

Desenvolvimento de Sistemas de Software

Licenciatura em Engenharia Informática

Departamento de Informática

Universidade do Minho

2023/2024

Práticas Laboratoriais

António Nestor Ribeiro
anr@di.uminho.pt

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

Conteúdo

1	Ficha Prática #01	1
1.1	Objectivos	1
1.2	Aplicações multi-camada	1
1.3	Um exemplo — TurmasApp	2
1.3.1	Camada de negócio	3
1.3.2	Camada de dados	4
1.4	Exercícios	7
1.4.1	Análise do código	7
1.4.2	Implementação de DAOs	8

Lista de Figuras

1.1	Padrão <i>Model-Delegate</i>	2
1.2	<i>Packages</i> do projecto	3
1.3	Arquitectura lógica de suporte ao Exercício	4
1.4	API da lógica de negócio — métodos da interface <i>ITurmasFacade</i>	5
1.5	Arquitectura da solução fornecida	6
1.6	Arquitectura pretendida, com DAOs	9

Ficha Prática #01

1.1 Objectivos

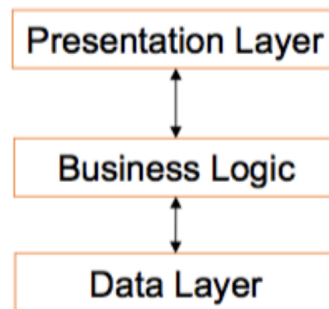
1. Relembrar o paradigma da Orientação aos Objectos e a linguagem de programação Java
2. Iniciar o estudo da estruturação de uma aplicação em três camadas (apresentação, lógica de negócio e dados)
3. Praticar a implementação da Camada de Dados em JDBC.

1.2 Aplicações multi-camada

A arquitectura de software multi-camadas é um dos padrões arquitecturais mais utilizados, permitindo controlar a crescente complexidade das aplicações. Trata-se de um tipo de arquitectura de software em que diferentes componentes de software, organizados em camadas, fornecem funcionalidades distintas. Uma vez que as camadas estão separadas, com pontos de interacção claramente definidos, fazer alterações a cada uma delas é mais fácil do que ter de lidar com toda a arquitectura em simultâneo.

A forma mais simples deste padrão é a arquitectura em três camadas. Esta contém os três elementos mais comuns de uma aplicação (ver Figura 1.1):

Camada de apresentação (*Presentation Layer* na figura) é a camada mais alta que está presente na aplicação. Esta camada fornece serviços de apresentação e manipulação de conteúdos ao utilizador, final através da interface com o utilizador, e permite isolar essa interface por forma a que o resto da aplicação não esteja dependente de uma interface com o utilizador concreta. Para apresentar o conteúdo, é essencial que interaja com os níveis abaixo na arquitectura.

Figura 1.1: Padrão *Model-Delegate*

Camada de negócio (*Business Logic* na figura) é a camada onde a lógica de negócio da aplicação é executada. A camada de negócio implementa o conjunto de regras que são necessárias para executar a aplicação de acordo com os requisitos definidos. Permite isolar a implementação da lógica de negócio das implementações das restantes camadas. Para tal oferece uma API à camada de apresentação, assegurando a transparência das operações que implementa.

Camada de dados (*Data Layer* na figura) é camada mais baixa da arquitectura e implementa a persistência (armazenamento e recuperação) dos dados da aplicação. Permite isolar o acesso aos dados (tipicamente armazenados num servidor de base de dados), por forma a que o resto da aplicação não esteja dependente da origem dos dados ou da estrutura sob a qual estão armazenados. Para garantir esta independência, a camada oferece uma API à camada de negócio, assegurando a transparência das operações de dados que implementa.

Nesta Ficha Prática, o foco está na Camada de Dados.

1.3 Um exemplo — TurmasApp

Juntamente com esta ficha, é disponibilizado o projecto Turmas3L. Trata-se de uma aplicação que permite registar alunos e turmas, e gerir a alocação de alunos às turmas. O projecto disponibilizado foi desenvolvido em IntelliJ¹ e pressupõe a existência de um servidor de base de dados MariaDB².

A aplicação está organizada nas três camadas descritas na Secção 1.2. A cada

¹Testado em IntelliJ IDEA 2024.1 (Ultimate Edition).

²Testado com a versão 11.5.2. Ver: <https://mariadb.org/>, visitado em 2024/09/12.

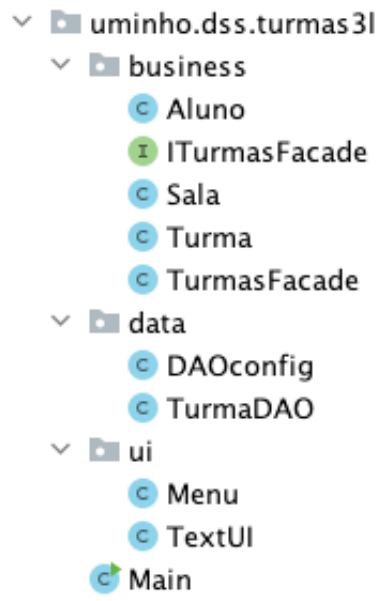


Figura 1.2: Packages do projecto

camada corresponde um *package* no projecto (ver Figura 1.2):

Camada de apresentação no *package* `uminho.dss.turmas3l.ui`. Esta camada implementa um menu de opções em modo texto.

Camada de negócio no *package* `uminho.dss.turmas3l.business`. Esta camada está descrita na Secção 1.3.1.

Camada de dados no *package* `uminho.dss.turmas3l.data`. Esta camada está descrita na Secção 1.3.2.

Pode consultar a documentação do projecto na pasta `doc`.

1.3.1 Camada de negócio

A arquitectura lógica desta camada é apresentada na Figura 1.3. A camada de negócio fornece à camada de apresentação a API definida na interface `ITurmasFacade` (ver Figure 1.4).

A classe `TurmasFacade` implementa a interface `ITurmasFacade` assumindo o papel de *Model* numa arquitectura de tipo MVC (Model-View Controller).

A classe `TurmasFacade` trabalha com dois [Map](#):

```
public class TurmasFacade implements ITurmasFacade {
```

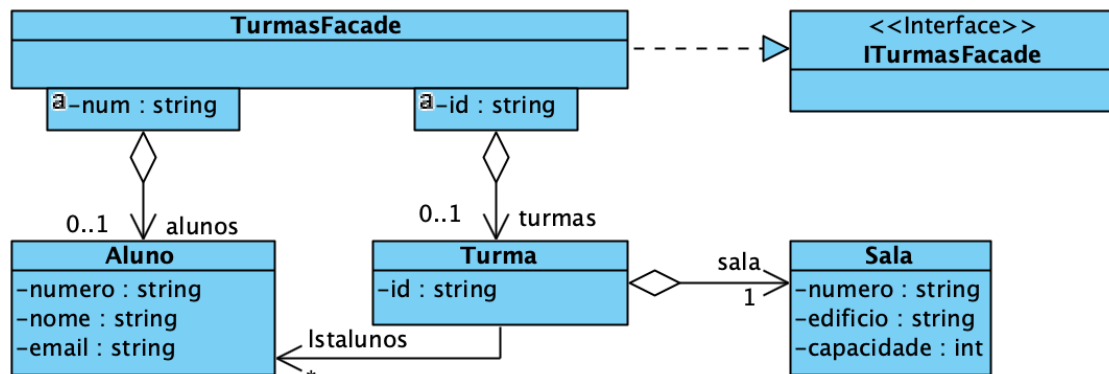


Figura 1.3: Arquitectura lógica de suporte ao Exercício

```

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

Os métodos desta classe trabalham sobre estes `Map`, sendo todos bastante simples.

As restantes classes desta camada representam as entidades Aluno, Turma e Sala.

1.3.2 Camada de dados

As estruturas de dados típicas do Java (`Maps`, `Collections`) existem apenas em memória, pelo que não persistem de uma utilização do programa para outra. Quando o programa termina, os dados em memória perdem-se. Para adicionar persistência a um programa (por exemplo, guardando os dados numa Base de Dados), devemos adicionar-lhe uma camada de dados.

Esta camada consiste, tipicamente, num conjunto de classes que implementam a persistência e acesso aos dados (classes DAO - do inglês *Data Access Objects*). O objetivo é substituir os objetos (`Maps`, `Collections`) que estão a guardar a informação em memória (correspondentes a associações um-papa-muitos no diagrama de classes), por DAO que guardam a informação de forma persistente numa base de dados. Que associações serão implementadas por DAO e que associações serão mantidos em memória, terá que ser decidido em função de cada situação concreta.

Para facilitar a substituição referida acima, os DAO deverão implementar a API dos objetos que estão a substituir. Assim, se pretendemos persistir uma associação qualificada (em Java, um `Map`), o DAO correspondente deverá implementar a interface `Map`.

O projecto disponibilizado com a ficha corresponde a uma fase intermédia da implementação da arquitectura apresentada na Figura 1.4, em que já foi parci-

Method Summary

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

Figura 1.4: API da lógica de negócio — métodos da interface ITurmasFacade

almente desenvolvido um DAO para representar a associação qualificada turmas (ver o código das classes `business::TurmasFacade` e `data::TurmaDAO`). A associação alunos, no entanto, ainda está implementada com um `HashMap<String, Aluno>` (ver construtor de `business::TurmasFacade`).

A arquitectura do estado actual da implementação é apresentada na Figura 1.5. Como se pode ver, a associação qualificada turmas foi substituída por uma instância do DAO `TurmaDAO`. **Note que, para simplificar o exercício, a lista de alunos em `business::Turma` foi substituída por uma lista de números de aluno.**

Esta camada pressupõe a existência de um servidor de base de dados MariaDB a correr com uma base de dados `turmas31`³ criada, bem como um utilizador `me`, com password `mypass`, criado⁴ e com privilégios de acesso remoto⁵. Quer o servidor a utilizar, quer o nome da base de dados e os detalhes do utilizador, podem ser

³CREATE DATABASE 'turmas31';

⁴CREATE USER IF NOT EXISTS 'me'@localhost IDENTIFIED BY 'mypass';

⁵GRANT ALL PRIVILEGES ON *.* TO 'me'@localhost;
FLUSH PRIVILEGES;

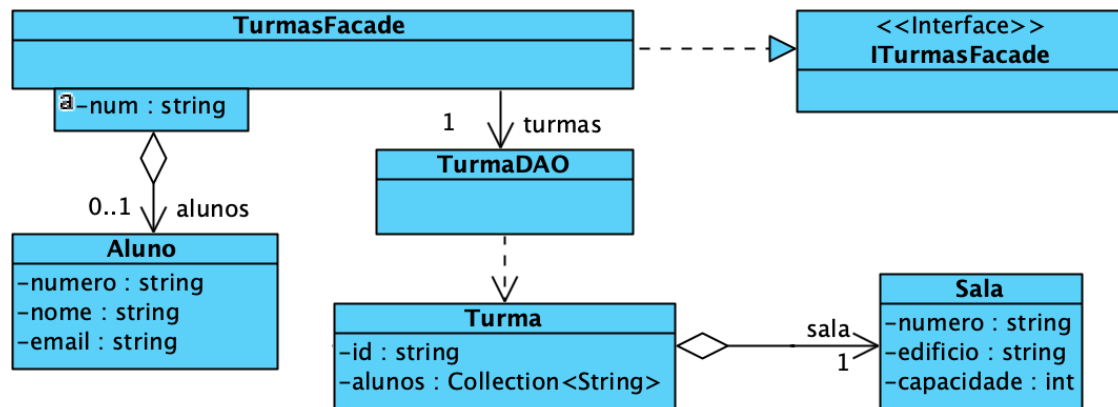


Figura 1.5: Arquitectura da solução fornecida

alterados na classe `data::DAOconfig`.

JDBC

A ligação ao servidor de base de dados requer a utilização de um *driver* JDBC. No caso do MariaDB o *driver* é fornecido pela biblioteca MariaDB Connector/J⁶, já incluída no projecto. O projecto está configurado para utilizar bibliotecas presentes na pasta `lib`. Para além da MariaDB Connector/J, a pasta contém ainda a biblioteca MySQL Connector/J (com o *driver* JDBC para MySQL). Caso o servidor utilizado seja outro, o ficheiro `.jar` da biblioteca correspondente⁷ deverá ser colocada na referida pasta `lib`.

Os passos usuais para utilização de JDBC consistem em:

1. Estabelecer uma ligação (`Connection`⁸) à Base de Dados — ver utilização de `DriverManager.getConnection` no construtor de `data::TurmaDAO`.
2. Criar um `Statement`⁹ a partir da `Connection` — ver utilização do método `Connection::createStatement`¹⁰ em `data::TurmaDAO`.
3. Executar operações com base nesse `Statement` — existem dois métodos essenciais para executar operações na interface `Statement`:
 - `executeUpdate` — para comandos SQL que alteram a Base de Dados (INSERT, DELETE, UPDATE, CREATE, DROP, ...)

⁶<https://mariadb.org/connector-java>, visitado em 2024/09/12.

⁷Basta pesquisar por "<nome servidor bd> jdbc" para encontrar o *driver* relevante.

⁸<https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html>, visitado em 2024/09/12.

⁹<https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html>, visitado em 2024/09/12.

¹⁰Método `createStatement` da API `Connection`.

- `executeQuery` – para comandos SQL que consultam a Base de Dados (SELECT)
4. Se necessário, processar os resultados — No caso de `executeQuery` é devolvido um `ResultSet`¹¹ (um iterador) sobre os resultados.
 5. Fechar a ligação — neste caso as ligações estão a ser fechadas automaticamente (ver mais abaixo).

O método `TurmaDAO::containsKey(Object key)` ilustra todos estes passos. A instrução `rs.next()` devolve verdade se existir um resultado no `ResultSet`, ou seja, se existir a chave:

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

1.4.1 Análise do código

1. Estude o código de modo a perceber a estruturação e funcionamento da aplicação (desenvolva diagramas arquitecturais e comportamentais para auxiliar

¹¹<https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html>, visitado em 2024/09/12.

a análise¹²).

- Note como `data::TurmaDAO` implementa a interface `Map<String, Turma>` (alguns dos métodos estão incompletos ou por implementar).
 - Note como o método `data::TurmaDAO::getInstance()` gera as tabelas, caso não existam, e analise o modelo relacional usado.
 - Note ainda a utilização de *try with resources* para garantir que as `Connection`, os `Statement` e os `ResultSet`, são correctamente fechados.
 - Note como o método `business::TurmasFacade::alteraSalaDeTurma(String, Sala)` efectua um `put` no DAO, do modo a garantir que os dados da turma são actualizados na camada de dados.
2. Compile e execute o projecto para verificar que tudo está a funcionar correctamente no seu ambiente. Note que não é possível executar operações sobre alunos e turmas.

1.4.2 Implementação de DAOs

Resolva agora os exercícios abaixo, que visam continuar o desenvolvimento da aplicação.

1. Desenvolva a classe `data::AlunoDAO` e actualize a classe `business::TurmasFacade`, de modo a que passe a utilizá-la (ver Figura 1.6).
 - (a) Para além do construtor de `alunos`, foi necessária mais alguma alteração na implementação da lógica de negócio (*package business*)?
 - (b) Foi necessário efectuar alguma alteração na camada de interface com o utilizador (*package ui*)?
2. Altere agora a sua solução, sabendo que a classe `Turma::getAlunos()` deverá devolver uma lista de alunos e não apenas uma lista com os números dos alunos (a solução poderá passar por a turma ter acesso a `AlunoDAO` para obter a informação dos alunos).
3. Por fim, considere que se pretende acrescentar à *Facade* um `Map` de salas.
 - (a) Implemente o DAO correspondente
 - (b) Acrescente à interface com o utilizador a gestão de salas.
 - (c) Altere o sistema para apenas permitir associar salas registadas às turmas.

¹²A Figura 1.5 já apresenta a diagrama de classes correspondente ao código. Poderá construir outros diagrama ao longo do semestre, à medida que eles forem leccionados.

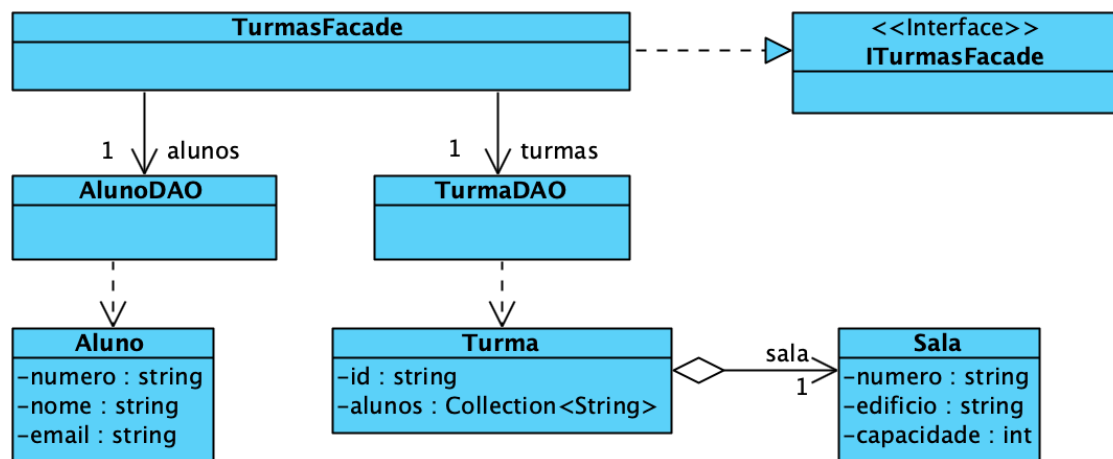


Figura 1.6: Arquitectura pretendida, com DAOs