# Aufbau der Software:

Das Projekt wurde grundsätzlich nach der Vorgabe der Lektoren erstellt (MVVM Pattern). Es beinhaltet einen Data Access Layer (kurz DAL), welcher für das Ansprechen von Files und Datenbanken zuständig ist. Einen Business Layer, welcher als zentrale Schnittstelle zwischen den Controllern und dem DAL fungiert. Des Weiteren gibt es noch die bereits erwähnten Controller, welche für die Anzeige der Daten an der UI und für die Events in der UI zuständig sind. Des Weiteren gibt es noch einen globalen (Singleton) Konfigurationsmanager, um an den verschiedenen Stellen des Programms Daten aus einem Configfile zu laden und speichern.

Das Programm wurde in Java 12 geschrieben, mit der UI durch JavaFX 12. Erweitert wurde das Projekt durch das Maven-Framework. Dieses handelt unter anderem die externen Bibliotheken, welche benutzt werden.

Aufbau nach Angabe der LV:

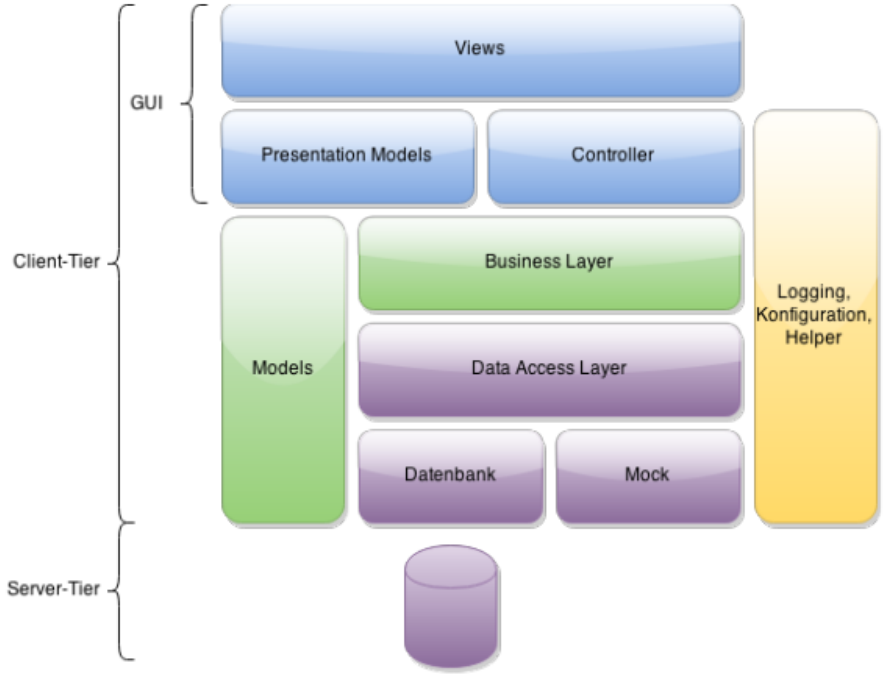


Figure 1: Architektur Java

*Abbildung 1Aufbau der Übungssoftware laut Lehrveranstaltung*

Hier sei zu erwähnen, dass von den unterstützenten Komponenten wie Logging oder Konfiguration (gelber Bereich) nur der Konfigurationsmanager implementiert wurde. Der Businesslayer (kurz BL) ist in meiner Lösung nicht nur für die Logik des Programmes zuständig, sondern auch für die Umwandlung der Grunddatenmodelle in Presentation Models (Diese erweitern die grundlegenden Daten Modelle um Methoden / Eigenschaften, welche die Controller zur Anzeige brauchen).

# Verwendete externe Bibliotheken:

Folgende externe Bibliotheken wurden benutzt:

```
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>12</version>
</dependency>
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>12</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1</version>
</dependency>
```

```
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.5</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>RELEASE</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-configuration2</artifactId>
    <version>2.0</version>
</dependency>
<dependency>
    <groupId>commons-beanutils</groupId>
    <artifactId>commons-beanutils</artifactId>
    <version>1.9.3</version>
</dependency>
```

Die javafx Bibliotheken „controls“ und „fxml“ liefern die Basis für die UI.

Von den Apache Commons wurde die Bibliothek DBCP2 geladen, welche es sehr einfach ermöglicht ein Connection-Pooling zur Datenbank zu implementieren, um hier zusätzliche Performance zu gewinnen.

Commons-IO wurde benutzt um zusätzliche Funktionen bezüglich des Filehandlings zu bekommen.

JUNIT ist das Testing-Framework welche benutzt werden sollte (Sollte, weil es sich zeitlich nicht mehr ausgegangen ist, hier Unittests zu schreiben)

Commons-Configuration2 ermöglicht es recht simpel einen vollständigen Configurationmanager zu implementieren, weil diese Bibliothek schon alle Funktion wie lesen und schreiben von z.B.: XML Files durch Angabe von Parameternamen unterstützt.

Commons-Beanutils wurde ursprünglich dazu verwendet um aus dem Projekt eine eigenständige JAR Datei zu generieren, leider konnte dies Aufgrund von Referenzproblemen mit der Derby-Bibliothek nicht erreicht werden.

Zum Schluss wurde noch Derby 10.15.1.3 benutzt, welche direkt als externe Bibliothek importiert wurde. Diese ist im Grunde die Treibersoftware zum ansprechen von Derby-Datenbanken. Es wurde zwar versucht auch hier die von Maven angebotenen Bibliotheken zu benutzen, dies brachte jedoch leider keinen Erfolg, weshalb hier auf die lokale Bibliothek zurückgegriffen werden musste.

# Implementation:

## Data Access Layer:

Der Data Access Layer (DAL) ist wie bereits erwähnt für alle File und Datenbankzugriffe zuständig.

```
long ret=0;
try {
    conn = DBCPDataSource.getConnection();
    prepStmt = conn.prepareStatement(PrepStatements.INSERT_PICTURE,Statement.RETURN_GENERATED_KEYS);
    prepStmt.setString( parameterIndex: 1, filePath);
    prepStmt.setString( parameterIndex: 2, x: "");  //keyword list
    prepStmt.setString( parameterIndex: 3, x: "");  //Description
    prepStmt.setTimestamp( parameterIndex: 4,convertToDatabaseColumn(LocalDateTime.now()));
    prepStmt.setString( parameterIndex: 5, x: "");   //Location
    prepStmt.setDouble( parameterIndex: 6,generateRandomDouble(0,30));
    prepStmt.setDouble( parameterIndex: 7,generateRandomDouble(0,30));
    prepStmt.setDouble( parameterIndex: 8,generateRandomDouble(0,30));
    prepStmt.setLong( parameterIndex: 9, x: new Random().nextInt( bound: 3)+1);

    long affectedRows = prepStmt.executeUpdate();
    if(affectedRows == 0){
        throw new SQLException("Creating Photographer failed, no rows affected");
    }
    ret = getGenKeys(prepStmt);
} catch (SQLException e) {
```

*Abbildung 2 Auszug aus dem DAL – Datenbankzugriff*

Der DAL benutzt für die Datenbankzugriffe Prepared Statements (Schon vorgefertigte static Strings) um hier die Sicherheit und Zugriffszeit zur Datenbank zu erhöhen. Damit sollten auch SQL-Injektions zum Teil verhindert werden.

Weiters werden Connections zur Datenbank über einen Connection-Pool gehandelt.

## Business Layer:

Der Business Layer (BL) validiert Daten, regt die Speicherung / Änderung / Laden von Daten durch den DAL an und wandelt die Basis Datenmodels in Presentationmodels um.

```
/**
 * Holen aller Bilder aus der Datenbank
 * @return (Type: List<P_Model_Picture>) Liste der geladenen Bilder
 */
public List<P_Model_Picture> getAllPictures() {
    List<P_Model_Picture> result = new ArrayList<>();
    List<PictureData> Pics = DAL.getAllPicturesParallel();
    for(PictureData p : Pics){
        result.add(new P_Model_Picture(p,getAuthorToPicture(p)));
    }
    return result;
}
```

*Abbildung 3 Auszug aus dem BL - Holen von Bildern aus der Datenbank (über DAL) und Umwandlung dieser*

```java
/**
 * Check ob die Daten eines Bildes valide sind (z.B.: Prüfen ob Stringlängen nicht zu groß für Tabelle)
 * @param pic (Type: P_Model_Picture) Das Bild das Überprüft wird
 * @return (Type: Boolean) True wenn Daten valide sind, sonst False
 */
public boolean isValidPicture(P_Model_Picture pic){
    AuthorData author = null;
    author = DAL.getAuthorByID(pic.getAuthor().getID());
    if(pic.getFilePath().length()<=255 && pic.getKeywords().length()<=300
            && pic.getDescription().length()<=300 && pic.getLocation().length()<=100 && author!=null)
        return true;
    return false;
}
```

*Abbildung 4 Validierung eines Bildes, welches gespeichert werden soll. (Angeregt vom zuständigen Controller)*

## Datenmodelle:

Die Basis Datenmodelle halten im Grunde nur die Daten, welche direkt in die Datenbank geschrieben werden. Um möglichst effizient mit der Datenbank sprechen zu können, haben diese nur sehr eingeschränkte Funktionalität. (Sie haben keine besonderen Funktionalitäten außer Getter und Setter Methoden.)

Die Presentationmodels im Gegenzug dazu wandeln diese Daten um, in eine Form, mit der die Controller möglichst direkt und einfach diese Daten anzeigen können.

```java
public class PictureData {
    private long _id;
    private String _filepath;
    private String _keywords;
    private String _desc;
    private LocalDateTime _date;
    private String _location;
    private double _focal;
    private double _exposure;
    private double _dazzle;
    private long _fk_author;
```

```java
public class P_Model_Picture {
    private String FilePath = null;
    private long ID = 0;
    private StringProperty FileName = new SimpleStringProperty();
    private StringProperty FileExt = new SimpleStringProperty();
    private StringProperty Keywords = new SimpleStringProperty();
    private StringProperty Description = new SimpleStringProperty();
    private StringProperty CreationTime = new SimpleStringProperty();
    private StringProperty Location = new SimpleStringProperty();
    private DoubleProperty FocalLength = new SimpleDoubleProperty();
    private DoubleProperty ExposureTime = new SimpleDoubleProperty();
    private DoubleProperty DazzleNumber = new SimpleDoubleProperty();
    private ObjectProperty<P_Model_Author> Author = new SimpleObjectProperty<>();
    private ImageView iv = null;
```

Wie man hier sieht, werden zum Beispiel Eigenschaften zur Verfügung gestellt wie, dass der Filepath aufgeteilt wurde und hier der Filename und die Fileendung herausgelöst wurde. Besonders auffällig hier ist, dass die Daten nicht mehr durch einfache Datentypen sondern durch Properties gehalten werden. Dies ermöglicht das sogenannte Databinding des Controllers von dem Datenmodel auf die UI.

## Controller:

Der Controller handelt das ganze Verhalten der UI und bringt die Daten zur Anzeige.

```
/**
 * Laden der Bildergallerie aus den verfügbaren Bildern. Jedes Bild bekommt ein setOnMouseClick
 * Eventhandler, welcher den Wechsel des Bildes veranlasst
 */
public void loadImagesGallery(){
    HBox box = new HBox();
    for(P_Model_Picture pic : Pictures){
        pic.getIv().fitHeightProperty().bind(galleryScrollPane.heightProperty());
        pic.getIv().setPreserveRatio(true);
        pic.getIv().setOnMouseClicked(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent mouseEvent) { changeMainPicture(pic); }
        });

        box.getChildren().add(pic.getIv());
    }
    galleryScrollPane.setVbarPolicy(ScrollPane.ScrollBarPolicy.NEVER);
    galleryScrollPane.setHbarPolicy(ScrollPane.ScrollBarPolicy.ALWAYS);
    galleryScrollPane.setContent(box);

    galleryScrollPane.setOnScroll(event -> {
        if(event.getDeltaX() == 0 && event.getDeltaY() != 0) {
            galleryScrollPane.setHvalue(galleryScrollPane.getHvalue() - event.getDeltaY() / box.getWidth()*4);
        }
    });

}
```

*Abbildung 5 Holen der Bilderdaten über den BL und anzeige in der Bildergallerie*

Wichtig ist, dass der Controller keine Datenlogik besitzt (Manipulieren von Daten…) sondern rein für die Anzeige zuständig ist. (ANMERKUNG: Wurde ehrlich gesagt nicht ganz sauber von mir implementiert)

## Configurationmanager:

Der Configurationmanager bietet die Möglichkeit Einstellungen der Applikation in einem Configfile zu speichern und diese Gegebenenfalls zu ändern. (z.B.: Datenbankpfad oder Anzeigegröße).

```
private double getPropertyDouble(String propertyName){
    try {
        return config.getDouble(propertyName);
    }
    catch(NoSuchElementException e){
        e.printStackTrace();
    }
    return -1;
}

private void setPropertyDouble(String PropertyName, double i){
    config.setProperty(PropertyName,i);
}
```

*Abbildung 6 Laden und speichern einer Konfiguration*

Dieser wurde als Singleton implementiert, um ihn von jeder Stelle im Programm aus ansprechen zu können.

## Abschließende Worte / Conclusio:

Der Einsatz des MVVM Patterns bringt auf jeden Fall viel Übersicht und vor allem gut getrennte Module in das Projekt. Es ist „leicht“ zu sagen, welche Funktionalität in welchen Programmteil steckt. Des Weiteren hilft es notfalls ganze Projekt-Teile, wenn nötig, auszutauschen.

Es war sehr herausfordernd das erste Mal mit einem UI-Framework wie JavaFX zu arbeiten. Ich habe hier besonders viel gelernt, dass man nicht immer alles selber schreiben muss, sondern, dass wenn man sich die Dokumentation etwas durchliest diese APIs schon von Haus aus sehr viel an Funktionalität mitbringen. Gerade der Einsatz von Propertybinding hat sehr viel Arbeit gespart.

Ich habe als erstes angefangen den DAL zu implementieren, weil ich mir dachte, wenn die Basis passt, kann man leicht darauf aufbauen. Jedoch bin ich im Laufe der Entwicklung draufgekommen, dass ich ganz andere, oder anders dargestellte Daten benötige. Heute würde ich den DAL nach Bedarf nachziehen, um mir hier Arbeit zu sparen. Ständig bereits überlegte und entwickelte Funktionen ändern zu müssen, weil es ja dann doch immer ganz anders kommt, als man es sich gedacht hat ist wirklich nicht optimal. Man hat dadurch ein Vielfaches an Arbeit.

Was ich durch ein Studienkollegen gelernt habe war, dass man Datenbankabfragen ganz einfach in Java parallelisieren kann. Dies hat mich sehr begeistert, denn 5000 Datenbankeinträge in 3ms zu laden ist schon sehr beeindruckend.

**Aber die wichtigste Erkenntnis:** 4 Tage vor Abgabedatum Anfangen ist einfach nur Blödheit! Da habe ich echt gelernt mich einfach doch das nächste Mal einfach ein paar Wochen früher hinsetzen und immer ein wenig zu tun, ist doch vermutlich wesentlich besser.