Architecture Logicielle

Quelques principes de conceptions

Florent Nicart

Université de Rouen

2016–2017

Abstraire

Tony Hoare [1]

« Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences. »

Gary Booch [2]

« An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer. »

---

1. Informaticien anglais créateur de Quicksort, la logique de Hoare, le langage formel Communicating Sequential Processes (CSP), etc.
2. Créateur de la méthode d'analyse et de conception orientée objet portant son nom.

Abstraire

Le *schmilblick*

Mon objet possède :

- un coffre,
- une têtière,
- une ou plusieurs gâches,
- une pêne,
- un cylindre,
- une poignée...à quoi pensez vous ?

Abstraire

Le *schmilblick*

Mon objet possède :

- un coffre,
- une têtière,
- une ou plusieurs gâches,
- une pêne,
- un cylindre,
- une poignée...à quoi pensez vous ?

Abstraire

Le *schmilblick*

Mon objet possède :

- un coffre,
- une têtière,
- une ou plusieurs gâches,
- une pêne,
- un cylindre,
- une poignée...à quoi pensez vous ?

Abstraire

Le *schmilblick*

Mon objet possède :

- un coffre,
- une têtière,
- une ou plusieurs gâches,
- une pêne,
- un cylindre,
- une poignée...à quoi pensez vous ?

Abstraire

Le *schmilblick*

Mon objet possède :

- un coffre,
- une têtière,
- une ou plusieurs gâches,
- une pêne,
- un cylindre,
- une poignée...à quoi pensez vous ?

Abstraire

Le *schmilblick*

Mon objet possède :

- un coffre,
- une têtière,
- une ou plusieurs gâches,
- une pêne,
- un cylindre,
- une poignée...à quoi pensez vous ?

Abstraire

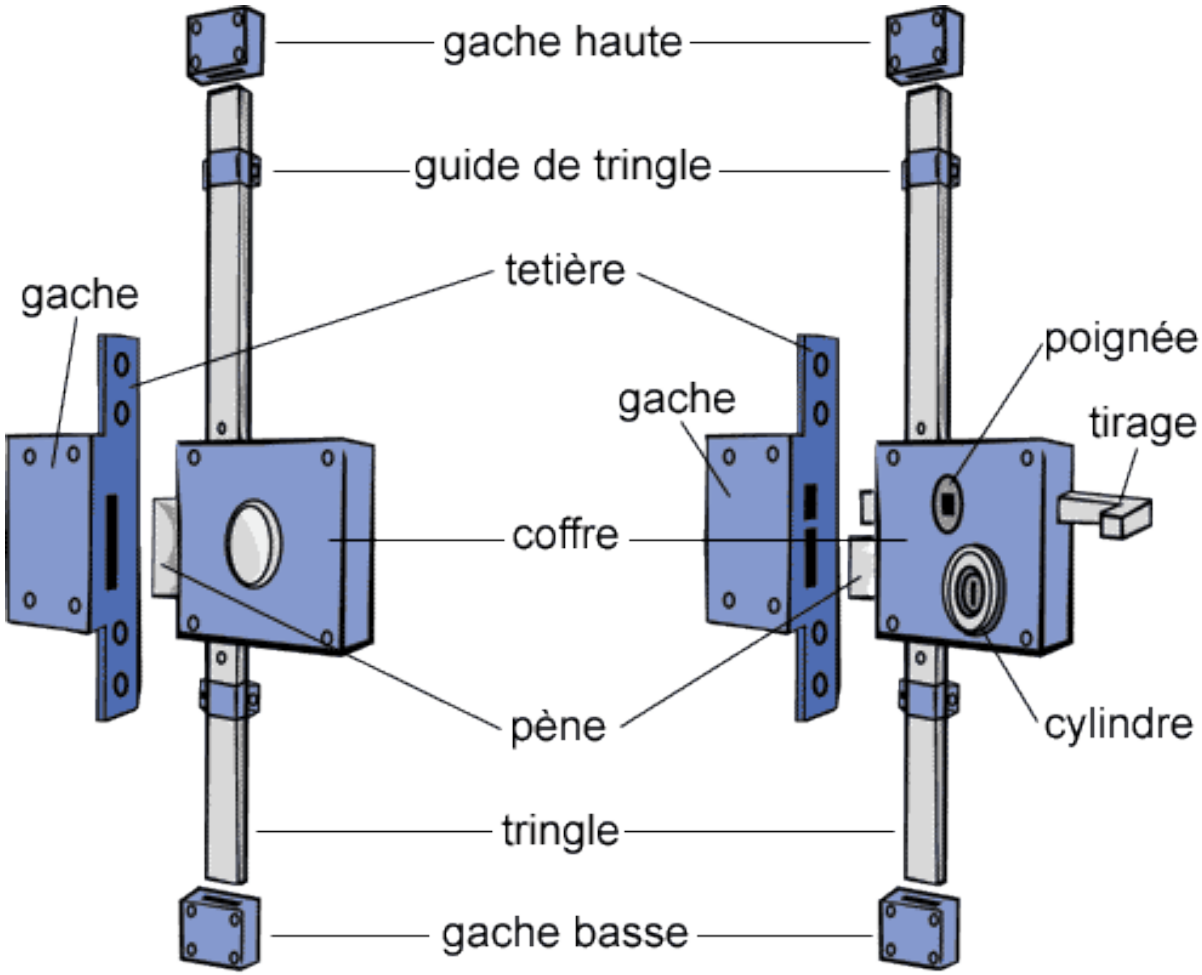
Le *schmilblick*

Mon objet possède :

- un coffre,
- une têtière,
- une ou plusieurs gâches,
- une pêne,
- un cylindre,
- une poignée...à quoi pensez vous ?

Abstraire

Le *schmilblick*



Abstraire

Pourquoi ?

- Abstraire permet de gérer la complexité des problèmes.
- Labstraction réalise une dichotomie entre le comportement dun objet (aspects extérieurs) et sa réalisation (constitution interne).
  - Linterface définit le vocabulaire connu de lobjet,
  - la réalisation donne une sémantique à ce vocabulaire.

Encapsuler

Grady Booch

« Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior ; encapsulation serves to separate the contractual interface of an abstraction and its implementation. »

Craig Larman [3]

« Encapsulation is a mechanism used to hide the data, internal structure, and implementation details of an object. All interaction with the object is through a public interface of operations. »

3. Craig Larman est un informaticien canadien spécialisé dans l'analyse et conception orientée objet

Encapsuler

En résumé

- Les classes doivent être opaques,
- ne doivent exposer que leurs interfaces,
- et ne pas laisser apparaître les détails de leur réalisation.
- C.-à.-d. utiliser des accesseurs et des modificateurs.

Remplacer

```
1 public double speed;
```

par

```
1 private double speed;
2 public double getSpeed() {
3     return speed;
4 }
5 public void setSpeed(double newSpeed) {
6     speed = newSpeed;
7 }
```

Remplacer

```
1 public boolean valid;
```

par

```
1 private boolean valid;
2 public boolean isValid() {
3     return valid;
4 }
5 public void setValid(boolean newValid) {
6     valid = newValid;
7 }
```

Encapsuler

Avantages

- Placer des contraintes (assertions dentrée et de sortie) sur les attributs :

```
1  public void setSpeed(double new_speed) {
2      if (new_speed < 0) throw new badValueException(...);
3
4      speed = new_speed;
5  }
```

ou

```
1  public void setSpeed(double new_speed) {
2      assert (new_speed >= 0) : "Valeur de vitesse negative !";
3
4      speed = new_speed;
5  }
```

- Ajouter des traitements supplémentaires :

```
1  public double setSpeed(double new_speed) {
2      speed = new_speed;
3      notifyObservers();
4  }
```

Encapsuler

Avantages

- Changer la représentation interne sans changer linterface :

Changement de

```
1  // Currently using internally Miles per Hour unit (mph) :
2  public void setSpeedInMPH(double new_speed) {
3      speedInMPH = new_speed;
4  }
5
6  public void setSpeedInKPH(double new_speed) {
7      speedInMPH = mph_to_kph(new_speed);
8  }
```

pour

```
1  // Now using metric units (kph, not mph) :
2  public void setSpeedInMPH(double new_speed) {
3      speedInKPH = mph_to_kph(new_speed);
4  }
5
6  public void setSpeedInKPH(double new_speed) {
7      speedInKPH = new_speed;
8  }
```

Encapsuler

Accesseurs et modificateurs : piège

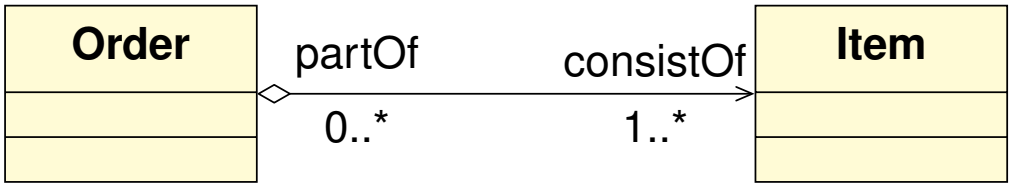
 Éviter d'utiliser les accesseurs et modificateurs dans le constructeur d'une classe dérivable sauf s'ils sont **finaux** !

- En effet, si la classe dérivée surcharge le modificateur, la superclasse sera initialisée selon la sémantique de la classe dérivée et non la sienne.

Principe n° 2 : composition ou héritage

Principe n° 2 : composition ou héritage

La composition



- Méthode de réutilisation d'objets en créant un nouvel objet composés d'autres objets.
- Créer des fonctionnalités en déléguant le traitement aux parties.
- L'interface du composé n'est pas directement exposé : on crée des fonctions enveloppes (*wrappers*) pour accéder à l'interface du composé.

La composition

Avantages

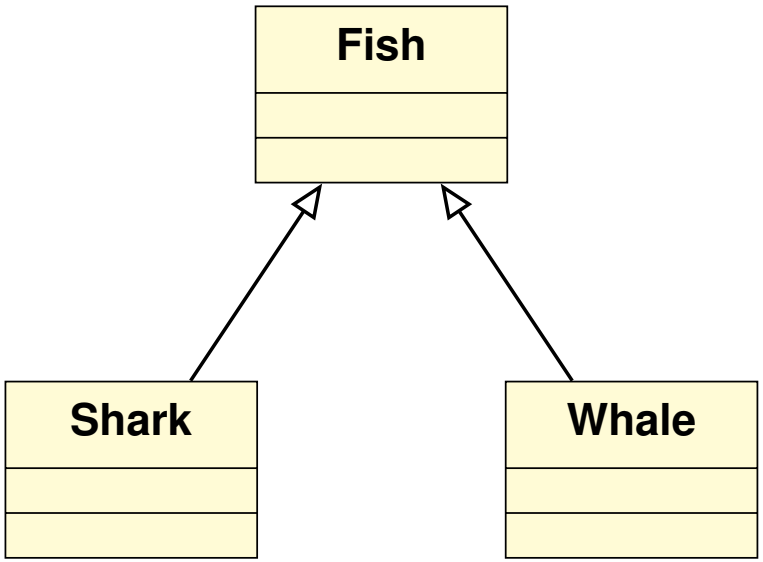
- Les composants peuvent être accédés depuis le composé uniquement par leur interface.
- Bonne encapsulation : les détails internes des composants sont masqués.
- Indépendances des implémentations.
- Chaque classe peut se concentrer sur une tâche.
- La composition peut être gérée dynamiquement.

Inconvénients

- La composition multiplie les objets et sous systèmes [4].
- Les interfaces doivent être soignées pour favoriser la réutilisabilité.

4. Rappel : pas d'aggrégation en Java.

L'héritage



- Méthode de réutilisation d'objets dans laquelle de nouvelles fonctionnalités sont obtenues en étendant l'implémentation d'un objet existant.

L'héritage

Généralisation :

La super classe factorise les attributs (valeurs ou méthodes) communs.

Spécialisation :

- La classe dérivée étend l'implémentation avec des attributs additionnels (en respectant l'invariant de la super...)
- Si l'invariant est faible utiliser une classe abstraite.
- Une classe dont toutes les méthodes sont abstraites est une interface.

L'héritage

Avantages

- L'extension est facile à écrire et à modifier puisque beaucoup d'attributs sont hérités.

Inconvénients

- Rompt avec l'encapsulation puisque la super classe est partiellement visible depuis la classe dérivée.
- Un changement d'implémentation d'une classe peut se répercuter sur son ascendance, comme sur sa descendance.
- L'héritage ne peut pas être manipulé à l'exécution.

Héritage ou composition

Un premier exemple

- Nous désirons utiliser une variante de HashSet qui trace les tentatives d'insertion.
- A priori du même type, nous dérivons la classe HashSet comme suit :

```
1  package verif;
2  import java.util.HashSet;
3
4  public class InstrumentedHashSet extends HashSet {
5      // The number of attempted element insertions
6      private int addCount = 0;
7      public InstrumentedHashSet() { super(); }
8      public InstrumentedHashSet(Collection c) { super(c); }
9      public InstrumentedHashSet(int initCap, float loadFactor) {
10         super(initCap, loadFactor);
11     }
12     public boolean add(Object o) {
13         addCount++; return super.add(o);
14     }
15     public boolean addAll(Collection c) {
16         addCount += c.size(); return super.addAll(c);
17     }
18     public int getAddCount() { return addCount; }
19 }
```

Héritage ou composition

Un premier exemple

- Nous faisons l'utilisation suivante de cette nouvelle classe :

```
1   ...
2       Collection c = new ArrayList();
3       c.add(new Integer(1));
4       c.add(new Integer(2));
5       c.add(new Integer(3));
6
7       InstrumentedHashSet ihs=new InstrumentedHashSet();
8       ihs.addAll(c);
9
10      System.out.println("Nombre de tentatives d'insertions : "+ihs.
            getAddCount());
11      ...
12  }
```

- Quel est le résultat de ce programme ?

Héritage ou composition

Un premier exemple

- Nous faisons l'utilisation suivante de cette nouvelle classe :

```
1  ...
2      Collection c = new ArrayList();
3      c.add(new Integer(1));
4      c.add(new Integer(2));
5      c.add(new Integer(3));
6
7      InstrumentedHashSet ihs=new InstrumentedHashSet();
8      ihs.addAll(c);
9
10     System.out.println("Nombre de tentatives d'insertions : "+ihs.
           getAddCount());
11     ...
12 }
```

- Quel est le résultat de ce programme ?

Héritage ou composition

Un premier exemple

- L'exécution de ce programme donne six tentatives d'insertion et non trois (la méthode addAll utilisant la méthode add, il existe plusieurs manières de résoudre ce problème.)
- Notons que pour dériver une classe nous devons connaître les détails de son code.
- Une solution, plus générale, est d'utiliser la composition, même si elle est techniquement longue [5].
- La classe InstrumentedSet est alors connu sous le nom de « *wrapper* » ou *enveloppe*.

5. Le patron *proxy dynamique* peut faciliter cette tâche.

Héritage ou composition

Un premier exemple V2

Une version avec la composition :

```
1  package verif;
2  import java.util.HashSet;
3
4  public class InstrumentedSet implements Set {
5     private final Set s;
6     private int addCount = 0;
7     public InstrumentedSet(Set s) { this.s = s; }
8     public boolean add(Object o) {
9        addCount++; return s.add(o);
10    }
11    public boolean addAll(Collection c) {
12       addCount += c.size(); return s.addAll(c);
13    }
14    public int getAddCount() { return addCount; }
15
16    // Forwarding methods (the rest of the Set interface methods)
17    public void clear() { s.clear(); }
18    public boolean contains(Object o) { return s.contains(o); }
19    public boolean isEmpty() { return s.isEmpty(); }
20    public int size() { return s.size(); }
21    public Iterator iterator() { return s.iterator(); }
22    public boolean remove(Object o) { return s.remove(o); }
23    public boolean containsAll(Collection c){ return s.containsAll(c); }
24    public boolean removeAll(Collection c){ return s.removeAll(c); }
25    public boolean retainAll(Collection c){ return s.retainAll(c); }
26    ...
27 }
```

Héritage ou composition

Un premier exemple V2

Notons que :

- Cette classe est un Set
- Elle a un constructeur dont l'argument est un Set
- Le composant Set peut être de tout type concret réalisant l'interface Set (donc HashSet...)
- Exemple :

```
1  List list = new ArrayList();
2  Set s1 = new InstrumentedSet(new TreeSet(list));
3  int capacity = 7; float loadFactor = .66f;
4  Set s2 = new InstrumentedSet(new HashSet(capacity,loadFactor));
```

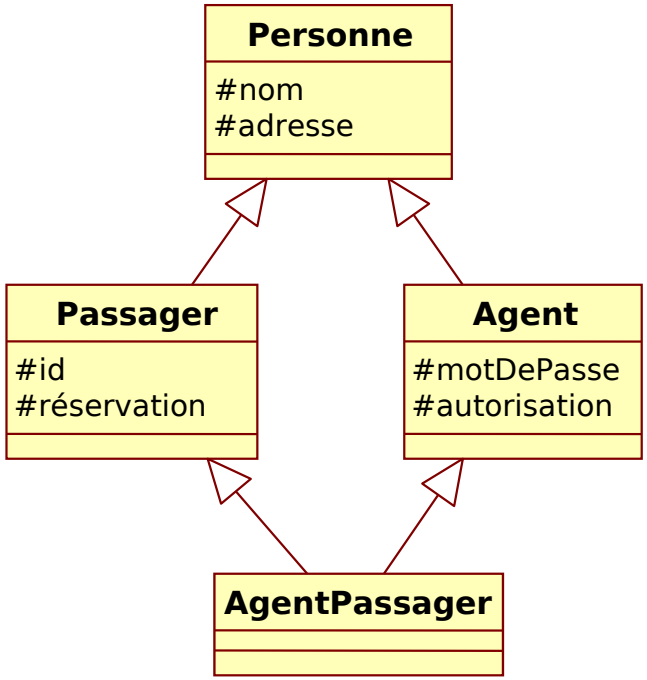
Règle de codage

N'utiliser l'héritage que lorsque tous les critères suivants sont remplis :

1. Une classe dérivée correspond à « est une sorte de »et non a « est un rôle joué par »
2. Une instance d'une classe dérivée ne doit pas devenir un objet d'une autre classe pendant le processus.
3. Une classe dérivée étend plutôt que surcharge ou annule les propriétés de sa super classe.
4. Une classe dérivée ne doit pas hériter d'une classe clairement utilitaire.
5. Pour une classe d'un domaine d'étude, une classe dérivée spécialise un rôle, une transaction ou un mécanisme.

Règle de codage

Exemple 1



✗ « est une sorte de » pas « un rôle joué par » : Passager, Agent sont des rôles

✗ Pas de transmutation possible : une instance de Passager peut devenir Agent ou AgentPassager...

✓ Étend plutôt que surcharge ou annule.

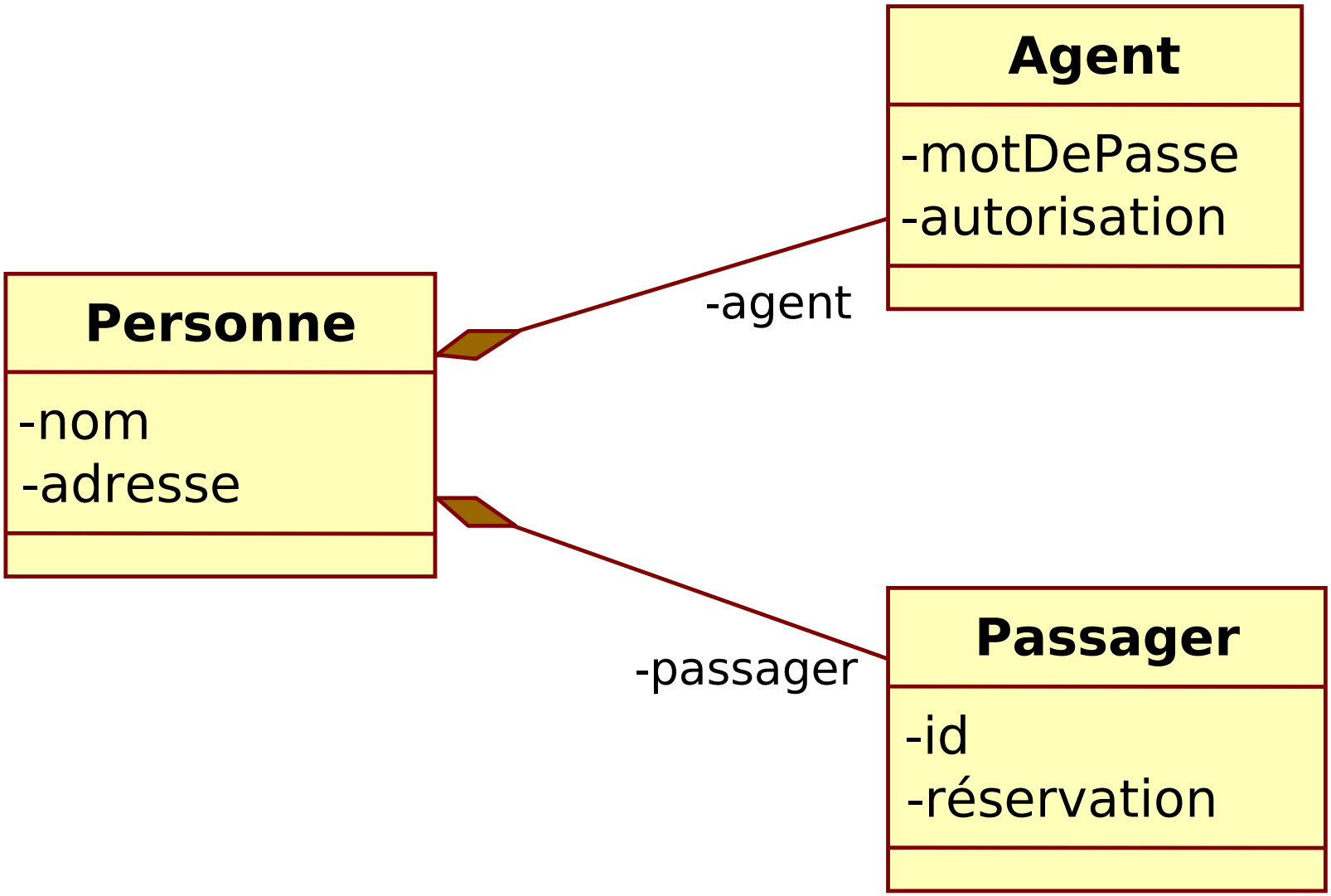
✓ N'étend pas une classe utilitaire.

✗ Dans le domaine du problème, spécialise un rôle, une transaction ou un mécanisme : Personne n'est pas un rôle, une transaction ou un mécanisme.

✗ **Pas d'héritage ici !**

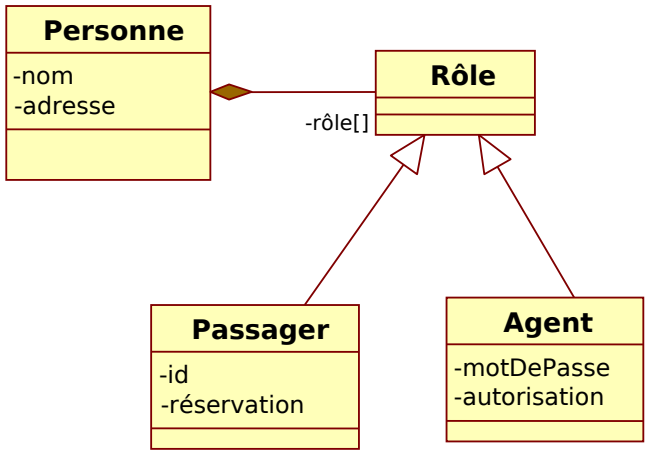
Règle de codage

Exemple 1 : composition seule



Règle de codage

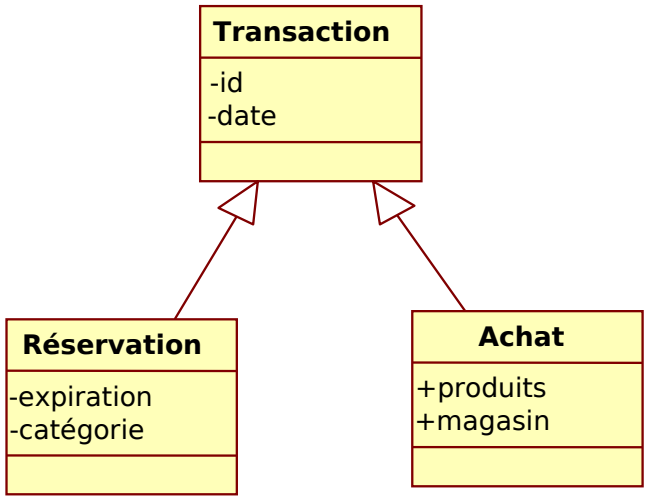
Exemple 1 : composition + héritage



- « est une sorte de » pas « un rôle joué par » : Passager, Agent sont deux sortes de rôles
- Pas de transmutation : une instance de Passager reste Passager, une instance d'Agent reste Agent et une Personne peut jouer plusieurs rôles...
- Étend plutôt que surcharge ou annule.
- N'étend pas une classe utilitaire.
- Dans le domaine du problème, spécialise un rôle, une transaction ou un mécanisme : Rôle est un rôle.
- **Composition + héritage ici !**

Règle de codage

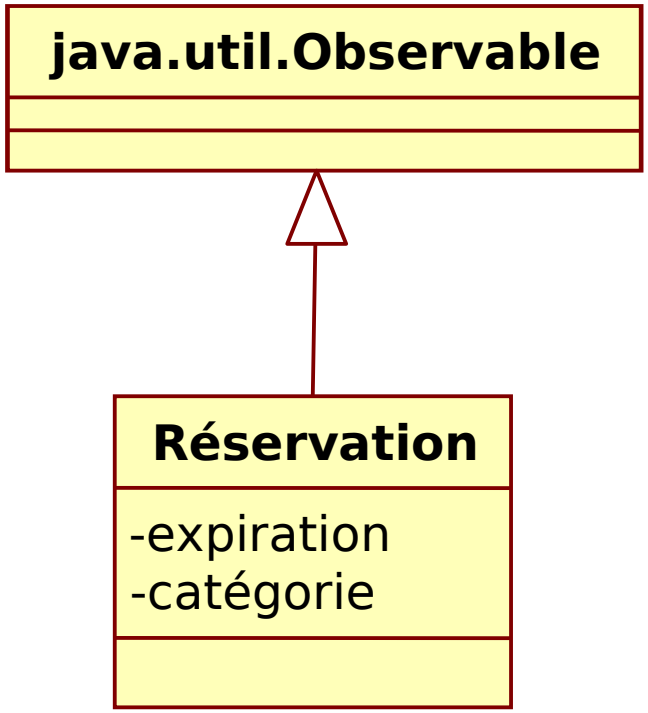
Exemple 2 : héritage



- « est une sorte de » pas « un rôle joué par » : réservation et achat sont deux sortes de transaction.
- Pas de transmutation : une réservation reste une Réservation, un achat reste un Achat...
- Étend plutôt que surcharge ou annule.
- N'étend pas une classe utilitaire.
- Dans le domaine du problème, spécialise un rôle, une transaction ou un mécanisme.
- **Héritage ici !**

Règle de codage

Exemple 3 : héritage



✗ « est une sorte de » pas « un rôle joué par » : une réservation n'est pas une sorte d'Observable.

✓ Pas de transmutation : une réservation reste une Réservation, un achat reste un Achat...

✓ Étend plutôt que surcharge ou annule : ?

✗ N'étend pas une classe utilitaire.

✗ Dans le domaine du problème, spécialise un rôle, une transaction ou un mécanisme : Observable n'est pas liée au domaine.

✓ **Pas d'héritage ici !**

Règle de codage

Conclusion

- La composition et l'héritage sont deux méthodes importantes de réutilisation.
- L'héritage est souvent utilisé à mauvais escient dans le développement Orienté Objet.
- Héritage et composition doivent coopérer.
- Dans la suite nous étudierons les patrons de conception qui sont des modèles à réutiliser.
- En cas de doute le principe reste

  Favoriser la Composition plutôt que l'héritage.

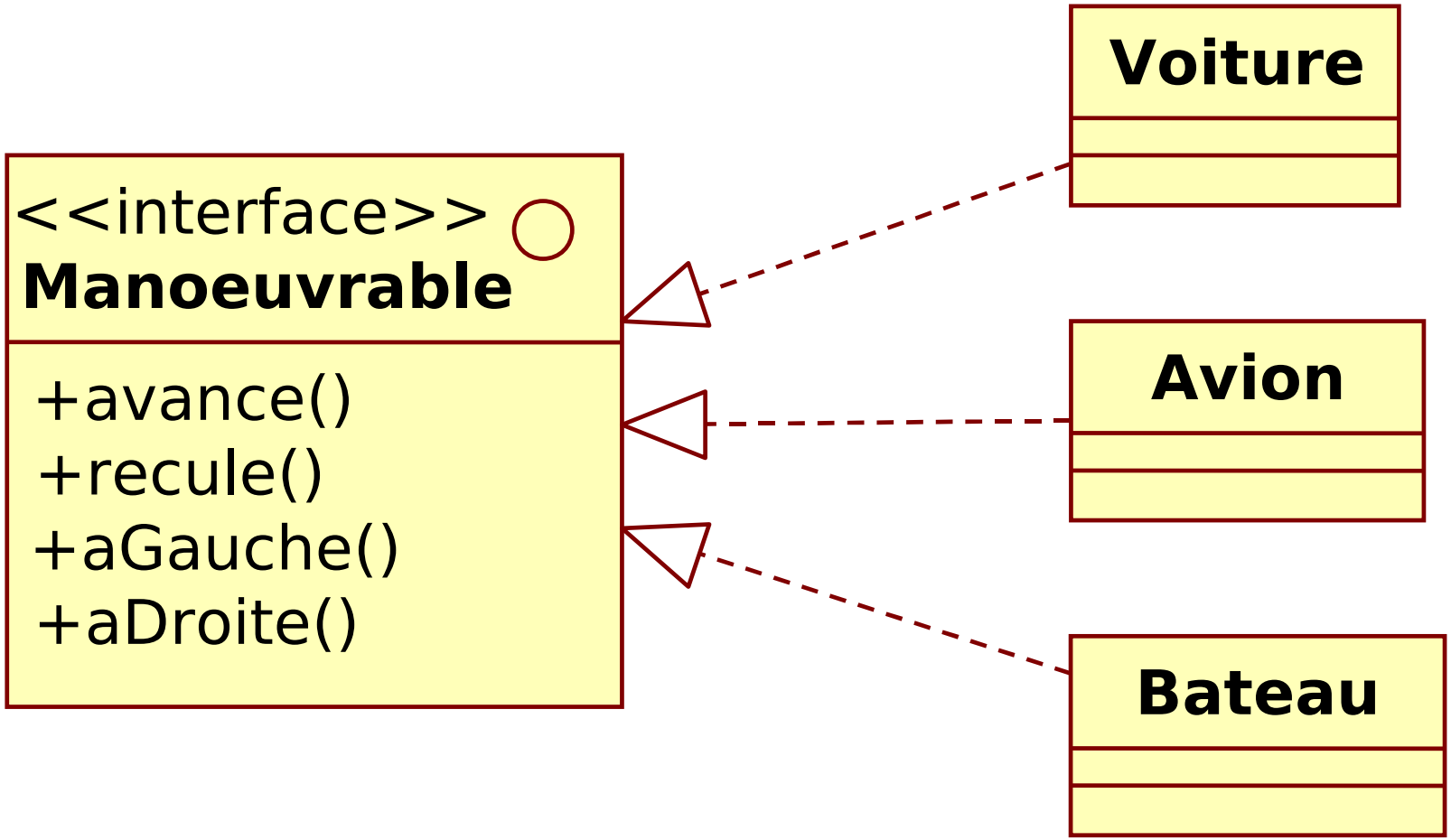
Principe n° 3 : favoriser les interfaces

Interface : syntaxe supportée par un objet

- Une *interface* est l'ensemble des méthodes d'un objet qui peuvent être invoquées par un autre objet.
- Un objet peut avoir plusieurs interfaces. Dans ce cas une interface est une partie des méthodes visibles de l'objet.
- Un type déclaré doit être une interface spécifique d'un objet.
- Différents objets peuvent avoir le même type et un objet peut avoir plusieurs types.
- Un objet ne devrait être connu des autres objets qu'à travers son (ses) interface(s).
- *Les interfaces sont la clé de l'extensibilité !*

Principe n° 3 : favoriser les interfaces

Interface : syntaxe supportée par un objet



Principe n° 3 : favoriser les interfaces

Héritage de classe vs héritage d'interface

Héritage de Classe

Définit le code d'une classe à partir d'autres classes.

Héritage d'interface

Décrit comment un objet peut être utilisé à la place d'autres objets.

- Le mécanisme d'héritage de C++ concerne les classes et interfaces : les interfaces sont simulées par des classes abstraites pures.
- Java sépare les deux concepts : les interfaces (héritage multiple) et les classes (héritage simple).
- Les interfaces Java facilitent la conception des objets en se concentrant sur le modèle du domaine à étudier.

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

SOLID est un acronyme désignant un ensemble de cinq principes [6] 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*

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

Single responsibility principle

Une responsabilité unique

Définition (Principe de responsabilité simple [7])

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 ».

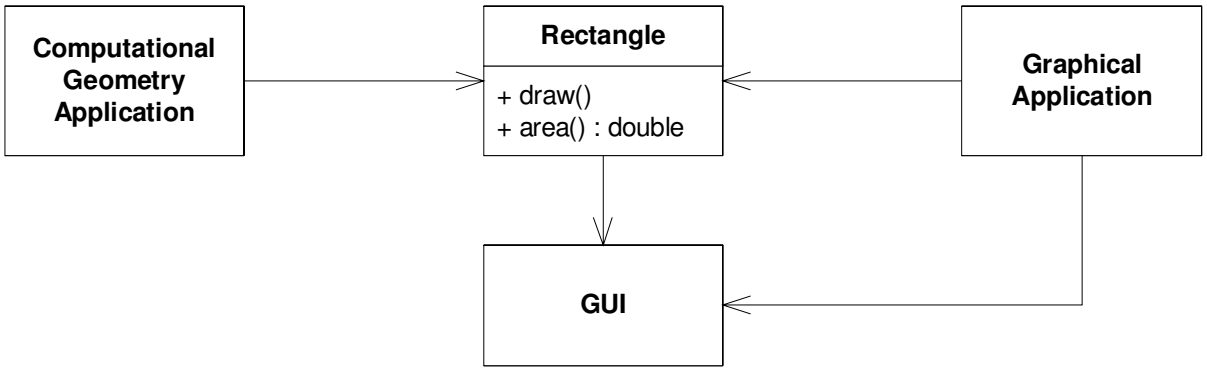
« THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE. »

- 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.

7. Robert C. *Single responsibility principle*

Single responsibility principle

Exemple - mauvaise conception



La classe *Rectangle* a deux responsabilités :

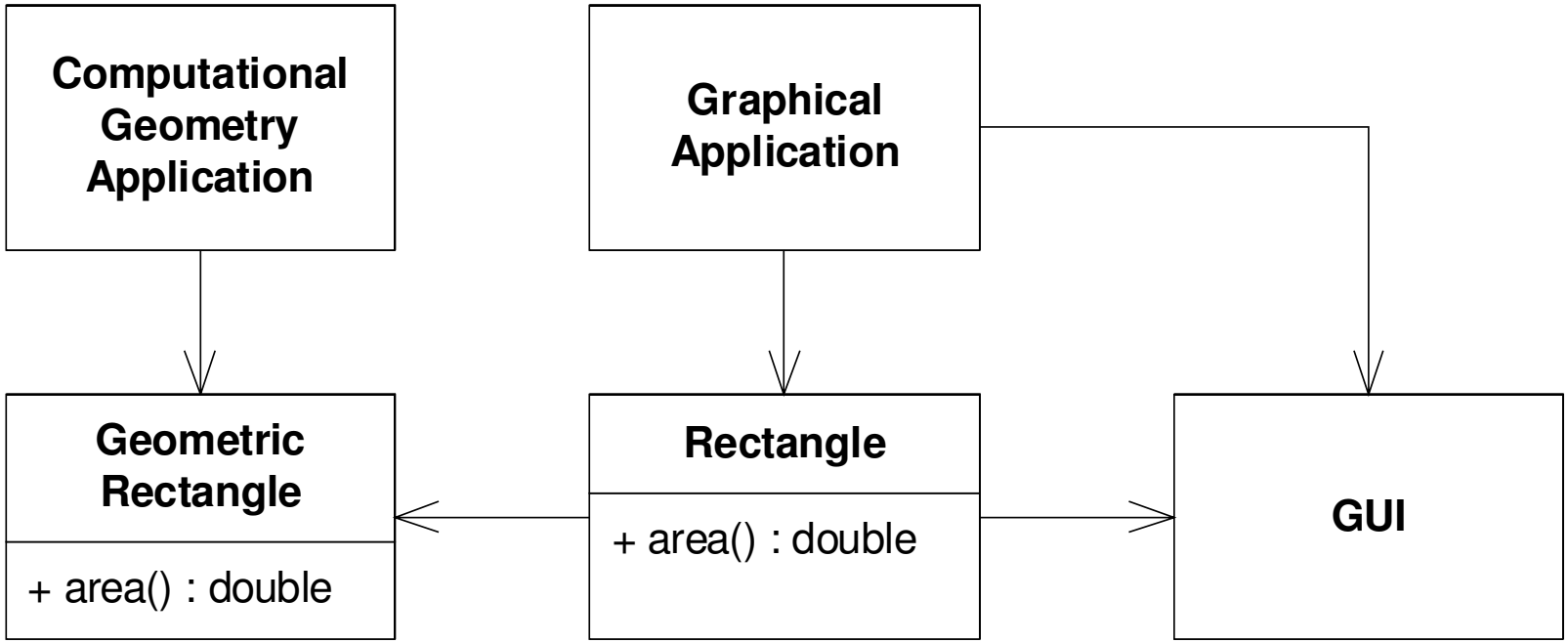
1. fournir le modèle mathématique d'un rectangle,
2. effectuer son rendu sur une interface graphique.

La classe *Rectangle* viole le SRP, conséquences :

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

Single responsibility principle

Exemple - bonne conception

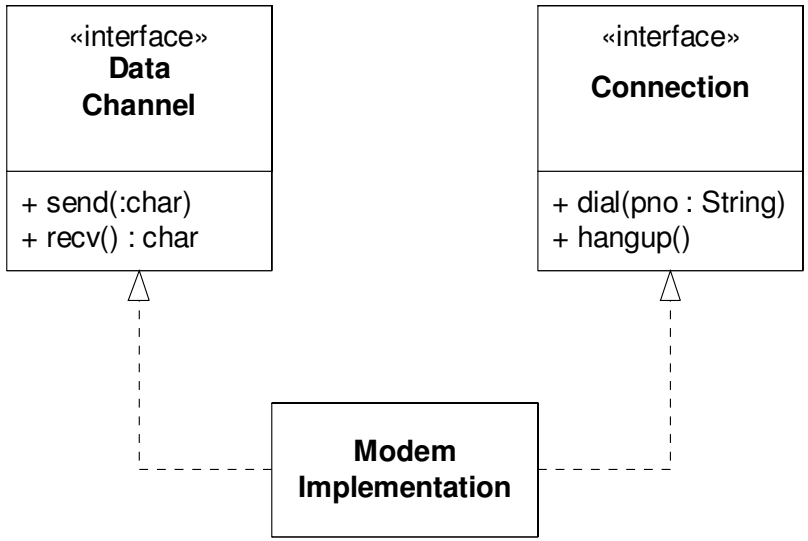


- Les deux responsabilités sont placées dans des classes séparées.
- Maintenant, un changement dans le rendu graphique n'affecte plus l'application de calcul.

Single responsibility principle

Identifier les responsabilités

```
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.
- Deux interfaces mais une implémentation (parfois nous n'avons pas le choix),
- mais au moins le code client peut être découplé.

Single responsibility principle

En résumé

- *Single Responsability Principle* est un des principes les plus simples mais paradoxalement un des plus difficile à mettre en oeuvre.
- Séparer les responsabilités, au moins au niveau des interfaces, permet de découpler les clients.

Open/Close principle

Ivar Jacobson [8]

« 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 [9]))

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

8. Informaticien suédois, un des concepteur d'UML.
9. 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 code source 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 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

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 dappliquer différents coefficients correcteurs aux prix des produits.

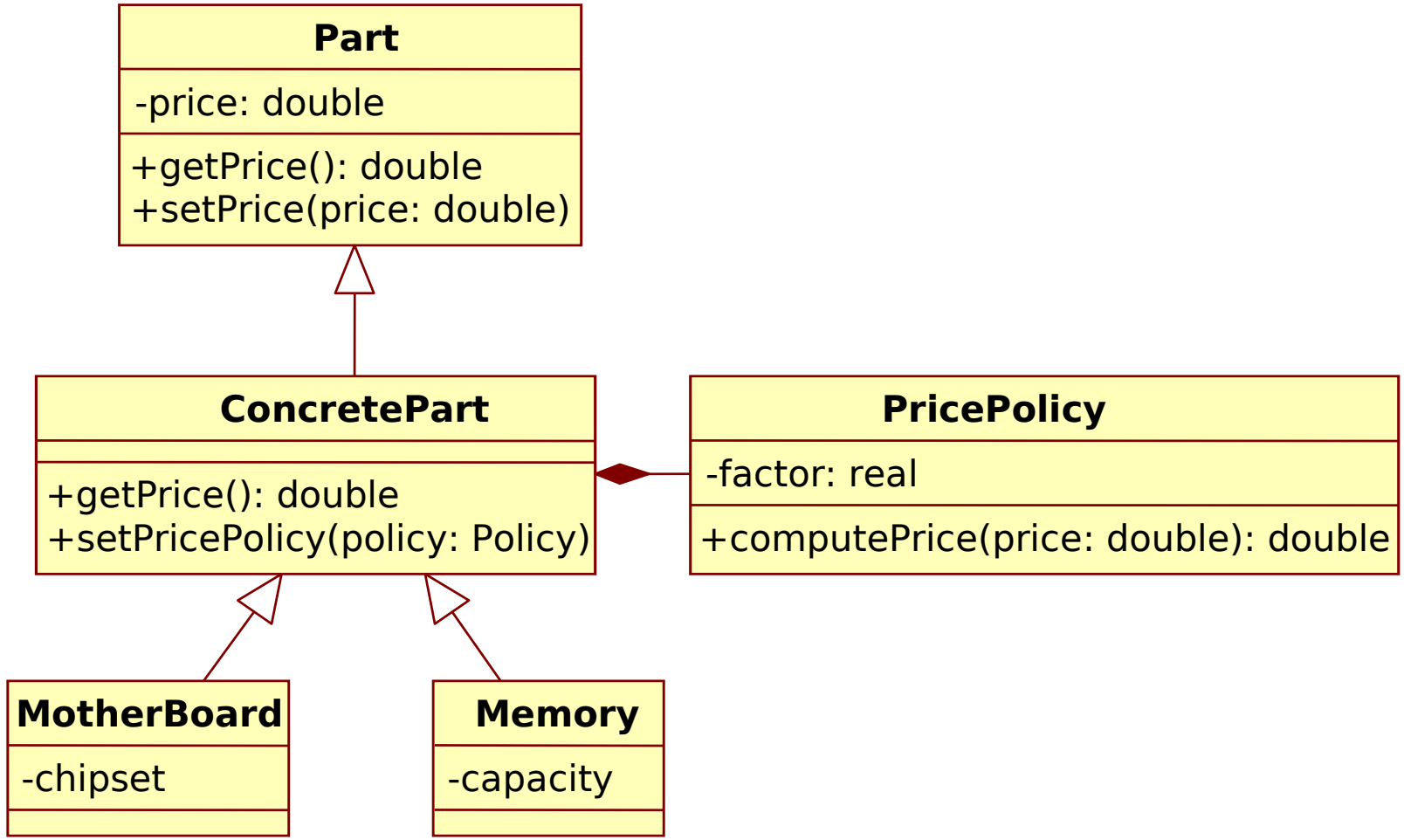
- Une **très mauvaise idée** est de réécrire le code :

```
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



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 dun 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 datteindre un haut niveau de robustesse et de réutilisation.
- En résumé : on fait du « **développement durable** ».

Liskov Substitution Principle

Définition (Liskov Substitution Principle (Barbara Liskov [10]))

« 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 }
```

10. 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 de la superclasse.
- Une telle fonction viole également OCP, puisque son code doit être modifié lorsqu'une sous classe est crée.
- Mais il existe des cas plus complexes.

Liskov Substitution Principle

Exemple

Soit le module :

```
1  package geo;
2
3  public class Rectangle { // no 0 <= width, height here
4     private double width;
5     private double height;
6     public Rectangle(double w, double h) {width=w;height=h;}
7     public double getWidth() { return width; }
8     public double getHeight() { return height; }
9     public void setWidth(double w) { width = w; }
10    public void setHeight(double h) {height = h; }
11    public double area() {return (width * height);}
12 }
```

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 ne nous concerne pas vraiment.
- 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 !

Liskov Substitution Principle

Exemple

- Définition de la calsse `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

```
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

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.
- LSP contribue à respecter OCP.

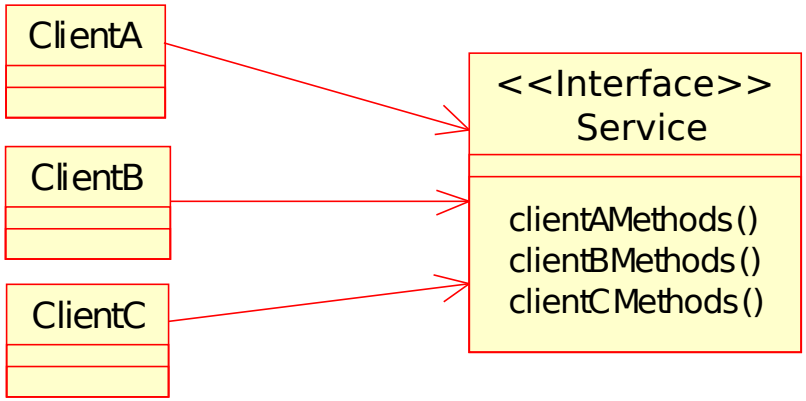
Interface Segregation Principle

- 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

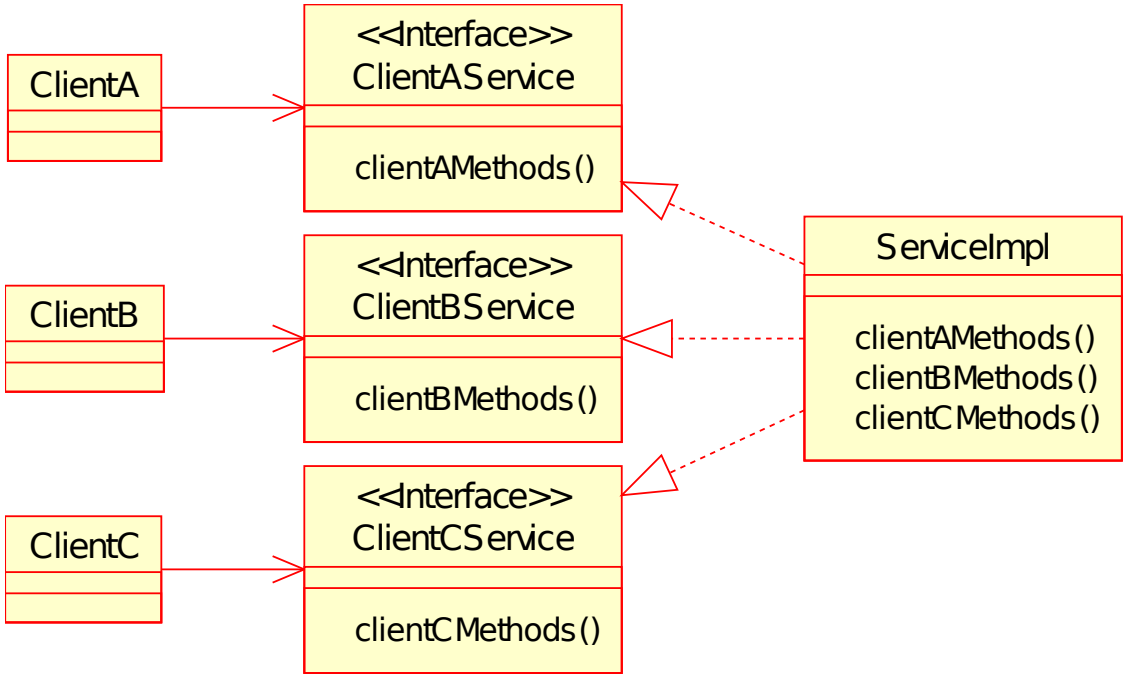


L'amalgame d'interfaces provoque :

- 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

Un système sécurisé :

- Une porte verrouillable.
- On souhaite ajouter une fonction de temporisation (signal sonore si la porte reste ouverte trop longtemp).

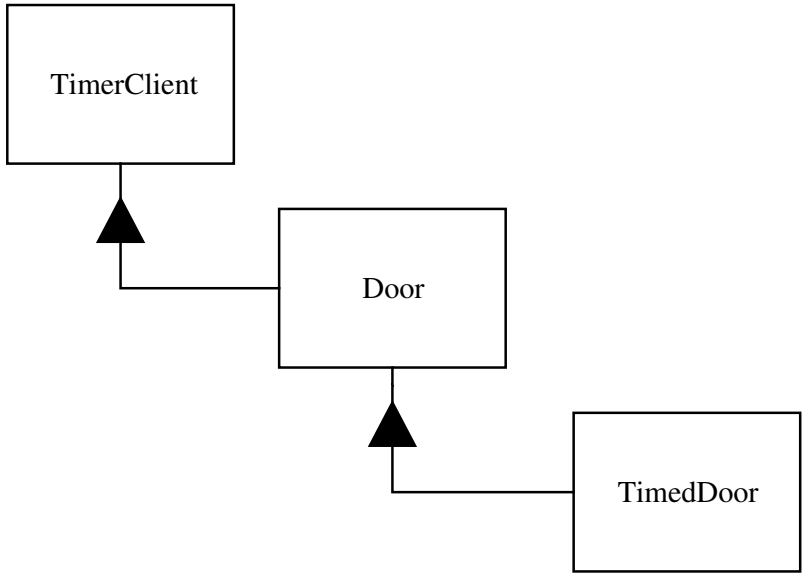
```
1  class Door {
2  public:
3      virtual void Lock() = 0;
4      virtual void Unlock() = 0;
5      virtual bool IsDoorOpen() = 0;
6  };
```

```
1  class Timer {
2  public:
3      void Register(int timeout,
4          TimerClient* client);
5  };
6
7  class TimerClient {
8  public:
9      virtual void TimeOut() = 0;
10 };
```

Comment faire en sorte qu'une porte (*Door*) puisse s'enregistrer auprès d'un temporisateur (*Timer*) ?

Interface Segregation Principle

Exemple 1



Une mauvaise solution : faire hériter *Door* de *TimerClient*.

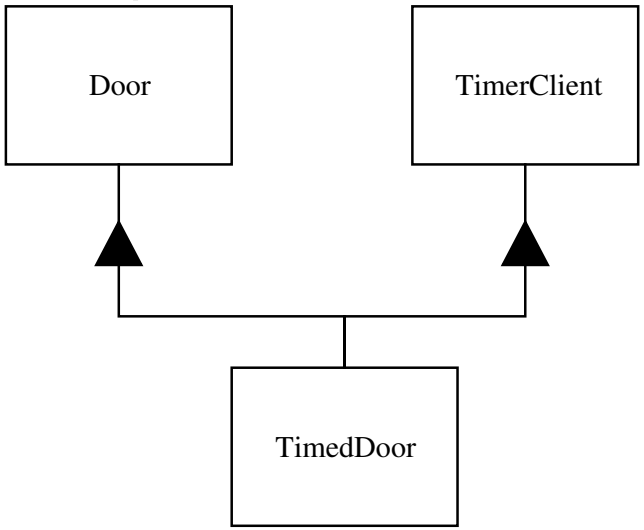
- Toutes les sortes de portes auront l'interface de *TimerClient*,
- et aussi : *Door* n'est pas une sorte de *TimerClient* !!!

Interface Segregation Principle

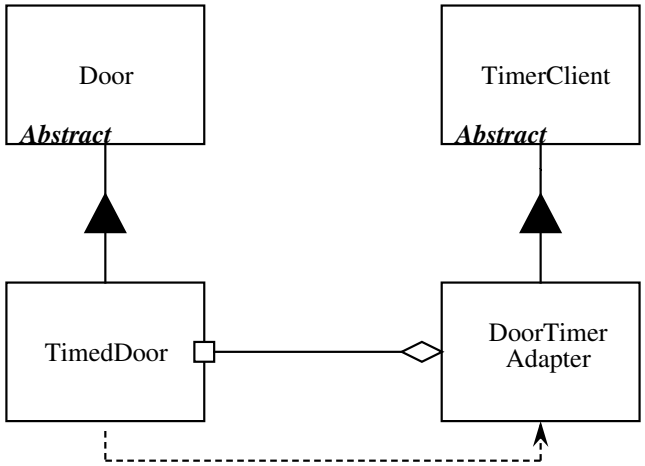
Exemple 1

Séparer les interfaces selon leur responsabilité.

Séparation par héritage multiple :



Séparation par délégation :



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

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. »*

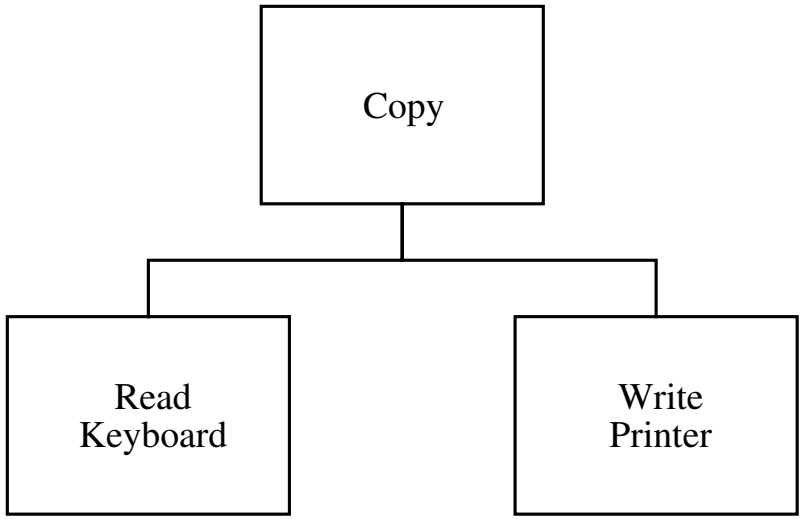
Dependency Inversion principle

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  }
```

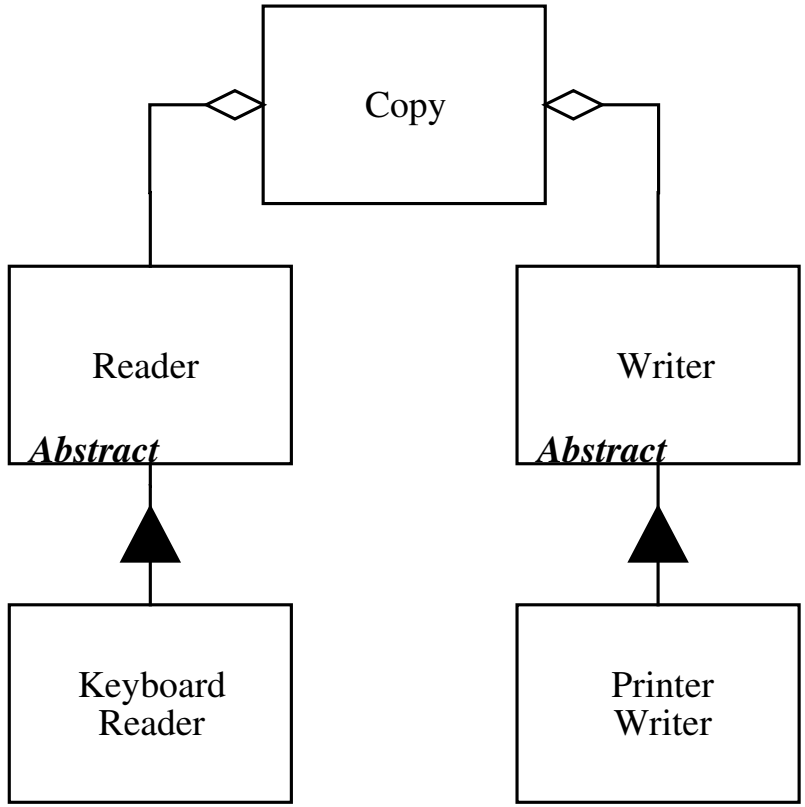


- Ici, le module de haut niveau dépend des modules de bas niveau,
- le principe DIP est violé.

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*.

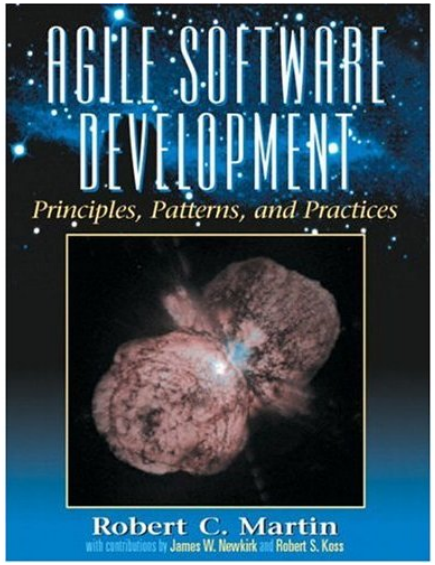
Dependency Inversion principle

En résumé

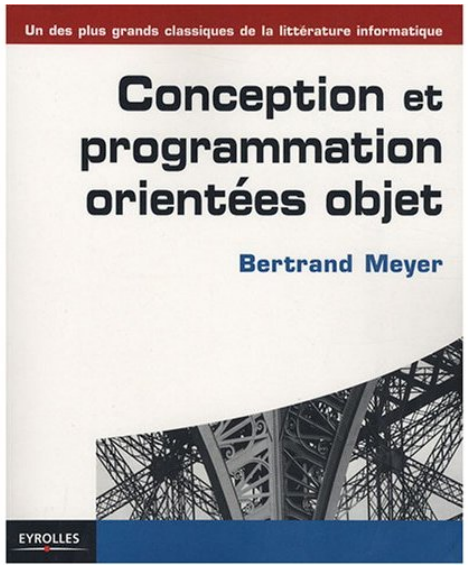
- *Dependency Inversion Principle* : les modules de haut niveau ne doivent pas dépendre de modules de bas niveau.
- Le respect de ce principe apporte :
  - une conception en couche,
  - des composants de haut niveau génériques/réutilisables.
- On parle d'**inversion de dépendances** car les interfaces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

Introduction aux *Design Patterns*

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.