ESDK best Practices Workshop

Cheat Sheet

abas ERP

Inhaltsverzeichnis

1 JAVA

1.1 KLASSE

Bauplan/Beschreibung eines Typs von Objekten. In einer Klasse wird beschrieben, welche Eigenschaften und Methoden ein Objekt dieser Klasse hat. Aus einer Klasse können Instanzen (die eigentlichen Objekte) erstellt werden.

```
public class Car {

    private String color;
    private int amountSeats;

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

```
Car car1 = new Car();
Car car2 = new Car();

car1.doSomething();
```

1.2 VERERBUNG

Klassen können Eigenschaften und Methoden von einer anderen Klasse erben. D. h. die Klasse besitzt selbst alle Eigenschaften und Methoden der sog. „Superklasse“ (Klasse, von der geerbt wird.)

```
public class Vehicle {

    public void drive(){
        System.out.println("driving");
    };
}
```

```
public class Car extends Vehicle{
```

```
Car car1 = new Car();
car1.drive()
```

1.3 INTERFACE

Ein Interface beschreibt Methoden, ohne diese zu Implementieren. Diese Interfaces können in Klassen implementiert werden. D. h. eine Klasse muss, wenn sie ein Interface implementiert, alle im Interface beschriebenen Methoden ausprogrammieren. Damit kann gewährleistet werden, dass eine Klasse X alle Methoden des Interfaces hat und somit überall verwendet werden kann, wo auch das Interface genutzt wird.

```
public interface Sellable {

    int getPrice();

}
```

```
public class Vehicle implements Sellable{

    @Override
    public int getPrice() {
        return 50;
    }
```

```
public class House implements Sellable {

    @Override
    public int getPrice() {
        return 100;
    }
}
```

```
Sellable sellable1 = new Vehicle();
Sellable sellable2 = new House();

sellable1.getPrice(); // 50
sellable2.getPrice(); // 100
```

1.4 ANNOTATION

Zusatzinformationen, die an Klassen, Felder, Methoden oder Parameter angehängt werden können. Diese können von anderen Code-Stellen und Frameworks genutzt werden, um bestimmte Funktionalitäten abzubilden.

```
@Override
public int getPrice() {
    return 100;
}
```

```
@ButtonEventHandler(field = "start", type = ButtonEventType.AFTER)
public void startAfter(ButtonEvent event, ScreenControl screenControl, DbContext ctx, QMDRUCK8D head)
```

1.5 SINGLE RESPONSIBLE PRINCIPLE

Beschreibt den Ansatz in Klassen & Methoden jeweils nur eine Aufgabe zu erfüllen. Anders ausgedrückt ist jede Klasse oder Methode ist nur für einen Aspekt verantwortlich.

Negativbeispiel: Hier wird ein Preis eines Autos ermittelt und eine Datei erstellt.

```
public void createCarAndWriteFile() throws IOException {
    Car car = new Car();

    BufferedWriter writer = new BufferedWriter(new FileWriter("test"));
    writer.write(car.getPrice());
    writer.close();
}
```

Positivbeispiel: Das Schreiben der Datei und das Ermitteln des Preises ist auf unterschiedliche Klassen aufgeteilt.

```
public class PriceExporter {

    public void exportCarPrice(Car car, PriceWriter priceWriter) throws IOException {
        int price = car.getPrice();
        priceWriter.writePriceToFile(price);
    }
}
```

```
public void writePriceToFile(int price) throws IOException {
    BufferedWriter writer = new BufferedWriter(new FileWriter("test"));
    writer.write(price);
    writer.close();
}
```

1.6 DRY – DON'T REPEAT YOURSELF

Beschreibt den Ansatz oder das Paradigma Code nicht zu duplizieren, sondern wiederzuverwenden.

Negativbeispiel:

```
int price = car.getPrice();
priceWriter.writePriceToFile(price);

Car car1 = new Car();
int factor1 = 5;
int price1 = factor1 * car1.getPrice();
priceWriter.writePriceToFile(price1);

Car car2 = new Car();
int factor2 = 5;
int price2 = factor1 * car2.getPrice();
priceWriter.writePriceToFile(price2);

Car car3 = new Car();
int factor3 = 5;
int price3 = factor1 * car3.getPrice();
priceWriter.writePriceToFile(price3);
```

Positivbeispiel:

```
public void export(PriceWriter priceWriter) throws IOException {
    exportPrice(new Car(), priceWriter);
    exportPrice(new Car(), priceWriter);
    exportPrice(new Car(), priceWriter);
}

private void exportPrice(Car car, PriceWriter priceWriter) throws IOException {
    int factor = 5;
    int price = factor * car.getPrice();
    priceWriter.writePriceToFile(price);
}
```

2 AUTOMATISCHE TESTS

2.1 UNIT TEST

Mit einem Unit Test wird eine einzelne Klasse oder Methode (Unit) getestet. Optimalerweise werden alle Abhängigkeiten der Unit im Test simuliert, um eventuelle Nebeneffekte auszuschließen.

2.2 INTEGRATION TEST

In einem Integration Test wird die Konfiguration und das Zusammenspiel mehrerer Komponenten gemeinsam getestet. Hier werden im Optimalfall keine Klassen simuliert, sondern die Anwendung in ihrer Konstellation getestet.

2.3 JUNIT

Java-Framework, womit Unit & Integrationtests geschrieben werden können.

2.4 MOCK

Als Mock bezeichnet man die Simulation einer Klasse, die beispielsweise als Abhängigkeit in einem Test genutzt wird. Die Mocks werden üblicherweise in den Tests selbst konfiguriert:

```
EDPSession edpSession = Mockito.mock(EDPSession.class);
Mockito.when(edpSession.createEditor()).thenReturn(mockEdpEditor);
```

2.5 ASSERTION

Zielannahme in einem Test, um zu überprüfen, ob ein bestimmtes Ergebnis erreicht wurde.

```
Assert.assertEquals(2, list.size());
```

3 AJO

3.1 EDITOROBJEKT

AJO-Objekt mit dem ein bestimmtes abas Datenbankobjekt manipuliert werden kann.

3.2 SELECTIONBUILDER

AJO-Objekt mit dem der Kopfteil eines bestimmten abas Objekts selektiert werden kann.

3.3 JFOP-SERVER

Java-Laufzeitumgebung, in der AJO-Programme ausgeführt werden.

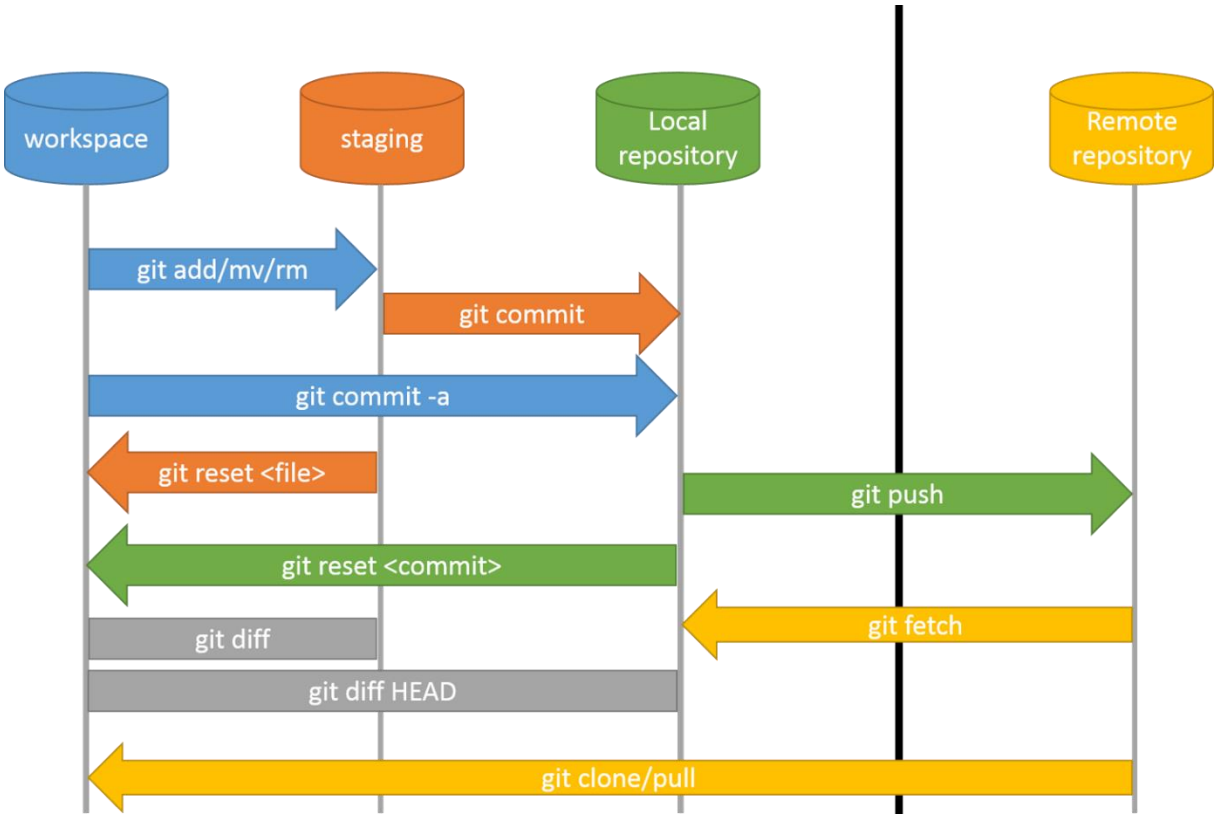
3.4 CLIENT-KONTEXT

Programm läuft auf dem lokalen Client und kommuniziert via EDP mit dem abas Server.

3.5 SERVER-KONTEXT

Programm läuft direkt auf dem abas Server bzw. ist in abas Events eingebunden.

4 GIT



Quelle: https://twiki.cern.ch/twiki/bin/view/BL4S/BL4SGitGuide

4.1 COMMIT

Erstellt einen neuen Commit (Neue Version/Entwicklungsstand) im Local Repository mit allen Änderungen, die aktuell in der Staging-Area liegen.

4.2 PUSH

Überträgt alle Änderungen vom Local Repository zum Remote Repository.

4.3 FETCH

Holt alle Änderungen aus dem Remote-Repository in das Local Repository. Der Workspace wird dabei aber nicht verändert.

4.4 MERGE

Fügt Änderungen zweier Branches zusammen. (Beispielsweise lokaler Master und Remote Master oder Branch 1 und Master).

4.5 PULL

Holt sich alle Änderungen aus dem Remote Repository und mergt diese in den aktuellen Branch.

4.6 CLONE

Kopiert ein Remote Git Repository und legt es lokal an.

4.7 BRANCH

Unterschiedliche Versionsstränge eines Projekts. Sinnvoll, um beispielsweise ein Feature isoliert zu entwickeln, ohne den Hauptstand des Projekts zu beeinflussen.

4.8 CHECKOUT

Wechsel von einem Branch zu einem anderen Branch oder Commit.

4.9 PULL REQUEST

Antragstellung oder Anfrage zum Mergen von Änderungen in einen Branch im Remote Repository.

Hat ein Entwickler eine Änderung gemacht und will diese in die Hauptversion (in der Regel Master) übernehmen, kann er online einen „Pull Request“ oder „Merge Request“ stellen. Dieser kann von anderen gereviewt werden. Erteilen die anderen Entwickler die Genehmigung, kann anschließend über einen Pull Request die Änderung in den Ziel-Branch übernommen werden.

4.10 REMOTE REPOSITORY

Zentrales Repository, mit dem alle Entwickler arbeiten. Dieses befindet sich in der Regel auf einem eigenen Server.

4.11 STAGING

Sammlung alle für einen Commit vorbereitete Änderungen. Dateien werden durch den „add“ Befehl in der Staging Area eingetragen.

4.12 MERGE KONFLIKT

Haben zwei Entwickler die gleiche Datei parallel bearbeitet und versuchen danach ihre Änderungen zu mergen, kommt es zu einem Merge Konflikt. Dieser muss händisch gelöst werden, in dem eine Person die Änderungen beider logisch zusammenführt. Hierfür wird in der Regel ein Merge Tool, wie beispielsweise Sourcetree verwendet.

Weitere Infos:

https://marklodato.github.io/visual-git-guide/index-de.html

https://rogerdudler.github.io/git-guide/

5 JIRA/SCRUM

5.1 ISSUE

Einzelne Aufgabe, die zu erledigen ist.

5.2 BACKLOG

Sammlung alle offenen Aufgaben, die aktuell in keinem Sprint bearbeitet werden.

5.3 KANBAN BOARD

Übersichtsboard für Aufgaben, mit mehreren Spalten, um den Status von Aufgaben, wie „ToDo“, „In Arbeit“, „Review“, „Erledigt“ zu visualisieren.

5.4 EPIC/MILESTONE

Zusammenfassung mehrerer zusammengehöriger Issues, mit dem größere Themenblöcke gruppiert werden.

5.5 SPRINT

Arbeitszeitraum mit fest definierten Aufgaben, die während der Laufzeit eines Sprints nicht geändert werden. Ein Sprint dauert üblicherweise 1 – 3 Wochen.

6 GRADLE

6.1 GRADLE

Build-Werkzeug um Softwareprojekte zu bauen und den „Build“ zu konfigurieren. Über Gradle werden u. A. die Abhängigkeiten, die Tests und die einzelnen Produktversionen organisiert.

6.2 BUILD.GRADLE

Zentrale Beschreibungsdatei eines Gradle-Projekts. Beinhaltet die Konfiguration eines Gradle Projekts.

7 DOCKER

7.1 DOCKER-COMPOSE

Konfigurationsdatei, um mehrere Docker-Container gemeinsam und in Kombination miteinander zu konfigurieren.

7.2 IMAGE

Vorlage für einen Docker-Container, auf Basis dessen dieser erstellt werden kann. Vergleichbar mit einem Betriebssystem Image.

7.3 CONTAINER

Laufzeitinstanz eines Docker-Images. Ein laufender Docker-Container kann mit einer virtuellen Maschine verglichen werden.

8 ESDK

8.1 NEXUS

Repository, welches kompilierte Software-Bibliotheken beinhaltet. Im ESDK-Umfeld werden die AJO-Bibliotheken im Nexus bereitgestellt.

9 CONTINIOUS INTEGRATION

9.1 JENKINS

Serveranwendung, die genutzt werden kann, um Softwareprojekte automatisiert zu bauen.