Design Principles

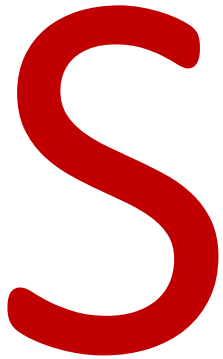
Single responsability principle

Open-closed principle

Liskov's substitution principle

Interface segregation principle

Dependency inversion principle



Single Responsability Principle

Single Responsability Principle

```
public class OrderService {

    public boolean process(Order order) {

        if (order.isValid()) {
            try {
                boolean savedOK = order.save();
                if (savedOK) {

                    String emailAdress = order.getEmailAdress();
                    String firstname = order.getFirstname();
                    String lastname = order.getLastname();

                    Email mail = new Email(emailAdress, firstname, lastname);
                    mail.send();
                    return true;
                }
            } catch (SQLException e) {
                log.error("Error saving", e);
            }
        }
        return false;
    }
}
```

```
public class OrderService {

    public boolean process(Order order) {
        if (order.isValid() && save(order)) {
            sendEmail(order);
         }
    }

    private boolean save(Order order) {
        try {
            order.save();
            return true;
        } catch (SQLException e) {
            log.error("Error saving", e);
        }
        return false;
    }

    private void sendEmail(Order order) {
        String emailAdress = order.getEmailAdress();
        String firstname = order.getFirstname();
        String lastname = order.getLastname();

        Email mail = new Email(emailAdress, firstname, lastname);
        mail.send();
    }
}
```

Single Responsability Principle

```
public class OrderService {

    public boolean process(Order order) {

        if (order.isValid() && new OrderDAOImp().save(order)) {

            new MailSenderImpl().sendEmail(order);

         }

    }
}
```

Single Responsability Principle

« A class should have one reason to change. »

- Une classe ne doit avoir qu'un seul objectif fonctionnel (une seule responsabilité)
- Changements motivés par les mêmes raisons

- Sinon: si une classe a plusieurs responsabilités, elle seront liées...

O

Open-Closed Principle

Open-Closed Principle

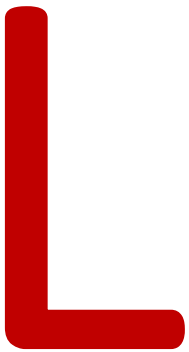
```
class PaymentService {
    int calculatePayment(int amount, Customer customer) {
        double discount = 0;

        switch (customer.type()) {
            case SILVER: discount = 0.1; break;
            case GOLD: discount = 0.2; break;
            ...
        }
        return amount - amount * discount;
    }
}
```

```
class Customer {
    double getDiscount() {
        return 0;
    }
}

class SilverCustomer extends Customer {
    double getDiscount() {
        return 0.1;
    }
}

class GoldCustomer extends Customer {
    double getDiscount() {
        return 0.2;
    }
}

class PaymentService {
    int calculatePayment(int amount, Customer customer) {
        double discount = customer.getDiscount();
        return amount - amount * discount;
    }
}
```

Open-Closed Principle

« Classes, methods should be open for extension, but closed for modifications. »

- On doit pouvoir étendre une classe sans changer son fonctionnement interne



Liskov’s Substitution Principle

Liskov's Substitution Principle

```
public class Rectangle {
    private int width;
    private int height;
    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public void setWidth(int w) { this.width = w; }
    public void setHeight(int h) { this.height = h; }
    public int area() {
        return width * height;
    }
}

public class Square extends Rectangle {
    public void setSide(int s) {
        this.setWidth(s);
        this.setHeight(s);
    }
    public void getSide() {return this.getWidth(); }
}
```

```
Square s = new Square();
s.setWidth(5);
s.setHeight(3);
System.out.println(s.area());
```

Liskov's Substitution Principle

« Subtypes must be substituable for their base types. »

- Lorsque du code référence une classe, il devrait pouvoir utiliser n'importe quelle instance de l'une de ses sous-classes

Interface Segregation Principle

Interface Segregation Principle

```
public interface Car {
    void startEngine();
    void accelerate();
}

public class Mustang implements Car {

    public void startEngine() {
        ...
    }

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

```
public interface Car {
    void startEngine();
    void accelerate();
    void backToThePast();
    void backToTheFuture();
}


public class DeloRean implements Car {

    public void startEngine() {
        ...
    }

    public void accelerate() {
        ...
    }

    public void backToThePast() {
        // back to the past...
    }

    public void backToTheFuture() {
        // back to the future...
    }
}
```

```
public class Mustang implements Car {

    public void startEngine() {
        ...
    }

    public void accelerate() {
        ...
    }

    public void backToThePast() {
        // because a Mustang
              can not back to the past!
        throw new
           UnsupportedOperationException();
    }

    public void backToTheFuture() {
        // because a Mustang
               can not back to the future!
        throw new
           UnsupportedOperationException();
    }
}
```

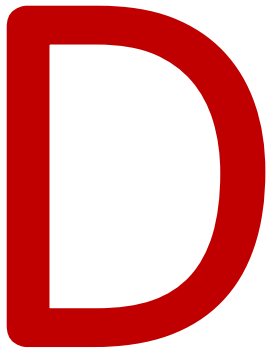
```
public interface Car {
    void startEngine();
    void accelerate();
}

public interface TimeMachine {
    void backToThePast();
    void backToTheFuture();
}

public class DeloRean implements Car,
                                  TimeMachine {

    public void startEngine() {
        ...
    }

    public void accelerate() {
        ...
    }

    public void backToThePast() {
        // back to the past...
    }

    public void backToTheFuture() {
        // back to the future...
    }
}
```

```
public class Mustang implements Car {

    public void startEngine() {
        ...
    }

    public void accelerate() {
        ...
    }

}
```

Interface Segregation Principle

« Clients should not be forced to depend on methods that they do not use. »

- Eviter les interfaces énormes
- Préférer plusieurs interfaces spécifiques



Dependency Inversion Principle

Dependency Inversion Principle

```
public class OrderService {

    public boolean process(Order order) {

        if (order.isValid() && new OrderDAOImp().save(order)) {

            new MailSenderImpl().sendEmail(order);

         }

    }
}
```

Dependency Inversion Principle

```
public class OrderService {

    private OrderDAO dao;
    private MessageSender sender;

    public OrderService(OrderDAO dao, MessageSender sender) {
        this.dao = dao;
        this.sender = sender;
    }

    public boolean process(Order order) {

        if (order.isValid() && dao.save(order)) {

            sender.sendEmail(order);

         }

    }
}
```

Dependency Inversion Principle

« High level modules should not depend on low level modules. Both should depend on abstractions. »

« Abstractions should not depend on details. Details should depend on abstractions. »

- Il faut dépendre des abstractions, pas des implémentations