SOLID PRINCIPLES

SOLID PRINCIPLES

- **S**ingle Responsibility
- **O**pen / Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

=> Prinzipien für wartbare, flexible und verständliche Software

SINGLE RESPONSIBILITY

Eine Klasse soll eine **einzige Verantwortlichkeit** haben.

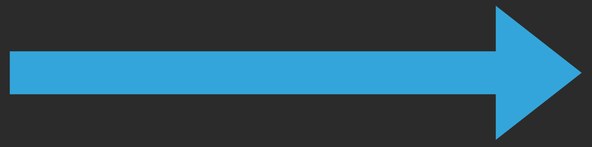
```
public class TextManipulator {

    public void replaceWord(String word, String replacement) {
    }

    public void appendText(String text) {
    }

    public void deleteWord(String word) {
    }

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



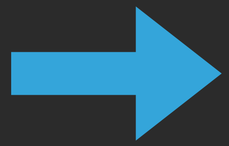
```
public class TextPrinter {

    public static void printText(String text) {
    }
}
```

Klassen sollen **offen für Erweiterungen** und **geschlossen gegenüber Modifikationen** sein.

```
public class TextPrinter {

    public void printText(String text) {
        System.out.println(text);
    }
}
```



```
public class TextPrinter {
    private static Logger log =
            LoggerFactory.getLogger(TextPrinter.class);
    private final boolean shouldLog;

    public TextPrinter() {
        this(false);
    }

    public TextPrinter(boolean shouldLog) {
        this.shouldLog = shouldLog;
    }

    public void printText(String text) {
        System.out.println(text);
        if (shouldLog) {
            log.info(text);
        }
    }
}
```

Neue Anforderung:
Der TextPrinter sollte optional zusätzlich den Text ins Log schreiben

Klassen sollen **offen für Erweiterungen** und
geschlossen gegenüber Modifikationen sein.

```
public class TextPrinter {

    public void printText(String text) {
        System.out.println(text);
    }
}
```

```
public class LoggingTextPrinter extends TextPrinter {
    private static final Logger log =
            LoggerFactory.getLogger(LoggingTextPrinter.class);

    @Override
    public void printText(String text) {
        super.printText(text);
        log.info(text);
    }
}
```

Neue Anforderung:
Der TextPrinter sollte optional zusätzlich
den Text ins Log schreiben

LISKOV SUBSTITUTION

Erbende Klassen sollen anstelle ihrer Elternklasse einsetzbar sein.

```
public class TextPrinter {

    public void printText(String text) {
        System.out.println(text);
    }
}
```

```
public class SummingPrinter extends TextPrinter {
    int sum;

    public SummingPrinter() {
        sum = 0;
    }

    @Override
    public void printText(String text) {
        super.printText(text);
        sum += Integer.parseInt(text);
    }
}
```

INTERFACE SEGREGATION

Ein Interface soll nur eng zusammengehörende Funktionen enthalten.

```
interface CoffeeMachine {
    void grindBeans();

    void brewCoffee();
}
```

```
class MyFancyCoffeeMachine implements CoffeeMachine {

    @Override
    public void grindBeans() {
        System.out.println("Grinding...");
    }

    @Override
    public void brewCoffee() {
        System.out.println("Brewing...");
    }
}
```

```
class FrenchPress implements CoffeeMachine {

    @Override
    public void grindBeans() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void brewCoffee() {
        System.out.println("Brewing...");
    }
}
```

INTERFACE SEGREGATION

Ein Interface soll nur eng zusammengehörende Funktionen enthalten.

```
interface CoffeeGrinder {
    void grindBeans();
}
```

```
interface CoffeeBrewer {
    void brewCoffee();
}
```

```
class MyFancyCoffeeMachine implements CoffeeGrinder,
        CoffeeBrewer {

    @Override
    public void grindBeans() {
        System.out.println("Grinding...");
    }

    @Override
    public void brewCoffee() {
        System.out.println("Brewing...");
    }
}
```

```
class FrenchPress implements CoffeeBrewer {

    @Override
    public void brewCoffee() {
        System.out.println("Brewing...");
    }
}
```

Ein Interface soll nur eng zusammengehörende Funktionen enthalten.

```
interface CoffeeGrinder {
    void grindBeans();
}
```

```
interface CoffeeBrewer {
    void brewCoffee();
}
```

Generics für Benutzung mehrerer Interfaces:

```
public <T extends CoffeeBrewer & CoffeeGrinder> void makeSomeCoffee(T machine) {
    machine.grindBeans();
    machine.brewCoffee();
    System.out.println("Enjoy!");
}
```

High-Level Klassen sollen nicht von Low-Level Klassen abhängen.
Beide sollen von Abstraktionen abhängen.

```
public class Project {
    private final BackEndDeveloper backEndDeveloper = new BackEndDeveloper();
    private final FrontEndDeveloper frontEndDeveloper = new FrontEndDeveloper();

    public void implement() {
        backEndDeveloper.writeJava();
        frontEndDeveloper.writeJavaScript();
    }
}
```

```
public class BackEndDeveloper {
    public void writeJava() {
    }
}
```

```
public class FrontEndDeveloper {
    public void writeJavaScript() {
    }
}
```

DEPENDENCY INVERSION

High-Level Klassen sollen nicht von Low-Level Klassen abhängen.
Beide sollen von Abstraktionen abhängen.

```
public class Project {
    private final List<Developer> developers;

    public Project(List<Developer> developers) {
        this.developers = developers;
    }

    public void implement() {
        developers.forEach(Developer::develop);
    }
}
```

```
public interface Developer {
    void develop();
}
```

```
public class BackEndDeveloper implements Developer {
    @Override
    public void develop() {
        writeJava();
    }
}
```

```
public class FrontEndDeveloper implements Developer {
    @Override
    public void develop() {
        writeJavaScript();
    }
}
```

BONUS - PACKAGES

- Packages helfen, zueinander zugehörige Klassen zu bündeln
- Vorteile:
  - Kleinere unabhängigere „Komponenten“
  - Erhöht Sinnhaftigkeit von Package-Private
- Namenskonvention: Eigentlich lower-Case, lowerCamelCase aber in Ordnung
- Je tiefer das Package, desto spezifischer, z.B. quinScape.azubis.calculator.userInputValidation

BONUS - BENENNUNG ALLGEMEIN

- Füllwörter vermeiden
  z.B. Utility, Value, Manager / Manage

- Abkürzungen vermeiden
  z.B. Util, Calc, Res

- Wenn exakte Benennung schwierig:
  => Ggf. macht Klasse / Methode zu viel

- „Geschützte“ Schlagwörter richtig verwenden
  z.B. get, set, Builder, Factory

WEITERFÜHRENDE LINKS

- https://www.baeldung.com/solid-principles
- https://www.microconsult.de/blog/2019/05/fl_solid-prinzipien/
- https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html
- https://intern.quinscape.de/confluence/display/CPLAC/Code+Style