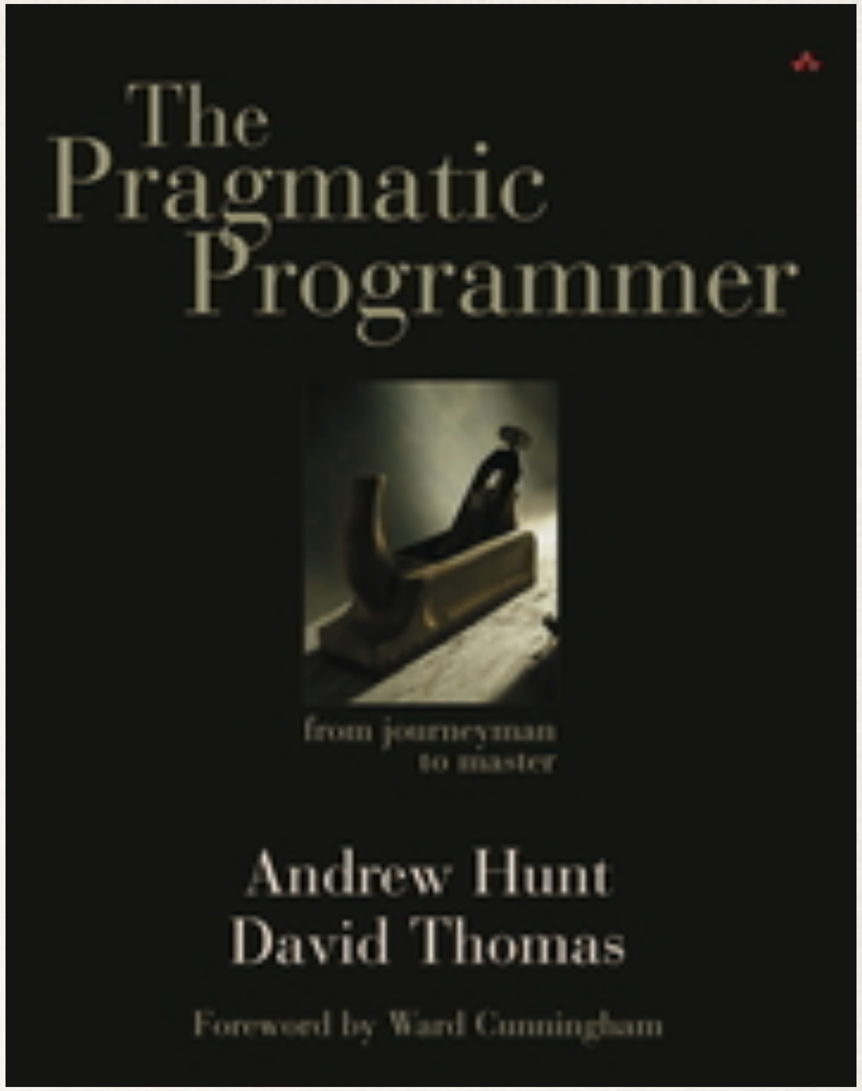
Principes SOLID
suivi de
Pragmatic Programming



The Pragmatic Programmer: From Journeyman to Master

by Andrew Hunt and David Thomas

Mireille Blay-Fornarino, Université Nice Sophia Antipolis, Département Info Iut, Septembre 2015

Objectifs de ce cours

- **Mieux comprendre votre rôle en tant que «Développeur»**

«Les développeurs avancés voient très vite l'intérêt, les débutants beaucoup moins. Quelques années plus tard, ils comprennent pourquoi c'était important!» Anonyme.

Au delà des méthodes

- Having a process is not the same as having the skills to carry out that process

— Jim Highsmith

Attention, version édulcorée (pragmatique?) du livre pour ne garder que ce qui peut vous «parler» dès à présent.

Ecrire du bon code :
Les principes S.O.L.I.D.


SOLID
Software Development is not a Jenga game

S.O.L.I.D : l'essentiel !

- **Single responsibility principle (SRP)** : une classe n'a qu'une seule responsabilité (ou préoccupation).
- **Open/closed principle (OCP)** : une classe doit être ouverte à l'extension (par héritage, par exemple) mais fermé à la modification (attributs privés, par exemple).
- **Liskov subtition principle (LSP)** : les objets d'un programme doivent pouvoir être remplacés par des instances de leurs sous-types sans «casser» le programme.
- **Interface segregation principle (ISP)** : il vaut mieux plusieurs interfaces spécifiques qu'une unique interface générique.
- **Dependency inversion principle (DIP)** : il faut dépendre des abstractions, pas des réalisations concrètes.

SOLID: Open/Closed Principle (OCP)

A class should have one, and only one, reason to change.
Robert C. Martin.

http://deviq.com/single-responsibility-principle

Principe ouvert / fermé
Open/Closed Principle (OCP)

> You should be able to extend a classes behavior, without modifying it.
> Robert C. Martin.

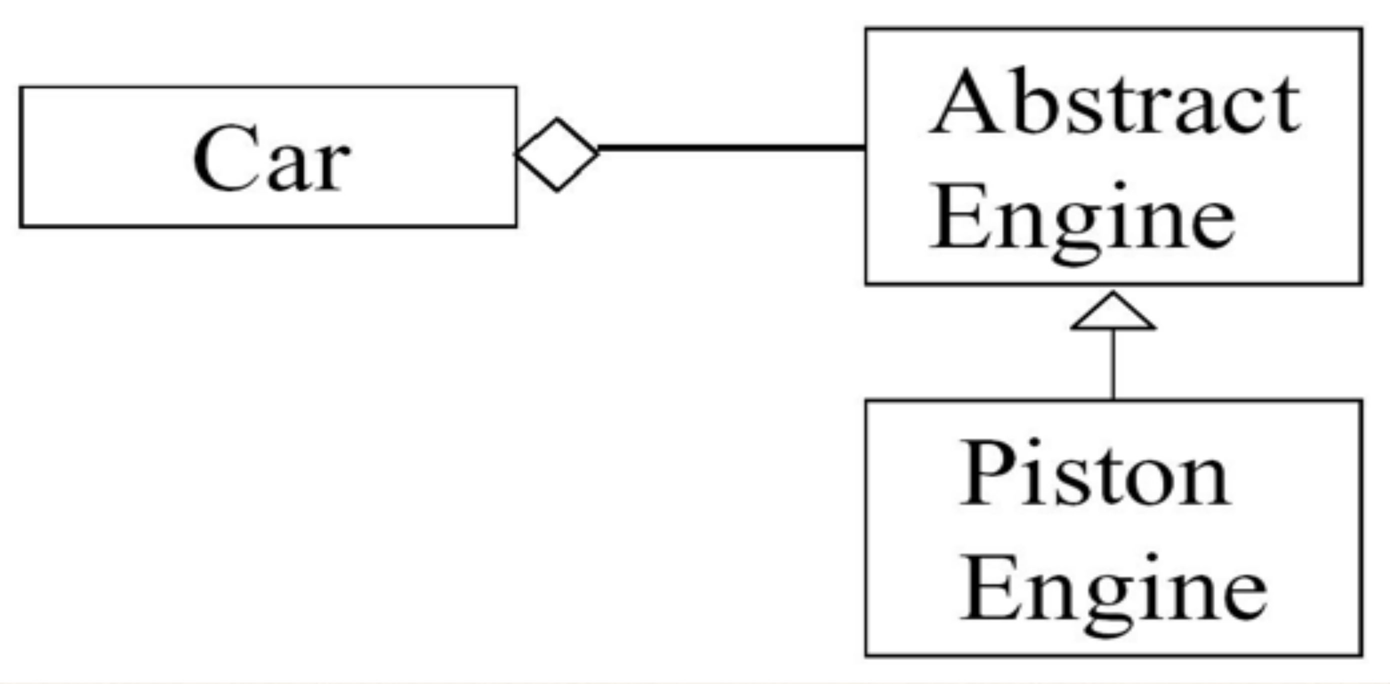
- **Les entités logicielles doivent être ouvertes à l'extension**
  - le code est extensible
- **mais fermées aux modifications**
  - Le code a été écrit et testé, on n'y touche pas.

http://afsy.fr/avent/2013/02-principes-stupid-solid-poo

Open the door …

- Comment faire en sorte que la voiture aille plus vite à l'aide d'un turbo?
  - Il faut changer la voiture
    - avec la conception actuelle…

… But Keep It Closed!

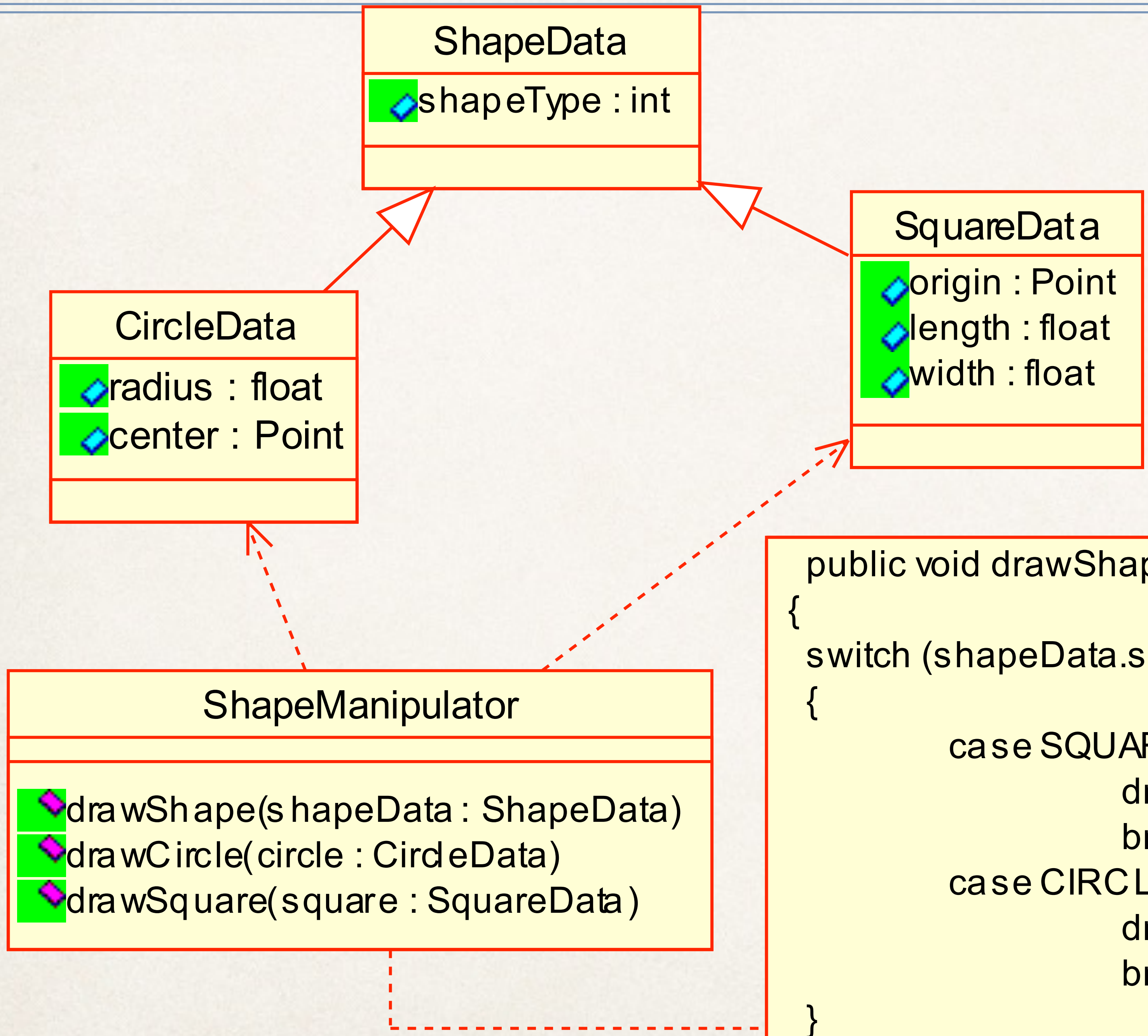


- On retient :
  - Une classe ne doit pas dépendre d'une classe Concrète.
  - Elle peut dépendre d'une classe abstraite …
  - et utiliser le polymorphisme

The Open/Closed Principle (OCP) Example

An Example of What not to do!

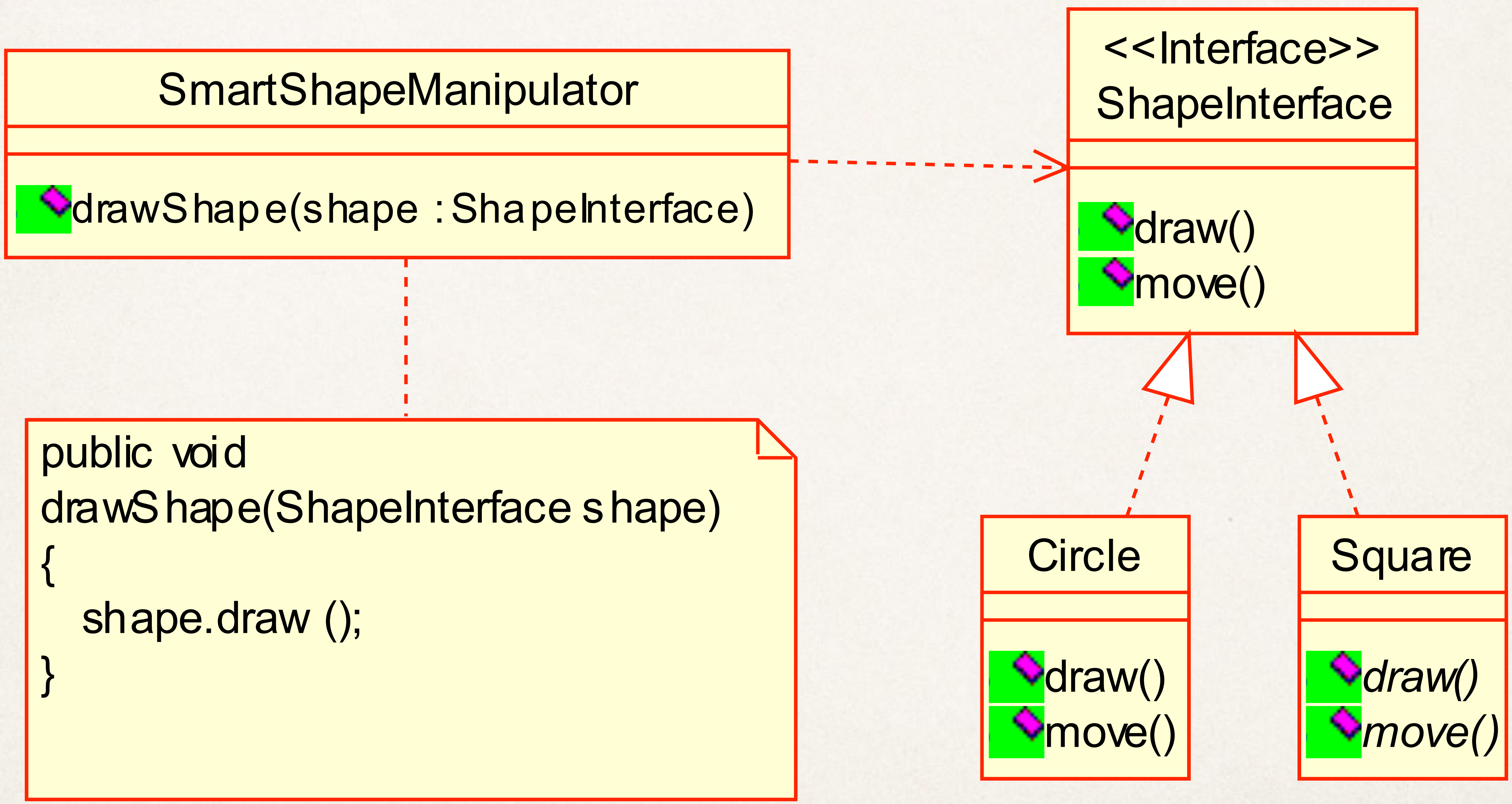
What is wrong with this code?




```
public void drawShape(ShapeData shapeData)
{
 switch (shapeData.shapeType)
 {
        case SQUARE:
                drawSquare((SquareData)shapeData);
                break;
        case CIRCLE:
                drawCircle((CircleData)shapeData);
                break;
 }
}
```


www2.cs.uh.edu/~svenkat/ooad/slides/PPT/session07.ppt

The Open/Closed Principle (OCP) Example



The Open-Closed Principle(OCP) : allons plus loin (1)

- Le travail de cette méthode est de calculer le prix total d'un ensemble de «parties»

```
public double totalPrice(Part[] parts) {
    double total = 0.0;
    for (int i=0; i<parts.length; i++) {
        total += parts[i].getPrice();
    }
    return total;
}
```

The Open-Closed Principle(OCP) : allons plus loin (2)

- «But the Accounting Department decrees that motherboard parts and memory parts should have a premium applied when figuring the total price.»
- Que pensez-vous du code suivant?

```
public double totalPrice(Part[] parts) {
    double total = 0.0;
    for (int i=0; i<parts.length; i++) {
        if (parts[i] instanceof Motherboard)
            total += (1.45 * parts[i].getPrice());
        else if (parts[i] instanceof Memory)
            total += (1.27 * parts[i].getPrice());
        else
            total += parts[i].getPrice();
    }
    return total;
}
```

The Open-Closed Principle(OCP) : allons plus loin (3)

- Des exemples de classes *Part* et *ConcretePart*

```
// Class Part is the superclass for all parts.
public class Part {
    private double price;
    public Part(double price) {
        this.price = price;}
    public void setPrice(double price) {
        this.price = price;}
    public double getPrice() {
        return price;}
}

// Class ConcretePart implements a part for sale.
// Pricing policy explicit here!
public class ConcretePart extends Part {
    public double getPrice() {
    // return (1.45 * price); //Premium
    return (0.90 * price); //Labor Day Sale
    }
}
```

Mais si maintenant on veut modifier la politique de gestion des prix, par exemple en lisant dans une base de données, en modifiant les facteurs de calcul des prix ....

The Open-Closed Principle(OCP) : allons plus loin (4)

- Une meilleure idée est d'avoir une classe *PricePolicy* qui permettra de définir différentes politiques de prix:

```
// The Part class now has a contained PricePolicy object.
public class Part {
     private double price;
     private PricePolicy pricePolicy;

     public void setPricePolicy(PricePolicy pricePolicy) {
          this.pricePolicy = pricePolicy;}

     public void setPrice(double price) {this.price = price;}
     public double getPrice() {return pricePolicy.getPrice(price);}
}
```

The Open-Closed Principle(OCP) : allons plus loin (5)

```
/**
* Class PricePolicy implements a given price policy.
*/
public class PricePolicy {
    private double factor;

    public PricePolicy (double factor) {
    this.factor = factor;
    }

public double getPrice(double price) {return price * factor;}
}
```

D’autres politiques comme un calcul de la ristourne par «seuils» sont maintenant possibles ...

The Open-Closed Principle(OCP) : allons plus loin (6)

- With this solution we can dynamically set pricing policies at run time by changing the PricePolicy object that an existing Part object refers to

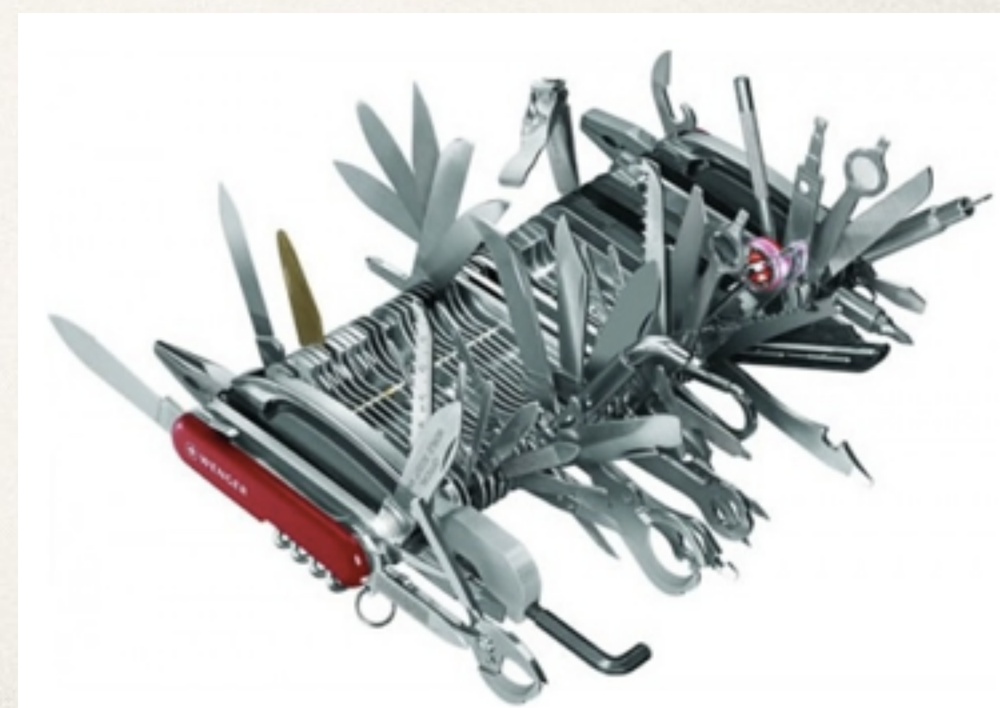
- Of course, in an actual application, both the price of a Part and its associated PricePolicy could be contained in a database

The Open-Closed Principle (OCP)

- Il est impossible que tous les éléments d'un système logiciel satisfasse l'OCP, mais l’objectif est de minimiser le nombre des éléments qui ne le satisfont pas.
- Le principe ouvert-fermé est vraiment au cœur de la conception OO.
- La conformité à ce principe donne un meilleur niveau de réutilisabilité et maintenabilité.

SOLID: Single Responsibility principle(SRP)

A class should have one, and only one, reason to change.
Robert C. Martin.



http://williamdurand.fr/from-stupid-to-solid-code-slides/#/4/4

Classe Student

```java
public class Student {

    private final String name;
    private final int section;

    // constructor
    public Student(String name, int section) {
        this.name = name;
        this.section = section;
    }
....}
```

```java
Student alice  = new Student("Alice",  2);
Student bob    = new Student("Bob",    1);
Student carol  = new Student("Carol",  2);
Student dave   = new Student("Dave",   1);
Student[] students = {dave, bob,alice};
for (int i = 0; i < students.length; i++)
       System.out.println(students[i]);
```

The single-responsibility principle

- **Example**:
  - Often we need to sort students by their name, or ssn.

www.cs.uofs.edu/~bi/2013f-html/se510/design-principles.ppt

Comparable…

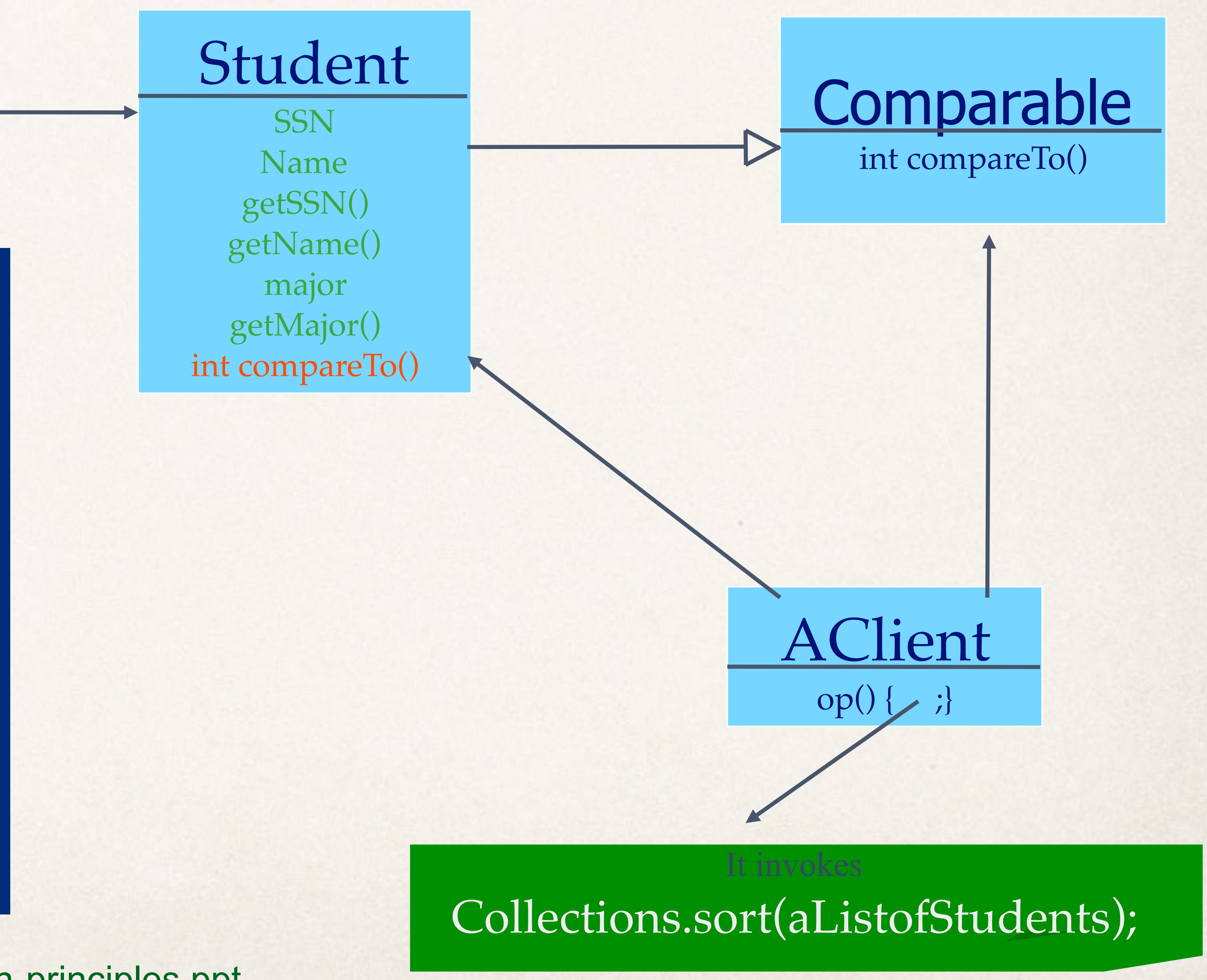
```java
public class Student implements Comparable<Student>{

    private static PrintStream StdOut = System.out;
    private final String name;
    private final int section;

    // constructor
    public Student(String name, int section) {
        this.name = name;
        this.section = section;
    }

    /*
    Compares this object with the specified object for order.
    Returns a negative integer, zero, or a positive integer
    as this object is less than, equal to,
    or greater than the specified object.
    */
    public int compareTo(Student oS) {
        return name.compareTo(oS.name);
    }
```

The single-responsibility principle



When a new requirement needs to sort students in a different order, Student, Register, and AClient all need to be recompiled, even Register has nothing to do with any ordering of Students.

www.cs.uofs.edu/~bi/2013f-html/se510/design-principles.ppt

The single-responsibility principle

- **Example**:
  - Often we need to sort students by their name, or ssn.

So one may make Class Student implement the Java Comparable interface.

```
class Student implements Comparable {
    int compareTo(Object o) { … }
};
```

- **BUT**:
  - Student is a business entity, it does not know in what order it should be sorted si the order of sorting is imposed by the client of Student.
  - Worse: every time students need to be ordered differently, we have to recompile Student and all its client.
  - Cause of the problems: we bundled two separate responsibilities (i.e., student as business entity with ordering) into one class – a violation of SRP

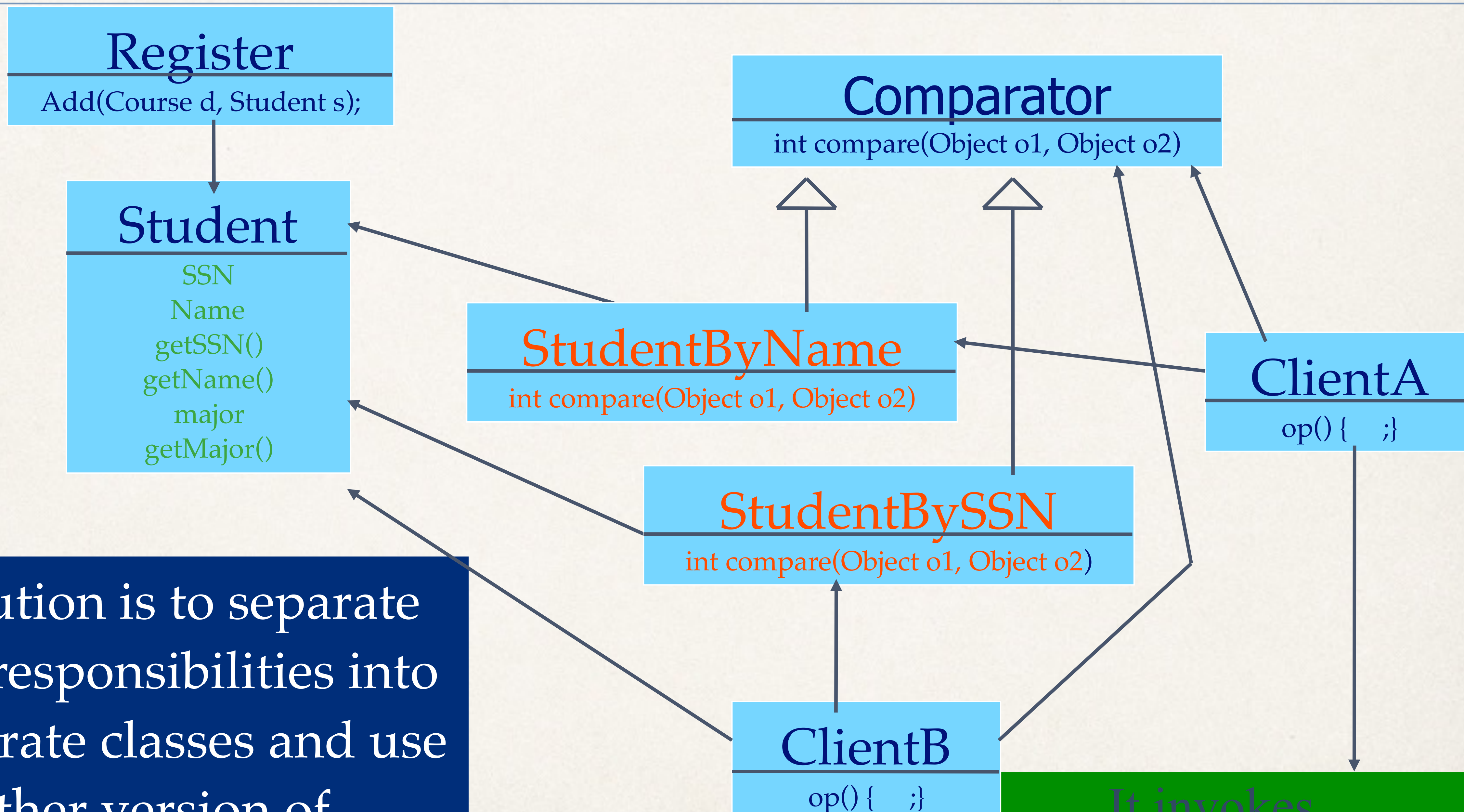
www.cs.uofs.edu/~bi/2013f-html/se510/design-principles.ppt

public interface Comparable<T>

int compareTo(T o)

- Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- The implementor must ensure : x.compareTo(y) == -y.compareTo(x) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)
- The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.
- Finally, the implementor must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.
- It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."
- In the foregoing description, the notation sgn(expression) designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

The single-responsibility principle



The solution is to separate the two responsibilities into two separate classes and use another version of Collections.sort().

It invokes
Collections.sort(aListofStudents, new StudentByName());

www.cs.uofs.edu/~bi/2013f-html/se510/design-principles.ppt

Les codes : Comparateurs

```
Interface Comparator<T>
Type Parameters:
T - the type of objects that may be compared by this comparator

int compare(T o1, T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

```java
class ByName implements Comparator<Student> {
        public int compare(Student a, Student b) {
            return a.name.compareTo(b.name);
        }
```

```java
Comparator<Student> byNameComparator =
   new ByName();
Comparator<Student> bySectionComparator=
   new BySection();
```

```java
class BySection implements Comparator<Student> {
        public int compare(Student a, Student b) {
            return a.section - b.section;
        }
```

Les codes : Comparer des étudiants

```
Student[] students = {
            larry, kevin, jen, isaac, grant, helia,
            frank, eve, dave, carol, bob, alice
        };

// sort by name and print results
Arrays.sort(students, byNameComparator);
for (int i = 0; i < students.length; i++)
            System.out.println(students[i]);

// now, sort by section and print results
Arrays.sort(students, bySectionComparator);
for (int i = 0; i < students.length; i++)
            System.out.println(students[i]);
```

SOLID: Liskov Substitution Principle (LSP)

Derived classes must be substitutable for their base classes.
Robert C. Martin.

http://williamdurand.fr/from-stupid-to-solid-code-slides/#/

Principe de substitution de Liskov
Liskov Substitution Principle (LSP)

Les **instances d'une classe**

doivent être **remplaçables** par des instances

de leurs **sous-classes** sans altérer le programme.

Principe de substitution de Liskov

- « Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T »
- Implications :
  - Le «contrat» défini par la classe de base (pour chacune de ses méthodes) doit être respecté par les classe dérivées
  - L'appelant n'a pas à connaître le type exact de la classe qu'il manipule : n'importe quelle classe dérivée peut être substituée à la classe qu'il utilise
- → Principe de base du polymorphisme :
  - Si on substitue une classe par une autre dérivée de la même hiérarchie: comportement (bien sûr) différent mais conforme.

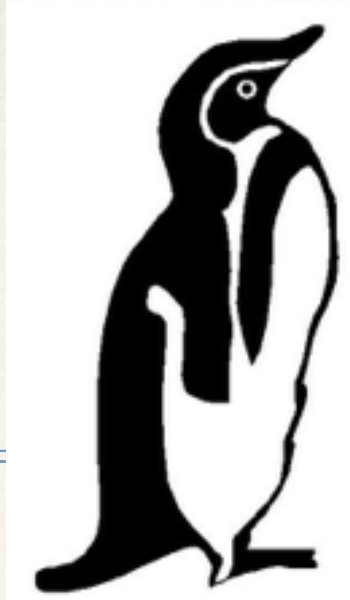
Inheritance *Appears* Simple

```
class Bird {                              // has beak, wings,...
  public: virtual void fly();   // Bird can fly
};


class Parrot : public Bird {      // Parrot is a bird
  public: virtual void mimic();   // Can Repeat words...
};

// ...
Parrot mypet;
mypet.mimic();    // my pet being a parrot can Mimic()
mypet.fly();      // my pet "is-a" bird, can fly
```

Penguins Fail to Fly!

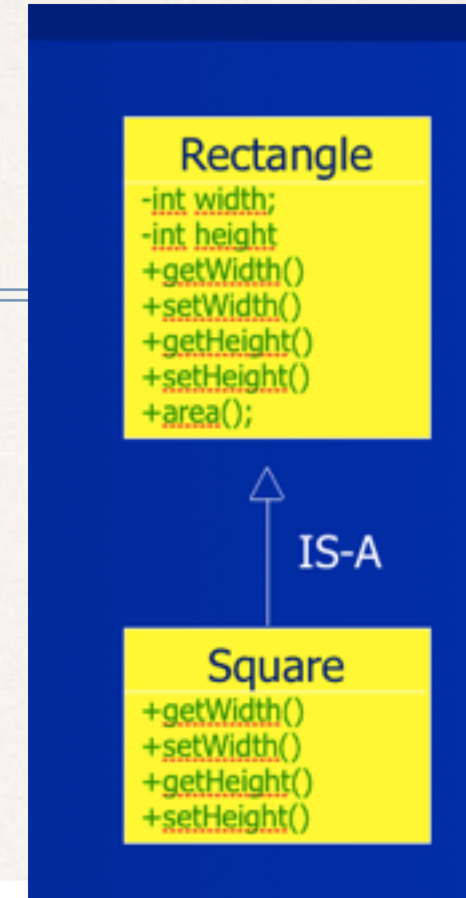


```
class Penguin : public Bird {
   public: void fly() {
      error ("Penguins don't fly!"); }
};
```

- Does not model: *"Penguins can't fly"*
- It models *"Penguins may fly, but if they try it is error"*
- Run-time error if attempt to fly → not desirable
- ***Think about Substitutability - Fails LSP***

```
void PlayWithBird (Bird& abird) {
     abird.fly();     // OK if Parrot.
     // if bird happens to be Penguin...OOOPS!!
}
```

Liskov Substitution Principe : contre-exemple




```
class Rectangle
  {
     int m_width;
     int m_height;
     public void setWidth(int width)
  {
         m_width = width;
     }
     public void setHeight(int h){
         m_height = ht;
     }
     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){
        super.setWidth(width);
        super.setHeight(width);
    }
public void setHeight(int height){
        super.setWidth(height);
        super.setHeight(height);
    }
  }
```

Liskov Substitution Principle

```
class LspTest
 {
  private static Rectangle getNewRectangle(){
        // it can be an object returned by some factory ...
        if ….
               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 and 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.
 }


}
```

LSP Related Heuristic

> It is illegal for a derived class, to override a base-class method with a NOP method

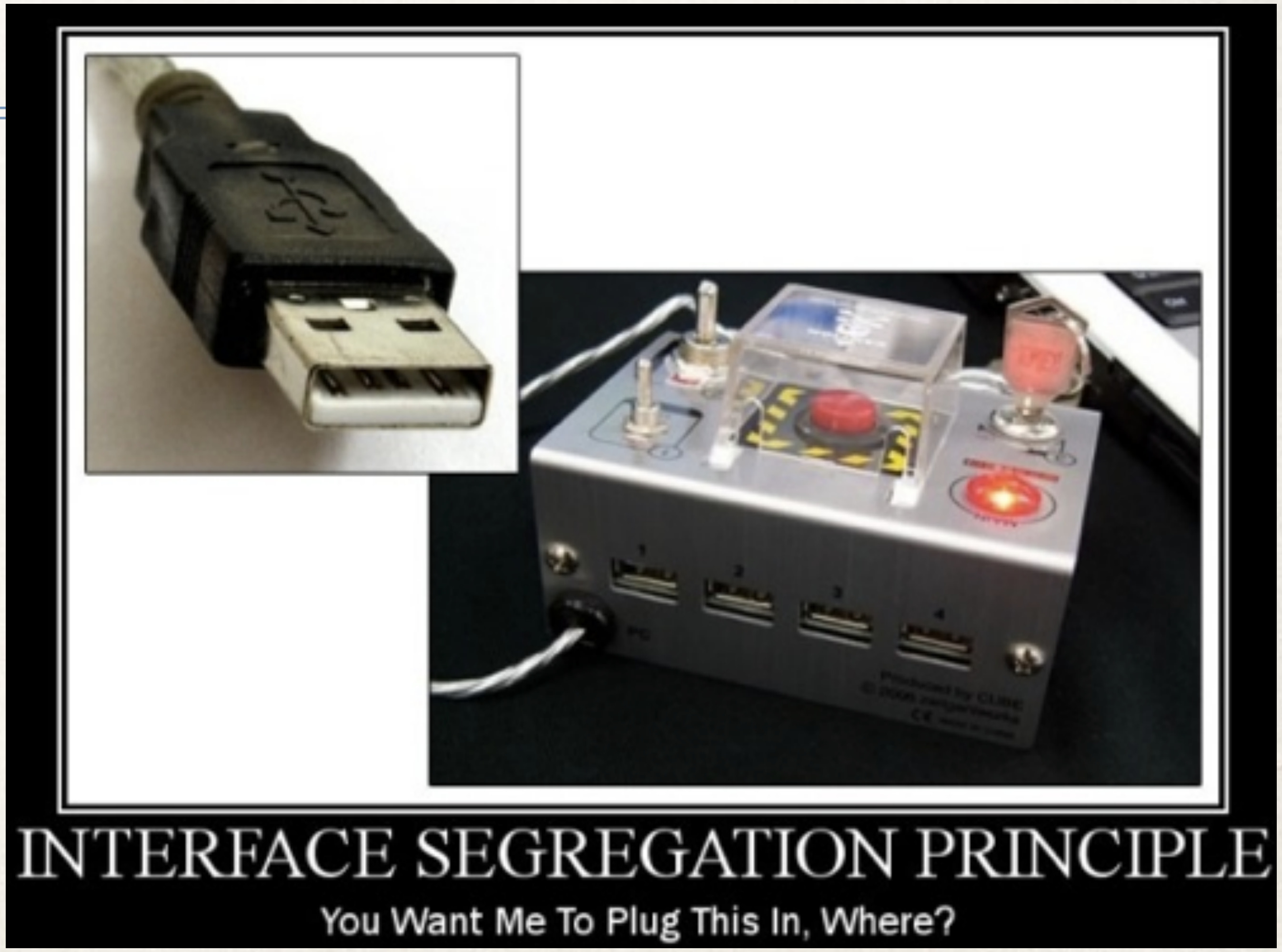
- NOP = a method that does nothing
- Solution 1: Inverse Inheritance Relation
  - if the initial base-class has only additional behavior
    - e.g. **`Dog - DogNoWag`**
- Solution 2: Extract Common Base-Class
  - if both initial and derived classes have different behaviors
  - for **`Penguins`** → **`Birds`**, **`FlyingBirds`**, **`Penguins`**

"Clients should not be forced to depend upon interfaces that they do not use."
— Robert Martin, ISP paper linked from The Principles of OOD

SOLID: Interface Segregation Principle (ISP)

Make fine grained interfaces that are client specific.
Robert C. Martin.

«Still, a man hears
What he wants to hear
And disregards the rest
La la la… »
Simon and Garfunkel, "The Boxer"



http://williamdurand.fr/from-stupid-to-solid-code-slides/#/

Program To An Interface, Not An Implementation

- An *interface* is the set of methods one object knows it can invoke on another object
- A class can implement many interfaces. (Essentially, an interface is a subset of all the methods that a class implements)
- A *type* is a specific interface of an object
- Different objects can have the same type and the same object can have many different types.
- An object is known by other objects only through its interface.
- Interfaces are the key to pluggability

Interface Example

```
/**
* Interface IManeuverable provides the specification
* for a maneuverable vehicle.
*/

public interface IManeuverable {
    public void left();
    public void right();
    public void forward();
    public void reverse();
    public void climb();
    public void dive();
    public void setSpeed(double speed);
    public double getSpeed();
}
```

Interface Example (Continued)

```
public class Car implements IManeuverable {
    // Code here.
}

public class Boat implements IManeuverable {
    // Code here.
}

public class Submarine implements IManeuverable {
    // Code here.
}
```

Interface Example (Continued)

- This method in some other class can maneuver the vehicle without being concerned about what the actual class is (car, boat, submarine) or what inheritance hierarchy it is in

```
public void travel(IManeuverable vehicle) {
    vehicle.setSpeed(35.0);
    vehicle.forward();
    vehicle.left();
    vehicle.climb();
}
```

Interface segregation principle

- Plusieurs «client-specific interfaces» sont mieux qu'une interface générale.
- Un client doit avoir des interfaces avec uniquement ce dont il a besoin
  - Incite à ne pas faire "extract interface" sans réfléchir
  - Incite à avoir des interfaces petites pour ne pas forcer des classes à implémenter les méthodes qu'elles ne veulent pas.
  - Peut amener à une multiplication excessive du nombre d'interfaces
    - à l'extrême : une interface avec une méthode (Penser à la cohésion...)
  - Utiliser l'expérience, le pragmatisme et le bon sens !

http://igm.univ-mlv.fr/ens/Master/M1/2013-2014/POO-DP/cours/1c-POO-x4.pdf

ISP Example: Timed door

```
class Door
{
  public:
  virtual void Lock() = 0;
  virtual void Unlock() = 0;
  virtual bool IsDoorOpen() = 0;
};
```

TimedDoor needs to sound an alarm when the door has been left open for too long. To do this, the TimedDoor object communicates with another object called a Timer.

ISP Example: Timed door

```
class Timer
{
  public:
  void Register(int timeout, TimerClient* client);
};
```

time of timeout

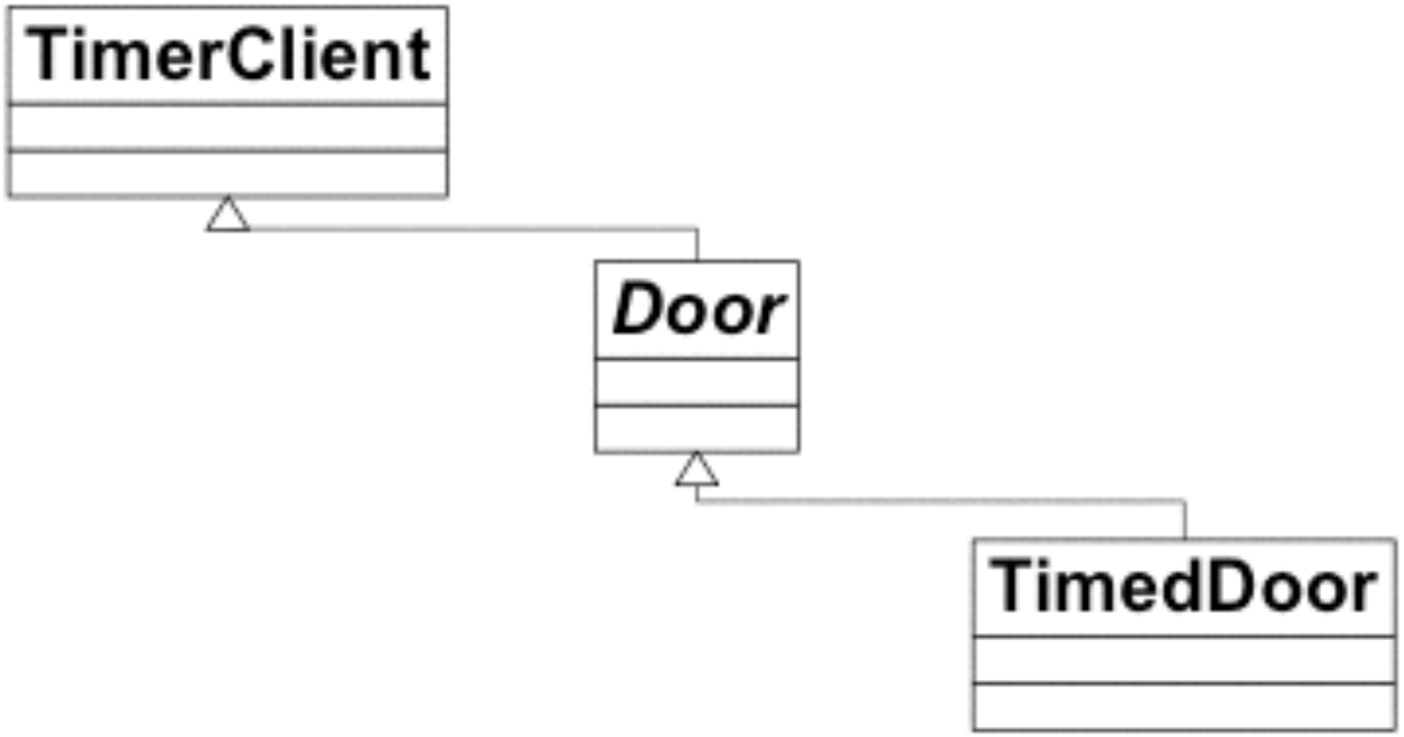
object to invoke TimeOut() on when timeout occurs

```
class TimerClient
{
  public:
  virtual void TimeOut() = 0;
};
```

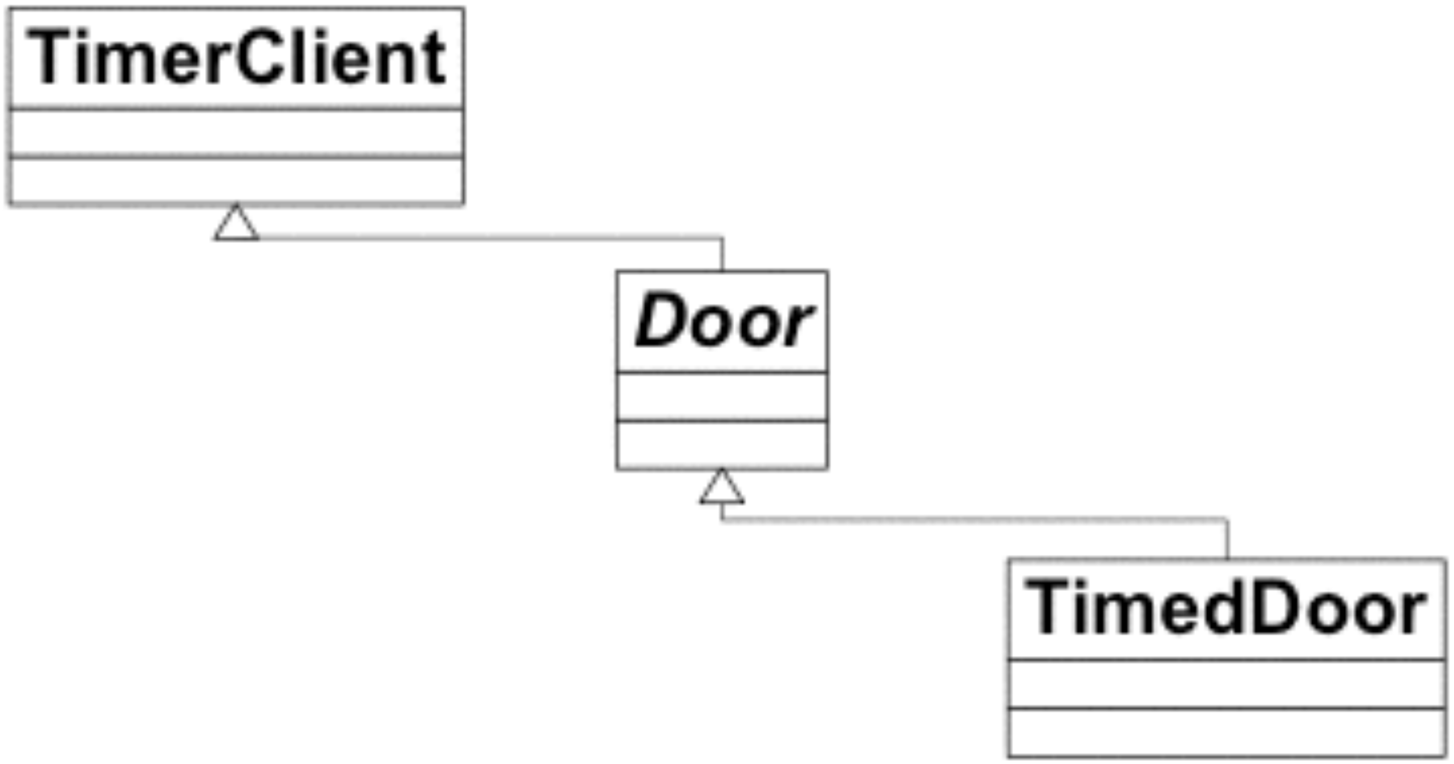
TimeOut method

How should we connect the TimerClient to a new TimedDoor class so it can be notified on a timeout?

InterfaceSegregationPrinciple Solution: yes or no?

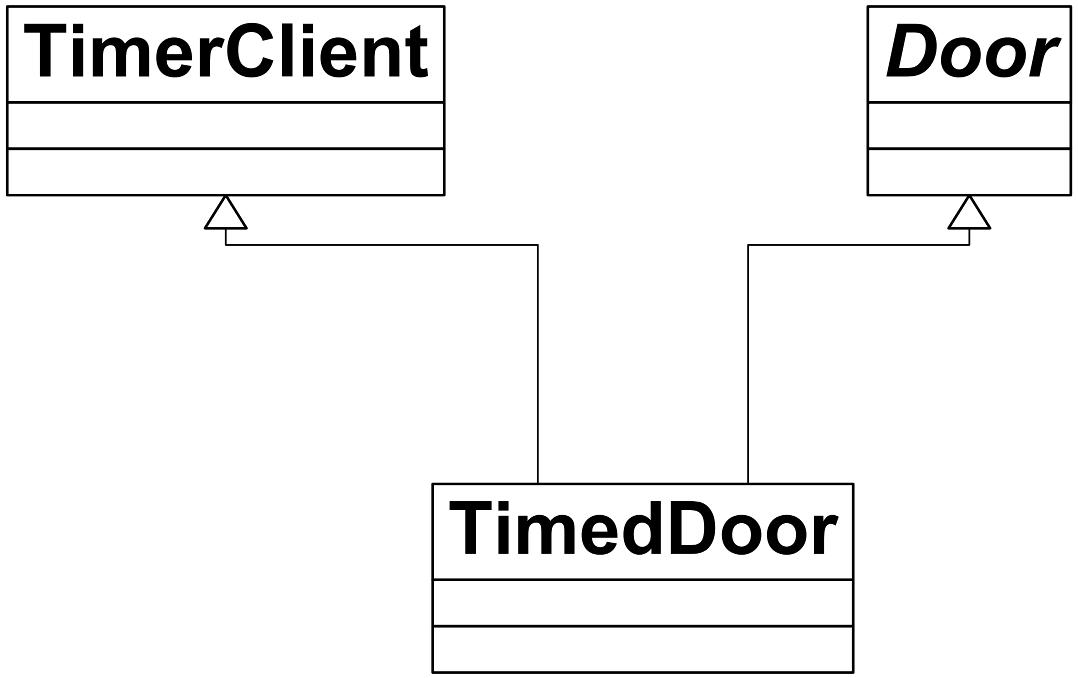


InterfaceSegregationPrinciple Solution: yes or no?

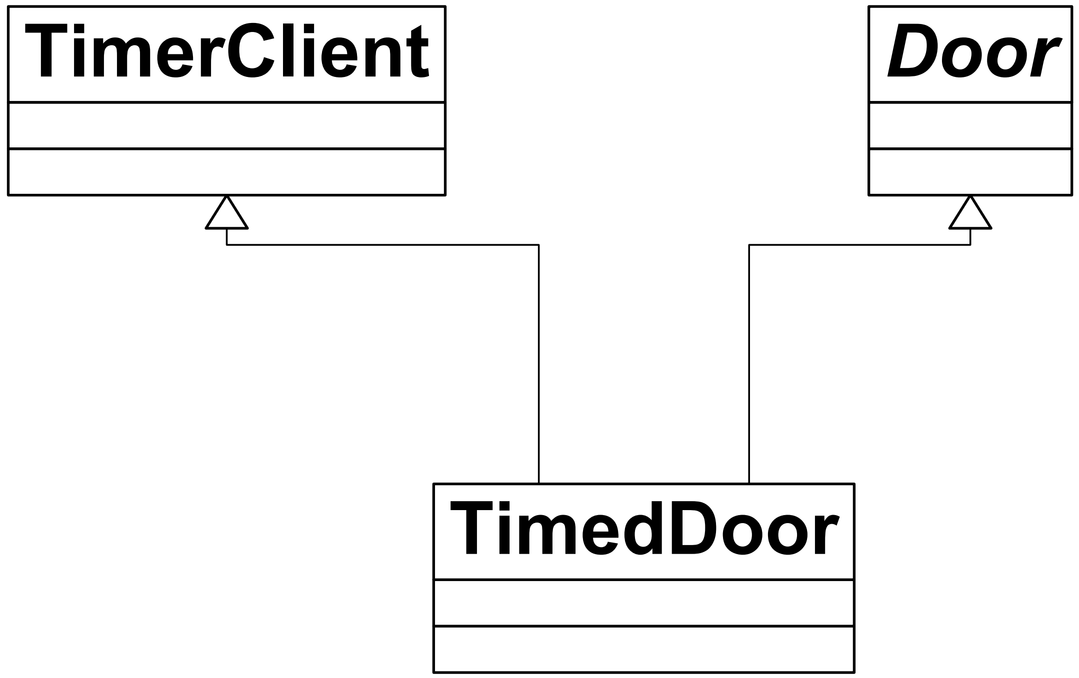


No, as it’s polluting the Door interface by requiring all doors to have a TimeOut() method

ISP Solution: yes or no?



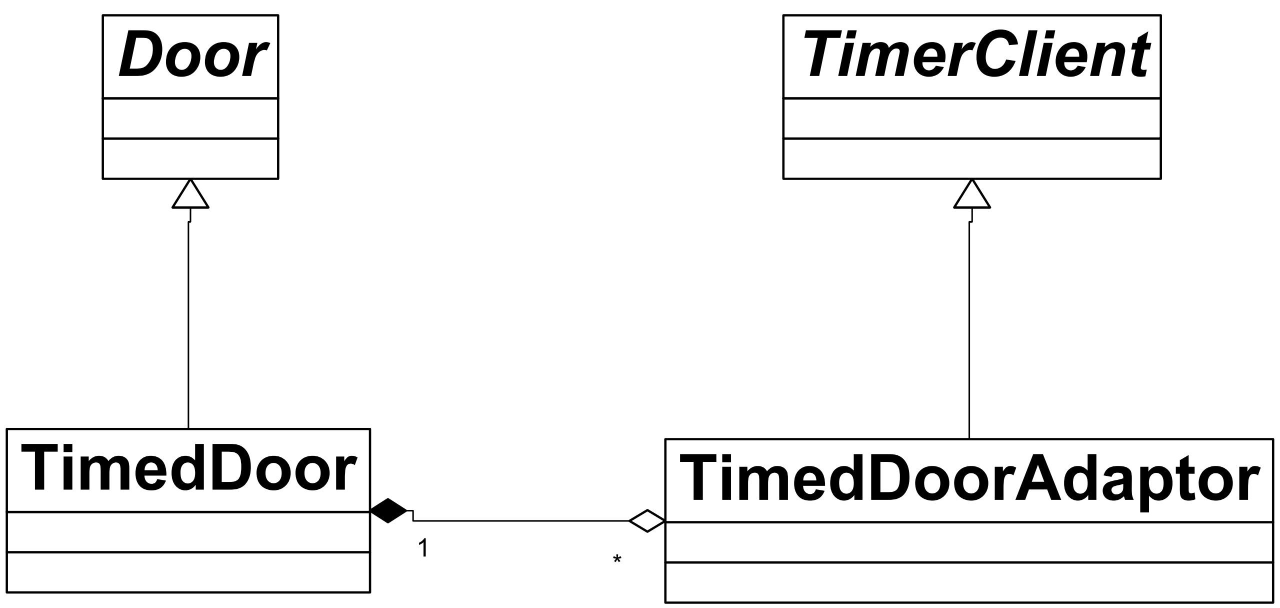
ISP Solution: yes or no?



Yes, separation through multiple inheritance

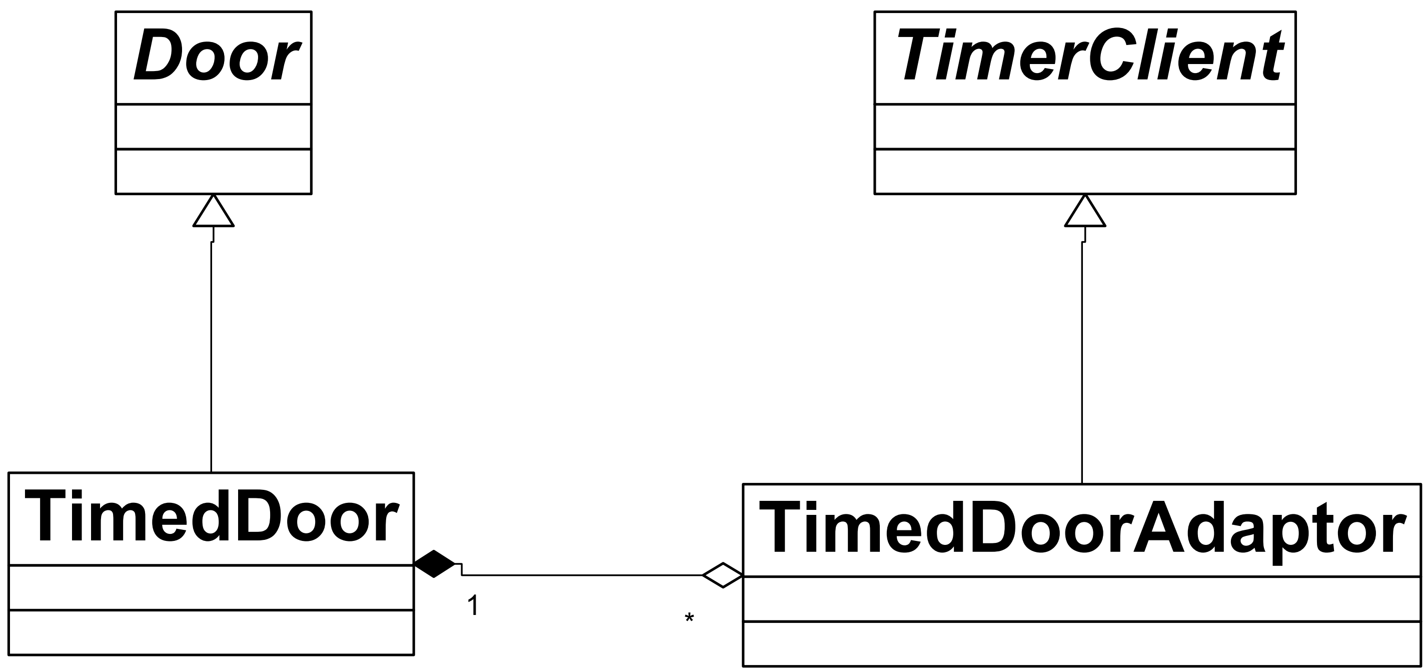
ISP solution: yes or no?

When the <u>Timer</u> sends the <u>TimeOut</u> message to the <u>TimedDoorAdapter</u>, the <u>TimedDoorAdapter</u> delegates the message back to the <u>TimedDoor</u>.



ISP solution: yes or no?

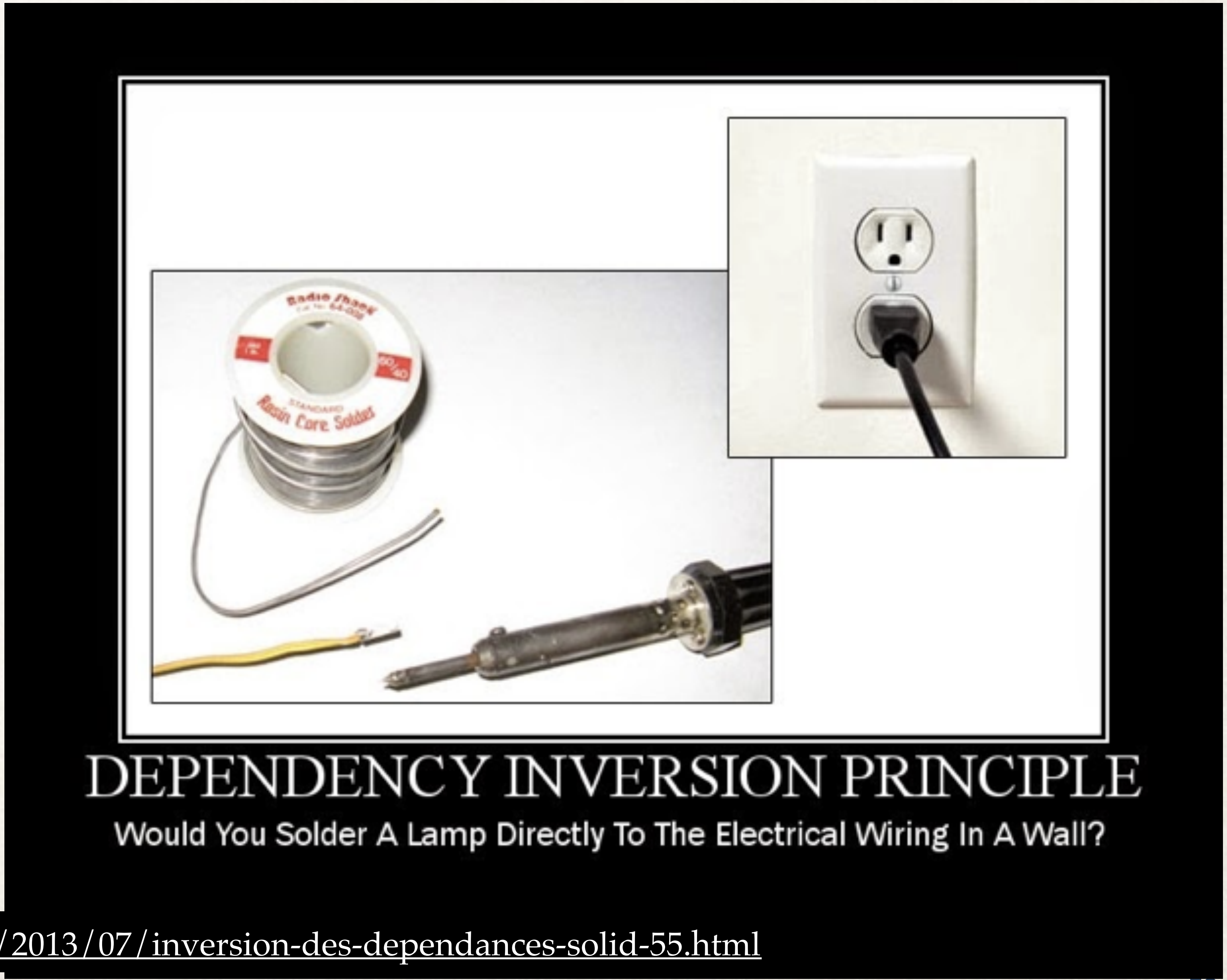
When the Timer sends the TimeOut message to the DoorTimerAdapter, the DoorTimerAdapter delegates the message back to the TimedDoor.



Yes, separation through delegation

Principe inversion de dépendance Dependency Inversion Principle (DIP)

Depend on abstractions,
not on concretions.
Robert C. Martin.



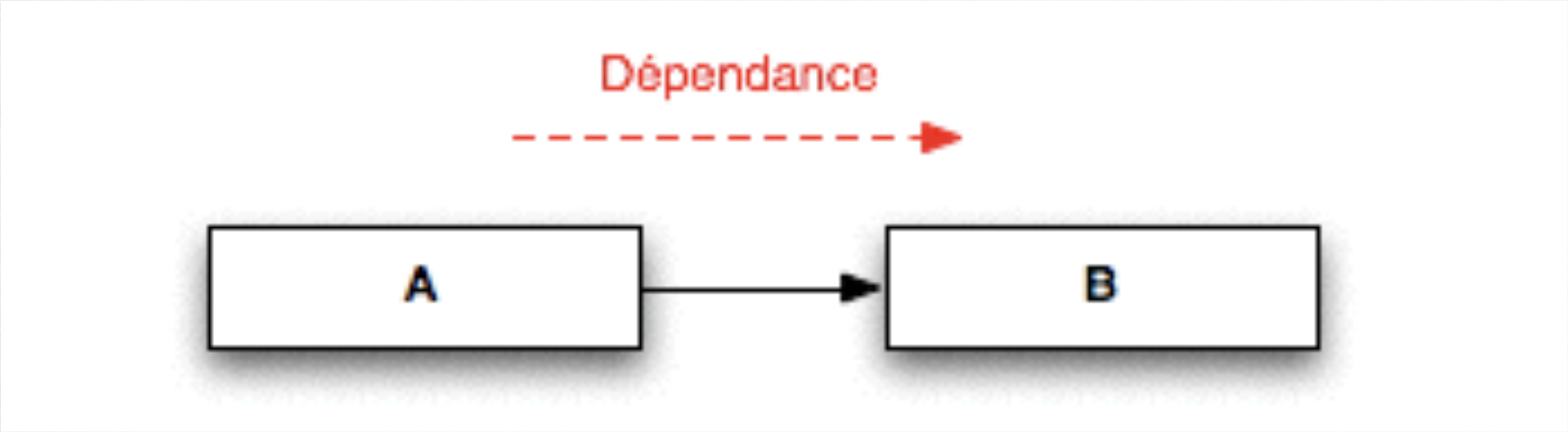
A LIRE !! http://cyrilgandon.blogspot.fr/2013/07/inversion-des-dependances-solid-55.html

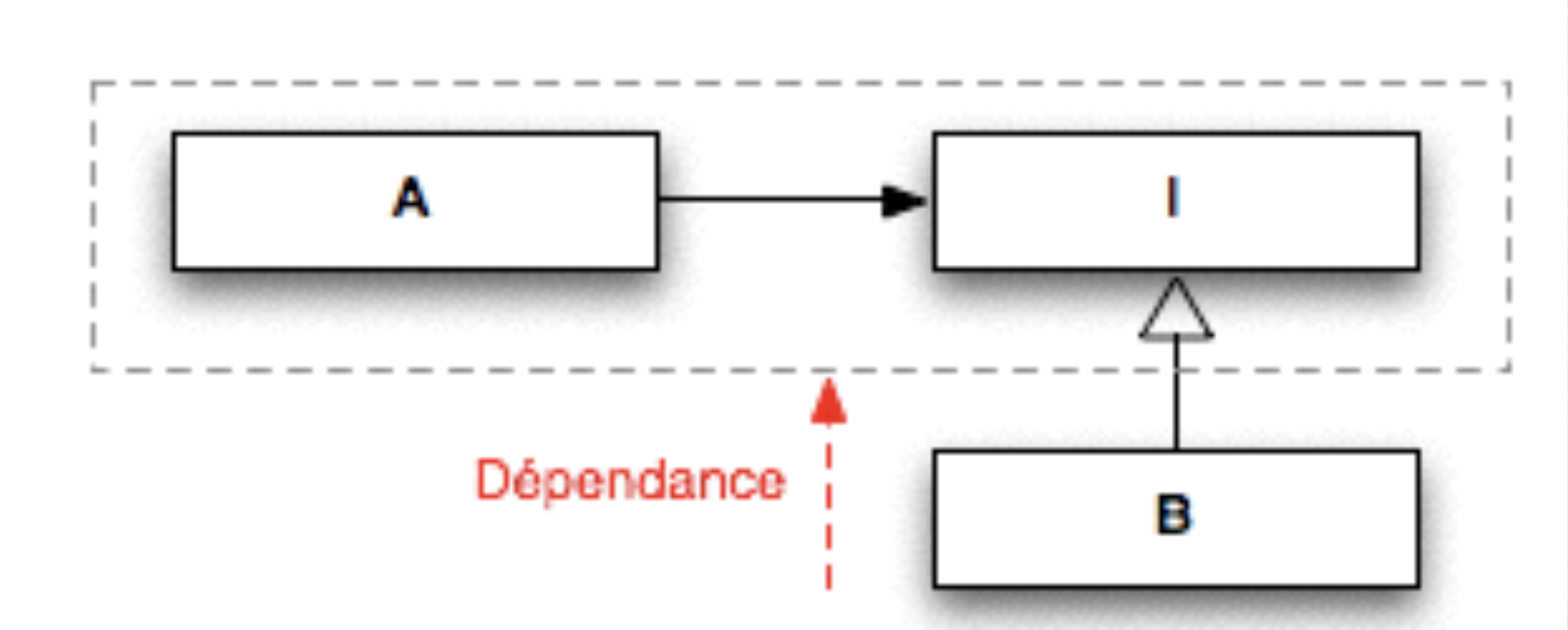
Inversion de dépendance

- Réduire les dépendances sur les classes concrètes
- «Program to interface, not implementation »
- Les abstractions ne doivent pas dépendre de détails.
  - → Les détails doivent dépendre d'abstractions.
- Ne dépendre QUE des abstractions, y compris pour les classes de bas niveau
- Permet OCP (principe) quand l'inversion de dépendance c'est la technique!

d'après http://igm.univ-mlv.fr/ens/Master/M1/2013-2014/POO-DP/cours/1c-POO-x4.pdf

Inversion de dépendance





Prenons un exemple !

```
public void drinkBeer(FavoriteBar bar)
{
    Beer beer = bar.orderBeer();
    this.raiseElbow();
    this.drink(beer);
}
```

http://cyrilgandon.blogspot.fr/2013/07/inversion-des-dependances-solid-55.html

Abstraction

```
interface IBar

{

    Beer OrderBeer();

}

class FavoriteBar implements IBar
        public Beer OrderBeer() {
                return new Beer("demi");
            }
```

Abstraction

```
public void drinkBeer(IBar bar)

{

    Beer beer = bar.orderBeer();

    this.raiseElbow();

    this.drink(beer);

}
```

Généralisation

```
Interface IOrderableBeverage

{

    Beverage order(BeverageType type);

}

public interface IBar extends IOrderableBeverage {

    Beer orderBeer();
}

class FavoriteBar implements IBar
```

Généralisation

Interface IOrderableBeverage

{

Beverage order(BeverageType type);

}

```
public class SmoothyBar implements IOrderableBeverage {

    public Beverage order(BeverageType type) {
        if (type.equals(BeverageType.ORANGEJUICE))
              return new OrangeJuice();
    }
}
```

Généralisation

```
public void drinkBeer(IOrderableBeverage bar)
{
    this.drink(bar, BeverageType.Beer);
}
public void drink(IOrderableBeverage bar, BeverageType type)
{
    Beverage beverage = bar.order(type);
    this.raiseElbow();
    this.drink(beverage);
}
```

Généralisation

```
public class Drinker {

    public void drinkBeer(IBar bar) {
        System.out.println("Ready for a 
beer");
        Beer beer = bar.orderBeer();
        this.raiseElbow();
        this.drink(beer);
    }

    private void drink(Beer beer) {
        System.out.println("je bois " + 
beer);
    }


    private void raiseElbow() {
        System.out.println("je leve le 
coude!");
    }
```

```
    private void drinkSome(IOrderableBeverage 
bar, BeverageType type) {
        System.out.println("No thanks, Only 
Beer for me");
        Beer bear2 = (Beer) 
bar.order(BeverageType.BEER);
        this.raiseElbow();
        this.drink(bear2);
    }
    public static void main (String [] args){
        Drinker dd = new Drinker();
        FavoriteBar myBar = new 
FavoriteBar();
        System.out.println("A beer ?");
        dd.drinkBeer(myBar);
        System.out.println("orange 
juice ?");
        dd.drinkSome(myBar, 
BeverageType.ORANGEJUICE);

    }
```

Et encore …

```
public interface IOrderableSnack {
    Snack order(SnackType type);
}
```

Software design principles- summary

- The single-responsibility principle
  - There is only one source that may the class to change
- The open-closed principle
  - Open to extension, closed for modification
- The Liskov substitution principle
  - A subclass must substitutable for its base class
- The dependency inversion principle
  - Low-level (implementation, utility) classes should be dependent on high-level (conceptual, policy) classes
- The interface segregation principle
  - A client should not be forced to depend on methods it does not use.

www.cs.uofs.edu/~bi/2013f-html/se510/design-principles.ppt

Autres éléments de bibliographie

- Coupling and Cohesion, Pfleeger, S., Software Engineering Theory and Practice. Prentice Hall, 2001.
- http://igm.univ-mlv.fr/ens/Master/M1/2013-2014/POO-DP/cours/1c-POO-x4.pdf