JDBC

DSS 2018/2019

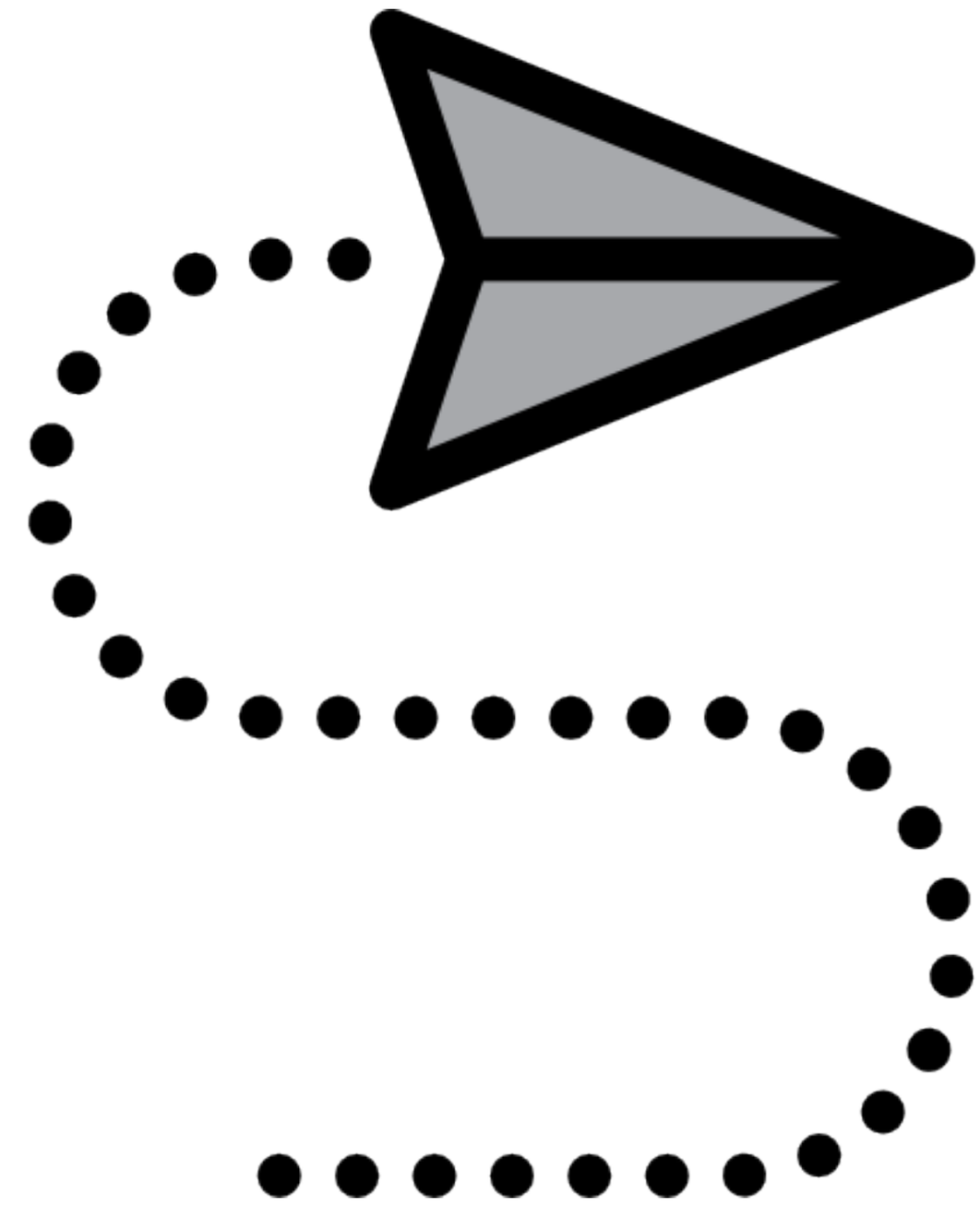
Rui Couto
rui.couto@di.uminho.pt

HASLab
HIGH-ASSURANCE SOFTWARE LABORATORY

INESCTEC
TECNOLOGIA E CIÊNCIA
| LABORATÓRIO ASSOCIADO

Outline

- Introduction
- The *Layers* pattern
- Proposal
- JDBC
- DAO
- Summary



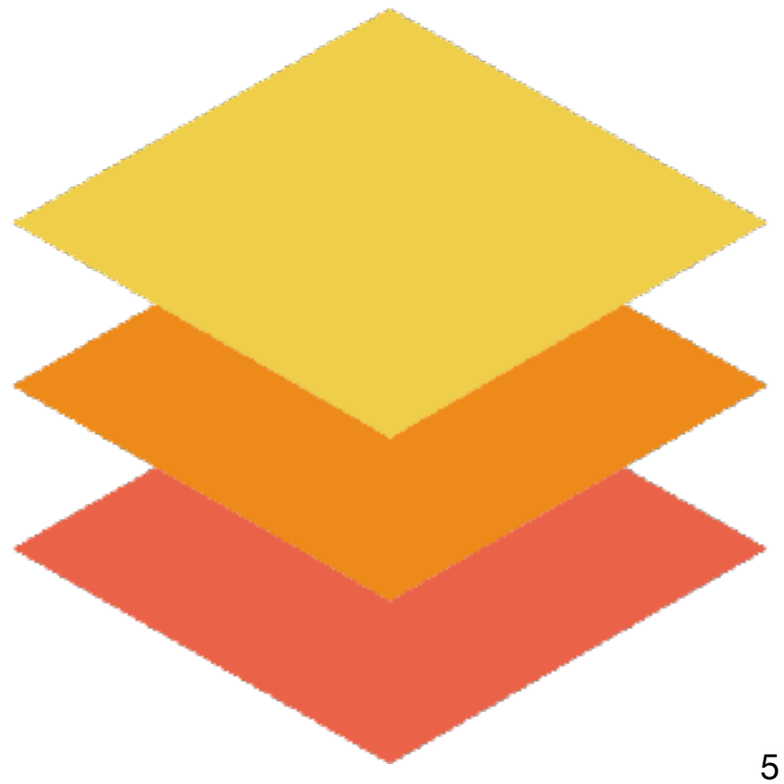
1. Introduction

Introduction

- A correct organisation of the code is:
  - One of the objectives of this course
  - Essential to ensure maintainability and scalability of the code
  - Useful to abstract SQL and language specific details
- The proposed approach is to isolate the SQL, as persistency, from the business logic
- 

2. The *Layers* pattern

From mud to structure



Layers pattern

- The three components correspond to a three layer instance of the *Layers* Pattern.

- *“The layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction“*

  Pattern-Oriented Software Architecture, Buschmann et al. 2001

- A three layer architecture is a common solution:

  - Presentation layer - Contains the classes responsible for the user interface.

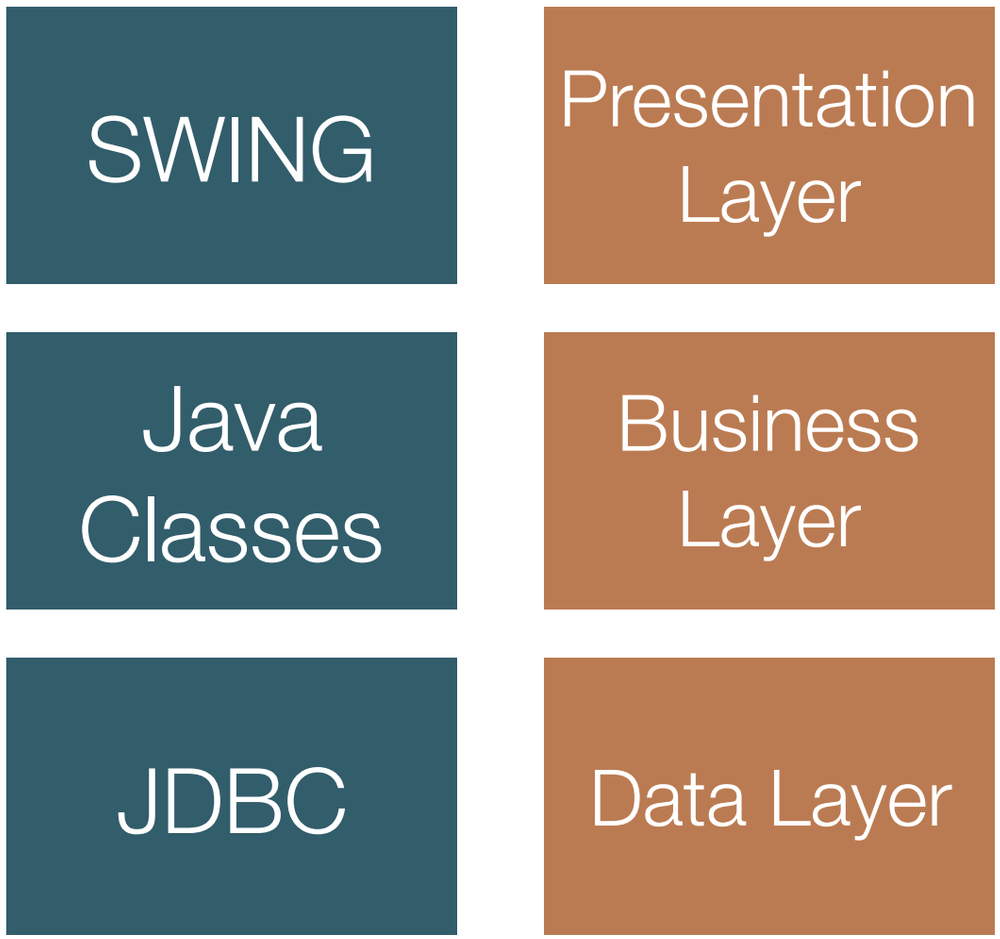
  - Business layer - Contains all the classes responsible for the business logic.

  - Data layer - contains the classes which provide persistence features.

Three layer architecture

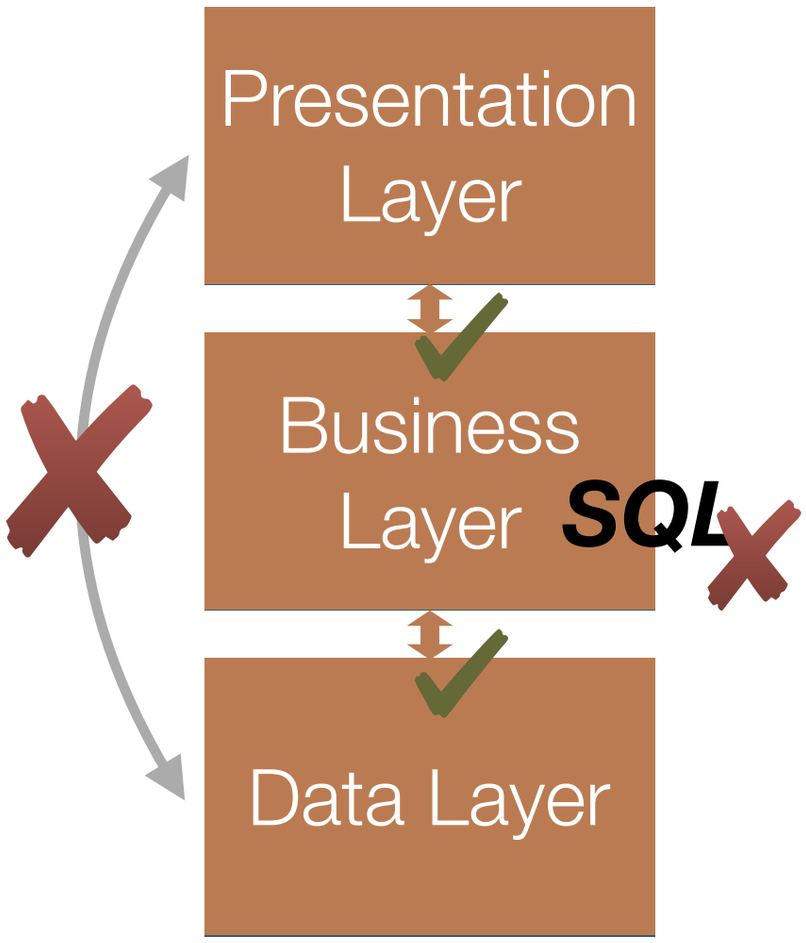
- A typical java application can easily be structured according to the layers pattern.

Three layer architecture



- The presentation layer is supported by SWING (already addressed).
- The business layer corresponds to the usual Java classes.
- The data layer is supported by JDBC.

Three layer architecture



- Some considerations:
  - Each layer communicates **only** with adjacent layers.
  - A layer **should not** deal with other layers responsibilities.
  - Each layer should have, a **facade** to communicate with other layers.

3. Business Layer

A music library

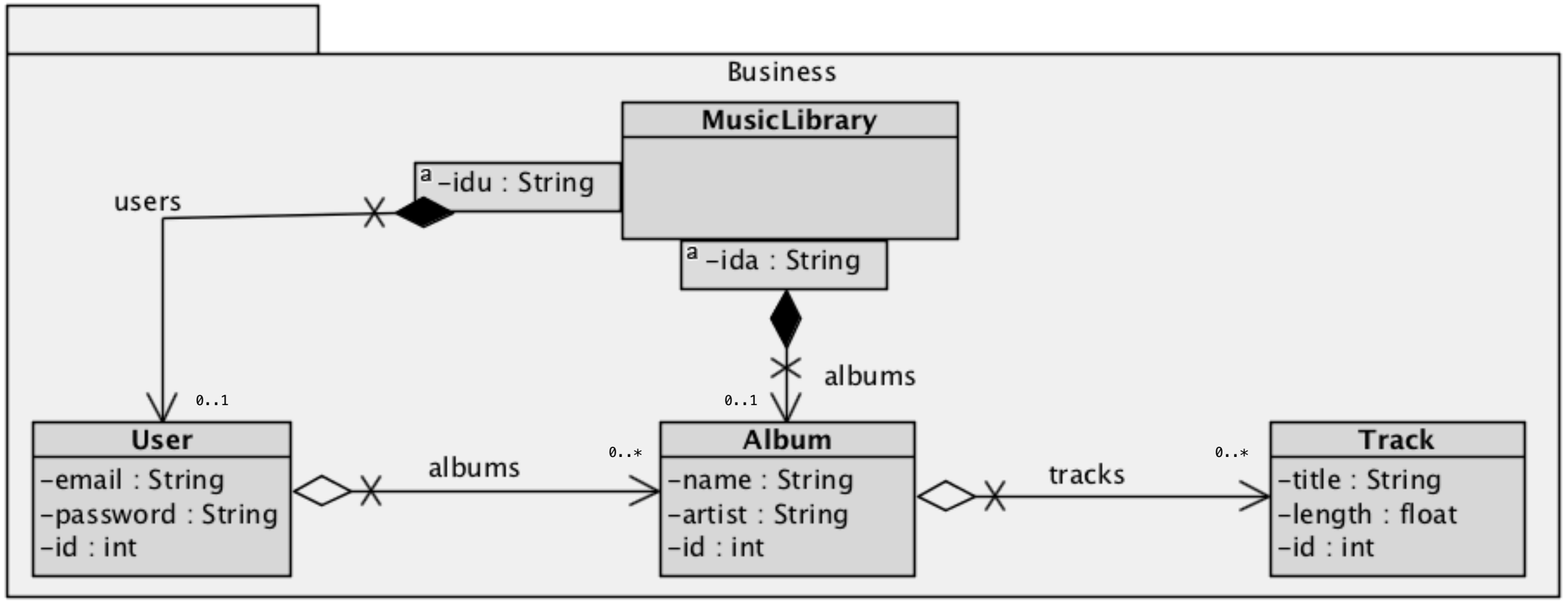
Proposal

- It is proposed the implementation of a music library management system.

- Entities:
  - A set of Users
  - A set of Albums, for each user.
  - A list of tracks, for each Album.

Structure

- The **MusicLibrary** contains both **Users** and **Albums.**
- The **User** is identified by its *email*.
- The **Album** *name* is its identifier, therefore, unique.
- Each **Track** belongs only to an **Album.**



Objectives

- Create the persistency Layer.
- Implement the features:
  - Save - persist an object
  - List - retrieve a list of objects
  - Get - retrieve a single object
  - Delete - remove an object
  - Count - get the number of existing objects
- Encapsulate the persistency as a separate layer

4. Data Layer

Standard database access

What is JDBC

- JDBC (Java Database Connectivity) is an API for database access, for Java applications.

- Is part of the Java platform.

- Vendors implement the API in order to provide libraries (i.e. Driver).

- JDBC is build on top of ODBC, an open standard for data access.

JDBC

- Vendors (e.g. MySQL, PostgeSQL, SQLite) provide connectors which implement the API, and provide access to the databases.

JDBC

- Connectors correspond to Java libraries (i.e. jar) which implement the connection.
- JDBC provides a common Java API, despite the used database.
- E.g.
  - Open a connection:
    ```
    Connection c = DriverManager.getConnection();
    ```
  - Create a query:
    ```
    c.prepareStatement(“SELECT ... ")
    ```

JDBC

- Architecturally, JDBC makes the bridge between Java, and SQL:

- ```
  PreparedStatement ps =
      cn.prepareStatement("SELECT * FROM `TABLE` WHERE ID < 10”);
  ```

- Mixing JDBC related objects with business logic objects will result in a confusing code.

- There is the need to encapsulate (abstract) these details, and establish a clear bridge between SQL and Java.
  ```

4.1 DAO

DAO

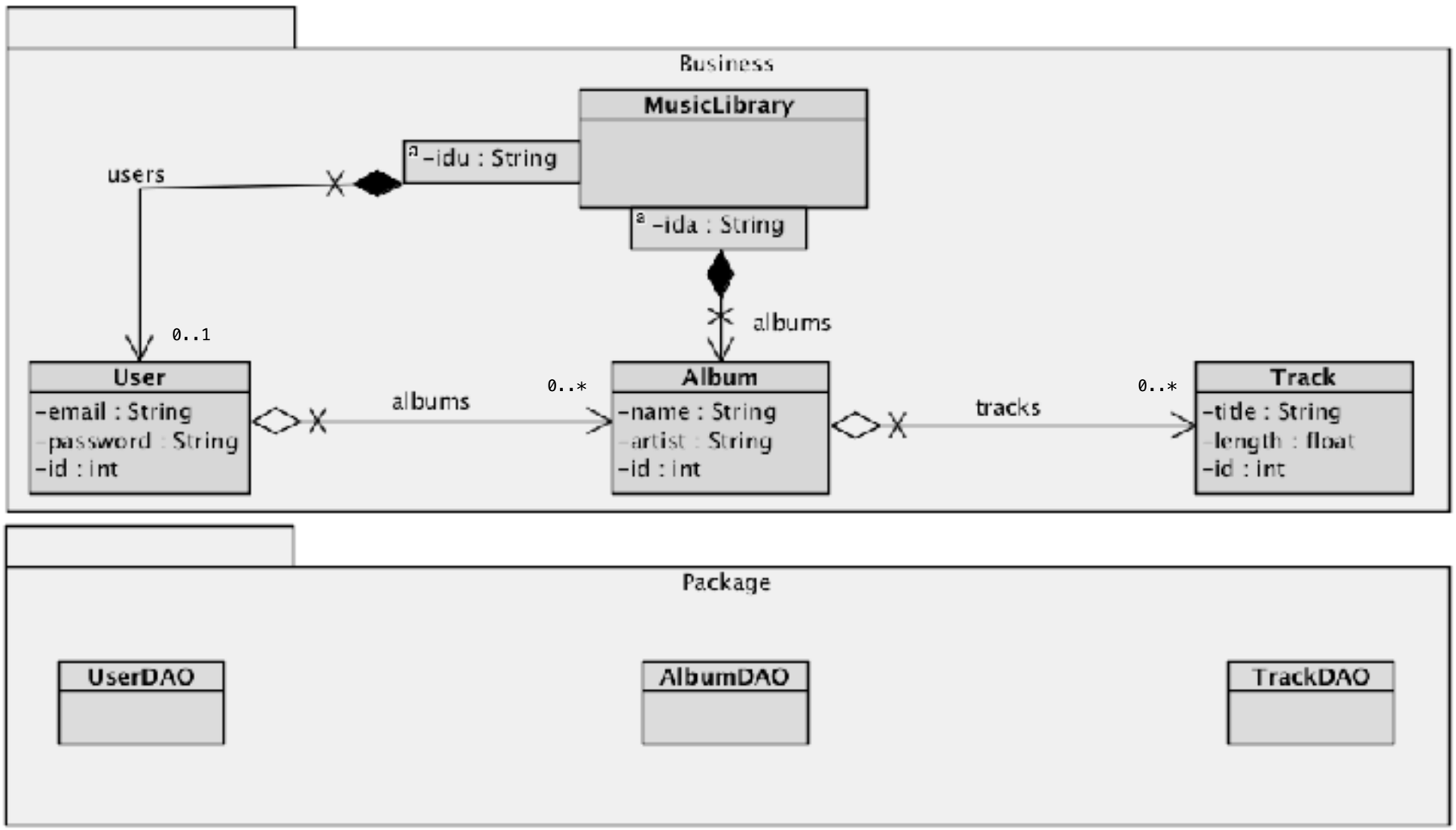
- Implementation of **D**ata **A**ccess **O**bjects (DAO) is a **pattern** to implement the persistency layer in a clean way.

- DAO are classes which:
  - Encapsulate SQL queries.
  - Persist objects into databases.
  - Create objects from the information in the databases.

- Are the facade for the data layer.

DAO

- DAOs can persist a single class, or set of classes.
  - I.e. not a DAO per class is required



DAO - Implementation

- For each DAO, we implement the required features:
  - Save - put(k : Object, o : Object) : void
  - Get - get(key : Object) : Object
  - List - list(?) : List<Object>
  - Delete - remove(key : Object) : void
  - Count - size() : int
- We have a partial implementation of the **Map** interface, an approach to uniform **DAO**s

4.2 Persistency strategies

SQL

- First, we need a table to represent a user.
- Tables to support DAOs are typically simple:

```
CREATE TABLE `User` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `email` VARCHAR(45) NOT NULL,
  `password` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`, `email`),
  UNIQUE INDEX `email_UNIQUE` (`email` ASC));
```

SQL

- In order to represent a Many-to-Many relationship (e.g. User to Album), a new table is required

```
CREATE TABLE `User_Album` (
  `iduser` int(11) NOT NULL,
  `idalbum` int(11) NOT NULL,
  PRIMARY KEY (`iduser`,`idalbum`)
);
```

SQL

- In order to represent a One-to-many relationship (e.g. Album to Track), a foreign key is required in the "contained" object/table

```
CREATE TABLE `Track` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(45) NOT NULL,
  `length` FLOAT NOT NULL,
  `idalbum` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `id_UNIQUE` (`id` ASC));
```

4.3 JDBC

API

Introduction

- JDBC: Java DataBase Connectivity - provides an API to interact with databases.
- JDBC implementations are provided by the vendors, as libraries (e.g. MySQL, Postgres, SQLITE, etc.).
  - A specific library for each database is required.
- Specific classes provide the methods to interact with the databases.

Process

- The common process is as follows:
- 1. Import the vendor library (once per project);
- 2. Initialise the driver (each time the application runs);
- 3. Establish a connection;
- 4. Execute operations;
- 5. Close the connection;

Connection

- (1) Importing the driver corresponds in adding a library to the project (depends on the IDE…).

- (2) Initialising the driver is done with "**`forName`**" method:

```
(void) Class.forName(<package>);
```

```
//Load the vendor driver for mysql:
Class.forName("com.mysql.jdbc.Driver");
//Load the vendor driver for Oracle:
Class.forName("oracle.jdbc.driver.OracleDriver");
//Load the vendor driver for PostgreSQL:
Class.forName("org.postgresql.Driver");
//Load the vendor driver for SQLite:
Class.forName("org.sqlite.JDBC");
```

Connection

- (3) The static method "`getConnection`" of the "`DriverManager`" class provides a "`Connection`".

- The connection String, depends on the vendor, and requires the database host, username and password.

```
Connection getConnection(String url);
Connection getConnection(String url, String login, String pass);
Connection getConnection(String url, java.util.Properties.info);
```

- MySLQ example:

```
//Establish the connection
Connection cn = DriverManager.getConnection("jdbc:mysql://
          <host>/<table>?user=<username>&password=<password>");
```

Connection

- This connection should be **opened before** making operation, and **close afterwards**.

- The connection has a timeout value.

- Timeouts, operations performed before opening the connection, and after closing it, throw an exception.

- Establishing the connection might fail, the `SQLException` should be captured.

- There is a **maximum number** of connections allowed.

Data operations

- Two kinds of operations can be performed:
  - (A) Modify data - insert new values into the database and/or modify existing data
  - (B) Extract data - read the existing data.
- The `Statement` class provides the methods to perform the data operations.

```
Connection conn = ...;
//create the statement
Statement st = con.createStatement();
```

Data operations

- (A) Modify data
  - Is done through the **executeUpdate** method.

```
//create the statement
Statement st;
try {
    st = con.createStatement();
    //rows is the number of affected rows
    int rows = st.executeUpdate("UPDATE/CREATE...");
} catch (SQLException e) {
    // handle exceptions
} finally {
    //close the connection
    con.close();
}
```

Data operations

- (B) Read Data
  - Is done through the **executeQuery** method.

```
Statement st;
ResultSet res;
String sql;
sql = "SELECT name FROM customers WHERE balance > 100000";
try {
    st = con.createStatement();
    res = st.executeQuery(sql);
} catch (SQLException e) {
    // handle exceptions
} finally {
    //close the connection
    con.close();
}
```

Data operations

- The resulting data is provided as a **ResultSet**.
- It is a interface, which acts an iterator over the returned records.

```
ResultSet res = ...;
//is there a next record?
boolean n = res.next();
//get the String in the corresponding column
String d = res.getString(col);
//get the String in column for the given name
String d = res.getString(name);
```

- The `ResultSet` is available until closing it or closing the corresponding `Statement`.

Data operations

- Complete example - Get the names of the customers with a bigger balance than a provided value

```
public List<String> customer(int val, Connection con) throws SQLException {
  List<String> names = new ArrayList<>();
  Statement st = con.createStatement();
  ResultSet rs;

  rs = st.executeQuery("SELECT name FROM account where balance > “ + val);

  while (rs.next()) {
     String name = rs.getString(“name”);
     names.add(name);
  }
  con.close();

  return names;
}
```

Prepared Statement

- JDBC doesn't handle security issues.
- The user is *always* malicious:

```
st.executeQuery("SELECT balance FROM account where name = '" +
uname + "'");
```

- What if the **uname** is:

```
' OR '1' = '1'; DELETE FROM account;
```

- This kind of attack is known as SQL injection.

Prepared Statement

- Prepared statements handle with these kind of issues.
- Queries are written with placeholders (**?**), and values are set by code.

```
public List<String> customer(int val, String email, Connection con) throws SQLException {
  List<String> names = new ArrayList<>();
  ResultSet rs;
  PreparedStatement st;

  st = con.prepareStatement("SELECT name FROM account where balance > ? and email = ?");

  st.setInt(1, val);
  st.setString(2, email);

  rs = st.executeQuery();

  while (rs.next()) {
     String name = rs.getString(“name”);
     names.add(name);
  }
  con.close()
  return names;
}
```

Transactions

- JDBC supports Transactions, as a way to perform sets of operations.

```
//Start the commit
con.setAutoCommit(false);
//Perform the transaction
con.commit();
//Discard the changes
con.rollback();
```

- As usual, `SQLException` should be handled.

Transactions

- Example of a transaction:

```
 try {
    //start the transaction
    con.setAutoCommit(false);
    st = con.prepareStatement(”INSERT
INTO ...");
    st.executeUpdate();
    st = con.prepareStatement(”UPDATE ...");
    st.executeUpdate();
    //perform the transaction
    con.commit();
} catch (SQLException e) {
    //discard the transaction
    con.rollback();
} finally {
    //close the connection
    con.close();
}
```

4.4 Code

Connection

- Establishing a connection to the database is a standard process

- Example for MySQL

```
//Load the vendor driver
Class.forName("com.mysql.jdbc.Driver");
//Establish the connection
Connection cn = DriverManager.getConnection(“jdbc:mysql://
          localhost/<table>?user=<username>&password=<password>”);
```

UserDAO - Save

- Save:
  - The objective is to persist an object information to a database
    - We should always use prepared statements
  - INSERT INTO USER(EMAIL, PASSWORD) VALUES (…)
  - We start by defining the query structure

```
PreparedStatement insertStmt =
cn.prepareStatement("INSERT INTO `User`(`email`,`password`) values(?,?);");
```

UserDAO - Save

- Next, we define the parameters:

```
insertStmt.setString(1, u.getEmail());
insertStmt.setString(2, u.getPassword());
```

- And finally execute the query:

```
//Use update to make changes in data
int numRows = insertStmt.executeUpdate();
//close connection
cn.close();
```

UserDAO - Get

- In order to retrieve, we use a Select query

```
PreparedStatement selectStmt = cn.prepareStatement("SELECT
* FROM `User` where email = ?");
```

- Set the parameters

```
selectStmt.setString(1, email);
```

- Execute the select

```
//User query to load data
ResultSet rs = selectStmt.executeQuery();
```

UserDAO - Get

- ResultSet holds the set of all rows matching the query (if any)
- The final step is to load the data to an object

```
User res;
if(rs.next()) {
    res = new User();
    res.setEmail(rs.getString("email"));
    res.setPassword(rs.getString("password"));
} else {
    throw new Exception("No user found for given mail");
}
//close connection
cn.close();
return res;
```

UserDAO - Get

- Final method (note that albums are missing!)

```
public User get(String email) throws Exception {
        Connection c = Connect.connect();
        if(c!=null) {
            User res;
            PreparedStatement ps = c.prepareStatement("SELECT * FROM
`User` where `email` = ?");
            ps.setString(1, email);
            ResultSet rs = ps.executeQuery();
            if(rs.next()) {
                res = new User();
                res.setEmail(rs.getString("email"));
                res.setPassword(rs.getString("password"));
                res.setId(rs.getInt("id"));
                c.close();
                return res;
            } else {
                throw new Exception("No user found for given mail");
            }
        } else {
            throw new Exception("Unable to establish connection");
        }
    }
```

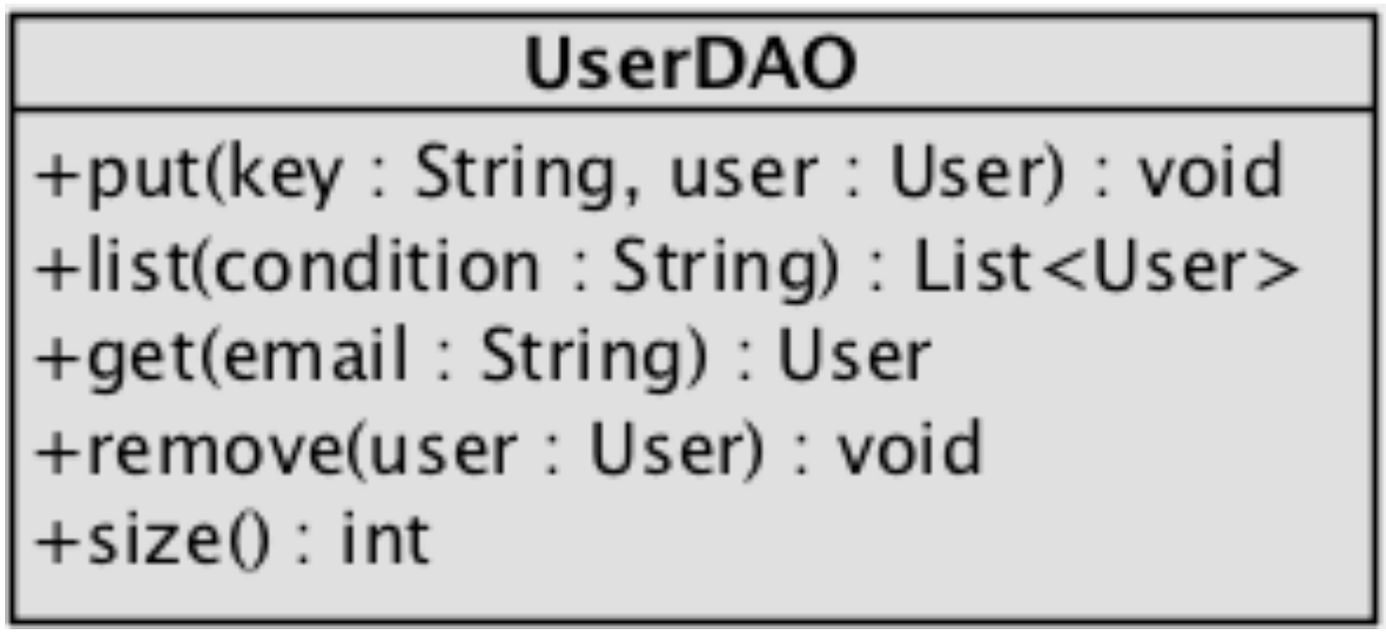
UserDAO - List

- Example of list method

```
public List<User> list(String condition) throws Exception{
        List<User> res = new ArrayList<>();
        Connection c = Connect.connect();
        if(c!=null) {
            PreparedStatement ps = c.prepareStatement("SELECT * FROM USER
WHERE " + condition);
            ResultSet rs = ps.executeQuery();
            while(rs.next()) {
                User u = new User();
                u.setId(rs.getInt("id"));
                u.setEmail(rs.getString("email"));
                u.setPassword(rs.getString("password"));
                res.add(u);
            }
        } else {
            throw new Exception("Unable to establish connection");
        }
        return res;
    }
```

UserDAO

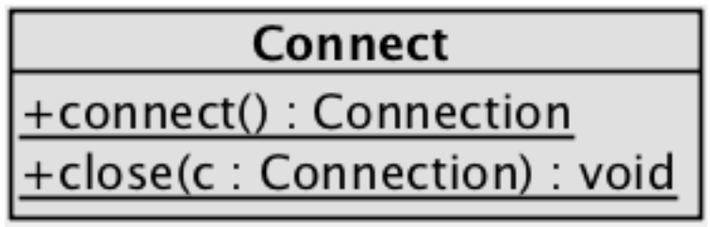
- Example for the UserDAO class.



4.4 Architectural improvements

Improving DAO

- The connection process is standard, and common to all classes.
- Can be abstracted to an utility class (static methods)



Improving DAO

```
public class Connect {
    private static final String URL = "<url>";
    private static final String SCHEMA = "<schema>";
    private static final String USERNAME = "<username>";
    private static final String PASSWORD = “<password>";

    public static Connection connect() {
        try {
            Class.forName(“com.mysql.jdbc.Driver");
            Connection cn =
DriverManager.getConnection("jdbc:mysql://"+URL+"/"+SCHEMA+"?
user="+USERNAME+"&password="+PASSWORD);
            return cn;
        } catch (Exception e) {
            //unable to connect
            e.printStackTrace();
        }
        return null;
    }

    public static void close(Connection connection) {
        try {
            connection.close();
        } catch (Exception e) {
            //nothing to close
        }
    }
}
```

Improving DAO

- Using the utility class

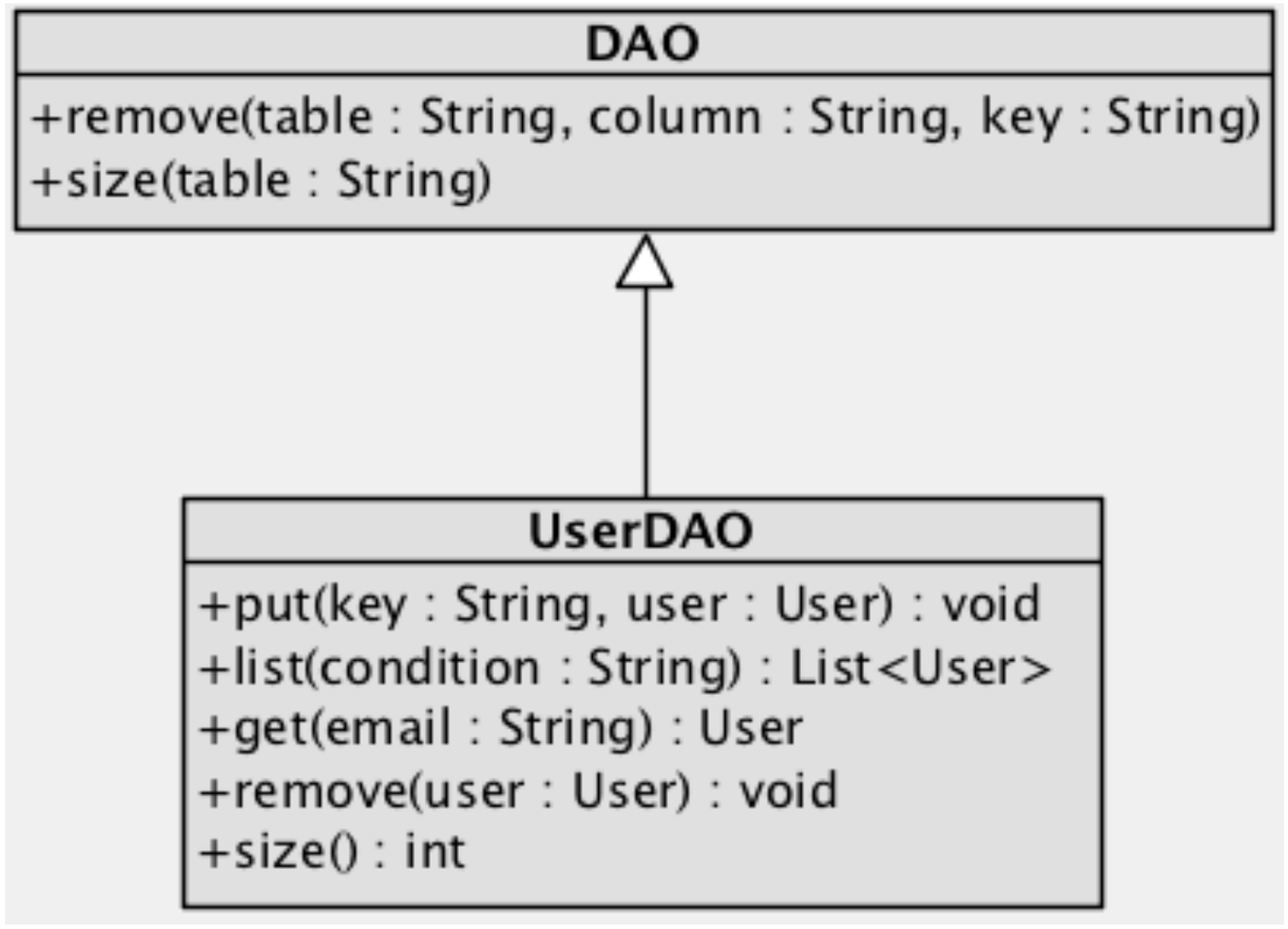
```
Connection c = Connect.connect();
if(c!=null) {
    //...
}
Connect.close(c);
```

Improving DAO

- Some DAO queries are common among all DAOs:
  - Delete: requires only the key and the table
  - Count: requires only the name of the table
- All DAOs can extend a DAO class which implements those common features.

UserDAO

- Example for the UserDAO class.



DAO - delete

- Create the generic statement

```
PreparedStatement ps = c.prepareStatement("DELETE FROM
`"+table+"` where `"+column+"` = ?");
```

- Set the parameters

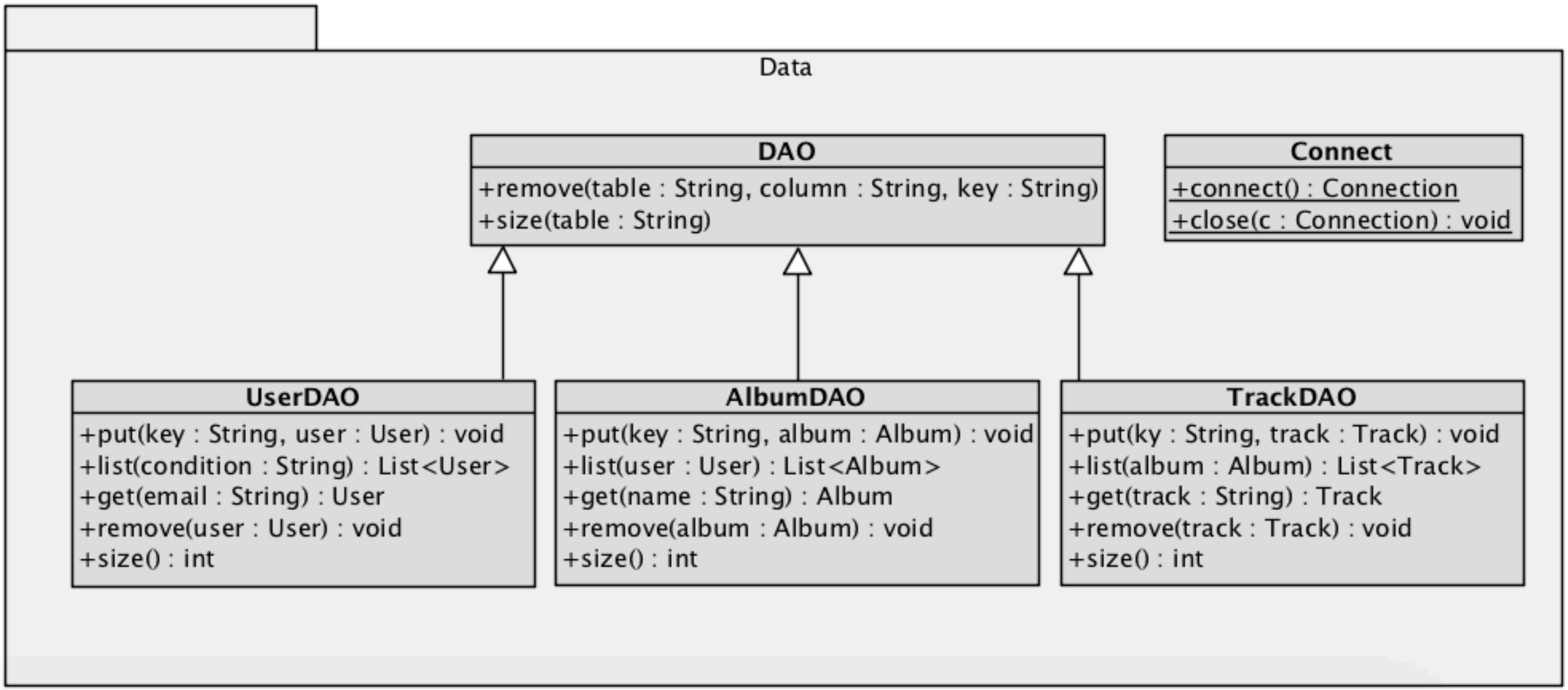
```
ps.setString(1, key);
```

- Execute the update

```
int nrows = ps.executeUpdate();
```

Data Layer

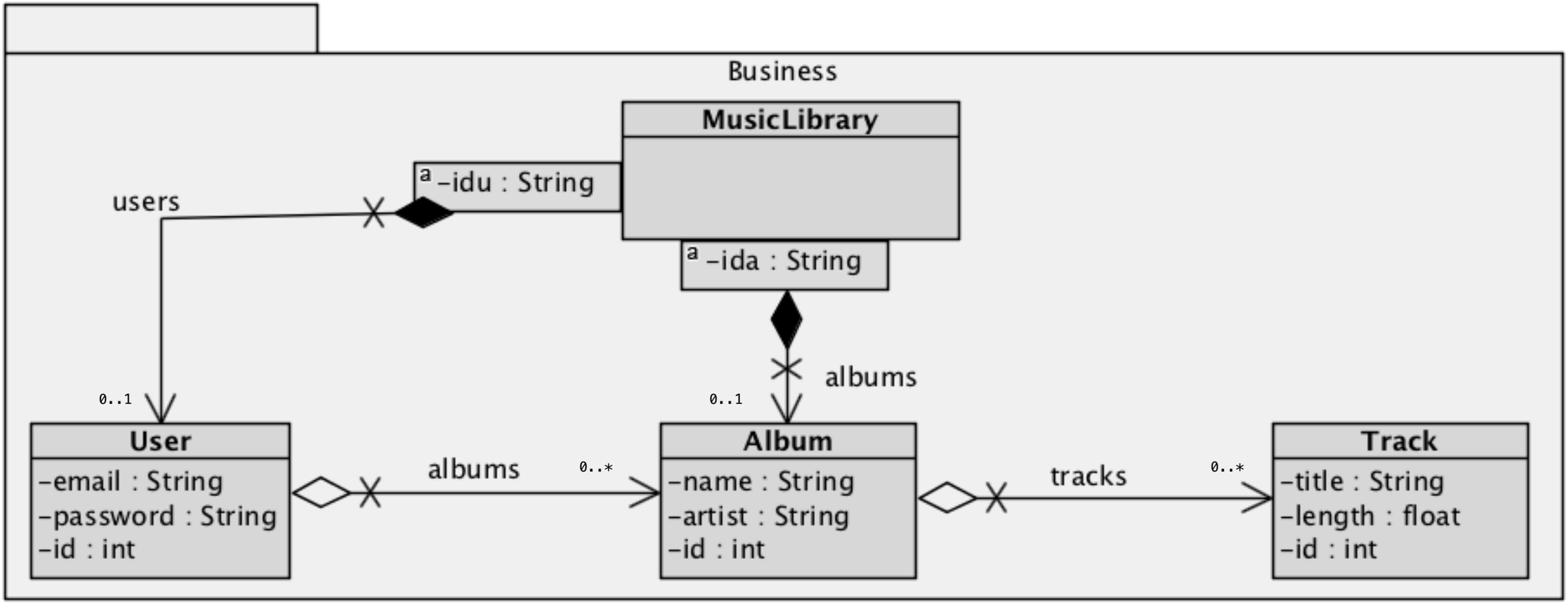
- Architecture of the data layer



4.5 Handling relationships

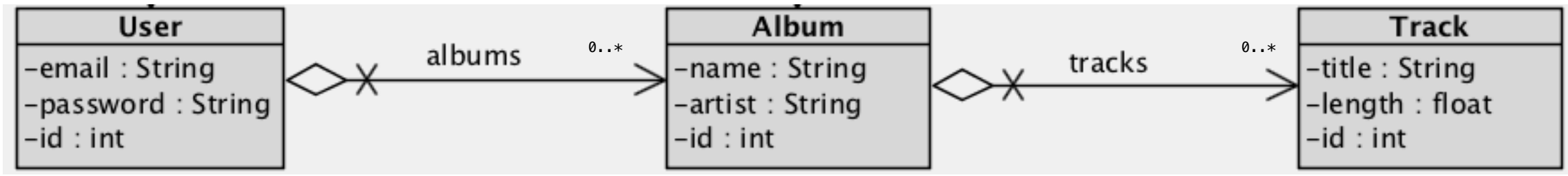
Load strategies

- When retrieving an user from the database, what to do about the albums?
- And about the Tracks?



Load strategies

- There are two possible approaches:
  - Eager: When loading the parent class, load also all the child elements.
  - Lazy: Load only the elements on demand.
- Which strategy for each relation?



Load strategies - Implementation

- Implementing Eager consists in loading all the entries, when the parent class is loaded (DAO)

```
User res;
PreparedStatement ps = c.prepareStatement("select * from User where email
= ?");
ps.setString(1, email);
ResultSet rs = ps.executeQuery();
if(rs.next()) {
    res = new User();
    res.setEmail(rs.getString("email"));
    res.setPassword(rs.getString("password"));
    res.setId(rs.getInt("id"));

    AlbumDAO ad = new AlbumDAO();
    res.setAlbums(ad.list(res));

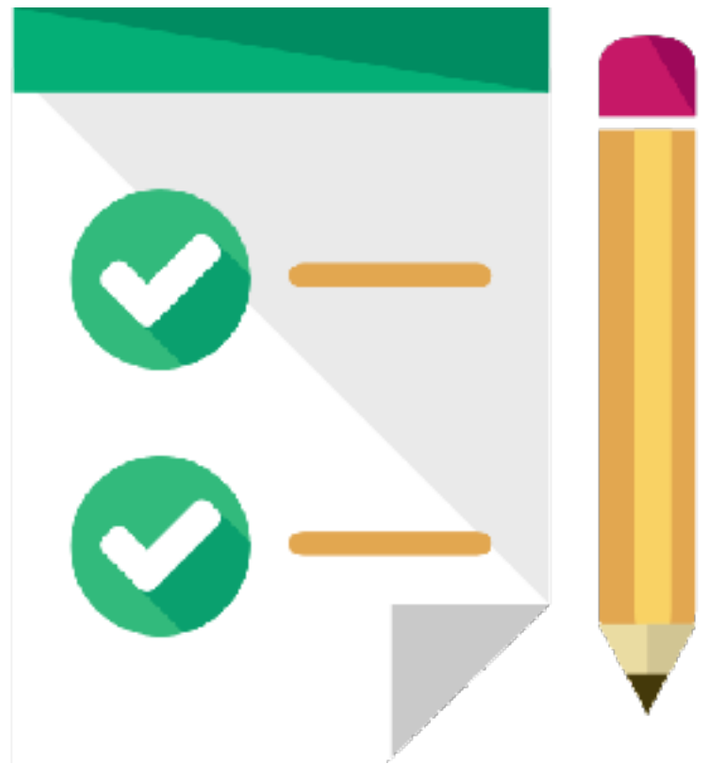
    c.close();
    return res;
} else {
     throw new Exception("No user found for given mail");
}
```

Load strategies - Implementation

- Implementing Lazy consists in loading the entries, only when required (Business class)

```
public List<Track> getTracks() {
    if(tracks == null) {
        TrackDAO td = new TrackDAO();
        tracks = td.list(this);
    }
    return tracks;
}
```

Summary



Summary

- The layers pattern provides an efficient approach to organise the java classes, and isolate the data layer
- DAOs provide the bridge from Java objects to the relational paradigm
- Persistency and data load has different approaches:
  - One table per class, classes for relationships
  - Eager and Lazy data load

JDBC

DSS 2018/2019

Rui Couto
rui.couto@di.uminho.pt

HASLab
HIGH-ASSURANCE SOFTWARE LABORATORY

INESCTEC
TECNOLOGIA E CIÊNCIA
| LABORATÓRIO ASSOCIADO