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

Fait suite au cours sur Pragmatic Programming et se termine par des éléments du même «livre»

Mireille Blay-Fornarino, Université Nice Sophia Antipolis, Département Info IUT, Octobre 2014

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.

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

- Single responsibility principle (SRP)
- Open-closed principle (OCP)
- Liskov substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

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

The single-responsibility principle

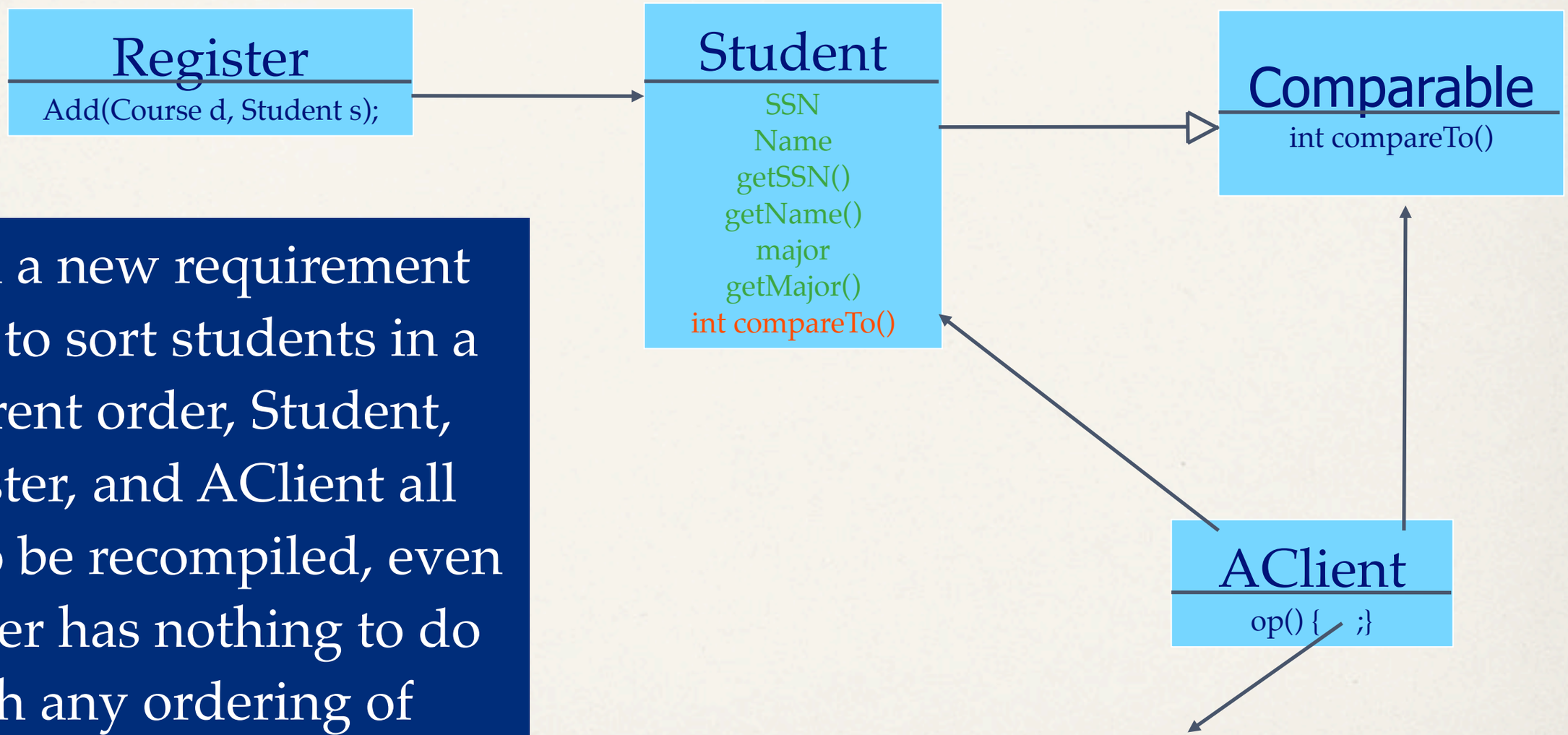
- **Example**:
  - Often we need to sort students by their name, or ssn. So one may make Class Student implement the Java Comparable interface.

```
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

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

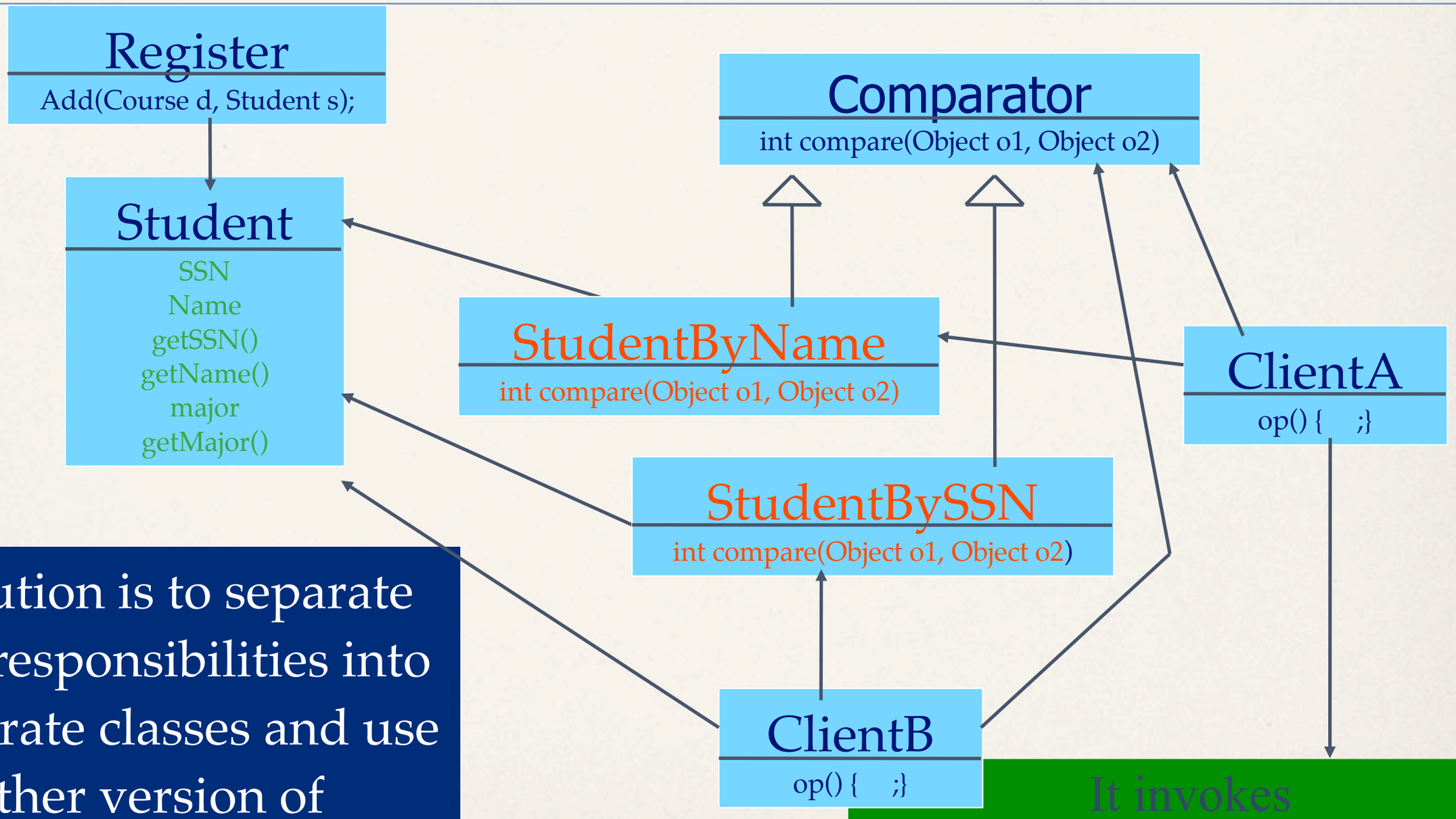
The single-responsibility principle



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



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 : 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;
        }
```

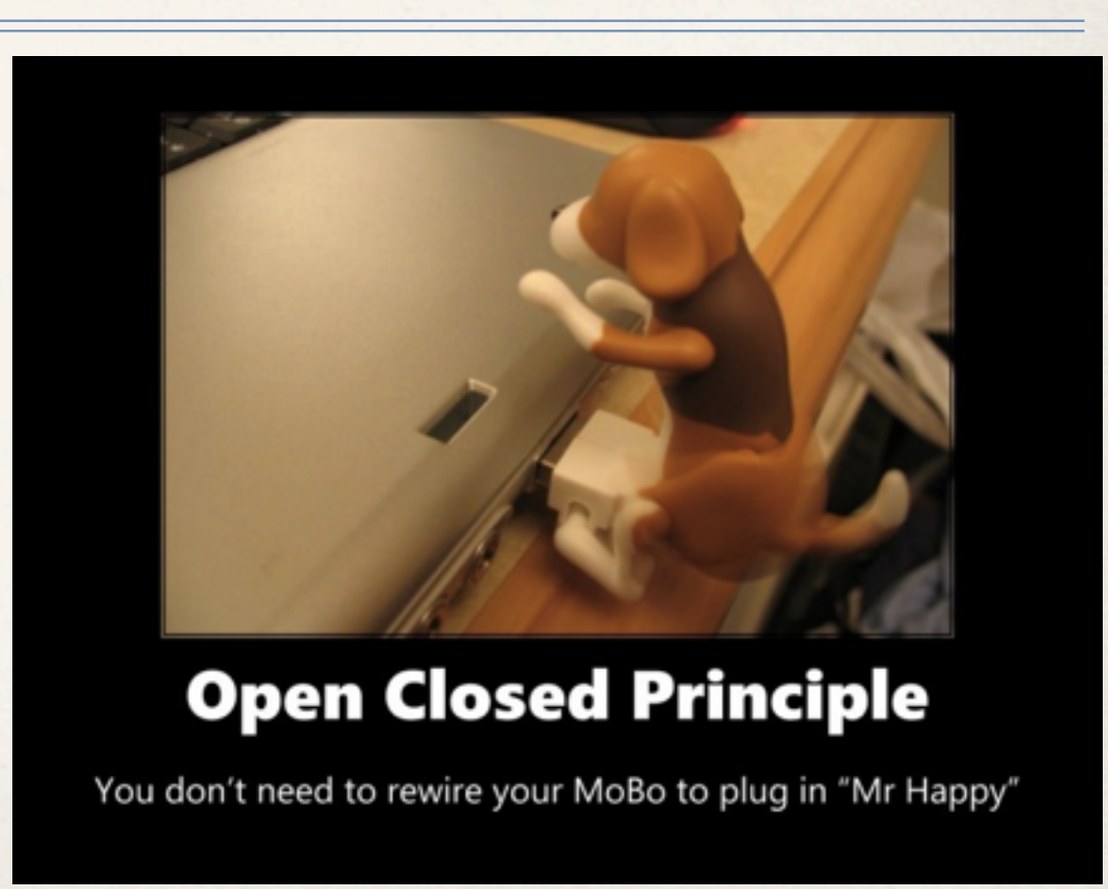
```
Comparator<Student> byNameComparator =
    new ByName();
Comparator<Student> bySectionComparator=
    new BySection();
```

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: 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.

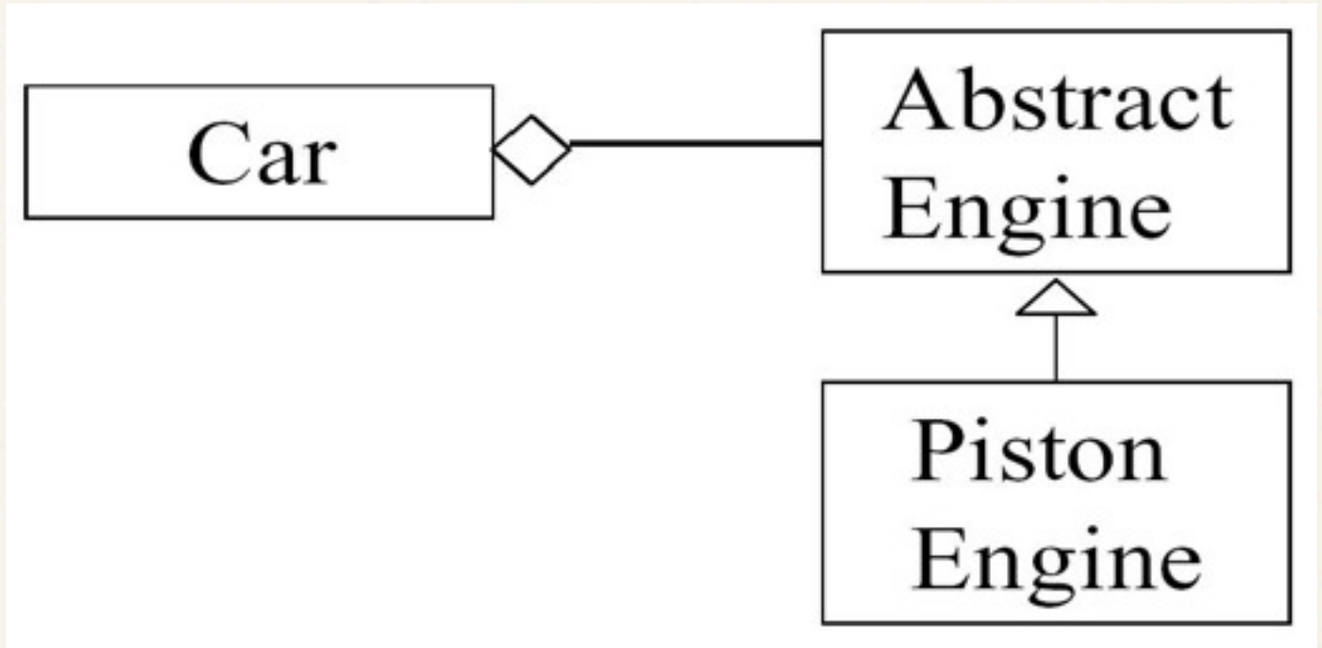
- **Les entités logicielles doivent être ouvertes à l'extension**
  - le code est extensible
- **mais fermées aux modifications**
  - Le code a été écrit et testé, on n'y touche pas.

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

Open the door …

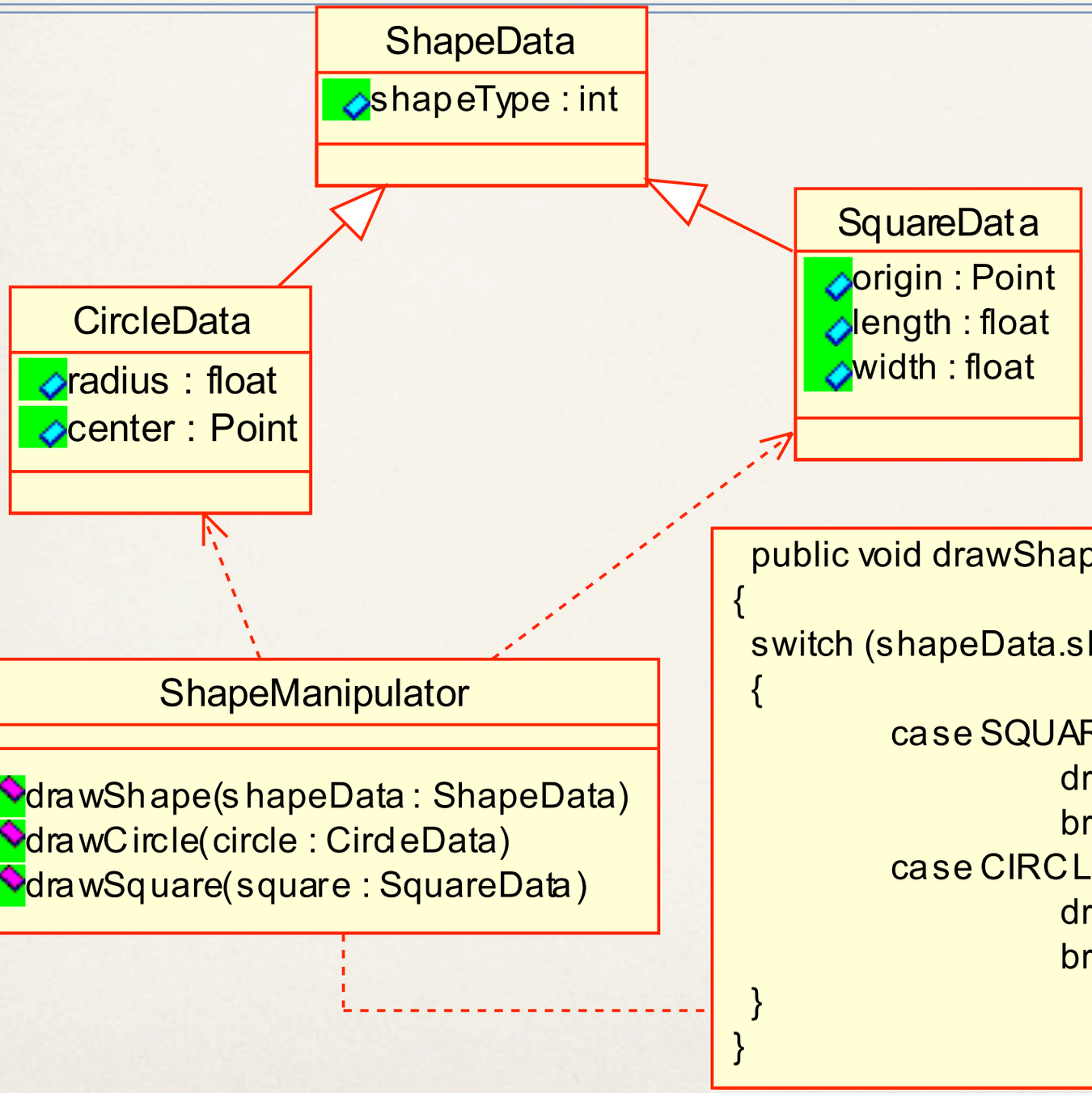
- Comment faire en sorte que la voiture aille plus vite à l'aide d'un turbo?
  - Il faut changer la voiture
    - avec la conception actuelle…

… But Keep It Closed!



- Une classe ne doit pas dépendre d'une classe Concrète
- Elle doit dépendre d'une classe abstraite
- ...et utiliser le polymorphisme

The Open/Closed Principle (OCP) Example



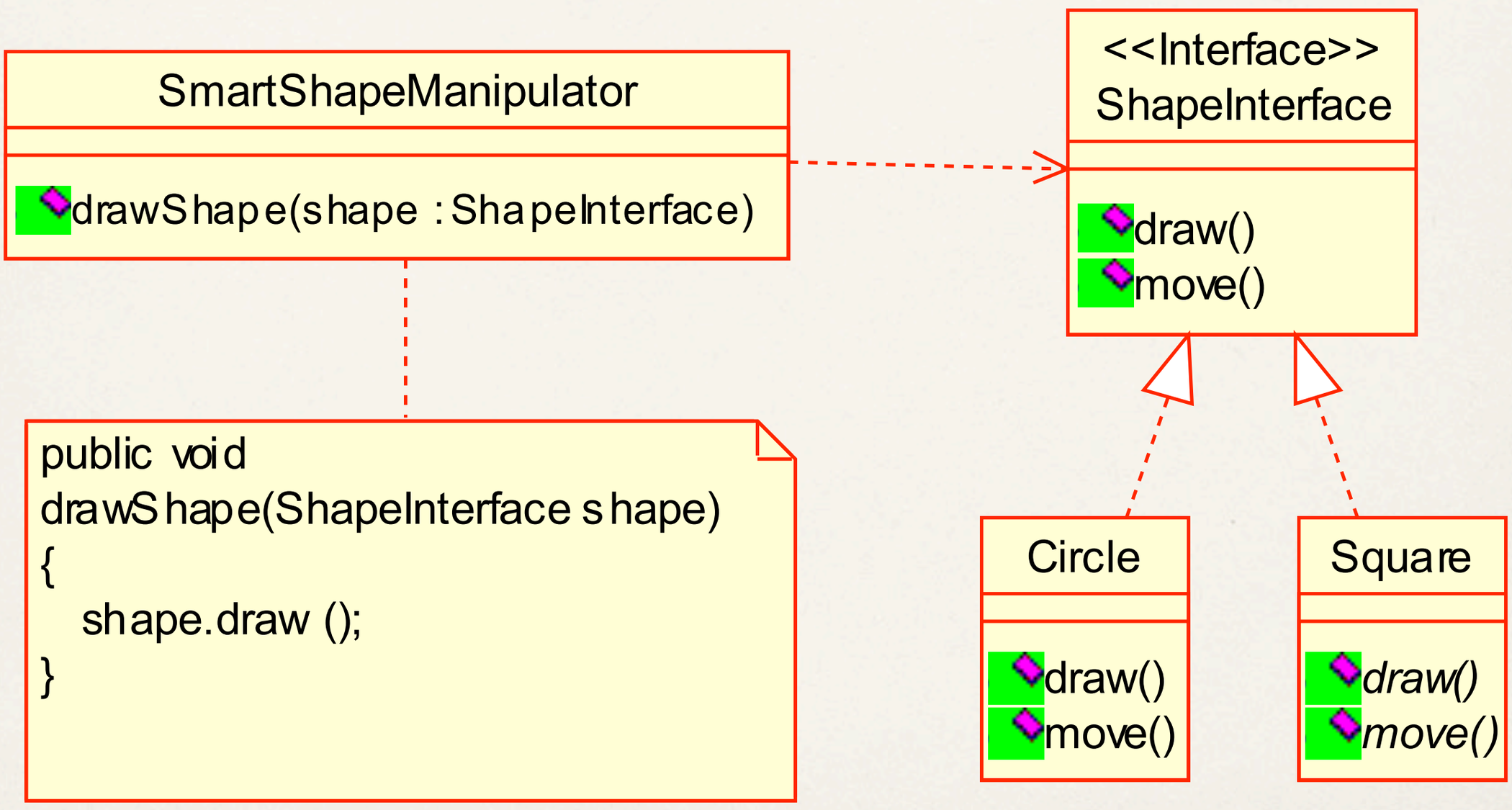
An Example of What not to do!

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



Heuristiques pour OCP

Mettre toutes les données d'un objet en privé!
Pas de Variables Globales!

- Modifier une donnée publique est «risqué» :
  - Il peut y avoir des effets de bord en des points que l'on attend pas!
  - Les erreurs peuvent être difficiles à identifier et leur correction déclencher d'autres erreurs.

RTTI* est une pratique dangereuse à éviter.

* RTTI = Run-Time Type Information

http://www.objectmentor.com/resources/articles/ocp.pdf

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)

- Here are example Part and ConcretePart classes:

```
// 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
        }
}
```

But now we must modify each subclass of Part whenever the pricing policy changes!

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

- A better idea is to have a PricePolicy class which can be used to provide different pricing policies:

```
// 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» est maintenant possible ...

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)

- It is not possible to have all the modules of a software system satisfy the OCP, but we should attempt to minimize the number of modules that do not satisfy it.

- The Open-Closed Principle is really the heart of OO design.

- Conformance to this principle yields the greatest level of reusability and maintainability.

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!

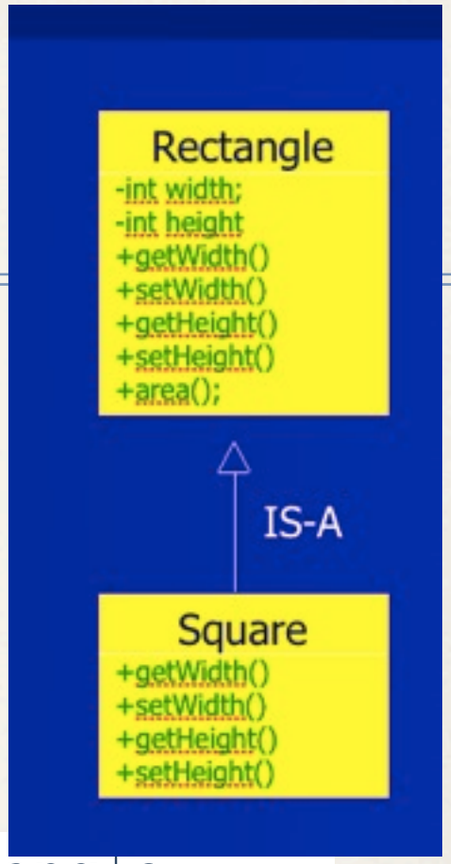
```
class Penguin : public Bird {
   public: void fly() {
      error ("Penguins don't fly!"); }
};
```

- 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(width);
        super.setHeight(width);
    }
  }
```

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

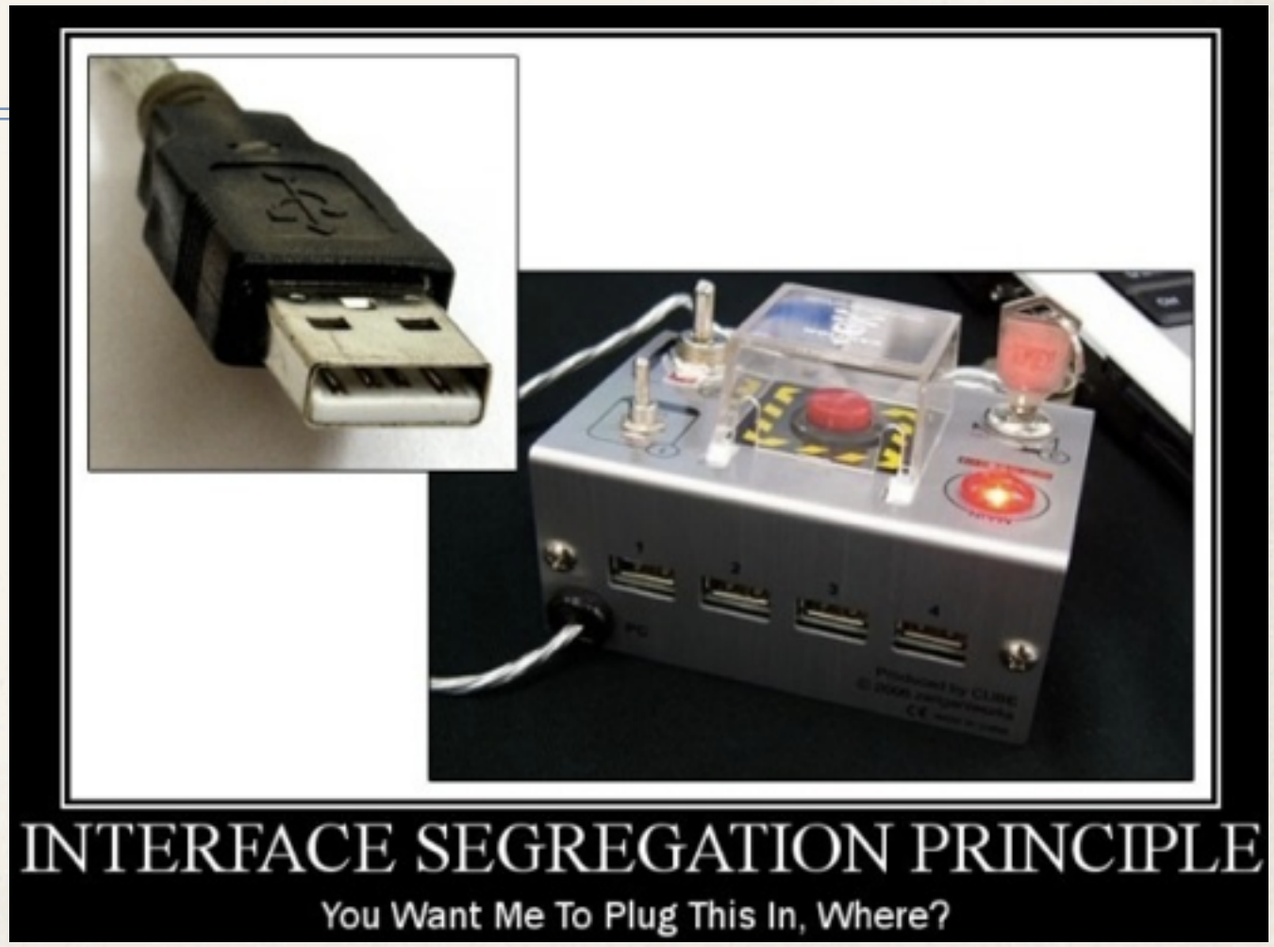
> It is illegal for a derived class, to override a base-class method with a NOP method

- 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`**

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"



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

- Many client-specific interfaces are better than one general-purpose interface.
- 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
    - Nécessaire : expérience, pragmatisme et le bon sens !

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

```java
public interface Animal {
    void fly();
    void run();
    void bark();
}
public class Bird implements Animal {
    public void bark() { /* do nothing */ }
    public void run() {
        // write code about running of the bird
    }
    public void fly() {
        // write code about flying of the bird
    }
}
public class Cat implements Animal {
    public void fly() { throw new Exception("Undefined cat
property"); }
    public void bark() { throw new Exception("Undefined cat
property"); }
    public void run() {
        // write code about running of the cat
    }
}
public class Dog implements Animal {
    public void fly() { }
    public void bark() {
        // write code about barking of the dog
    }
    public void run() {
        // write code about running of the dog
    }
}
```

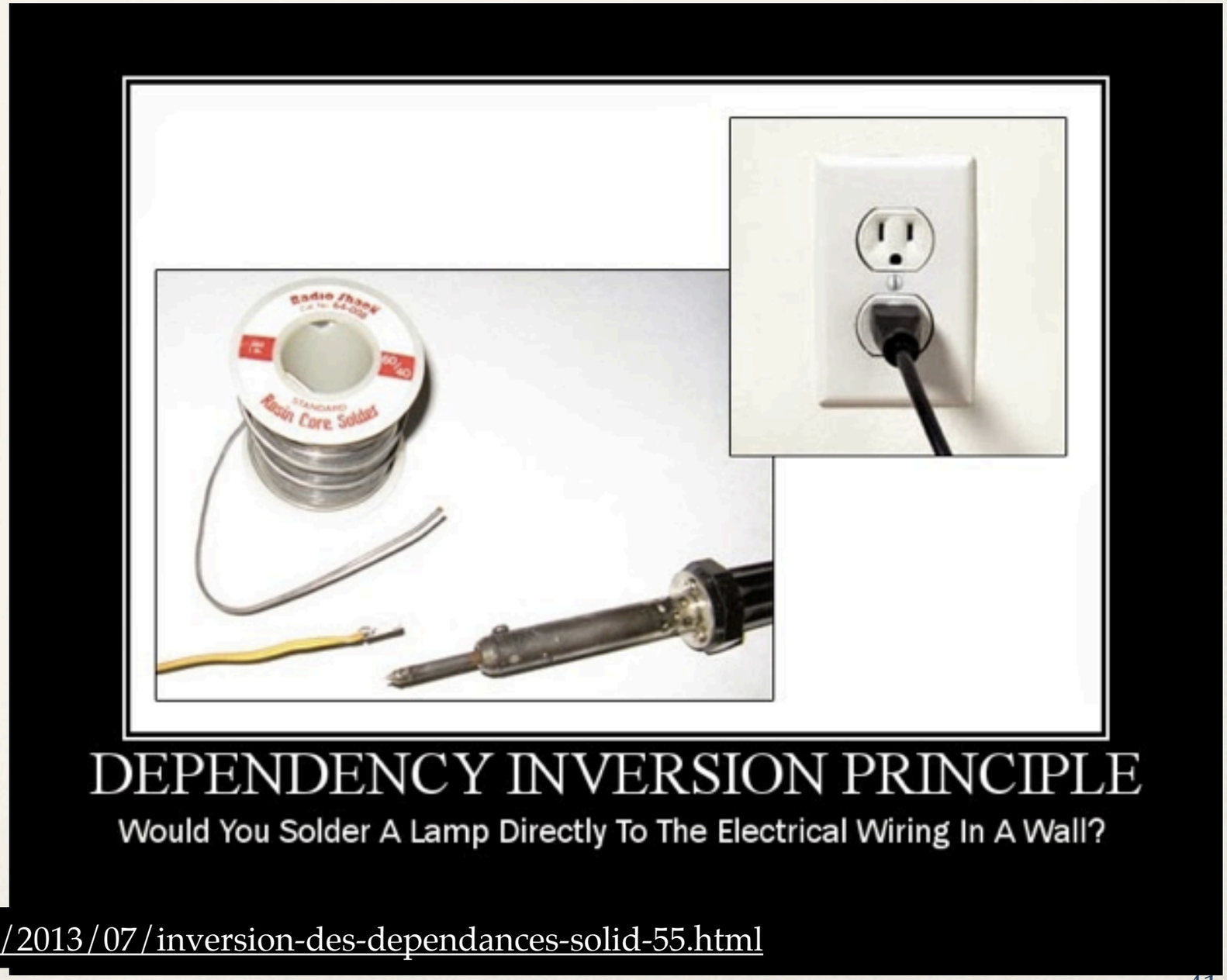
http://codebalance.blogspot.fr/2010/09/oop-solid-rules-interface-segregation.html

```java
public interface Flyable {
    void fly();
}
public interface Runnable {
    void run();
}
public interface Barkable {
    void bark();
}
public class Bird implements Flyable, Runnable {
    public void run() {
        // write code about running of the bird
    }
    public void fly() {
        // write code about flying of the bird
    }
}
public class Cat implements Runnable{
    public void run() {
        // write code about running of the cat
    }
}
public class Dog implements Runnable, Barkable {
    public void bark() {
        // write code about barking of the dog
    }
    public void run() {
        // write code about running of the dog
    }
}
```

http://codebalance.blogspot.fr/2010/09/oop-solid-rules-interface-segregation.html

Principe inversion de dépendance
Dependency Inversion Principle (DIP)

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



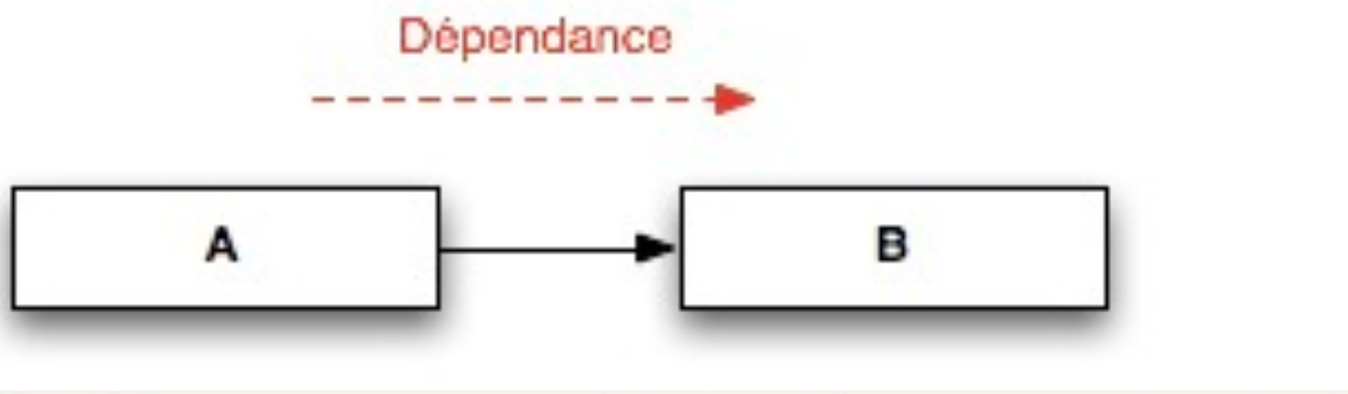
A LIRE !! http://cyrilgandon.blogspot.fr/2013/07/inversion-des-dependances-solid-55.html

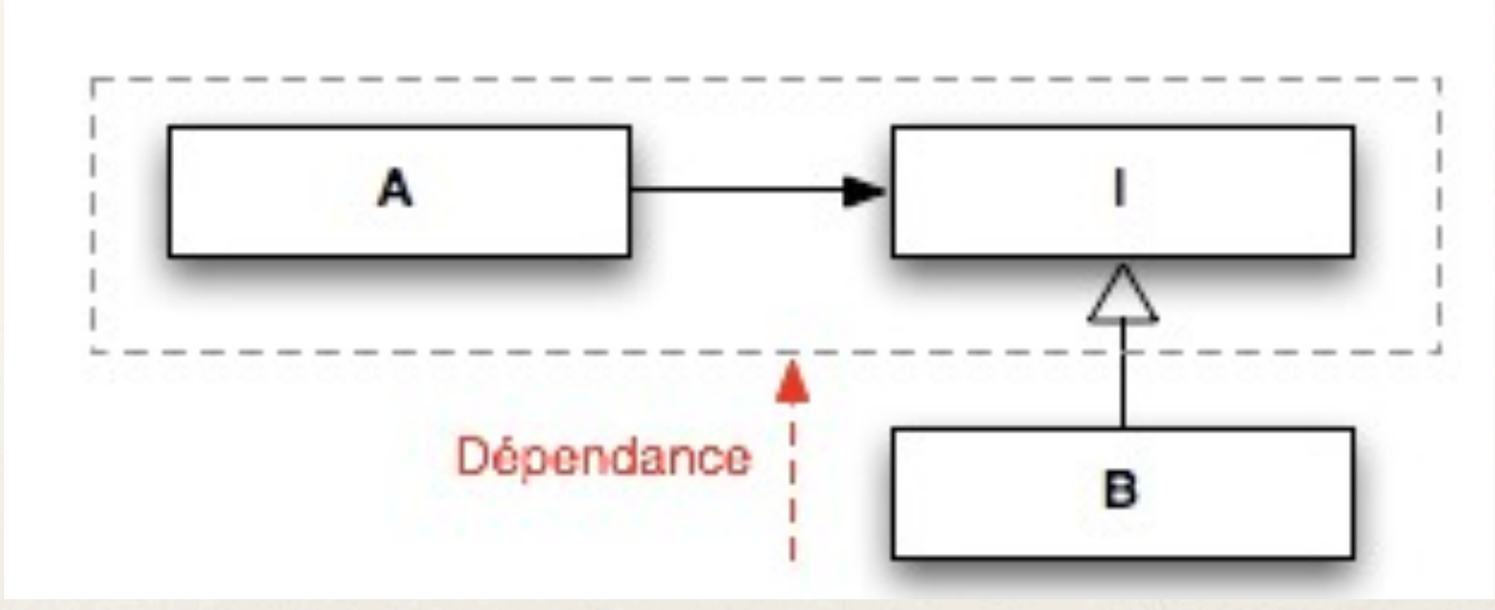
Inversion de dépendance

- 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 (concept) 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





Design to an Interface

- **Abstract classes/interfaces:**
  - tend to change much less frequently
  - abstractions are 'hinge points' where it is easier to extend/modify
  - shouldn't have to modify classes/interfaces that represent the abstraction (OCP)

- Exceptions
  - Some classes are very unlikely to change;
    - therefore little benefit to inserting abstraction layer
    - Example: String class
  - In cases like this can use concrete class directly
    - as in Java or C++

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

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»

Premature Optimization

```php
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!

http://nikic.github.io/2011/12/27/Dont-be-STUPID-GRASP-SOLID.html

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»

«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...

http://www.jhackens.com/2006/10/jardinier/

«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

http://www.jhackens.com/2006/10/jardinier/

«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.

Et si no… moustiq…

A la question d'un journaliste « N'est-ce pas bizarre de passer de Windows 8 à Windows 10 ? », Microsoft a répondu : « Si vous regardez ce produit dans son intégralité, vous conviendrez que c'est un nom plus approprié. » L'appellation Windows 10 donne de « l'importance du chiffre 1, qui montre à la fois l'unicité de la plate-forme, du développement applicatif et de la boutique d'application », explique Nicolas Petit, directeur de la division « marketing et operations » chez Microsoft France.

A Microsoft developer explains in a Reddit thread why Windows 10 is called Windows 10.

Répondre · Retweeter · Favori · Plus

Microsoft dev here, the internal rumours are that early testing revealed just how many third party products that had code of the form

```
if(version.StartsWith("Windows 9"))
{ /* 95 and 98 */
} else {
```

and that this was the pragmatic solution to avoid that.

About 4,342 results

WindowsAttachProvider.java in jdk-6 (https://bitbucket.org/nkabir/jdk-6)

```
40.         String os = System.getProperty("os.name");
41.         if (os.startsWith("Windows 9") || os.equals("Windows Me")) {
42.             throw new RuntimeException(
```

MaxPathLength.java in ManagedRuntimeInitiative (git://github.com/GregBowyer/ManagedRuntimeInitiative.git) Show 2 matches

```
46.             isWindows = true;
47.             if (osName.startsWith("Windows 9") ||
48.                 osName.startsWith("Windows Me"))
```

Selon ce développeur Microsoft, bon nombre de codes sources vérifient une version de Windows 95 ou 98 en testant "en dur" une chaîne de caractère qui commen par `"Windows 9"`.

Effectivement, si l'on utilise par exemple le moteur de recherche searchcode la chaîne `"if(version,startswith("windows 9")"` qui correspond en Java à la reche de `"Windows 9"`dans le début de la chaîne de caractères de la version, on obtient un certain nombre de résultats dans les codes open source qui n'augurent rien d bon pour l'ensemble des codes de la planète :

https://searchcode.com/?q=if%28version%2Cstartswith%28%22windows+9%22%29

```
49.                 || osName.equals("Windows ME"))
```


http://www.zdnet.fr/actualites/pourquoi-windows-10-et-pas-windows-9-39807217.htm