Principles of Object-Oriented Design

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

SOLID principles of object-oriented programming.

SOLID principles of object-oriented programming.

Single responsibility principle

SOLID principles of object-oriented programming.

Single responsibility principle

Open-closed principle

SOLID principles of object-oriented programming.

Single responsibility principle

Open-closed principle

Liskov substitution principle

SOLID principles of object-oriented programming.

Single responsibility principle

Open-closed principle

Liskov substitution principle

Interface segregation principle

SOLID principles of object-oriented programming.

Single responsibility principle

Open-closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

One should depend upon abstractions, not concretions!

Single responsibility principle

A class should have only a single responsibility!

Single responsibility principle

A class should have only a single responsibility!

A **responsibility** is a family of functions that serves one particular **actor**.

Single responsibility principle

A class should have only a single responsibility!

A **responsibility** is a family of functions that serves one particular **actor**.

An **actor** for a responsibility is the single **source of change** for that responsibility.

Single responsibility principle

Bad example:

```
public class Order {
    ...
    public int getId() {...}

    public String getDescription {...}

    public duble getPrice() {...}

    public void printPage() {
        ...
    }
    ...
}
```

Single responsibility principle

Correct refactoring:

```
public class Order {
    ...
    public int getId() {...}

    public String getDescription {...}

    public duble getPrice() {...}
    ...
}

public interface OrderPrinter {
  public void print(Order o);
}
```

Single responsibility principle

```
public class PlainTextPrinter implements OrderPrinter {
   public void print(Order b) { ... }
}

public class HtmlPrinter implements OrderPrinter {
   public void print(Order b) { ... }
}
```

Single responsibility principle

```
class Book {
    public String getTitle() {...}
    public String getAuthor() {...}
    public void turnPage() {...}
    public Page getCurrentPage() {...}
    public Location getLocation() {
        // returns the position in the library
        // ie. shelf number & room number
    }
}
```

Single responsibility principle

```java
class Book {
    public String getTitle() {...}
    public String getAuthor() {...}
    public void turnPage() {...}
    public Page getCurrentPage() {...}
    public Location getLocation() {
        // returns the position in the library
        // ie. shelf number & room number
    }
}
```

Question: does the above class violate the SRP?

Single responsibility principle

When we design a *software solution* we should. . .

1. Find and define the **actors**.
2. Identify the **responsibilities** that serve those actors.
3. Group our functions and classes so that each has only **one allocated responsibility**.

Open-closed principles

Software entities should be open for extension, but closed for modification!

Open-closed principles

**Software entities should be open for extension,
but closed for modification!**

The origin of the term is due to **Bertrand Meyer** who used it in his 1988 book *Object Oriented Software Construction*.

Open-closed principles

Software entities should be open for extension, but closed for modification!

The origin of the term is due to **Bertrand Meyer** who used it in his 1988 book *Object Oriented Software Construction*.

- A module will be said to be **open** if it is still available for extension.
- A module will be said to be **closed** if it is available for use by other modules (well-defined, stable description).

Open-closed principles

**Software entities should be open for extension,
but closed for modification!**

The origin of the term is due to **Bertrand Meyer** who used it in his 1988 book *Object Oriented Software Construction*.

- A module will be said to be **open** if it is still available for extension.
- A module will be said to be **closed** if it is available for use by other modules (well-defined, stable description).

A class is **closed**, since it may be compiled, stored in a library, baselined, and used by client classes.

Open-closed principles

**Software entities should be open for extension,
but closed for modification!**

The origin of the term is due to **Bertrand Meyer** who used it in his 1988 book *Object Oriented Software Construction*.

- A module will be said to be **open** if it is still available for extension.
- A module will be said to be **closed** if it is available for use by other modules (well-defined, stable description).

A class is **closed**, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also **open**, since any new class may use it as parent, adding new features.

Open-closed principles

Bad example:

```java
public class Rectangle {
  private final double width;
  private final double height;

  public Rectangle( double width , double height ) {
    this.width = width;
    this.height = height;
  }

  public double getWidth() {
    return width;
  }

  public double getHeight() {
    return height;
  }
}
```

Open-closed principles

Bad example:

```
public class AreaCalculator {

  public double computeArea( Rectangle[] shapes ) {
    double area = 0;
    for( int i=0 ; i<shapes ; i++ ) {
      area += shapes[i].getWidth()*shapes[i].getHeight();
    }
  }

}
```

Open-closed principles

Correct refactoring:

```
public interface Shape {

  public double getArea();

}

public class AreaCalculator {

  public double computeArea( Shape[] shapes ) {
    double area = 0;
    for( int i=0 ; i<shapes ; i++ ) {
      area += shapes[i].getArea();
    }
  }

}
```

Open-closed principles

Correct refactoring:

```
public class Rectangle implements Shape {
  private final double width;
  private final double height;

  public Rectangle( double width , double height ) {
    this.width = width;
    this.height = height;
  }

  public double getWidth() { return width; }

  public double getHeight() { return height; }

  public double getArea() { return width*height; }
}
```

Open-closed principles

Correct refactoring:

```java
public class Circle implements Shape {
  private final double radius;

  public Circl( double radius ) {
    this.radius = radius;
  }

  public double getRadius() {
    return radius;
  }

  public double getArea() {
    return Math.PI*Math.pow(radius,2);
  }
}
```

Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program!

Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program!

The concept of this principle was introduced by **Barbara Liskov** in a 1987 conference keynote and later published in a paper together with **Jannette Wing** in 1994.

Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program!

The concept of this principle was introduced by **Barbara Liskov** in a 1987 conference keynote and later published in a paper together with **Jannette Wing** in 1994.

Their original definition is as follows:

> Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S where S is a subtype of T.

Liskov substitution principle

Bad example:

```
public class Rectangle implements Shape {
  private double width = 0;
  private double height = 0;

  public double getWidth() { return width; }

  public double getHeight() { return height; }

  public double getArea() { return width*height; }

  public void setWidth(double width) { this.width = width;
 }

  public void setHeight(double height) { this.height =
 height; }
}
```

Liskov substitution principle

Bad example:

```
public class Square extends Rectangle {
  public void setWidth(double width) {
    super.setWidth( width );
    super.setHeight( width );
  }

  public void setHeight(double height) {
    super.setHeight( width );
    super.setWidth( width );
  }
}
```

Question: do Rectangle and Square classes satisfy the Liskov substitution principle?

Liskov substitution principle

Answer: NO!

Liskov substitution principle

Answer: NO!

```
public class Class {

  public void checkArea( Rectangle r ) {
    r.setWidth( 10 );
    r.setHeight( 20 );
    if (r.getArea() != 200) {
      throw new IllegalStateException('Bad area!')
    }
  }

}
```

Liskov substitution principle

Answer: NO!

```
public class Class {

  public void checkArea( Rectangle r ) {
    r.setWidth( 10 );
    r.setHeight( 20 );
    if (r.getArea() != 200) {
      throw new IllegalStateException('Bad area!')
    }
  }

}
```

Solution?

Liskov substitution principle

Answer: NO!

```
public class Class {

  public void checkArea( Rectangle r ) {
    r.setWidth( 10 );
    r.setHeight( 20 );
    if (r.getArea() != 200) {
      throw new IllegalStateException('Bad area!')
    }
  }

}
```

Solution? Square is not a subclass of Rectangle!

Interface segregation principle

Many client-specific interfaces are better than one general-purpose interface!

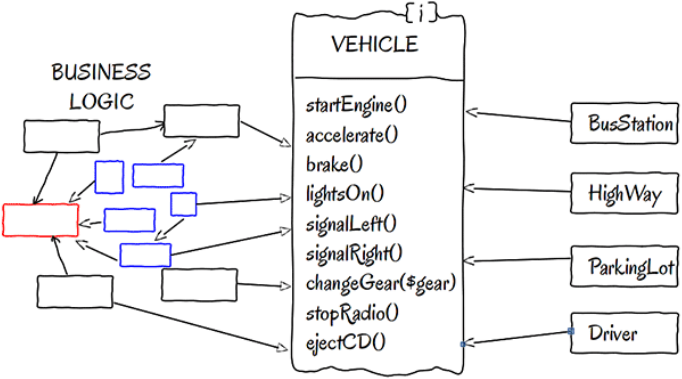
Interface segregation principle

Many client-specific interfaces are better than one general-purpose interface!

The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

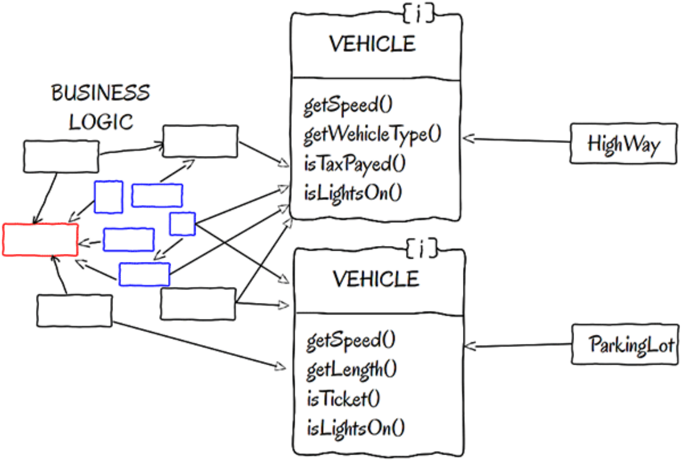
Interface segregation principle

Bad example:



Interface segregation principle

Correct refactoring:



Dependency inversion principle

One should depend upon abstractions, not concretions!

Dependency inversion principle

One should depend upon abstractions, not concretions!

The principle states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

To be continued. . .