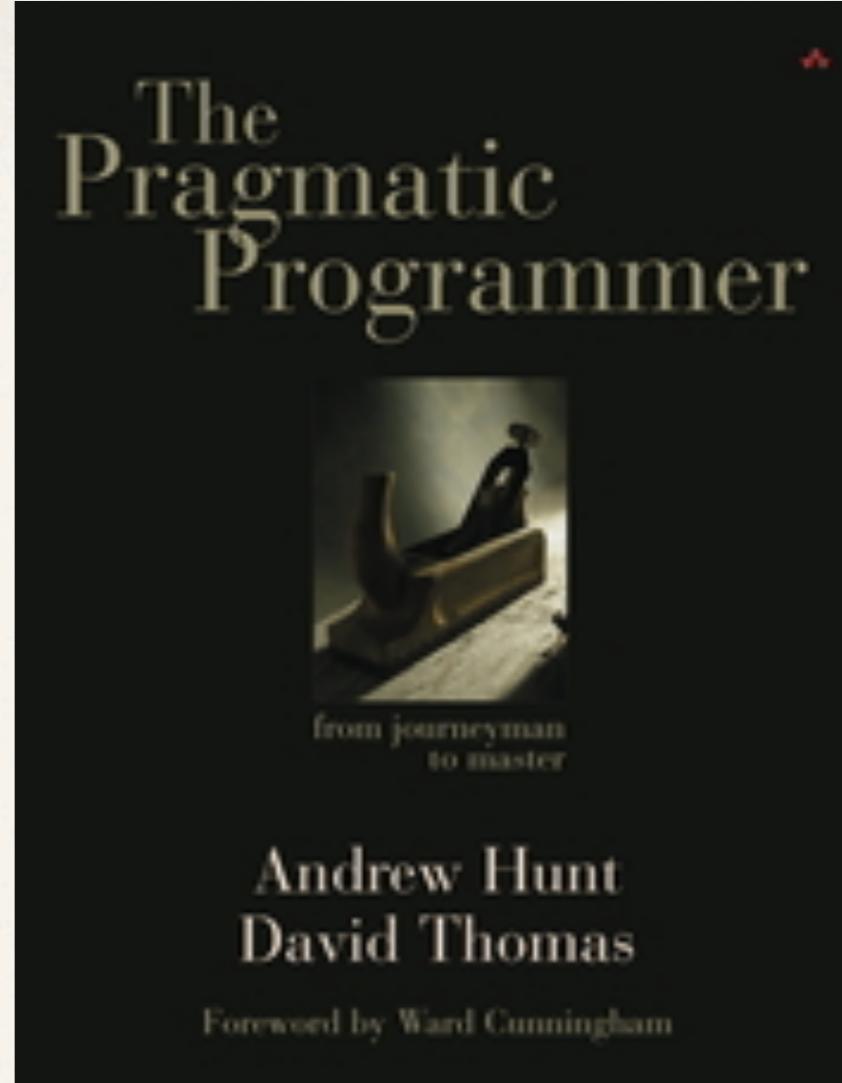


Principes SOLID suivi de

Pragmatic Programming

The Pragmatic Programmer: From Journeyman to Master

by Andrew Hunt and David Thomas



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
é Dulcorée (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.

Mireille Blay-Fornarino, Université Nice Sophia Antipolis, Département Info IUT, Octobre 2015



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 substitution 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.



Open Closed Principle

You don't need to rewire your MoBo to plug in "Mr Happy"

<http://deviq.com/single-responsibility-principle> 7

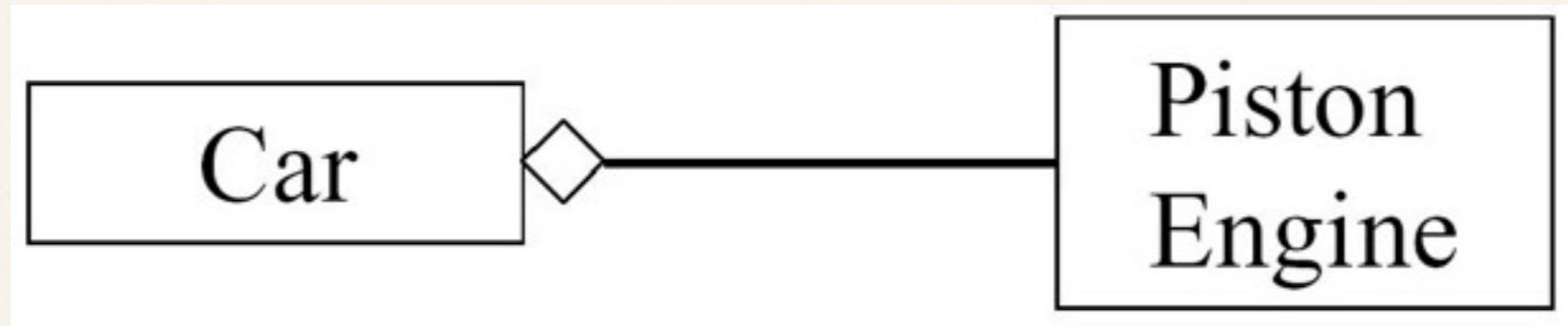
Principe ouvert / fermé Open/Closed Principle (OCP)

You should be able to extend a classes behavior, without modifying it.

Robert C. Martin.

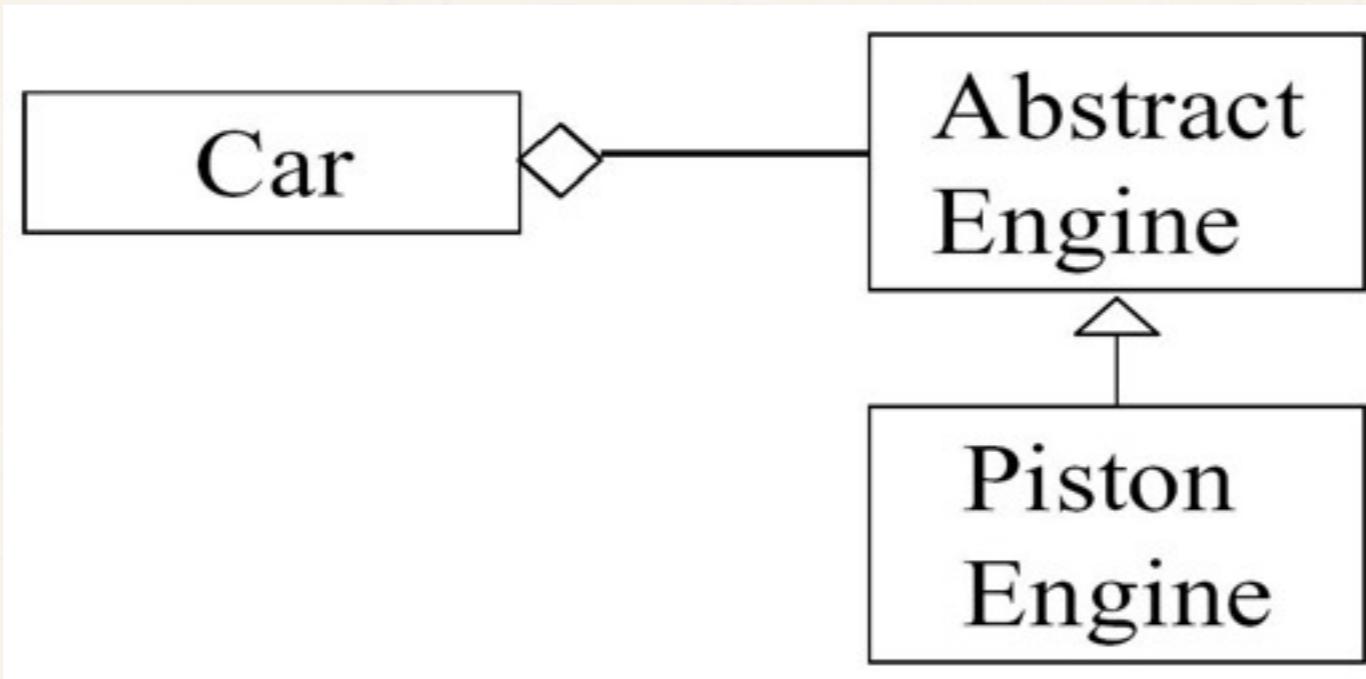
- ✿ Les entités logicielles doivent être ouvertes à l'extension
 - le code est extensible
- ✿ mais fermées aux modifications
 - Le code a été écrit et testé, on n'y touche pas.

Open the door ...



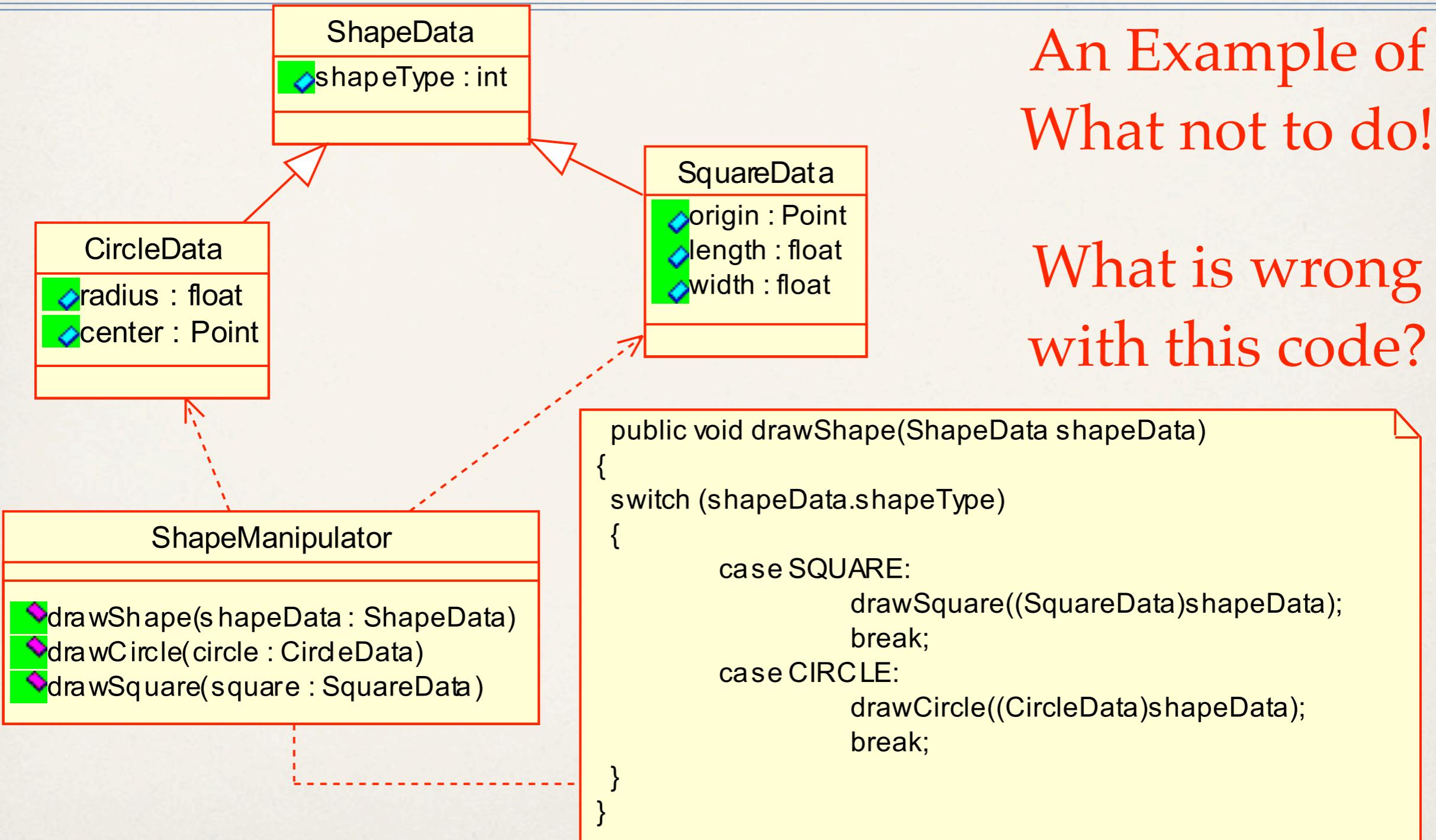
- * Comment faire en sorte que la voiture aille plus vite à l'aide d'un turbo?
 - Il faut changer la voiture
 - avec la conception actuelle...

... But Keep It Closed!

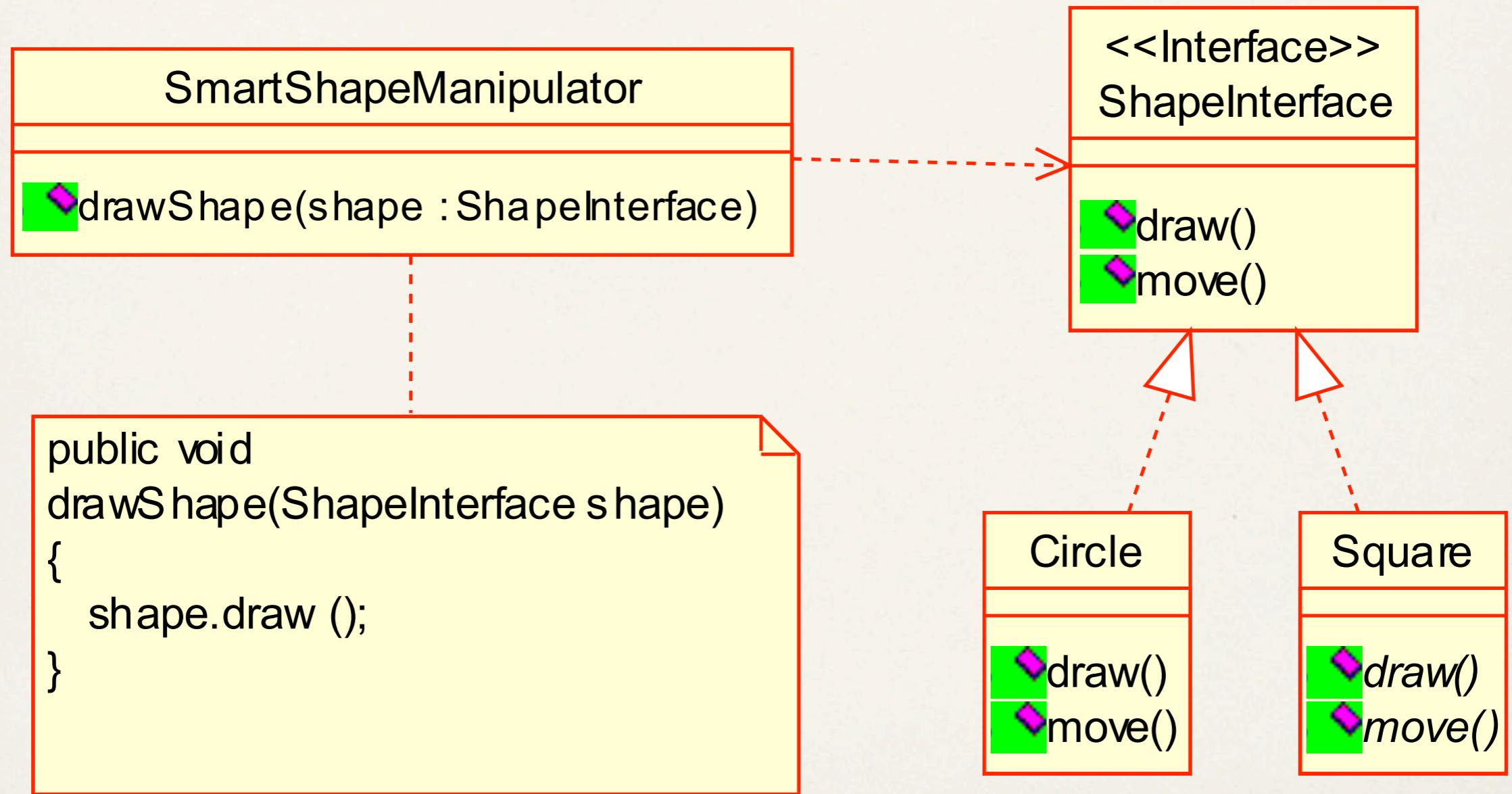


- ✿ On retient :
 - Une classe **ne doit pas dépendre d'une classe Concrète**.
 - Elle peut dépendre d'une classe abstraite ...
 - et utiliser le polymorphisme

The Open/Closed Principle (OCP) Example



The Open/Closed Principle (OCP) Example



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

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»

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) {
```

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;
```

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++) {
```

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();
```

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();  
  
    }  
}
```

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 (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 (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 ...

On pourrait aussi faire quoi, pour encore décroître le couplage?

The Open-Closed Principle(OCP) : allows 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>

20

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) { ... }  
};
```

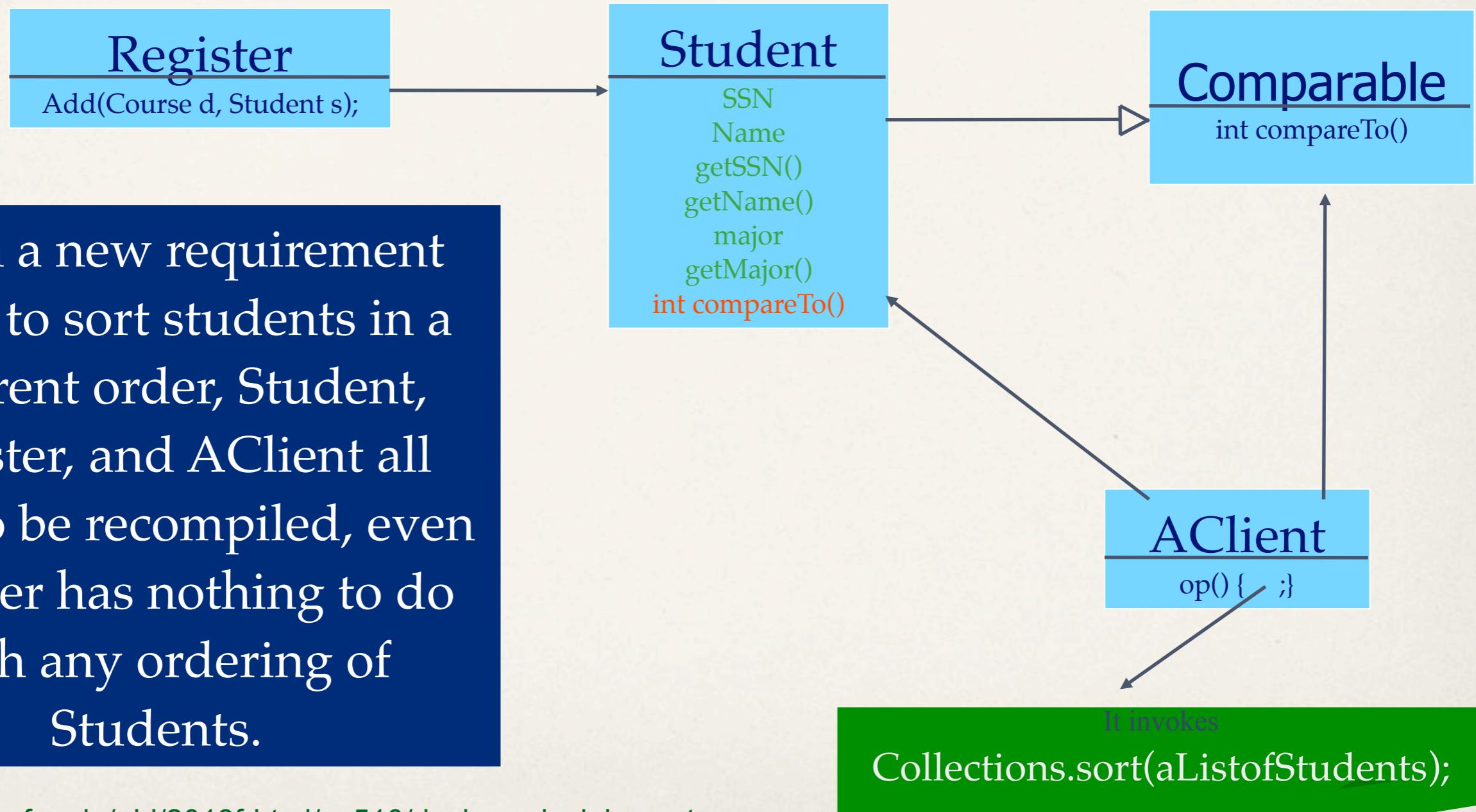
- * **BUT:**

- Student is a business entity, it does not know in what order it should be sorted since 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 a business entity with ordering) into one class – a violation of SRP

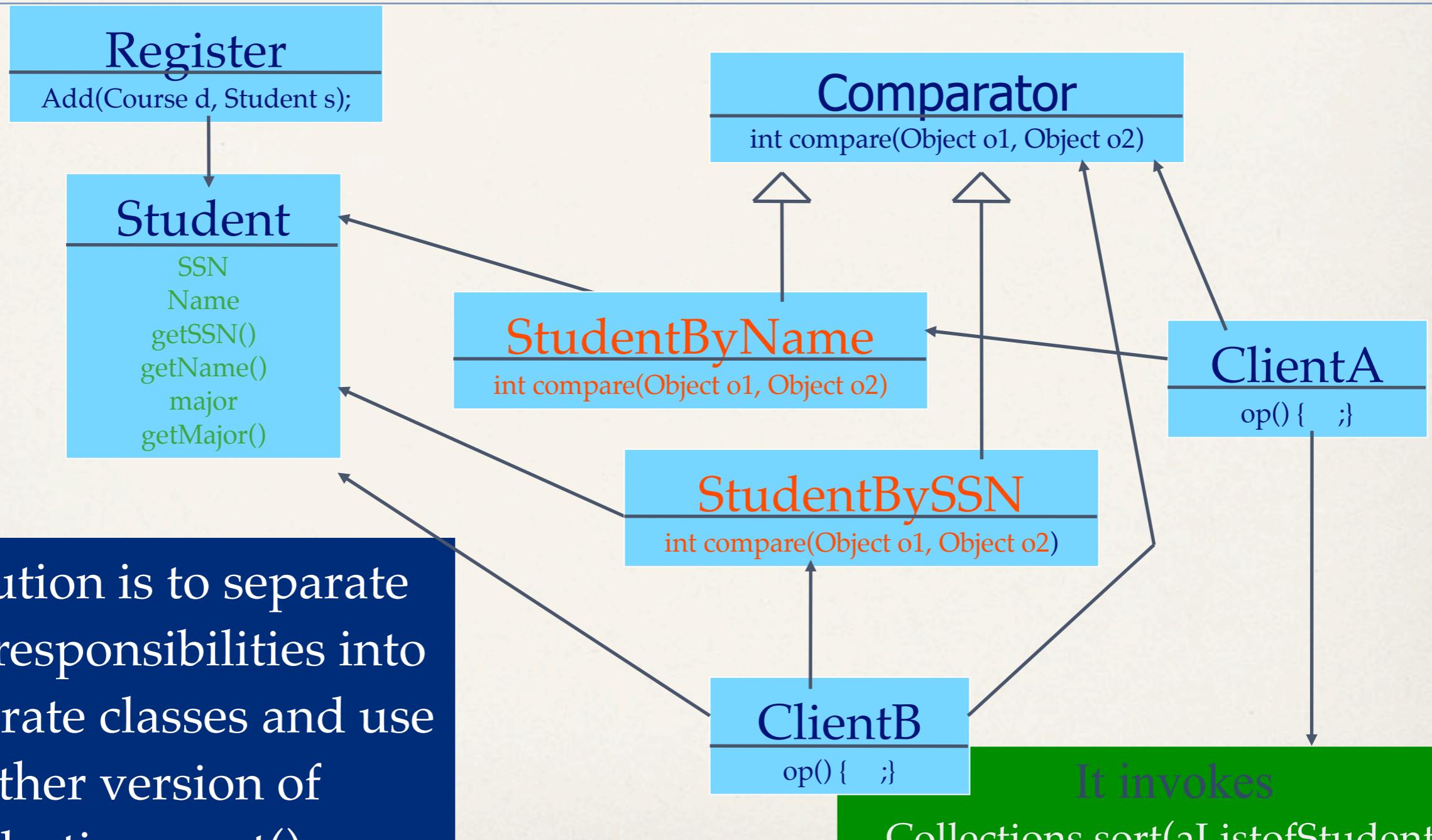
public interface Comparable<T>

- * 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 $\text{sgn}(x.compareTo(z)) == \text{sgn}(y.compareTo(z))$, 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 $\text{sgn}(\text{expression})$ 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 single-responsibility principle



Les codes : Classe Student

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

```
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]);
```

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.

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

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

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

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.

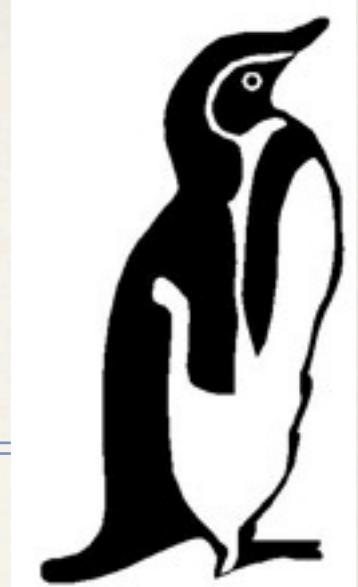
Inheritance *Appears* Simple

```
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!"); }  
};
```

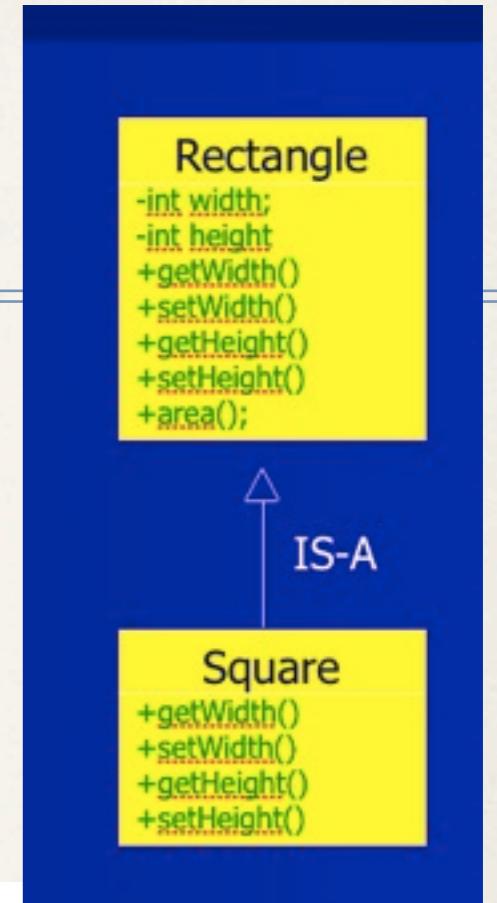
- Does not model: “*Penguins can't fly*”
- It models “*Penguins may fly, but if they try it is error*”
- Run-time error if attempt to fly → not desirable
- ***Think about Substitutability - Fails LSP***

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

Liskov Substitution Principle : 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 ...
        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"



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

<http://williamdurand.fr/from-stupid-to-solid-code-slides/#/36>

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 !

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);
};

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

time of timeout

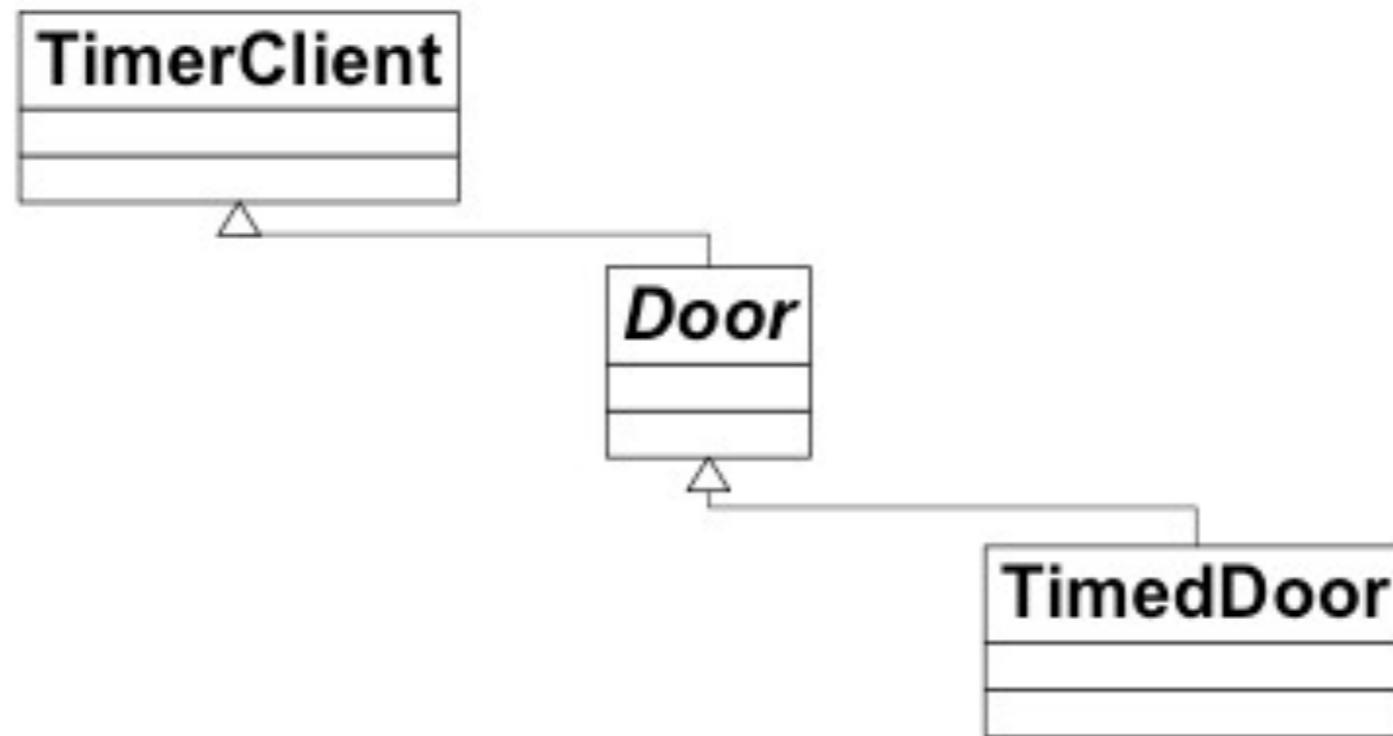
object to invoke TimeOut() on
when timeout occurs

TimeOut method

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

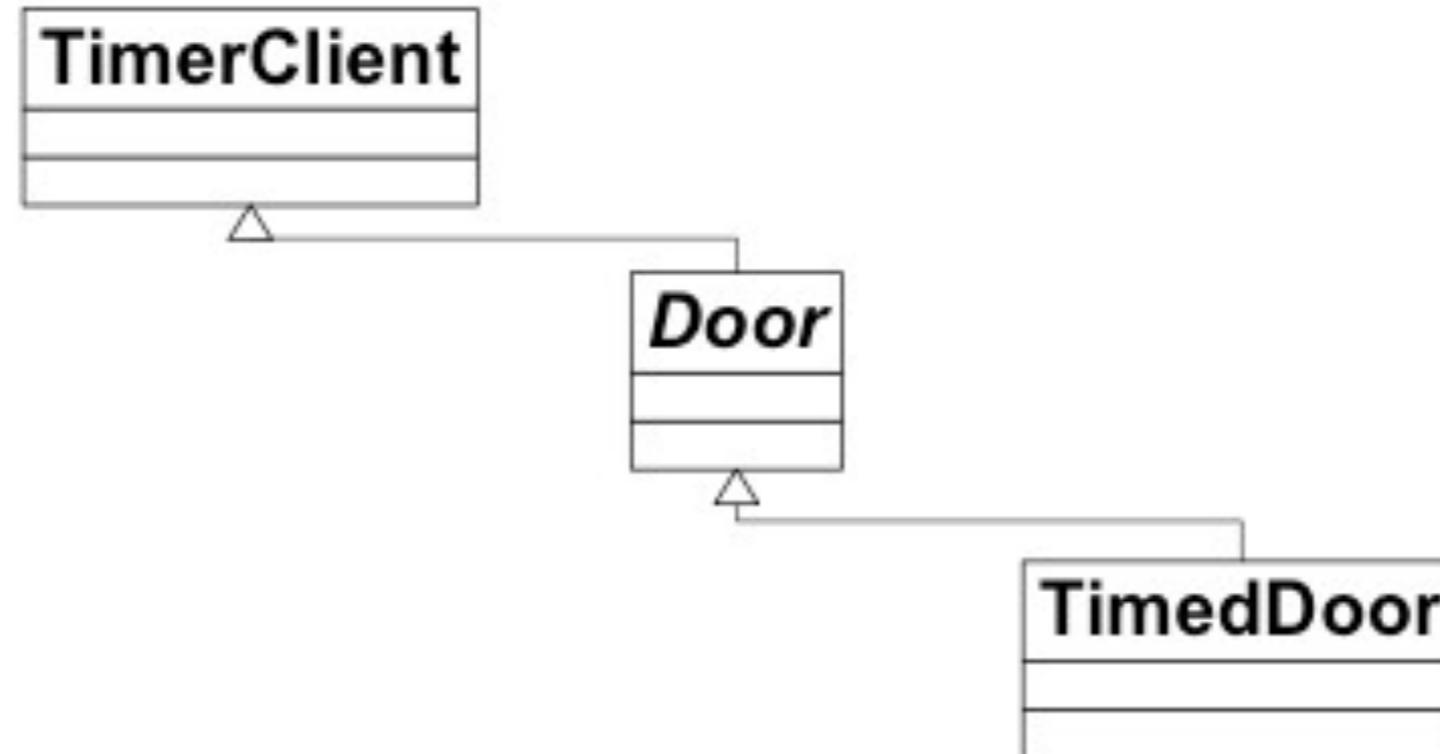
Interface Segregation Principle

Solution: yes or no?



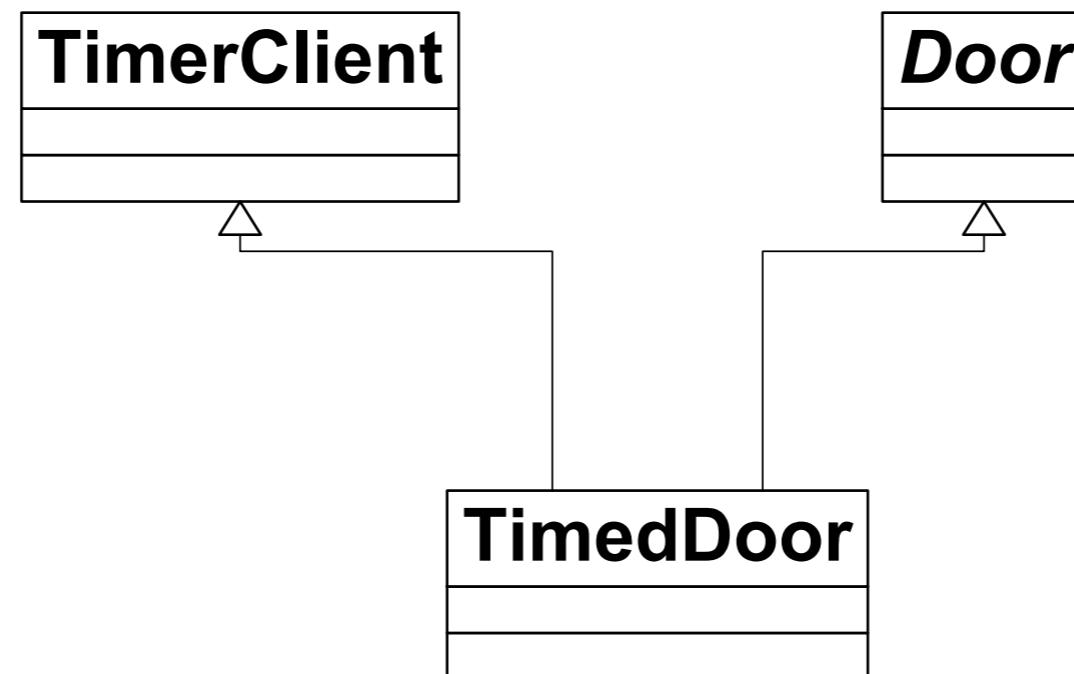
Interface Segregation Principle

Solution: yes or no?

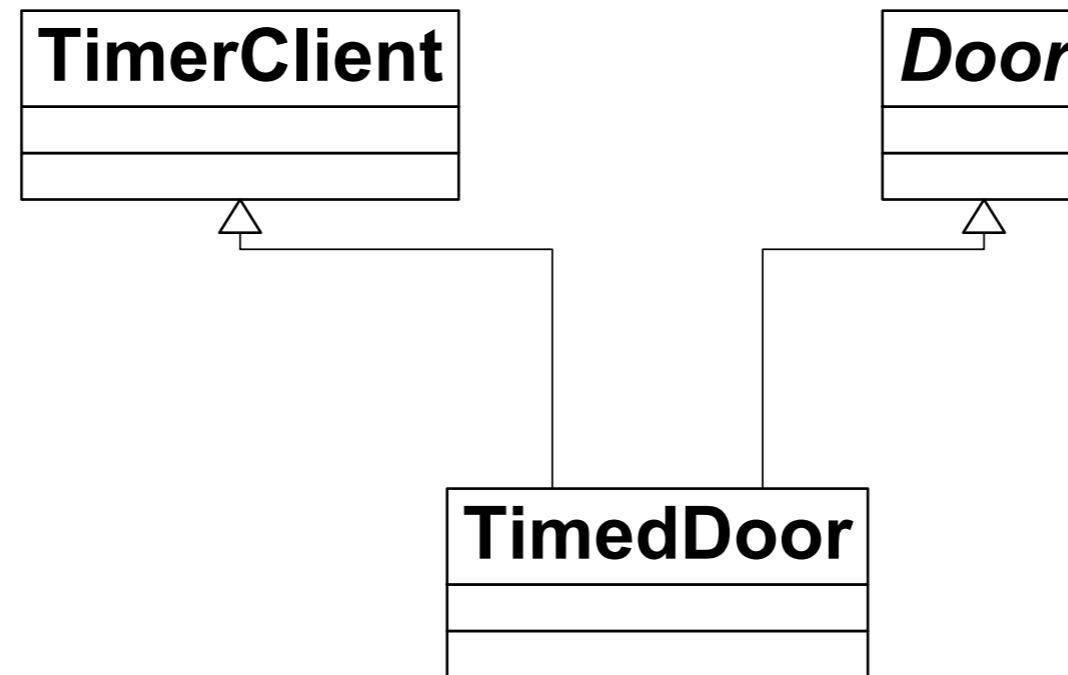


No, as it's polluting the Door interface by requiring all doors to have a TimeOut() method

ISP Solution: yes or no?



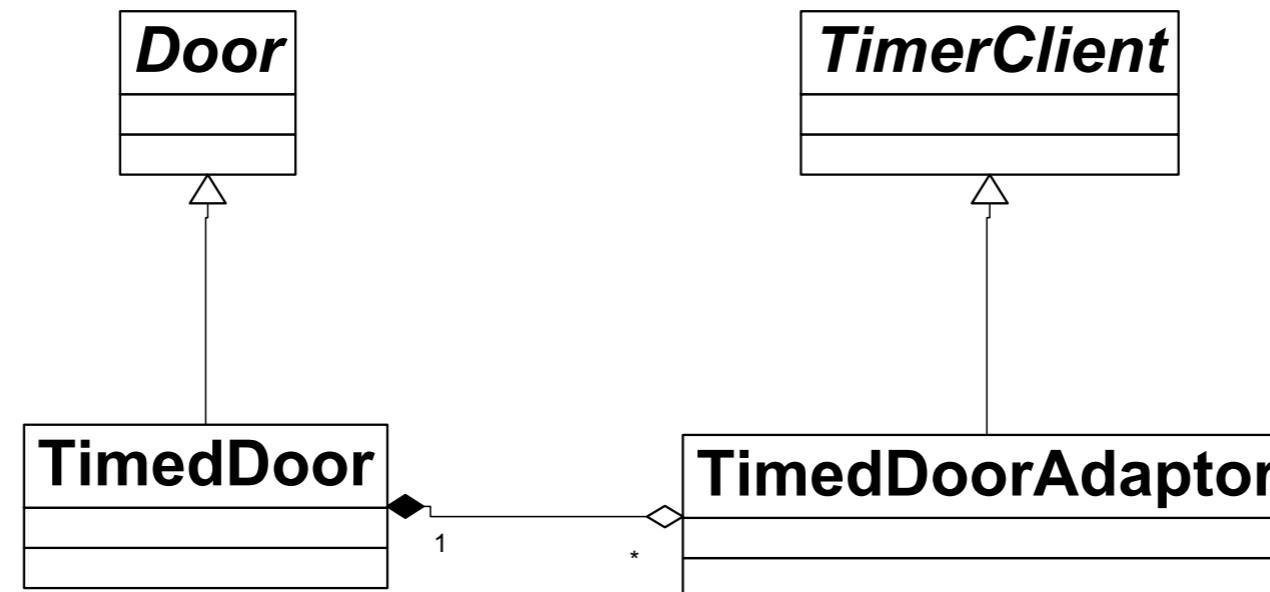
ISP Solution: yes or no?



Yes, separation through multiple inheritance

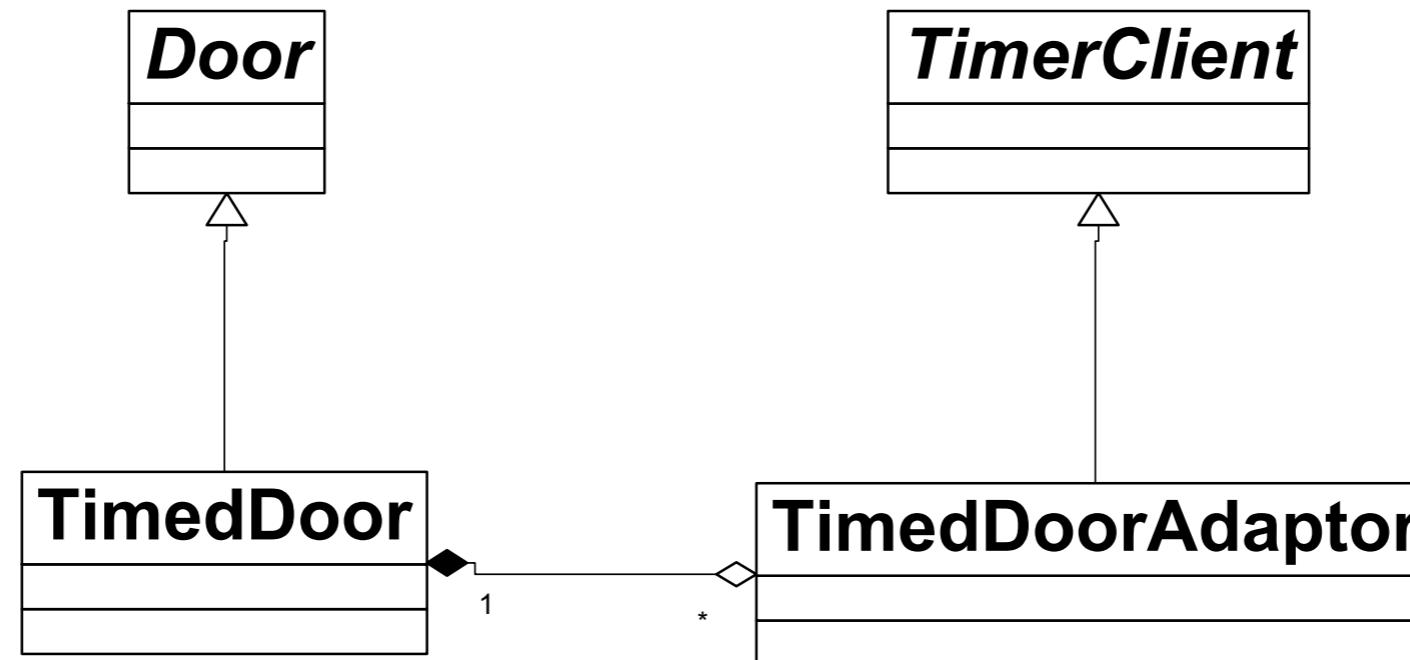
ISP solution: yes or no?

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



ISP solution: yes or no?

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

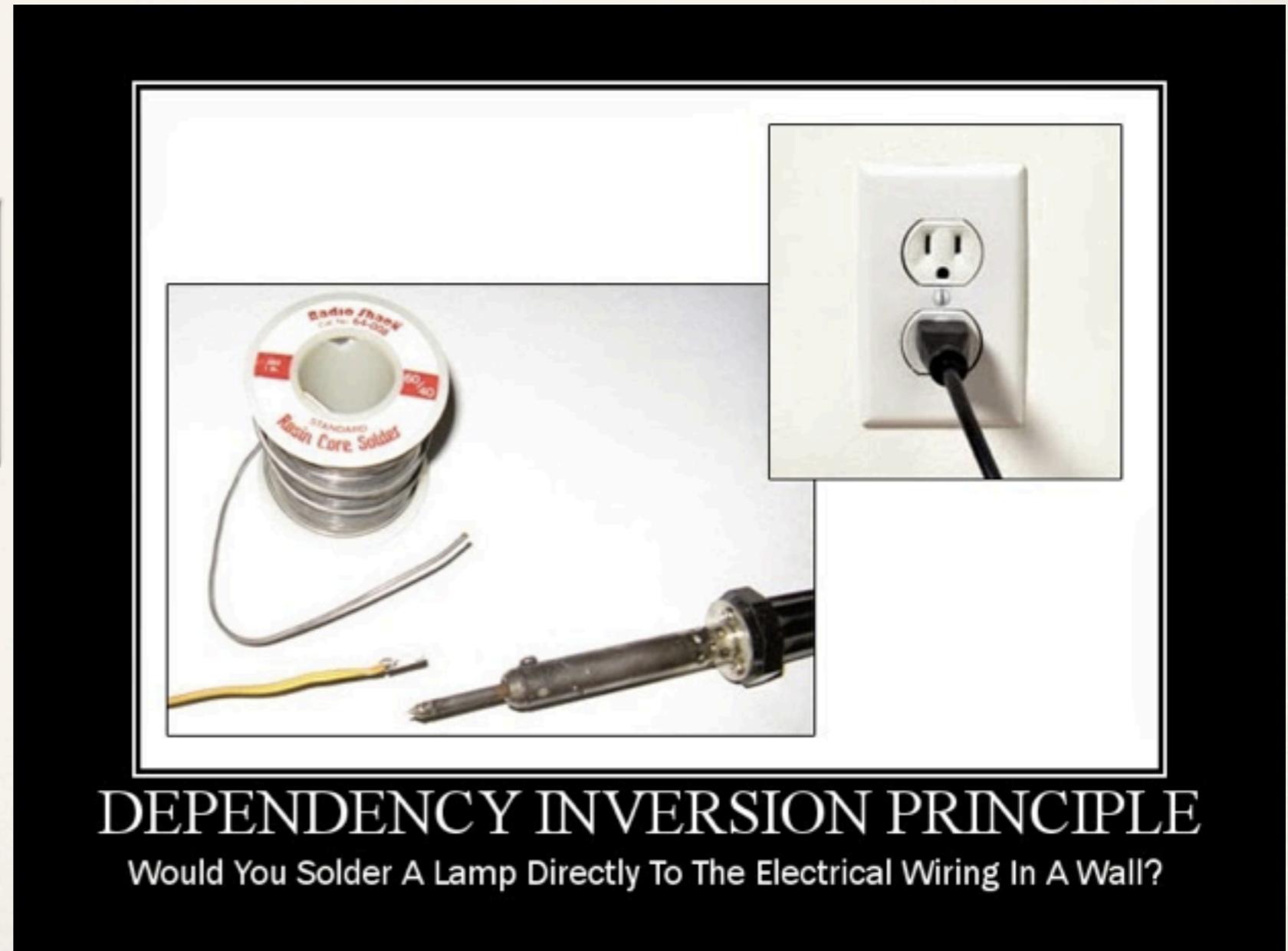


Yes, separation through delegation

Principe inversion de dépendance Dependency Inversion Principle (DIP)

Depend on abstractions,
not on concretions.

Robert C. Martin.

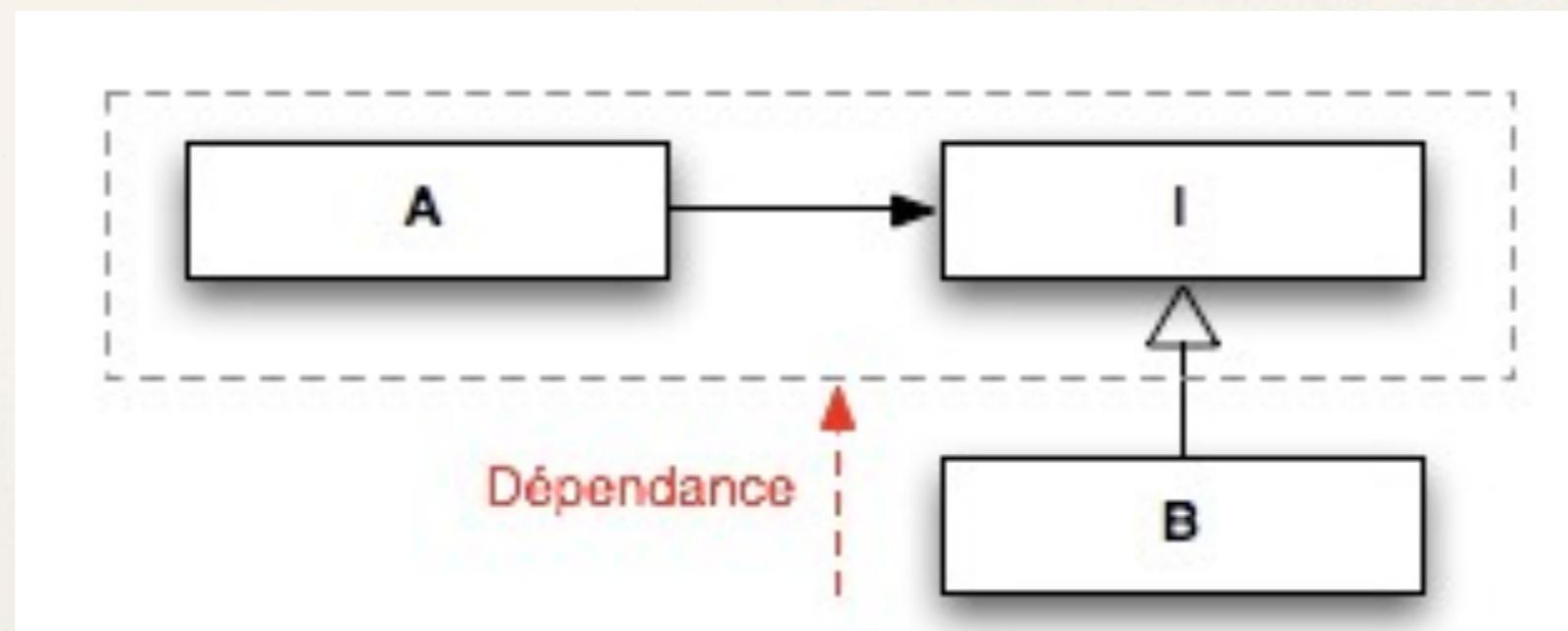
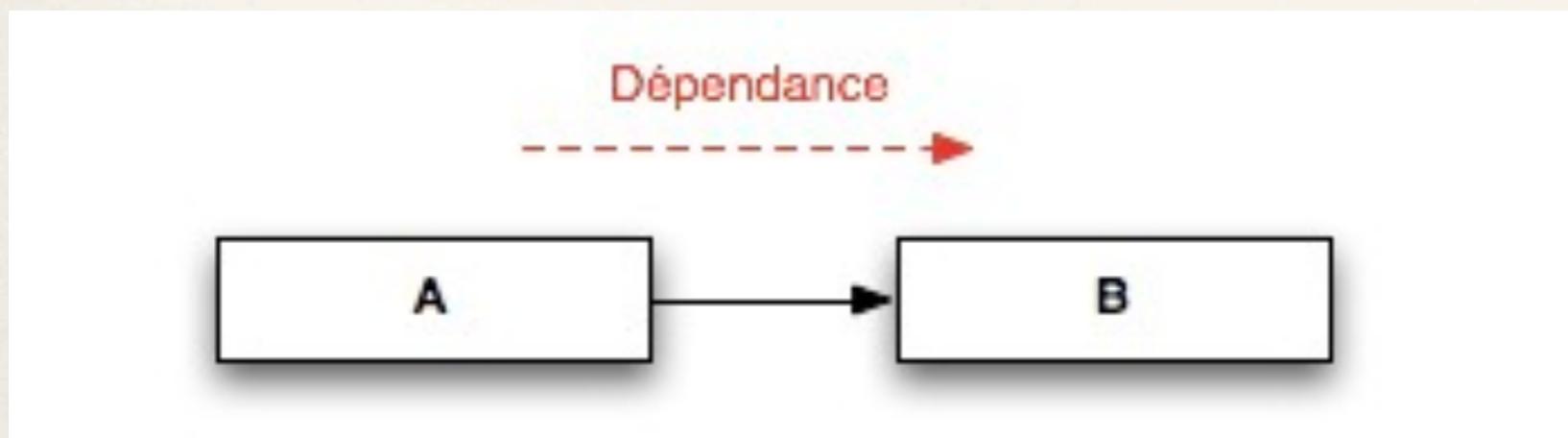


A LIRE !! <http://cyrilgandon.blogspot.fr/2013/07/inversion-des-dependances-solid-55.html>

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!

Inversion de dépendance



Exemple de couplage fort

```
package com.objis.spring.demoinjection;

public class Saxophone implements Instrument {
    public void jouer() {
        System.out.println("TOOT TOOT TOOT");
    }
}
```

```
package com.objis.spring.demoinjection;

public class MusicienSansInjection {

    private String morceau;
    private Saxophone instrument ;

    public void joueInstrument() throws PerformanceException {
        System.out.println("Le Saxophone joue morceau " + morceau);
        instrument.jouer();
    }

    public MusicienSansInjection(String morceau) {
        this.morceau = morceau;
        instrument = new Saxophone();
    }
}
```

Problèmes couplage fort

- Difficile de tester la Classe Musicien
- Difficile de réutiliser la Classe Musicien

solutions

1) Masquer l'implémentation avec Interfaces !

Relâcher le couplage

- Pour travailler avec un objet possédant un savoir-faire, nous déclarons une interface que l'objet doit implémenter.
- Cela crée un couplage faible entre l'objet demandeur et l'objet appelé. Ils n'ont pas besoin de se connaître mutuellement.

Exemple de couplage faible

```
package com.objis.spring.demoinjection;

public class Saxophone implements Instrument {
    public void jouer() {
        System.out.println("TOOT TOOT TOOT");
    }
}
```

Ici les classes sont
indépendantes.
Couplage faible

```
package com.objis.spring.demoinjection;

public class Piano implements Instrument {

    public void jouer() {
        System.out.println("PLINK PLINK PLINK");
    }
}

package com.objis.spring.demoinjection;

public class Musicien implements Performeur {

    private String morceau;
    private Instrument instrument ;

    public void performe() throws PerformanceException {
        System.out.print("joue " + morceau + " : ");
        instrument.jouer();
    }

    public void setMorceau(String morceau) {
        this.morceau = morceau;
    }

    public void setInstrument(Instrument instrument) {
        this.instrument = instrument;
    }
}
```

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.

Autres éléments de bibliographie

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

**Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.**

Martin Fowler

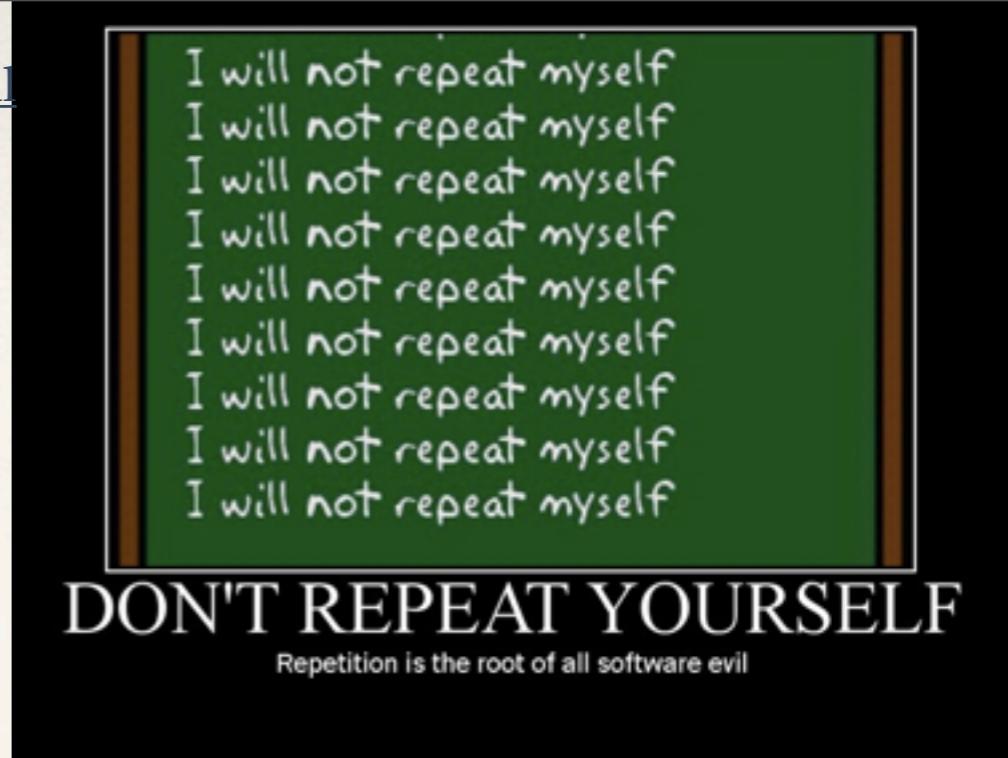
L'art du «Codage»

«Conventional wisdom says that once a project is in the coding phase, the work is mostly mechanical, transcribing the design into executable statements. We think that this attitude is the single biggest reason that many programs are ugly, inefficient, poorly structured, unmaintainable, and just plain wrong.

Coding is not mechanical. If it were, all the CASE tools that people pinned their hopes on in the early 1980s would have replaced programmers long ago. There are decisions to be made every minute—decisions that require careful thought and judgment if the resulting program is to enjoy a long, accurate, and productive life.»

Hunt, Thomas «The pragmatic Programmer»

1. Eviter la duplication
2. Composition versus Héritage
3. Optimisation
4. Programmation par coïncidence
5. Estimation des algorithmes
6. Point de vue sur le «refactoring»



Ecrire du bon code :

Don't Repeat Yourself (DRY)

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

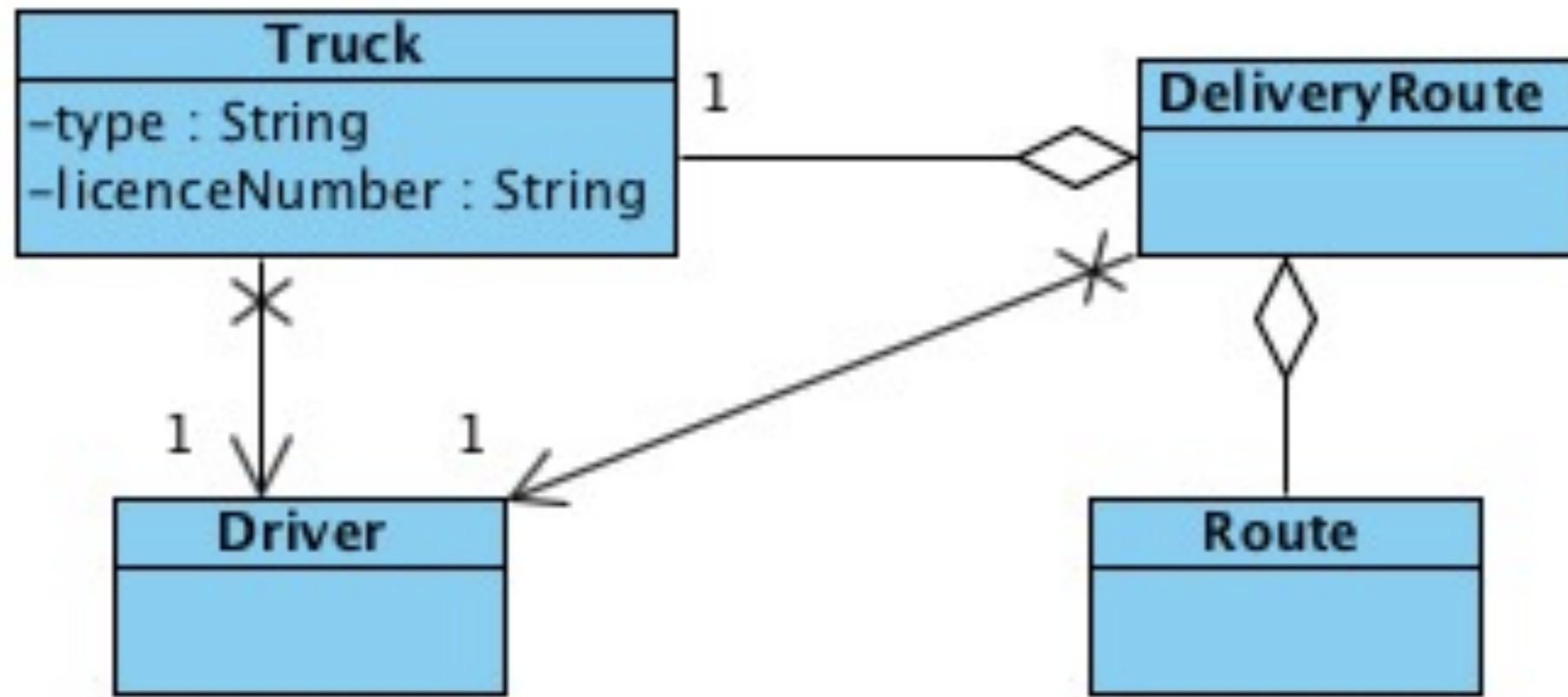
Causes de duplication des codes :

- Imposition a priori de l'environnement,
- Inattention,
- Facilité,
- Multiplicité des développeurs

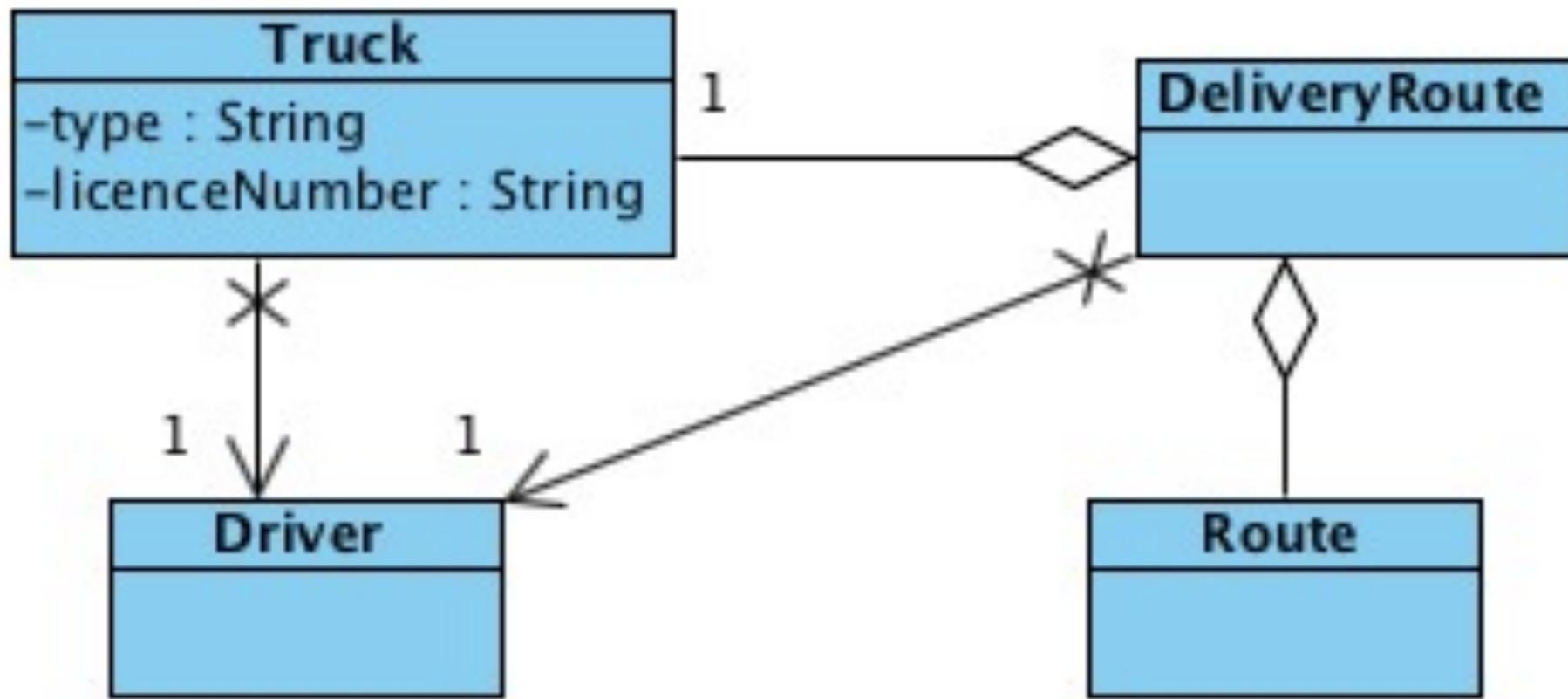
DRY (1) : Des exemples de duplications imposées et des solutions

- ❖ Documentation du code
 - => ne garder les commentaires détaillés que pour le haut niveau.
- ❖ Multiples représentations d'une information (coté client et serveur par exemple, une classe miroir d'une table dans la BD) => des filtres, des générateurs de code, metadata et générateur, génération de la classe à partir de la BDD ou du schéma, ou du modèle,...
- ❖ Les langages forcent des duplications : utiliser des outils!

Dry (2) : Des exemples de duplication par inattention et des solutions

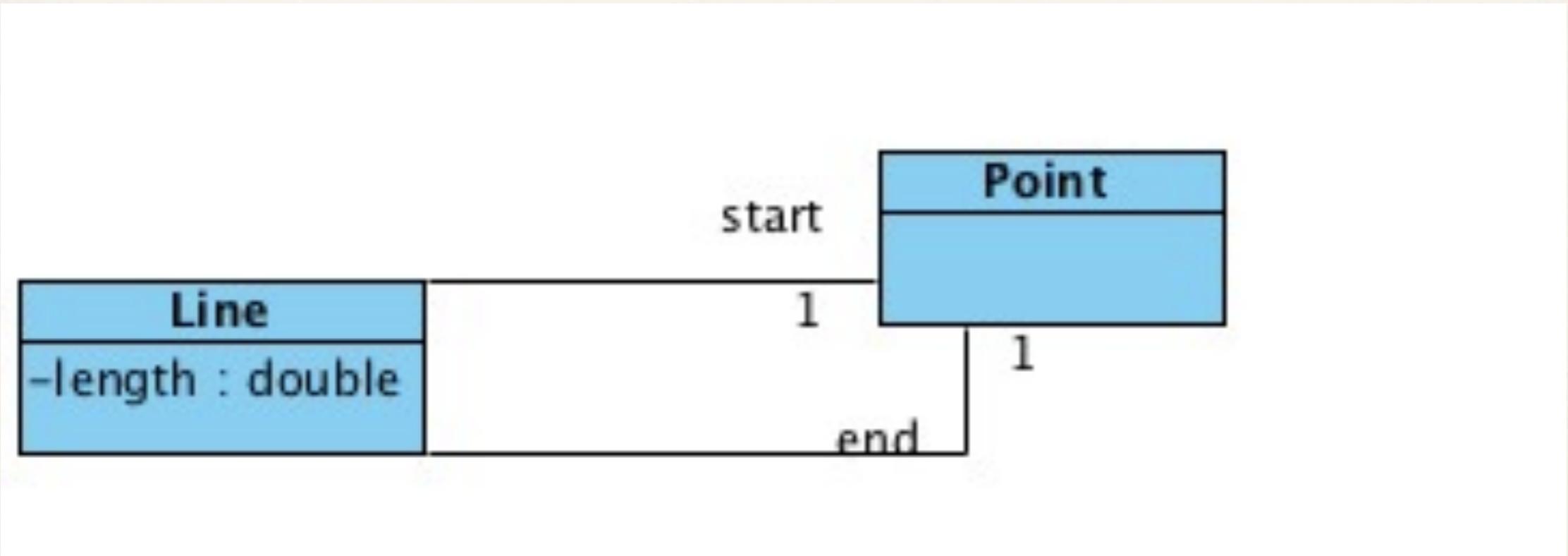


Dry (2) : Des exemples de duplication par inattention et des solutions



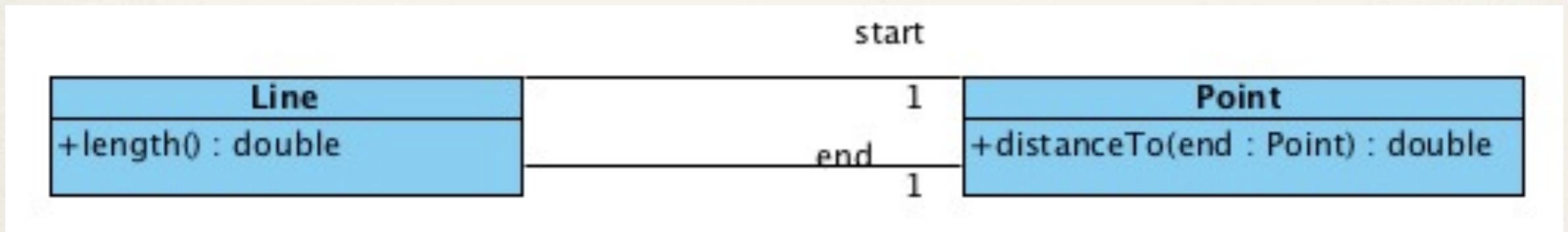
- Que faut-il modifier pour changer un chauffeur ? N'y a t'il pas une connaissance dupliquée?

Dry (2) : Des exemples de duplications par inattention et des solutions



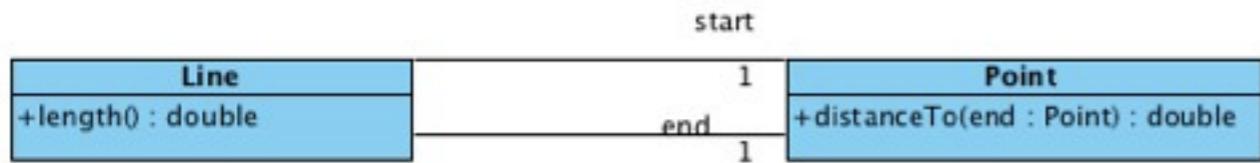
- ❖ Qu'est-ce qui est dupliqué?

Dry (2) : Des exemples de duplications par inattention et des solutions



- ❖ `return start.distanceTo(end);`

Dry (2) : Des exemples de duplications par inattention et des solutions



- Optimisation : le modèle ne change pas forcément

Version «paresseuse» : on ne calcule la longueur que si besoin !

```
private boolean changed;
private double length;
private Point start, end;

public Line(Point start, Point end) {
    super();
    this.start = start;
    this.end = end;
    changed=true;
    this.getLength();
}

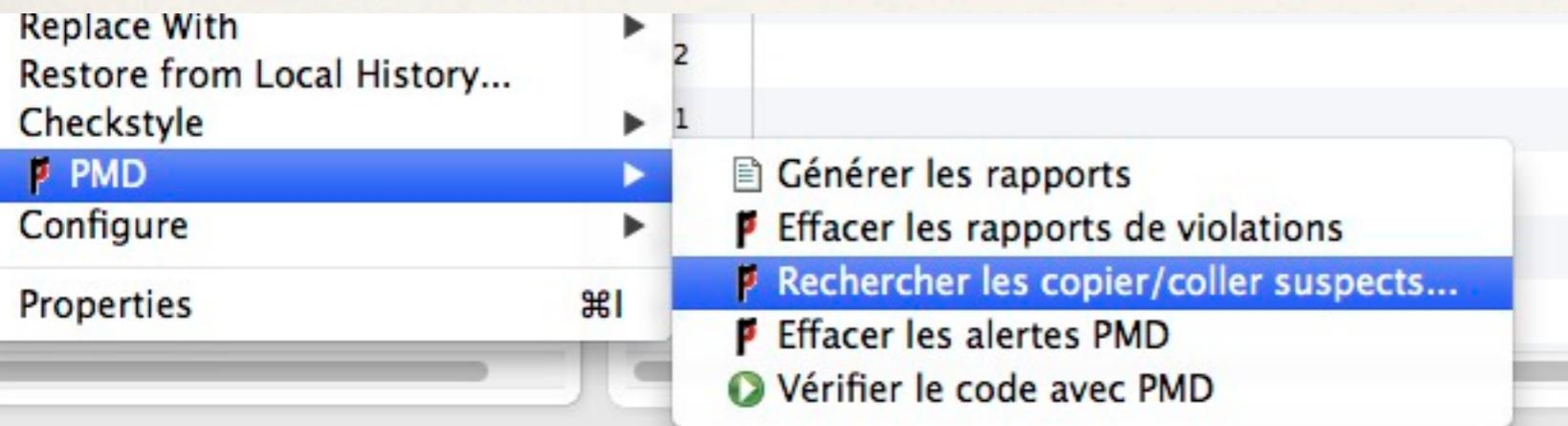
public Point getStart() {
    return start;
}

public void setStart(Point start) {
    this.start = start;
    changed = true;
}
public Point getEnd() {
    return end;
}
public void setEnd(Point end) {
    this.end = end;
    changed = true;
}
public double getLength() {
    if (changed) {
        length = start.distanceTo(end);
        changed = false;
    }
    return length;
}
```

Dry(3) : Des exemples de duplications par multi-développeurs et des solutions

- ❖ Vérification du numéro de sécurité social ... 10 000 programmes définissant des vérification équivalentes

- ✓ Ok, un bon manager peut éviter certaines duplications
 - Mais aussi la communication entre développeurs
 - Les outils de recherche de codes dupliqués.



voir en TD.

Ecrire du bon code : Préférer la composition à l'héritage

Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is safe to use inheritance within a package, where the subclass and the superclass implementations are under the control of the same programmers. It is also safe to use inheritance when extending classes specifically designed and documented for extension. Inheriting from ordinary concrete classes across package boundaries, however, is dangerous.

Joshua Bloch



<http://verraes.net/2014/05/final-classes-in-php/>

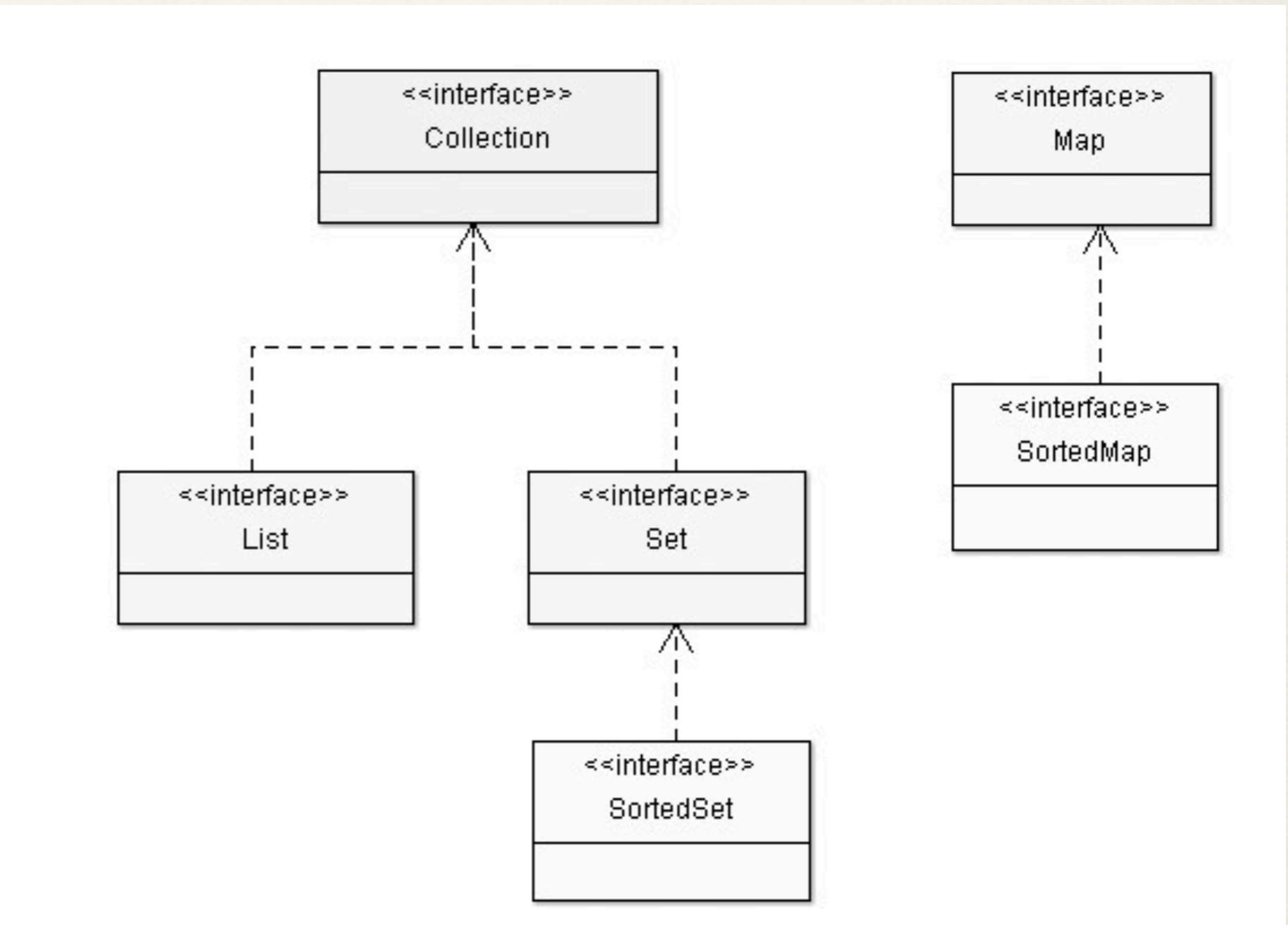
Héritage & Composition

- ✿ Préférer la composition à l'héritage
 - L'héritage a été mis en avant pour la réutilisation
 - Trop !
 - Souvent, l'héritage est rigide et la composition est souple

Composition

- ❖ Method of reuse in which new functionality is obtained by creating an object *composed of* other objects
- ❖ The new functionality is obtained by delegating functionality to one of the objects being composed
- ❖ Sometimes called *aggregation* or *containment*, although some authors give special meanings to these terms

Inheritance vs Composition Example



```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

Inheritance vs Composition Example

- Suppose we want a variant of HashSet that keeps track of the number of attempted insertions. So we subclass HashSet as follows:

```
public class InstrumentedHashSet extends HashSet {  
  
    // The number of attempted element insertions  
    private int addCount = 0;  
  
public InstrumentedHashSet() {super();}  
public InstrumentedHashSet(Collection c) {super(c);}  
public InstrumentedHashSet(int initCap, float loadFactor) {  
    super(initCap, loadFactor);  
}
```

<http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf>

Inheritance vs Composition

Example (Continued)

```
public boolean add(Object o) {  
    addCount++;  
    return super.add(o);  
}  
  
public boolean addAll(Collection c) {  
    addCount += c.size();  
    return super.addAll(c);  
}  
  
public int getAddCount() {  
    return addCount;  
}  
}
```

<http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf>

Inheritance vs Composition

Example (Continued)

```
public static void main(String[] args) {  
    InstrumentedHashSet s = new InstrumentedHashSet();  
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));  
    System.out.println(s.getAddCount());  
}
```

- Let's test it!

Inheritance vs Composition

Example (Continued)

```
public static void main(String[] args) {  
    InstrumentedHashSet s = new InstrumentedHashSet();  
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));  
    System.out.println(s.getAddCount());  
}
```

- Let's test it! 6, why ??

Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {  
    InstrumentedHashSet s = new InstrumentedHashSet();  
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));  
    System.out.println(s.getAddCount());  
}
```

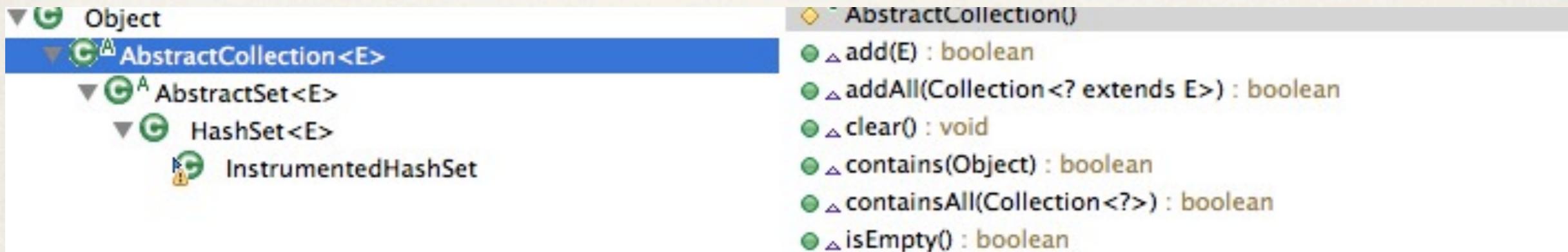
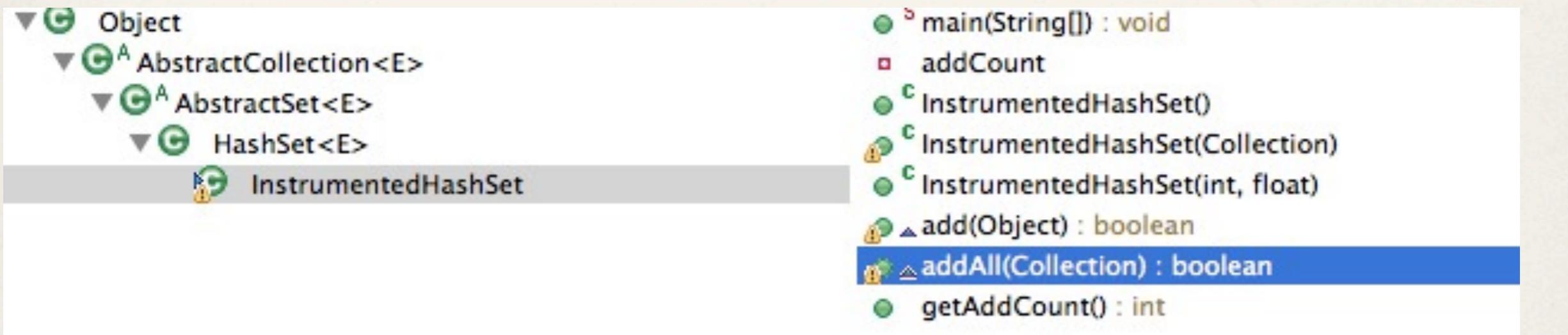
- Let's test it! 6, why ??



Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {  
    InstrumentedHashSet s = new InstrumentedHashSet();  
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));  
    System.out.println(s.getAddCount());  
}
```

- Let's test it! 6, why ??



Inheritance vs Composition

Example (Continued)

```
public static void main(String[] args) {  
    InstrumentedHashSet s = new InstrumentedHashSet();  
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));  
    System.out.println(s.getAddCount());  
}
```

- * Let's test it! 6, why ??

```
144:     public boolean addAll(Collection<? extends E> c)  
145:     {  
146:         Iterator<? extends E> itr = c.iterator();  
147:         boolean modified = false;  
148:         int pos = c.size();  
149:         while (--pos >= 0)  
150:             modified |= add(itr.next());  
151:         return modified;  
152:     }
```

Inheritance vs Composition

Example (Continued)

- Implementation details of our superclass affected the operation of our subclass.
- The best way to fix this is to use composition. Let's write an InstrumentedSet class that is composed of a Set object. Our InstrumentedSet class will duplicate the Set interface, but all Set operations will actually be forwarded to the contained Set object.
- InstrumentedSet is known as a **wrapper** class, since it wraps an instance of a Set object.

Inheritance vs Composition

Example (Continued)

- L'implementation des superclasses a affecté l'opération de la sous-classe.
- Une autre approche est d'utiliser la composition : La nouvelle classe «InstrumentedSet» n'est plus une sorte de «Set» mais est composée d'un ensemble d'objet.
- Cette classe « InstrumentedSet» duplique l'interface «Set», mais toutes les opérations sont déléguées à l'objet ensemble contenu.
- *InstrumentedSet is known as a wrapper class, since it wraps an instance of a Set object.*

Inheritance vs Composition

Example (Continued)

```
public class InstrumentedSet implements Set {  
    private final Set s;  
    private int addCount = 0;  
  
    public InstrumentedSet(Set s) {this.s = s;}  
  
    public boolean add(Object o) {  
        addCount++;  
        return s.add(o);  
    }  
  
    public boolean addAll(Collection c) {  
        addCount += c.size();  
        return s.addAll(c);  
    }  
  
    public int getAddCount() {return addCount;}}
```

```
// Forwarding methods (the rest of the Set interface methods)
public void clear() { s.clear(); }
public boolean contains(Object o) { return s.contains(o); }
public boolean isEmpty() { return s.isEmpty(); }
public int size() { return s.size(); }
public Iterator iterator() { return s.iterator(); }
public boolean remove(Object o) { return s.remove(o); }
public boolean containsAll(Collection c)
    { return s.containsAll(c); }
public boolean removeAll(Collection c)
    { return s.removeAll(c); }
public boolean retainAll(Collection c)
    { return s.retainAll(c); }
public Object[] toArray() { return s.toArray(); }
public Object[] toArray(Object[] a) { return s.toArray(a); }
public boolean equals(Object o) { return s.equals(o); }
public int hashCode() { return s.hashCode(); }
public String toString() { return s.toString(); }
}
```

Inheritance vs Composition

Example (Continued)

```
public static void main(String[] args) {  
    InstrumentedSet s1 = new InstrumentedSet(new HashSet());  
    s1.addAll(Arrays.asList(new String[] {"Snap", "Crackle", "Pop"}));  
    System.out.println(s1 + " " + s1.getAddCount());  
}
```

[Pop, Snap, Crackle] 3

Inheritance vs Composition

Example (Continued)

```
public static void main(String[] args) {  
    List list = new ArrayList();  
    InstrumentedSet s1 = new InstrumentedSet(new TreeSet(list));  
    s1.addAll(Arrays.asList(new String[] {"Snap", "Crackle", "Pop"}));  
    System.out.println(s1 + " " + s1.getAddCount());  
}
```

[Crackle, Pop, Snap] 3

Inheritance vs Composition

Example (Continued)

- ❖ Note several things:
 - This class is a Set
 - It has one constructor whose argument is a Set
 - The contained Set object can be an object of any class that implements the Set interface (and not just a HashSet)
 - This class is very flexible and can wrap any preexisting Set object
- ❖ Example:

```
List list = new ArrayList();
Set s1 = new InstrumentedSet(new TreeSet(list));
```

```
int capacity = 7;
float loadFactor = .66f;
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

Advantages/Disadvantages of Inheritance

- ❖ Advantages:
 - New implementation is easy, since most of it is inherited
 - Easy to modify or extend the implementation being reused

- ❖ Disadvantages:
 - Breaks encapsulation, since it exposes a subclass to implementation details of its superclass
 - "White-box" reuse, since internal details of superclasses are often visible to subclasses
 - Subclasses may have to be changed if the implementation of the superclass changes
 - Implementations inherited from superclasses can not be changed at runtime

Advantages/Disadvantages Of Composition

- ❖ **Avantages:**

- Objets contenus sont accessibles par la classe contenant uniquement à travers leurs interfaces
- Réutilisation «Boîte noire» car les détails internes des objets contenus ne sont pas visibles : Bonne encapsulation
- Réduit les dépendances de mise en œuvre
- Chaque classe se concentre sur sa propre tâche
- La composition peut être définie dynamiquement lors de l'exécution à travers des objets qui acquièrent des références à d'autres objets du même type

- ❖ **Inconvénients:**

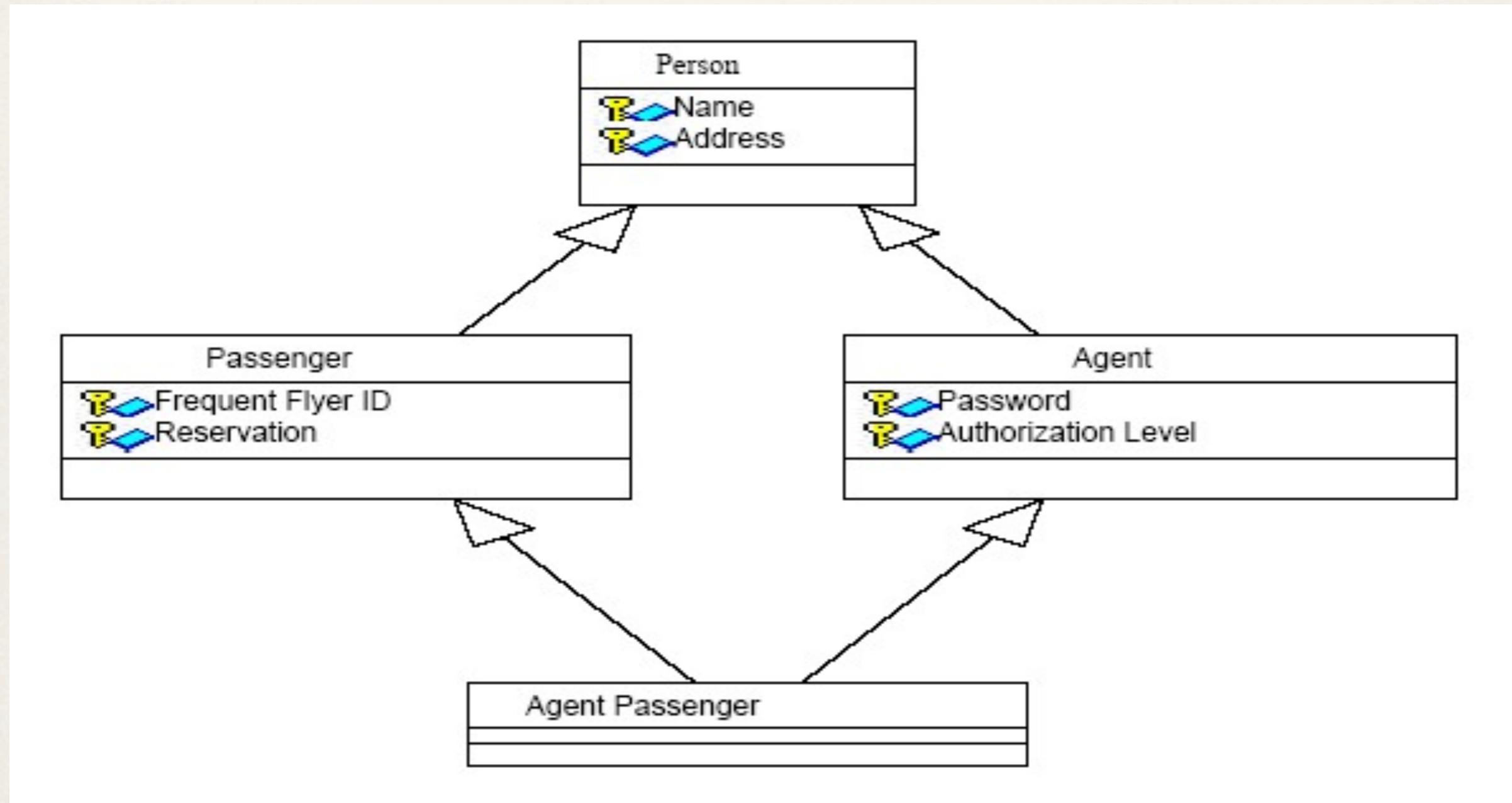
- Les systèmes résultant ont tendance à avoir plus d'objets

Coad's Rules of Using Inheritance

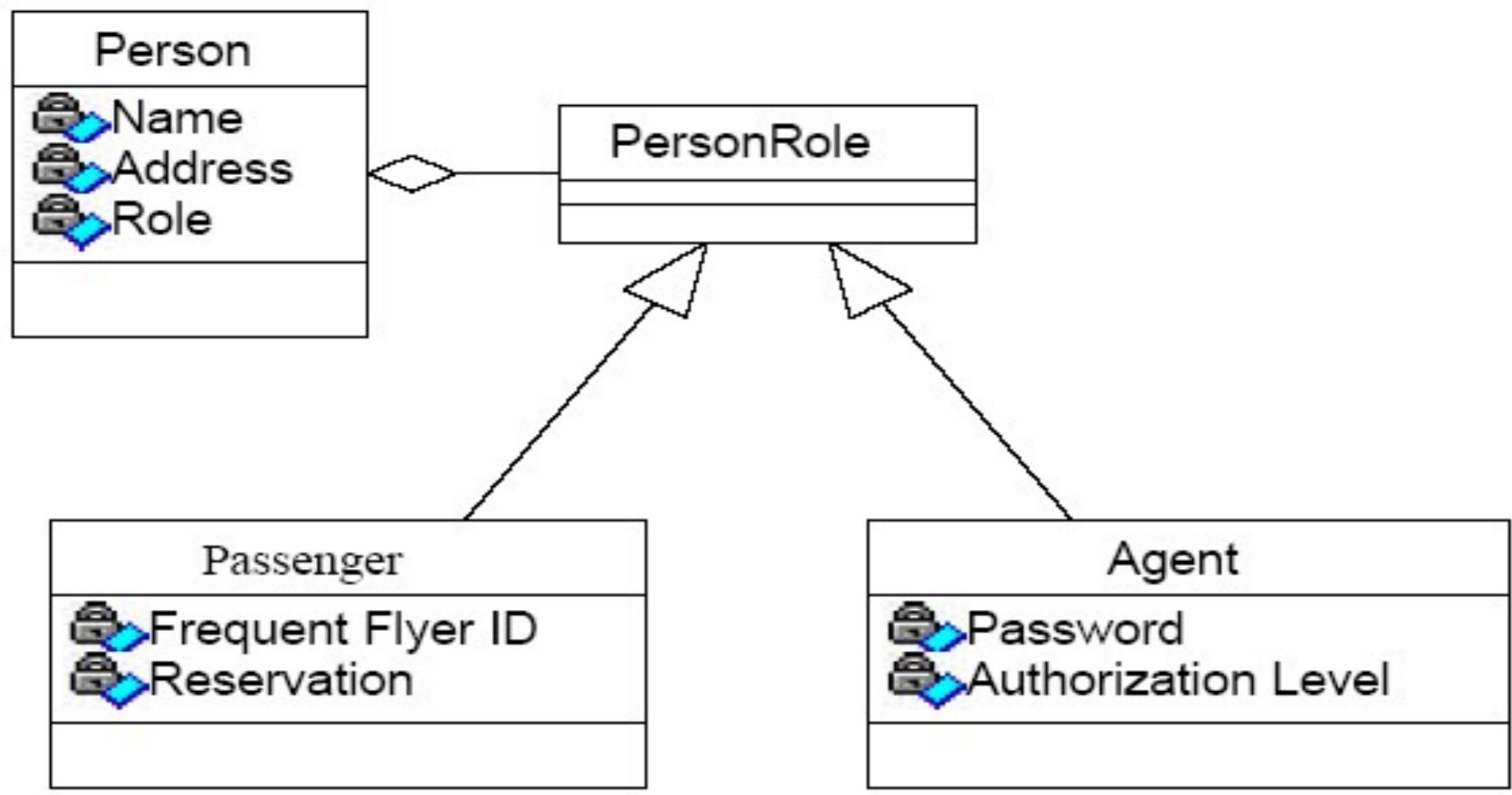
- * Use inheritance only when all of the following criteria are satisfied:
 - A subclass expresses "is a special kind of" and not "is a role played by a"
 - An instance of a subclass never needs to become an object of another class
 - A subclass **extends**, rather than **overrides** or **nullifies**, the responsibilities of its superclass
 - A subclass does not extend the capabilities of what is merely an utility class

Inheritance ?

Example 1



Composition ? Example 2



Premature Optimization

```
if (isset($frm['title_german']) [strcspn($frm['title_german'], '<>')]))  
{  
    // ...  
}
```

Optimiser les points vraiment utiles !

Ne mettez pas en péril la lisibilité et la maintenance de votre code pour de pseudo micro-optimisations!

Ne gâcher pas votre temps!

Eviter la programmation par coïncidence

«Developers who don't actively think about their code are programming by coincidence—the code might work, but there's no particular reason why »

Hunt, Thomas «The pragmatic Programmer»

- «Mon code est tombé en marche, enfin !»...
 - coïncidence? Accidents d'implémentation ?

paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);

```
public void reinit(){  
    size(650, 550);  
    background(255, 255, 255);  
    image(loadImage("background.png"), 0, 0);  
}  
  
public void setup() {  
    frameRate(4);  
    reinit();  
    memoriseCarts() ;....  
    // on veut garder la main sur le jeu c'est mousePressed qui stimule des redraw  
    noLoop();  
}  
  
public void draw() {  
    afficherJoueurs();  
    //afficherCartes();  
}
```

Eviter la programmation par coïncidence

«Developers who don't actively think about their code are programming by coincidence—the code might work, but there's no particular reason why »

Hunt, Thomas «The pragmatic Programmer»

- Ben, chez moi, ça marche...
 - coïncidence? Accidents de contexte ? Hypothèses implicites ?
- «A oui! ça ne peut pas marcher parce que tu n'as pas mis le code sous bazarland»

Penser à «Estimer» vos algorithmes

- Comment le programme se comportera s'il y a 1000 enregistrements? 1 000 000? Quelle partie optimiser?
- * Quelles dépendances entre par exemple la taille des données (longueur d'une liste, par exemple) et le temps de calcul? et la mémoire nécessaire?
- S'il faut 1s pour traiter 100 éléments, pour en traiter 1000, en faut-il :
 - 1, ($O(1)$) : temps constant
 - 10 ($O(n)$) : temps linéaire
 - 3 ($O(\log(n))$) : temps logarithmique
 - 100 ($O(n^2)$)
 - 10^{263} ($O(e^n)$) : temps exponentiel

Penser à «Estimer» vos algorithmes

- ❖ Tester vos estimations
- ❖ Optimiser si cela est **utile** et en tenant compte du **contexte**.

«Refactoring» ou l'art du «jardinage logiciel»

« Rather than construction, software is more like gardening—it is more organic than concrete. You plant many things in a garden according to an initial plan and conditions. Some thrive, others are destined to end up as compost. You may move plantings relative to each other to take advantage of the interplay of light and shadow, wind and rain. Overgrown plants get split or pruned, and colors that clash may get moved to more aesthetically pleasing locations. You pull weeds, and you fertilize plantings that are in need of some extra help. You constantly monitor the health of the garden, and make adjustments (to the soil, the plants, the layout) as needed»

Hunt, Thomas «The pragmatic Programmer»



La théorie des fenêtres cassées ou Eviter l'entropie du système

Ne pas laisser de fenêtre cassée :

- Réparer les codes
- Corriger les design dès que les défauts sont détectés.
- Si vous ne pouvez pas régler le problème, le circonscrire : annoter le code, noter «Not Implemented», ...
- Mais ne laisser pas des codes se déteriorer ou c'est l'ensemble de l'application qui en pâtira.



* Etude urbaine en 1982 : http://en.wikipedia.org/wiki/Broken_windows_theory

La théorie des fenêtres cassées ou Eviter l'entropie du système

Codez toujours en pensant que celui qui maintiendra votre code est un psychopathe qui connaît votre adresse.

Martin Golding

Ne pas laisser de fenêtre cassée :

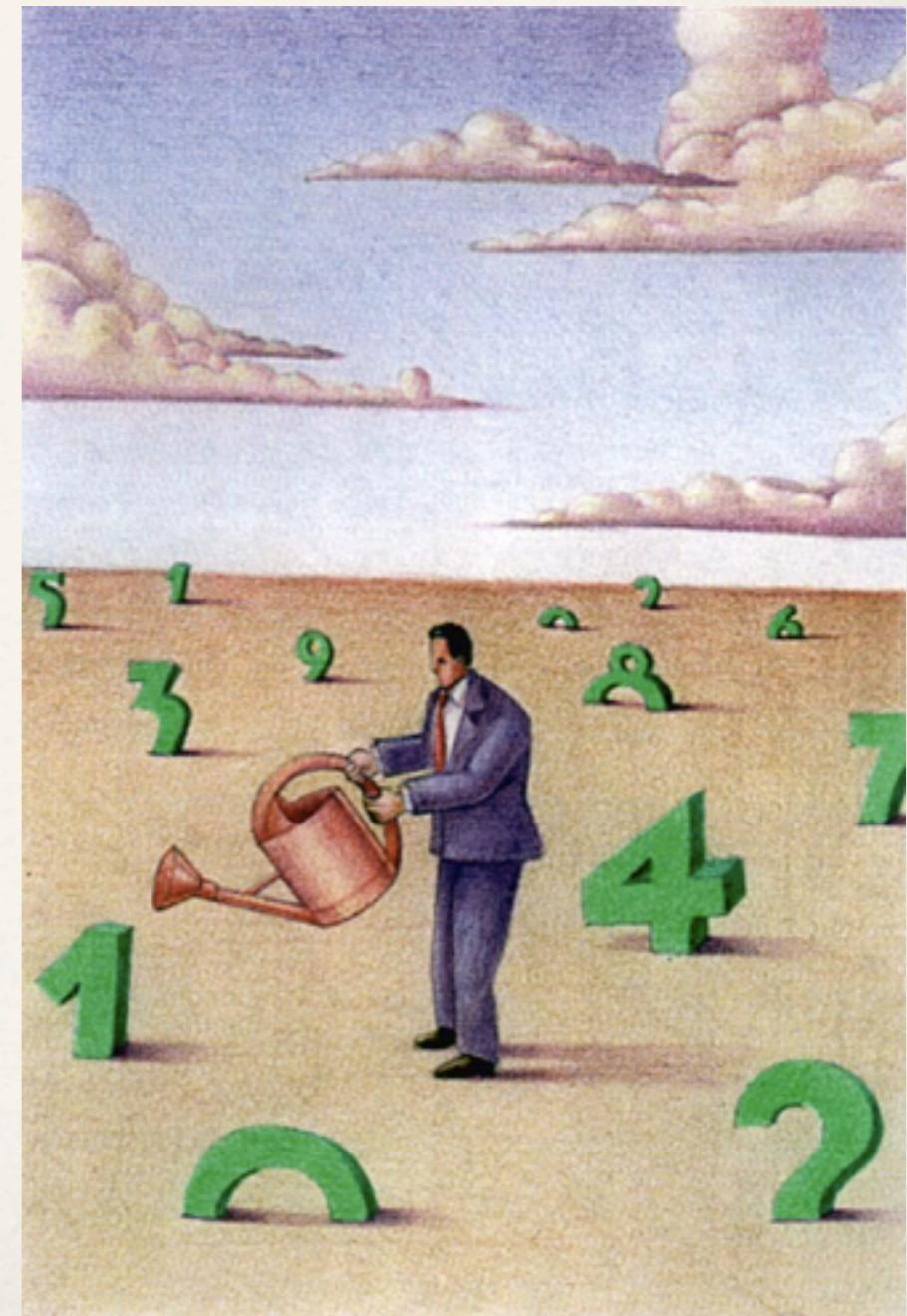
- Réparer les codes
- Corriger les design dès que les défauts sont détectés.
- Si vous ne pouvez pas régler le problème, le circonscrire : annoter le code, noter «Not Implemented», ...
- Mais ne laisser pas des codes se déteriorer ou c'est l'ensemble de l'application qui en pâtira.



* Etude urbaine en 1982 : http://en.wikipedia.org/wiki/Broken_windows_theory

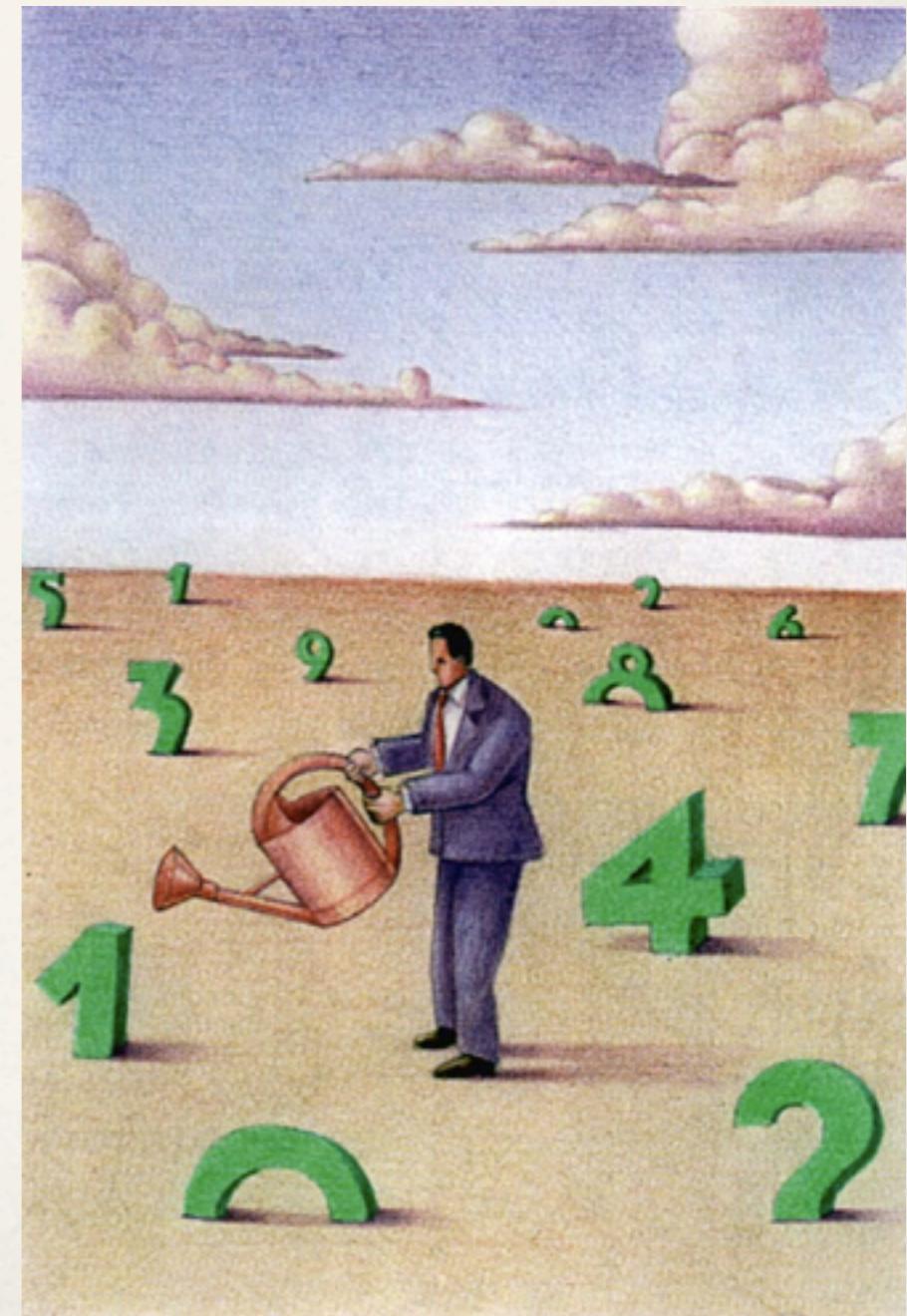
«Refactoring» : quand ?

- ❖ Pour éliminer les «fenêtres cassées»
- ❖ Pour améliorer le design : duplication (DRY), couplage, performance, ...
- ❖ Pour ajuster en fonction des besoins et des demandes de changements
- ❖ **Souvent, dès le début**
- Réfléchissez comme un jardinier pas comme un maçon...



«Refactoring» : comment ?

- ❖ Utilisez des outils pour identifier les changements (cf. cours Métriques)
- ❖ Utilisez des outils pour factoriser (Par exemple, Eclipse et les outils d extractions de méthodes, ...)
- ◎ **Organiser le refactoring**
 - Planifiez, Mettez des priorités, Mémorisez les changements à faire
 - Soyez sûr de vos tests avant de refactoriser
 - Progressez pas à pas



«Refactoring» : exemple (2)

This Java code is part of a framework that will be used throughout your project. Refactor it to be more general and easier to extend in the future.

```
public class Window {  
    public Window(int width, int height) { ... }  
    public void setSize(int width, int height) { ... }  
    public boolean overlaps(Window w) { ... }  
    public int getArea() { ... }  
}
```

«Refactoring» : exemple (2)

This case is interesting. At first sight, it seems reasonable that a window should have a width and a height. However, consider the future. Let's imagine that we want to support arbitrarily shaped windows (which will be difficult if the Window class knows all about rectangles and their properties). We'd suggest abstracting the shape of the window out of the Window class itself.

«Refactoring» : exemple (2)

```
public abstract class Shape {  
// ...  
public abstract boolean  
    overlaps(Shape s);  
public abstract int getArea();  
}
```

```
-  
public class Window {  
    private Shape shape;  
    public Window(Shape shape) {  
        this.shape = shape;  
        ... }  
    public void setShape(Shape shape) {  
        this.shape = shape;  
        ... }  
    public boolean overlaps(Window w) {  
        return shape.overlaps(w.shape);  
    } public int getArea() {  
        return shape.getArea();  
    }  
}
```

«Refactoring» : exemple (2)

Note that in this approach we've used delegation rather than subclassing: a window is not a "kind-of" shape—a window "has-a" shape. It uses a shape to do its job. You'll often find delegation useful when refactoring.

We could also have extended this example by introducing a Java interface that specified the methods a class must support to support the shape

functions. This is a good idea. It means that when you extend the concept of a shape, the compiler will warn you about classes that you have affected. We recommend using interfaces this way when you delegate all the functions of some other class.

Conclusion

- * En route vers des designs patterns issus du GOF et quelques patterns d'architecture, mais les principes présentés restent toujours vrais et doivent diriger vos choix de mise en oeuvre.

Prendre 10 s pour s'interroger

- Pourquoi je fais cela? Quel est mon objectif?
- Est-ce ainsi que cela doit être fait?
- Est-ce nécessaire de le faire?
- Que signifie «terminé» dans ce cas?

