DESIGN PATTERNS

PLAN DU COURS

- Diagrammes de classes UML
- Principes SOLID :
  - Single Responsibility Principle (SRP)
  - Open/Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
- Patrons de conception

Développement d'une application

Les différentes phases du développement d'une application :

- Définition du cahier des charges
- Conception de l'architecture générale
- Conception détaillée
- Implémentation
- Tests unitaires
- Intégration
- Qualification (vérification de la conformité aux spécifications)
- Mise en production
- Maintenance : correction des bugs + **évolutions**

Les évolutions

Prise en compte d'une évolution :

- Modification du cahier des charges
- Adaptation de l'architecture
- Implémentation des nouvelles fonctionnalités
- Tests unitaires
- Intégration
- Qualification (vérification de la conformité aux spécifications)
- Mise en production

Programme bien conçu

Un programme est **"bien conçu"** s'il permet de :

- Absorber les changements avec un minimum d'effort
- Implémenter les nouvelles fonctionnalités sans toucher aux anciennes
- Modifier les fonctionnalités existantes en modifiant localement le code

Objectifs :

- Limiter les modules impactés
  - ⇒ Simplifier les tests unitaires
  - ⇒ Rester conforme à la partie des spécifications qui n'ont pas changé
  - ⇒ Facilité l'intégration
- Gagner du temps

Remarque : Le développement d'une application est une suite d'évolutions

Les 5 principes (pour créer du code) SOLID

- Single Responsibility Principle (SRP) :
  Une classe ne doit avoir qu'une seule responsabilité
- Open/Closed Principle (OCP) :
  Programme ouvert pour l'extension, fermé à la modification
- Liskov Substitution Principle (LSP) :
  Les sous-types doivent être substituables par leurs types de base
- Interface Segregation Principle (ISP) :
  Éviter les interfaces qui contiennent beaucoup de méthodes
- Dependency Inversion Principle (DIP) :
  Les modules d'un programme doivent être indépendants
  Les modules doivent dépendre d'abstractions

Single Responsability Principle (SRP)

Principe :

Une classe ne doit avoir qu'une seule responsabilité (Robert C. Martin).

Signification et objectifs :

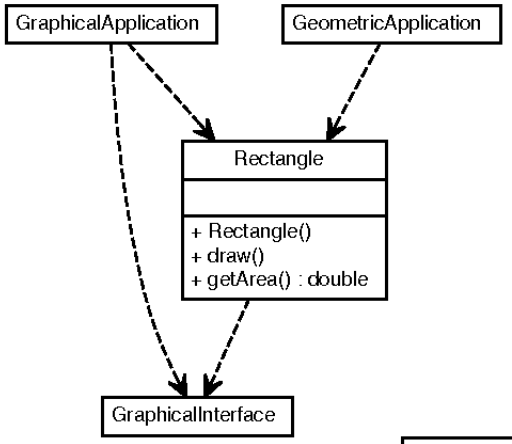
- Une responsabilité est une **"raison de changer"**
- Une classe ne doit avoir qu'une seule raison de changer

Pourquoi ?

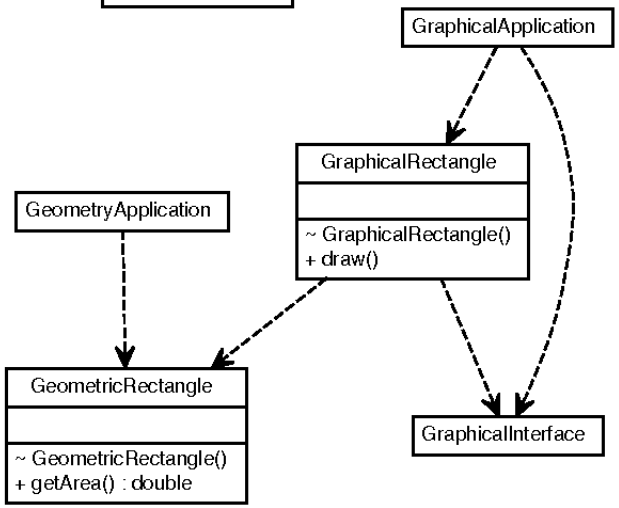
- Si une classe a plusieurs responsabilités, elles sont couplées
- Dans ce cas, la modification d'une des responsabilités nécessite de :
  - tester à nouveau l'implémentation des autres responsabilités
  - modifier potentiellement les autres responsabilités
  - déployer à nouveau les autres responsabilités
- Donc, vous risquez de :
  - introduire des bugs
  - rendre moins locales les modifications

Single Responsability Principle (SRP)

Violation de SRP :



Séparation des responsabilités :



Open/Closed Principle (OCP)

Principe :

Programme ouvert pour l'extension, fermé à la modification

Signification :

Vous devez pouvoir ajouter une nouvelle fonctionnalité :

- en ajoutant des classes (Ouvert pour l'extension)
- sans modifier le code existant (fermé à la modification)

Avantages :

- Le code existant n'est pas modifié $\Rightarrow$ augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

Open/Closed Principle (OCP)

```
class Rectangle { public Point p1, p2; }
class Circle { public Point c; public int radius; }

class GraphicTools {
    static void drawRectangle(Rectangle r) { ... }
    static void drawCircle(Circle c) { ... }

    static void drawShapes(Object[] shapes) {
        for (Object o : shapes) {
            if (o instanceof Rectangle) {
                Rectangle r = (Rectangle)o;
                drawRectangle(r);
            }
            else if (o instanceof Circle) {
                Circle c = (Circle)o;
                drawCircle(c);
            }
        }
    }
}
```

Open/Closed Principle (OCP)

Solution : Abstraction et interfaces !

```
interface Shape { public void draw(); }

class Rectangle implements Shape {
        public Point p1, p2;
        void draw() { ... }
}

class Circle implements Shape {
        public Point c; public int radius;
        void draw() { ... }
}

class GraphicTools {
    static void drawShapes(Shape[] shapes) {
        for (Shape s : shapes)
            s.draw();
    }
}
```

Liskov Substitution Principle (LSP)

Principe :

Les sous-types doivent être substituables par leurs types de base

Signification :

Si une classe **A** étend une classe **B** (ou implémente une interface **B**) alors un programme **P** écrit pour manipuler des instances de type **B** doit avoir le même comportement s'il manipule des instances de la classe **A**.

Avantages :

- Diminution de la complexité du code
- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables

Liskov Substitution Principle (LSP)

Une classe qui permet de représenter un rectangle géométrique :

```
public class Rectangle {
    private double w;
    private double h;

    public void setWidth(double w) { this.w = w; }
    public void setHeight(double h) { this.h = h; }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
    public double getArea() { return w*h; }
}
```

Liskov Substitution Principle (LSP)

Un carré **est** un rectangle donc on devrait pouvoir écrire :

```
public class Square extends Rectangle {

    public void setWidth(double w) {
        super.setWidth(w);
        super.setHeight(w);
    }

    public void setHeight(double h) {
        super.setWidth(h);
        super.setHeight(h);
    }

}
```

Liskov Substitution Principle (LSP)

Violation de LSP :

```
public void test(Rectangle r) {
        r.setWidth(2);
        r.setHeight(3);
        if (r.getArea()!=3*2)
            System.out.println("bizarre␣!");
}
```

La mauvaise question :

Un carré **est**-il un rectangle ?

La bonne question :

Pour les utilisateurs,
votre carré **a**-t-il le même **comportement** que **votre** rectangle ?

La réponse :

Dans ce cas, **NON**

Liskov Substitution Principle (LSP)

Une solution :

```
public abstract class RectangularShape {
    public abstract double getWidth();
    public abstract double getHeight();
    public double getArea() { return getWidth()*getHeight(); }
}

public class Rectangle extends RectangularShape {
    private double w;
    private double h;

    public void setWidth(double w) { this.w = w; }
    public void setHeight(double h) { this.h = h; }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
}
```

Liskov Substitution Principle (LSP)

Suite de la solution :

```
class Square extends RectangularShape {
    private double s;
    public void setSideLength(double s) { this.s = s; }
    public double getWidth() { return s; }
    public double getHeight() { return s; }

}
```

Utilisation :

```
public void testRectangle(Rectangle r) {
        r.setWidth(2); r.setHeight(3);
        if (r.getArea()!=3*2) System.out.println("jamais !");
}

public void testSquare(Square s) {
        s.setSideLength(2);
        if (s.getArea()!=2*2) System.out.println("jamais !");
}
```

Interface Segregation Principle (ISP)

Principe :

Éviter les interfaces qui contiennent beaucoup de méthodes

Signification et objectifs :

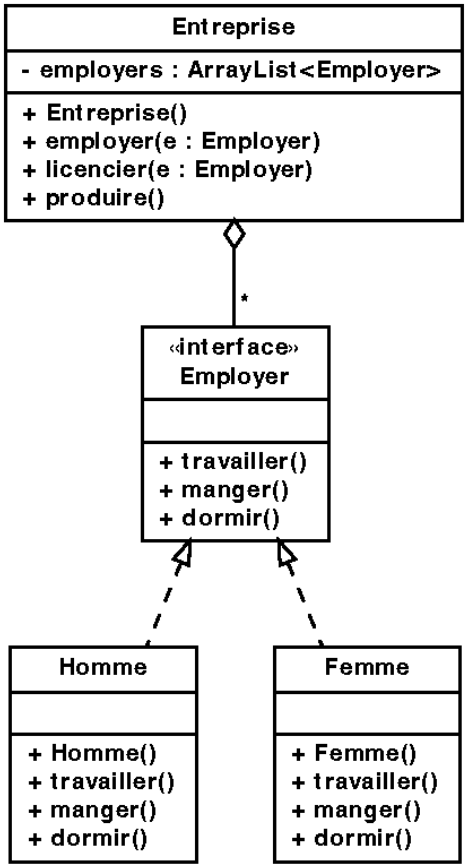
- Découper les interfaces en responsabilités distinctes (SRP)
- Quand une interface grossit, se poser la question du rôle de l'inferface
- Éviter de devoir implémenter des services qui n'ont pas à être proposés par la classe qui implémente l'interface
- Limiter les modifications lors de la modification de l'interface

Avantages :

- Le code existant est moins modifié $\Rightarrow$ augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

Interface Segregation Principle (ISP)

Exemple de violation de ISP :



Interface Segregation Principle (ISP)

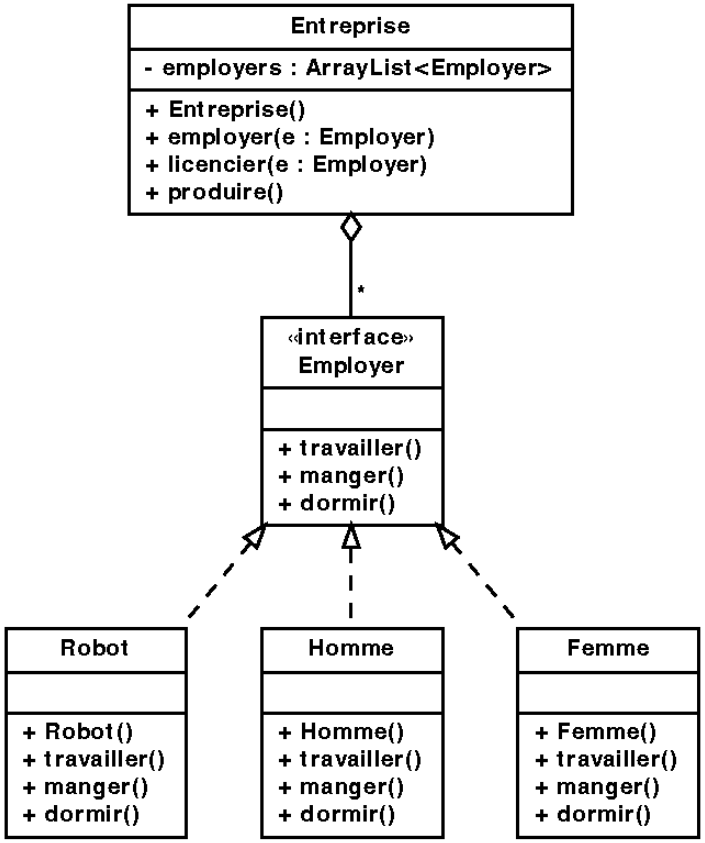
```
public interface Employer {
    void travailler();
    void manger();
    void dormir();
}

public class Femme implements Employer {
    public void travailler() {
        System.out.println("Je suis une femme et je travaille");
    }
    public void manger() {
        System.out.println("Je suis une femme et je mange");
    }
    public void dormir() {
        System.out.println("Je suis une femme et je dors");
    }
}
```

Interface Segregation Principle (ISP)

```
public class Entreprise {
    private final ArrayList<Employer> employers;

    public Entreprise() {
        employers = new ArrayList<Employer>();
    }

    public void employer(Employer e) {
        employers.add(e);
    }

    public void licencier(Employer e) {
        employers.remove(e);
    }

    public void produire() {
        for (Employer e : employers)
            e.travailler();
    }
}
```

Interface Segregation Principle (ISP)

Un robot peut travailler :



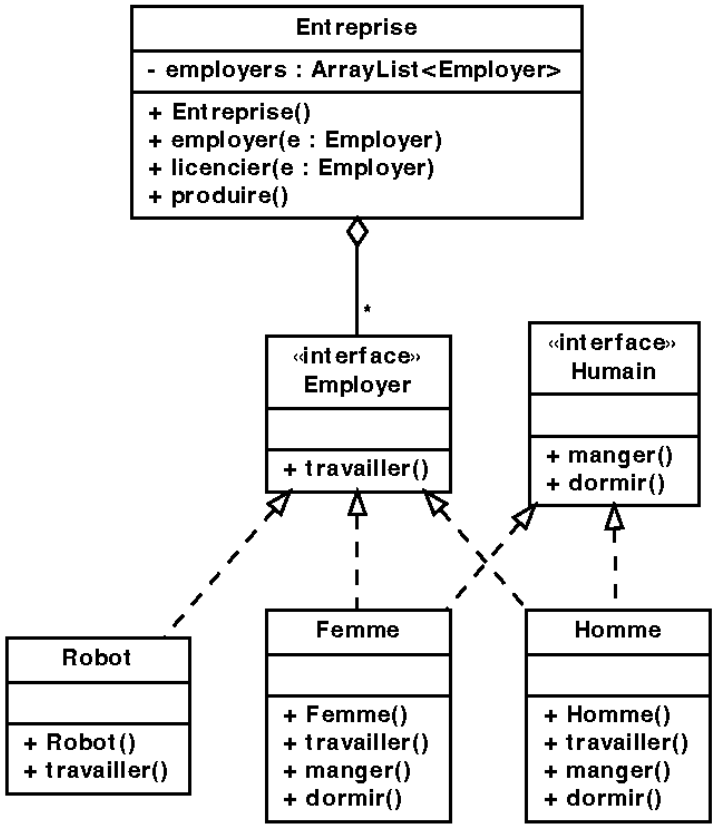
Interface Segregation Principle (ISP)

Problème : un robot est un travailleur qui ne sait pas dormir et manger

```
public class Robot implements Employer {
    public void travailler() {
        System.out.println("je travaille");
    }

    public void manger() {
        System.out.println("je ne sais pas le faire");
    }

    public void dormir() {
        System.out.println("je ne sais pas le faire");
    }
}
```

Interface Segregation Principle (ISP)

Solution : découper l'interface *Travailleur* en deux



Dependency Inversion Principle (DIP)

Principe :

Les modules d'un programme doivent être indépendants
Les modules doivent dépendre d'abstractions

Signification et objectifs :

- Découpler les différents modules de votre programme
- Les lier en utilisant des interfaces
- Décrire correctement le comportement de chaque module
- Permet de remplacer un module par un autre module plus facilement

Avantages :

- Les modules sont plus facilement réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités
- L'intégration est rendue plus facile