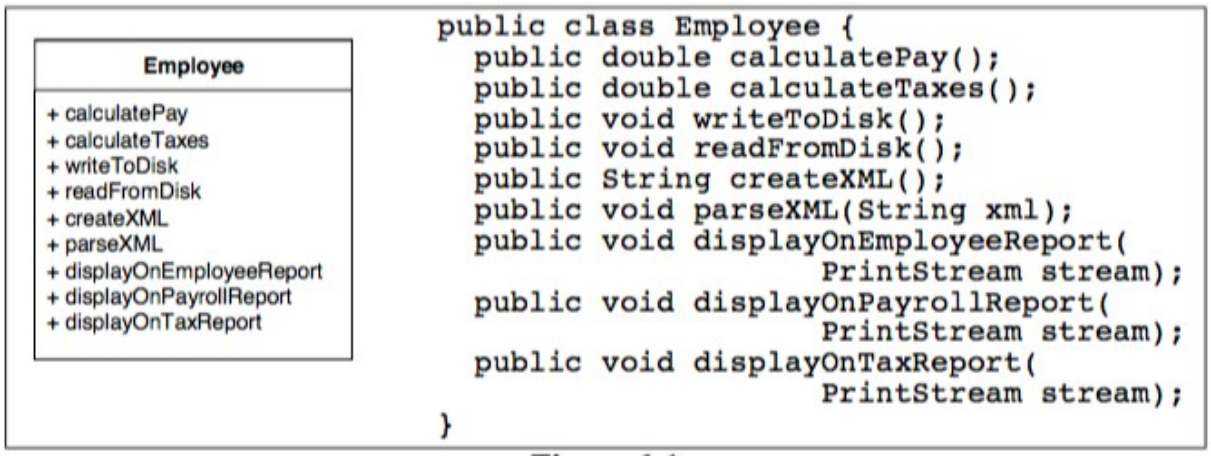
SOLID

Objectifs

Comprendre et savoir appliquer les principes SOLID.

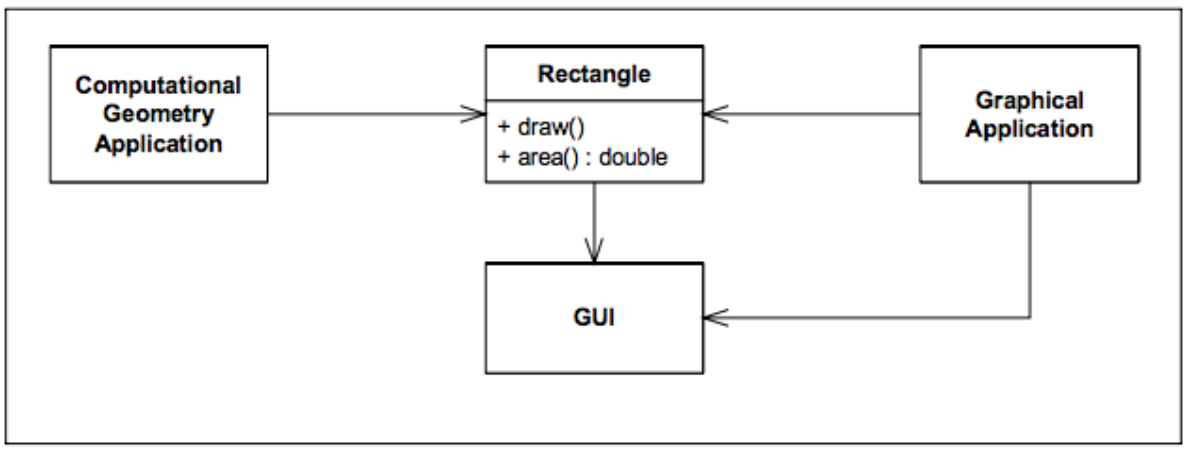
Exercice 1 – Single Responsability Principle

Voici une classe *fragile* car elle a trop de responsabilités (calcul de paye et de taxe, persistance par lecture/écriture dans un fichier, conversion lecture/écriture vers un format XML, différents formattages pour plusieurs rapports) :

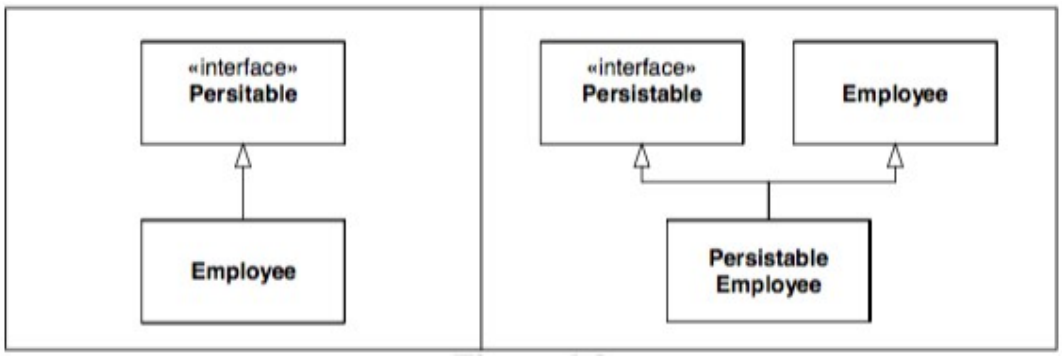


TO DO : Proposez une séparation des préoccupations pour que chaque classe ne fasse qu'une chose et n'ait qu'une seule raison de changer.

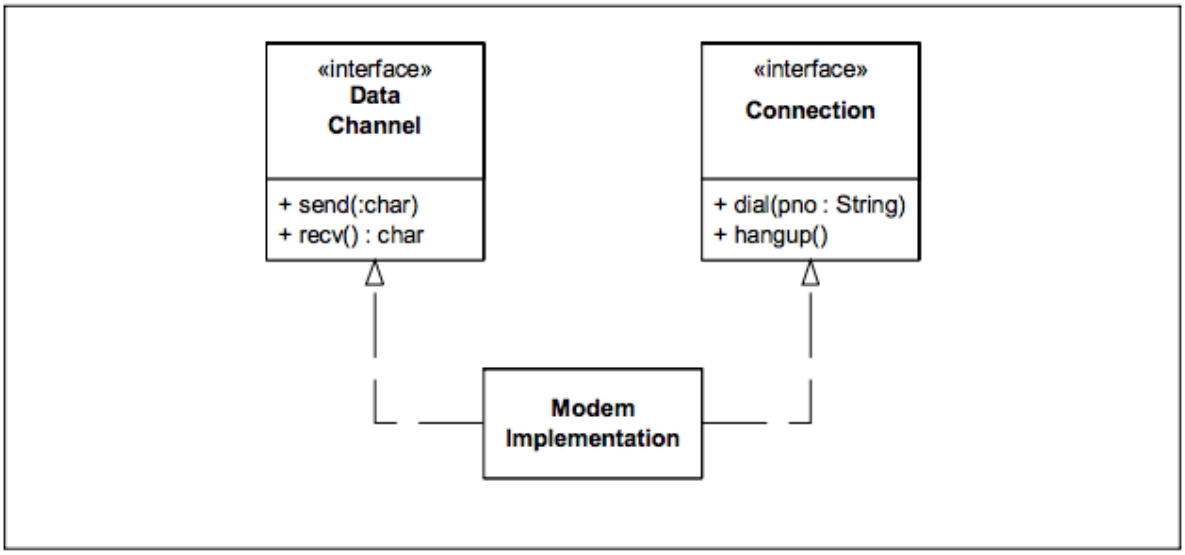
Dans le prochain cas, 2 applications différentes utilisent la même classe Rectangle. L'une réalise seulement des calculs géométriques, l'autre est amenée à dessiner des rectangles à l'écran. La règle du SRP n'est pas respectée : Rectangle a 2 responsabilités. Cela entraine par exemple que la première application doit forcément inclure la classe GUI même si elle ne l'utilise pas. Pire, si l'application graphique devait évoluer et était amenée à modifier la classe Rectangle il faudrait alors recompiler / retester / re-déployer également l'application de calculs géométriques…



TO DO : Proposez une solution qui respecte le SRP



TO DO : Avantages et inconvénients des 2 approches pour gérer la persistence d'employés ?



TO DO : Que pensez vous du cas précédent ? Le SRP est il respecté ?

Exercice 2 – Open-Close Principle

Ci-après un extrait de code (incomplet) et un diagramme de classe associé.

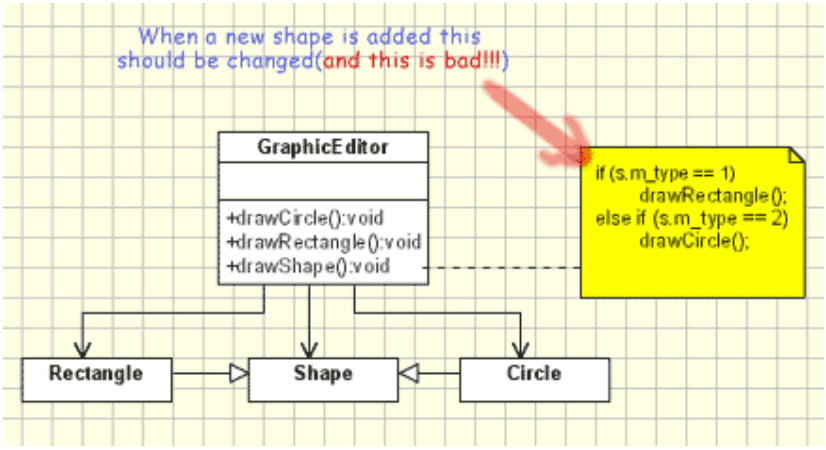
```
class GraphicEditor {

      public void drawShape(Shape s) {
            if (s.m_type==1)
                  drawRectangle(s);
            else if (s.m_type==2)
                  drawCircle(s);
      }
      public void drawCircle(Circle r) {....}
      public void drawRectangle(Rectangle r) {....}
 }

 class Shape {
      int m_type;
 }

 class Rectangle extends Shape {
      Rectangle() {
            super.m_type=1;
      }
 }

 class Circle extends Shape {
      Circle() {
            super.m_type=2;
      }
 }
```



TO DO : Proposez une solution qui respecte le OCP.

Exercice 3 – Liskov's Substitution Principle

Ci-après un extrait de code qui ne respecte pas le LSP.

```
class Rectangle {
          protected int m_width;
          protected int m_height;

          public void setWidth(int width){
                    m_width = width;
          }
          public void setHeight(int height){
                    m_height = height;
          }
          public int getWidth(){
                    return m_width;
          }
          public int getHeight(){
                    return m_height;
          }
          public int getArea(){
                    return m_width * m_height;
          }
}

class Square extends Rectangle {
          public void setWidth(int width){
                    m_width = width;
                    m_height = width;
          }
          public void setHeight(int height){
                    m_width = height;
                    m_height = height;
          }
}

class LspTest{
          private static Rectangle getNewRectangle() {
                    // it can be an object returned by some factory ...
                    return new Square();
          }
          public static void main (String args[])
          {
                    Rectangle r = LspTest.getNewRectangle();

                    r.setWidth(5);
                    r.setHeight(10);
                    // user knows that r it's a rectangle.
            // It assumes that he's able to set the width & height as for the base class

                    System.out.println(r.getArea());
                    // now he's surprised to see that the area is 100 instead of 50.
          }
}
```

TO DO : Proposez une solution (code ou diag classe UML) qui respecte le LSP.

Exercice 4 – Interface Segregation Principle

Ci-après un extrait de code incomplet qui ne respecte pas le ISP.

```
interface IWorker {
          public void work();
          public void eat();
}
class Worker implements IWorker{
          public void work() {
                    // ....working
          }
          public void eat() {
                    // ...... eating in launch break
          }
}
class SuperWorker implements IWorker{
          public void work() {
                    //.... working much more
          }
          public void eat() {
                    //.... eating in launch break
          }
}
class Manager {
          IWorker worker;
          public void setWorker(IWorker w) {
                    worker=w;
          }
          public void manage() {
                    worker.work();
          }
}
```

TO DO : Où est le problème ? Proposez une solution (code ou diag classe UML) qui respecte le ISP.

Exercice 5 – Dependency Inversion Principle

Ci-dessous un exemple où le DIP n'est pas appliqué.

Considérons que la classe *Manager* est une classe de haut niveau et que *Worker* est celle de bas niveau. Nous avons besoin d'ajouter un nouveau module à notre application pour décrire les changements dans la structure de l'entreprise déterminés par l'emploi de nouveaux travailleurs spécialisés (nouvelle classe *SuperWorker*).

Considérons que la classe *Manager* est complexe et qu'elle contient beaucoup de logique métier complexe. Nous devons donc maintenant modifier cette logique métier afin d'introduire le nouveau concept de *SuperWorker*.

Inconvénients :
• il faut modifier la classe *Manager* (comme elle est complexe cela prendra donc beaucoup de temps (donc coût) pour faire les changements) ;
• certaines fonctionnalités de *Manager* peuvent être impactées ;
• la classe de test unitaire associée à *Manager* est à re-écrire également.

```
// Dependency Inversion Principle - Bad example

class Worker {

        public void work() {

                // ....working
        }
}

class Manager {

        Worker worker;

        public void setWorker(Worker w) {
                worker = w;
        }
        public void manage() {
                worker.work();
        }
}
class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```

TO DO :

1. Modélisez par un diagramme de classe la mauvaise version
2. Identifiez la mauvaise application du DIP parmi les relations modélisées
3. Modélisez puis codez une version respectant le DIP en appliquant la technique d'inversion de dépendance.

Exercice 6 – SOLID

Le code complet d'un jeu de dé très simple est disponible sur l'espace-cours.

TO DO :

1. Trouver et justifier en quoi les 5 principes SOLID ne sont pas valides.
2. Codez une version qui respecte les principes SOLID.