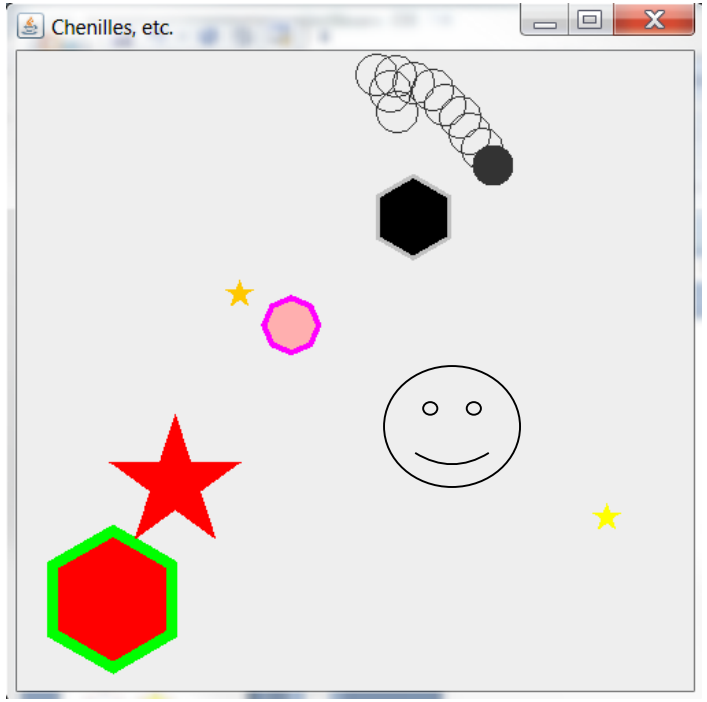
TP : Formes Animées

Philippe Genoud
dernière mise à jour :21/01/2025 21:50

Le problème

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



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

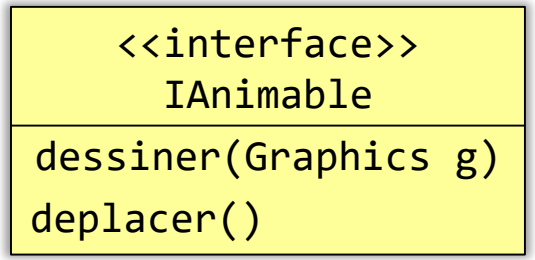
Il faut remplacer le type **Chenille** par un type le plus général possible.

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

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

```
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 CopyOnWriteOArrayList<>();

   ...
  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);
        }
    }

    public void deplacerChenilles() {
        for (Chenille c : lesChenilles) {
            c.deplacer();
        }
   }
}
```

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

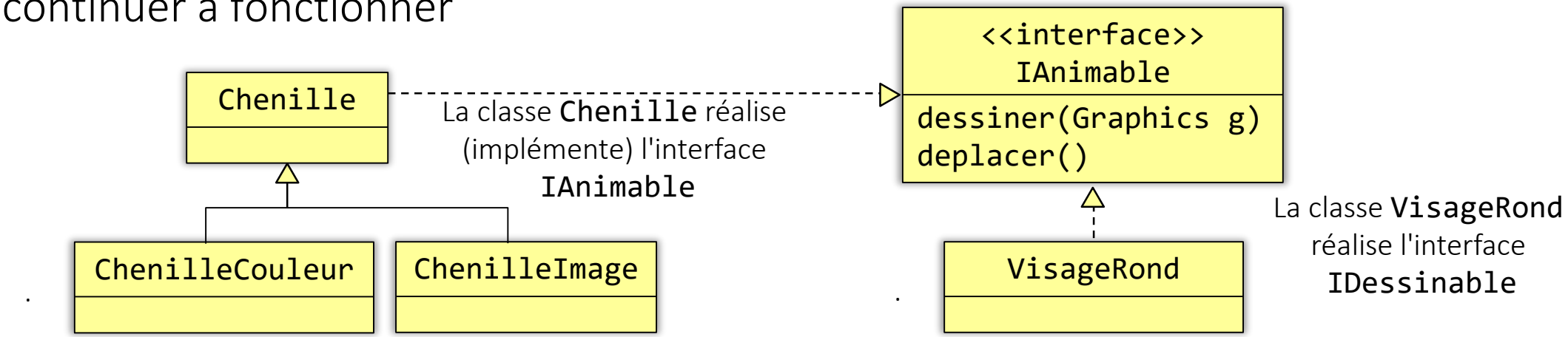


IDessinable.java

```
import java.awt.Graphics;

public interface IAnimable {

    void dessiner(Graphics g);
    void deplacer();

}
```

```
import javax.swing.JPanel;
...
public class Dessin extends JPanel{

    /**
     * stocke la liste des objets de cette zone de dessin.
     */
    private final List<IAnimable> lesObjets = new CopyOnWriteOArrayList<>(); ;

   ...
  public void ajouterObjet(IAnimable ch) {
        if (! lesObjets.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 (IAnimable o : lesObjets) {
            o.dessiner(g);
        }
    }

    public void  deplacerObjets {
        for (IAnimable o : lesObjets) {
            o.deplacer();
        }
   }
}
```

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

```
import java.awt.Graphics;

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

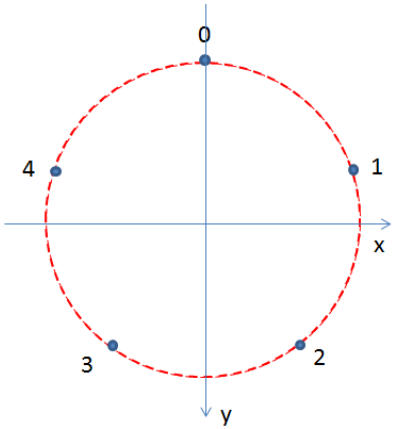
```
import java.awt.Graphics;

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

```
import java.awt.Graphics;
...
public class ChenilleImage extends Chenille {
    ...
}
```

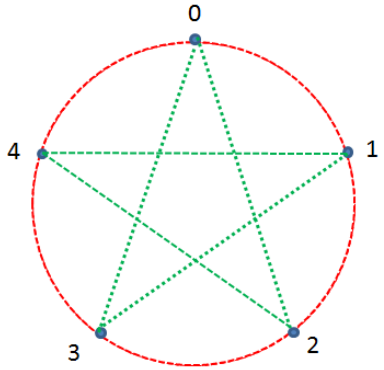
Le programme principal

```
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
        d.ajouterObjet(new ChenilleImage(d, 10, imgVador));
        d.ajouterObjet(new ChenilleImage(d, 10, imgLeila));
        for (int i = 0; i < 8; i++) {
            d.ajouterObjet(new ChenilleCouleur(new Color((float) Math.random(), (float) Math.random(),
                                        (float) Math.random()), d, 10, 10));
        }
        d.ajouterObjet(new VisageRond(d, 300, 500, 40, 60));
        while (true) {
            d.repaint(); // la zone de dessin se réaffiche
            d.pause(50); // un temps de pause pour avoir le temps de voir le nouveau dessin
            d.deplacerObjets(); // fait réaliser aux objets un déplacement élémentaire
        }
    }
}
```

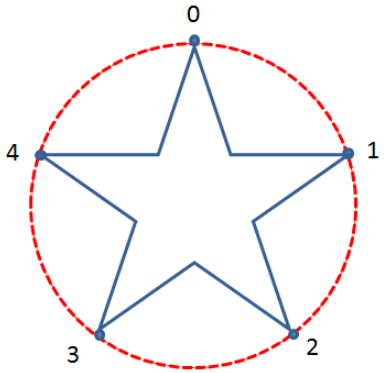
- étoiles à 5 branches



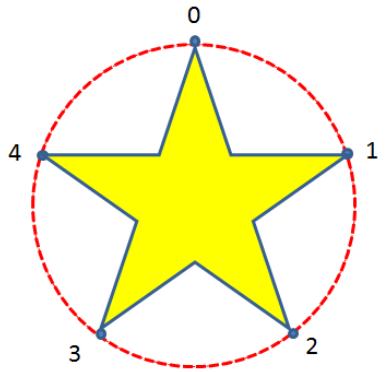
1) calcul des sommets du polygone régulier



2) construction du chemin reliant les points



3) Dessin du contour



4) Remplissage de la forme

```
int nbSommets = 5;

// calcul des sommets de l'étoile
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);
```

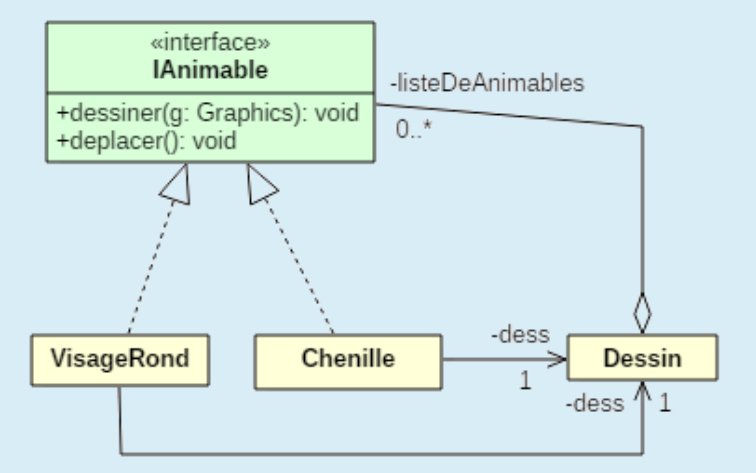
attributs :

- `r` rayon de l'étoile
- `x, y` centre de l'étoile
- couleur du trait
- épaisseur du trait
- couleur de remplissage
- l'objet `Path2D` qui défini le tracé du contour de l'étoile

code à exécuter à la construction de l'objet

code de la méthode dessiner

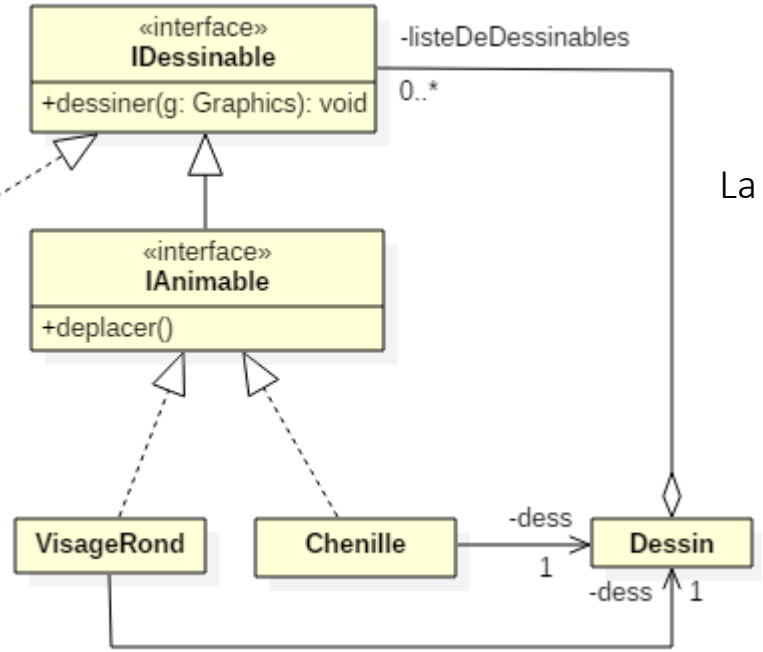
un nouveau type d'objet les étoiles fixes



Etoile
-couleurTrait: Color {readOnly} -couleurRemplissage: Color {readOnly} -epaisseurTrait: float {readOnly} -x: int {readOnly} -y: int {readOnly} -r: int {readOnly} -contour: Path2D
«constructor»+Etoile(x: int, y: int, r: int, float epTrait, cTrait: Color, cRemp: Color) +dessiner(g: Graphics): void

Les étoiles étant fixes, quid de `deplacer()` ?

Distinguer les objets uniquement dessinables des objet animables



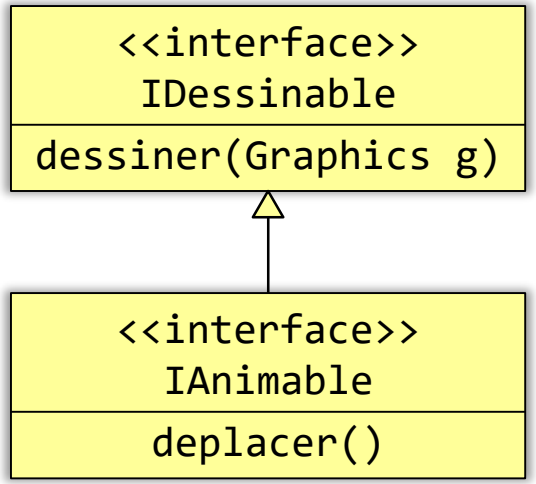
Etoile
-couleurTrait: Color {readOnly} -couleurRemplissage: Color {readOnly} -epaisseurTrait: float {readOnly} -x: int {readOnly} -y: int {readOnly} -r: int {readOnly} -contour: Path2D
«constructor»+Etoile(x: int, y: int, r: int, float epTrait, cTrait: Color, cRemp: Color) +dessiner(g: Graphics): void

La liste des objets gérés par le dessin est une liste d'objets `IDessinables`

- Définir **IAnimable** comme sous-type d'une nouvelle interface **IDessinable**

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

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

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

   public void deplacerObjets() {
        // 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

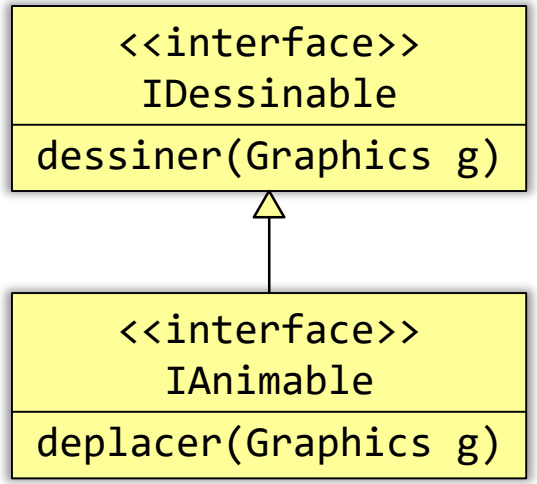
```
import java.awt.Graphics;

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

IAnimable.java

```
import java.awt.Graphics;

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

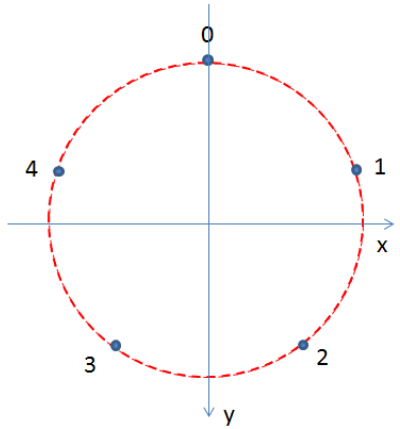
- Définir **IAnimable** comme sous-type d'une nouvelle interface **IDessinable**

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

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

    ...
   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();
               }
        }
   }
}
```

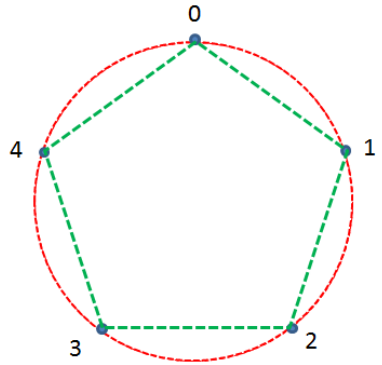


il faut vérifier que l'objet est bien animable et dans ce cas faire un *downcasting* pour pouvoir appeler la méthode **déplacer**.

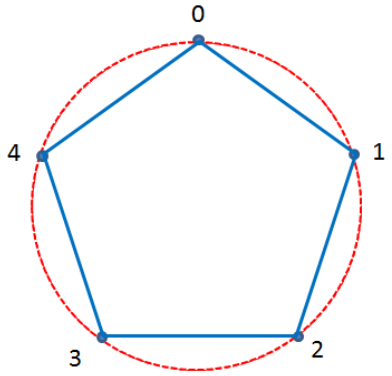
- polygones régulier (ex. pentagone)



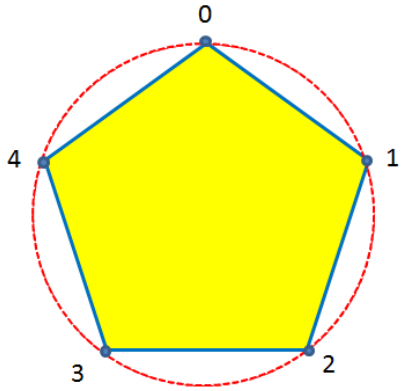
1) calcul des sommets du polygone régulier



2) construction du chemin reliant les points

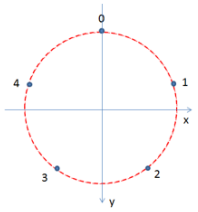


3) Dessin du contour

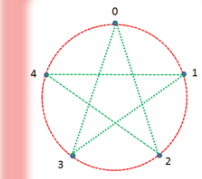


4) Remplissage de la forme

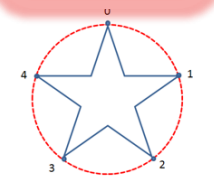
```
int nbSommets = 5;

// 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(path);

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



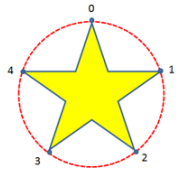
1) calcul des sommets du polygone régulier



2) construction du chemin reliant les points



3) Dessin du contour



4) Remplissage de la forme

Start Page | codeEtoile <-> codePolyRegulier [Diff]

Swap Export... Options...

Graphical | Textual

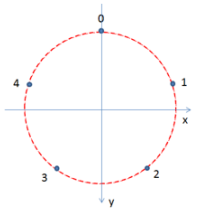
P:\ENSEIGNEMENT\ServeursWEB\tds\corrections\TPChenillesEtFormes\SupportCoursCorrection\codeEtoile.txt

```
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);
```

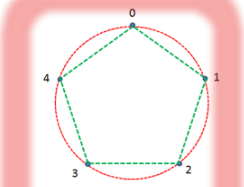
1/1

P:\ENSEIGNEMENT\ServeursWEB\tds\corrections\TPChenillesEtFormes\SupportCoursCorrection\codePolyRegulier.txt

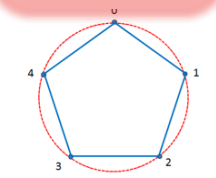
```
int nbSommets = 5;

// 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(path);
```



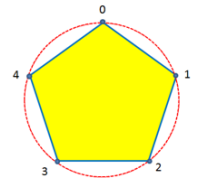
1) calcul des sommets du polygone régulier



2) construction du chemin reliant les points



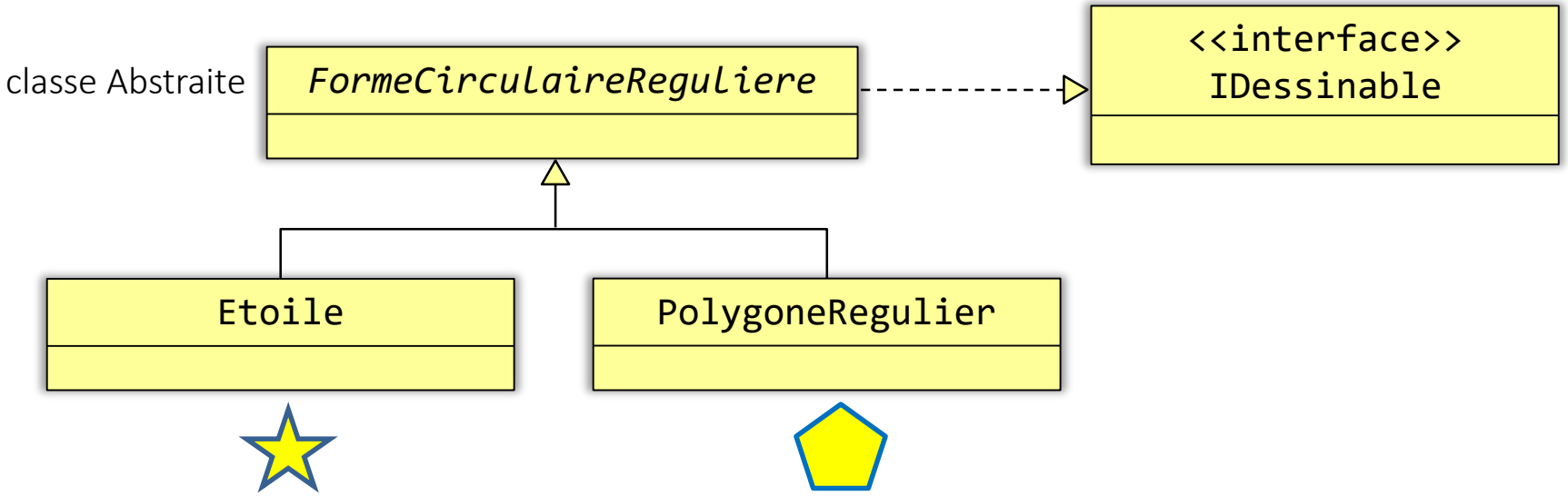
3) Dessin du contour



4) Remplissage de la forme

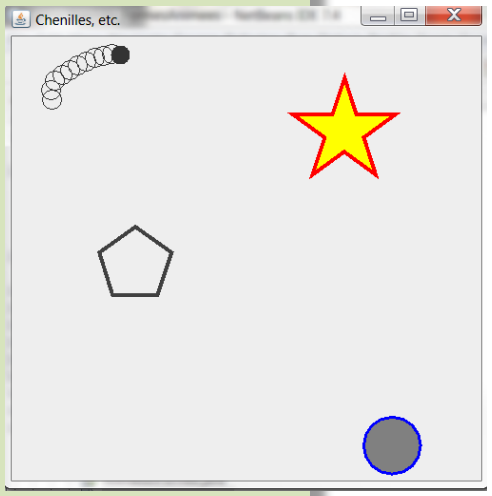
- polygones réguliers et étoiles à 5 branches partagent de nombreuses caractéristiques :
  - *ces sont des formes inscrites dans un cercle, dont le contour a les sommets répartis régulièrement sur le cercle circonscrit.*
- Comment partager ces caractéristiques au niveau du code ?

- En généralisant les concepts Etoile et Polygone Régulier

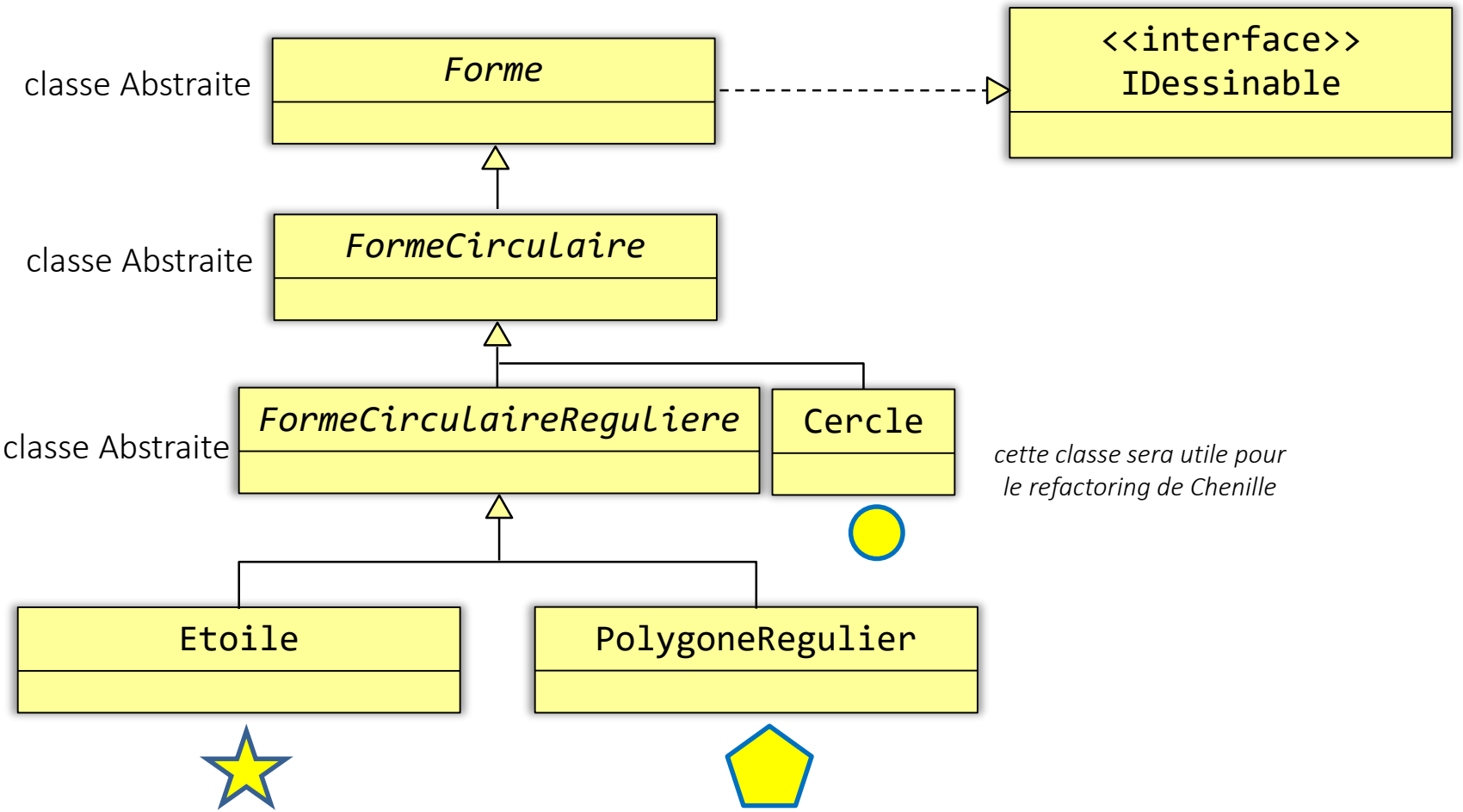


Le programme principal

```
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

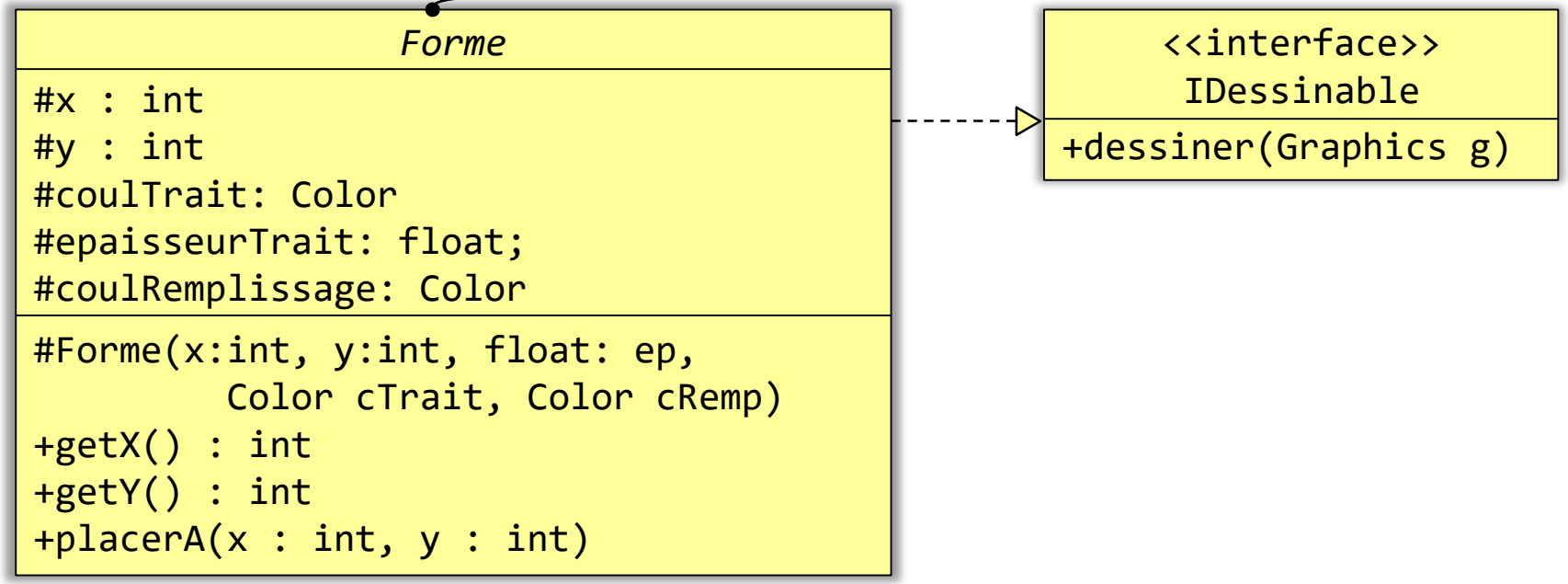


.

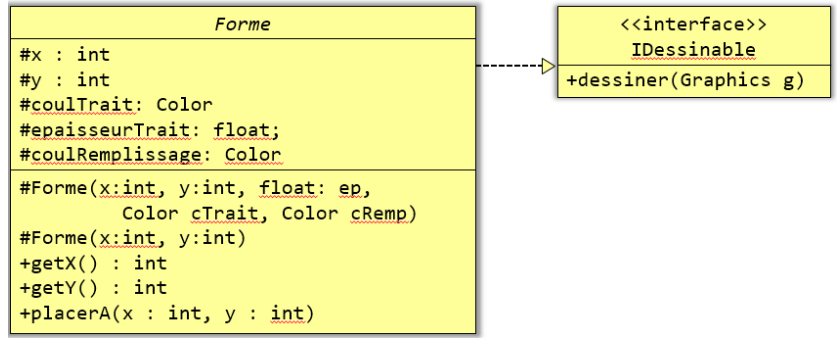
Forme : un objet dessinable défini par un point de référence (qui permet de positionner la forme dans le plan)

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)`

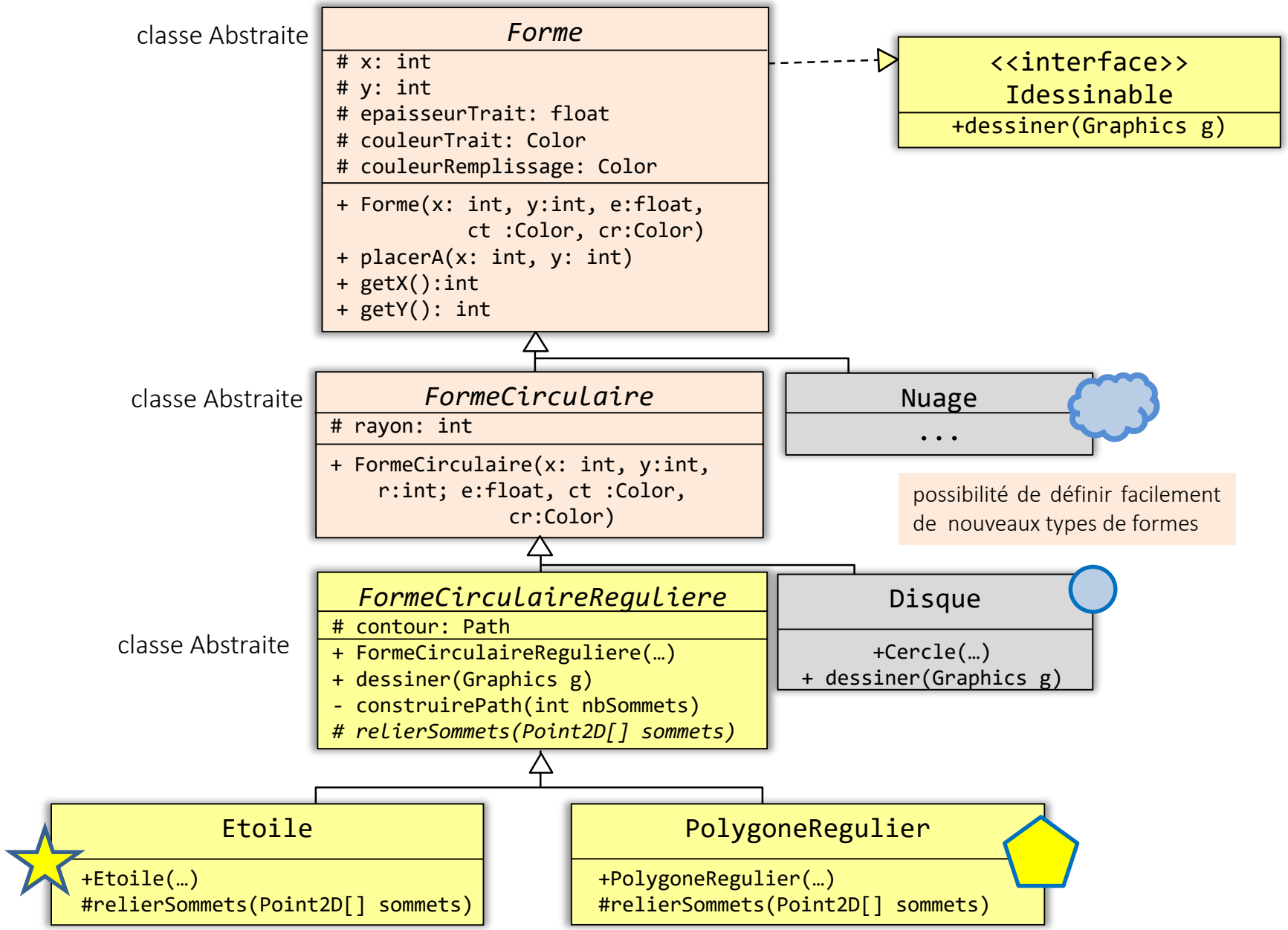


Forme.java

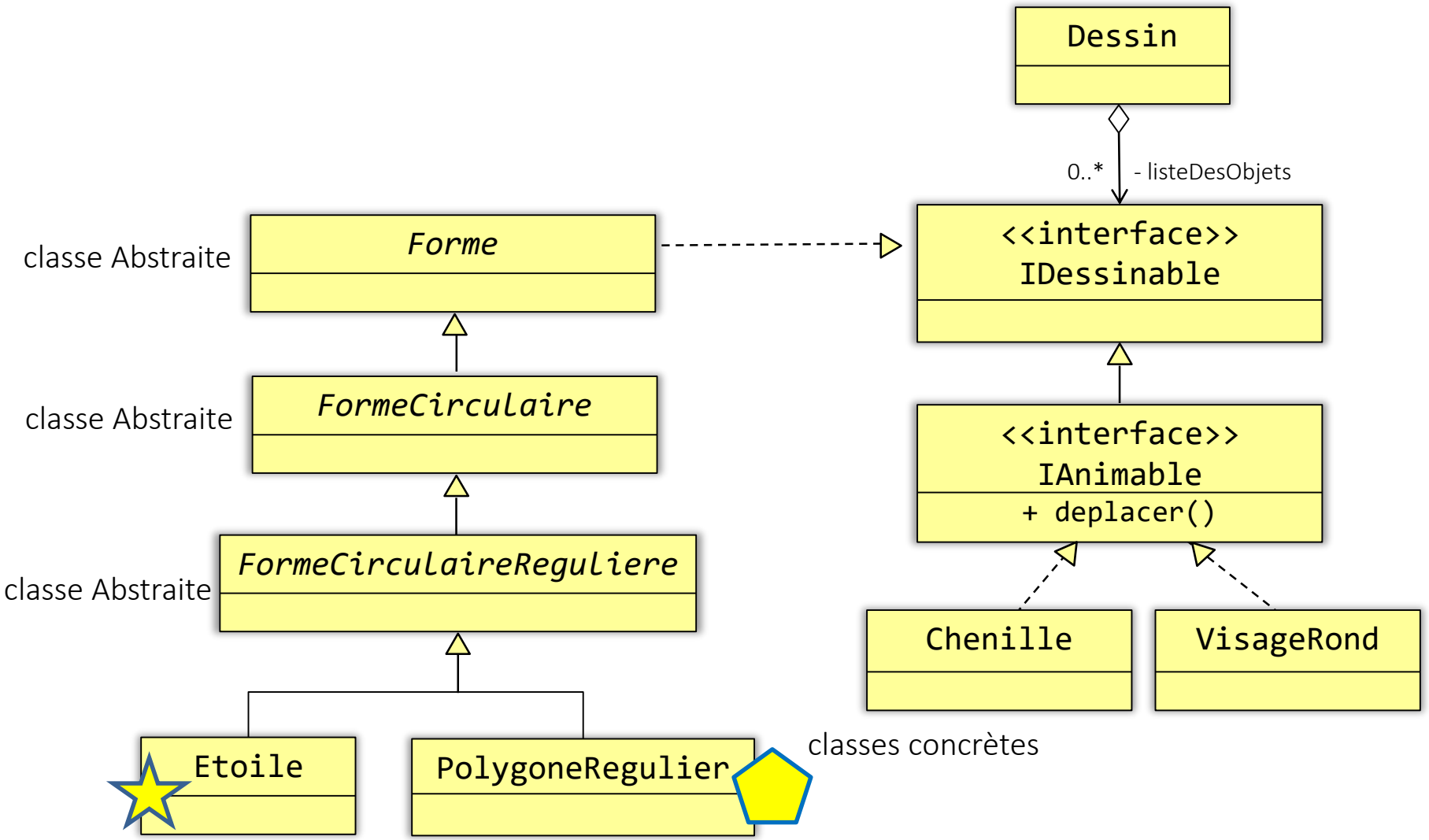
```
import java.awt.Color;
import java.awt.Rectangle

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);
    }
```

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

    public int getX() {
         return x;
     }

     public int getY() {
         return y;
     }

}  // Forme
```

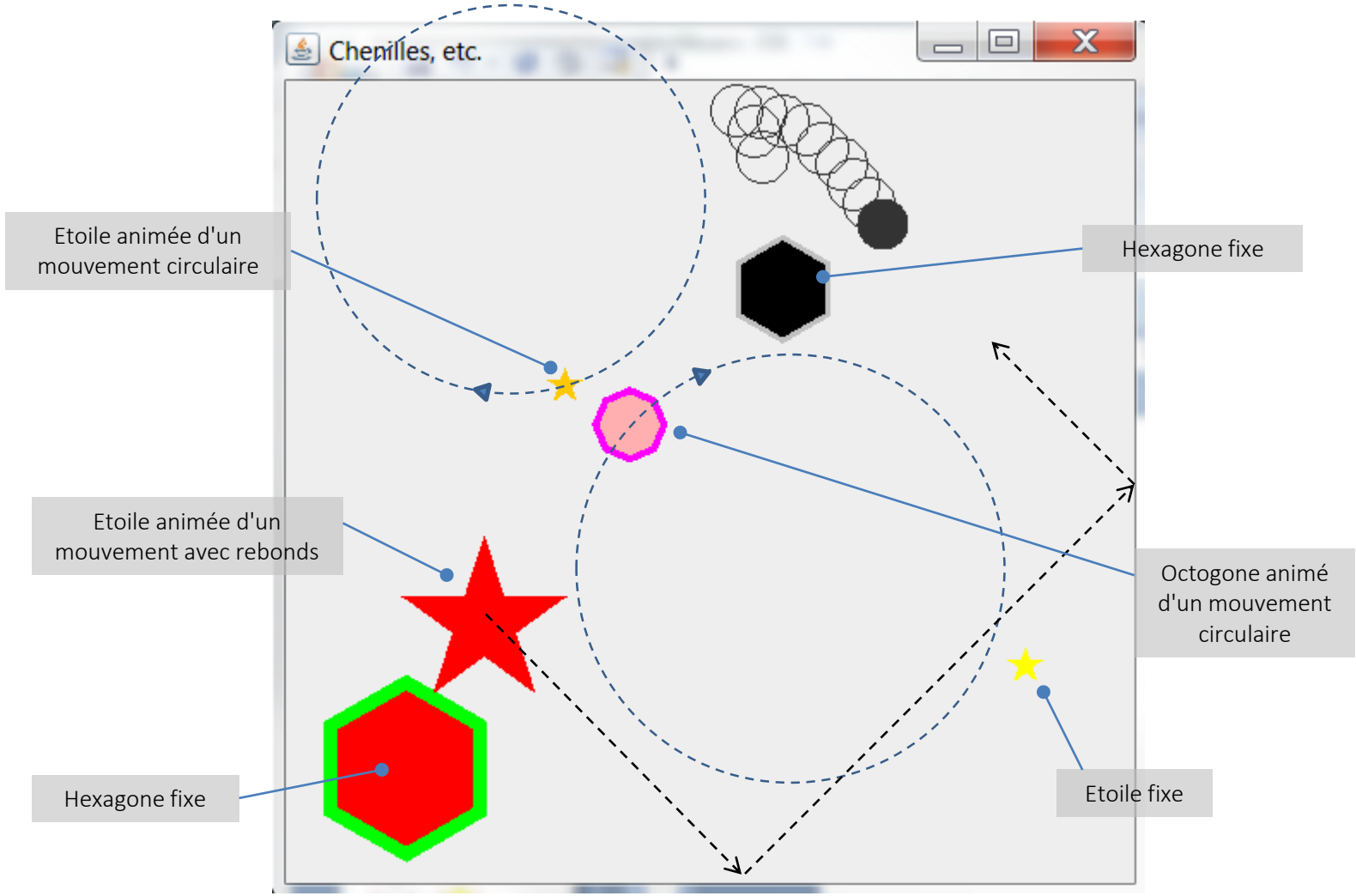
Généralisation de la notion de forme



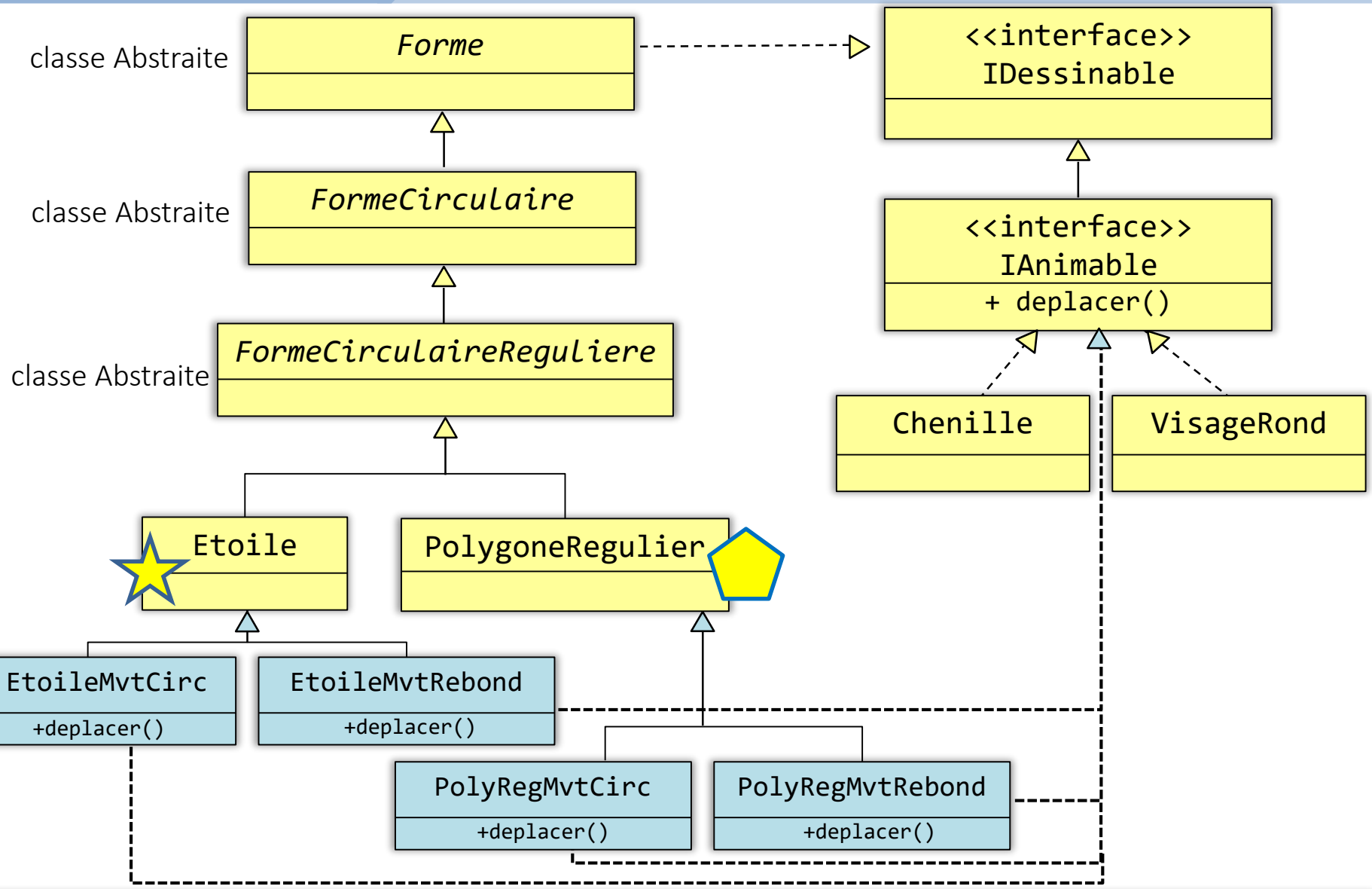
Animer certaines formes



Animer certaines formes

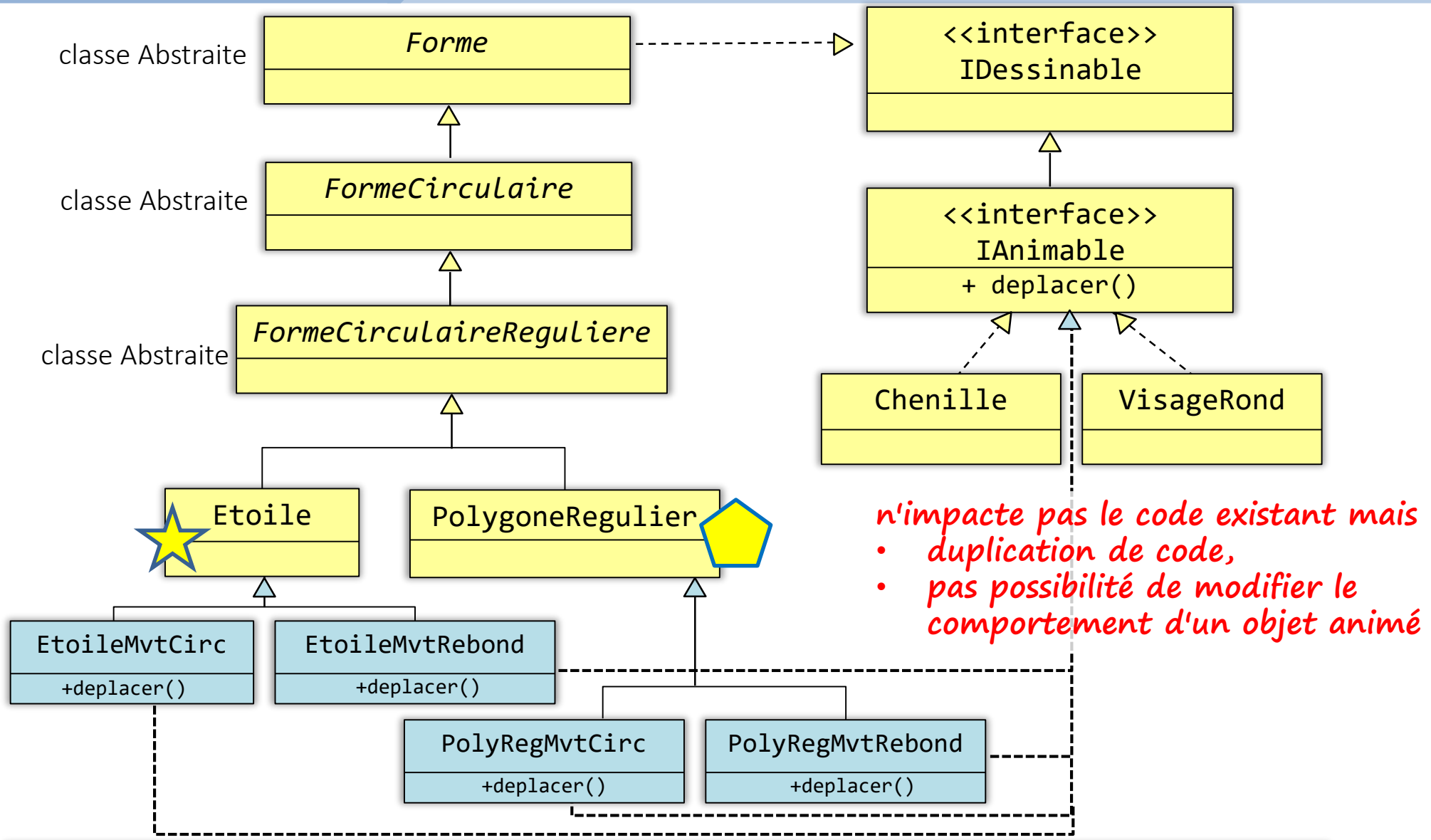


Animer certaines formes



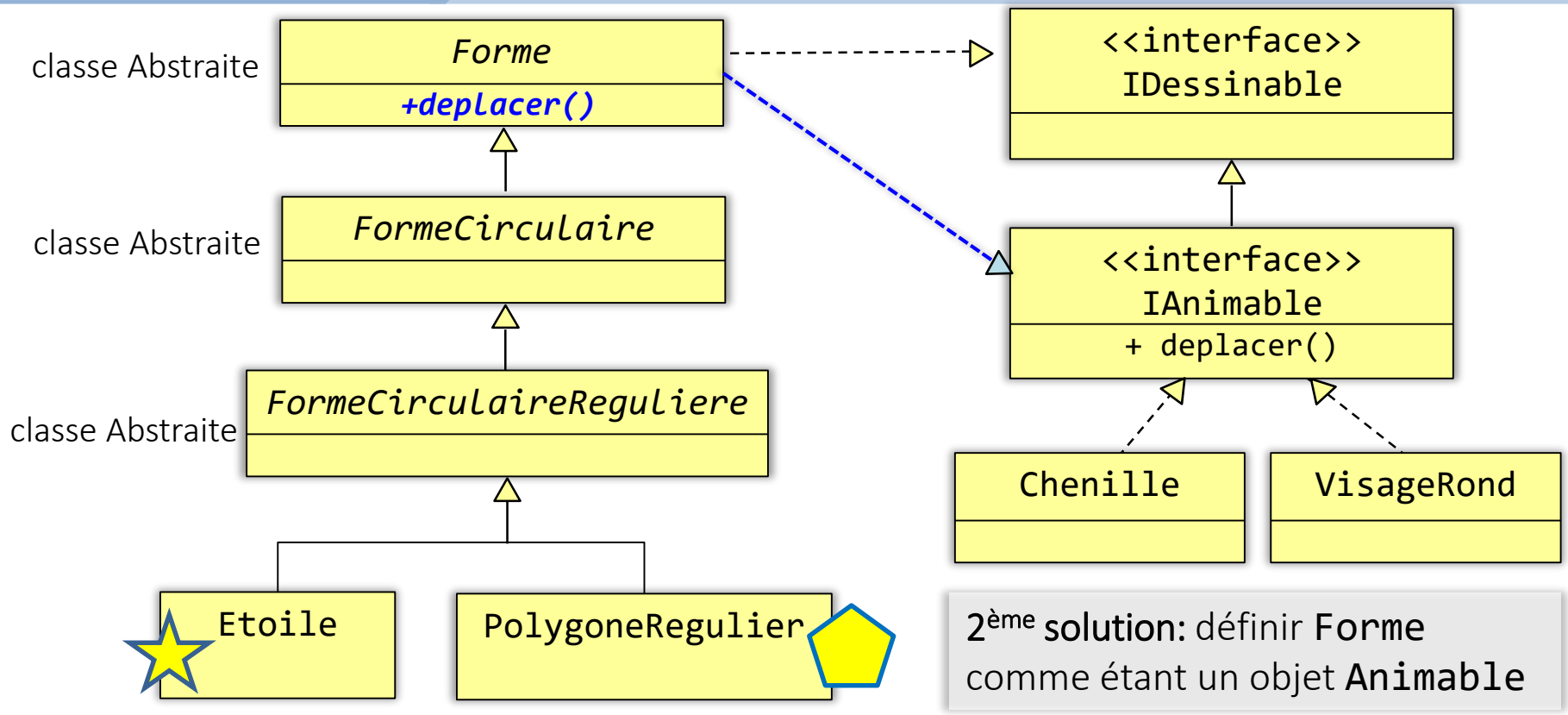
1ère solution: sous classer les différents types de formes en les dotant d'un comportement d'animation

Animer certaines formes



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

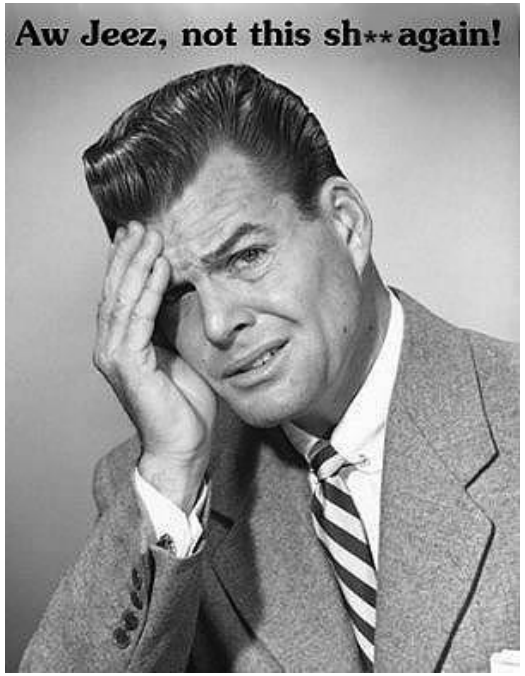
Animer certaines formes



- 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

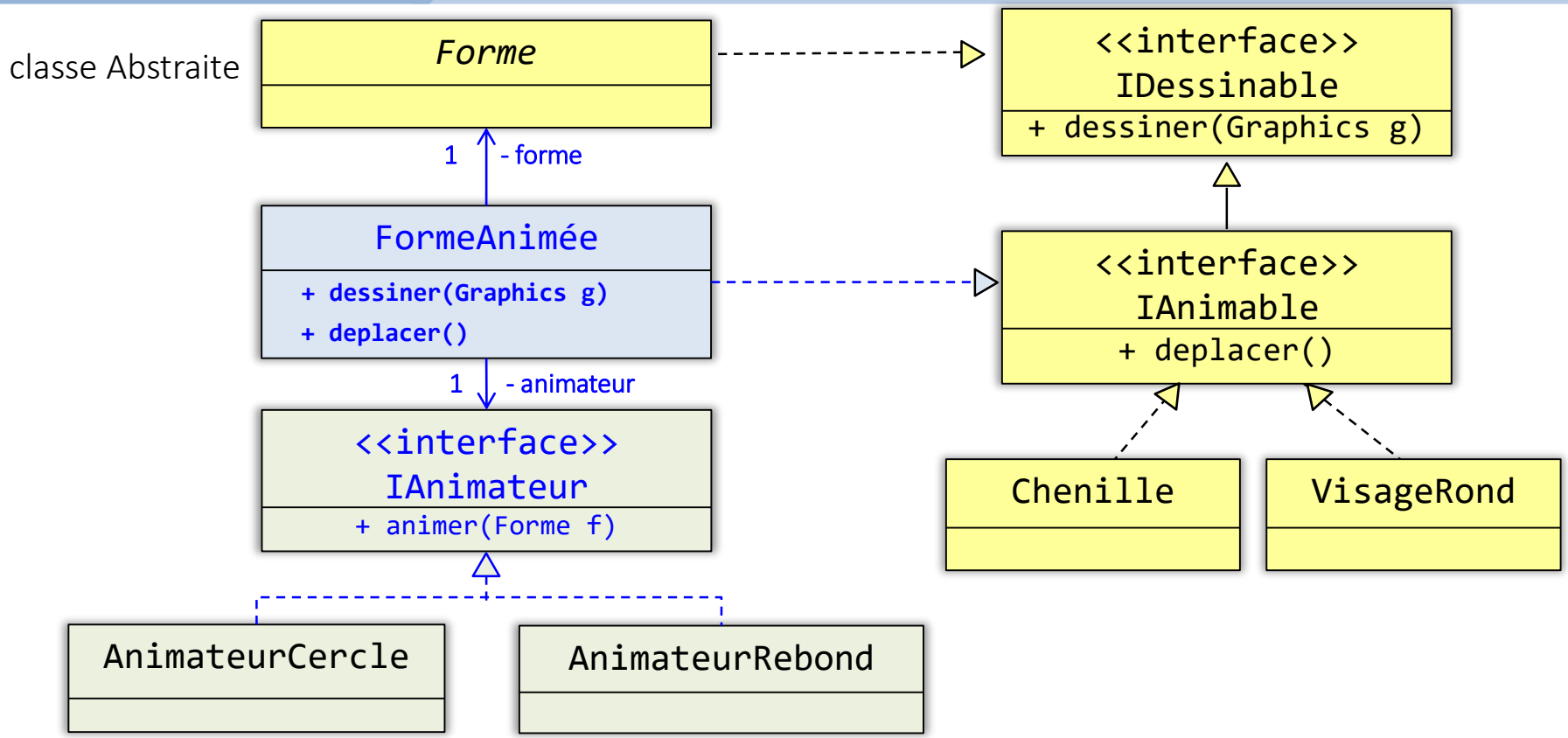
Solution lourde, peu flexible et extensible, impact possible sur le code existant

- 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**

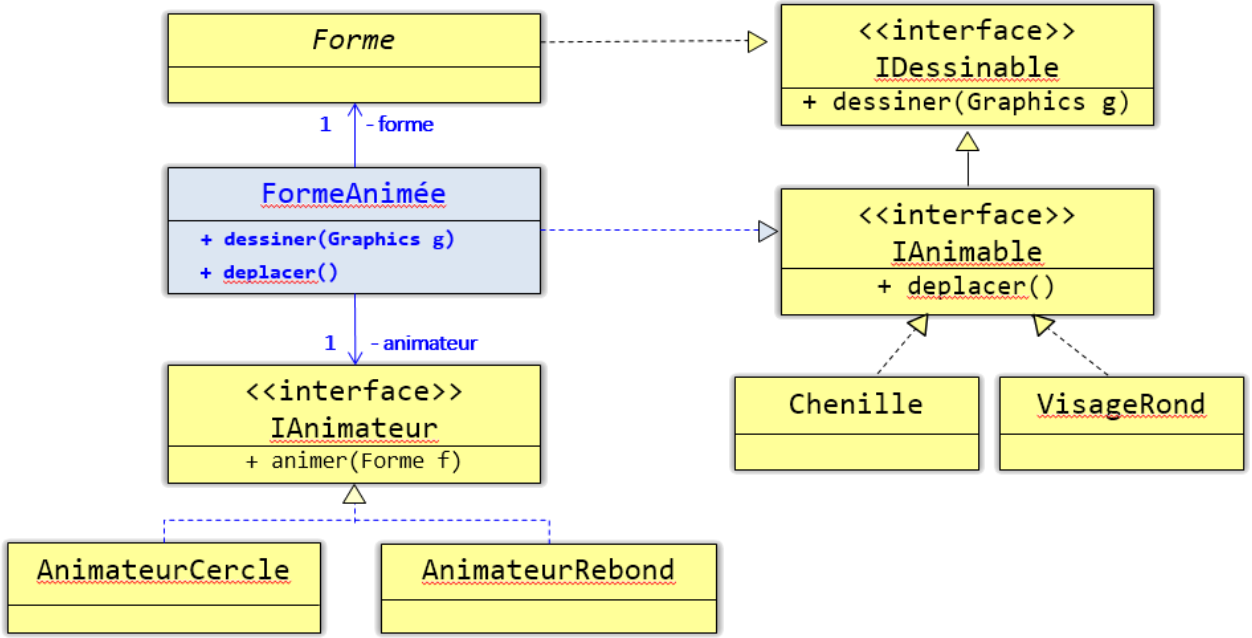
la classe FormeAnimee

FormeAnimee.java

```
public class FormeAnimee implements IAnimable {

    protected Forme forme;
    protected IAnimateur animateur;

    public FormeAnimee(Forme forme, IAnimateur animateur) {
        this.forme = forme;
        this.animateur = animateur;
    }

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

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

}
```

délégation de l'animation à l'animateur

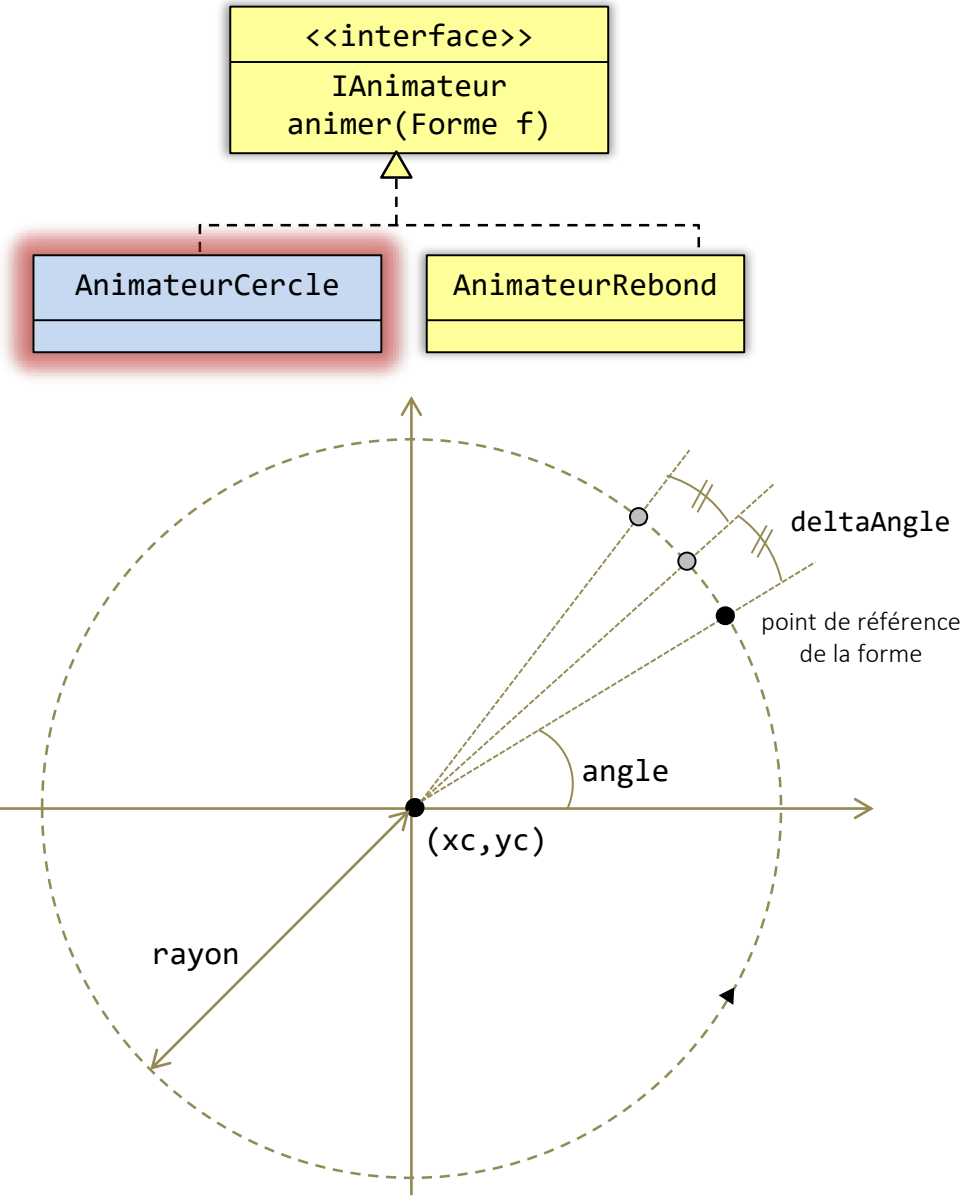
délégation du dessin à la forme



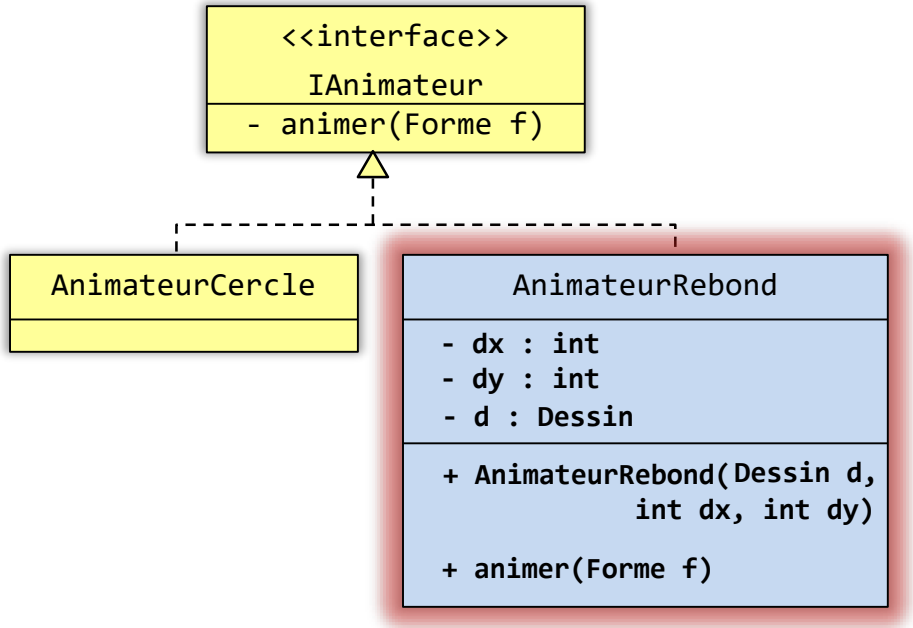
La classe AnimateurCercle

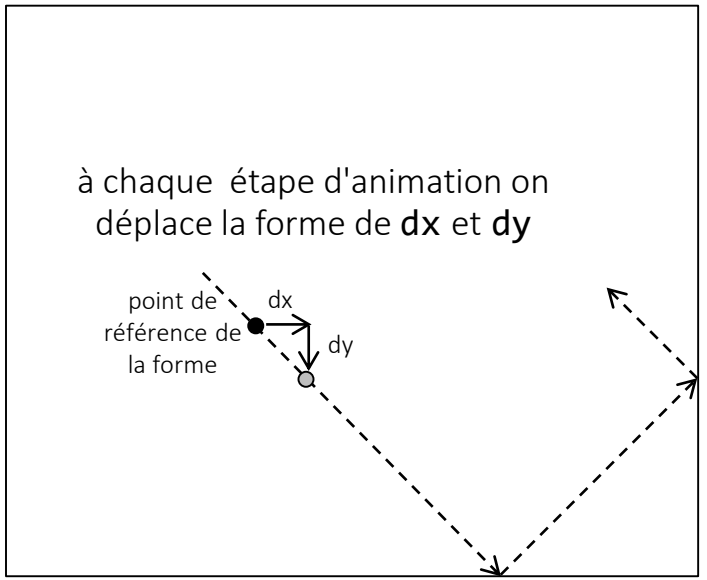
AnimateurCercle.java

```
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

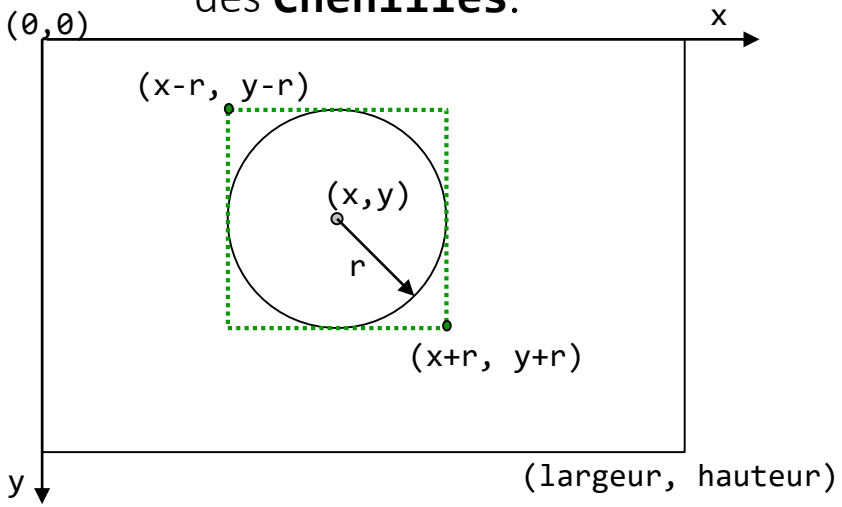
```
public void animer(Forme f) {

    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 ?

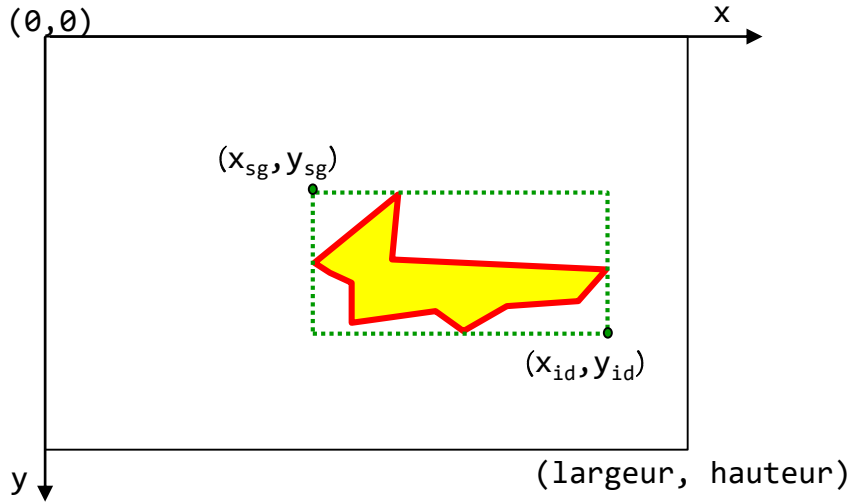
regardons ce qui a été fait pour les **VisageRond** ou les **Tête** des **Chenilles**.



sort à gauche `x - r < 0` || sort à droite `x + r > dessin.getLargeur()`
||
`y - r < 0` || `y + r > dessin.getHauteur()`
sort en haut sort en bas

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 ?

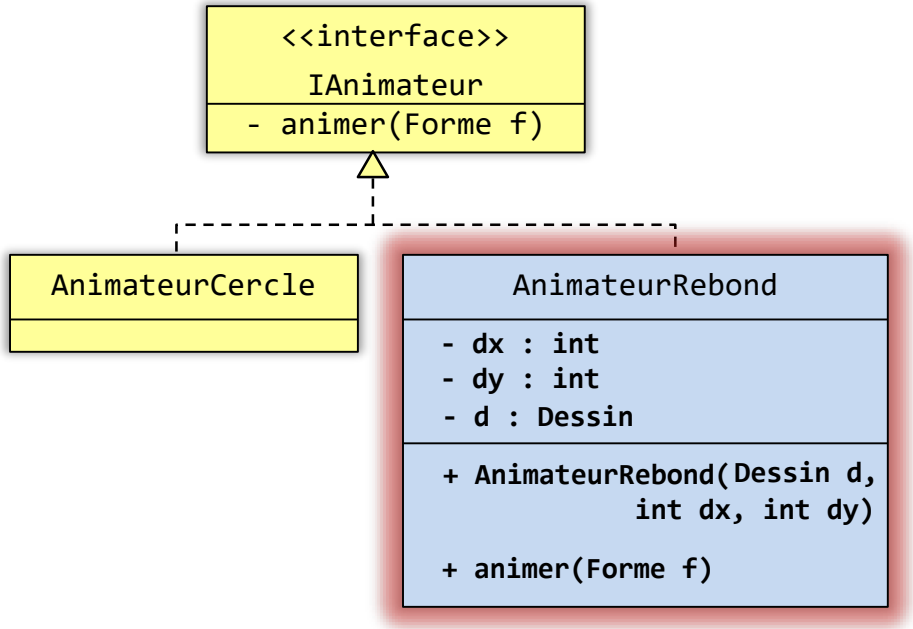
généralisons cela pour supporter n'importe quel type de **Forme**.

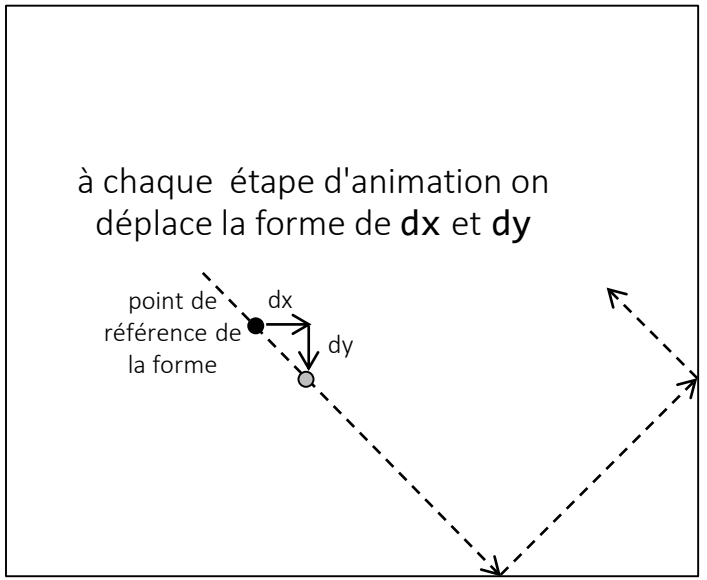


sort à gauche $x_{sg} < 0$ || sort à droite $x_{id} > dessin.getLargeur()$
||
$y_{sg} < 0$ || $y_{id} > dessin.getHauteur()$
sort en haut sort en bas

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**

```
public void animer(Forme f) {
    Rectangle rect = f.getRectEnglobant();
    if ( sortAGauche(rect)  ||  sortADroite(rect)  ) {
       dx = -dx;
    }
    if ( sortEnHaut(rect)  ||  sortEnBas(rect)  ) {
       dy = -dy;
    }
    int newx = f.getX() + dx;
    int newy = f.getY() + dy;

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

il suffit que les formes puissent donner leur rectangle englobant

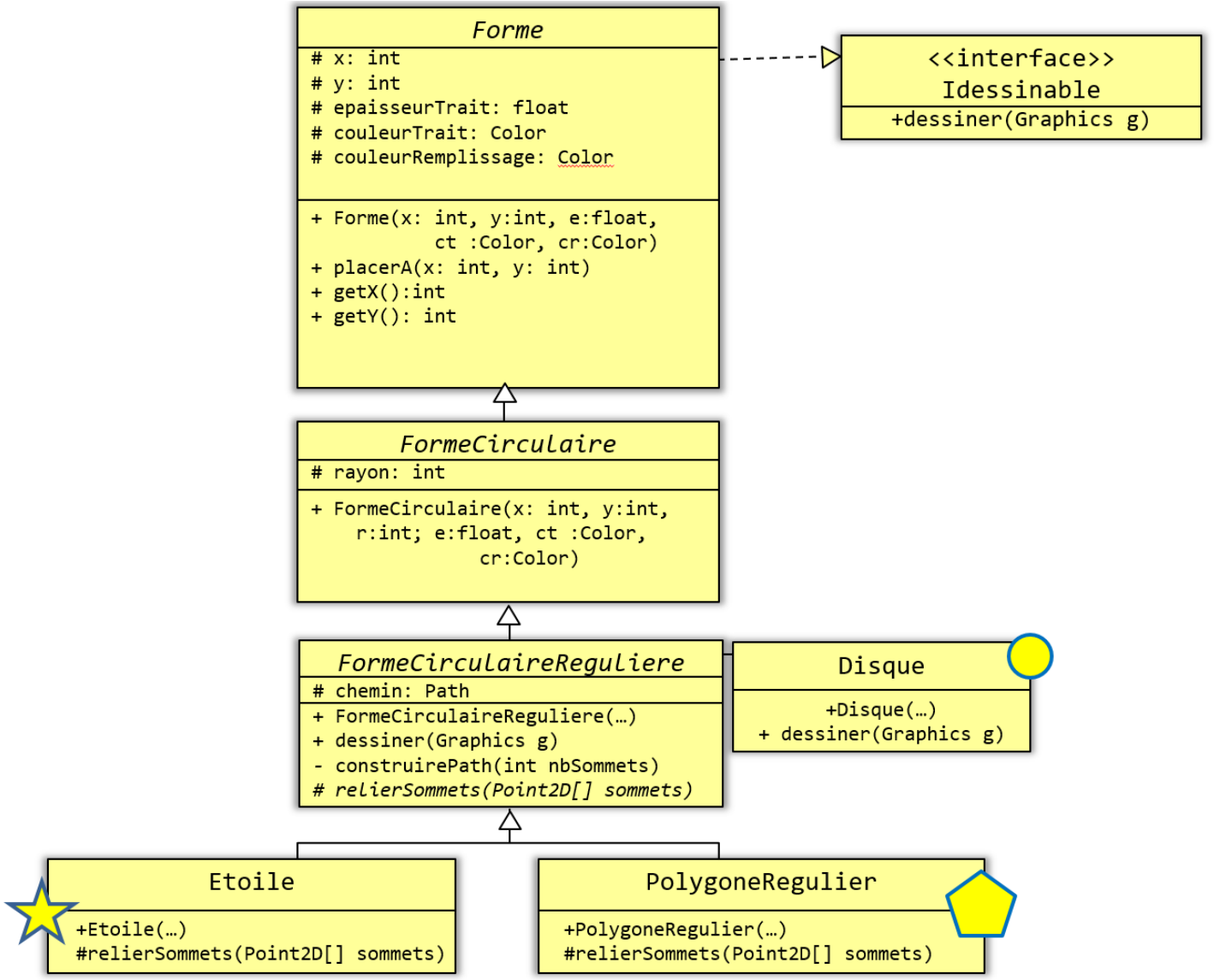
```
private  boolean sortAGauche(Rectangle r) {=
   return r.getX() < 0;
}
```

```
private  boolean sortADroite(Rectangle r) {
   return r.getX() + r.getWidth() > d.getLargeur();
}
```

...

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 ?

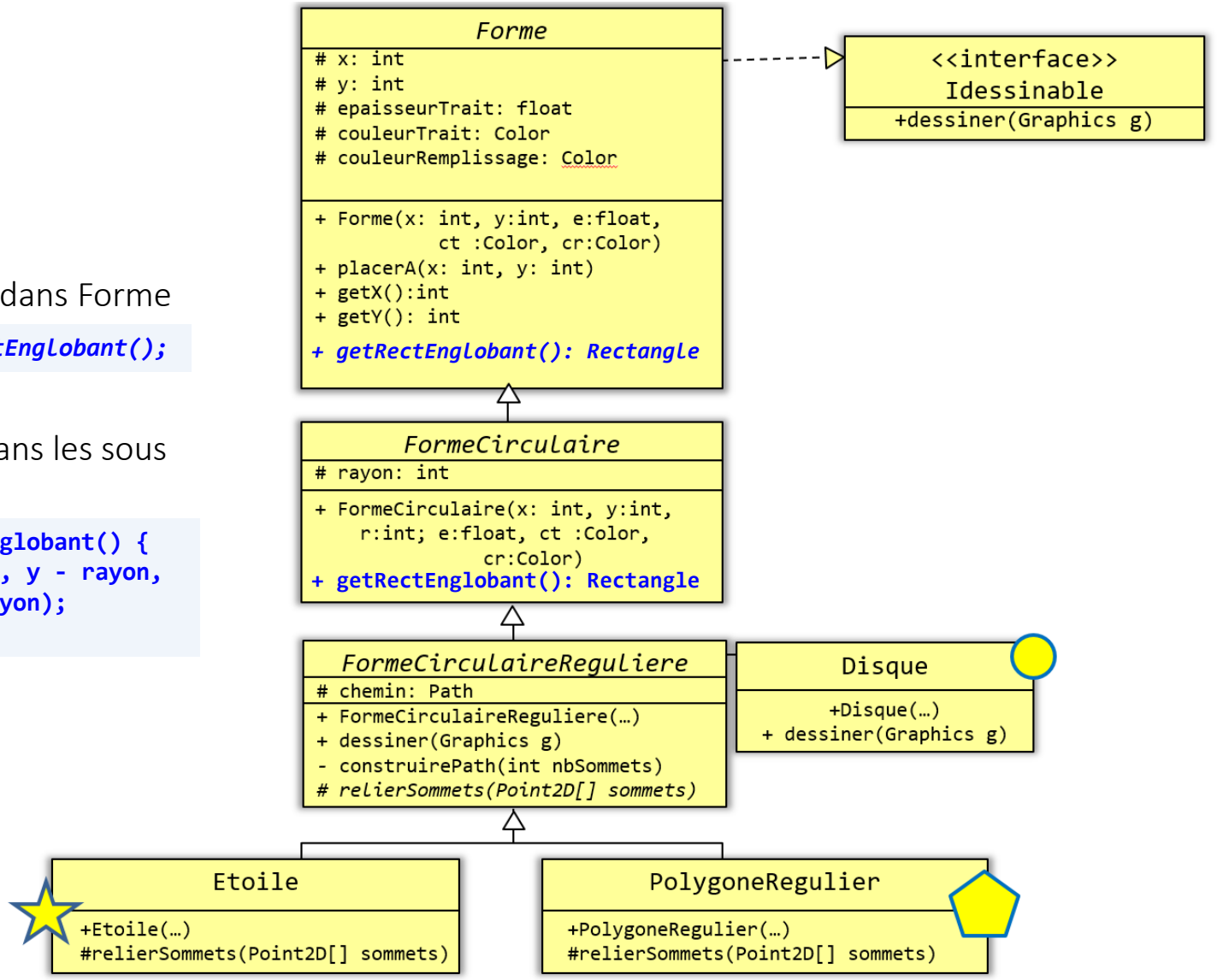
une solution simple

définir une méthode abstraite dans Forme

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

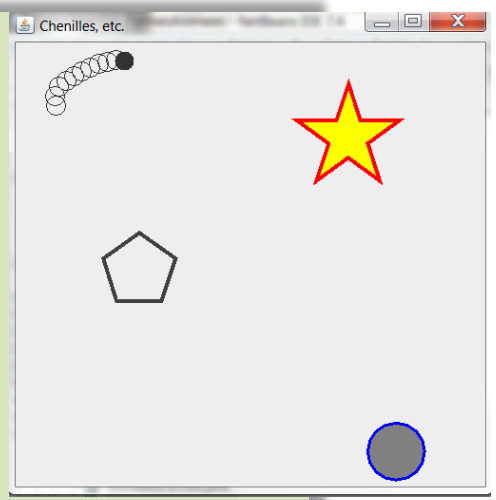
implémenter cette méthode dans les sous classes

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



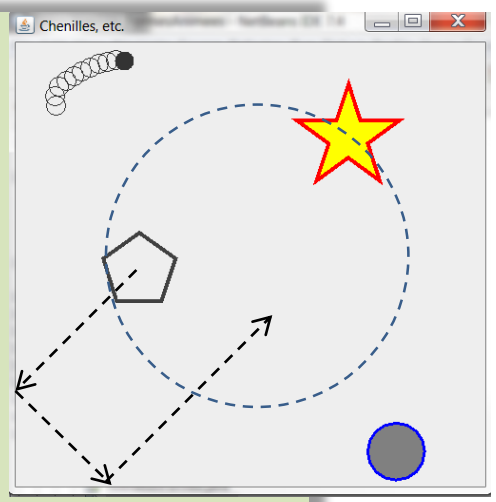
Le programme principal

```
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.BLACK, Color.GRAY));
        // 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
```



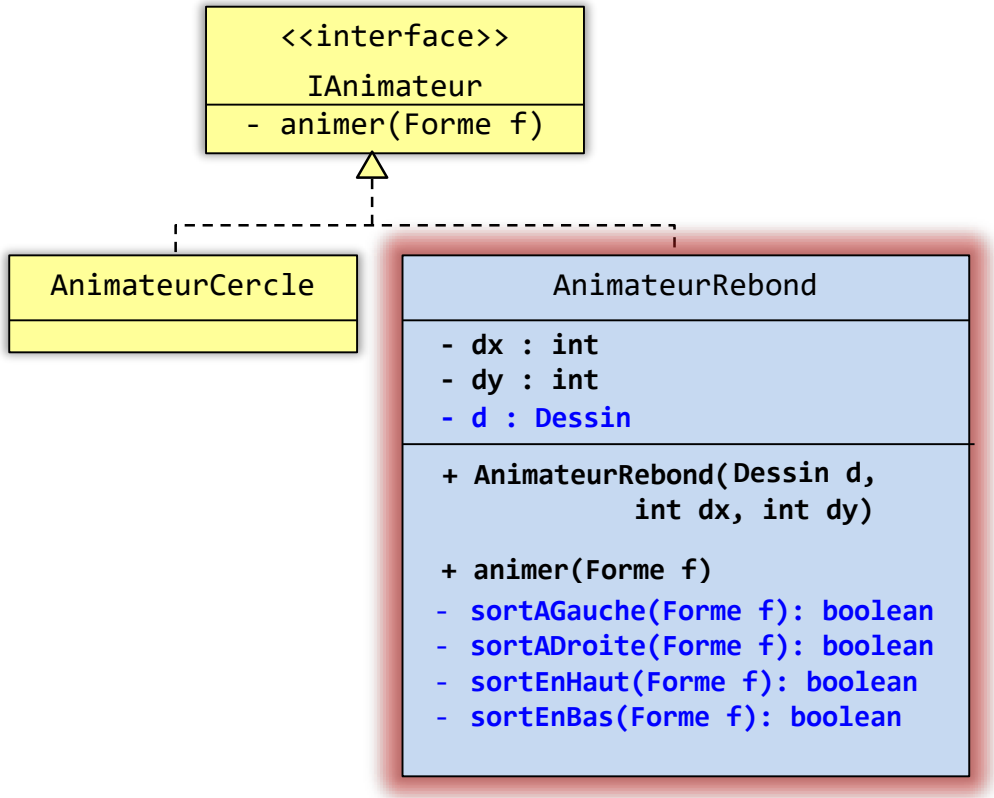
Le programme principal

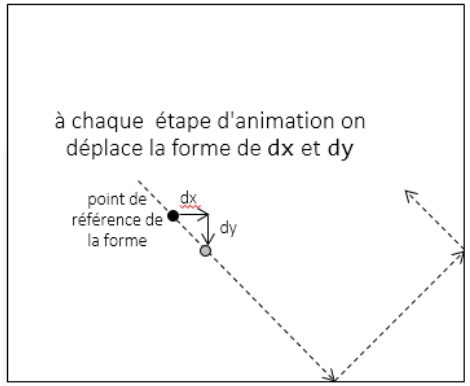
```
public class AnimationFormes {

    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.BLACK, Color.GRAY));
        // 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 temp
            d.pause(50);
            // fait réaliser un déplacement élément
            d.animer();
        }
    }
} // AnimationFormes
```



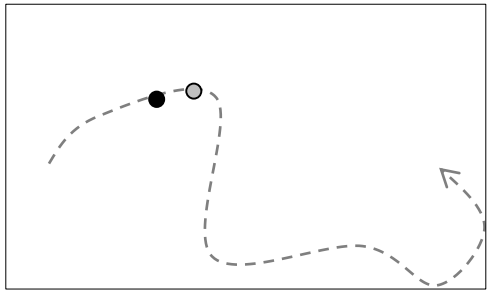

```
d.ajouterObjet(new FormeAnimee(
   new Etoile(350, 100, 50, 8.f, Color.RED, Color.YELLOW),
   new AnimateurCercle(250, 250, 180, 0, 5)
));
d.ajouterObjet(new FormeAnimee(
   new PolygoneRegulier(5,130, 240, 40, 5, 4.0f, Color.BLACK, Color.GRAY),
   new AnimateurRebond(d,5, 5)
));
```

Extension à d'autres types d'animateurs

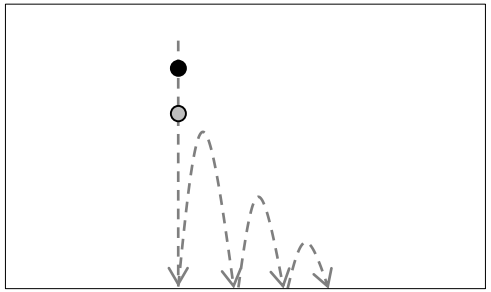




rebond lorsque la forme sort de la zone de dessin on change le signe de dx ou dy



Animer les formes avec un mouvement similaire aux chenilles

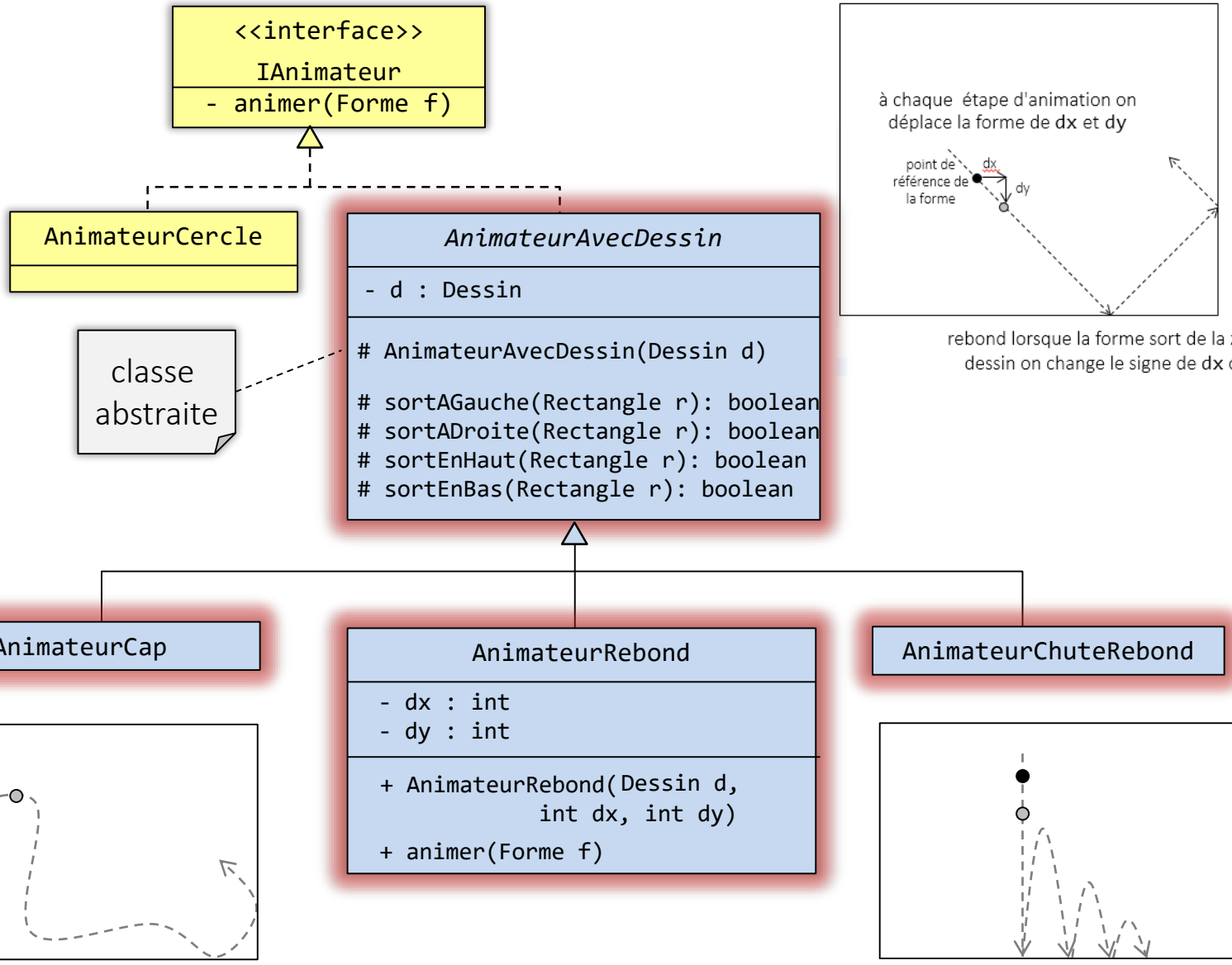


Animer les formes avec un mouvement de chute avec rebond

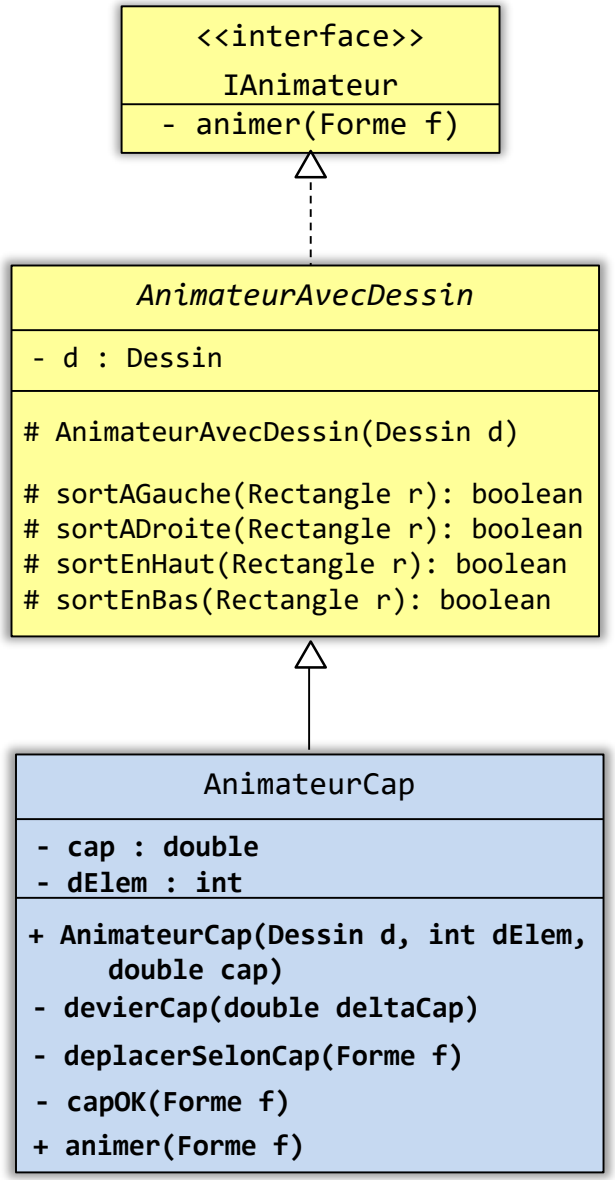
D'autres animateurs que l'**AnimateurRebond** peuvent avoir besoin de connaître la position de la forme par rapport à la zone de dessin.

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

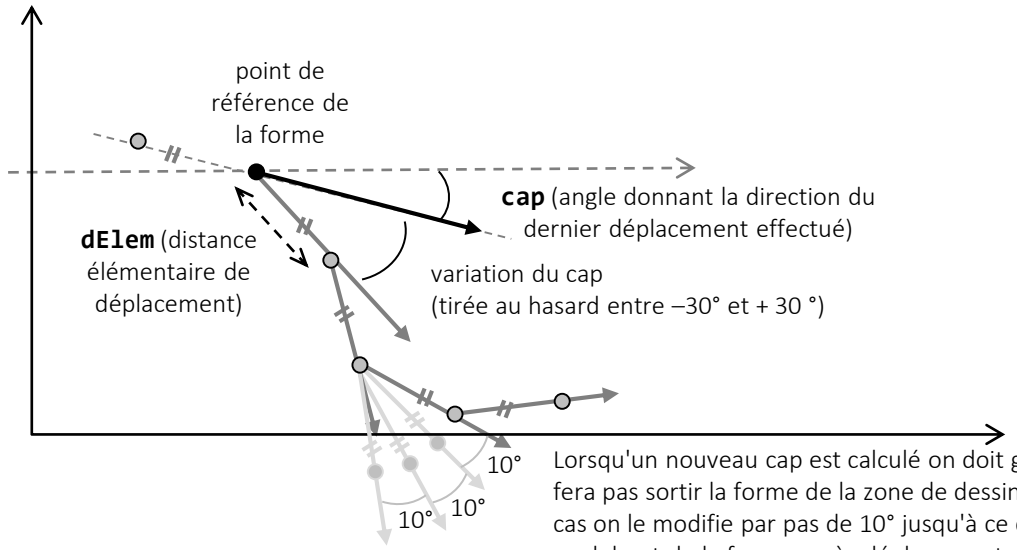
La classe AnimateurAvecDessin



La classe AnimateurCap



Animer les formes avec un mouvement similaire aux chenilles



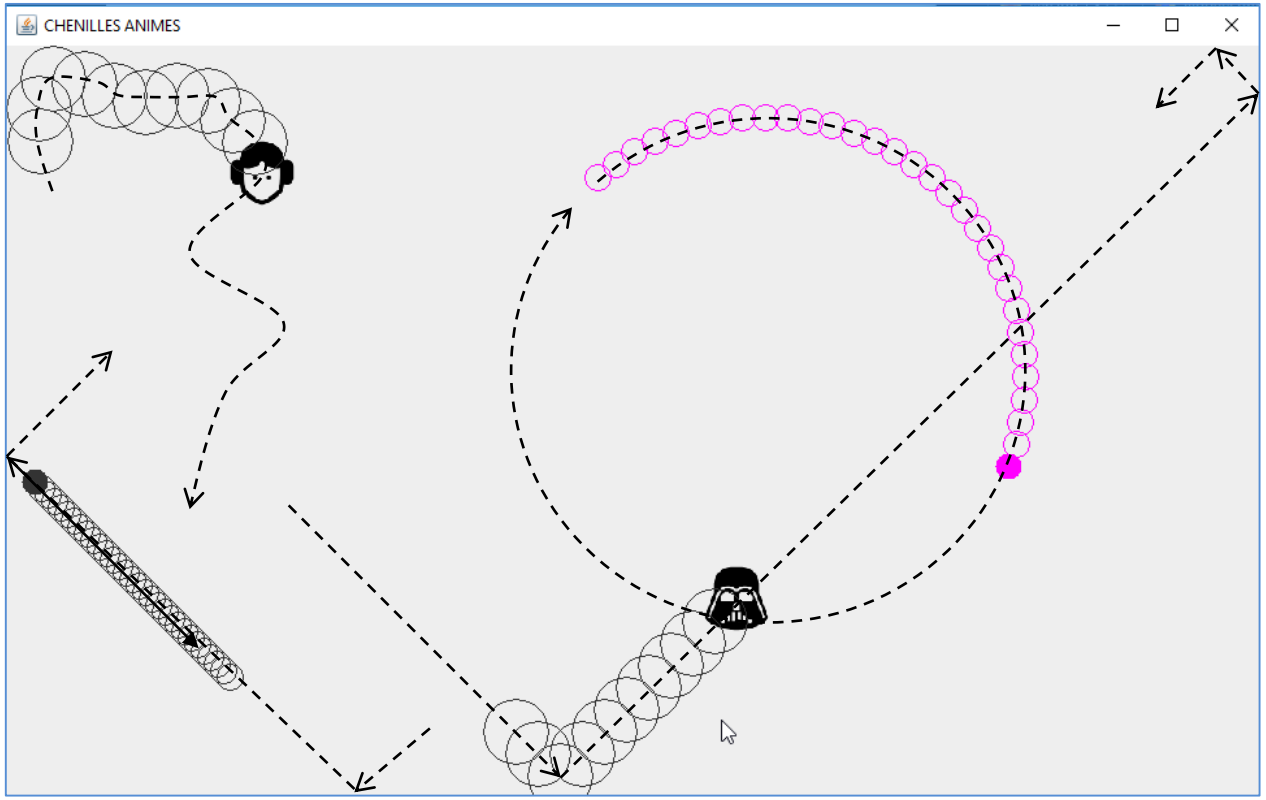
Lorsqu'un nouveau cap est calculé on doit garantir qu'il ne fera pas sortir la forme de la zone de dessin. Si ce n'est pas le cas on le modifie par pas de 10° jusqu'à ce que le rectangle englobant de la forme après déplacement soit entièrement dans le dessin

```
@Override
public void animer(Forme f) {
    // calcule un nouveau cap qui garanti que la forme reste dans la zone
    // de dessin
    this.devierCap(-30.0 + Math.random() * 60.0);
    while (! capOK(f)) {
        this.devierCap(10);
    }
    // fait avancer la forme
    this.deplacerSelonCap(f);
}
```

```
private boolean capOK(IForme f) {
        Rectangle r = new Rectangle(f.getRectEnglobant());
        r.translate(
             (int) (deplacementElem * Math.cos(Math.toRadians(cap))),
             (int) (deplacementElem * Math.sin(Math.toRadians(cap)))
         );
        return !sortAGauche(r) && !sortADroite(r)
                    && !sortEnHaut(r) && !sortEnBas(r);
    }
```

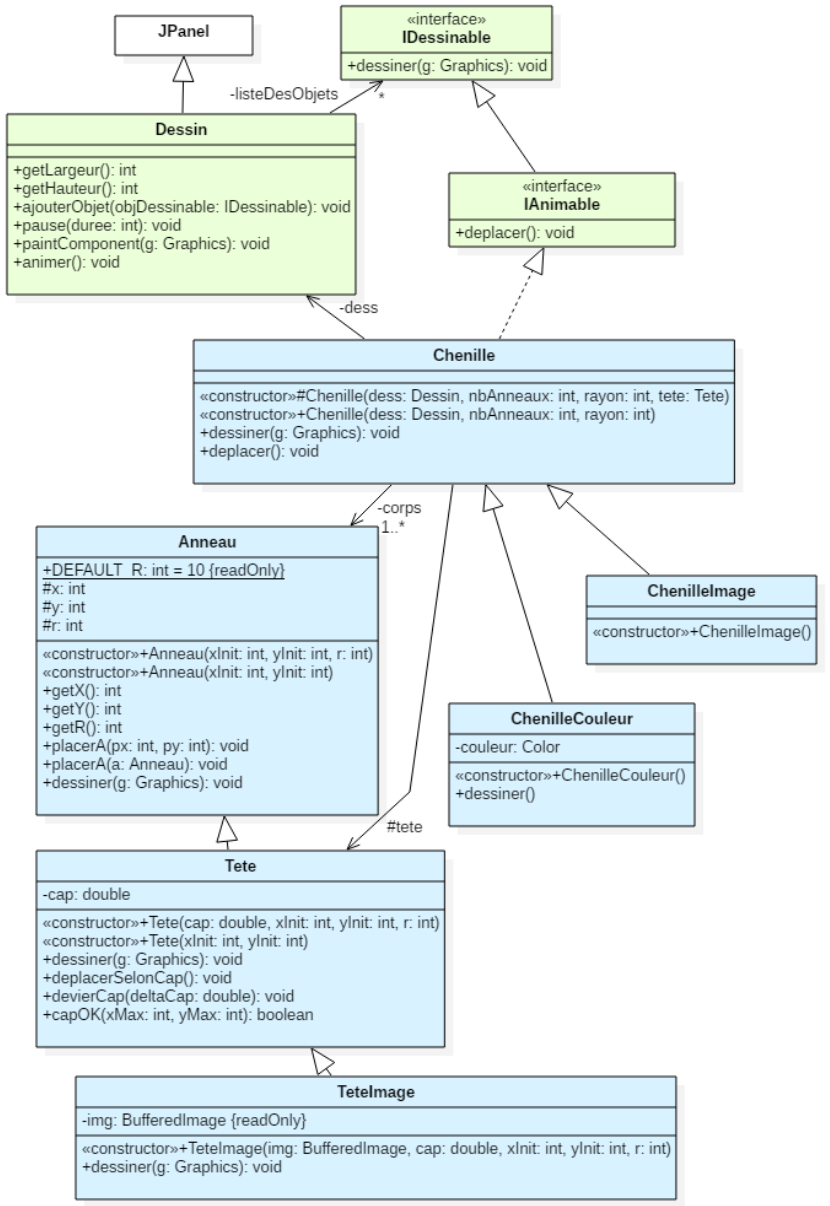
Animer les Chenilles

- Comme pour les formes on souhaite pouvoir animer les chenilles avec différents mouvements.



- Chenille 'normale' avec mouvement rectiligne avec rebond
- Chenille couleur avec mouvement circulaire uniforme
- Chenille 'Leila' avec déplacement aléatoire
- Chenille 'Vador' avec mouvement rectiligne avec rebond

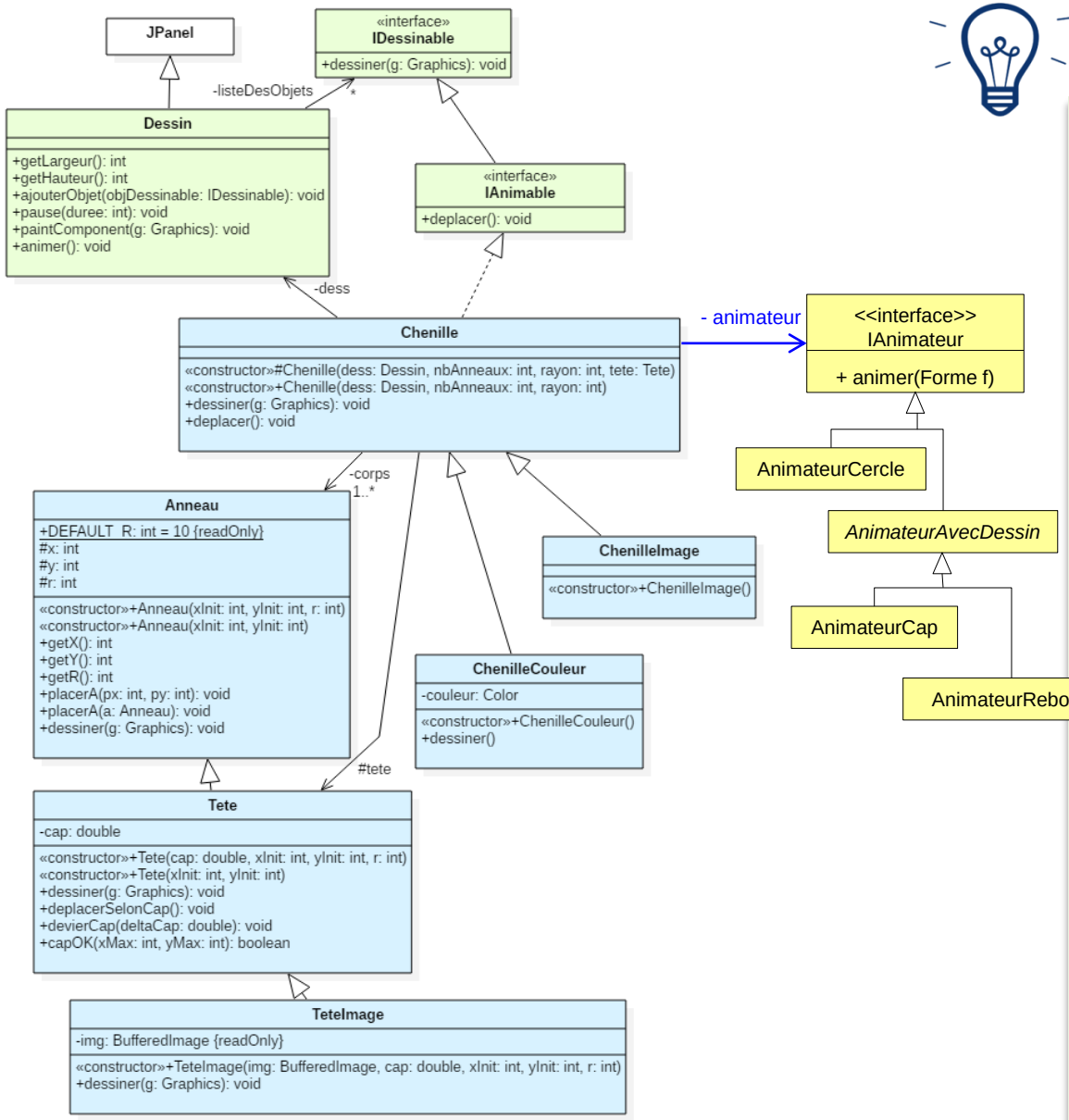
Animer les chenilles



Comment procéder en réutilisant le code des animateurs écrit précédemment ?

```
public class Chenille implements IAnimable {

    protected Tete tete;
    private Anneau[] corps;
    private Dessin dess;

    public Chenille(Dessin dess, int nbAnneaux, int rayon) {
      ...
    }

    public void dessiner(Graphics g) {
       ...
    }

    @Override
    public void deplacer() {
       // déplacer les anneaux l'anneau i prend la place de l'anneau i-1
       for (int i = corps.length - 1; i > 0; i--) {
            corps[i].placerA(corps[i - 1]);
        }
        // l'anneau 0 prend la place de la tête
        corps[0].placerA(tete);

        // modifier le cap de manière aléatoire en garantissant
        // que le nouveau cap maintient la tête dans la zone de dessin
        tete.devierCap(Math.random() * 60 - 30);
        while (!tete.capOK(this.dess.getLargeur(), this.dess.getHauteur())) {
            tete.devierCap(10.0);
        }

        // déplacer la tête
        tete.deplacerSelonCap();
    }
}
```

Animer les chenilles



Confier l'animation de la tête à un objet IAnimateur

```
public class Chenille implements IAnimable {

    protected Tete tete;
    private Anneau[] corps;
    private Dessin dess;

    public Chenille(Dessin dess, int nbAnneaux, int rayon) {
      ...
    }

    public void dessiner(Graphics g) {
       ...
    }

    @Override
    public void deplacer() {
       // déplacer les anneaux l'anneau i prend la place de l'anneau i-1
       for (int i = corps.length - 1; i > 0; i--) {
            corps[i].placerA(corps[i - 1]);
        }
        // l'anneau 0 prend la place de la tête
        corps[0].placerA(tete);

        // modifier le cap de manière aléatoire en garantissant
        // que le nouveau cap maintient la tête dans la zone de dessin
        tete.devierCap(Math.random() * 60 - 30);
        while (!tete.capOK(this.dess.getLargeur(), this.dess.getHauteur())) {
            tete.devierCap(10.0);
        }

        // déplacer la tête
        tete.deplacerSelonCap();
    }
}
```

rajouter un attribut

```
private IAnimateur animateur;
```

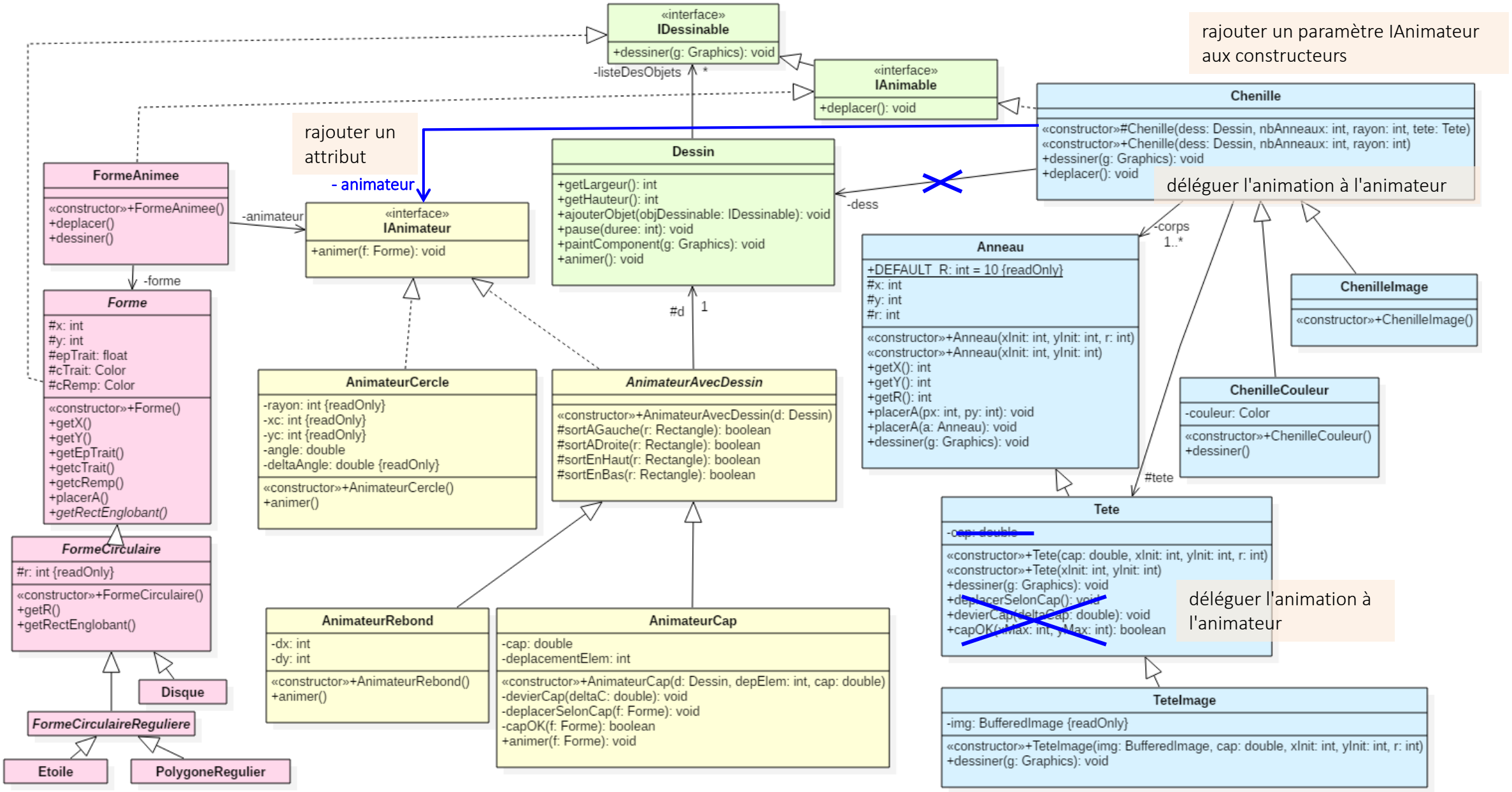
rajouter un paramètre aux constructeurs

```
IAnimateur animateur;
```

```
animateur.animer(tete);
```

déléguer l'animation à l'animateur

Animer les Chenilles



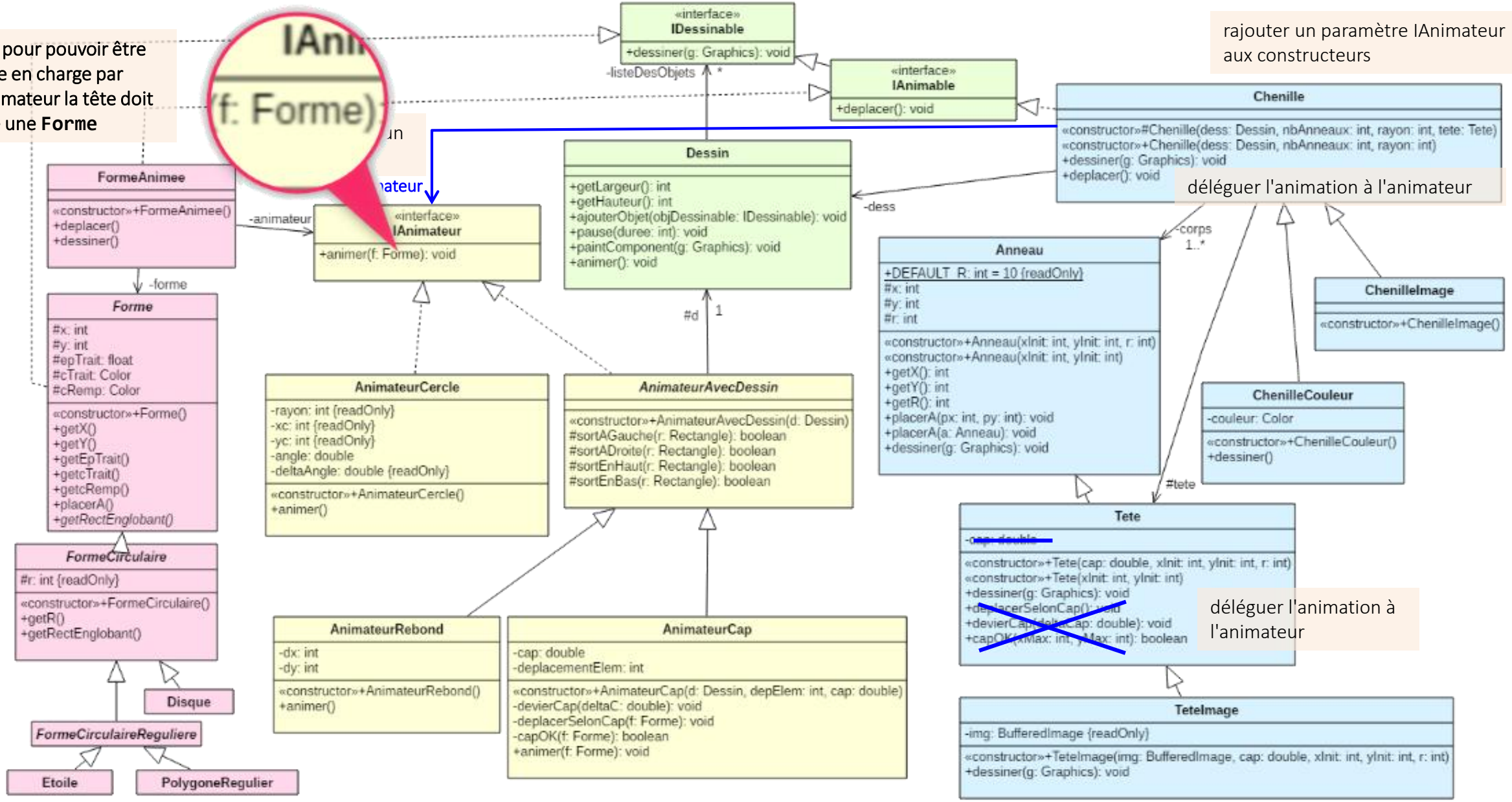
Animer les Chenilles

Pb : pour pouvoir être prise en charge par l'animateur la tête doit être une **Forme**

rajouter un paramètre IAnimateur aux constructeurs

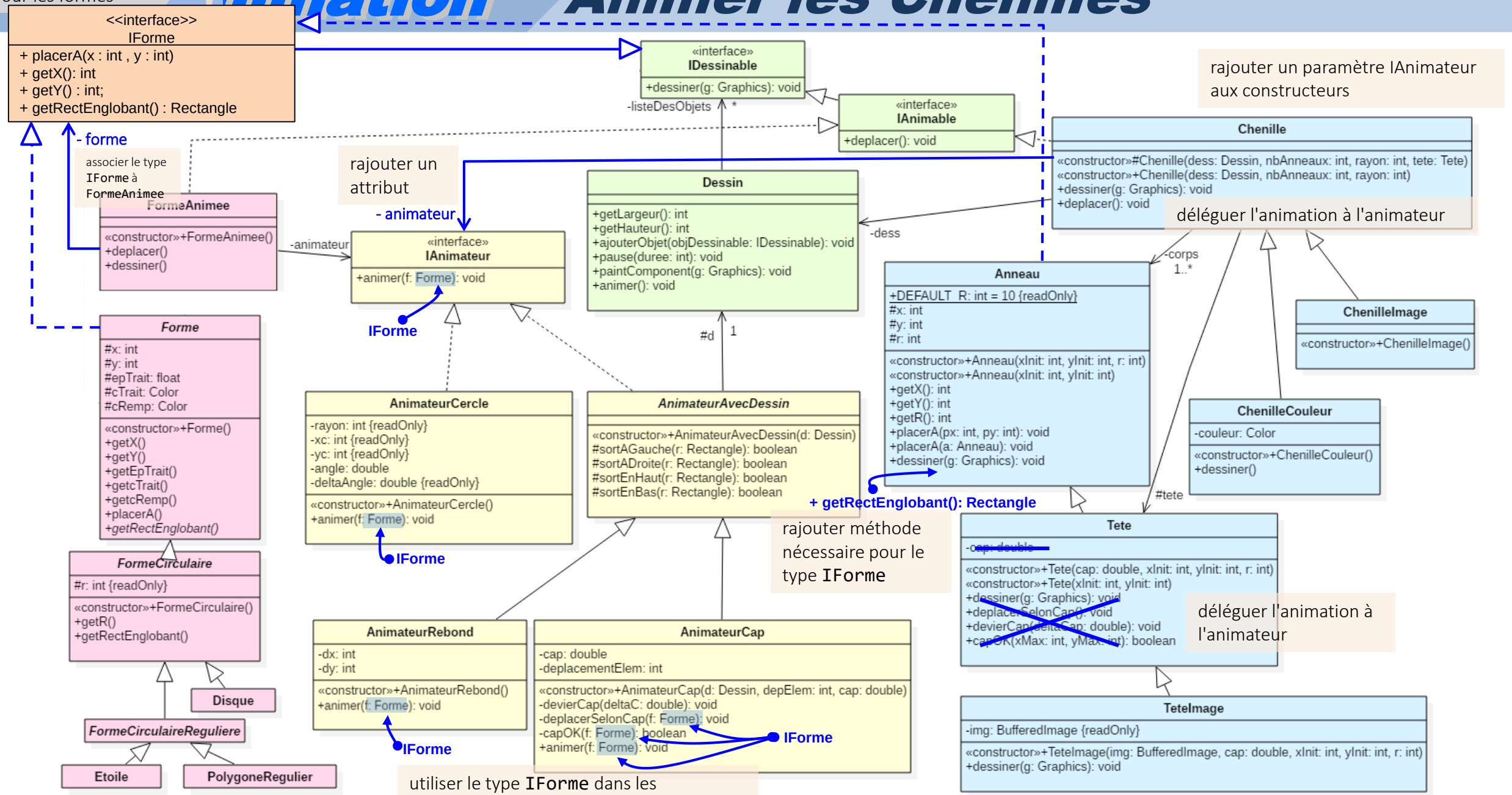
déléguer l'animation à l'animateur

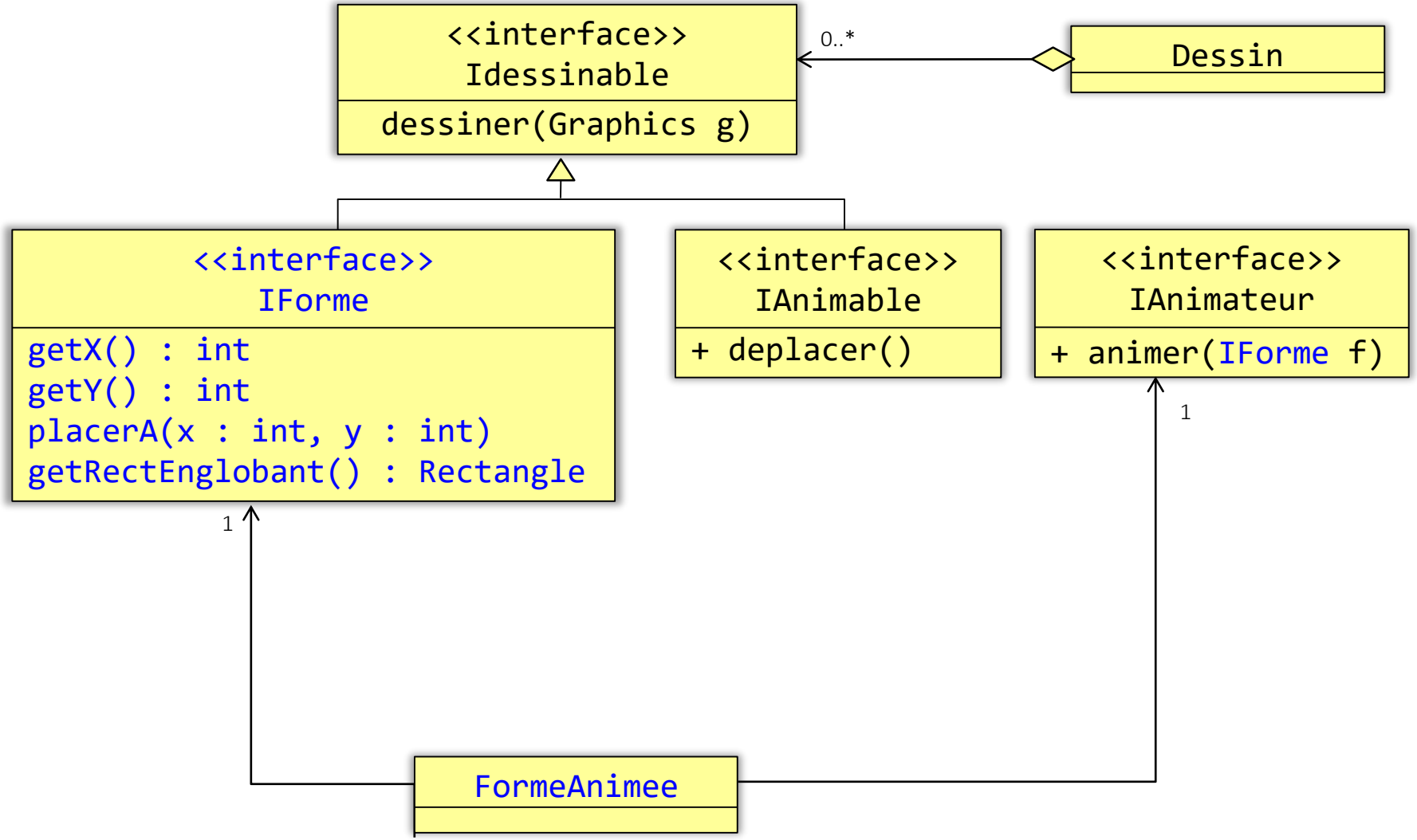
déléguer l'animation à l'animateur



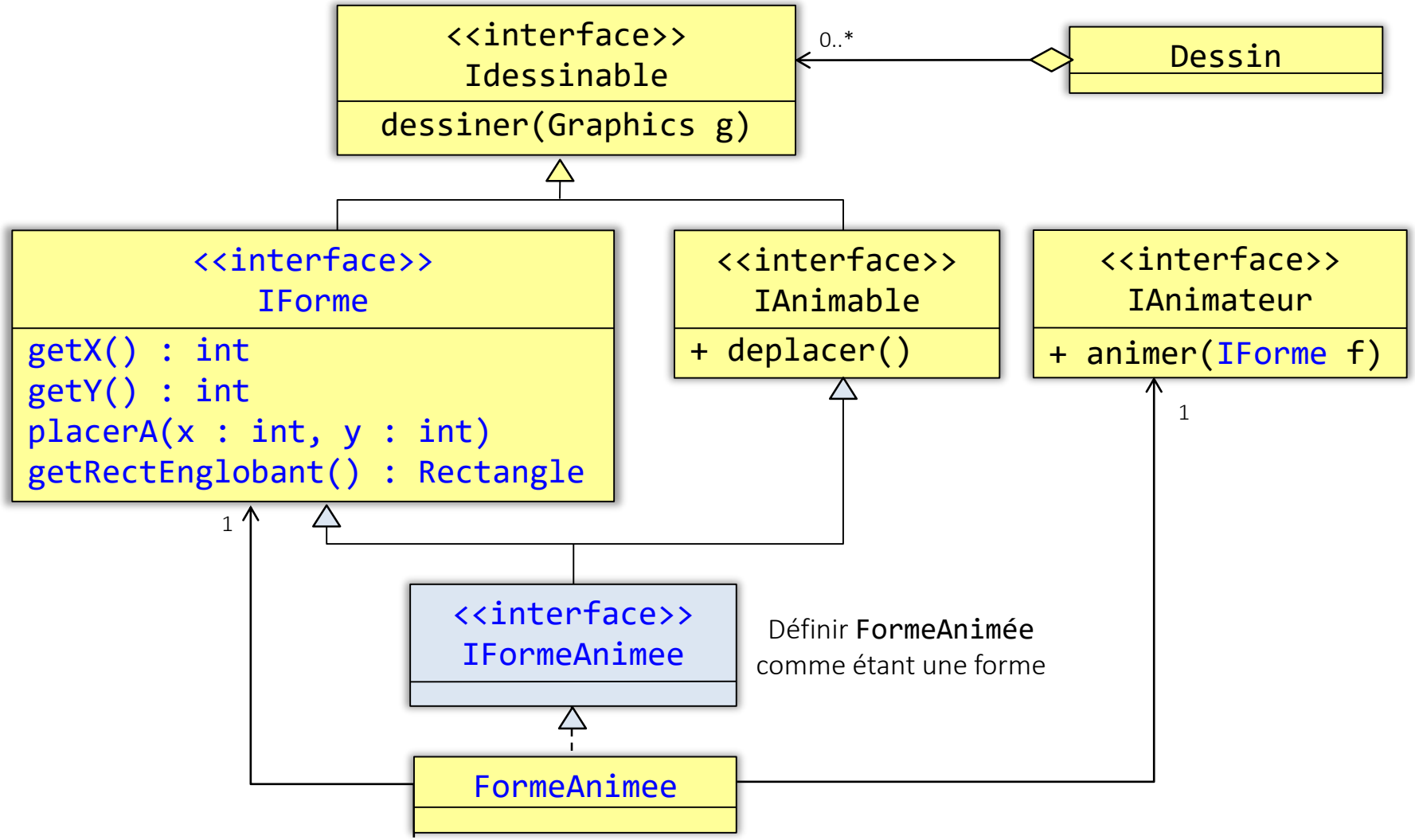
Animer les Chenilles

définir un type abstrait pour les formes

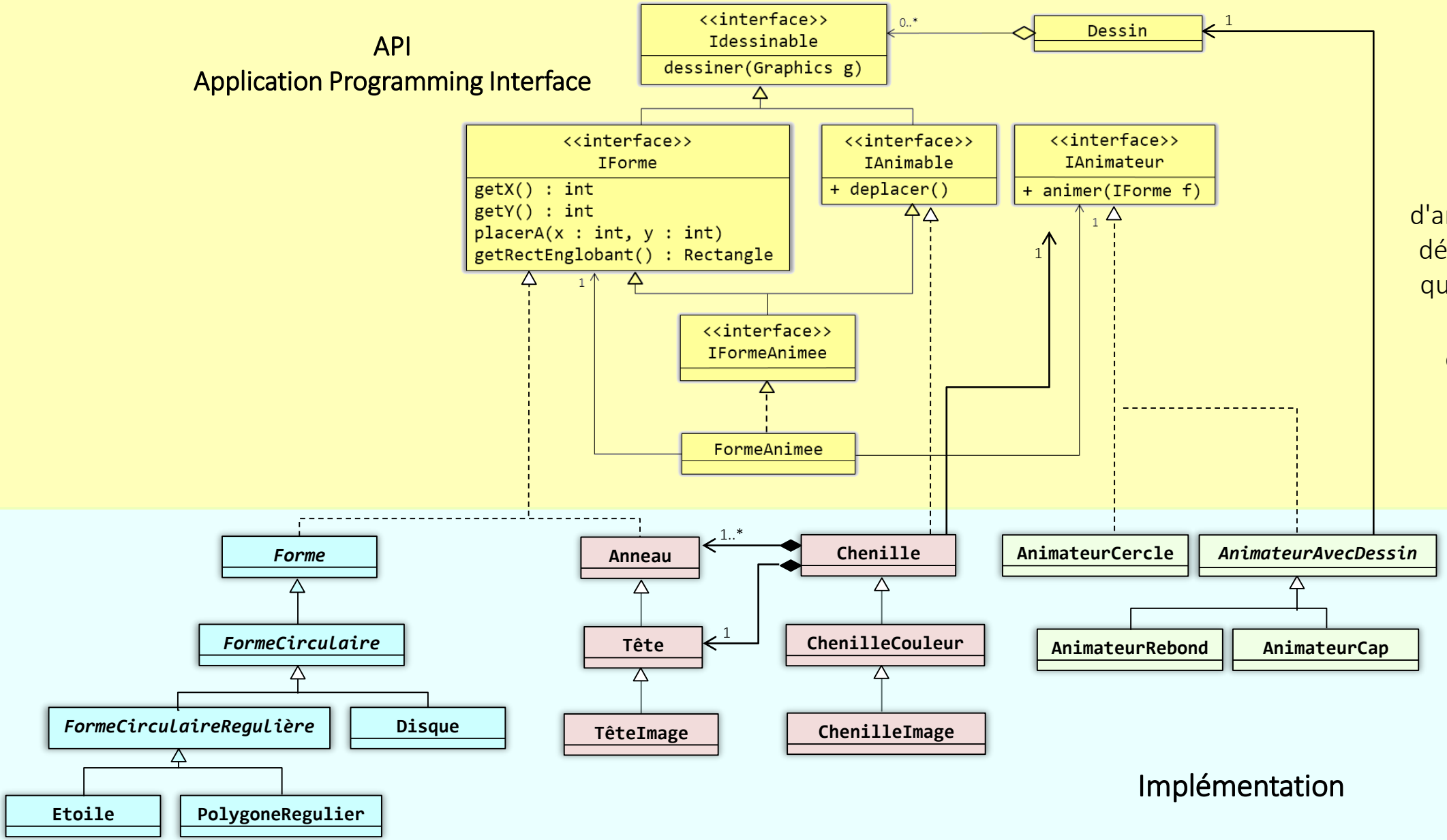




- Les formes animées ne peuvent pas être vues comme des formes…


<<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



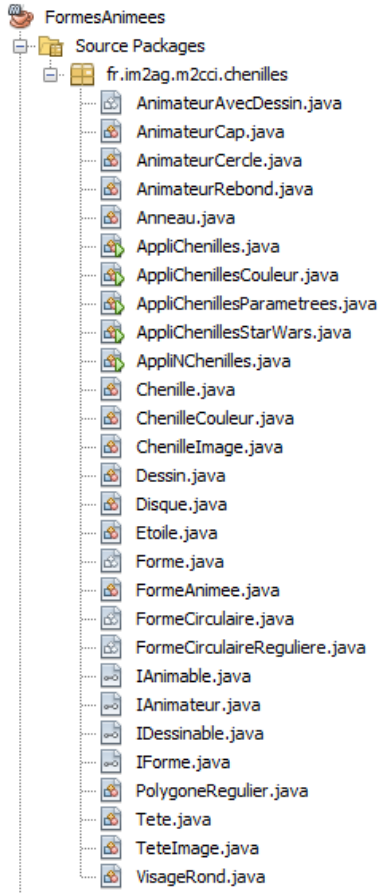
Implémentation

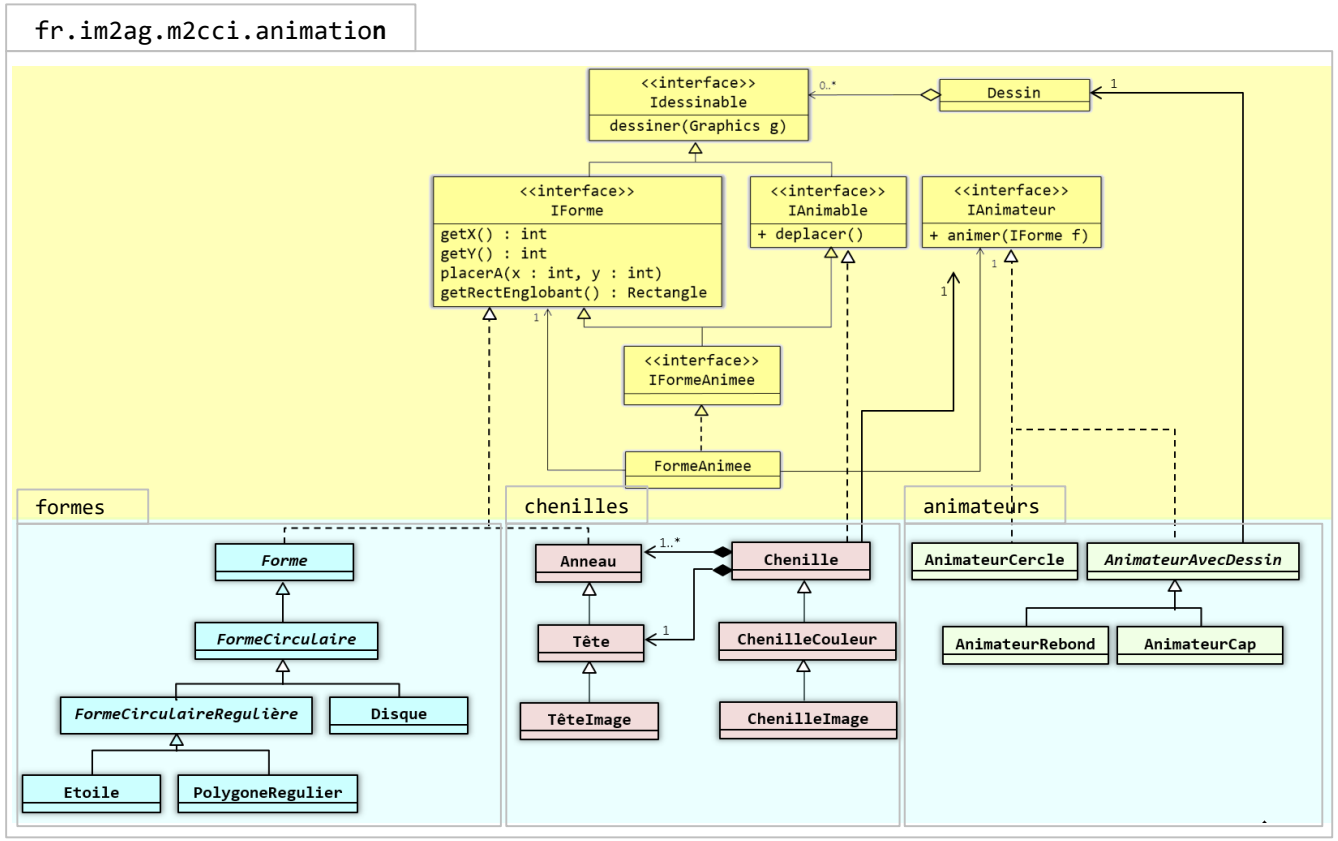
une application d'animation peut être définie en n'utilisant que les types donné dans l'API

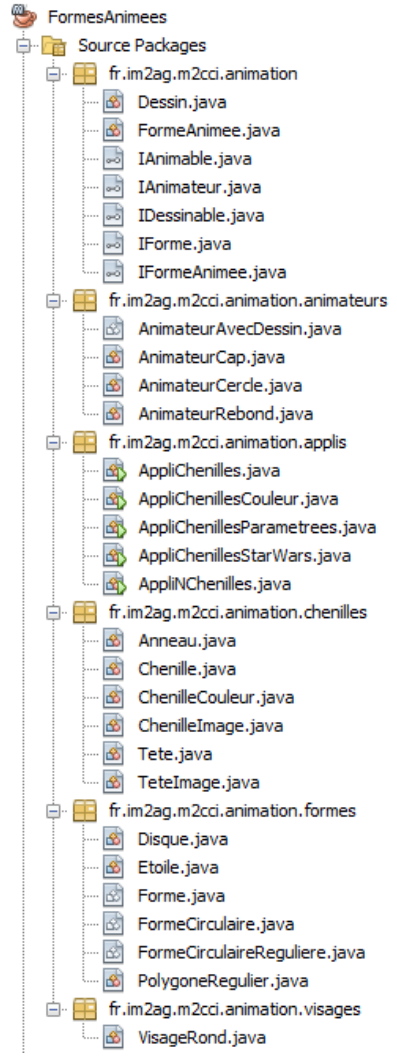
en instanciant des objets de l'implémentation fournie

l'implémentation peut facilement être étendue ou remplacée par une autre implémentation

- Refactoring en packages







Formes et rectangle englobant

- comment gérer le rectangle englobant au niveau des formes ?

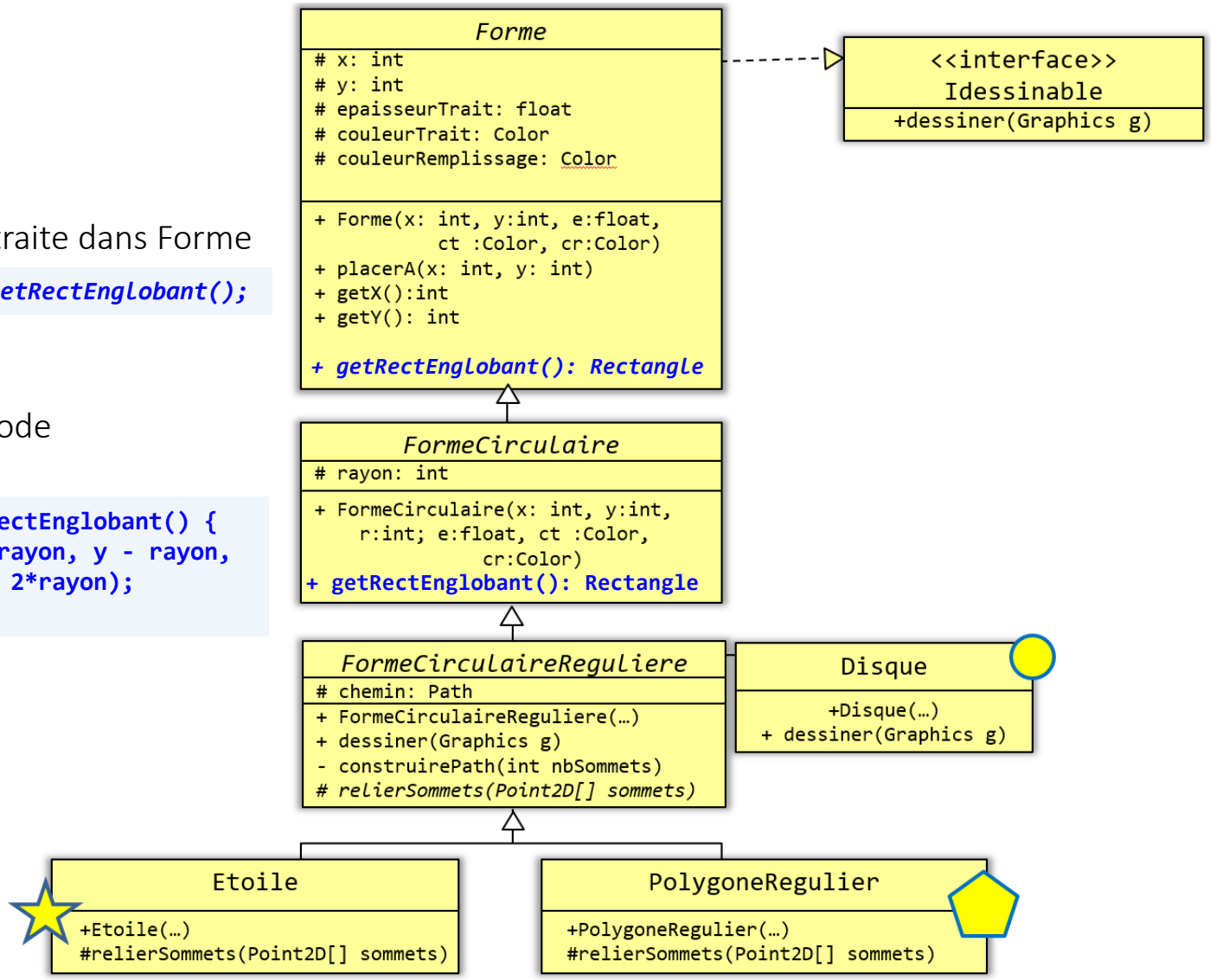
solution 1:

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 en particulier l'opération d'instanciation

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;
```

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

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

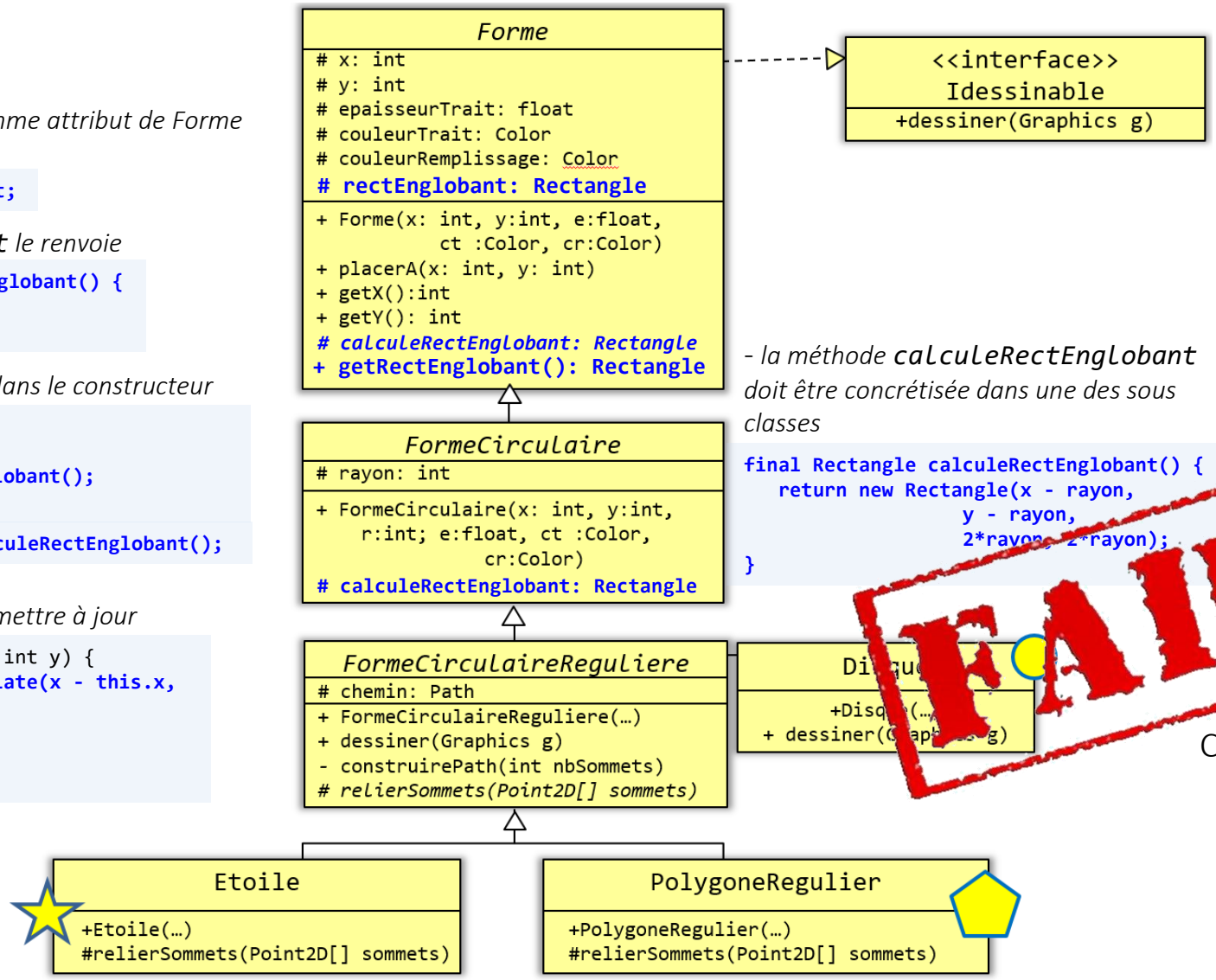
- le rectangle englobant initialisé dans le constructeur

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

```
protected abstract Rectangle calculeRectEnglobant();
```

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

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

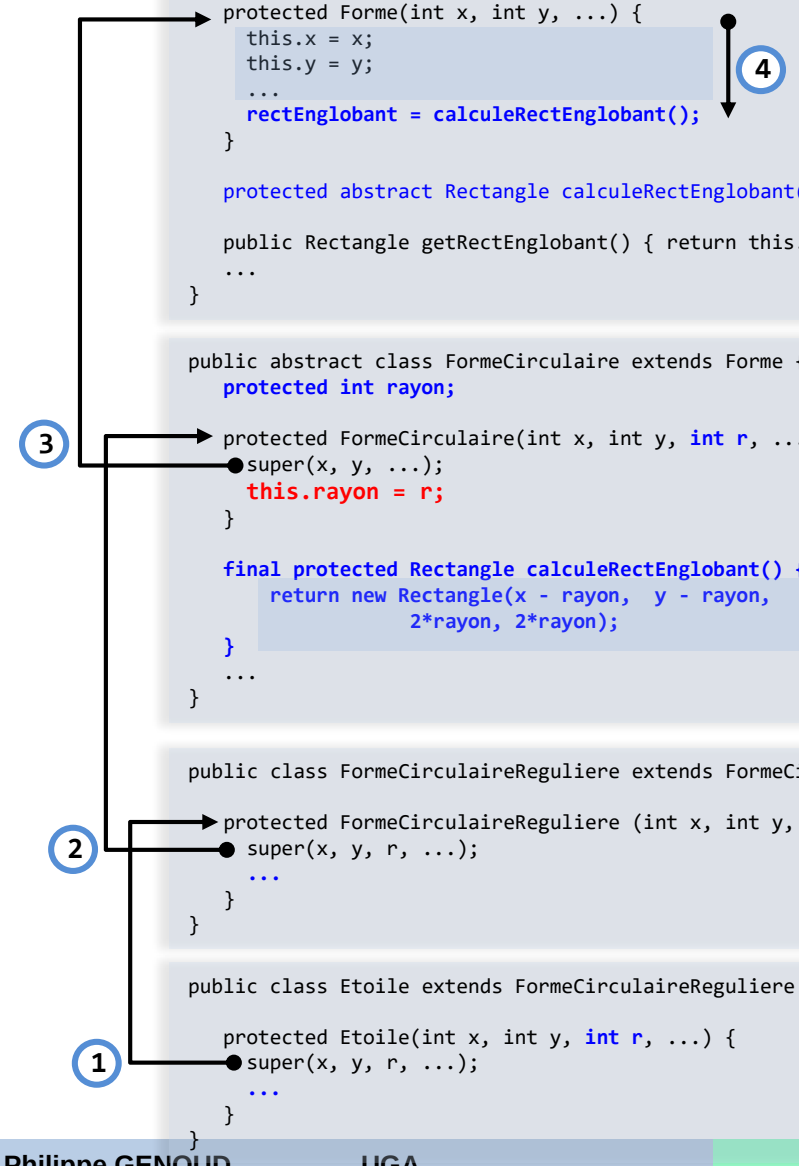
Formes et rectangle englobant

```
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 FormeCirculaire(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);
   }
   ...
}
```

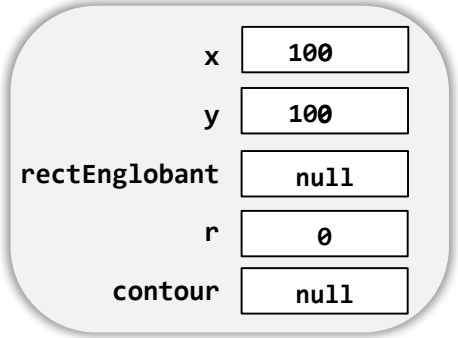
```
public class FormeCirculaireReguliere extends FormeCirculaire {

   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, ...);
     ...
   }
}
```




```
new Etoile(100,100, 50, …);
```

a) allocation de la mémoire
(tous les attributs sont initialisés avec une valeur nulle)



b) exécution du constructeur

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

- échec à cause du chaînage des constructeurs

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 FormeCirculaire(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 `getRectEnglobant`