LP IDSE - GL

# SOLID Principles

Based on Mireille Blay & Simon Urli courses

29/11/2016

Cécile Camillieri

Nice Côte d'Azur

Université Nice Sophia Antipolis

# S.O.L.I.D

- **Single Responsibility Principle (SRP):** a class has only one responsibility (or concern)
- **Open/Closed Principle (OCP):** a class should be open for extension (by inheritance for example) but closed for modification (ex: private attributes)
- **Liskov Substitution Principle (LSP):** objects of a program can be replaced by their subtypes without “breaking” the system.
- **Interface Segregation Principle (ISP):** several specific interfaces are better than a single generic interface.
- **Dependency Inversion Principle (DIP):** you must depend on abstractions, not concrete implementations.

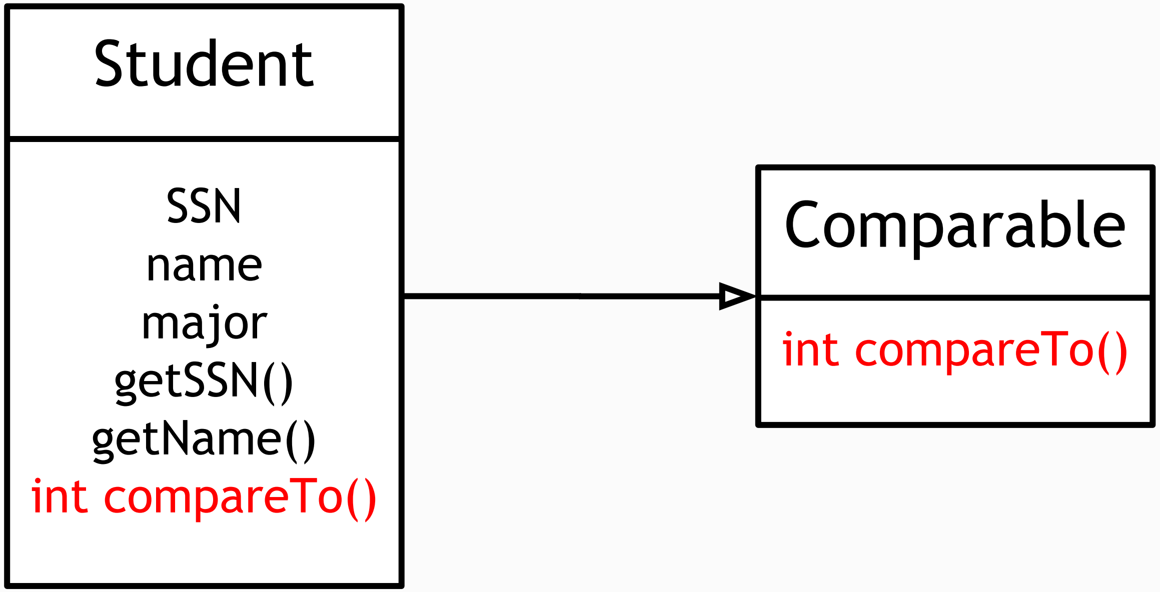
# S.O.L.I.D

- **Single Responsibility Principle (SRP):** a class has only one responsibility (or concern)
- **Open/Closed Principle (OCP):** a class should be open for extension (by inheritance for example) but closed for modification (ex: private attributes)
- **Liskov Substitution Principle (LSP):** objects of a program can be replaced by their subtypes without “breaking” the system.
- **Interface Segregation Principle (ISP):** several specific interfaces are better than a single generic interface.
- **Dependency Inversion Principle (DIP):** you must depend on abstractions, not concrete implementations.

# Single Responsibility Principle

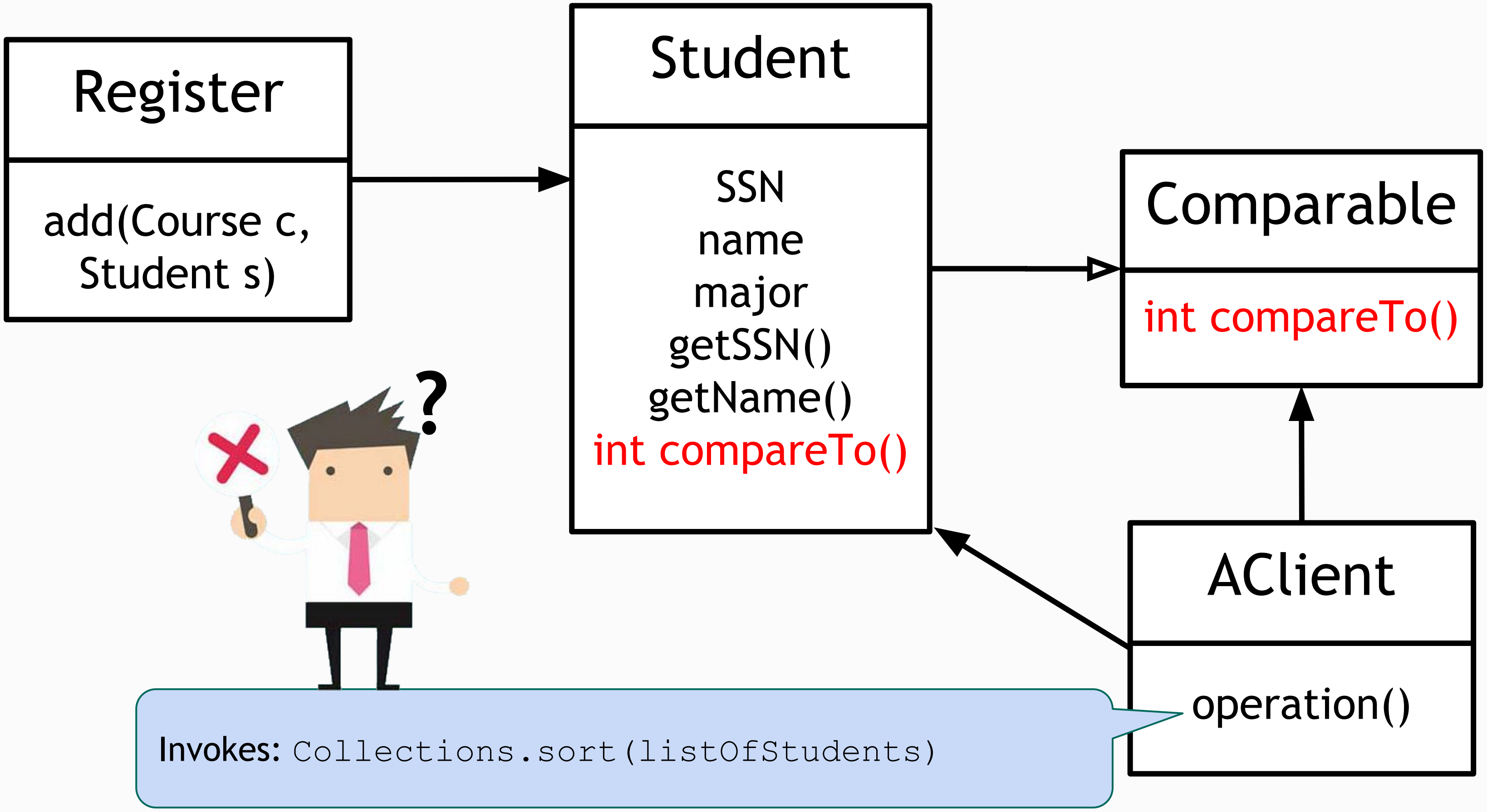
- We often need to sort students by name or SSN.
- So one may implement the Java Comparable Interface

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



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

# Single Responsibility Principle



Register

add(Course c, Student s)

Student

SSN
name
major
getSSN()
getName()
int compareTo()

Comparable

int compareTo()

AClient

operation()

Invokes: `Collections.sort(listOfStudents)`

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

# Single Responsibility Principle

- We often need to sort students by name or SSN.
- So one may implement the Java Comparable Interface

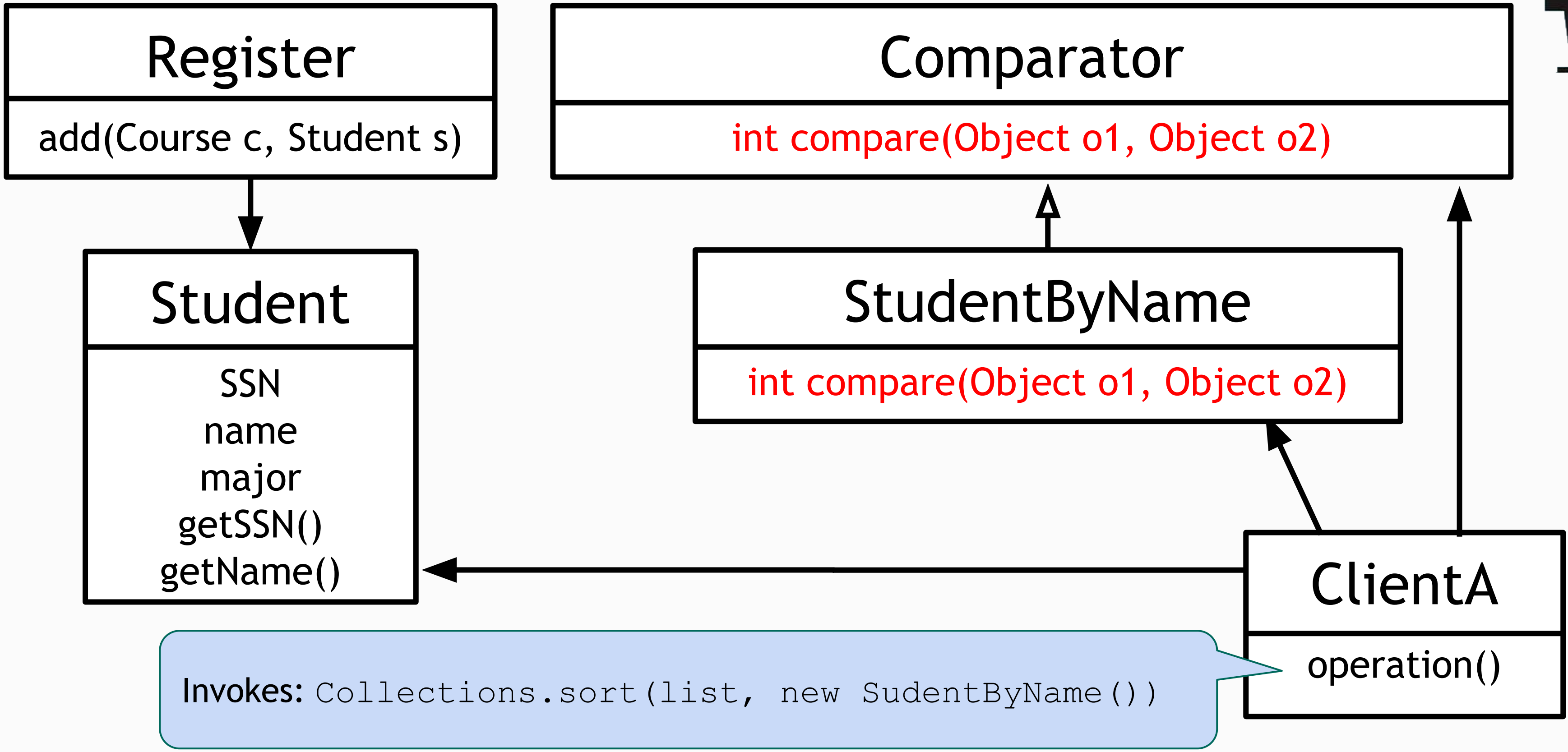
```
class Student implements Comparable {
    int compareTo(Object o) { … }
};
```

**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

# Single Responsibility Principle



Register

add(Course c, Student s)

Comparator

int compare(Object o1, Object o2)

Student

SSN
name
major
getSSN()
getName()

StudentByName

int compare(Object o1, Object o2)

ClientA

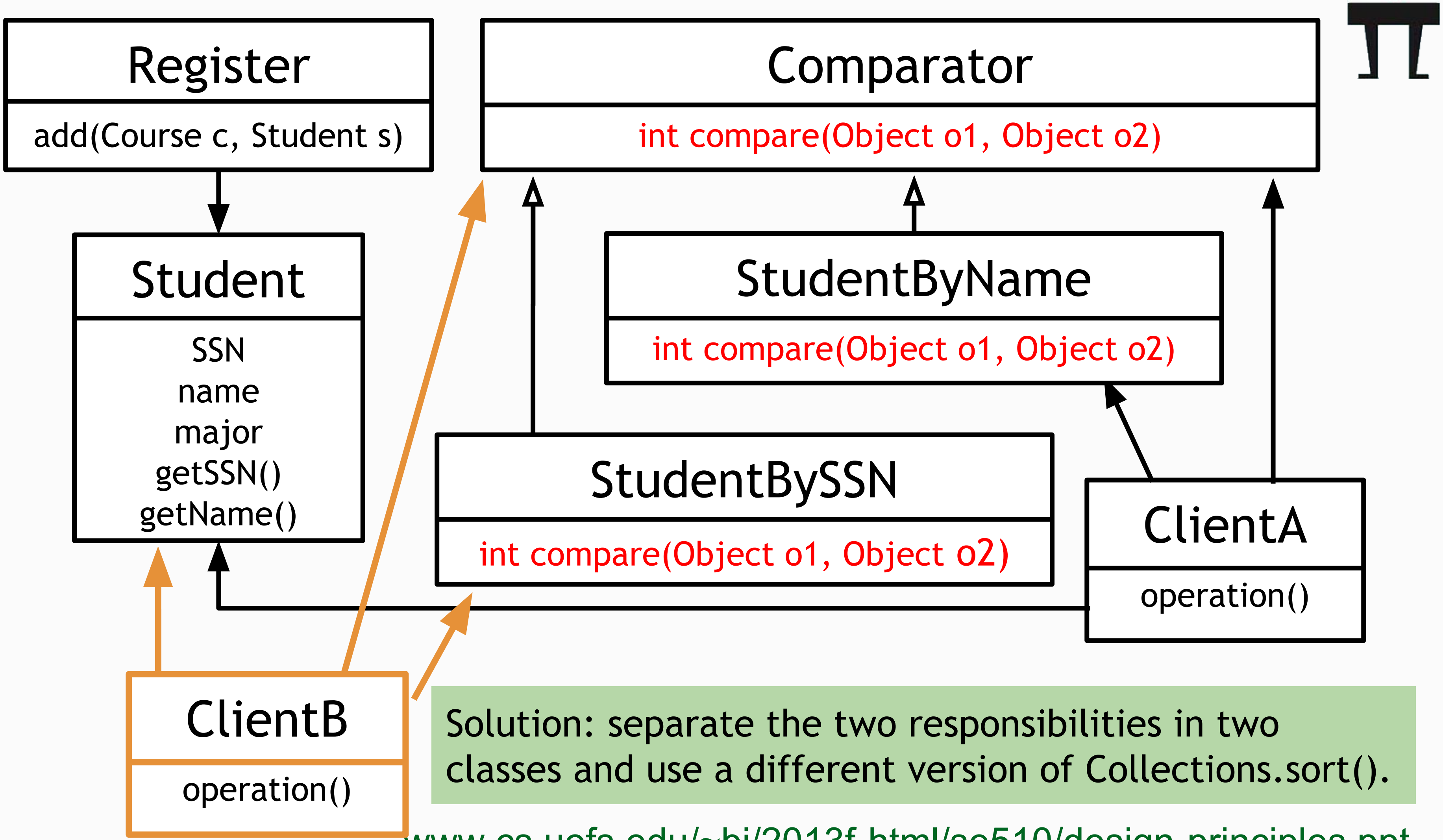
operation()

Invokes: `Collections.sort(list, new SudentByName())`

Solution: separate the two responsibilities in two classes and use a different version of Collections.sort().

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

# Single Responsibility Principle



Solution: separate the two responsibilities in two classes and use a different version of Collections.sort().

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

# S.O.L.I.D

- **Single Responsibility Principle (SRP):** a class has only one responsibility (or concern)
- **Open/Closed Principle (OCP):** a class should be open for extension (by inheritance for example) but closed for modification (ex: private attributes)
- **Liskov Substitution Principle (LSP):** objects of a program can be replaced by their subtypes without “breaking” the system.
- **Interface Segregation Principle (ISP):** several specific interfaces are better than a single generic interface.
- **Dependency Inversion Principle (DIP):** you must depend on abstractions, not concrete implementations.

# Open/Closed Principle

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

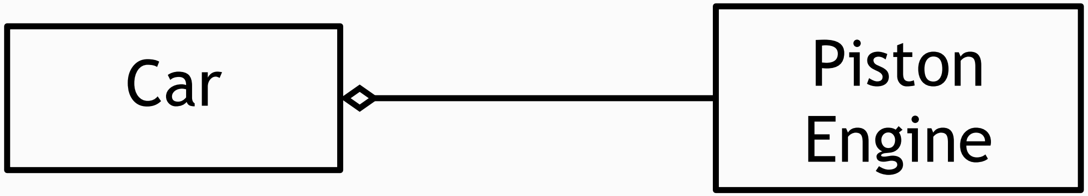
- Software entities must be **open to extension**

  => Code is extensible

- But **closed to modifications**

  => Code has been written, tested, we won’t touch it anymore

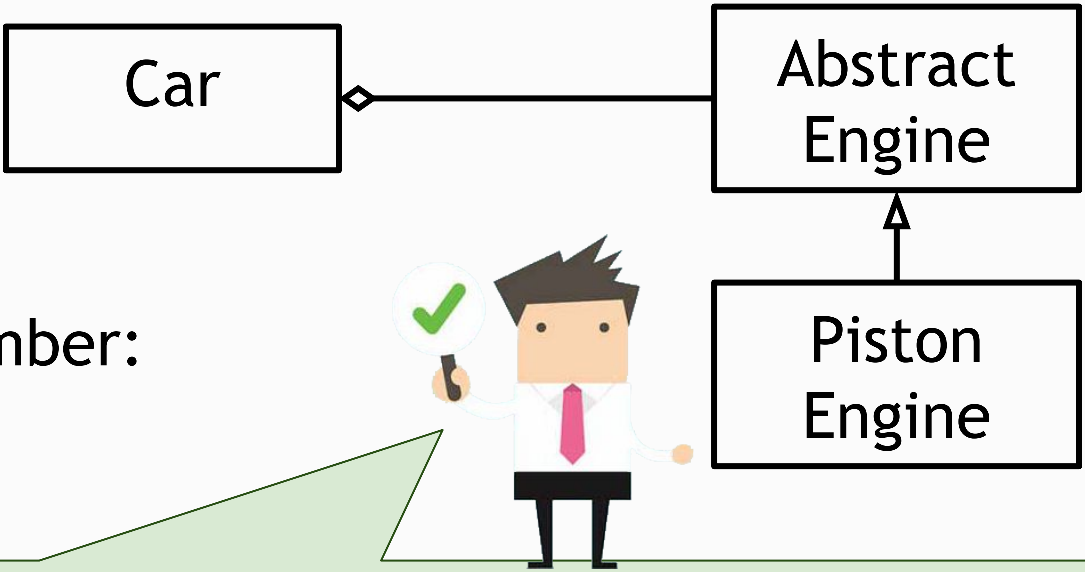
# Open the door...



- How do we make the car faster?

With the current conception, we have to change the car...

# … But keep it closed!



- Remember:

- A class **must not** depend on another concrete class.
  => Depend on an abstract class…
  and use polymorphism.

# S.O.L.I.D

- **Single Responsibility Principle (SRP):** a class has only one responsibility (or concern)
- **Open/Closed Principle (OCP):** a class should be open for extension (by inheritance for example) but closed for modification (ex: private attributes)
- **Liskov Substitution Principle (LSP):** objects of a program can be replaced by their subtypes without “breaking” the system.
- **Interface Segregation Principle (ISP):** several specific interfaces are better than a single generic interface.
- **Dependency Inversion Principle (DIP):** you must depend on abstractions, not concrete implementations.

# Liskov Substitution Principle

- Instances of a class should be replaceable by instances of their subclasses without breaking the program.

- If a property P is true for an instance of type T, P must stay true for any instance y of a subtype of T.

- The contract of a class must be respected by its subclasses
- The caller does not need to know the exact class it is using: any derived class can be substituted to the one used.

# Liskov Substitution Principle

- Instances of a class should be replaceable by instances of their subclasses without breaking the program.

- If a property P is true for an instance of type T, P must stay true for any instance y of a subtype of T.

- This is a basic property of polymorphism:
  - If we substitute a class by another derived class from the same hierarchy, behavior is (of course) different, but follows the same rules.

# 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
```

# But penguins fail to fly !

```
class Penguin : public Bird {
   public: void fly() {
       error (“Penguins don’t fly!”); }
};
```

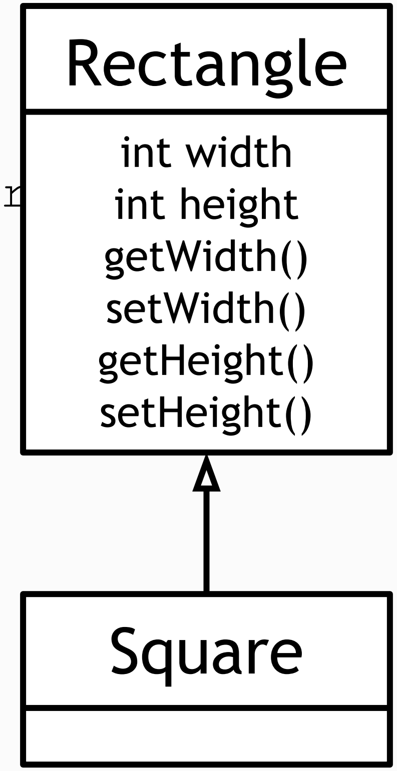
- This does not model “Penguins can’t fly”
- It models: “Penguind may fly, but if they try it is an error”
- Run-time error if attempt to fly -> not desirable

Think about sustainabilty - The contract is broken

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

# LSP: Another counter-example

```
class LspTest {
    private static Rectangle getNewRectangle(){
        // it can be an object returned by some factor
        return new Square();
    }

    public static void main (String args[]){
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);
}}
```



Rectangle

int width
int height
getWidth()
setWidth()
getHeight()
setHeight()

Square

- User knows the object is a rectangle
- She assumes that the area will be 5x10 = 50
- But it is 100 !

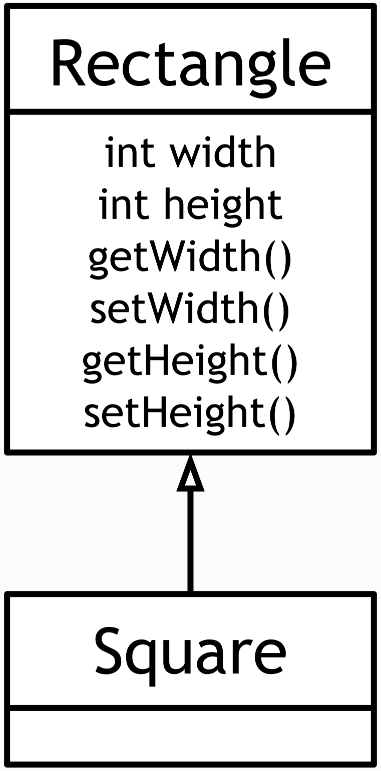
# Liskov Substitution Principle

- Solution: Square should not be a subclass of Rectangle but a completely independant class.

This does not change the fact that a square is a rectangle !

- Square **represents** the concept of a square
- Rectangle **represents** the concept of a rectangle
- But a representation **does not** share the same properties of what it represents!

=> Good code does not mean following exactly real life.



# S.O.L.I.D

- **Single Responsibility Principle (SRP):** a class has only one responsibility (or concern)
- **Open/Closed Principle (OCP):** a class should be open for extension (by inheritance for example) but closed for modification (ex: private attributes)
- **Liskov Substitution Principle (LSP):** objects of a program can be replaced by their subtypes without “breaking” the system.
- **Interface Segregation Principle (ISP):** several specific interfaces are better than a single generic interface.
- **Dependency Inversion Principle (DIP):** you must depend on abstractions, not concrete implementations.

# Interface Segregation Principle

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

- An *interface* is the set of methods one object knows it can invoke on another object.
- A class can implement many interfaces (an interface is a subset of all the methods 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();
}

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

# Interface example

- We can maneuver the vehicle without being concerned about the actual class (car, boat, submarine) or its inheritance hierarchy.

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

# ISP example: Timed door

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

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

**CSE 403, Spring 2008, Alverson**

# ISP example: Timed door

duration before timeout

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

object to invoke TimeOut() on when a timeout occurs

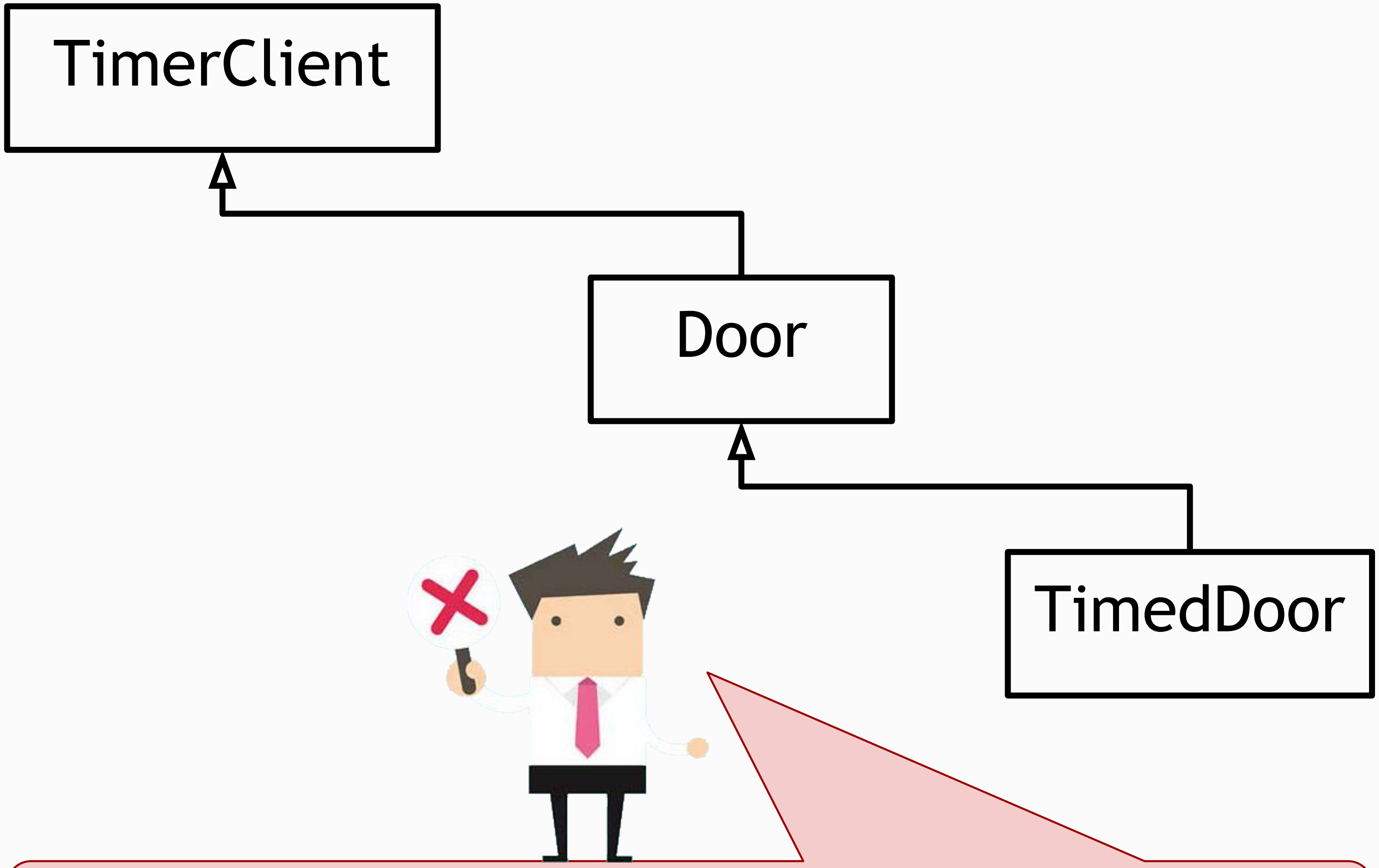
timeout method

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

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

CSE 403, Spring 2008, Alverson

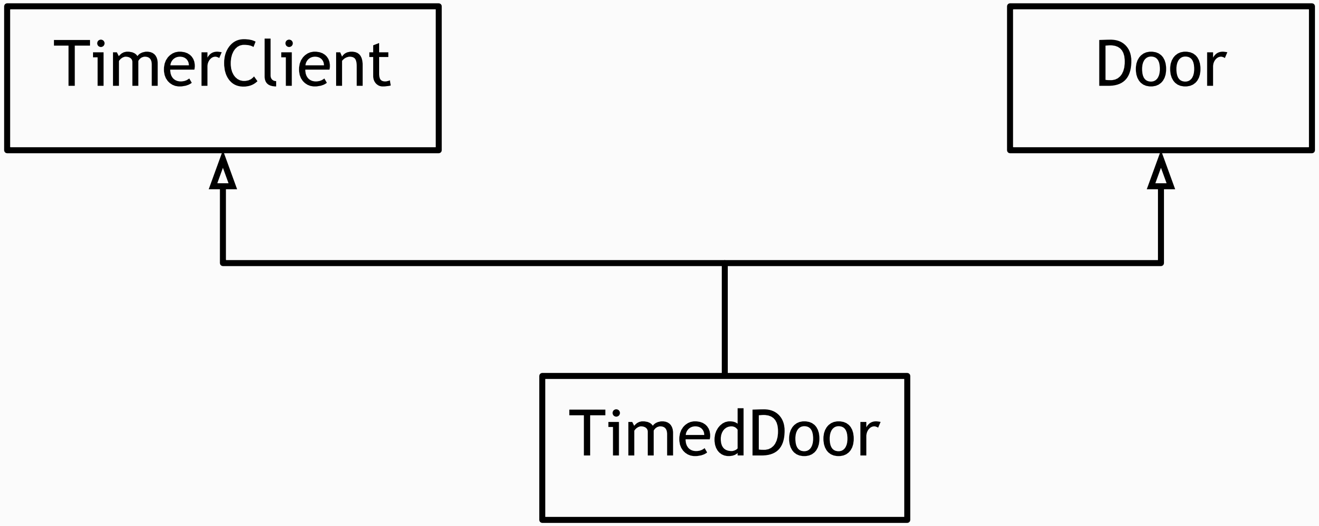
# Timed Door Solution: Yes or No?



No, it is polluting the Door interface by requiring all doors to have a TimeOut() method.

CSE 403, Spring 2008, Alverson

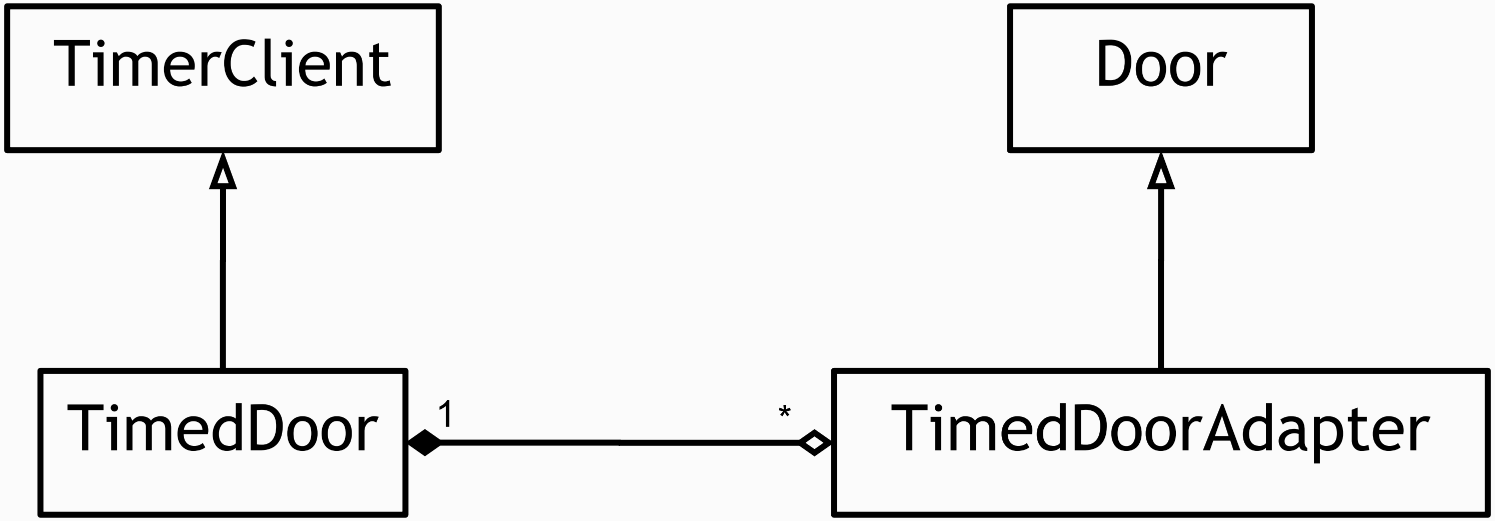
# Timed Door Solution: Yes or No?



Yes, separation through multiple inheritance

CSE 403, Spring 2008, Alverson

# Timed Door Solution: Yes or No?



TimerClient

Door

TimedDoor

1

*

TimedDoorAdapter

Yes, separation through multiple inheritance

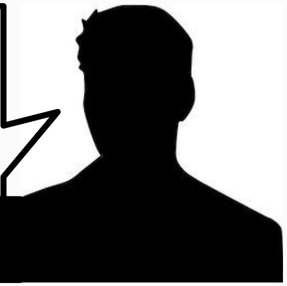
CSE 403, Spring 2008, Alverson

# S.O.L.I.D

- **Single Responsibility Principle (SRP):** a class has only one responsibility (or concern)
- **Open/Closed Principle (OCP):** a class should be open for extension (by inheritance for example) but closed for modification (ex: private attributes)
- **Liskov Substitution Principle (LSP):** objects of a program can be replaced by their subtypes without “breaking” the system.
- **Interface Segregation Principle (ISP):** several specific interfaces are better than a single generic interface.
- **Dependency Inversion Principle (DIP):** you must depend on abstractions, not concrete implementations.

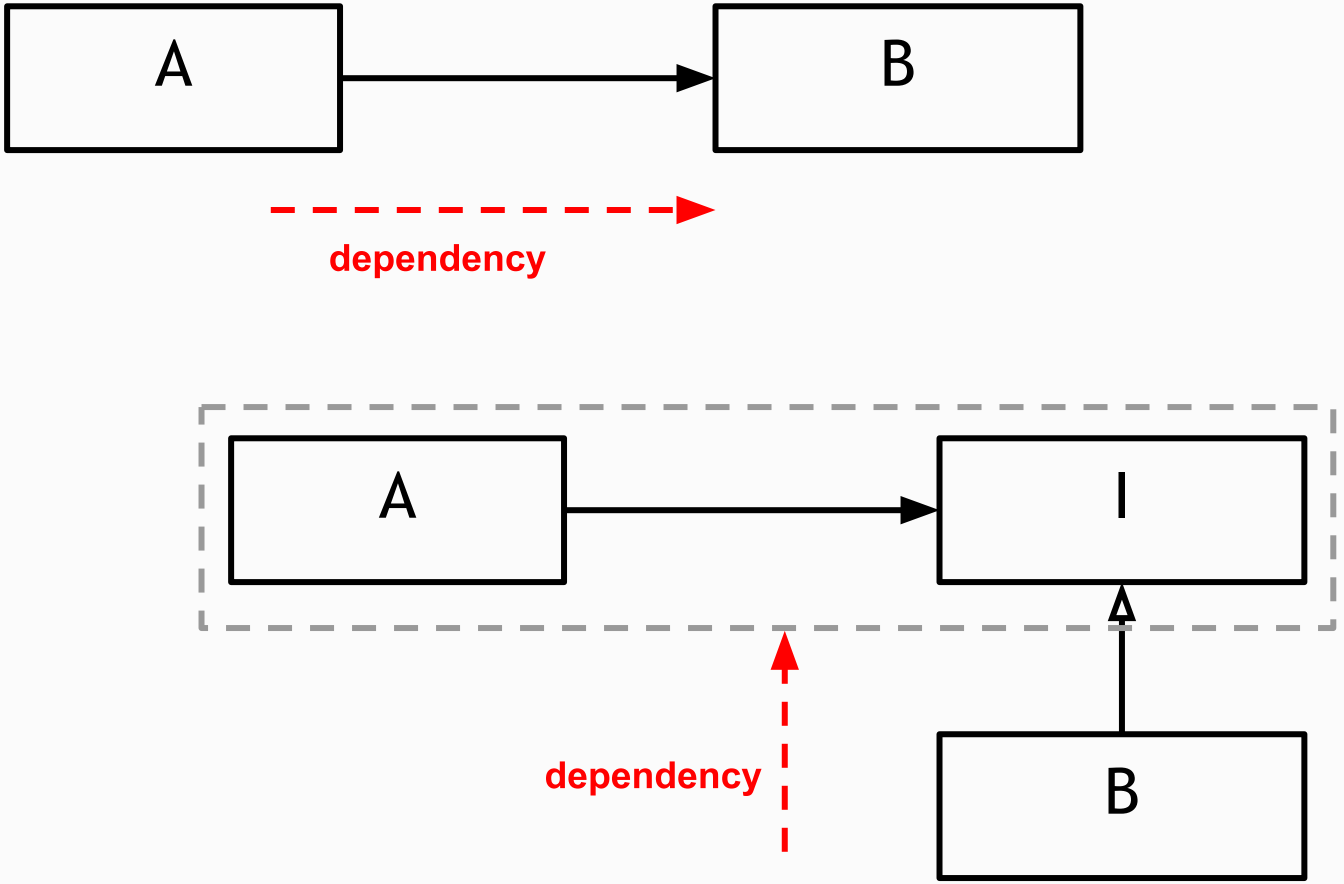
# Dependency Inversion Principle

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

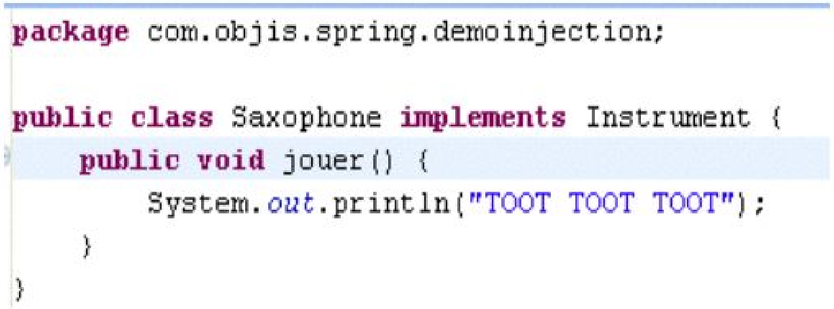


- Reduce dependencies on concrete classes
- “ Program to interface, not implementation”
- Abstractions must not depend on details.
  - Details should depend on abstractions.
- ONLY depend on abstractions, even for low level classes.
- Allows the Open/Closed Principle when DIP is the technique

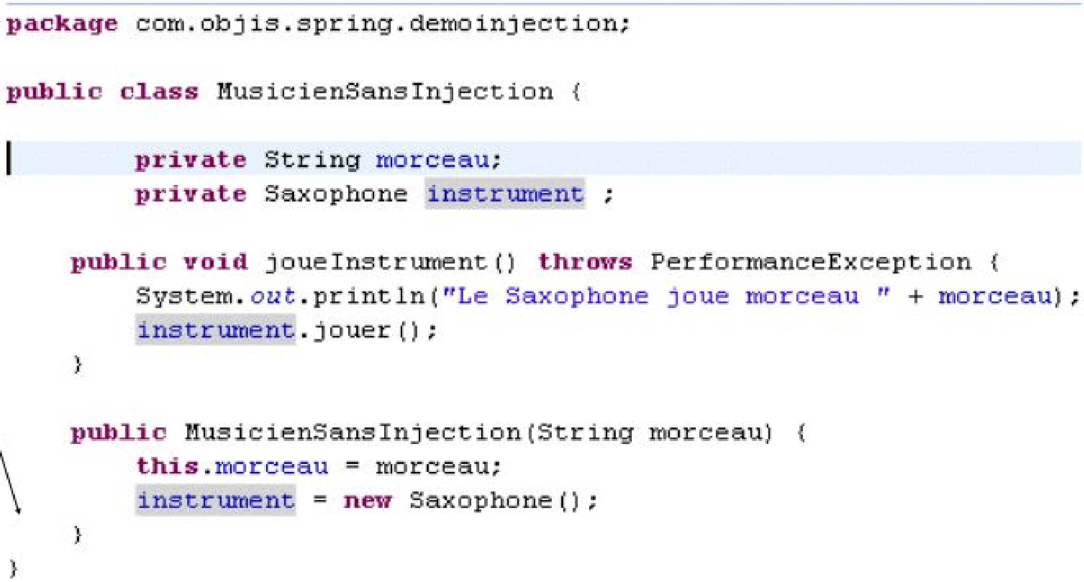
# Dependency Inversion



# Exemple de couplage fort

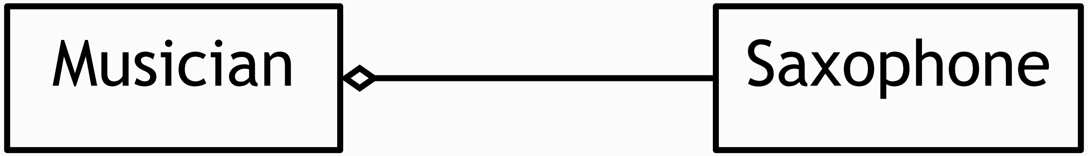


```
package com.objis.spring.demoinjection;

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



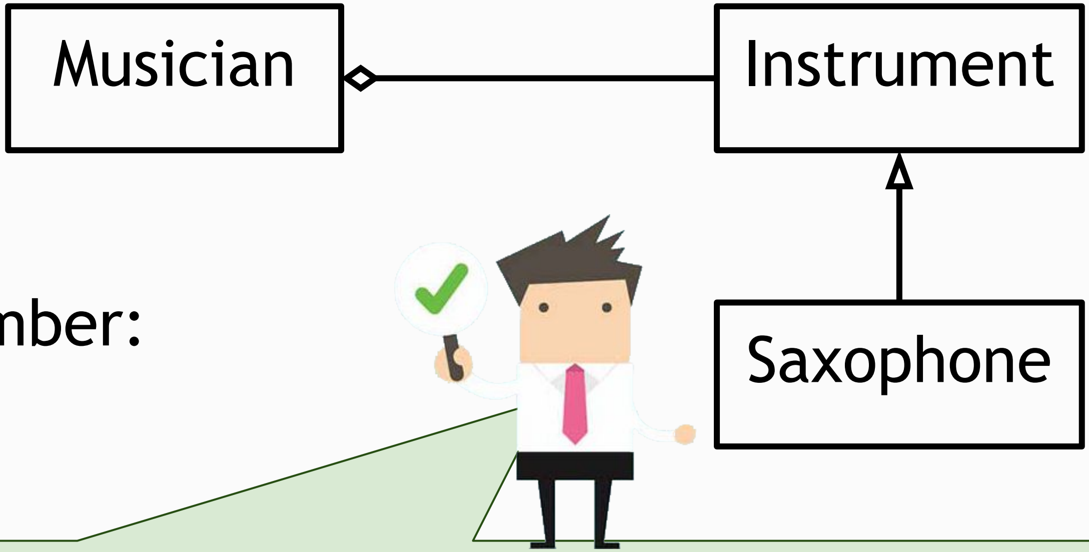
```
package com.objis.spring.demoinjection;

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();
    }
}
```

# Problems with strong coupling



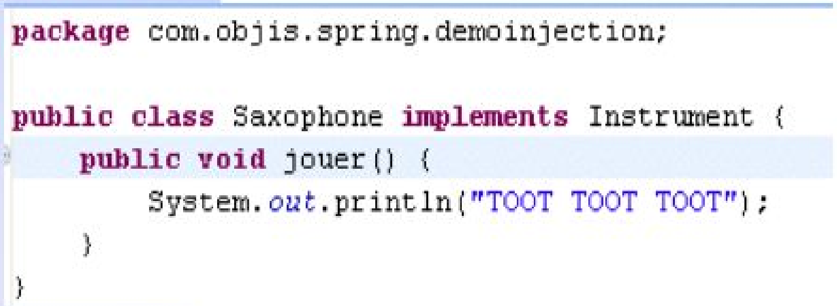
- Hard to test the Musician class
- Hard to reuse the Musician class

# Loosen the coupling !



- Remember:
  - Mask the implementation with an interface
  - This creates a loose coupling between the calling object and the called object. They do not need to know each other.

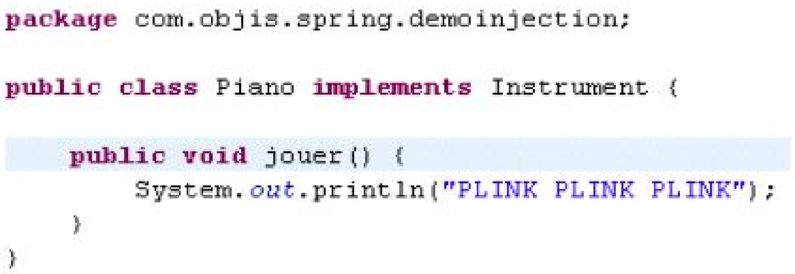
# Exemple de couplage faible



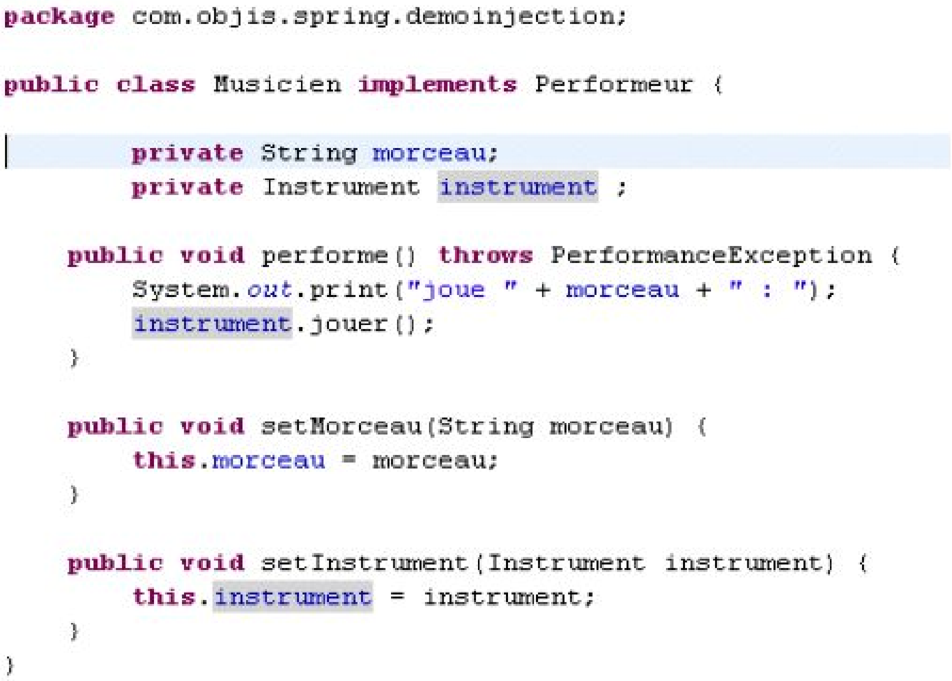
```java
package com.objis.spring.demoinjection;

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

Ici les classes sont indépendantes.

Couplage faible



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

public class Piano implements Instrument {

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



```java
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;
    }
}
```

# Sum up

# S.O.L.I.D

- **Single Responsibility Principle (SRP):** a class has only one responsibility (or concern)
- **Open/Closed Principle (OCP):** a class should be open for extension (by inheritance for example) but closed for modification (ex: private attributes)
- **Liskov Substitution Principle (LSP):** objects of a program can be replaced by their subtypes without “breaking” the system.
- **Interface Segregation Principle (ISP):** several specific interfaces are better than a single generic interface.
- **Dependency Inversion Principle (DIP):** you must depend on abstractions, not concrete implementations.

# ?