# S.O.h.i.D

## A step to be a craftsman

# S.O.L.I.D

## Single Responsibility Principle

# WHAT DO YOU THINK ?

```java
public class UserService {

   private final String smtplogin;
   private Database _database;

   public UserService(String smtplogin) {
       this.smtplogin = smtplogin;
       _database = Database.getInstance();
   }

   public void register(String email, String password)
   {
       if (!email.contains("@"))
           throw new ValidationException("Email is not an email!");
       User user = new User(email, password);
       _database.save(user);
       SmtpClient _smtpClient = SmtpClient.connect(Smtp.getConf(), smtplogin);
       _smtpClient.send(new MailMessage("mysite@nowhere.com", email) {{
           subject = "HEllo fool!";
       }});
   }
}
```

bad-code-snippets/src/main/java/snippets/cleancode/p1_srp/UserService.java

https://www.lasurvie.fr/wp-content/uploads/2017/11/2-test-avis-comparatif-pinces-victorinox.png

# WHAT DO YOU THINK ?

```java
public String fizzbuzz(int number) {

   String res = "";

   if (number % 3 == 0) {
       res += "fizz";
   }

   if (number % 5 == 0) {
       res += "buzz";
   }

   if (res == "") {
       res = String.valueOf(number);
   }

   return res;
}
```

# SINGLE RESPONSIBILITY PRINCIPLE

*An artefact should have one and only one reason to change, meaning that an artefact should have only one job.*

## SMELLS

- *Large Class*
- *Long Method*
- *Lot of methods*
- *High Coupling/Low cohesion*
- *Helper class*
- *Multiple functional/technical concepts at the same*

# S.O.L.I.D

## Open/Close Principle

# WHAT DO YOU THINK ?

```java
public static class UserFilter implements Filter<User>{

   private final String firstname;

   public UserFilter(String firstname) {
       this.firstname = firstname;
   }

   @Override
   public boolean accepts(User user) {
       return firstname.equals(user.getFirstname());
   }
}
```

```java
public interface Filter<T> {

   boolean accepts(T value);

}
```

# WHAT DO YOU THINK ?

```java
public static class UserFilter implements Filter<User> {

    private final String firstname;
    private final String lastname;

    public UserFilter(String firstname, String lastname){
       this.firstname = firstname;
       this.lastname = lastname;
    }

@Override
public boolean accepts(User user) {
   return (firstname == null || firstname.equals(user.getFirstname()))
           && (lastname == null || lastname.equals(user.getLastname()));
}
```

```java
public interface Filter<T> {

   boolean accepts(T value);

}
```

# WHAT DO YOU THINK ?

```java
public static class FirstNameFilter implements Filter<User> {

    private final String firstname;

    public FirstNameFilter(String firstname) { this.firstname = firstname; }

    @Override
    public boolean accepts(User user) {
        return firstname.equals(user.getFirstname());
    }
}
```

```java
public static class LastNameFilter implements Filter<User> {

private final String lastname;

    public LastNameFilter(String lastname) { this.lastname = lastname; }

    @Override
    public boolean accepts(User user) {
        return lastname.equals(user.getLastname());
    }
}
```

```java
public interface Filter<T> {

   boolean accepts(T value);

}
```

```java
Filter<User> userFilter = Filters.and(new FirstNameFilter(firstname), new LastNameFilter(lastname));
```

# OPEN / CLOSE PRINCIPLE

*Objects or entities should be open for extension,*
*but closed for modification.*

## SMELLS

- *Complex switch/Lot of ifs*
- *High cyclomatic complexity*

# S.O.L.I.D

## Liskov Substitution Principle

# WHAT DO YOU THINK ?

```java
public class Rectangle {

    private int width, height;

    public void setDimension(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int area() {
        return width * height;
    }
}


public class Square extends Rectangle {

    @Override
    public void setDimension(int width, int height) {
        if (width != height)
            throw new IllegalArgumentException();
        super.setDimension(width, width);
    }
}
```

bad-code-snippets/src/main/java/snippets/solid/p2_lsp/Rectangle.java

# WHAT DO YOU THINK ?

```java
public class Rectangle {

    private int width, height;

    public void setDimension(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int area() {
        return width * height;
    }
}


public class Square extends Rectangle {

    @Override
    public void setDimension(int width, int height) {
        if (width != height)
            throw new IllegalArgumentException();
        super.setDimension(width, width);
    }
}
```

```java
public class Client {

    public int enlarge(Rectangle rectangle) {
       int height = rectangle.height;
       rectangle.setDimension(height * 3, height)
    }
}
```

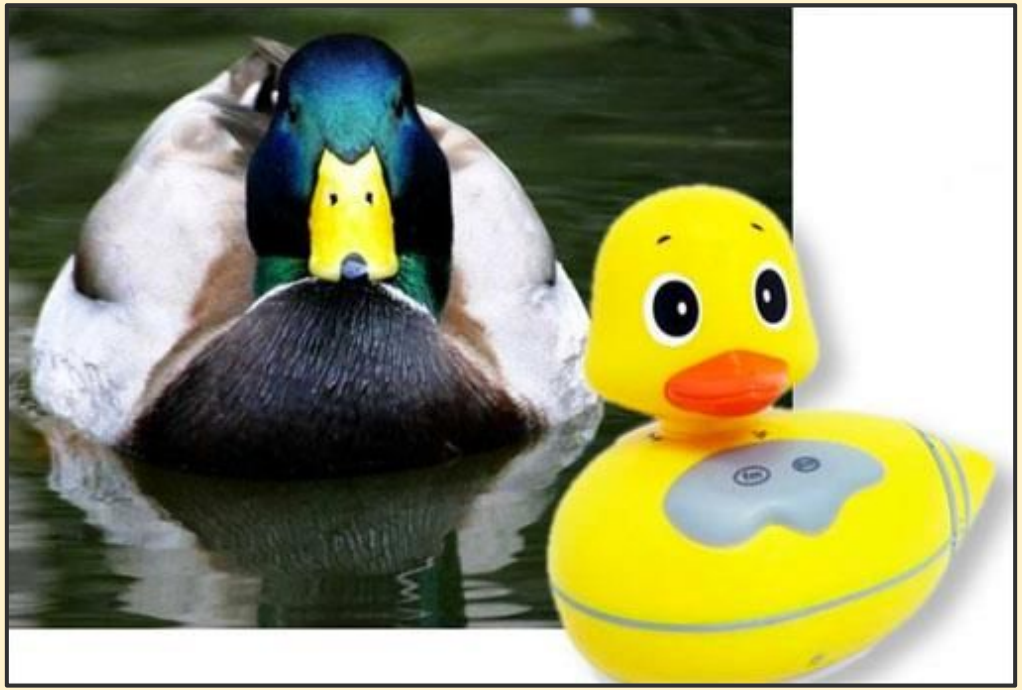
# LISKOV SUBSTITUTION PRINCIPLE

*Every subclass/derived class should be substitutable for their base/parent class*

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

## SMELLS

- *You have to check for the type provided (e.g. instanceof)*



# Liskov Substitution Principle

“ If it looks like a duck, quacks like a duck, but needs battery.
You probably have the wrong abstraction

# S.O.h.iD

## Interface Segregation Principle

SCANNER

COPIER


FAX

PHONE

PRINTER

# WHAT DO YOU THINK ?

```
public interface MultiFunctionPrinter {

   void fax(Page page, PhoneNumber number);

   void scan(Page page, EmailAddress address);

   void call(PhoneNumber number);

   void copy(Page page, int number);

   void print(Page page);

}
```

# WHAT DO YOU THINK ?

```
public interface MultiFunctionPrinter {

    public interface Fax {
        void fax(Page page, PhoneNumber number);
    }

    public interface Scanner {

        void scan(Page page, EmailAddress address);
    }

    public interface Phone {
        void call(PhoneNumber number);
    }

    public interface Copier {
        void copy(Page page, int number);
    }

    public interface Printer {
        void print(Page page);
    }
```

# INTERFACE SEGREGATION PRINCIPLE

*A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use*

## SMELLS

- *Fat interface/Class with lot of methods*
- *Interface has multiple responsibilities*
- *Difficulties to expose a subset of responsibilities*

# S.O.L.I.D

## Dependency Inversion Principle

# WHAT DO YOU THINK ?

```java
public class OrderProcessor {

   public double calculateTotal(Order order, Connection cnx) throws SQLException {
       double itemTotal = order.getItemTotal();
       double discountAmount = DiscountCalculator.calculateDiscount(order);
       double taxAmount = 0.0d;
       if (order.getCountry() == US)
           taxAmount = findTaxAmount(order, cnx);
       else if (order.getCountry() == UK)
           taxAmount = findVatAmount(order, cnx);
       double total = itemTotal - discountAmount + taxAmount;
       return total;
   }

   private double findVatAmount(Order order, Connection cnx) throws SQLException {
       Resources r = new Resources();
       try {
           PreparedStatement statement = r.push(cnx.prepareStatement( "select amount from vat where country=?" ));
           statement.setString(1, order.getCountry().name());
           ResultSet resultSet = r.push(statement.executeQuery());
           return resultSet.getDouble(1);
       }finally {
           r.dispose();
       }
   }

   private double findTaxAmount(Order order, Connection cnx) throws SQLException {
       ...
   }
}
```

# WHAT DO YOU THINK ?

```java
public interface DiscountCalculator {

   double calculateDiscount(Order order);

}

public interface Taxes {

   double findTaxAmount(Order order);

}

public class OrderProcessor {

   private final DiscountCalculator discountCalculator;
   private final Taxes taxes;

   public OrderProcessorRefactored(DiscountCalculator discountCalculator, Taxes taxes) {
       this.discountCalculator = discountCalculator;
       this.taxes = taxes;
   }

   public double calculateTotal(Order order, Connection cnx) throws SQLException {
       double itemTotal = order.getItemTotal();
       double discountAmount = discountCalculator.calculateDiscount(order);
       double taxAmount = taxes.findTaxAmount(order);
       double total = itemTotal - discountAmount + taxAmount;
       return total;
   }
}
```

bad-code-snippets/src/main/java/snippets/solid/p5_ocp/OrderProcessor.java

# DEPENDENCY INVERSION PRINCIPLE

*Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.*

## SMELLS

- *Dependencies between classes (vs interface)*
- *Monolithic architecture*
- *Abstraction depends on details/implementation*