# Desenvolvimento de Sistemas de Software

**Licenciatura em Engenharia Informática**
Departamento de Informática
Universidade do Minho

2023/2024

---

## Laboratory Practices

---

António Nestor Ribeiro
anr@di.uminho.pt

José Creissac Campos
jose.campos@di.uminho.pt

# Contents

# List of Figures

# Practical Sheet #01

## 1.1 Objectives

1. Review the Object-Oriented Paradigm and the Java Programming Language.
2. Study the three-layer application pattern (presentation, business logic, and data).
3. Practice implementing the Data Layer using JDBC.

## 1.2 Multi-Layer Applications

The multi-layer software architecture is one of the most commonly used architectural patterns, allowing for the control of the growing complexity of applications. It is a type of software architecture where different software components, organised into layers, provide distinct functionalities. Since the layers are separated with clearly defined points of interaction, making changes to each of them is easier than dealing with the entire architecture simultaneously.

The simplest form of this pattern is the three-layer architecture. It contains the three most common elements of an application (see Figure 1.1):

**Presentation Layer** is the highest layer present in the application. This layer provides presentation and content manipulation services to the user, usually through a user interface, and allows isolating that interface so that the rest of the application is not dependent on a specific user interface. It interacts with the layers below in the architecture to store and process data.

**Business Logic Layer** is where the application's business logic is executed. The business logic layer implements the rules required to execute the application according to the defined requirements. It isolates the implementation of business

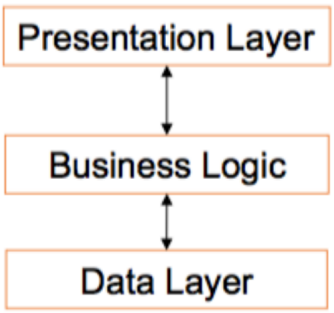



Figure 1.1: *Model-Delegate* Pattern

logic from the implementations of the other layers. To do so, it provides an API to the presentation layer, ensuring transparency of the operations it implements.

**Data Layer** is the lowest layer of the architecture and implements data persistence (storage and retrieval) for the application. It isolates access to data (typically stored in a database server) so that the rest of the application is not dependent on the data source or the structure under which the data is stored. To ensure this independence, the layer provides an API to the business layer, ensuring transparency of the data operations it implements.

In this Practical Sheet, the focus is on the Data Layer.

## 1.3 An Example - TurmasApp

The `Turmas3L` project is provided along with this exercise sheet. It is an application that allows registering students and classes and managing the allocation of students to classes. The provided project was developed in IntelliJ[1] and assumes the existence of a MariaDB[2] database server.

The application is organised into the three layers described in Section 1.2. Each layer corresponds to a package in the project (see Figure 1.2):

**Presentation Layer** in the `uminho.dss.turmas3l.ui` package. This layer implements a text-based menu of options.

**Business Logic Layer** in the `uminho.dss.turmas3l.business` package.

**Data Layer** in the `uminho.dss.turmas3l.data` package.

You can refer to the project's documentation in the `doc` folder.

[1]Tested in IntelliJ IDEA 2024.1 (Ultimate Edition).
[2]Tested with version 11.5.2. See: `https://mariadb.org/`, accessed on 2024/09/12.

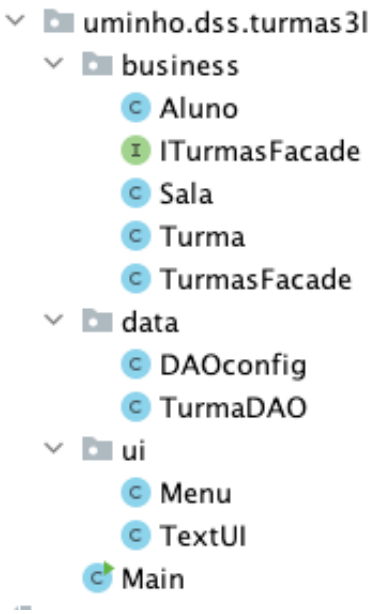



Figure 1.2: Project Packages

### 1.3.1 Business Logic Layer

The logical architecture of this layer is presented in Figure 1.3. The business logic layer provides the API defined in the `ITurmasFacade` interface to the presentation layer (see Figure 1.4).

The `TurmasFacade` class implements the `ITurmasFacade` interface, playing the role of the *Model* in an MVC (Model-View-Controller) architecture.

The `TurmasFacade` class works with two `Map` instances:

```
public class TurmasFacade implements ITurmasFacade {

    private Map<String, Turma> turmas;
    private Map<String, Aluno> alunos;
```

The methods of this class operate on these `Map`s, and they are all relatively straightforward.

This layer's remaining classes represent the Aluno, Turma, and Sala entities.

### 1.3.2 Data Layer

Typical Java data structures (Maps, Collections) exist only in memory, so they do not persist from one program run to another. When the program ends, the data in memory is lost. To add persistence to a program (for example, by saving the data in a

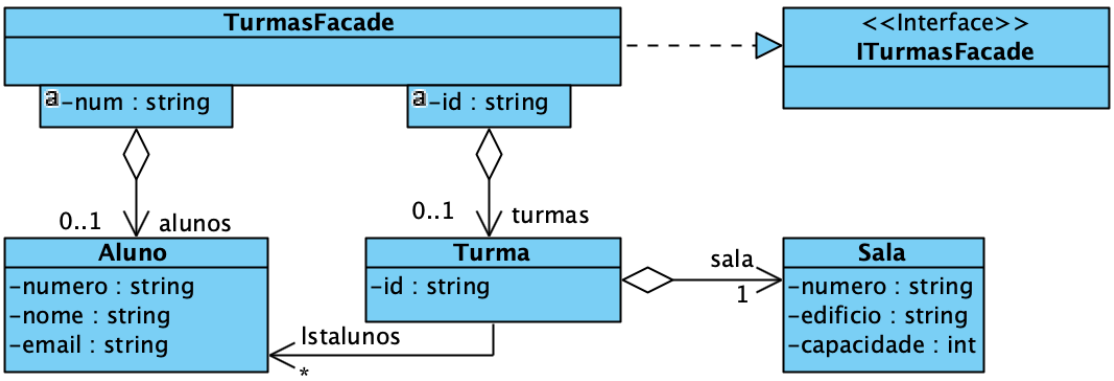



Figure 1.3: Logical Architecture to Support the Exercise

database), we need to add a data layer.

This layer typically consists of classes implementing data persistence and access (Data Access Object - DAO) classes. The goal is to replace the objects (Maps, Collections) that are storing information in memory (corresponding to one-to-many associations in the class diagram) with DAOs that store the information persistently in a database. Which associations will be implemented by DAOs and which will be kept in memory will have to be decided based on each specific situation.

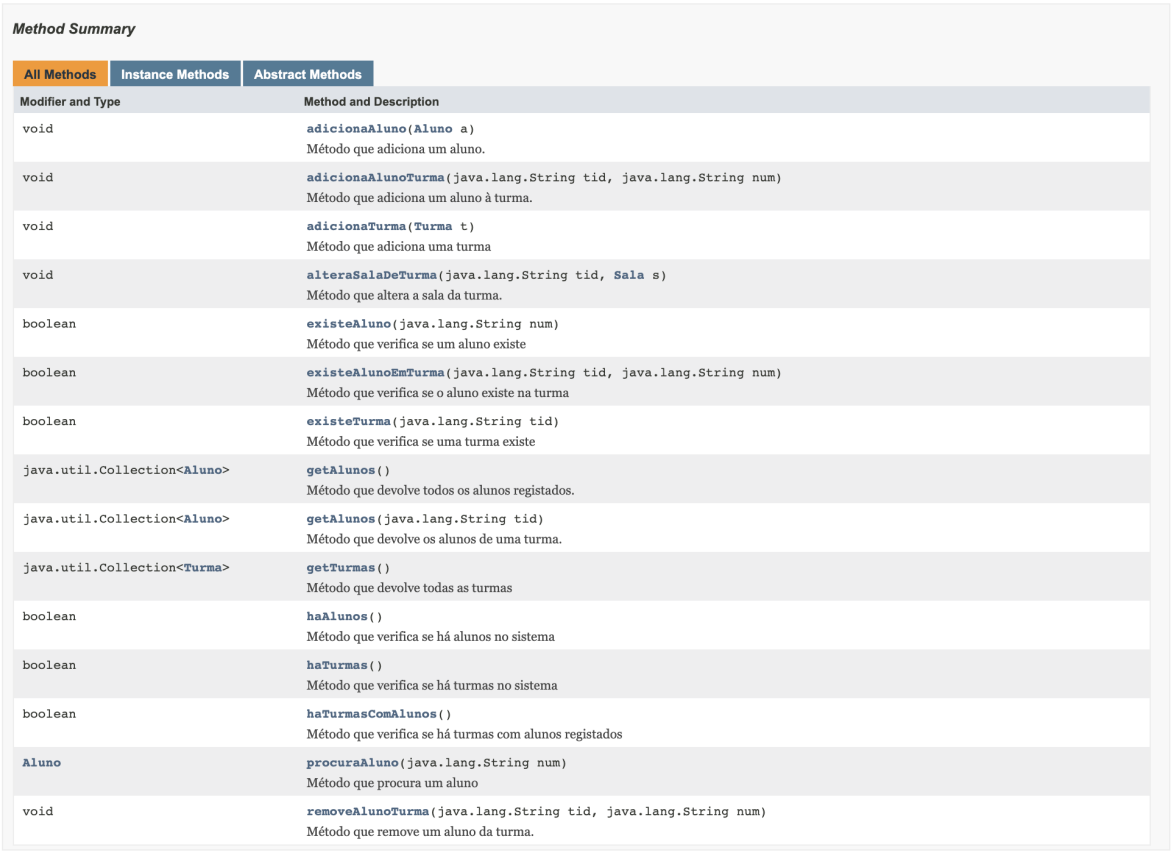
To facilitate the aforementioned replacement, the DAOs should implement the API of the objects they are replacing. Thus, if we want to persist a qualified association (in Java, a `Map`), the corresponding DAO should implement the `Map` interface.

**The project provided on Backboard corresponds to an intermediate stage of the implementation** of the architecture presented in Figure 1.4, where a Data Access Object (DAO) for the qualified association `turmas` has been partially developed (see the code in the `business::TurmasFacade` and `data::TurmaDAO` classes). However, the `alunos` association is still implemented with a `HashMap<String, Aluno>` (see the constructor of `business::TurmasFacade`).

The architecture of the current implementation state is presented in Figure 1.5. **Note that, to simplify the exercise, the list of students in `business::Turma` has been replaced by a list of student numbers**.

This layer assumes the existence of a running MariaDB database server with a `turmas3l` database created[3], as well as a user `me` with the password `mypass`[4] with remote access privileges[5]. The database server to use, the database name, and user details can be changed in the `data::DAOconfig` class.

[3] `CREATE DATABASE 'turmas3l';`
[4] `CREATE USER IF NOT EXISTS 'me'@localhost IDENTIFIED BY 'mypass';`
[5] `GRANT ALL PRIVILEGES ON *.* TO 'me'@localhost;`
`FLUSH PRIVILEGES;`



***Method Summary***

All Methods | Instance Methods | Abstract Methods

| Modifier and Type | Method and Description |
|---|---|
| void | **adicionaAluno**(**Aluno** a)<br>Método que adiciona um aluno. |
| void | **adicionaAlunoTurma**(java.lang.String tid, java.lang.String num)<br>Método que adiciona um aluno à turma. |
| void | **adicionaTurma**(**Turma** t)<br>Método que adiciona uma turma |
| void | **alteraSalaDeTurma**(java.lang.String tid, **Sala** s)<br>Método que altera a sala da turma. |
| boolean | **existeAluno**(java.lang.String num)<br>Método que verifica se um aluno existe |
| boolean | **existeAlunoEmTurma**(java.lang.String tid, java.lang.String num)<br>Método que verifica se o aluno existe na turma |
| boolean | **existeTurma**(java.lang.String tid)<br>Método que verifica se uma turma existe |
| java.util.Collection<**Aluno**> | **getAlunos**()<br>Método que devolve todos os alunos registados. |
| java.util.Collection<**Aluno**> | **getAlunos**(java.lang.String tid)<br>Método que devolve os alunos de uma turma. |
| java.util.Collection<**Turma**> | **getTurmas**()<br>Método que devolve todas as turmas |
| boolean | **haAlunos**()<br>Método que verifica se há alunos no sistema |
| boolean | **haTurmas**()<br>Método que verifica se há turmas no sistema |
| boolean | **haTurmasComAlunos**()<br>Método que verifica se há turmas com alunos registados |
| **Aluno** | **procuraAluno**(java.lang.String num)<br>Método que procura um aluno |
| void | **removeAlunoTurma**(java.lang.String tid, java.lang.String num)<br>Método que remove um aluno da turma. |

Figure 1.4: Business Logic API - Methods of the `ITurmasFacade` Interface

### JDBC

Connecting to the database server requires the use of a JDBC driver. In the case of MariaDB, the driver is provided by the MariaDB Connector/J library[6], which is already included in the project. The project is configured to use libraries located in the `lib` folder. In addition to MariaDB Connector/J, the folder contains the MySQL Connector/J library (with the JDBC driver for MySQL). If a different server is used, the corresponding `.jar` file for the library[7] should be placed in the `lib` folder.

The typical steps for using JDBC are as follows:

1. Establish a connection (`Connection`[8]) to the database. Notice the usage of the `DriverManager.getConnection` method in the constructor of `data::TurmaDAO`.

2. Create a `Statement`[9] from the `Connection` object. Notice the usage of the

[6] `https://mariadb.org/connector-java`, accessed on 2024/09/12
[7] You can search for "<database server name> jdbc" to find the relevant driver.
[8] `https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html`, accessed on 2024/09/12.
[9] `https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html`, accessed on 2024/09/12.

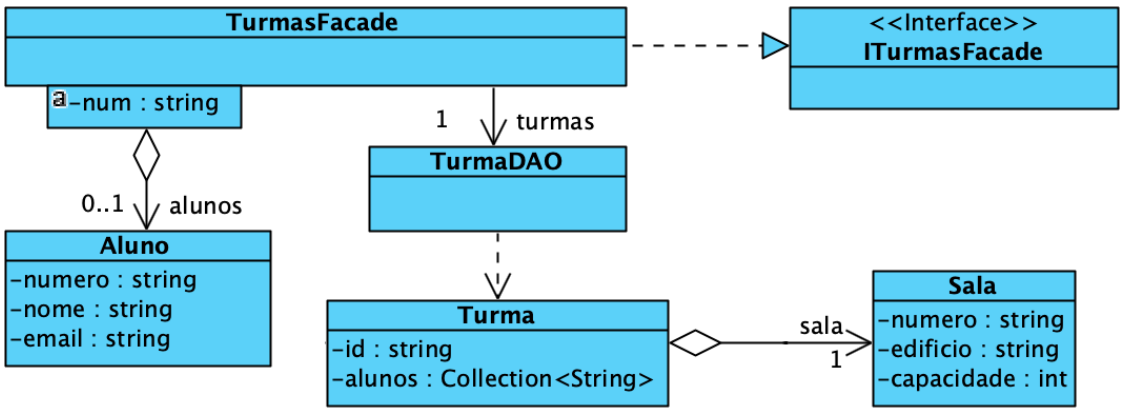



Figure 1.5: Architecture of the Provided Solution

`Connection::createStatement`[10] method in `data::TurmaDAO`.

3. Execute operations using that `Statement`. There are two essential methods to execute operations in the `Statement` interface:
   - `executeUpdate` – for SQL commands that alter the database (INSERT, DELETE, UPDATE, CREATE, DROP, ...)
   - `executeQuery` – for SQL commands that query the database (SELECT)
4. If necessary, process the results. In the case of `executeQuery`, a `ResultSet`[11] (an iterator) of results is returned.
5. Close the connection. In this case, the connections are automatically closed (see below).

The `TurmaDAO::containsKey(Object key)` method illustrates all of these steps. The `rs.next()` statement returns true if a result exists in the `ResultSet`, meaning if the key exists:

```
public boolean containsKey(Object key) {
  boolean r;
  try (Connection conn =
              DriverManager.getConnection(DAOconfig.URL,
                                          DAOconfig.USERNAME,
                                          DAOconfig.PASSWORD);
       Statement stm = conn.createStatement();
       ResultSet rs =
```

[10] Método `createStatement` da API `Connection`.
[11] `https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html`, visitado em 2024/09/12.

```
            stm.executeQuery("SELECT Id FROM turmas WHERE Id='"
                                        + key.toString() + "'")) {
    r = rs.next();
  } catch (SQLException e) {
    // Database error!
    e.printStackTrace();
    throw new NullPointerException(e.getMessage());
  }
  return r;
}
```

## 1.4 Exercises

### 1.4.1 Code Analysis

1. Study the code to understand the structure and functioning of the application (develop architectural and behavioural diagrams to aid in the analysis[12]).
   - Note how `data::TurmaDAO` implements the `Map<String,Turma>` interface (some methods are incomplete or not implemented).
   - Observe how the `data::TurmaDAO::getInstance()` method generates tables if they do not exist and analyse the relational model used.
   - Also, note the use of try-with-resources to ensure that `Connection`, `Statement`, and `ResultSet` are properly closed.
   - Notice how the `business::TurmasFacade::alteraSalaDeTurma(String, Sala)` method performs a `put` in the DAO to ensure that the data for the class is updated in the data layer.
2. Compile and run the project to verify everything functions correctly in your environment. Note that it is not possible to perform operations on students and classes.

### 1.4.2 DAO Implementation

Now, solve the exercises below, which aim to continue the development of the application.

[12] Figure 1.5 already presents the class diagram corresponding to the code. You can create other diagrams as needed throughout the semester as they are taught.

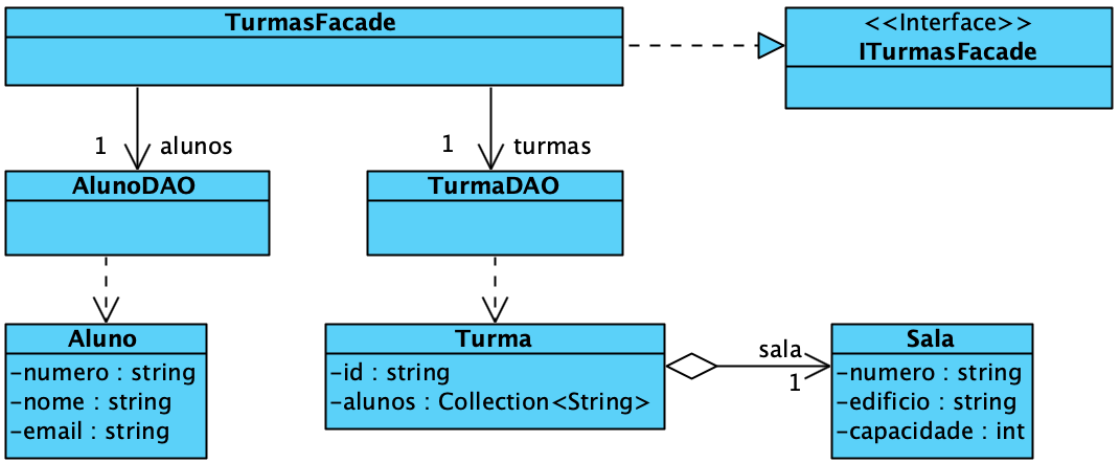



Figure 1.6: Intended Architecture with DAOs

1. Develop the `data::AlunoDAO` class and update the `business::TurmasFacade` class to start using it (see Figure 1.6).

   (a) Apart from the constructor of `alunos`, were there any other changes needed in the implementation of the business logic (`business`)?

   (b) Were there any changes required in the user interface layer (`ui`)?

2. Now, change your solution so that the `Turma::getAlunos()` method returns a list of students rather than just a list of student numbers (the solution may involve the class `Turma` having access to `AlunoDAO` to obtain student information).

3. Finally, consider that you want to add a `Map` of rooms to the *Facade*.

   (a) Implement the corresponding DAO.

   (b) Add room management to the user interface.

   (c) Modify the system to allow associating registered rooms only with classes.