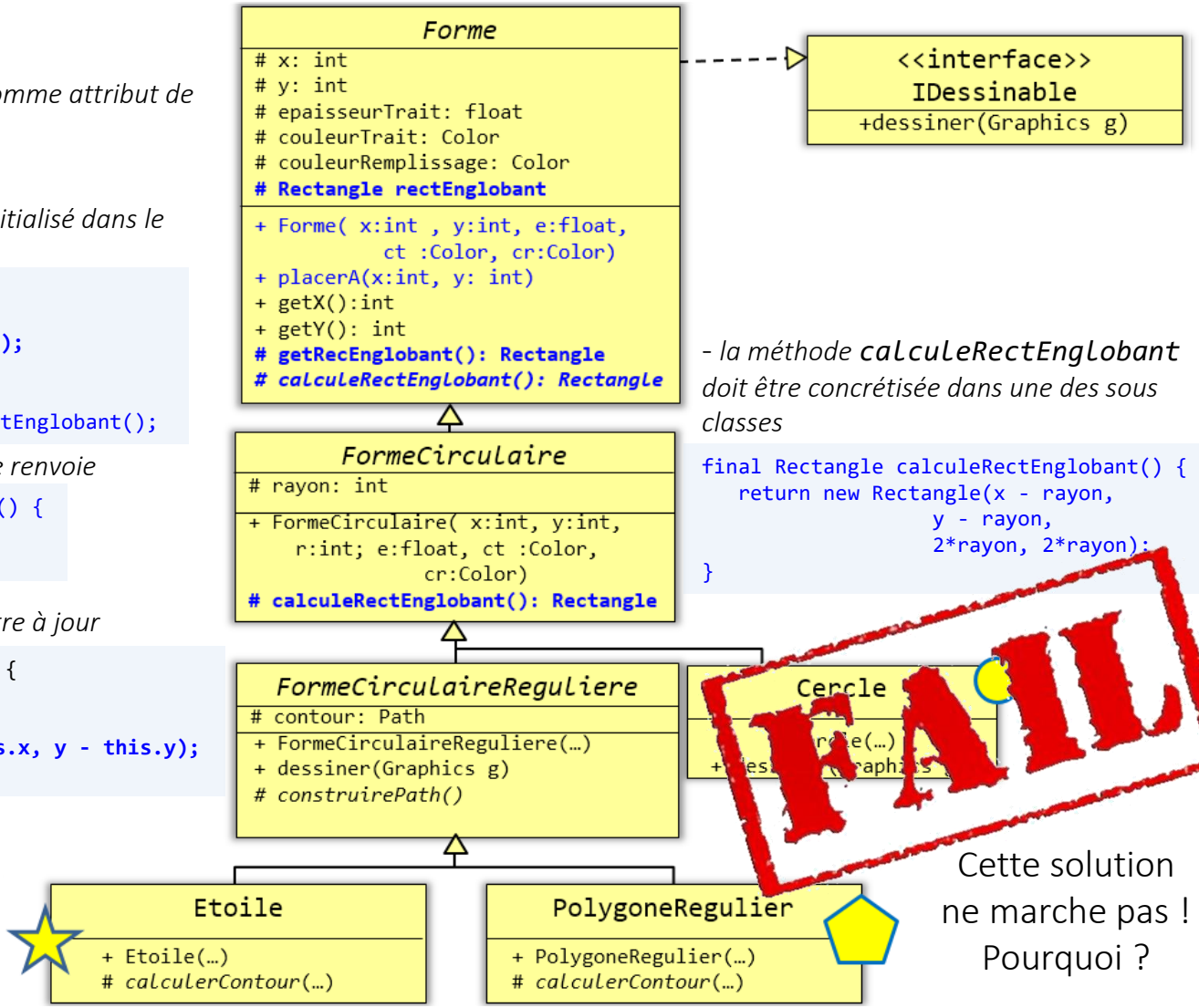
```
protected Forme(….) {
   ...
   rectEnglobant = calculeRectEnglobant();
}

protected abstract Rectangle calculeRectEnglobant();
```

*- la méthode* **`getRectEnglobant`** *le renvoie*

```
public final Rectangle getRectEnglobant() {
   return rectEnglobant;
}
```

*- la méthode* **`placerA`** *doit le mettre à jour*

```
public final void placerA(int x, int y) {
    this.x = x;
    this.y = y;
    this.rectEnglobant.translate(x - this.x, y - this.y);
}
```




*- la méthode* **`calculeRectEnglobant`** *doit être concrétisée dans une des sous classes*

```
final Rectangle calculeRectEnglobant() {
   return new Rectangle(x - rayon,
                        y - rayon,
                        2*rayon, 2*rayon);
}
```

Cette solution ne marche pas ! Pourquoi ?

+ consultation plus efficace
- plus "complexe" à mettre en oeuvre

# Formes et rectangle englobant

```
public abstract class Forme implements IDessinable {
   protected int x;
   protected int y;
   protected Rectangle rectEnglobant;

   protected Forme(int x, int y, ...) {
     this.x = x;
     this.y = y;
     ...
     rectEnglobant = calculeRectEnglobant();
   }

   protected abstract Rectangle calculeRectEnglobant();

   public Rectangle getRectEnglobant() { return this.rectEnglobant); }
   ...
}
```

```
public abstract class FormeCirculaire extends Forme {
   protected int rayon;

   protected Forme(int x, int y, int r, ...) {
     super(x, y, ...);
     this.rayon = r;
   }

   final protected Rectangle calculeRectEnglobant() {
       return new Rectangle(x - rayon,  y - rayon,
                   2*rayon, 2*rayon);
   }
   ...
}
```

```
public class FormeCirculaireReguliere extends FormeCirculaire {

   protected FormeCirculaireReguliere (int x, int y, int r, ...) {
     super(x, y, r, ...);
     ...
   }
}
```

```
public class Etoile extends FormeCirculaireReguliere {

   protected Etoile(int x, int y, int r, ...) {
     super(x, y, r, ...);
     ...
   }
}
```

- échec à cause du chaînage des constructeurs

```
new Etoile(100,100, 50, …);
```

Quand ce code est exécuté, le rayon n'a pas encore été initialisé, le rectangle englobant est créé avec une largeur et hauteur nulles

# Formes et rectangle englobant

```
public abstract class Forme implements IDessinable {
   protected int x;
   protected int y;
   protected Rectangle rectEnglobant;

   protected Forme(int x, int y, ...) {
     this.x = x;
     this.y = y;
     ...
     rectEnglobant = calculeRectEnglobant();
   }

   protected abstract Rectangle calculeRectEnglobant();

   public Rectangle getRectEnglobant() {
       if (this.rectEnglobant == null) {
          this.rectEnglobant = calculeRectEnglobant();
       }
       return this.rectEnglobant);
    }
   ...
}
```

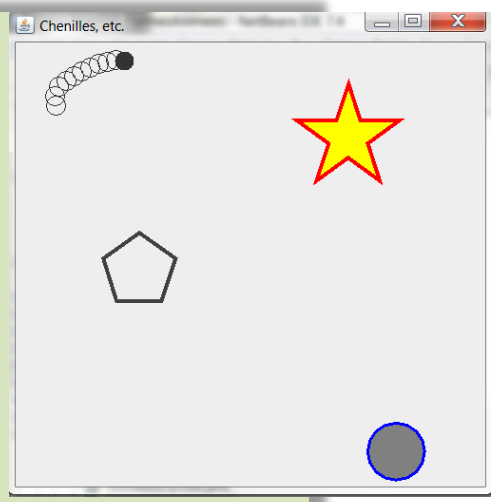
```
public abstract class FormeCirculaire extends Forme {
   protected int rayon;

   protected Forme(int x, int y, int r, ...) {
     super(x, y, ...);
     this.rayon = r;
   }

   final protected Rectangle calculeRectEnglobant() {
       return new Rectangle(x - rayon,  y - rayon,
                   2*rayon, 2*rayon);
   }
   ...
}
```

```
new Etoile(100,100, 50, …);
```

- ne pas passer par les constructeurs pour initialiser le rectangle englobant. Faire cette initialisation au premier appel de `getRectEglobant`

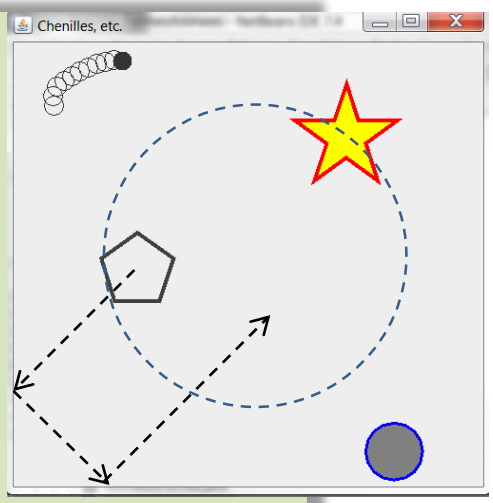
# Le programme principal

```
public class AnimationFormes {

    public static void main(String[] args) {

        // création de la fenêtre de l'application
        JFrame laFenetre = new JFrame("Chenilles, etc.");
        laFenetre.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        laFenetre.setSize(512, 512);
        // création de la zône de dessin dans la fenêtre
        Dessin d = new Dessin();
        laFenetre.getContentPane().add(d);
        // affiche la fenêtre
        laFenetre.setVisible(true);

        // les objets fixes de la zone de dessin
        d.ajouterObjet(new Cercle(400, 430, 30, 3.0f, Color.BLUE, Color.GRAY));
        d.ajouterObjet(new Etoile(350, 100, 50, 8.f, Color.RED, Color.YELLOW));
        d.ajouterObjet(new PolygoneRegulier(130, 240, 40, 5, 4.0f, Color.DARK_GRAY, null));
        // la chenille animée
        d.ajouterObjet(new Chenille(d,10,10));
        d.ajouterObjet(new VisageRond(d,10,10));

        while (true) {
            // la zone de dessin se réaffiche
            d.repaint();
            // un temps de pause pour avoir le temps de voir le nouveau dessin
            d.pause(50);
            // fait réaliser un déplacement élémentaire aux objets animables
            d.animer();
        }
    }
} // AnimationFormes
```



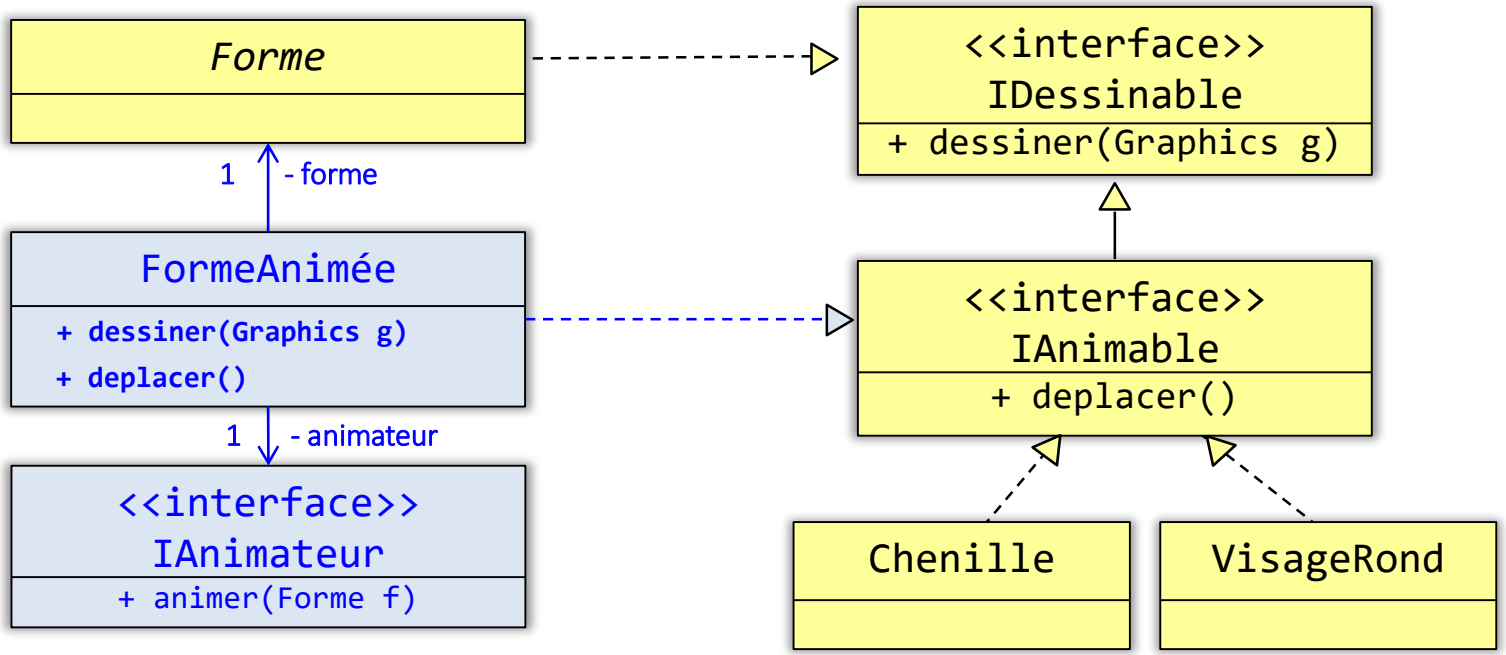
# Le programme principal

```
public class AnimationFormes {

    public static void main(String[] args) {

        // création de la fenêtre de l'application
        JFrame laFenetre = new JFrame("Chenilles, etc.");
        laFenetre.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        laFenetre.setSize(512, 512);
        // création de la zône de dessin dans la fenêtre
        Dessin d = new Dessin();
        laFenetre.getContentPane().add(d);
        // affiche la fenêtre
        laFenetre.setVisible(true);

        // les objets fixes de la zone de dessin
        d.ajouterObjet(new Cercle(400, 430, 30, 3.0f, Color.BLUE, Color.GRAY));
        d.ajouterObjet(new Etoile(350, 100, 50, 8.f, Color.RED, Color.YELLOW));
        d.ajouterObjet(new PolygoneRegulier(130, 240, 40, 5, 4.0f, Color.DARK_GRAY, null));
        // la chenille animée
        d.ajouterObjet(new Chenille(d,10,10));
        d.ajouterObjet(new VisageRond(d,10,10));

        while (true) {
            // la zone de dessin se réaffiche
            d.repaint();
            // un temps de pause pour
            d.pause(50);
            // fait réaliser un dépla
            d.animer();
        }
    }
} // AnimationFormes
```

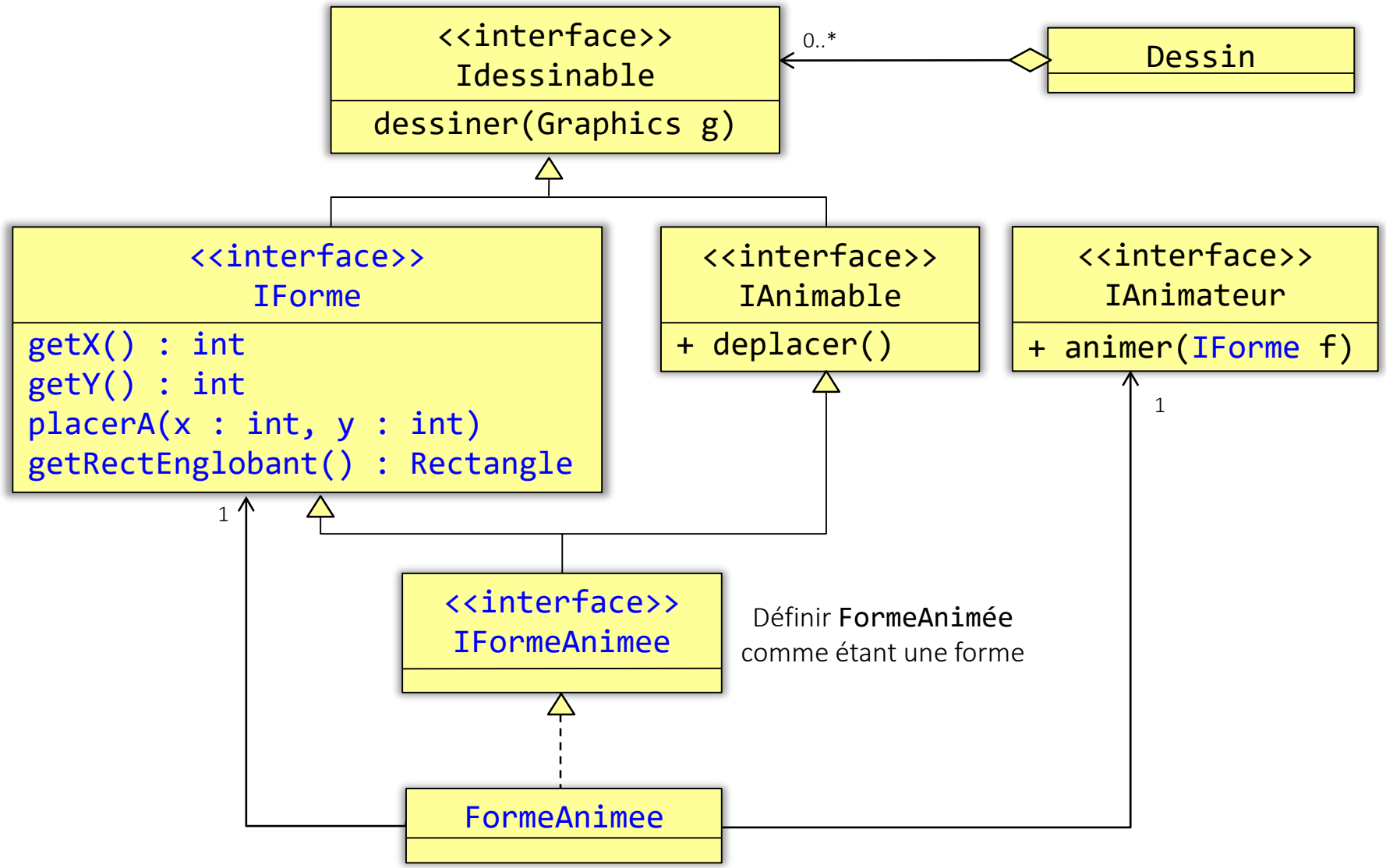



```
d.ajouterObjet(new FormeAnimee(
   new Etoile(350, 100, 50, 8.f, Color.RED, Color.YELLOW),
   new AnimateurCercle(250, 250, 180, 0, 5)
));
d.ajouterObjet(new FormeAnimee(
   new PolygoneRegulier(130, 240, 40, 5, 4.0f, Color.DARK_GRAY, null),
   new AnimateurRebond(d,5, 5)
));
d.ajouterObjet(new Chenille(d, 10, 10));
```

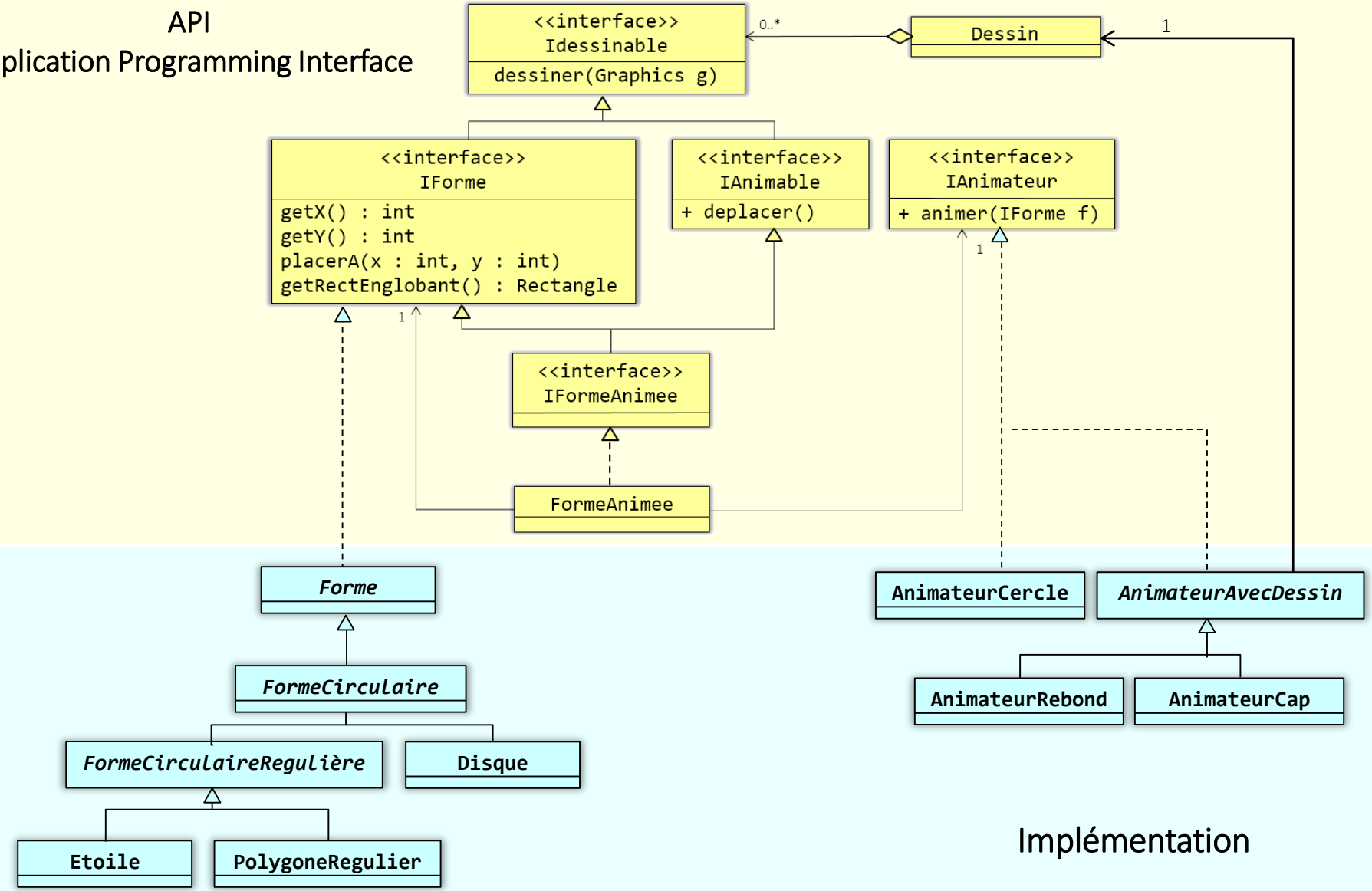
classe Abstraite




- Les formes animées ne peuvent être vues comme des formes…


<<interface>>
Idessinable
dessiner(Graphics g)
0..*
Dessin
<<interface>>
IForme
getX() : int
getY() : int
placerA(x : int, y : int)
getRectEnglobant() : Rectangle
<<interface>>
IAnimable
+ deplacer()
<<interface>>
IAnimateur
+ animer(IForme f)
1
1
<<interface>>
IFormeAnimee
Définir FormeAnimée
comme étant une forme
FormeAnimee

API
Application Programming Interface




Implémentation