

Java Programm - Bücherverwaltung

Marco Bräuer

04.10.2021

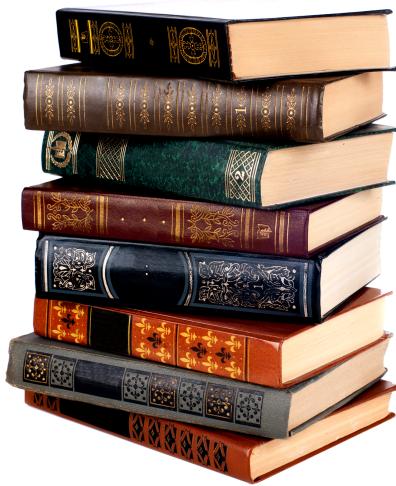


Abbildung 1: Logo Büchersammlung

Inhaltsverzeichnis

1 Analyse	4
1.1 Anforderung an das Programm	4
1.1.1 Eigenschaften eines Buches	4
1.2 Anwendungsfälle	4
1.3 Projektbegründung	5
1.4 Programmschnittstellen	5
1.5 Projektphasen	5
1.6 Entwicklungsprozess	5
2 Entwurf	6
2.1 Zielplattform	6
2.2 Architekturdesign	6
2.3 Entwurf der Benutzeroberflächen	6
2.4 Datenmodell	7
3 Implementierung	9
3.1 Konventionen bei der Programmierung	9
3.2 Einteilung des Projektes	9
3.3 Datenstruktur	10
3.3.1 Bibliotheken für Datenbank und JSON	10
3.3.2 Datenbank	11
3.3.3 Verbindung zur Datenbank	11
3.3.4 API	12
3.4 Logik	13
3.4.1 Konzept	13
3.4.2 Objektklassen	14
3.4.3 Listenklassen	15
3.5 Benutzeroberfläche	17
3.5.1 Design der Oberfläche	17
3.5.2 Listen erzeugen und füllen	19
3.5.3 Listen ausgeben	19
3.6 Test	21
4 Dokumentation	22
4.1 Benutzerdokumentation	22
5 Fazit	23
5.1 Soll/Ist-Vergleich	23
5.2 Lesson Learned	23

1 Analyse

1.1 Anforderung an das Programm

Es wird ein neues Programm von Beginn an entwickelt.

Das Programm soll eine Bücherliste tabellarisch darstellen. Jedes Buch soll mehrere Eigenschaften besitzen und diese sollen mit dargestellt werden.

Dem Programm soll eine Datenbank zugrunde liegen aus der zu Programmstart geladen wird.

Über eine Maske sollen neue Bücher hinzugefügt werden können und ausgewählte Bücher müssen gelöscht werden können. Die einzelnen Eigenschaften der Felder müssen geändert werden können.

Über ein Eingabefeld soll eine Suche nach der ISBN Nummer möglich sein.

Gefundene Ergebnisse sollen angezeigt werden und bei Bedarf der Bücherliste hinzugefügt werden können.

1.1.1 Eigenschaften eines Buches

- Nummer in der Liste
- Name des Buches
- Name des Autors
- Stichwort/ Genre
- Jahr der Erstauflage
- ISBN Nummer
- Ort/Platz im Regal

1.2 Anwendungsfälle

Es soll für einen Einzelplatz für einen Anwender mit einer privaten Büchersammlung entwickelt werden.

1.3 Projektbegründung

Bei einer Büchersammlung ist es Sinnvoll einen geordneten Überblick zu haben. Vor allem um Bücher schnell zu finden sowohl für Bücher im Bestand als auch für neue Bücher, die hinzugefügt werden sollen.

1.4 Programmschnittstellen

Die Datenbank als Speicher ist die erste Schnittstelle.

Eine geeignete online API zu einer Bücherdatenbank soll für die ISBN-Suche verwendet werden.

1.5 Projektphasen

Projektphase	geplante Zeit
Analyse	8h
Entwurf	8h
Implementierung	16h
Dokumentation	8h
Gesamt	40h

Tabelle 1: Projektphasen

1.6 Entwicklungsprozess

Bei der Bearbeitung des Projekts wird das Wasserfallmodell verfolgt. Vom Problem zum Programm in folgenden Schritten:

- Analyse
- Entwurf
- Implementierung
- Test

2 Entwurf

2.1 Zielplattform

Die Anwendung soll auf Windows PC lauffähig sein.

Windows 10 wird als Standard System vorausgesetzt.

Als Programmiersprache wird Java 8 verwendet und mit der IDE Eclipse wird der Quellcode geschrieben.

Als Datenbank wird eine lokale SQLite Datenbank eingesetzt.

2.2 Architekturdesign

Ich teile die Entwicklung des Programms in 3 Ebenen.

- Datenbank
- Logik
- Oberfläche

Die Einteilung dient der Übersicht in der Erstellung des Quellcodes und für eine bessere Dynamik.

So kann z.B. durch die Abgrenzung der Datenbankebene von der Logikebene schnell ein anderes Datenbanksystem als Grundlage gewechselt werden.

Zum Beispiel von **SQLite** zu **MySql Server**.

Die Trennung von Logik zu Oberfläche bietet den Vorteil, dass spätere Anpassungen in der Bedienung oder ein komplett neues Design zügig geändert werden kann ohne die Logik umschreiben zu müssen. Außerdem könnten die Ebenen auch separat in anderen Projekten wiederverwendet werden.

2.3 Entwurf der Benutzeroberflächen

Ein GUI ist geeignet, um schnell und bedienerfreundlich die Anforderungen an das Programm zu erfüllen.

Zum Erstellen der Benutzeroberfläche verwende ich den Window Builder, der als Addon in der Eclipse IDE kostenlos dazu geladen werden kann.

Die Informationen über die einzelnen Bücher werden, in einem Grid gelistet, dargestellt.

Bücherverwaltung 1.0							X
löschen bearbeiten hinzufügen	Liste durchsuchen ...						
	Nummer	Titel	Autor	Jahr	Genre	Verlag	ISBN
	1	Testtitel1	AutorXY	1988	Horror	V1	564684848
	2	Testtitel2	AutorKP	1972	Roman	V2	669684968
	3	Testtitel3	AutoPZ	2005	Biografie	V3	987968749
	...						
	99	Testtitel98	AutorKP	2011	Novelle	V5	846813584
	100	Testtitel100	AutoPZ	2019	Biografie	V1	541554856
	Im Internet suchen ...						
	Nummer	Titel	Autor	Jahr	Genre	Verlag	ISBN
	1	Gefunden 1	AutorXY	2000	Schulbuch	V1	789748777
	Neues Buch erstellen						
	Nummer	Titel	Autor	Jahr	Genre	Verlag	ISBN
	1	NEU	NEUNEU	2000	Krimi	V9	188614684

Abbildung 2: Entwurf der Benutzeroberfläche

2.4 Datenmodell

Zu Beginn des Programms wird eine Verbindung zur Datenbank hergestellt. Danach werden Listen erzeugt und diese mit den Daten aus der Datenbank gefüllt. Während der Laufzeit werden alle Darstellungen und interne Suchen über die Listen vollzogen. Änderungen, wie das bearbeiten einzelner Felder, das Löschen oder das Hinzufügen werden nach aktualisieren der Listen in die Datenbank geschrieben. Zum Programm Ende wird die Verbindung zur Datenbank getrennt. In der folgenden Abbildung wird ein Modell des Programms dargestellt.

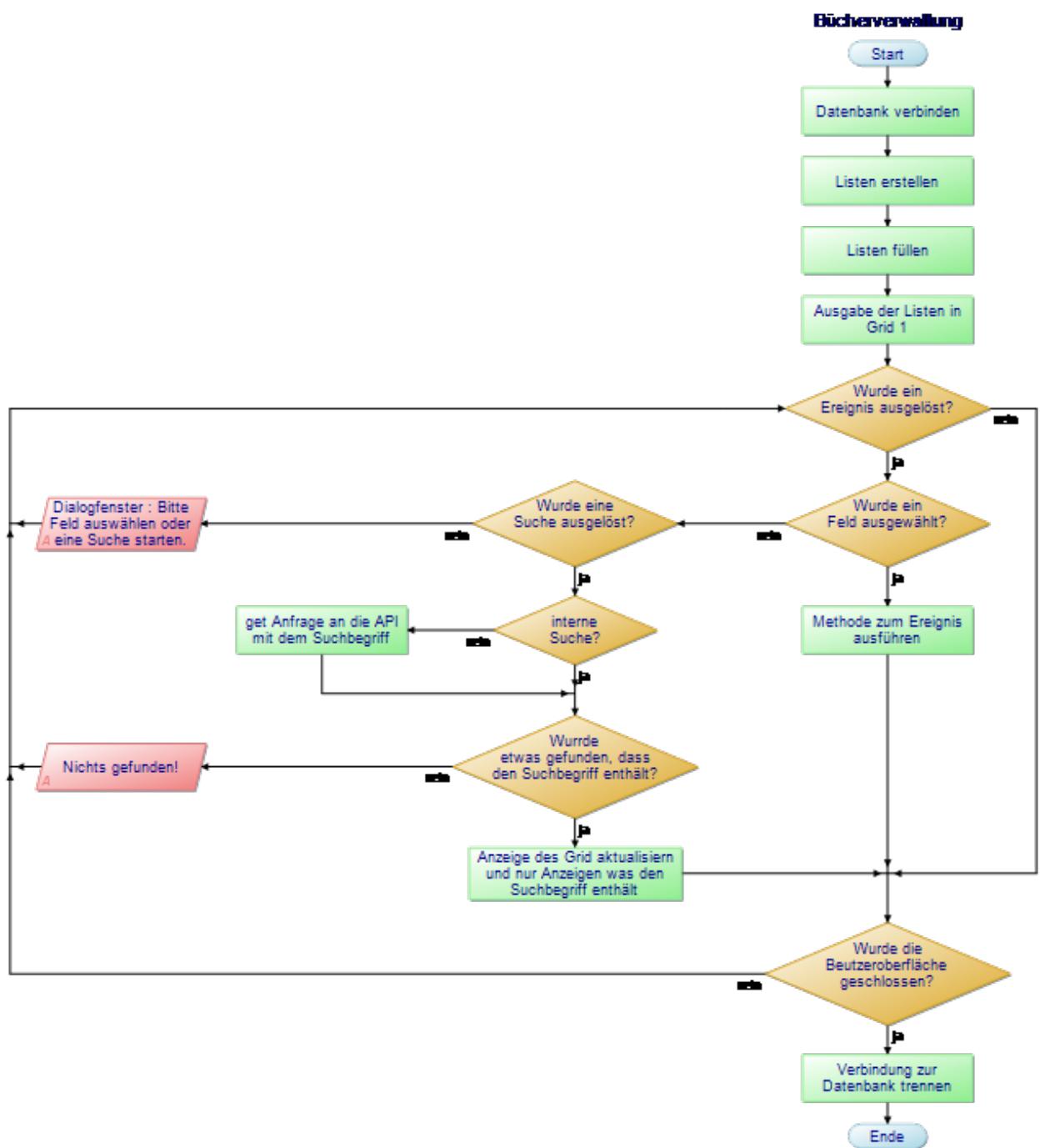


Abbildung 3: Programmablaufplan

3 Implementierung

3.1 Konventionen bei der Programmierung

Klassennamen bekommen ein führendes **T**, Eigenschaften ein führendes **F** und Parameter ein führendes **A**.

Die Klasse mit dem JFrame für die Benutzeroberfläche bekommt ein führendes **U**.

Die verwendete Sprache beim Benennen der erzeugten Wörter im Code ist Englisch.
Namen von Packages werden klein geschrieben.

3.2 Einteilung des Projektes

Zuerst erstelle ich ein neues Java Projekt mit dem Namen **bookDB**.

In diesem Projekt lege ich dann 3 Packages an:

- database
- logic
- userinterface

Im Package database erzeuge ich eine Klasse **TDatabase**, die für die Verbindung zur Datenbank und zur API zuständige Methoden enthalten soll.

Im Package logic erstelle ich für jede Eigenschaft des Buches und für das Buch selbst eine Klasse und eine Klasse, die eine Array Liste dazu enthält.

Für die Verarbeitung der Json -Werte wird eine Klasse und eine Klasse mit zugehöriger Liste erstellt.

Klassenname	Name der Liste
TAuthor	TAuthorList
TGenre	TGenreList
TLocation	TLocationList
TBook	TBookList
TJson	TJsonList
TConstants	-

Tabelle 2: Klassennamen vom Package logic

Außerdem wird eine Klasse **TConstants** erzeugt, die alle Konstanten für das Programm enthält. Als Programmkonstanten definiere ich:

- Name der Datenbank
- Pfad zur Datenbank
- Name der Datei mit dem API -Key
- Pfad zur API -Key Datei
- API -Url

Über den Marketplace der IDE installiere ich den Windowbuilder.

Im package userinterface erstelle ich eine Klasse **UMain**, der mit dem Windowbuilder ein JFrame erzeugt. Hier kann mithilfe des Designers eine Benutzeroberfläche erstellt werden.

Zum testen und erstellen von schwierigem Code erstelle ich eine Klasse **TListTester** als ausführbares Java Programm.

3.3 Datenstruktur

3.3.1 Bibliotheken für Datenbank und JSON

Damit ich eine Verbindung zu meiner erstellten Datenbank herstellen und SQL Befehle im Code implementieren kann, lade ich Java SQL Treiber aus dem Internet für das Verwenden der JDBC Klassen.

Quelle: <http://www.java2s.com/Code/Jar/s/Downloadssqlitejdbc372jar.htm>

Für das parsen von JSON lade ich mir ebenfalls eine ausführbare *.jar Datei aus dem Internet.

Quelle: <https://jar-download.com/artifacts/org.json>

Für den Zugriff auf diese Bibliotheken kopiere ich die 2 *.jar Dateien in meinen Projektordner und in der IDE füge ich diese durch Rechtsklick auf die Dateien zum *Buildpath* des Projektes hinzu.

Jetzt müssen noch 2 Einträge in der *module-info* Datei erfolgen:

- requires java.sql
- requires java.json

3.3.2 Datenbank

Das Programm benötigt zum speichern und wiederherstellen der Benutzerdaten eine Datenbank. Dafür verwende ich eine lokale **SQLite** Datenbank.

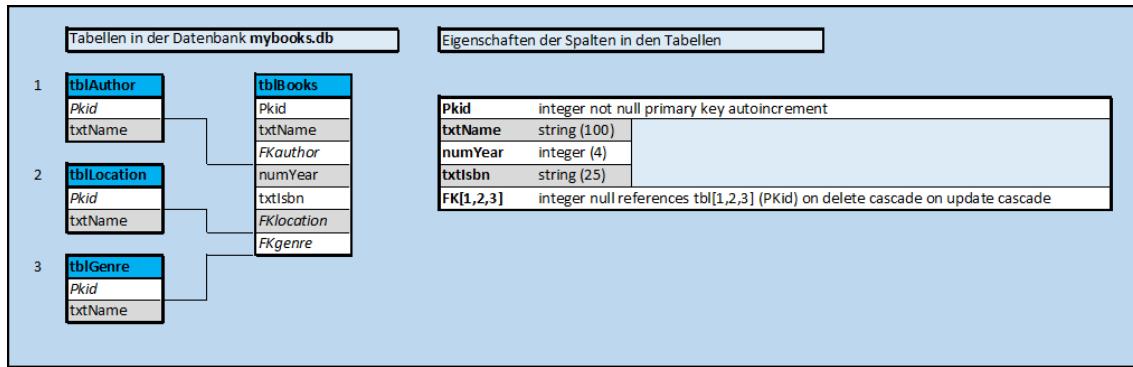


Abbildung 4: Entwurf der Datenbank

Für das Erstellen der Datenbank **mybooks.db**, benutze ich das Tool *Database.net*, welches mir im Unterricht bei der RTG bereitgestellt wurde.

Über die Eingabemaske erzeuge ich die Tabellen mit einfachen SQLite befehlen. Die Tabellen *tblAuthor*, *tblLocation* und *tblGenre* sind einfache *1:n* Beziehungen zur Tabelle *tblBooks*.

Beim erzeugen der Fremdschlüssel verwende ich *cascade*, damit beim löschen später die verknüpften Fremdschlüsselbeziehungen mit behandelt werden und dazu nötige SQL Befehle entfallen.

3.3.3 Verbindung zur Datenbank

Zum Start des Programms soll eine Verbindung zur Datenbank erfolgen und nach Beenden des Programms soll diese getrennt werden.

Hierfür erzeuge ich nur eine Instanz der Klasse **TDatabase** während der Laufzeit des Programms. Mit *public static final* vor dem Klassennamen wird das in JAVA sichergestellt. Zum Verbinden und trennen der Datenbank verwende ich 2 Methoden:

- `public void connect()`
- `public void disconnect()`

Die Syntax habe ich im Netz auf der Seite Stackoverflow im Forum recherchiert.

```
public void connect() {  
    try {  
        if (connection != null)  
            return;  
        System.out.println("Creating Connection to Database...");  
        connection = DriverManager.getConnection("jdbc:sqlite:" + TConstants.CDatabasePath);  
        if (!connection.isClosed())  
            System.out.println("...Connection established");  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
  
    Runtime.getRuntime().addShutdownHook(new Thread() {  
        public void run() {  
            try {  
                if (!connection.isClosed() && connection != null)  
                    connection.close();  
                if (connection.isClosed())  
                    System.out.println("Connection to Database closed");  
            } catch (SQLException e) {  
                e.printStackTrace();  
            }  
        }  
    });  
}
```

Abbildung 5: Methode connect

3.3.4 API

Für die Suche nach Büchern über das Internet benutze ich die **GoogleBook API**.

Auf der Google API Seite richte ich mir unter meinem Benutzerkonto eine neue Applikation ein und lasse mir einen Schlüssel (key) dafür erzeugen. Außerdem beschränke ich diesen *key* auf die GoogleBook API.

Diesen Key speichere ich mir lokal in einer Textdatei und lese diesen zu Beginn des Programms mit der Methode *getApiKey* aus.

Jetzt können Anfragen über die API-URL (die ich in der Klasse TConstants festlege), gestellt werden. In diesem Fall verwende ich die URL:

<https://www.googleapis.com/books/v1/volumes>

Darüber werden dann die Suchanfragen gestellt und eine Json wird bei erfolgreicher Verbindung zurückgegeben.

Für das holen der JSON verwende ich die Methode

public void getJson()

Die Syntax habe ich im Netz auf der Seite Stackoverflow im Forum recherchiert. Alle benötigten Klassen und zugehörige Methoden, wie z.B. die HTTP Verbindung mit SSL Verschlüsselung und der Methode **GET** aus der Klasse **java.net.url**, können über Mausklick importiert werden.

```

public String getJson(String searchquery) {
    String url = TConstants.CApiUrl + "?q=" + searchquery + "&maxResults=2&key=" + getApiKey();
    String JsonString = "";
    try {
        URL obj = new URL(url);
        HttpsURLConnection con = (HttpsURLConnection) obj.openConnection();
        con.setRequestMethod("GET");
        // add request header
        con.setRequestProperty("User-Agent", "Mozilla/5.0");
        int responseCode = con.getResponseCode();

        System.out.println("\nSending 'GET' request to URL : " + url);
        System.out.println("Response Code : " + responseCode);

        BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
        String inputLine;
        StringBuffer response = new StringBuffer();
        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        in.close();
        JsonString = response.toString();
        return JsonString;
    } catch (MalformedURLException e) {
        JOptionPane.showMessageDialog(null, "Fehler bei der angegebenen Url, bitte auth key prüfen.");
        e.printStackTrace();
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null, "Fehler bei der Internetverbindung.");
        e.printStackTrace();
    }
    return JsonString;
}

```

Abbildung 6: Methode getJson

3.4 Logik

3.4.1 Konzept

Zur Umsetzung der Aufgabenstellung im Quellcode benutze ich Objektlisten von Objekten. Diese Listen werden zu Programmstart in der **UMain** erstellt und mit den Methoden aus der Logik, die auf die Datenbank zugreifen, gefüllt.

Damit setze ich die Darstellung und Bearbeitung der Daten um.

3.4.2 Objektklassen

Jede Klasse eines Objekts besteht aus Eigenschaften, die Privat deklariert werden. Diese Klassen verfügen über einen Konstruktor in dem die Parameter in dessen Eigenschaften übergeben werden. Mit öffentlichen *getter* und *setter* Methoden kann auf die Eigenschaften zugegriffen werden.

Die Objektklassen bekommen jeweils eine *save* und eine *delete* Methode, zum speichern, bzw. löschen der Objekte in der Datenbank.

```
public class TAuthor {  
    // interface  
    // PROPERTIES  
    private int FID;  
    private String FName;  
  
    // implement  
    // CONSTRUCTOR  
    // @param to @property --> compile+  
    public TAuthor(int AID, String AName) {  
        this.FID = AID;  
        this.FName = AName;  
    }  
  
    // *****  
    // PROPERTY READ id WRITE id  
    public int getID() {  
        return FID;  
    }  
  
    public void setID(int ID) {  
        this.FID = ID;  
    }  
  
    // PROPERTIE read Name write Name  
    public String getName() {  
        return FName;  
    }  
  
    public void setName(String Name) {  
        this.FName = Name;  
    }  
}
```

Abbildung 7: Aufbau einer Objektklasse

3.4.3 Listenklassen

Die Listen sind Objektlisten, die mit den Daten der Datenbank nach Vorlage des zugehörigen Objektes gefüllt werden.

Jede Listenklasse besitzt eine Methode `public void setContent()` die ein select * state-

```
public class TAuthorList extends ArrayList<TAuthor> {
    private static final long serialVersionUID = -123456789012345678L;
    List<TAuthor> FArtistList;

    public TAuthorList(List<TAuthor> AArtistList) {
        this.FArtistList = AArtistList;
    }
}
```

Abbildung 8: Erzeugen einer Objektliste in Java

ment ausführt und alle Einträge aus der Datenbank ausliest. Danach werden Objekte erzeugt und mit den Werten aus der Datenbank belegt und in die Liste eingetragen.

```
public void setContent() {
    int tempId;
    String tempName;
    TAuthor tempArtist;

    try {
        Statement stmt = TDatabase.connection.createStatement();

        ResultSet rs = stmt.executeQuery("SELECT * FROM [tblAuthor];");
        while (rs.next()) {
            tempName = rs.getString("txtName"); // Werte holen
            tempId = rs.getInt("PKid");

            tempArtist = new TAuthor(tempId, tempName); // Objekt erstellen
            this.add(tempArtist); // Objekt der Liste hinzufügen
        }
        rs.close();
        stmt.close();
    } catch (SQLException e) {
        JOptionPane.showMessageDialog(null, "Fehler beim Laden der Daten in die Autor Liste");
    }
}
```

Abbildung 9: Füllen der Objektliste

Ein JSON Objekt baue ich wie alle anderen Objekte auf, aber die *delete* und *save* Methoden benötige ich nicht.

Bei der JSON Objektliste verwende ich eine Methode *parseJson* um die erhaltene Json nach Programm relevanten Daten zu untersuchen und Objekte mit diesen Werten zu erstellen, die dann in die Liste eingetragen werden.

```
// METHODEN
public void parseJson(String searchquery) throws JSONException {
    TDatabase database1 = TDatabase.getInstance();
    String data = database1.getJson(searchquery);
    System.out.println(data);
    JSONObject jObjectMain = new JSONObject(data);
    // JSONArray jsonArray = (JSONArray) jObject.get("items");
    JSONArray jsonArrayItems = jObjectMain.getJSONArray("items");
    for (int i = 0; i < jsonArrayItems.length(); i++) {
        // String tempId = jsonArrayItems.getJSONObject(i).getString("id");

        String tempName = jsonArrayItems.getJSONObject(i).getJSONObject("volumeInfo").getString("title");
        //String tempId = jsonArrayItems.getJSONObject(i).getString("id");
        String tempYear = jsonArrayItems.getJSONObject(i).getJSONObject("volumeInfo").getString("publishedDate");
        String tempGenre = jsonArrayItems.getJSONObject(i).getJSONObject("volumeInfo").getString("subtitle");
        String tempAuthor = jsonArrayItems.getJSONObject(i).getJSONObject("volumeInfo").getJSONArray("authors")
            .getString(0);
        String tempIdentifier = jsonArrayItems.getJSONObject(i).getJSONObject("volumeInfo")
            .getJSONArray("industryIdentifiers").get(0).toString();
        String tempIsbn = tempIdentifier.split("\"")[3];

        TJson tempJson = new TJson(-1, tempName, tempAuthor, tempYear, tempIsbn, tempGenre); // Objekt
        // erstellen
        this.add(tempJson);
    }
}
```

Abbildung 10: Füllen der Json Objektliste

3.5 Benutzeroberfläche

3.5.1 Design der Oberfläche

Da ich nur eine Woche Zeit hatte um das Projekt zu realisieren habe ich mich für ein sehr einfaches Design entschieden.

Eine Form auf der mehrere Toolbars liegen mit 2 mal JTable reichen zum erfüllen der Aufgabe. In die Toolbars habe ich jeweils JTextfields, JLabels und JButtons gelegt. Die JTable habe ich über rechtsklick auf den aufgespannten JTable mit einem Scrollpane umgeben, damit alle Daten erreicht werden können.

In einer späteren phase des Designs habe ich den beiden Suchfeldern noch einzelne Buttons hinzugefügt, die notwendig sind um alle Aufgaben des Programms zu realisieren. Das war zu Beginn der Programmierung noch nicht abzusehen.

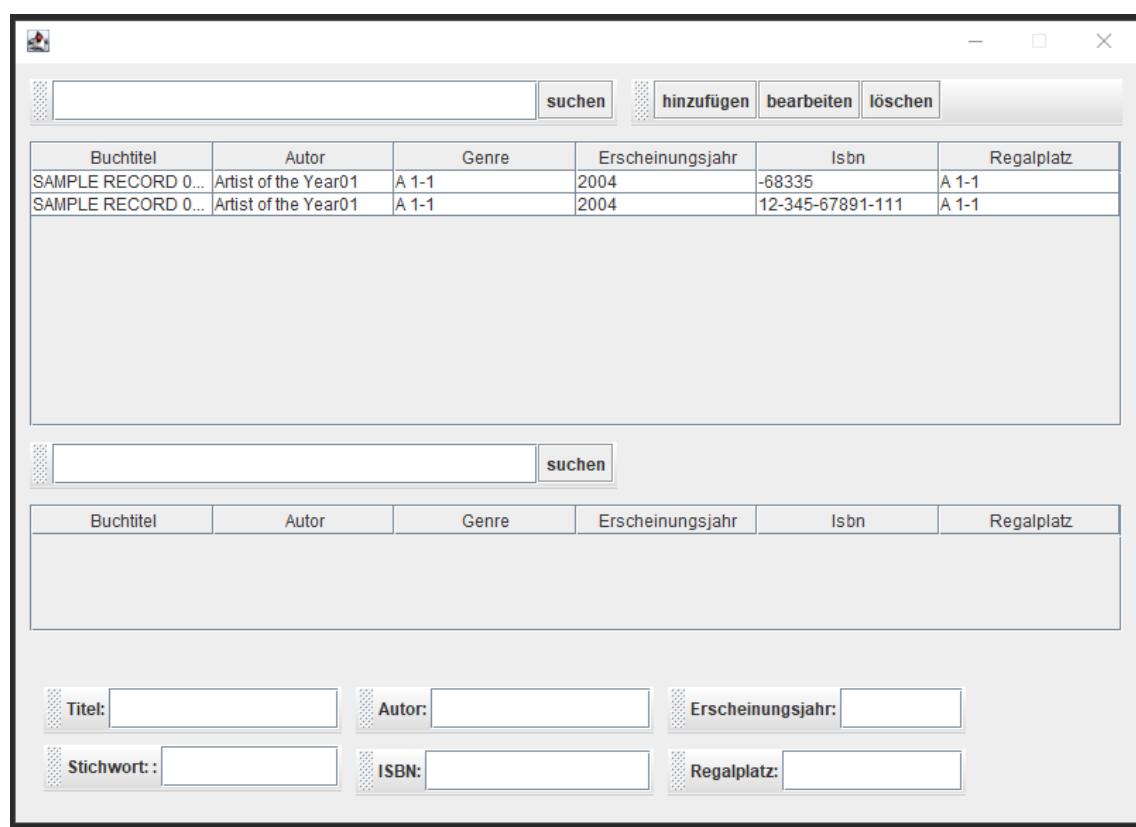


Abbildung 11: Erstes Ausführbares Design

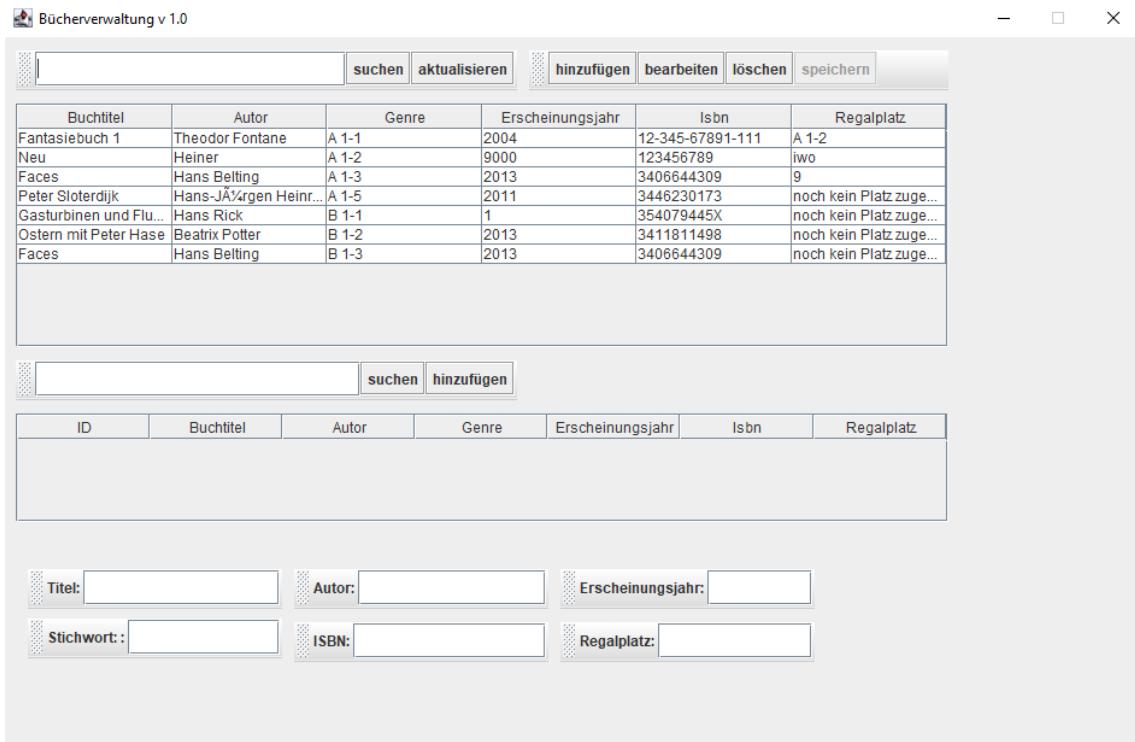


Abbildung 12: Zweites Ausführbares Design

Den oberen JTable *grdMain* benutze ich zur Ausgabe der Tabelle *tblBooks* aus der Datenbank. Den unteren JTable *grdWeb* benutze ich zur Ausgabe der aus dem Netz über die API geholten Werte aus der JSON Liste.

Beide Tabellen erhalten feste Spalten mit den Eigenschaften eines Buches, wobei ich den zugehörigen Primärschlüssel eines Buches aus der Datenbank in der Spalte *ID* unsichtbar anzeigen lasse, indem ich die Spaltenbreite auf 0 setze.

Zum füllen eines JTable benötigt man ein *model*, dass mit Objekten gefüllt wird. Indem ich dem Tabel das model mit dem Objekt mit den Spalten übergebe, erhalte ich die Header.

```
Object[] columns = { "ID", "Buchtitel", "Autor", "Genre", "Erscheinungsjahr", "Isbn", "Regalplatz" };
DefaultTableModel modelList = new DefaultTableModel();
DefaultTableModel modelWeb = new DefaultTableModel();

Object[] rowList = new Object[7];
Object[] rowWeb = new Object[7];
```

Abbildung 13: Objekt mit den Header Spalten und dem model

```

void setGrdMainHeader() {

    modelList.setColumnIdentifiers(columns);
    grdMain.setModel(modelList);

}

```

Abbildung 14: Setzen der Header

3.5.2 Listen erzeugen und füllen

Die Listen werden als Objektlisten in der **UMain** instanziiert und gefüllt. Beim hinzufügen, bearbeiten und löschen werden in den Ereignissen der Buttons Methoden aufgerufen, die über die Eigenschaften der Listen die Änderungen vornehmen und mit den Methoden der Objekte die Datenbank aktualisieren.

```

void createLists() {
    Authorlist1 = new TAuthorList(new ArrayList<TAuthor>()); // init
    Locationlist1 = new TLocationList(new ArrayList<TLocation>()); // init
    Genrelist1 = new TGenreList(new ArrayList<TGenre>()); // init
    Bookslist1 = new TBooksList(new ArrayList<TBook>()); // init
    JsonList1 = new TJsonList(new ArrayList<TJson>()); // init

}

```

Abbildung 15: Erzeugen der Listen

```

void setListContent() {
    Authorlist1.setContent(); // fill list
    Locationlist1.setContent(); // fill list
    Genrelist1.setContent(); // fill list
    Bookslist1.setContent(Authorlist1, Locationlist1, Genrelist1); // fill list
}

```

Abbildung 16: Füllen der Listen

3.5.3 Listen ausgeben

Bei der Ausgabe der Listen werden die vorher instanzierten models der JTable mit allen Werten der Listen gefüllt und angezeigt. Beim Auswählen eines Objektes aus der Anzeige wird über ein **Maus Klick Event** und einer Methode die aktuelle Reihe als Index

zwischen gespeichert. Darüber kann das jeweilige Objekt bearbeitet werden.
Bei der Suchfunktion im oberen Teil wird nur nach einer vorhandenen Isbn in der Liste gesucht.
Bei der Suche im Web über die **GoogleBook API** wird nach allen Eigenschaften eines Buch Objektes gesucht.

```
void setGridContent() {
    modelList.setRowCount(0);
    for (int i = 0; i < Bookslist1.size(); i++) {

        rowList[0] = Bookslist1.get(i).getID();
        rowList[1] = Bookslist1.get(i).getName();
        rowList[2] = Bookslist1.get(i).getAuthor().getName();
        rowList[3] = Bookslist1.get(i).getGenre().getName();
        rowList[4] = Bookslist1.get(i).getYear();
        rowList[5] = Bookslist1.get(i).getIsbn();
        rowList[6] = Bookslist1.get(i).getLocation().getName();

        modelList.addRow(rowList);
    }
}
```

Abbildung 17: Ausgabe im JTable *grdMain*

```

private void getWebResult() throws JSONException, UnsupportedEncodingException {
    JsonList1.removeAll(JsonList1);
    if (txtWebSearch.getText().equals("")) {
        JOptionPane.showMessageDialog(null, "Bitte etwas in das Suchfeld eingeben.");
        return;
    }

    JsonList1.parseJson(txtWebSearch.getText());
    modelWeb.setRowCount(0);
    for (int i = 0; i < JsonList1.size(); i++) {

        rowWeb[0] = JsonList1.get(i).getID();
        rowWeb[1] = JsonList1.get(i).getName();
        rowWeb[2] = JsonList1.get(i).getAuthor();
        rowWeb[3] = JsonList1.get(i).getGenre();
        rowWeb[4] = JsonList1.get(i).getYear();
        rowWeb[5] = JsonList1.get(i).getIsbn();
        rowWeb[6] = "noch kein Platz zugeordnet";

        modelWeb.addRow(rowWeb);
    }
}

```

Abbildung 18: Ausgabe im JTable *grdWeb*

3.6 Test

Beim testen der Suchfunktionen traten Fehler bei der verwendung mehrer Wörter auf, da noch keine UTF-Net encoding implementiert ist und die einzelnen leerzeichen zwischen den Wörtern noch nicht richtig ersetzt werden. Außerdem kam es bei manchen Suchbegriffen zu einem Fehler bei der Zuordnung des Knotenpunktes *subtitle* für die Eigenschaft *Genre*.

Außerdem kann es noch beim manuellen Erstellen von neuen Büchern zu doppelten Einträgen in der Datenbank kommen.

The screenshot shows a Java Swing application window. At the top, there is a search bar with the text "Peter" and two buttons: "suchen" (search) and "hinzufügen" (add). Below the search bar is a JTable with the following data:

ID	Buchtitel	Autor	Genre	Erscheinungsjahr	Isbn	Regalplatz
-1	Peter Sloterdijk	Hans-Jürgen H...	die Kunst des Phi...	2011	3446230173	noch kein Platz zu...
-1	Ostern mit Peter ...	Beatrix Potter	ein Pop-up-Bilder...	2013	3411811498	noch kein Platz zu...

Abbildung 19: Testen einer Internetsuche

4 Dokumentation

4.1 Benutzerdokumentation

Eine Benutzerdokumentation ist aus Zeitgründen entfallen.

5 Fazit

5.1 Soll/Ist-Vergleich

Beim Soll/Ist-Vergleich ist vor allem aufgefallen, dass die Implementierung und Dokumentation deutlich länger gedauert hat als angenommen.

Die Anforderungen, die während der, kürzer als geplant ausgefallenen, Analyse und im Entwurf gestellt wurden, sind zu etwa 90 Prozent erfüllt worden.

Projektphase	geplante Zeit
Analyse	4h
Entwurf	4h
Implementierung	24h
Dokumentation	8h
Gesamt	40h

Tabelle 3: Projektphasen

5.2 Lesson Learned

Abschließend kann ich feststellen, dass ein Design mit dem Window-Builder und Eclipse nicht zu empfehlen ist. Dadurch, dass alle Anforderungen an ein Benutzerfreundliches Design sehr Zeitaufwendig im Gegensatz zu anderen Entwicklungsumgebungen sind, dient es meiner Meinung nach nur für kleinere Oberflächen für Zweck Programme die wenige Eingaben erfordern.

Gut hervorheben kann ich die einfache Einbindung von HTTP und SSL Klassen und die Verwendung von SQLite Datenbanken. Das hat sehr gut funktioniert.