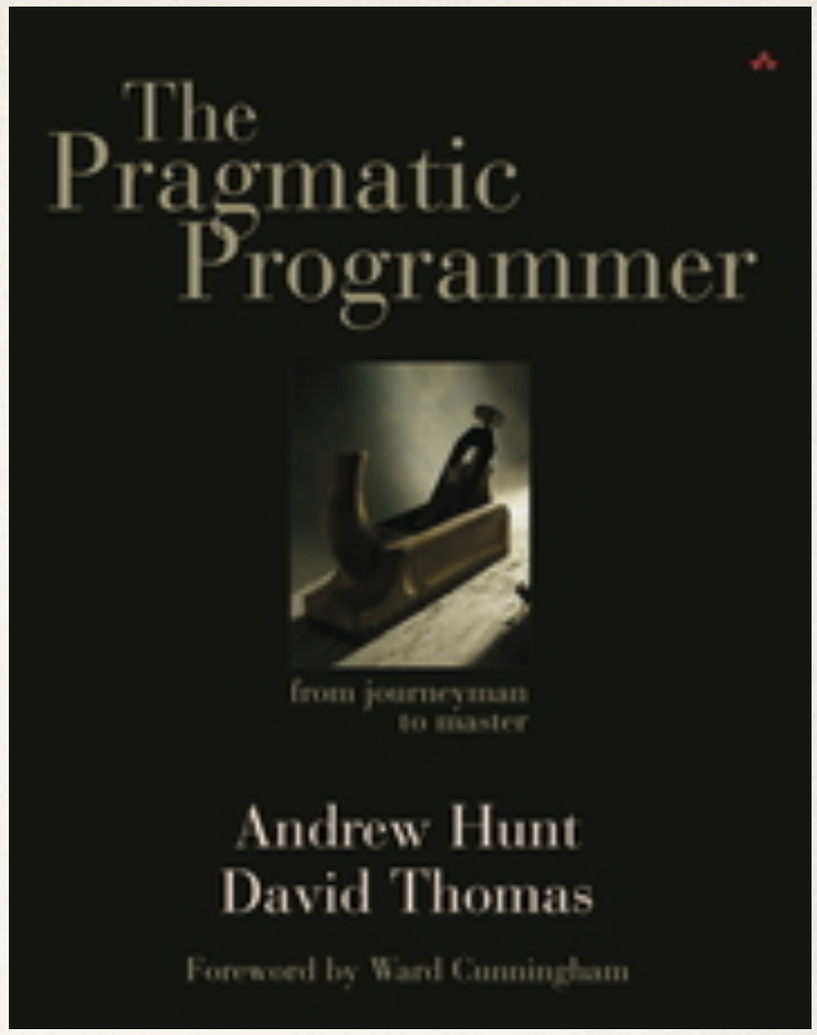


Principes SOLID suivi de Pragmatic Programming

The Pragmatic Programmer: From Journeyman to Master

by Andrew Hunt and David Thomas

Mireille Blay-Fornarino, Université Nice Sophia Antipolis, Département Info Iut, Octobre 2017

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.

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 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

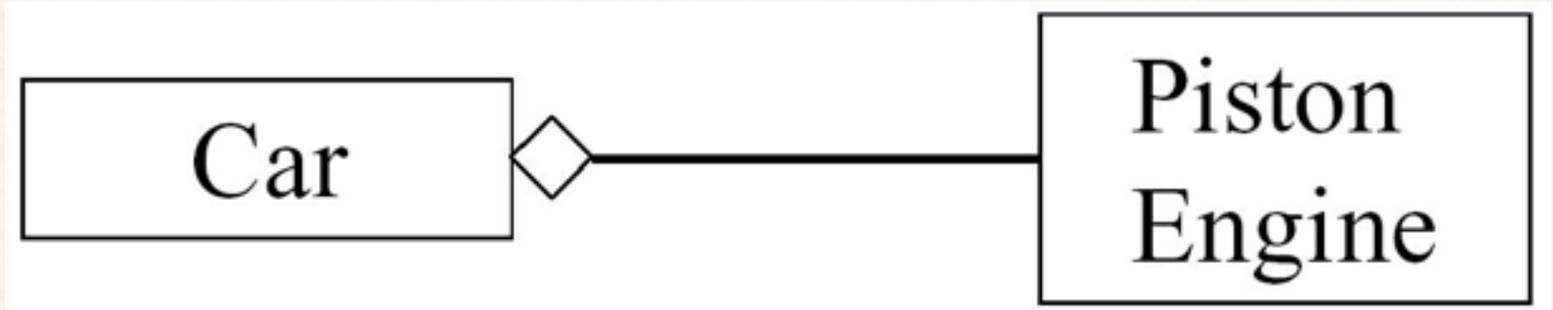
Principe ouvert / fermé
Open/Closed Principle (OCP)

> 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.

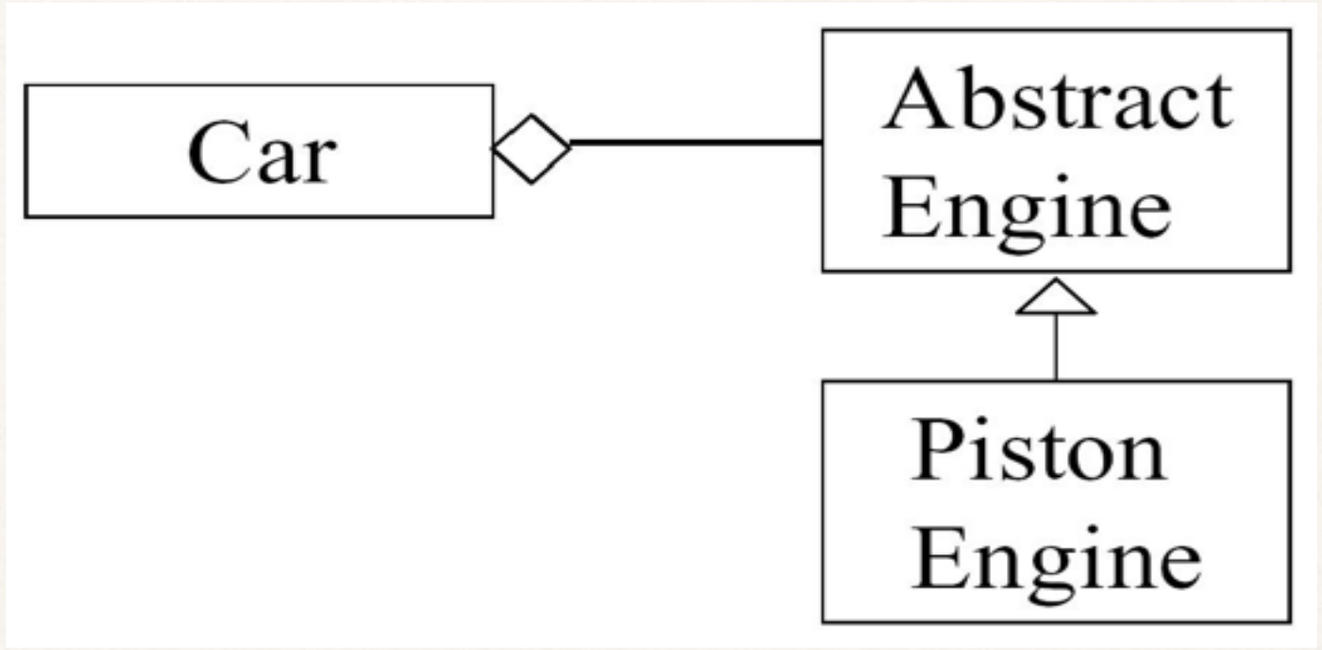
http://afsy.fr/avent/2013/02-principes-stupid-solid-poo

Open the door …



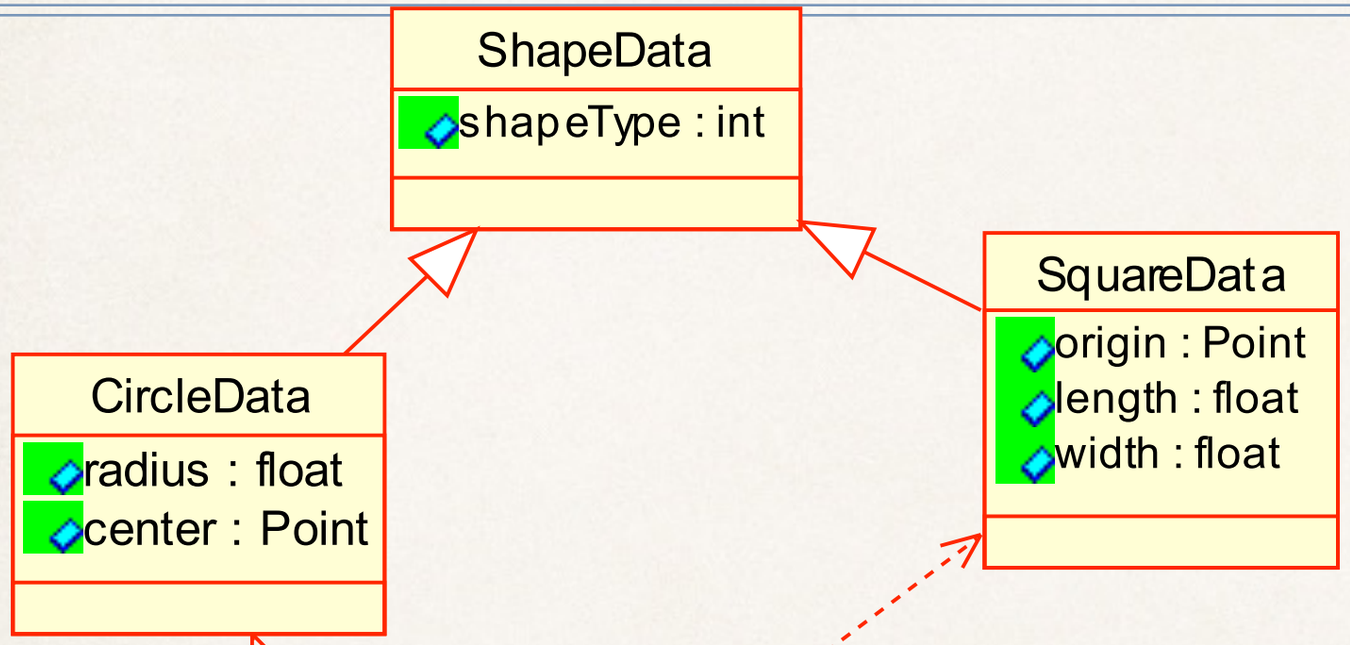
- Comment faire en sorte que la voiture aille plus vite à l'aide d'un turbo?
  - Il faut changer la voiture
    - avec la conception actuelle…

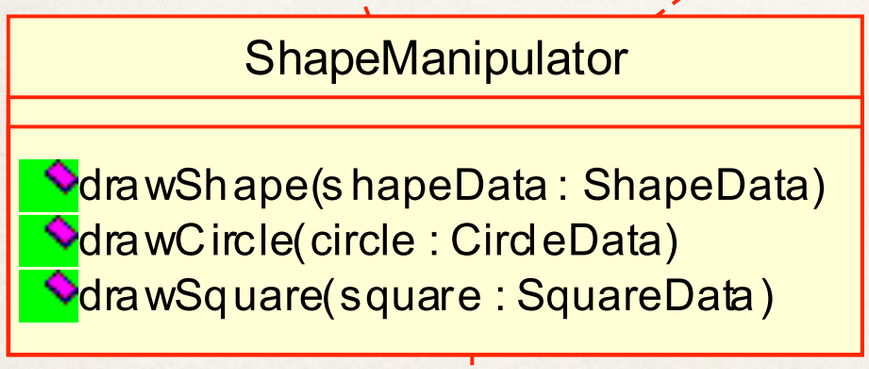
… But Keep It Closed!



- 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



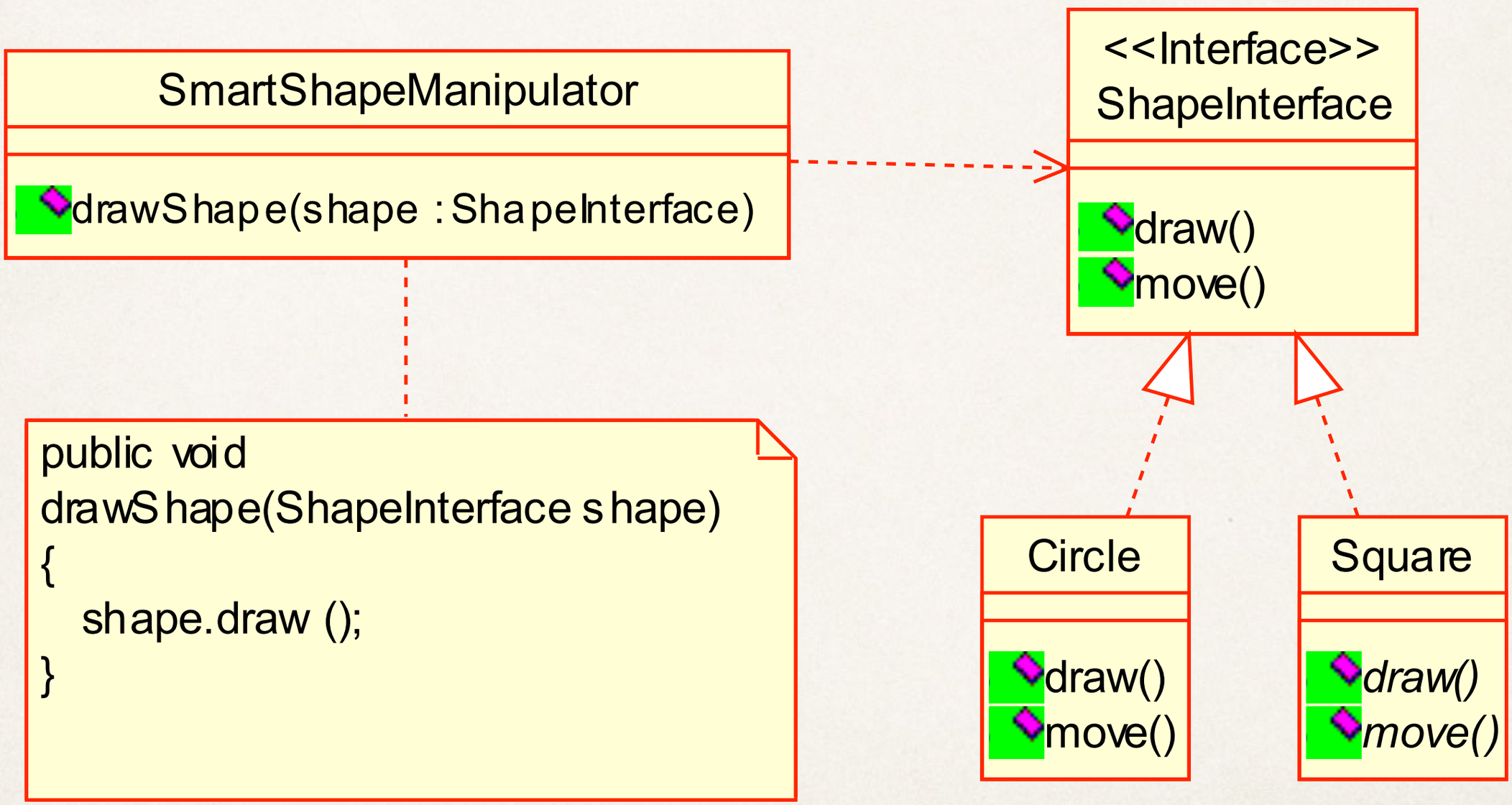



```
public void drawShape(ShapeData shapeData)
{
 switch (shapeData.shapeType)
 {
        case SQUARE:
                drawSquare((SquareData)shapeData);
                break;
        case CIRCLE:
                drawCircle((CircleData)shapeData);
                break;
 }
}
```

An Example of What not to do!

What is wrong with this code?

The Open/Closed Principle (OCP) Example



ATTENTION UML à REVOIR…..

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.

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

```
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]);
```

The single-responsibility principle

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

Comparable…

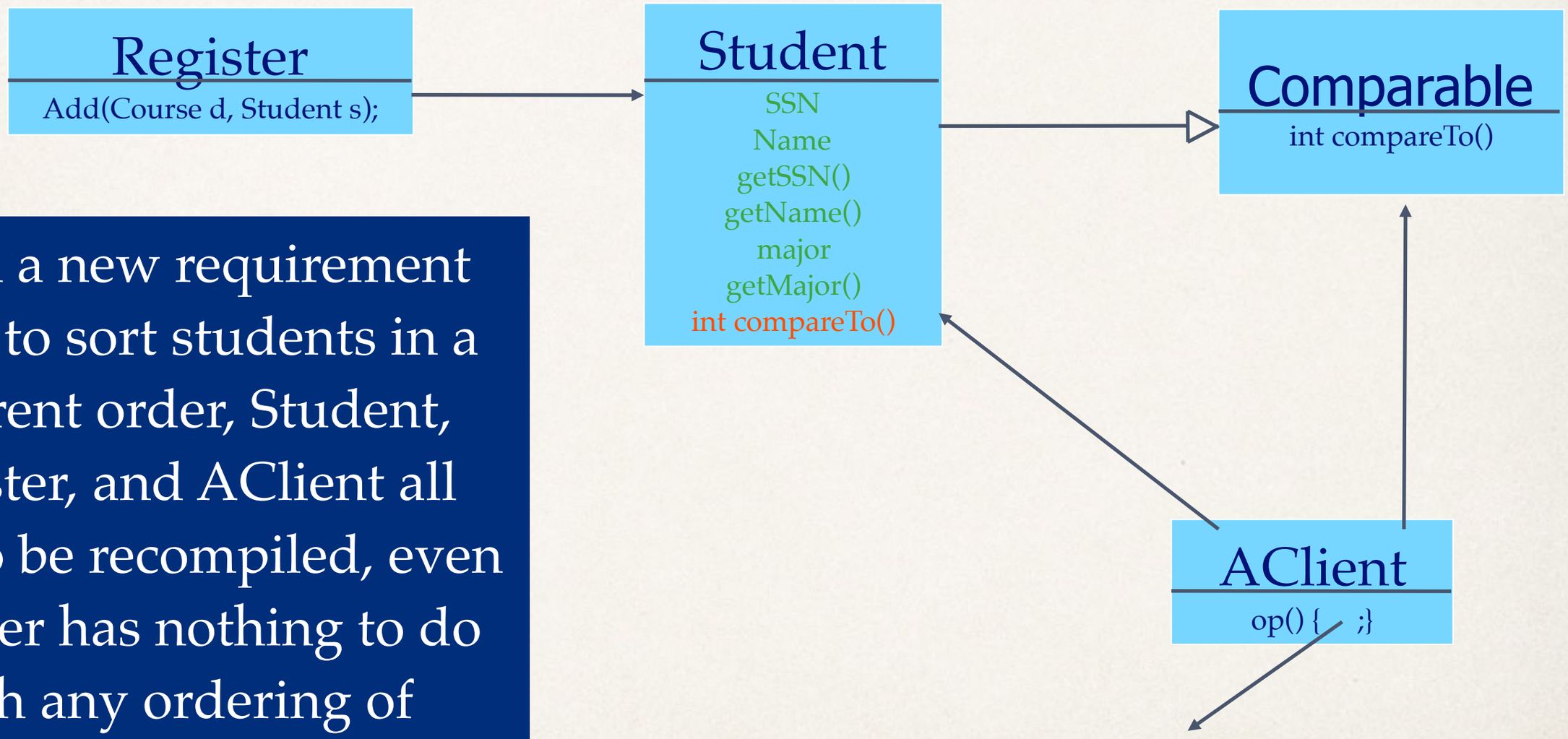
```java
public class Student implements Comparable<Student>{

    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.

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:

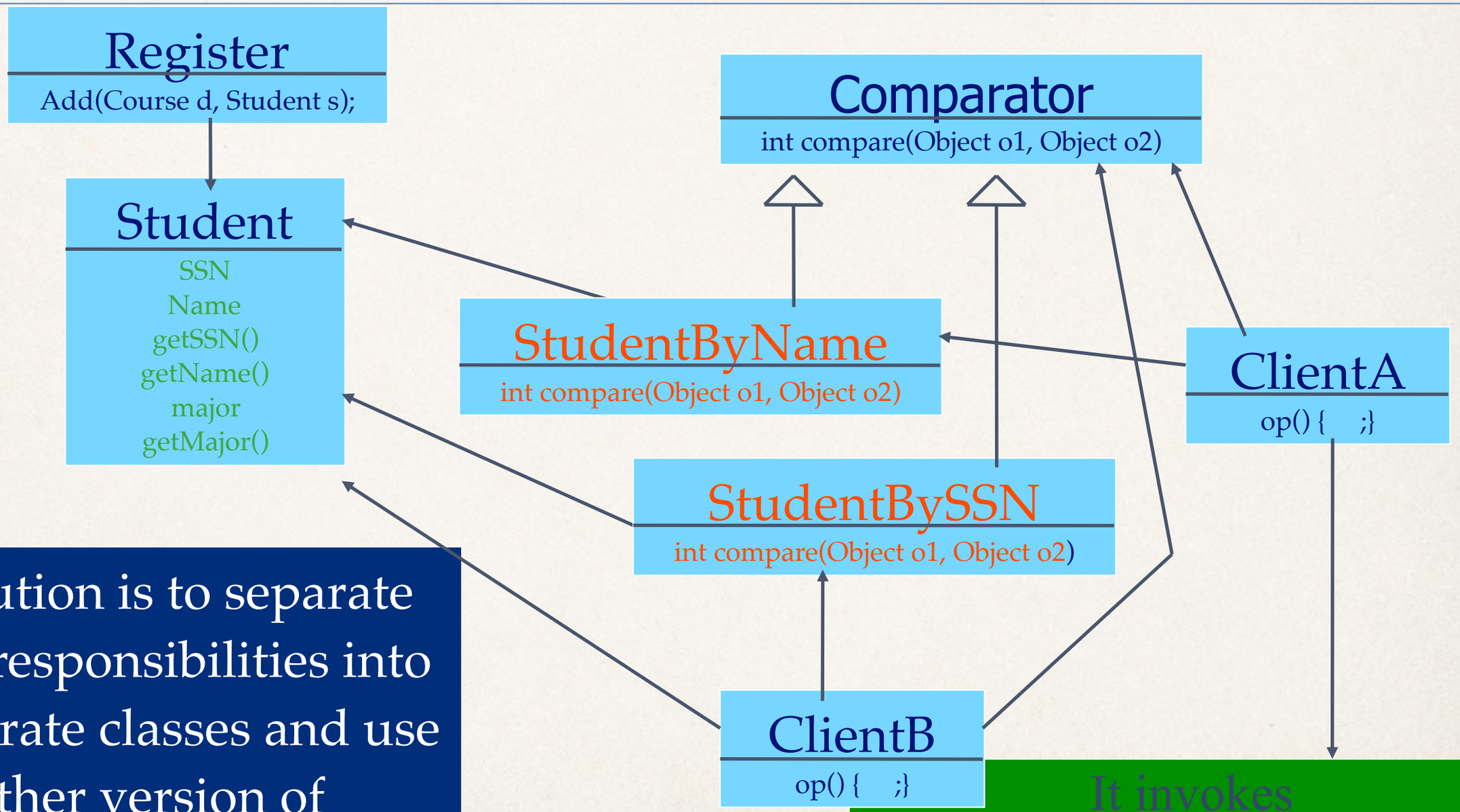
- Student is a business entity, it does not know in what order it should be sorted sinc the order of sorting is imposed by the client of Student.
- Worse: every time students need to be ordered differently, we have to recompile Student and all its client.
- **Cause of the problems:** we bundled **two 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 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.

```
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]);
```