Architecture Logicielle

Quelques principes de conceptions

Florent Nicart

Université de Rouen

2017–2018

Les principes S.O.L.I.D.

SOLID est un acronyme désignant un ensemble de cinq principes [1] permettant d'aboutir à la conception d'un système facile à maintenir et à étendre dans le temps :

S	SRP	*Single responsibility principle*
O	OCP	*Open/closed principle*
L	LSP	*Liskov substitution principle*
I	ISP	*Interface segregation principle*
D	DIP	*Dependency inversion principle*

1. Voir `http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`.

S.R.P.

–

Single Responsibility Principle

Single Responsibility Principle

Une responsabilité unique

Définition (Principe de responsabilité simple [2])

Chaque objet doit avoir une unique responsabilité qui doit être entière encapsulée dans sa classe.

Définition (Responsabilité)

Une responsabilité, pour une classe, correspond à une « raison de changer ».

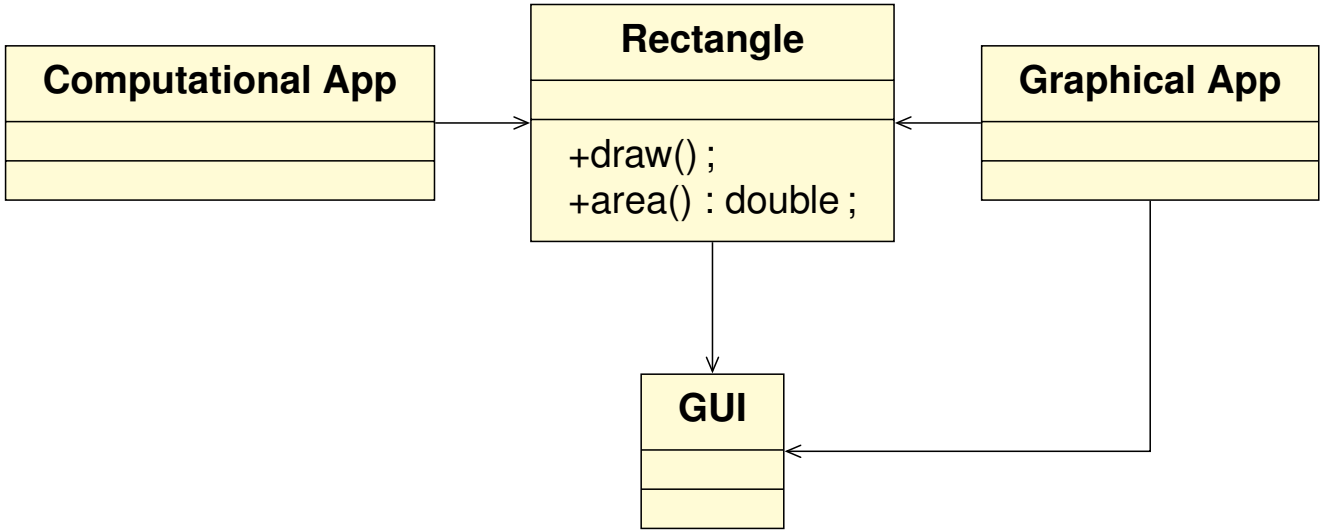
« THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE. »

- Lorsqu'une classe cumule plusieurs responsabilités, celles-ci deviennent couplés.
- La modification d'une responsabilité peut alors compromettre la capacité de la classe à réaliser les autres.

2. Robert C. *Single responsibility principle*

Single Responsibility Principle

Exemple d'une mauvaise conception



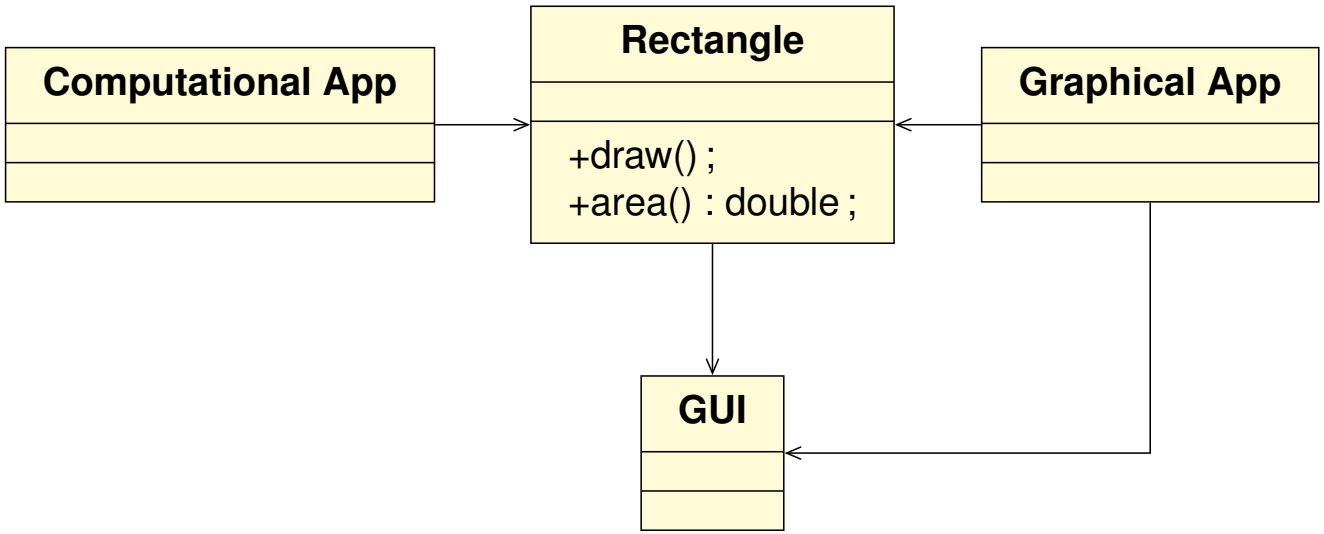
Une classe `Rectangle` est utilisée par deux applications :

- une application de calcul de surface (`Computational App` qui n'utilise que la méthode `area`,
- une application de rendu graphique (`Graphical App`) qui utilise la méthode `draw` pour dessiner dans une GUI [3]

3. Graphical User Interface

Single Responsibility Principle

Exemple d'une mauvaise conception



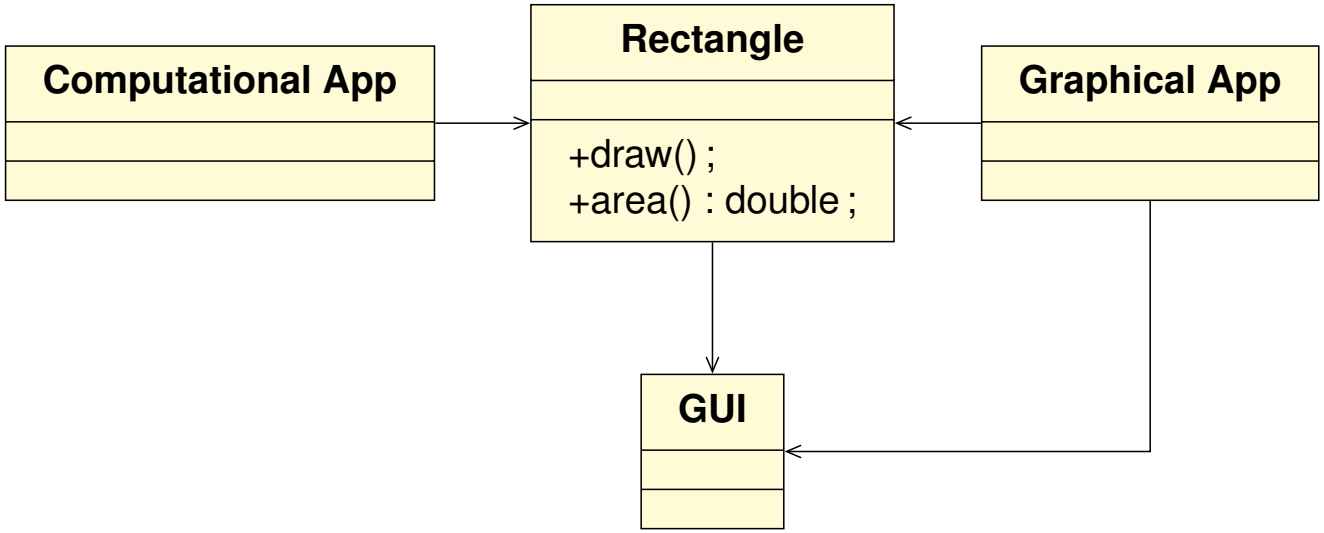
La classe *Rectangle* a deux responsabilités :

1. fournir le modèle mathématique d'un rectangle,
2. effectuer son rendu sur une interface graphique.

La classe *Rectangle* viole le SRP.

Single Responsibility Principle

Exemple d'une mauvaise conception



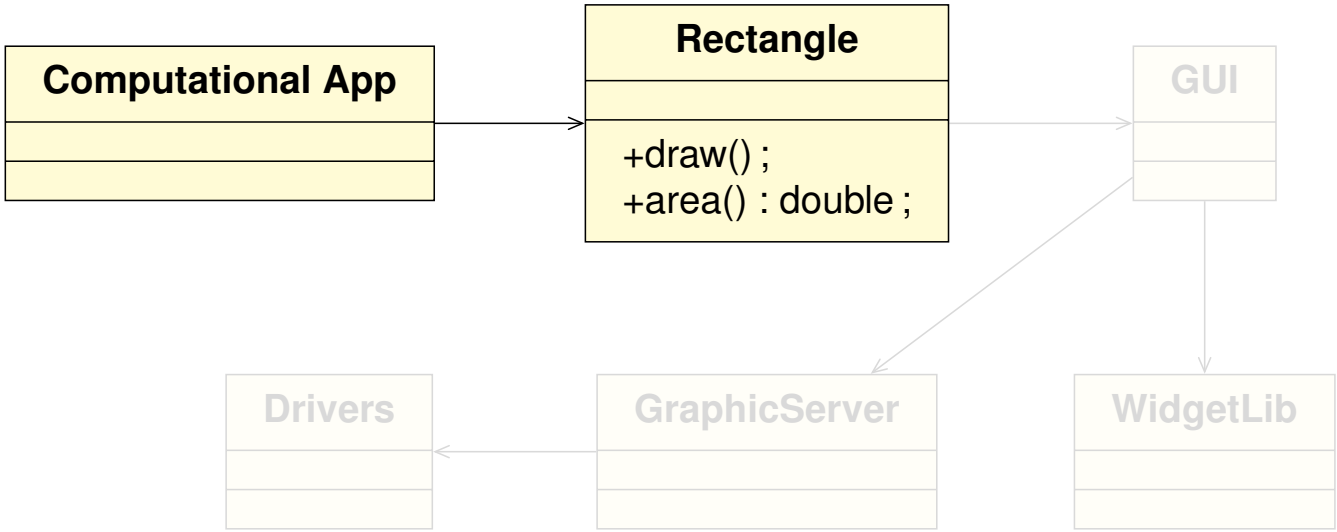
Conséquences du viol du SRP :

- une application de calcul géométrique est couplée à une interface graphique (dépendances inutiles).
- Un changement de GUI peut nous forcer à recompiler/tester/déployer l'application de calcul.

Single Responsibility Principle

Conséquences d'une mauvaise conception

Du point de vue de l'application de calcul :

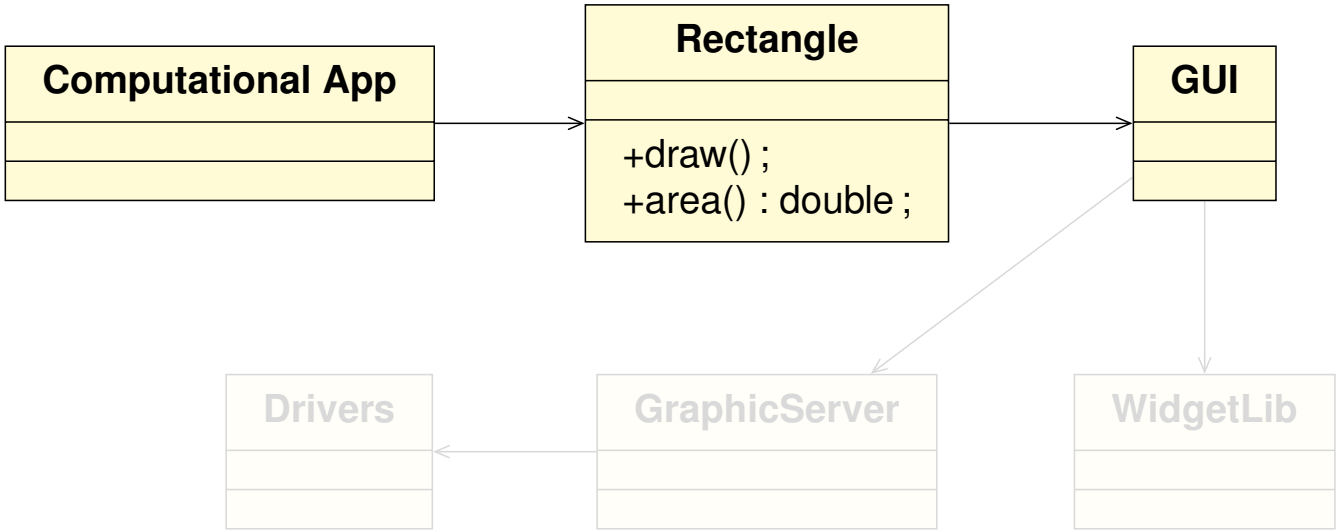


- Il existe des machines sans interface graphique (les serveurs),
- sur lesquels on aurait bien aimé pouvoir exécuter l'application de calcul.

Single Responsibility Principle

Conséquences d'une mauvaise conception

Du point de vue de l'application de calcul :

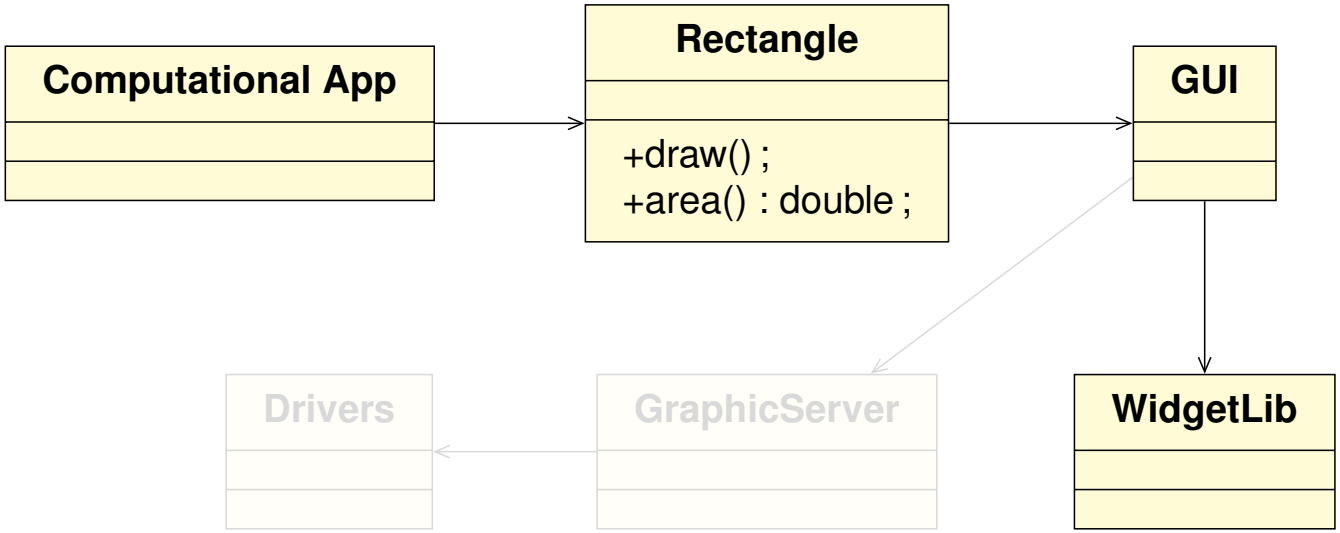


- Il existe des machines sans interface graphique (les serveurs),
- sur lesquels on aurait bien aimé pouvoir exécuter l'application de calcul.

Single Responsibility Principle

Conséquences d'une mauvaise conception

Du point de vue de l'application de calcul :

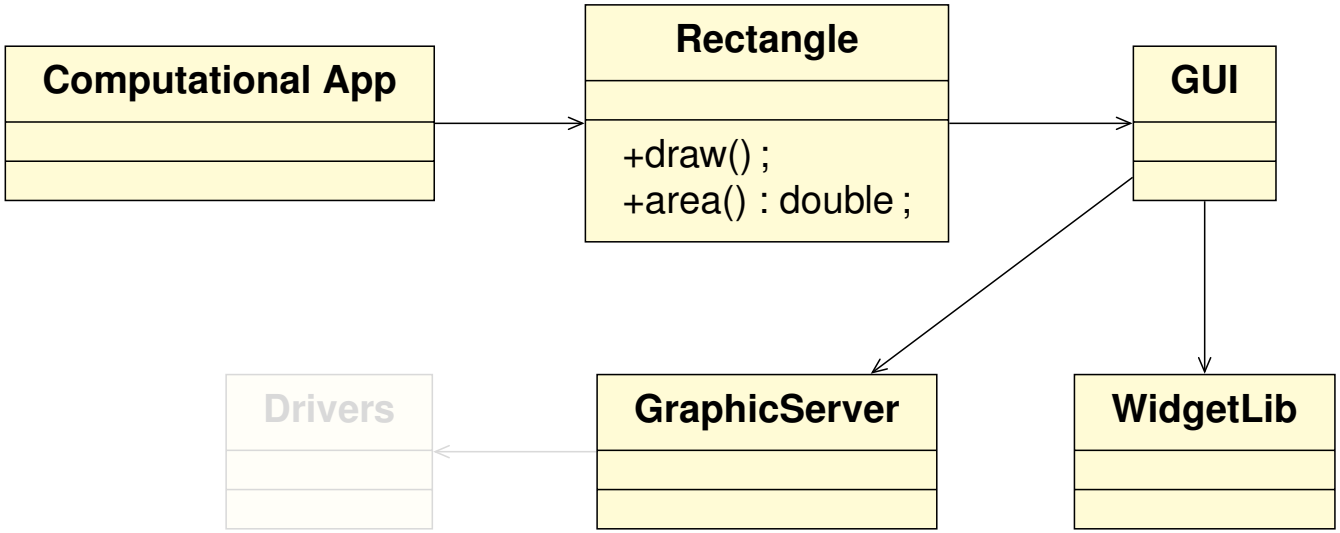


- Il existe des machines sans interface graphique (les serveurs),
- sur lesquels on aurait bien aimé pouvoir exécuter l'application de calcul.

Single Responsibility Principle

Conséquences d'une mauvaise conception

Du point de vue de l'application de calcul :

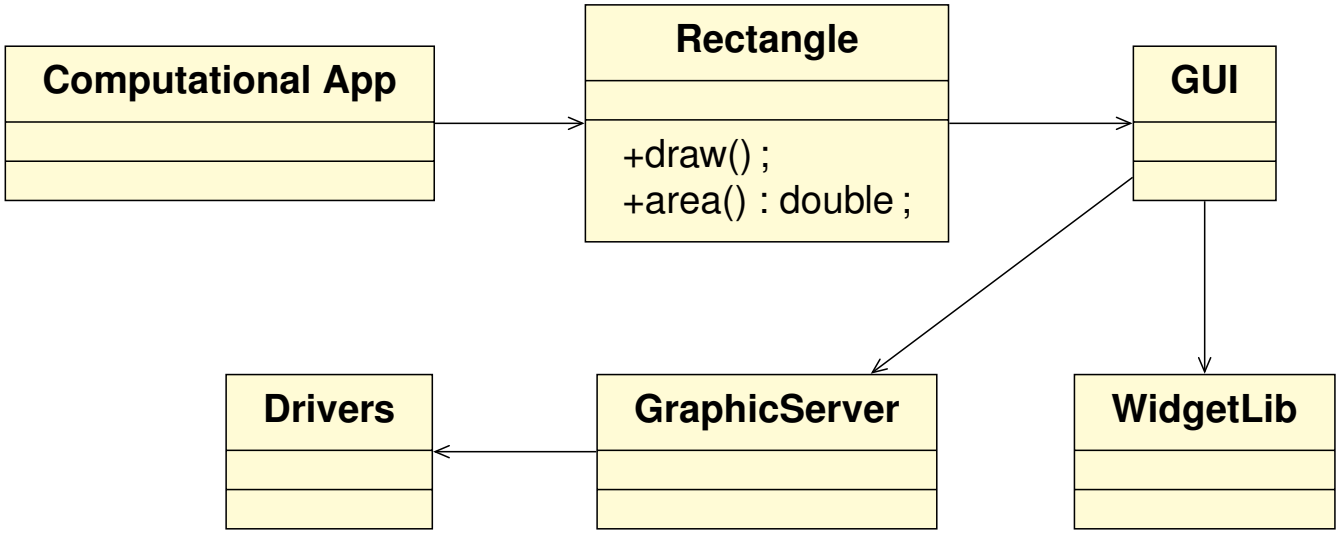


- Il existe des machines sans interface graphique (les serveurs),
- sur lesquels on aurait bien aimé pouvoir exécuter l'application de calcul.

Single Responsibility Principle

Conséquences d'une mauvaise conception

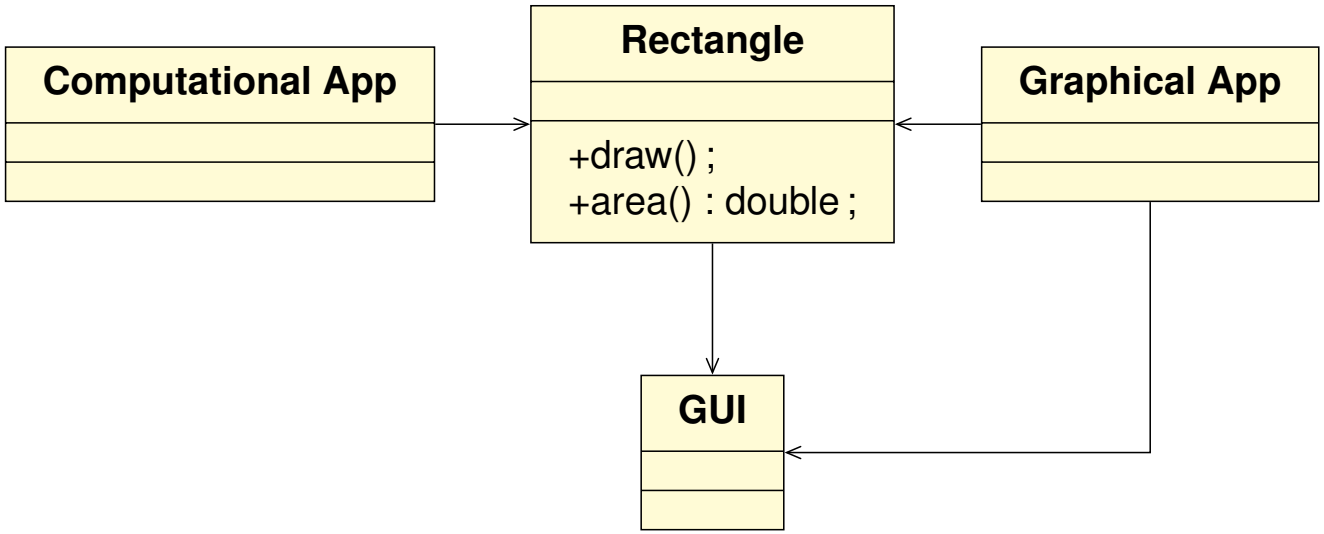
Du point de vue de l'application de calcul :



- Il existe des machines sans interface graphique (les serveurs),
- sur lesquels on aurait bien aimé pouvoir exécuter l'application de calcul.

Single Responsibility Principle

Exemple d'une mauvaise conception–Conséquences

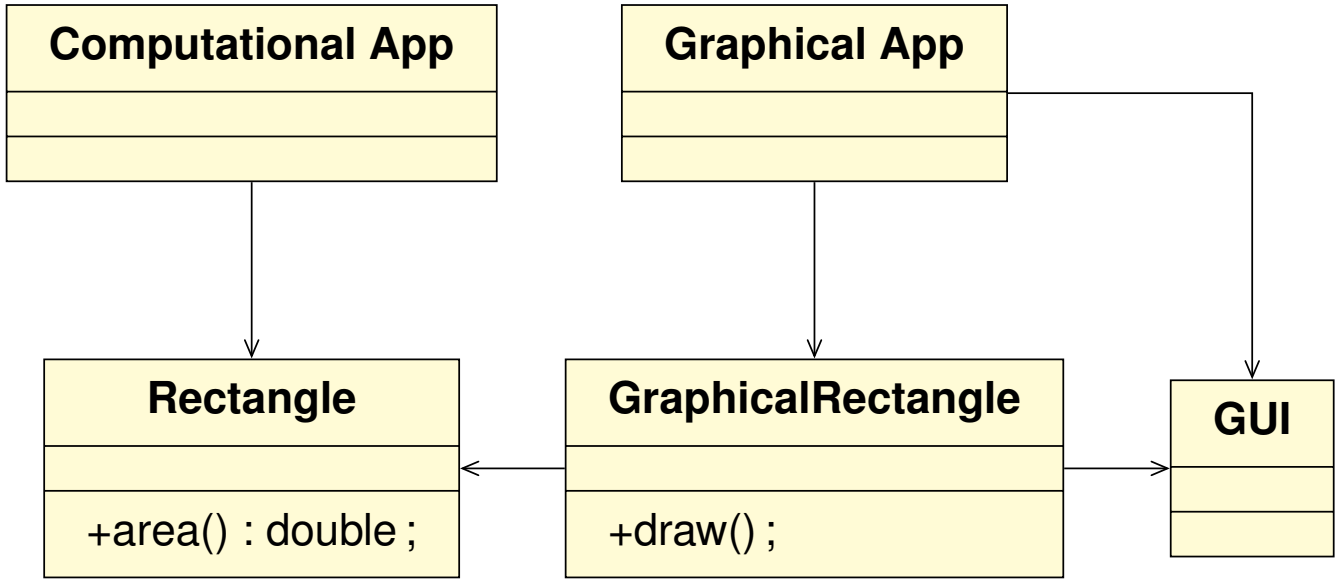


Conséquences :

- une application de calcul géométrique est couplée à une interface graphique.
- Un changement de code dans la GUI peut nous forcer à recompiler/tester/déployer l'application de calcul.

Single Responsibility Principle

Exemple - bonne conception

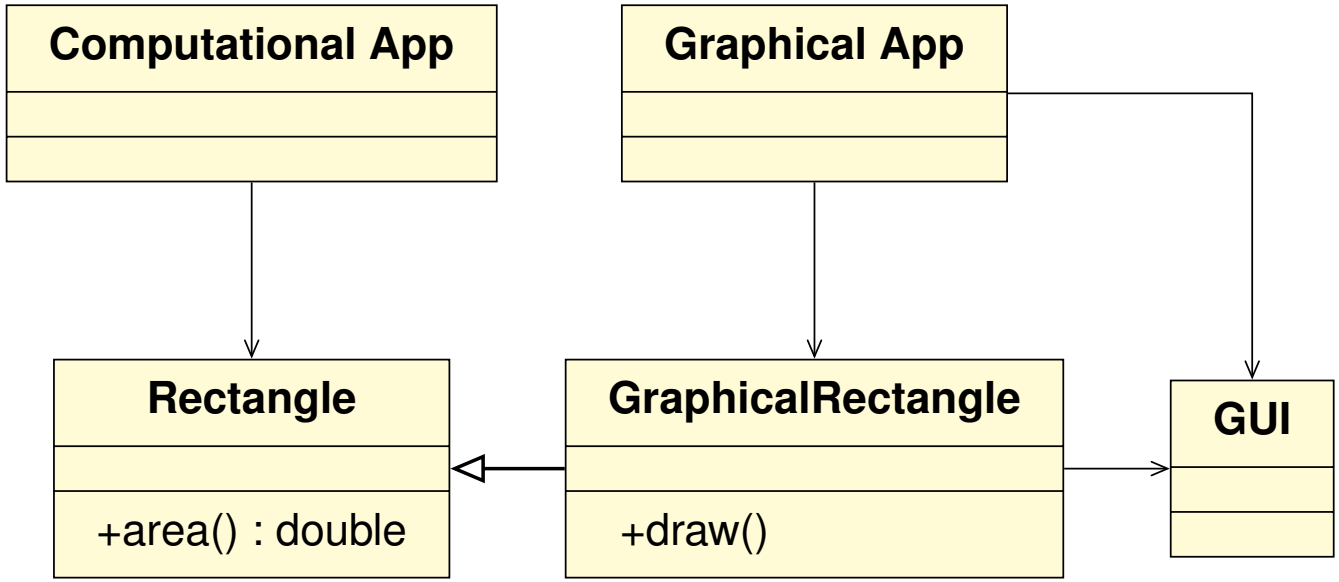


- Les deux responsabilités sont placées dans des classes séparées.
- Maintenant, un changement dans le rendu graphique n'affecte plus l'application de calcul.

Single Responsibility Principle

Exemple - bonne conception ?

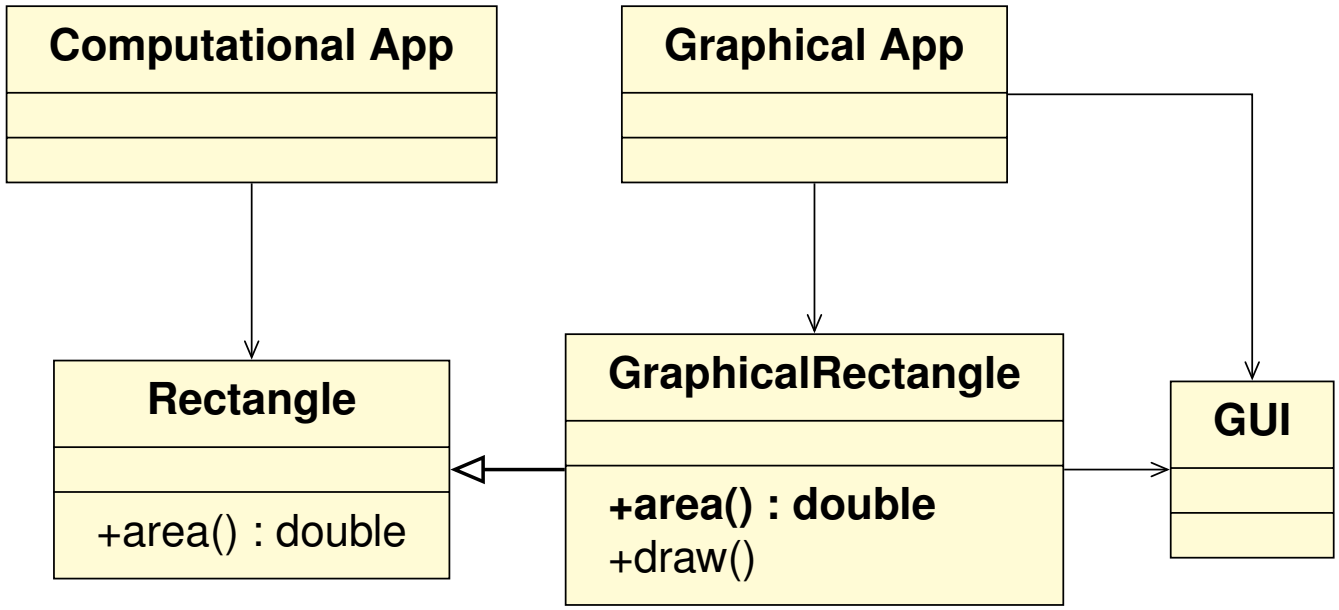
Pourrait-on envisager ceci ?



Single Responsibility Principle

Exemple - bonne conception ?

Pourrait-on envisager ceci ?

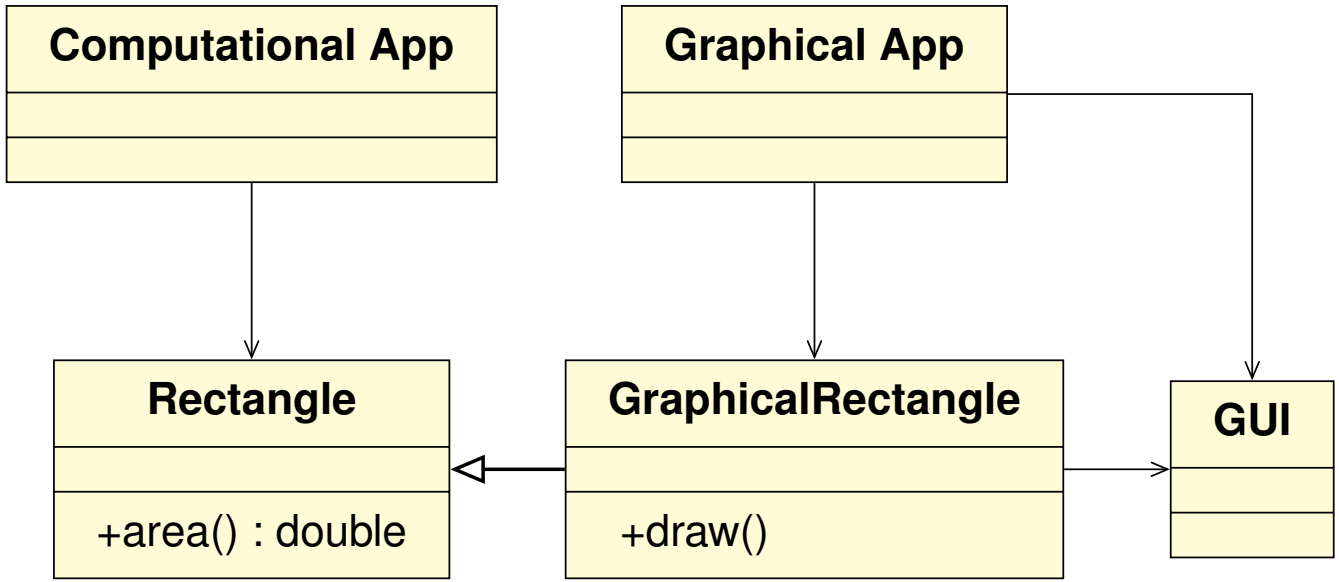


→ OK si `GraphicalRectangle` est un `Rectangle` dans `Graphical App`.

Single Responsibility Principle

Exemple - bonne conception ?

Pourrait-on envisager ceci ?



Une « responsabilité unique » peut se traduire par :

« Le code d'un module (une classe) doit correspondre à une seule responsabilité. »

Single Responsibility Principle

Identifier les responsabilités

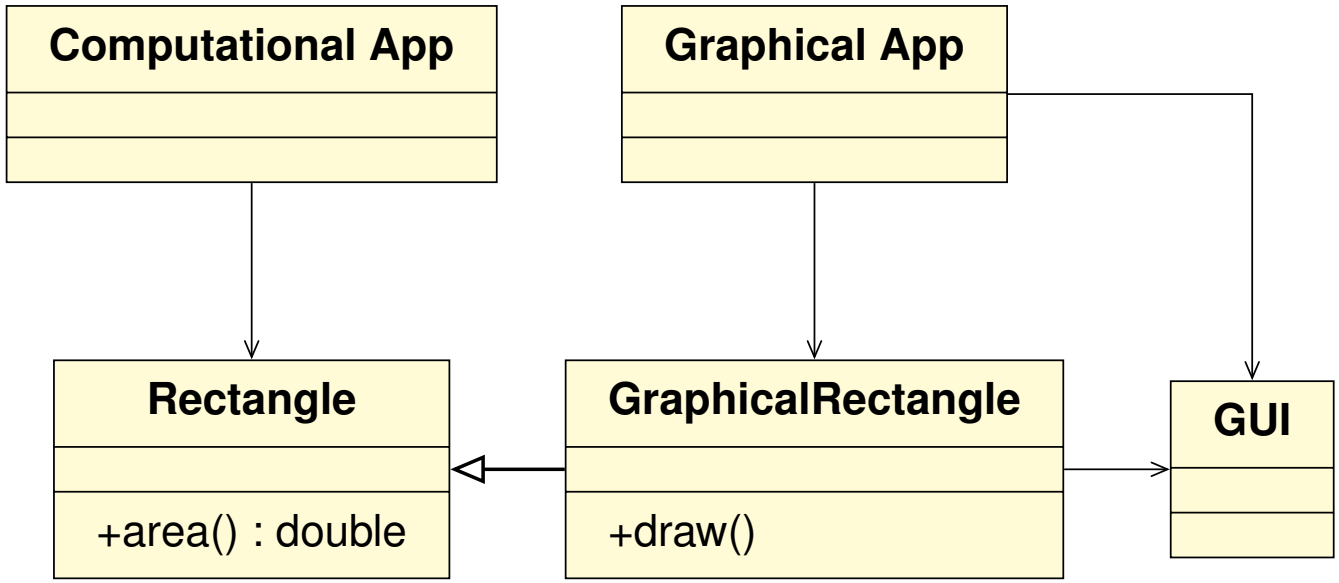
À quel moment identifier les responsabilités ?

Single Responsibility Principle

Identifier les responsabilités

À quel moment identifier les responsabilités ?

- Trivial à partir de la vue globale :

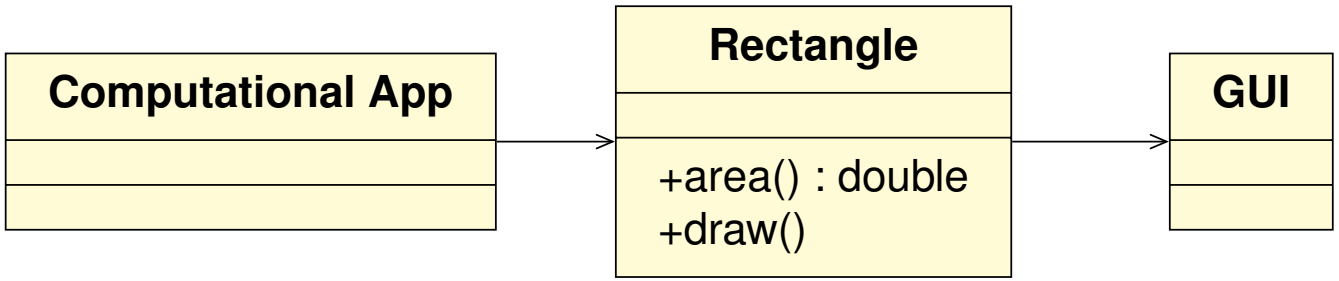


Single Responsibility Principle

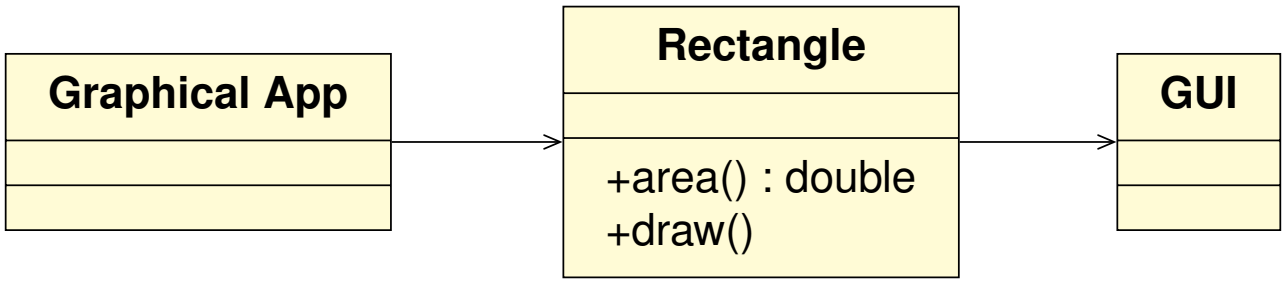
Identifier les responsabilités

À quel moment identifier les responsabilités ?

- Au moment de la conception de `Computational App` :



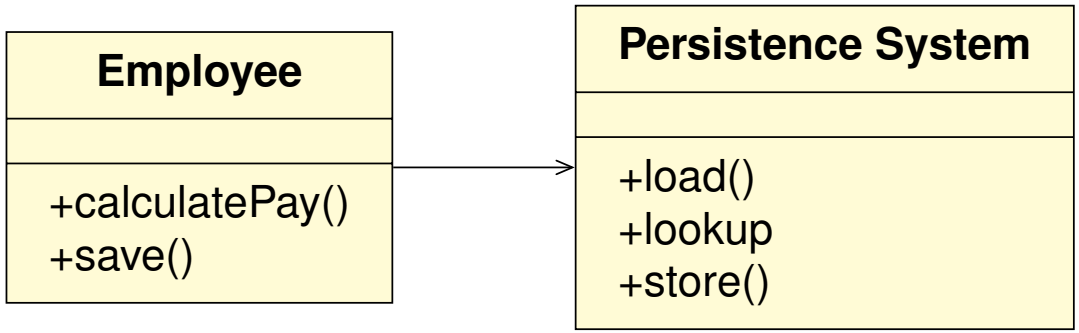
- Au moment de la conception de `Graphical App` :



Single Responsibility Principle

SRP et interfaces

La délégation garantit-elle systématiquement le SRP ?

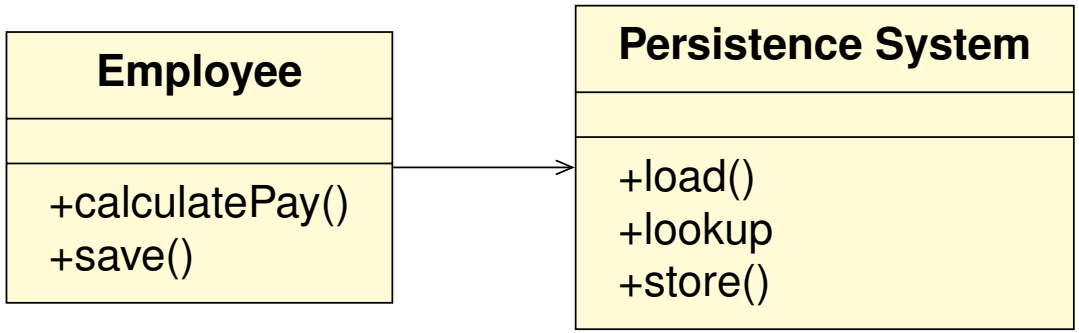


- `Employee` délègue la persistance à d'autres classes,
- mais son interface publique exhibe deux responsabilités : la manipulation d'un employé (métier) et son stockage (méthode `save()`).
- Le SRP s'applique aussi aux interfaces (voir ISP). Nous verrons des patrons pour externaliser des fonctionnalités.

Single Responsibility Principle

SRP et interfaces

La délégation garantit-elle systématiquement le SRP ?

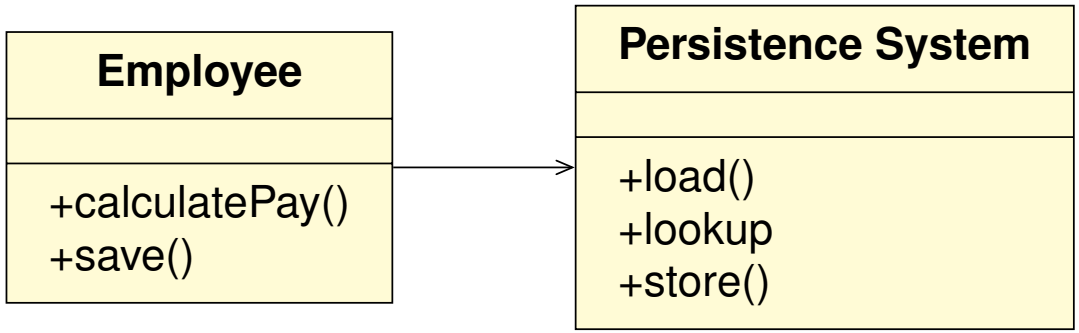


- `Employee` délègue la persistance à d'autres classes,
- mais son interface publique exhibe deux responsabilités : la manipulation d'un employé (métier) et son stockage (méthode `save()`).
- Le SRP s'applique aussi aux interfaces (voir ISP). Nous verrons des patrons pour externaliser des fonctionnalités.

Single Responsibility Principle

SRP et interfaces

La délégation garantit-elle systématiquement le SRP ?



- `Employee` délègue la persistance à d'autres classes,
- mais son interface publique exhibe deux responsabilités : la manipulation d'un employé (métier) et son stockage (méthode `save()`).
- Le SRP s'applique aussi aux interfaces (voir ISP). Nous verrons des patrons pour externaliser des fonctionnalités.

Single Responsibility Principle

En résumé

- Le *Single Responsability Principle* consiste à encapsuler individuellement les responsabilités,
- Séparer les responsabilités, au moins au niveau des interfaces, permet de découpler les clients.
- Le SRP est un des principes les plus simples mais paradoxalement un des plus difficiles à mettre en oeuvre.
- Il peut conduire à une modularisation extrême et à un nombre (trop) élevé de classes.
- Coût du respect d'un principe – principe de Pareto ou principe des 80-20 :

En gestion de projet : 80 % de la réalisation nécessite 20 % de l'effort.

O.C.P.

–

Open/Close Principle

Open/Close Principle

Ivar Jacobson [3]

« All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version. »

Comment créer des composants stables face à l'évolution du système ? Réponse :

Définition (Open/Close Principle (Bertrand Meyer [4]))

« Software entities (classes, modules, functions, etc.) should be **open for extension**, *but* **closed for modification**. »

3. Informaticien suédois, un des concepteur d'UML.
4. Informaticien français, créateur du langage objet Eiffel.

Open/Close Principle

Ouvert pour l'extension

Le comportement du module peut être étendu pour satisfaire de nouveaux besoins.

Fermé pour la modification

Le contrat existant du module ne doit pas changer.

- Une fois terminée, l'implémentation d'une classe ne doit être modifiée que pour corriger des erreurs.
- L'ajout de nouvelles fonctionnalités doit préférablement donner lieu à une nouvelle classe.
- La nouvelle classe peut réutiliser le code existant (héritage ou composition) en toute sécurité, puisque celui ci est stable.

Open/Close Principle

Attention !

- Respecter l'OCP ne se limite pas à s'interdire de modifier du code !
- L'OCP est l'un des principes que le enfreint dans le futur à cause d'une conception dans le présent !

Open/Close Principle

Exemple

- Considérons la fonction *totalPrice* :
- Cette fonction totalise le prix de chaque élément de type Part figurant dans le tableau parts.

```
1  public double totalPrice(Part[] parts) {
2     double total = 0.0;
3     for (int i=0; i<parts.length; i++) {
4        total += parts[i].getPrice();
5     }
6     return total;
7  }
```

- Si Part est une classe de base ou une interface et si le polymorphisme est utilisé cette fonction peut traiter de nouveaux types dérivant de ou réalisant Part.
- Cette fonction vérifie *OCP*.

Open/Close Principle

Exemple : évolution

Évolution du système

Le département financier décide d'appliquer différents coefficients correcteurs aux prix des produits.

- Une **très mauvaise idée** est de réécrire le code :

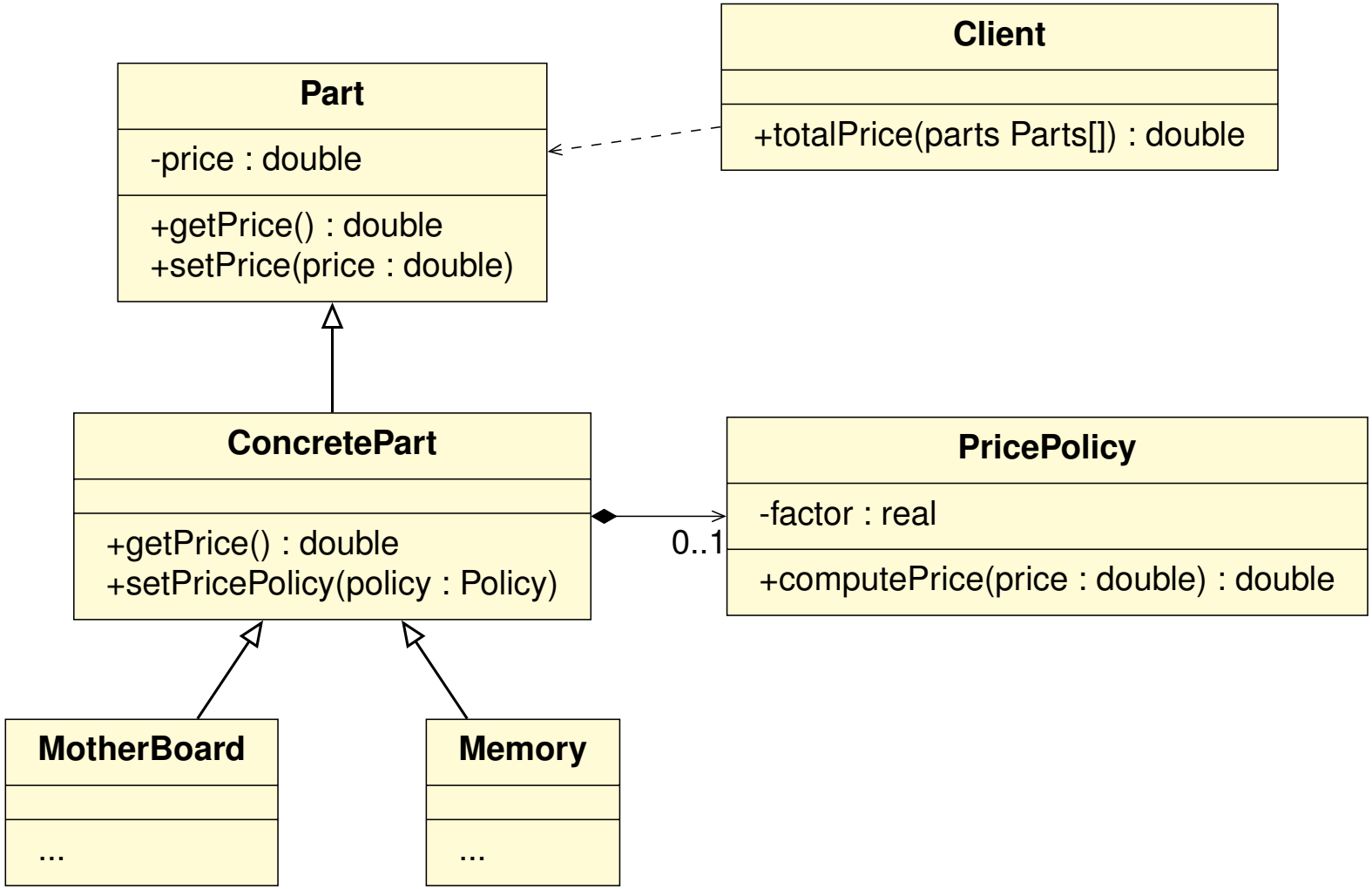
```
1  public double totalPrice(Part[] parts) {
2     double total = 0.0;
3
4     for (int i=0; i<parts.length; i++) {
5        if (parts[i] instanceof Motherboard)
6           total += (1.45 * parts[i].getPrice());
7        else if (parts[i] instanceof Memory)
8           total += (1.27 * parts[i].getPrice());
9        else
10          total += parts[i].getPrice();
11    }
12    return total;
13 }
```

- L'ajout d'un produit nécessite la modification de *totalPrice*.

Open/Close Principle

Exemple : solution

XXX



Open/Close Principle

Exemple : l'existant

- Ancienne réalisation (code existant) :

```
1  package ocp;
2  public class Part {
3      private double price;
4      public Part(double price) {
5          this.price = price;
6      }
7      public void setPrice(double price) {
8          this.price = price;
9      }
10     public double getPrice() {
11         return price;
12     }
13 }
```

Open/Close Principle

Exemple : l'évolution

- Nouvelle réalisation (code ajouté) :

```
1  package ocp;
2  public class PricePolicy {
3      private float factor;
4      public PricePolicy(float factor) {
5          this.factor = factor;
6      }
7      public double computePrice(double price) {
8          return price * factor;
9      }
10 }
```

```
1  package ocp;
2  public class ConcretePart extends Part {
3      private PricePolicy pricePolicy;
4      public ConcretePart(double price) {
5          super(price);
6      }
7      public void setPricePolicy(PricePolicy pricePolicy) {
8          this.pricePolicy = pricePolicy;
9      }
10     public double getPrice() {
11         if (pricePolicy==null) return super.getPrice();
12         return pricePolicy.computePrice(super.getPrice());
13     }
14 }
```

Open/Close Principle

Exemple : l'existant

- Avec cette solution la politique de prix peut être gérée dynamiquement.
- Il est également possible de prédéfinir des politiques caractéristiques par introduction de constantes statiques dans la classe PricePolicy.
- Bien entendu dans une application, prix et facteurs promotionnels seraient conservés dans une base de données.

Open/Close Principle

En résumé

- *Open/Closed Principle* est au coeur de la conception orientée objet.
- En pratique, il est impossible que tous les modules d'un système logiciel respectent ce principe à 100%.
- De plus, on peut toujours trouver des aspects par rapport auxquels le module n'est pas fermé.
- Respecter ce principe permet d'atteindre un haut niveau de robustesse et de réutilisation.
- En résumé : on fait du « **développement durable** ».

L.S.P.

–

Liskov Substitution Principle

Liskov Substitution Principle

Définition (Liskov Substitution Principle (Barbara Liskov [5])

« Functions that use pointers or references to classes must be able to use objects of derived classes without knowing it. »

- Exemple de violation du principe :

```
1  void DrawShape(Shape s) {
2      if (s instanceOf Square)
3          DrawSquare((Square)s);
4      else if (s instanceOf Circle)
5          DrawCircle((Circle)s);
6  }
```

5. Informaticienne américaine, créatrice du premier langage objet CLU.

Liskov Substitution Principle

- Le principe de substitution de Liskov (LSP) est clairement associé au polymorphisme pur.
- La méthode :

```
1 public void drawShape(Shape s) {
2     // Code.
3 }
```

- doit supporter chaque classe dérivée existante ou à venir de la superclasse Shape ou réalisant l'interface Shape.

Liskov Substitution Principle

- Lorsqu'une fonction ne satisfait pas LSP, ce peut être parce qu'elle utilise des références explicites sur des classes dérivées du type de la référence manipulée.
- Une telle fonction viole également OCP, puisque son code doit être modifié lorsqu'une sous classe est crée.
- Mais il existe des cas moins triviaux.

Liskov Substitution Principle

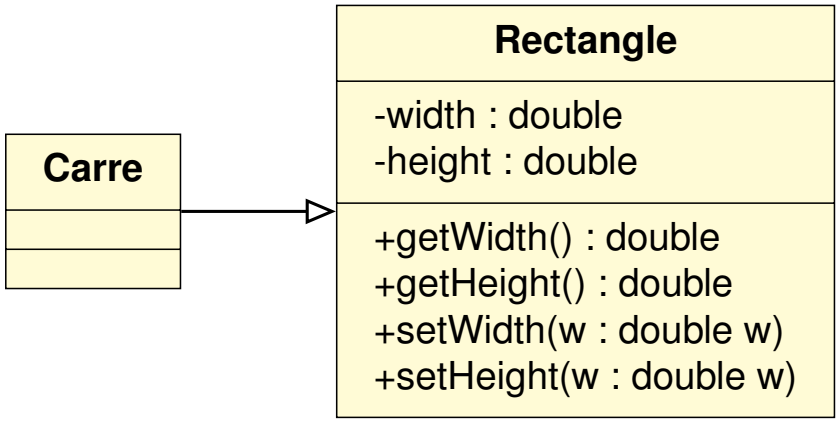
Exemple

- Soit le module :

```
1  package geo;
2
3  public class Rectangle {
4      private double width;
5      private double height;
6
7      public Rectangle(double w, double h) {width=w; height=h;}
8
9      public double getWidth() { return width; }
10     public double getHeight() { return height; }
11
12     public void setWidth(double w) { width = w; }
13     public void setHeight(double h) {height = h; }
14
15     public double area() {return (width * height);}
16 }
```

Liskov Substitution Principle

Exemple



- Mathématiquement un carré est un rectangle, donc la classe Carré doit hériter de la classe Rectangle. Voyons voir !
- Un carré n'utilise pas largeur et hauteur donc chaque carré perd un peu de mémoire, mais ceci n'est pas le plus important ici.

Liskov Substitution Principle

Exemple

- Par contre les méthodes `setWidth()` et `setHeight()` ne sont pas vraiment appropriées pour un Carré.
- Nous ne pouvons annuler ces deux méthodes, donc nous les redéfinissons !
- Définition de la classe `Carre` à partir de la classe `Rectangle` :

```
1  package geo;
2  public class Carre extends Rectangle {
3      public Carre(double s) {
4          super(s, s);
5      }
6      public void setWidth(double w) {
7          super.setWidth(w);
8          super.setHeight(w);
9      }
10     public void setHeight(double h) {
11         super.setHeight(h);
12         super.setWidth(h);
13     }
14 }
```

Liskov Substitution Principle

Exemple

- Que produit le programme de test suivant ?

```
1  package geo;
2  public class Main {
3     // Define a method that takes a Rectangle reference.
4     public static void testLSP(Rectangle r) {
5        r.setWidth(4.0); r.setHeight(5.0);
6        System.out.println("Width is 4.0 and Height is 5.0" +
7           ", so Area is "+ r.area());
8        if (r.area() == 20.0) System.out.println("Looking good!\n");
9        else
10          System.out.println("Huh?? What kind of rectangle is this??\n");
11    }
12
13    public static void main(String[] args) {
14       // Create a Rectangle and a Square
15       Rectangle r = new Rectangle(1.0, 1.0);
16       Carre s = new Carre(1.0);
17       testLSP(r); testLSP(s);
18    }
19 }
```

Liskov Substitution Principle

Exemple

- Résultat du test :

   *Width is 4.0 and Height is 5.0, so Area is 20.0*
  Looking good !

   *Width is 4.0 and Height is 5.0, so Area is 25.0*
  Huh ?? What kind of rectangle is this ??
- Explication : le développeur pense avec raison que la largeur et la hauteur d'un rectangle sont indépendantes.
- Décidemment un Carré dynamique ne peut hériter d'un Rectangle dynamique. Ceci n'est pas en contradiction avec les notions mathématiques où les objets sont statiques.

Liskov Substitution Principle

Sous-typage de Liskov

Cet exemple parfait pour rappeler la différence entre le sous-typage des TDA et celui de nos classes

- Lorsque nous dérivons une classe ou implémentons une interface, nous créons un sous-type logiciel.
- Le compilateur contrôle cette relation et nous garantit une conformité du sous-type seulement sur la liaison du code.
- Le sous-typage de Liskov s'applique aux TDA [6] et va plus loin que la compatibilité « technique » : un sous-type doit avoir un comportement compatible avec celui attendu de son super-type (contrat).
- Seul le développeur peut garantir ce dernier point.

6. Types de données abstrait

Liskov Substitution Principle

En résumé

- *Liskov Substitution Principle* peut être vu comme un principe de conception par contrat (la super classe ou l'interface).
- Il repose sur le polymorphisme : toute classe d'une arborescence d'héritage (resp. implémentant une interface donnée), doit pouvoir être manipulée de la même manière que la super-classe (resp. interface).
- Toutes les classes dérivant d'une même classe ou d'une même interface doivent être **substituables** sans aucune connaissance.
- Elles doivent respecter le contrat défini par leur ancêtre.
- LSP contribue à respecter OCP.

I.S.P.

–

Interface Segregation Principle

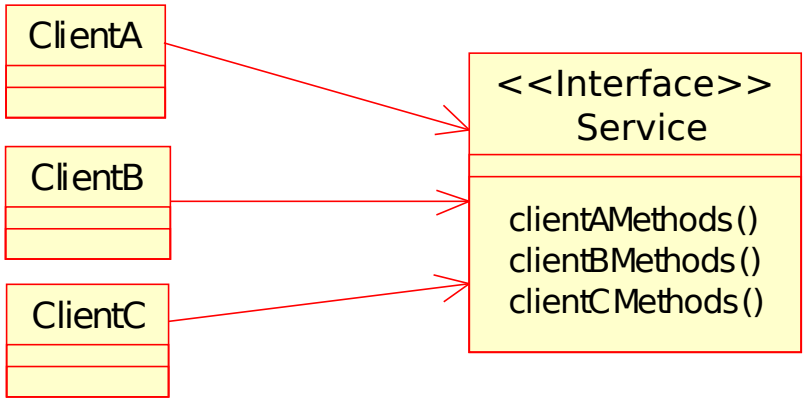
Interface Segregation Principle

- ISP adresse le problème de l'obésité chez les interfaces.
- Principe formulé par Robert C. Martin lorsqu'il était consultant chez Xerox.

Petite histoire : Xerox avait créé une nouvelle imprimante avec de nombreuses fonctionnalités. Le logiciel fonctionnait parfaitement mais était devenu au fil du temps impossible à maintenir ou à étendre. Le problème : une unique classe *Job* effectuait presque toutes les tâches.

Interface Segregation Principle

Obésité/Pollution d'interface

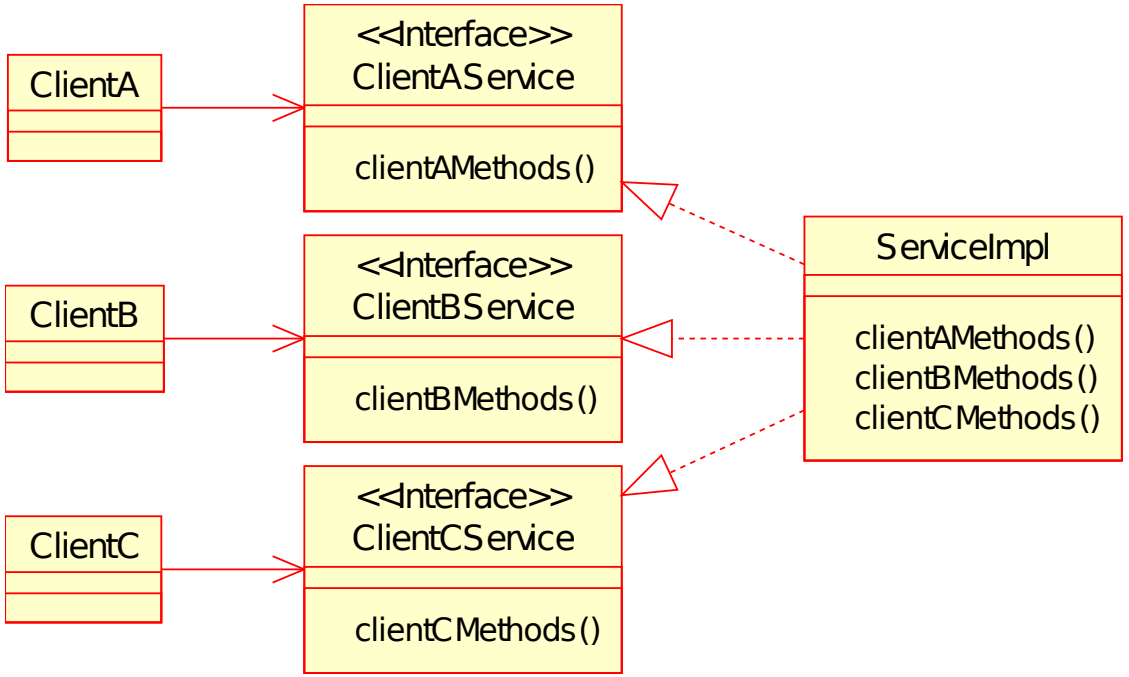


L'amalgame d'interfaces provoque :

- la pollution des interfaces. Ex : une tâche d'aggrafage a accès aux fonctions d'impressions (et inversement) ;
- le couplage des clients : une modification de l'une d'elle provoque la recompilation/test/déploiement des clients qui ne sont pas concernés.

Interface Segregation Principle

Obésité/Pollution d'interface



Définition (Interface Segregation Principle)

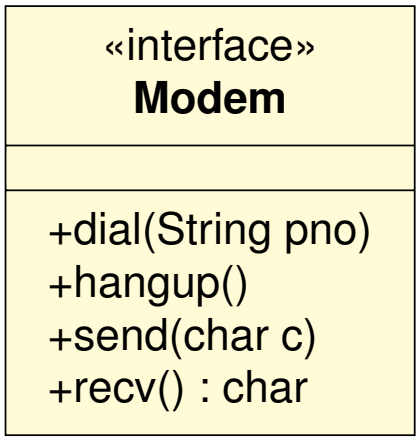
« Clients should not be forced to depend upon interfaces that they do not use. »

Plusieurs clients implique plusieurs interfaces !

Interface Segregation Principle

Exemple 1

- Soit l'interface représentant les opérations applicables à un modem :



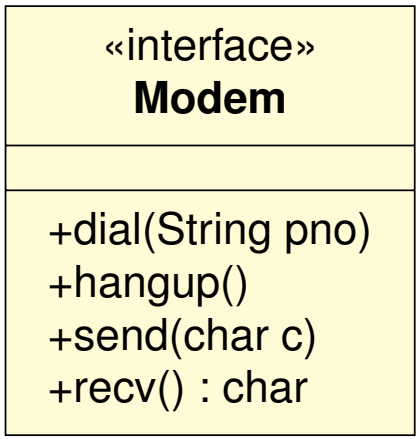

```
1  interface Modem {
2      public void dial(String pno);
3      public void hangup();
4      public void send(char c);
5      public char recv();
6  }
```

- La gestion de la connexion et le transfert des données sont deux responsabilités qui peuvent être séparées.

Interface Segregation Principle

Exemple 1

- Soit l'interface représentant les opérations applicables à un modem :



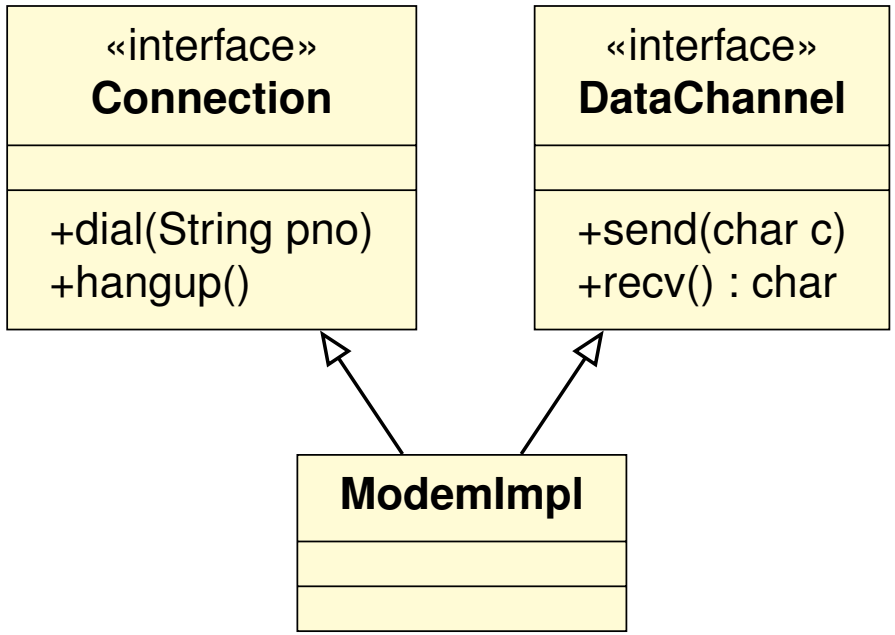

```
1  interface Modem {
2      public void dial(String pno);
3      public void hangup();
4      public void send(char c);
5      public char recv();
6  }
```

- La gestion de la connexion et le transfert des données sont deux responsabilités qui peuvent être séparées.

Interface Segregation Principle

Exemple 1

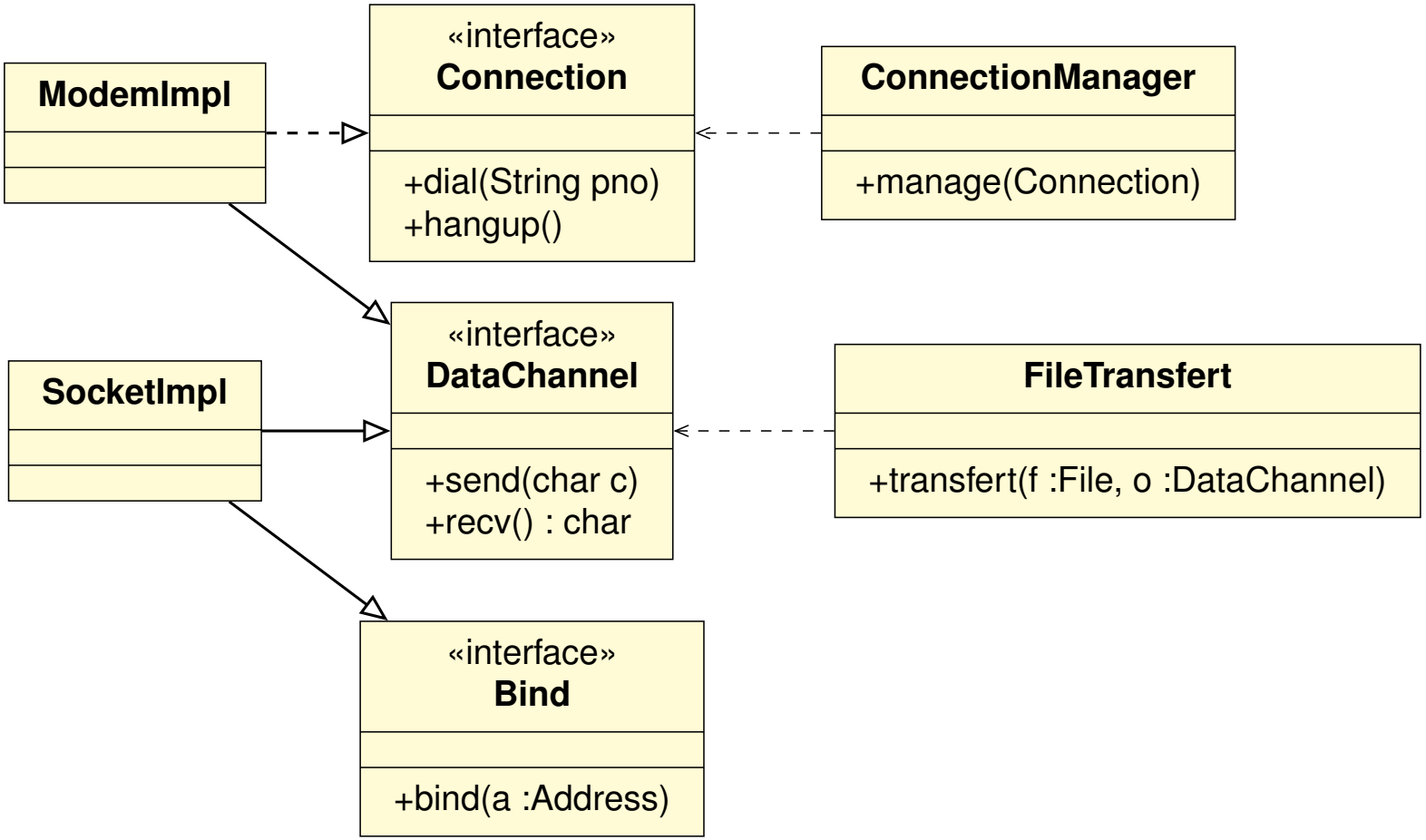
- Même si au final une seule classe fournira les deux implémentations, les responsabilités sont découplées :



Interface Segregation Principle

Exemple 1 - bénéfices

- Les clients sont également découplés :



Interface Segregation Principle

En résumé

- *Interface Segregation Principle* : des clients distincts impliquent des interfaces distinctes.
- Le respect de ce principe apporte :
  - Une meilleure cohésion : le système est plus compréhensible et plus robuste.
  - Un couplage plus faible : maintenance accrue et meilleure résistance au changement.
- Application de SRP à la conception d'interfaces

D.I.P.

–

Dependency Inversion principle

Dependency Inversion principle

Définition

1. *« HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS. »*
2. *« ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS. »*

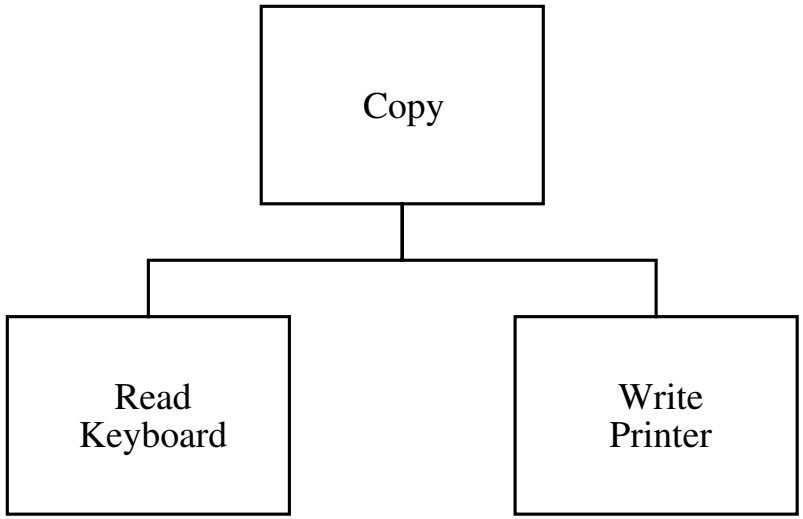
Dependency Inversion principle

Exemple : mauvaise approche

```
1  void Copy() {
2      int c;
3      while ((c = ReadKeyboard()) != EOF)
4          WritePrinter(c);
```

Mais si nous voulons copier vers le disque :

```
1  enum OutputDevice {printer, disk};
2  void Copy(outputDevice dev) {
3      int c;
4      while ((c = ReadKeyboard()) != EOF)
5          if (dev == printer)
6              WritePrinter(c);
7          else
8              WriteDisk(c);
9  }
```

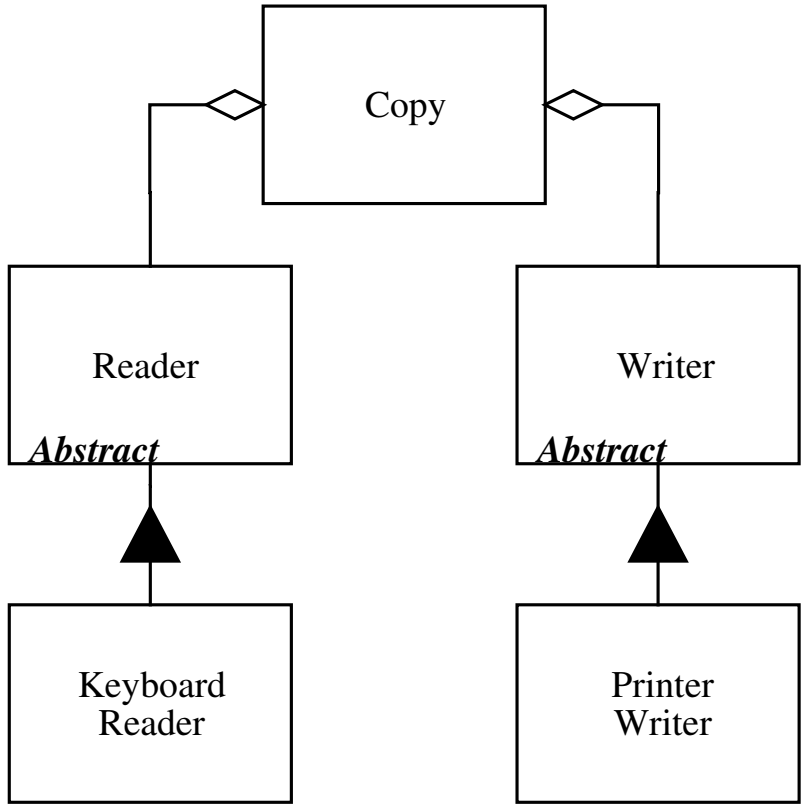


- Ici, le module de haut niveau dépend des modules de bas niveau,
- le principe DIP est violé.

Dependency Inversion principle

Exemple : une meilleure approche

```
1  class Reader {
2  public:
3      virtual int Read() = 0;
4  };
5
6  class Writer {
7  public:
8      virtual void Write(char) = 0;
9  };
10
11 void Copy(Reader& r, Writer& w) {
12     int c;
13     while((c=r.Read()) != EOF)
14         w.Write(c);
15 }
```



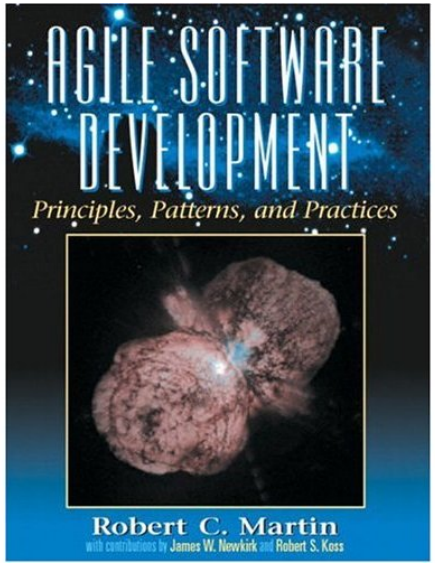
- Ici, c'est le module de haut niveau qui impose un contrat aux modules de bas niveau à travers les interfaces *Reader* et *Writer*.

Dependency Inversion principle

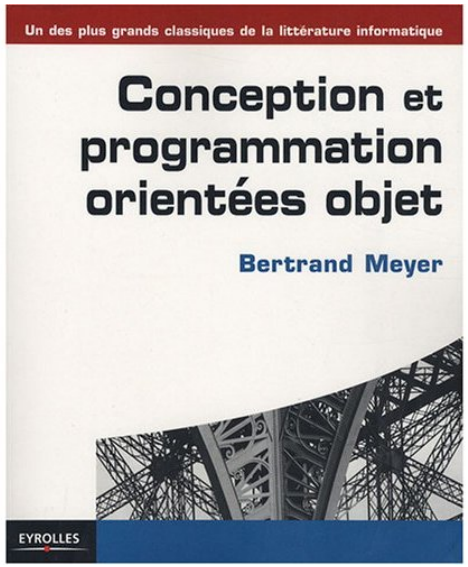
En résumé

- *Dependency Inversion Principle* : les modules de haut niveau ne doivent pas dépendre de modules de bas niveau.
- Le respect de ce principe apporte :
  - une conception en couche,
  - des composants de haut niveau génériques/réutilisables.
- On parle d'**inversion de dépendances** car les interfaces sont imposées par la couche de haut niveau au couches inférieures.

Quelques références



Agile Software Development : Principles, Patterns, and Practices., *Robert C. Martin*, Prentice Hall (2002).
ISBN-13 : 978-0135974445.



Conception et programmation orientées objets, *Bertand Meyer*, Eyrolles (3 janvier 2008).
ISBN-13 : 978-2212122701.