



HTBL Imst
AUFBAULEHRGANG FÜR INFORMATIK



Übungszettel – Datenpersistenz

Name: Michael Bogensberger

Datum: 02.02.2022

1	Inhalt	
2	Entwicklungsumgebung einrichten.....	3
3	Projekt anlegen.....	3
3.1	MySQL Connector Dependency hinzufügen.....	3
4	Verbindung zu Datenbank herstellen.....	3
5	Einfaches CRUD Beispiel mit JDBC	4
5.1	CRUD Beispiel erweitern.....	4
6	JDBC-Zugriff mit dem DAO Entwurfsmuster.....	5
6.1	DAO – Data Access Object	5
6.2	Singleton Pattern.....	6
6.3	Cli Klasse	7
6.4	Domain Package	8
6.5	Base Repository.....	8
6.6	MyCourseRepository Repository.....	9
6.7	MySQLCourseRepository Klasse	9
6.8	showAllCourses Methode	10
6.9	Assert Klasse.....	11
6.10	Kurs durch ID bekommen (getById Methode)	12
6.11	Neuen Eintrag hinzufügen (insert Methode)	12
6.12	Cli anpassen (insert Methode)	13
6.13	Kurs updaten (update Methode).....	13
6.14	Cli anpassen (update Methode)	14
6.15	Kurs löschen (deleteById Methode).....	15
6.16	Kurs durch Namen oder Beschreibung finden	15
6.17	Alle laufende Kurse finden.....	16
6.18	UML Diagramme.....	17
7	JDBC und DAO – Studenten	18
8	JDBC und DAO – Buchungen.....	18
	Alle laufende Kurse finden	Fehler! Textmarke nicht definiert.
9	ResultSet.next() Hinweis	19
10	Abbildungsverzeichnis.....	20

2 Entwicklungsumgebung einrichten

Zunächst müssen wir die Entwicklungsumgebung einrichten. Dafür installieren wir uns zunächst XAMPP. Der Name "XAMPP" ist eine Abkürzung für Apache, MySQL, Perl und PHP. Das "X" am Anfang bezieht sich darauf, dass das Programm auf verschiedenen Betriebssystemen wie Windows, Linux oder Mac OS X läuft. Gegebenenfalls muss man in XAMPP Ports ändern. Falls man zum Beispiel die MySQL Workbench installiert hat, kann es sein, dass der Port 3306 bereits belegt ist. Hierfür empfiehlt sich auf den Port 3307 zu wechseln. Danach kann man im Browser die URL: localhost aufrufen. Von dort an kann man in phpMyAdmin einsteigen.

3 Projekt anlegen

Als nächsten legen wir in IntelliJ ein neues Maven Projekt an. Dafür wählen wir Maven aus und verwenden dieses Mal keinen Archetype.

3.1 MySQL Connector Dependency hinzufügen

Nun fügen wir die MySQL Connector Dependency hinzu. Diese wird benötigt, um mit der Datenbank kommunizieren zu können. Dazu fügen wir folgendes in der pom.xml hinzu.

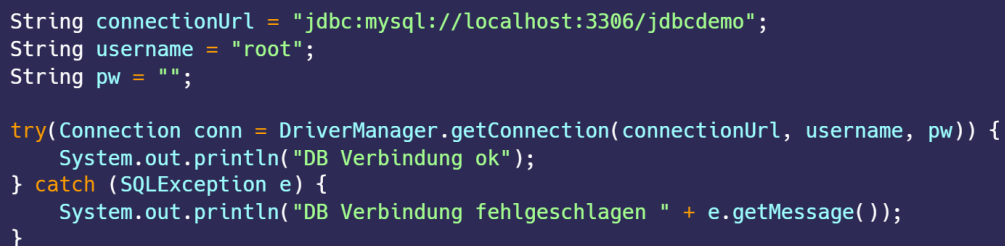


```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.27</version>
  </dependency>
</dependencies>
```

Abbildung 1 MySQL Connector Dependency

4 Verbindung zu Datenbank herstellen

In folgendem Codeabschnitt wird dargestellt, wie man mithilfe der MySQL Connector Dependency eine Verbindung zur Datenbank herstellen kann.



```
String connectionUrl = "jdbc:mysql://localhost:3306/jdbcdemo";
String username = "root";
String pw = "";

try(Connection conn = DriverManager.getConnection(connectionUrl, username, pw)) {
    System.out.println("DB Verbindung ok");
} catch (SQLException e) {
    System.out.println("DB Verbindung fehlgeschlagen " + e.getMessage());
}
```

Abbildung 2 Datenbankverbindung herstellen

5 Einfaches CRUD Beispiel mit JDBC

Nun war es die Aufgabe mithilfe der Videos ein einfaches CRUD Programm mithilfe von JDBC zu gestalten. Dabei geht es im Wesentlichen um das Auslesen von Studenten, dem Einfügen von Studenten sowie dem Löschen von Studenten. Der Code zum Beispiel ist hier zu finden: [jdbcCrudExample](#).

Hier ist der Aufbau der Student Tabelle zu sehen:

					id	name	email
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	2 Niklas niklas@g.at
<input type="checkbox"/>		Bearbeiten		Kopieren		Löschen	3 Michael mbogensberger@gmail.com

Table 1 Student Table

5.1 CRUD Beispiel erweitern

Als nächstes sollten wir das vorherige Beispiel um eine weitere Tabelle ergänzen und dieses in Java implementieren. Dazu habe ich der Datenbank eine Kurs-Tabelle hinzugefügt. Jeder Student hat mehrere Kurse. Jeder Kurs hat zudem mehrere Studenten. Der Code ist im Gleichen GitHub Repository zu finden wie in der vorherigen Aufgabe.

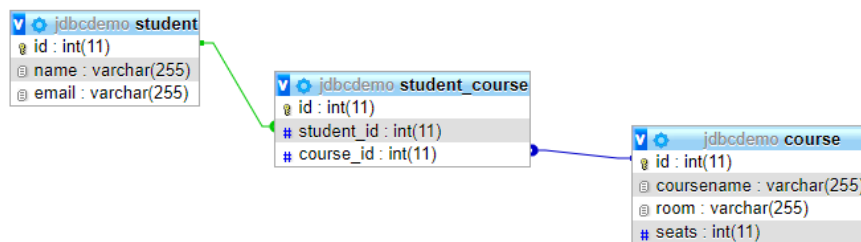


Table 2 Student - Course - Modell

6 JDBC-Zugriff mit dem DAO Entwurfsmuster

Die nächste Aufgabe war es, sich mithilfe der Videos sich mit dem DAO Entwurfsmuster vertraut zu machen. Der Code zur folgenden Aufgabe ist hier zu finden: [GitHub Repo](#)

6.1 DAO – Data Access Object

DAO ist ein Entwurfsmuster das einem ermöglicht den Zugriff auf Daten so zu kapseln, das jene Datenquelle relativ einfach getauscht werden kann. Dadurch wird die Programmlogik von technischen Details der Datenspeicherung befreit. Man will also nicht den bestehenden Code angreifen müssen, um Funktionalität hinzufügen zu können. DAO wird also zwischen der Datenbank und dem Code geschaltet. Dadurch wird die Kopplung auf ein Minimum heruntergefahren. Wenn sich der JDBC Treiber zum Beispiel ändert, ist der grundsätzliche Code unabhängig. Das entkoppeln ist das zentrale des DAO Design Patterns.

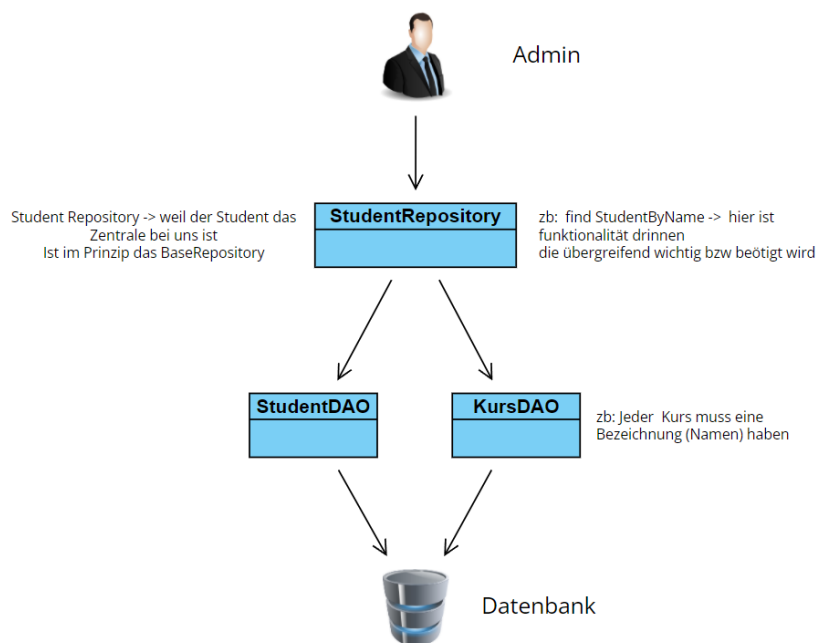


Abbildung 3 Kurssystem DAO

Wie wir sehen können bringt uns also DAO den großen Vorteil der geringen Kopplung. Dies ist auch nochmal in folgender Grafik veranschaulicht.

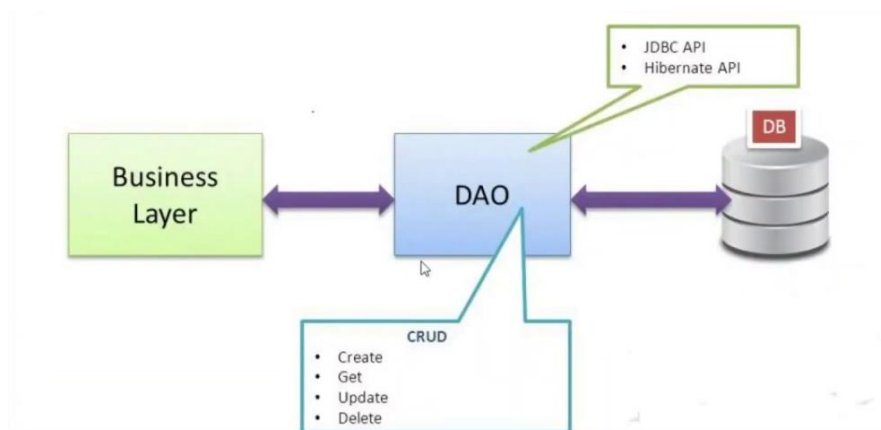
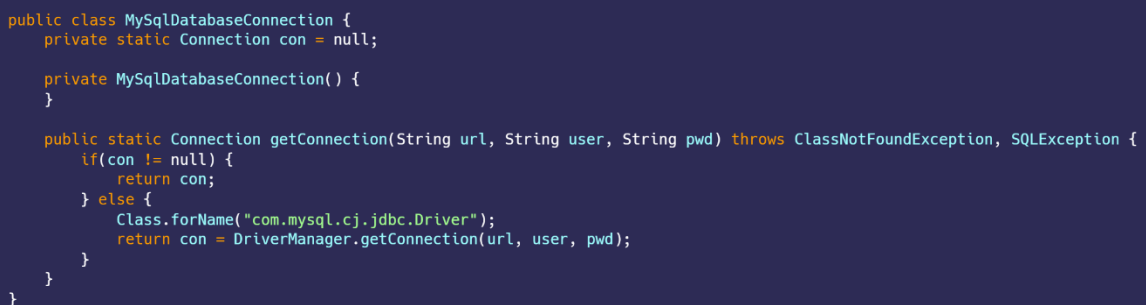


Abbildung 4 DAO Design Pattern / Kopplung

6.2 Singleton Pattern

Um die MySQL Connection nur einmal aufbauen zu können verwenden wir das Singleton Pattern. Beim Singleton Pattern handelt es sich in Java um genau eine Klasse. Diese darf nur ein einziges Mal instanziiert werden. Während der Programmlaufzeit existiert nur ein einziges Objekt. Das Singleton Pattern wurde zudem schon einmal von mir beim Übungszettel Mikroarchitektur beschrieben.

Dabei erstellen wir die `MySQLDatabaseConnection` Klasse. Diese liefert die Connection zurück. Hier ist es eben wichtig dass nur eine Connection existiert. Dabei setzen wir den Konstruktor auf `private`. Somit kann dieser nicht aufgerufen werden. Danach checken wir ob die Connection bereits existiert. Existiert diese nicht, wird eine neue Connection erstellt und zurückgegeben.

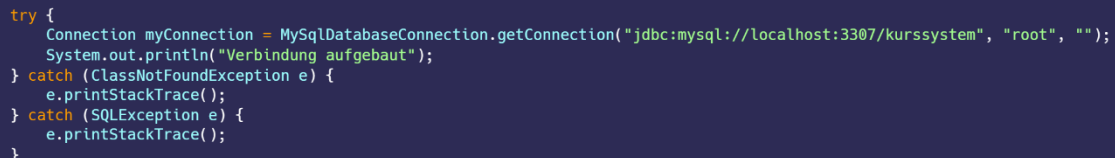


```
public class MySQLDatabaseConnection {
    private static Connection con = null;

    private MySQLDatabaseConnection() {
    }

    public static Connection getConnection(String url, String user, String pwd) throws ClassNotFoundException, SQLException {
        if (con != null) {
            return con;
        } else {
            Class.forName("com.mysql.cj.jdbc.Driver");
            return con = DriverManager.getConnection(url, user, pwd);
        }
    }
}
```

Abbildung 5 MySQLDatabaseConnection Klasse



```
try {
    Connection myConnection = MySQLDatabaseConnection.getConnection("jdbc:mysql://localhost:3307/kurssystem", "root", "");
    System.out.println("Verbindung aufgebaut");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}
```

Abbildung 6 Verbindungserstellung in der Main Methode

6.3 Cli Klasse

Für die Ausgabe erstellen wir uns eine eigene Cli Klasse. Dort erstellen wir uns im Konstruktor einen Scanner und ein Switch Case mit dem wir durch die verschiedenen Auswahlmöglichkeiten iterieren.

```
public class Cli {  
    Scanner scan;  
  
    public Cli() {  
        this.scan = new Scanner(System.in);  
    }  
  
    public void start() {  
        String input = "-";  
  
        while(!input.equals("x")) {  
            showMenue();  
            input = scan.nextLine();  
            switch(input) {  
                case "1":  
                    System.out.println("Kurseingabe");  
                    break;  
                case "2":  
                    System.out.println("Alle Kurse anzeigen");  
                    break;  
                case "x":  
                    System.out.println("beenden");  
                    break;  
                default:  
                    inputError();  
                    break;  
            }  
        }  
        scan.close();  
    }  
  
    private void inputError() {  
        System.out.println("Bitte nur die Zahlen der Menüauswahl eingeben!");  
    }  
  
    private void showMenue() {  
        System.out.println("----- KURSMANAGEMENT -----");  
        System.out.println("(1) Kurs eingeben \t (2) Alle Kurse anzeigen \t (x) Ende");  
    }  
}
```

Abbildung 7 Cli Klasse

6.4 Domain Package

Nun erstellen wir uns ein Domain Package und darin die Entitäten als Klassen. Zunächst legen wir eine Course Klasse und eine Course Type Klasse an. Die Course Type Klasse ist ein Enum und enthält die verschiedenen Course Typen. Danach erstellen wir eine BaseEntity Klasse. Jene ist dafür da, die ID nicht immer händisch eintragen zu müssen. Sie kümmert sich also um die Logik der IDs. In der Course Klasse haben wir die jeweiligen Getter und Setter und zwei Konstruktoren. Einen bei dem die ID mit übergeben wird und gesetzt wird, und einen wo keine ID mit übergeben wird. In diesem Fall wird die ID auf NULL gesetzt.

Nun schaut unser Package folgendermaßen aus:

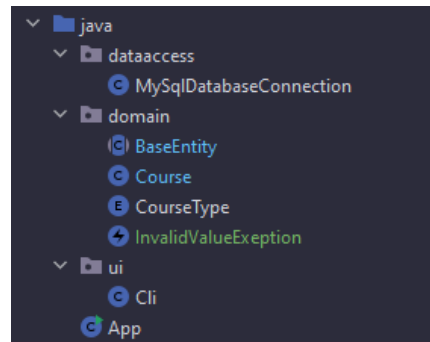


Abbildung 8 Package Struktur

6.5 Base Repository

Nun erstellen wir uns ein BaseRepository. Dieses kann ich nun für jede Art von Entität verwenden. Dabei werden Generics verwendet. Dabei gibt es einmal T für den Typen und I für die ID. Folgende Methoden müssen implementiert werden.

```
public interface BaseRepository<T,I> {  
  
    Optional<T> insert(T entity);  
    Optional<T> getById(I id);  
    List<T> getAll();  
    Optional<T> update(T entity);  
    void deleteById(I id);  
  
}
```

Abbildung 9 BaseRepository

6.6 MyCourseRepository Repository

Nun erstellen wir ein CourseRepository. Dieses erbt das BaseRepository. Dabei wird dieses mit Course und Long getypt. Course als Entitytyp und Long als Schlüsseltyp. Der Vorteil ist, dass ich das BaseRepository immer verwenden kann und jetzt zb im CourseRepository zusätzlich Funktionen spezifizieren.

```
public interface MyCourseRepository extends BaseRepository<Course, Long>{

    List<Course> findAllCoursesByName(String name);
    List<Course> findAllCoursesByDescription(String description);
    List<Course> findAllCoursesByNameOrDescription(String searchText);
    List<Course> findAllCoursesByStartDate(Date startDate);
    List<Course> findAllCoursesByCourseType(CourseType courseType);
    List<Course> findAllRunningCourses();

}
```

Abbildung 10 MyCourseRepository

6.7 MySqlConnectionRepository Klasse

Als nächstes erstellen wir eine MySqlConnectionRepository Klasse. Hier wird jetzt klar gemacht welche Technologie (MySQL) verwendet wird. Diese Klasse implementiert nun die MyCourseRepository Klasse. Nun müssen wir also alle Methoden der beiden obigen Repositories implementieren.

Dazu erstellen wir uns zunächst eine Connection. Da wir ja die MySqlConnectionDatabaseConnection Klasse haben können wir diese dafür verwenden.

```
private Connection con;

public MySqlConnectionRepository() throws SQLException, ClassNotFoundException {
    this.con = MySqlConnectionDatabaseConnection.getConnection("jdbc:mysql://localhost:3306/kurssystem", "root", "");
}
```

Abbildung 11 Database Connection (MySqlConnectionRepository)

Nun fangen wir an die getAll() Methode mit Funktionalität zu befüllen. In der while Schleife befüllen wir nun die ArrayList mit den jeweiligen Werten. Hier sollte man genau auf die Labels achten! Nun haben wir ja eine Enumeration für den Kurstypen (CourseType). Wir bekommen jedoch einen String aus der DB zurück. Deshalb müssen wir hier den String noch in einen CourseType konvertieren. Diese sieht nun wie folgt aus:

```
@Override
public List<Course> getAll() {
    String sql = "SELECT * FROM courses";
    try {
        PreparedStatement preparedStatement = con.prepareStatement(sql);
        ResultSet resultSet = preparedStatement.executeQuery();
        ArrayList<Course> courseList = new ArrayList<>();
        while(resultSet.next()) {
            courseList.add(
                new Course(
                    resultSet.getLong("id"),
                    resultSet.getString("name"),
                    resultSet.getString("description"),
                    resultSet.getInt("hours"),
                    resultSet.getDate("begindate"),
                    resultSet.getDate("enddate"),
                    CourseType.valueOf(resultSet.getString("coursetype"))
                )
            );
        }
        return courseList;
    } catch (SQLException e) {
        throw new DatabaseException("Database error occurred!");
    }
}
```

Abbildung 12 getAll Methode (MySqlConnectionRepository)

Nun gehen wir in die Cli Klasse und fügen im Konstruktor ein MyCourseRepository hinzu. Danach fügen wir im Switch Case im zweiten Case die Methode showAllCourses hinzu. Diese müssen wir natürlich noch implementieren.

```

Scanner scan;
MyCourseRepository repo;

public Cli(MyCourseRepository repo) {
    this.repo = repo;
    this.scan = new Scanner(System.in);
}

```

Abbildung 13 Cli Klasse (repo)

6.8 showAllCourses Methode

Nun können wir die showAllCourses Methode implementieren. Dazu erstellen wir zunächst eine List vom Typ Course. Danach befüllen wir die Liste über das Repository mit getAll(). Nun checken wir ob die liste leer ist. Ist sie es nicht, gehen wir durch die jeweiligen Kurse in der Liste durch und geben diese aus. Dazu haben wir ja auch die toString Methode implementiert. Jene wird nun im „sout“ verwendet. Danach umgeben wir das ganze noch mit einem try catch und sagen was passiert wenn Fehler auftreten.

```

private void showAllCourses() {
    List<Course> list = null;
    try {
        list = repo.getAll();
        if (list.size() > 0) {
            for (Course course : list) {
                System.out.println(course);
            }
        } else {
            System.out.println("Kursliste leer!");
        }
    } catch (DatabaseException databaseException) {
        System.out.println("Datenbankfehler bei Anzeige aller Kurse: " + databaseException.getMessage());
    } catch (Exception exception) {
        System.out.println("Unbekannter Fehler bei anzeige aller Kurse: " + exception.getMessage());
    }
}

```

Abbildung 14 showAllCourses Methode

Schließlich können wir in die App Klasse bzw. Main Methode wechseln. Hier erstellen wir ein Cli Objekt. Das wichtige ist nun, dass wir dem Cli Objekt ein MySqlCourseRepository mitgeben.

```

Cli myCli = null;

try {
    myCli = new Cli(new MySqlCourseRepository());
    myCli.start();
} catch (SQLException e) {
    System.out.println("Datenbankfehler: " + e.getMessage() + " SQL State: " + e.getSQLState() );
} catch (ClassNotFoundException e) {
    System.out.println("Datenbankfehler: " + e.getMessage() );
}

```

Abbildung 15 App Klasse

Das Zusammenspiel mit den Repositories wird in folgender Grafik nochmal genauer dargestellt. Hier ist gut zu sehen welche Klasse welche Repositories verwendet. Zurzeit verwendet die Cli nur das MyCourseRepository. Das kann sich jedoch mit zusätzlicher Funktionalität ändern.

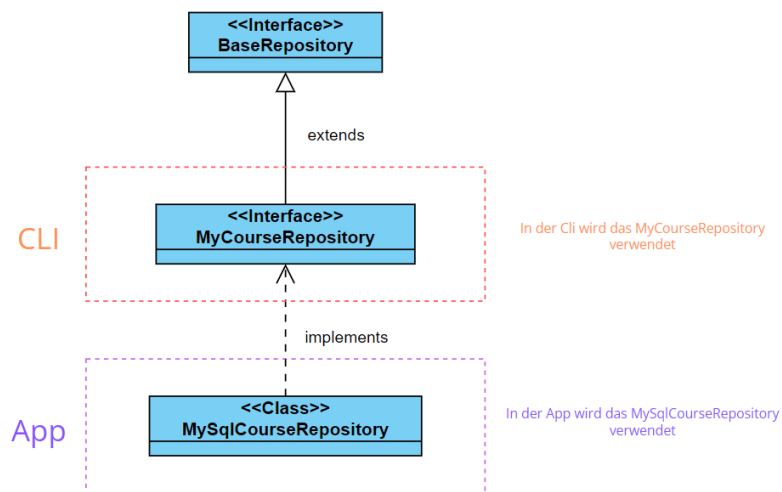


Abbildung 16 Grafik - Repositories

6.9 Assert Klasse

Nun erstellen wir uns ein util Package. Darin erstellen wir uns eine Assert Klasse. Diese dient dazu zu checken ob ein Objekt null ist oder nicht.

```
public class Assert {
    public static void notNull(Object o) {
        if (o == null) {
            throw new IllegalArgumentException("Reference must not be null");
        }
    }
}
```

Abbildung 17 Assert Klasse

6.10 Kurs durch ID bekommen (getById Methode)

Nun programmieren wir die Funktionalität der getById Methode.

```
@Override
public Optional<Course> getById(Long id) {
    Assert.notNull(id);
    if(countCoursesInDbWithId(id) == 0) {
        return Optional.empty();
    } else {
        try {
            String sql = "SELECT * FROM courses WHERE id = ?";
            PreparedStatement preparedStatement = con.prepareStatement(sql);
            preparedStatement.setLong(1, id);
            ResultSet resultSet = preparedStatement.executeQuery();
            resultSet.next();

            Course course = new Course(
                resultSet.getLong("id"),
                resultSet.getString("name"),
                resultSet.getString("description"),
                resultSet.getInt("hours"),
                resultSet.getDate("begindate"),
                resultSet.getDate("enddate"),
                CourseType.valueOf(resultSet.getString("coursetype"))
            );

            return Optional.of(course);
        } catch (SQLException sqlException) {
            throw new DatabaseException(sqlException.getMessage());
        }
    }
}
```

Abbildung 18 getById Methode

6.11 Neuen Eintrag hinzufügen (insert Methode)

Nun programmieren wir die insert Methode. Wie wir sehen ähnelt diese Methode ebenfalls den anderen. Wichtig ist hier jedoch, dass wir oben beim preparedStatement noch sagen, dass wir die generierten Keys zurückbekommen wollen. Unten erstellen wir uns dann ein ResultSet mit den generierten Schlüsseln. Wenn nun ein Key im ResultSet existiert, rufen wir die getById Methode auf und übergeben als ID die erste Stelle vom ResultSet, da wir ja nur einen Datensatz einfügen können wir das gleich schon so definieren.

```
@Override
public Optional<Course> insert(Course entity) {
    Assert.notNull(entity);

    try {
        String sql = "INSERT INTO `courses` ( `name`, `description`, `hours`, `begindate`, `enddate`, `coursetype` ) VALUES ( ?,?,?,?,? )";
        PreparedStatement preparedStatement = con.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
        preparedStatement.setString(1, entity.getName());
        preparedStatement.setString(2, entity.getDescription());
        preparedStatement.setInt(3, entity.getHours());
        preparedStatement.setDate(4, entity.getBeginDate());
        preparedStatement.setDate(5, entity.getEndDate());
        preparedStatement.setString(6, entity.getCourseType().toString());

        int affectedRows = preparedStatement.executeUpdate();
        if(affectedRows == 0) {
            return Optional.empty();
        }

        ResultSet generatedKeys = preparedStatement.getGeneratedKeys();
        if(generatedKeys.next()) {
            return this.getById(generatedKeys.getLong(1));
        } else {
            return Optional.empty();
        }
    } catch (SQLException sqlException) {
        throw new DatabaseException(sqlException.getMessage());
    }
}
```

Abbildung 19 insert Methode

6.12 Cli anpassen (insert Methode)

Nun passen wir noch die Cli Klasse an. Wichtig ist hier, dass wir alle Werte einlesen und alle Fehlerprüfungen durchführen. Danach legen wir einen neuen Kurs (course Objekt) mit den eingegebenen Daten an und übergeben diesen der insert Methode. Wenn das Optional infolgedessen nicht leer ist, geben wir den Kurs aus.

```
Optional<Course> optionalCourse = repo.insert(
    new Course(name,description,hours,dateFrom,dateTo,courseType)
);

if(optionalCourse.isPresent()) {
    System.out.println("Kurs angelegt: " +optionalCourse.get());
} else {
    System.out.println("Kurs konnte nicht angelegt werden!");
}
```

Abbildung 20 Cli - insert

6.13 Kurs updaten (update Methode)

Als nächstes programmieren wir die update Methode. Hier übergeben wir einen Kurs und updaten die jeweiligen Felder basierend auf der ID des übergebenen Kurses. Diese sieht dann wie folgt aus:

```
@Override
public Optional<Course> update(Course entity) {
    Assert.notNull(entity);
    String sql = "UPDATE `courses` SET `name` = ?, `description` = ?, `hours` = ?, `begindate` = ?, `enddate` = ?, `coursetype` = ? WHERE `courses`.`id` = ?";

    if(countCoursesInDbWithId(entity.getId()) == 0) {
        return Optional.empty();
    }

    try {
        PreparedStatement preparedStatement = con.prepareStatement(sql);
        preparedStatement.setString(1, entity.getName());
        preparedStatement.setString(2, entity.getDescription());
        preparedStatement.setInt(3, entity.getHours());
        preparedStatement.setDate(4, entity.getBeginDate());
        preparedStatement.setDate(5, entity.getEndDate());
        preparedStatement.setString(6, entity.getCourseType().toString());
        // for WHERE id = ?
        preparedStatement.setLong(7, entity.getId());

        int affectedRows = preparedStatement.executeUpdate();

        if(affectedRows == 0) {
            return Optional.empty();
        } else {
            return this.getById(entity.getId());
        }
    } catch (SQLException sqlException) {
        throw new DatabaseException(sqlException.getMessage());
    }
}
```

Abbildung 21 update Methode

6.14 Cli anpassen (update Methode)

Natürlich müssen wir nun wieder die Cli Klasse anpassen. Wichtig ist hier, dass wir die ID von jenem Kurs bekommen der zu ändern ist. Danach suchen wir uns den Kurs mit der übergebenen ID heraus.

```
Optional<Course> courseOptional = repo.getById(courseId);
```

Danach geben wir den gefundenen Kurs aus und lesen die restlichen Werte ein. Wir sagen aber, dass wenn man für zb die Beschreibung nicht eingibt (nur Enter drückt) nichts geändert werden soll.

Nun rufen wir die update Methode auf. Hier können wir auf jeden Fall mal die ID übergeben. Nun sagen wir, wenn zb der übergebene Name leer ist, dann füllen wir hier einfach den Namen vom bereits gefundenen Kurs ein. Andernfalls übergeben wir den eingebenden Namen.

```
Optional<Course> optionalCourseUpdated = repo.update(
    new Course(
        course.getId(),
        name.equals("") ? course.getName() : name,
        description.equals("") ? course.getDescription() : description,
        hours.equals("") ? course.getHours() : Integer.parseInt(hours),
        dateFrom.equals("") ? course.getBeginDate() : Date.valueOf(dateFrom),
        dateTo.equals("") ? course.getEndDate() : Date.valueOf(dateTo),
        courseType.equals("") ? course.getCourseType() : CourseType.valueOf(courseType)
    )
);
```

Abbildung 22 update - neues Kurs Objekt

Hierbei handelt es sich um eine kurzschreibweise einer IF-Else Anweisung. Dies funktioniert folgendermaßen:

```
condition ? expression1 : expression2 ;
```

Abbildung 23 if-else short version

Abschließen rufen wir noch `ifPresentOrElse()` auf dem geupdateten Kurs auf. Hier können wir sagen, wenn es vorhanden ist, geben wir den Kurs (c) aus. Andernfalls geben wir aus, dass der Kurs nicht geändert werden konnte.

```
optionalCourseUpdated.ifPresentOrElse(
    (c) -> System.out.println("Kurs aktualisiert: " + c),
    () -> System.out.println("Kurs konnte nicht aktualisiert werden!")
);
```

Abbildung 24 update - ifPresentOrElse

6.15 Kurs löschen (deleteById Methode)

Nun programmieren wir die deleteById Methode. Diese ist vergleichsweise einfach gestrickt. Natürlich müssen wir dann auch wieder die Cli Klasse anpassen. Dazu wird im Repository nun nur folgender Code benötigt:

```
@Override
public void deleteById(Long id) {
    Assert.notNull(id);
    String sql = "DELETE FROM courses WHERE id = ?";

    try {

        if(countCoursesInDbWithId(id) == 1) {
            PreparedStatement preparedStatement = con.prepareStatement(sql);
            preparedStatement.setLong(1, id);
            preparedStatement.executeUpdate();
        }

    } catch (SQLException sqlException) {
        throw new DatabaseException(sqlException.getMessage());
    }
}
```

Abbildung 26 deleteById Methode

6.16 Kurs durch Namen oder Beschreibung finden

Nun programmieren wir die findAllCoursesByNameOrDescription Methode. Dazu haben wir folgendes SQL Statement. Die „LOWER“ Anweisung setzt die Werte lediglich auf Lower Case. Nun fügen wir wieder die Werte in die Fragezeichen ein. Hier haben wir noch jeweils Prozentzeichen. Diese dienen dazu, alle Werte zu finden die irgendwo jenen übergebenen String beinhalten. Danach gehen wir wieder alle zurückgebenden Zeilen durch und fügen jeweils die Kurse in eine ArrayList hinzu. Danach geben wir diese einfach zurück.

```
@Override
public List<Course> findAllCoursesByNameOrDescription(String searchText) {
    try {
        String sql = "SELECT * FROM courses WHERE LOWER(description) LIKE LOWER(?) OR LOWER(name) LIKE ?";
        PreparedStatement preparedStatement = con.prepareStatement(sql);
        preparedStatement.setString(1, "%" + searchText + "%");
        preparedStatement.setString(2, "%" + searchText + "%");
        ResultSet resultSet = preparedStatement.executeQuery();

        ArrayList<Course> courseList = new ArrayList<>();

        while(resultSet.next()) {
            courseList.add(
                new Course(
                    resultSet.getLong("id"),
                    resultSet.getString("name"),
                    resultSet.getString("description"),
                    resultSet.getInt("hours"),
                    resultSet.getDate("begindate"),
                    resultSet.getDate("enddate"),
                    CourseType.valueOf(resultSet.getString("coursetype"))
                ));
        }

        return courseList;
    } catch (SQLException sqlException) {
        throw new DatabaseException(sqlException.getMessage());
    }
}
```

Abbildung 27 findAllCoursesByNameOrDescription Methode

6.17 Alle laufende Kurse finden

Als nächstes implementieren wir die `findAllRunningCourses` Methode. Diese ähnelt wieder stark der obigen Methode. Das wichtigste was sich hier ändert ist das SQL Statement. Nun schaut das ganze folgendermaßen aus:

A screenshot of a code editor with a dark blue background and light green text. The code is in Java and implements the `findAllRunningCourses` method. It starts with an `@Override` annotation, followed by the method signature `public List<Course> findAllRunningCourses() {`. Inside, a SQL query is defined: `String sql = "SELECT * FROM courses WHERE NOW() < enddate";`. A `try` block contains the logic to prepare the statement, execute the query, and iterate through the results. The `while` loop uses `resultSet.next()` to check for more rows. For each row, a `Course` object is created with values from the `resultSet`. The `catch` block handles `SQLException` by throwing a `DatabaseException`. The method ends with `return courseList;` and a closing brace.

```
@Override
public List<Course> findAllRunningCourses() {
    String sql = "SELECT * FROM courses WHERE NOW() < enddate";
    try {
        PreparedStatement preparedStatement = con.prepareStatement(sql);
        ResultSet resultSet = preparedStatement.executeQuery();
        ArrayList<Course> courseList = new ArrayList<>();

        while (resultSet.next()) {
            courseList.add(
                new Course(
                    resultSet.getLong("id"),
                    resultSet.getString("name"),
                    resultSet.getString("description"),
                    resultSet.getInt("hours"),
                    resultSet.getDate("begindate"),
                    resultSet.getDate("enddate"),
                    CourseType.valueOf(resultSet.getString("coursetype"))
                ));
        }
        return courseList;
    } catch (SQLException sqlException) {
        throw new DatabaseException(sqlException.getMessage());
    }
}
```

Abbildung 28 `findAllRunningCourses` Methode

6.18 UML Diagramme

Der nun erstellte Code lässt sich in folgendem UML Diagramm darstellen:

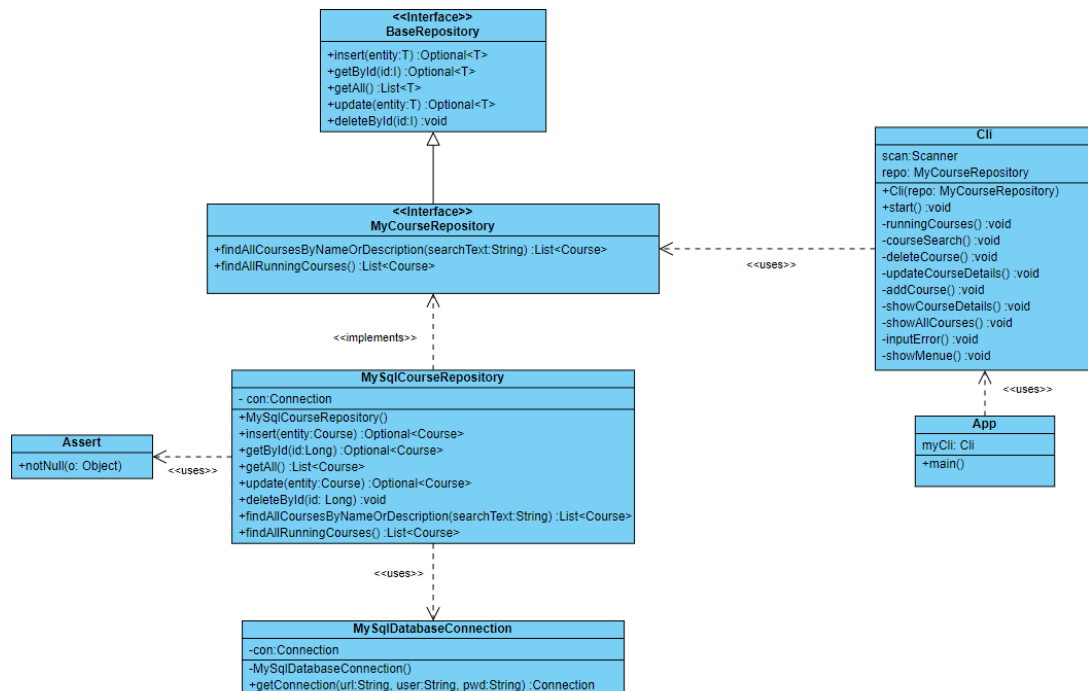


Abbildung 29 DAO - UML Diagramm

Für das Domain Package ergibt sich folgendes UML Diagramm:

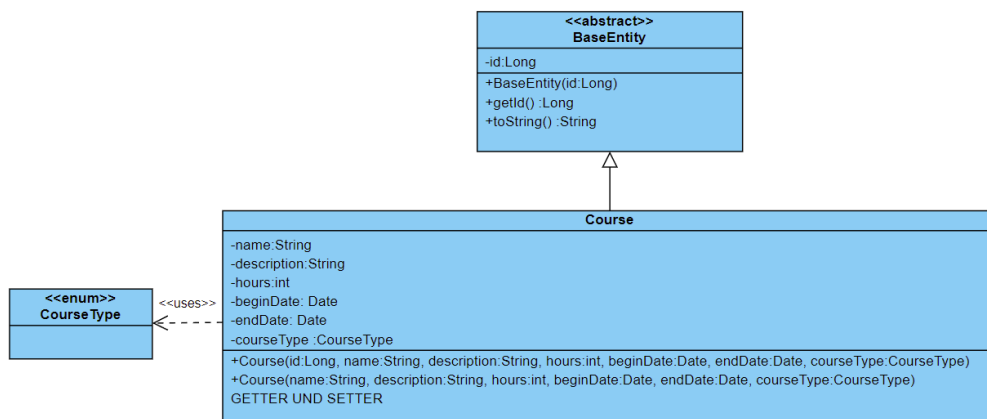


Abbildung 30 DAO - Domain Package - UML

7 JDBC und DAO – Studenten

Erweitere die fertig nachprogrammierte Applikation mit einem DAO für CRUD für eine neue Domänenklasse „Student“:

- Studenten haben eine Student-ID, einen VN, einen NN, ein Geburtsdatum
- Domänenklasse implementieren (Setter absichern, neue Exceptions definieren, Business-Regeln selbst wählen - z.B. dass Name nicht leer sein darf)
- Eigenes Interface MyStudentRepository von BaseRepository ableiten. MyStudentRepository muss mindestens 3 studentenspezifische Methoden enthalten (z.B. Studentensuche nach Namen, Suche nach ID, Suche nach bestimmtem Geburtsjahr, Suche mit Geburtsdatum zwischen x und y etc.).
- Implementierung von MyStudentRepository durch eine neue Klasse MySqlStudentRepository analog zum MySqlCourseRepository.
- Erweiterung des CLI für die Verarbeitung von Studenten und für spezifische Studenten-Funktionen (z.B. Student nach dem Namen suchen)

Der Sourcecode zur Übung ist unter folgendem Link im GitHub zu finden: [JDBC-Student-Übung](#)

8 JDBC und DAO – Buchungen

Gib einen textuellen Vorschlag ab, wie man die bisher programmierte Applikation für die Buchung von Kursen durch Studenten erweitern könnte. Beschreibe, wie eine neue Buchungs-Domänenklasse aussehen sollte, wie man ein DAO für Buchungen dazu entwickeln sollte, wie man die CLI anpassen müsste und welche Anwendungsfälle der Benutzer brauchen könnte (wie etwa „Buchung erstellen“). Verwende zur Illustration insb. auch UML-Diagramme.

Zunächst müsste man eine N zu M Verbindung zwischen dem Studenten und dem Kurs hinzufügen. Die nun erstellte Zwischentabelle wird nun Buchungen genannt. So kann ein Student mehrere Buchungen haben und ein Kurs kann zu mehreren Buchungen gehören. In der Buchung könnte dann z.B. das Buchungsdatum und der Buchungszeitraum stehen.

Im „dataaccess“ Package würde man nun eine MySqlBuchungRepository Klasse und ein MyBuchungRepository Interface brauchen. Im „domain“ Package würde man eine Klasse Buchung brauchen. In der Cli Klasse müsste man einfach die Anwendungsfälle hinzufügen. Diese wären z.B.:

- BaseRepo:
 - Buchung erstellen
 - Buchung updaten
 - Buchung löschen
 - Alle Buchungen ausgeben
 - Spezifische Buchung ausgeben
- MyBuchungRepository:
 - Buchungen zwischen Datum X und Datum Y finden
 - Abgelaufene Buchungen finden
 - Laufende Buchungen finden

In folgender Abbildung wäre eine Mögliche Version der Implementierung als ER-Modell zu sehen:

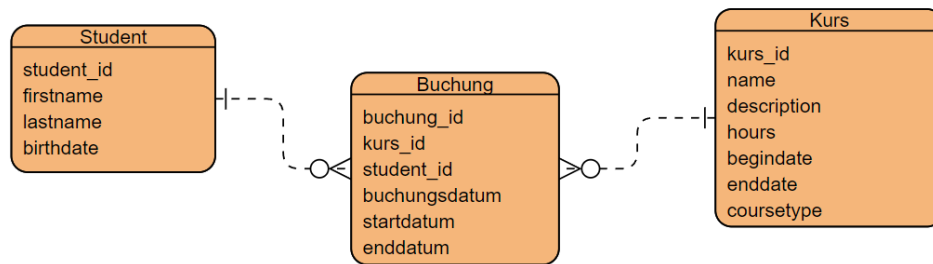


Abbildung 31 Buchung - ER-Modell

Ein dazugehöriges UML Diagramm könnte dann folgendermaßen aussehen:

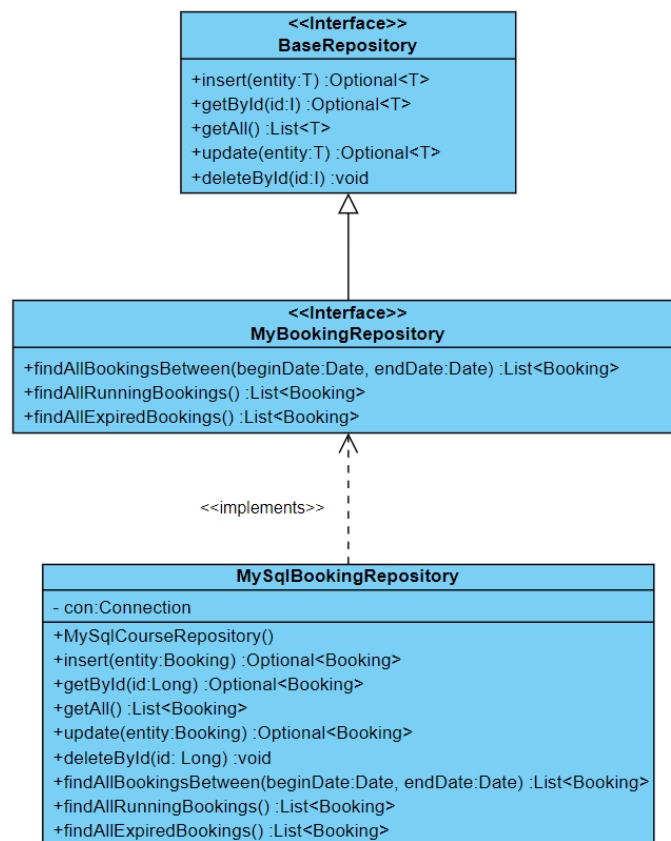


Abbildung 32 UML- Buchung

9 ResultSet.next() Hinweis

Wenn ich mehrere Rows (Zeilen) zurück bekomme ist es klar das ich mit .next() durch die jeweiligen Zeilen springen muss. Wenn ich jedoch nur eine Zeile zurückbekomme muss ich auch einmal .next() aufrufen. Deshalb ja nicht den Befehl .next() vergessen!

10 Abbildungsverzeichnis

Abbildung 1 MySql Connector Dependency.....	3
Abbildung 2 Datenbankverbindung herstellen	3
Abbildung 3 Kurssystem DAO.....	5
Abbildung 4 DAO Design Pattern / Kopplung.....	5
Abbildung 5 MySqlConnection Klasse	6
Abbildung 6 Verbindungserstellung in der Main Methode.....	6
Abbildung 7 Cli Klasse	7
Abbildung 8 Package Struktur	8
Abbildung 9 BaseRepository	8
Abbildung 10 MyCourseRepository	9
Abbildung 11 Database Connection (MySqlCourseRepository)	9
Abbildung 12 getAll Methode (MySqlCourseRepository)	9
Abbildung 13 Cli Klasse (repo)	10
Abbildung 14 showAllCourses Methode.....	10
Abbildung 15 App Klasse.....	10
Abbildung 16 Grafik - Repositories.....	11
Abbildung 17 Assert Klasse	11
Abbildung 18 getByld Methode.....	12
Abbildung 19 insert Methode	12
Abbildung 20 Cli - insert.....	13
Abbildung 21 update Methode	13
Abbildung 22 update - neues Kurs Objekt	14
Abbildung 23 if-else short version	14
Abbildung 24 update - ifPresentOrElse.....	14
Abbildung 25 deleteById Methode	15
Abbildung 26 deleteById Methode.....	15
Abbildung 27 findAllCoursesByNameOrDescription Methode	15
Abbildung 28 findAllRunningCourses Methode	16
Abbildung 29 DAO - UML Diagramm	17
Abbildung 30 DAO - Domain Package - UML.....	17
Abbildung 31 Buchung - ER-Modell	19
Abbildung 32 UML- Buchung.....	19