Desenvolvimento de Sistemas de Software

Licenciatura em Engenharia Informática
Departamento de Informática
Universidade do Minho

2021/2022

Práticas Laboratoriais

António Nestor Ribeiro
anr@di.uminho.pt

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

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 programação segundo o padrão *Model-Delegate*.

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, ou seja, apresentação de conteúdos ao utilizador final através da interface, e permite isolar a interface com o utilizador por forma a que o resto da aplicação não esteja dependente de uma interface concreta. Para apresentar o conteúdo, é essencial que interaja com os níveis abaixo na arquitectura.

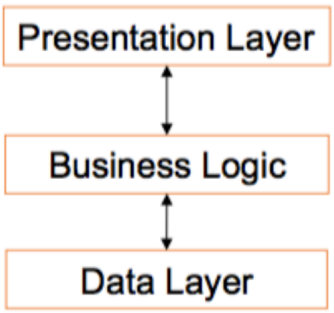


Figura 1.1: Padrão *Model-Delegate*

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.

1.3 Padrão *Model-Delegate*

A implementação da camada de apresentação recorre aos seus próprios padrões. O padrão *Model-Delegate* é uma variação do padrão *Model-View-Controller* (MVC). Duas das três responsabilidades da MVC, nomeadamente a *View* e o *Controller*, são reunidas no *User Interface Delegate* (ver Figura 1.2). Neste padrão, podem então ser distinguidas dois tipos de classes:

- Classes do *Model*: responsáveis por manter a ligação à lógica de negócio e dados da aplicação
- Classes do *Delegate*: responsáveis por apresentar informação e interagir com

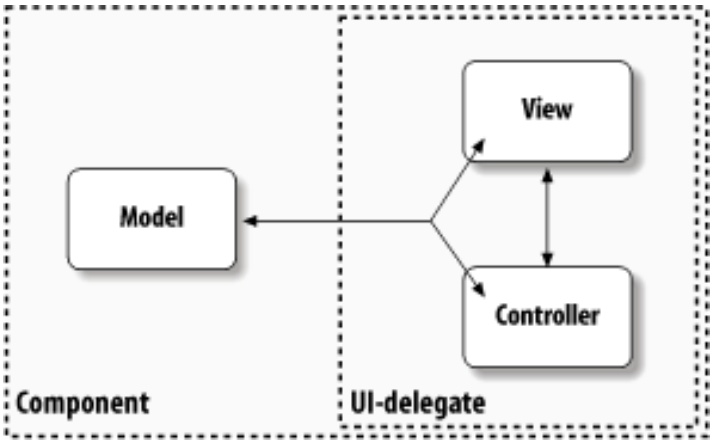


Figura 1.2: Padrão *Model-Delegate*

o utilizador.

O padrão foi criado durante o desenvolvimento do Java Swing, para resolver os problemas criados pela forte interdependência entre *View* e *Controller*[1].

1.4 Um exemplo — TurmasApp

Juntamente com esta ficha, é disponibilizado o projecto `Turmas3L`[2]. Trata-se de uma aplicação que permite registar alunos e turmas, e gerir a alocação de alunos às turmas. A aplicação está organizada nas três camadas descritas na Secção 1.2.

A cada camada corresponde um *package* no projecto (ver Figura 1.3):

Camada de apresentação no *package* `uminho.dss.turmas3l.ui`. Esta camada está descrita na Secção 1.4.1.

Camada de negócio no *package* `uminho.dss.turmas3l.business`. Esta camada está descrita na Secção 1.4.2.

Camada de dados no *package* `uminho.dss.turmas3l.data`. Esta camada está descrita na Secção 1.4.3.

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

[1] Ver `https://www.oracle.com/java/technologies/a-swing-architecture.html` (visitado em 01/10/2021).

[2] Criado em IntelliJ IDEA Ultimate 2021.1.

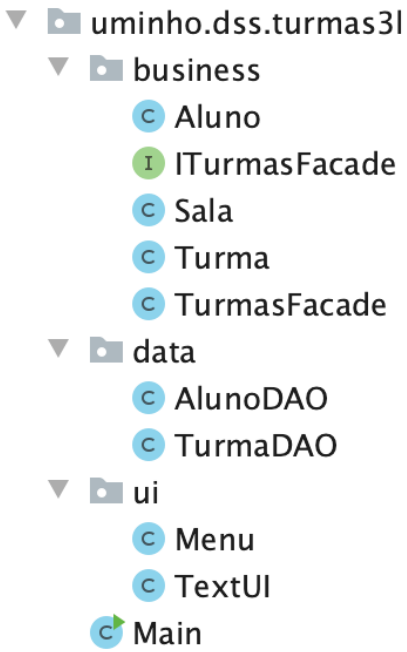


Figura 1.3: *Packages* do projecto

1.4.1 Camada de apresentação

Esta camada segue o padrão *Model-Delegate*. O *Delegate* é implementado pela classe `TextUI`, sendo o *Model* qualquer classe que implemente a interface `ITurmasFacade` (ver linha 24 da classe `TextUI`). Neste caso concreto, utiliza-se uma instância da classe `TurmasFacade` (ver construtor de `TextUI`).

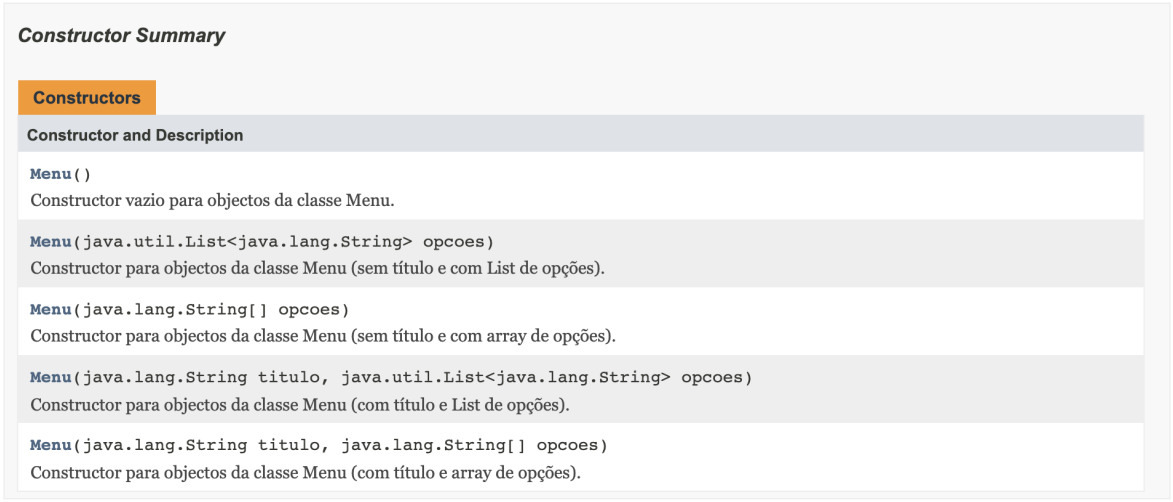
A classe `TextUI` recorre à classe `Menu` para implementar uma interface com menus em modo texto.

Utilização da classe `Menu`

A classe `Menu` assegura a execução da interface através de menus em modo texto. Um exemplo de utilização pode ser visto na classe `TextUI`. Note que esta classe não está completa, faltando completar a definição dos menus.

Para criar um menu é necessário definir a lista de opções a apresentar e, para cada opção, qual a sua pré-condição e qual o *event handler* a utilizar. A pré-condição (do tipo `PreCondition`) deverá retornar um valor booleano e é calculada sempre que o menu é apresentado. A opção só fica disponível se o valor da pré-condição for verdade.

O *event handler* é o método que será chamado caso a opção seja seleccionada pelo utilizador. Este método tipicamente irá invocar um método do *Model* ou



Constructor Summary

Constructors

Constructor and Description
`Menu()` Constructor vazio para objectos da classe Menu.
`Menu(java.util.List<java.lang.String> opcoes)` Constructor para objectos da classe Menu (sem título e com List de opções).
`Menu(java.lang.String[] opcoes)` Constructor para objectos da classe Menu (sem título e com array de opções).
`Menu(java.lang.String titulo, java.util.List<java.lang.String> opcoes)` Constructor para objectos da classe Menu (com título e List de opções).
`Menu(java.lang.String titulo, java.lang.String[] opcoes)` Constructor para objectos da classe Menu (com título e array de opções).

Figura 1.4: Construtores da classe `Menu`

executer um outro menu. O primeiro caso utiliza-se para apresentar informação ao utilizador e/ou para ler dados e executar operações. O método `adicionarAlunoATurma()` é um exemplo de um *event handler* deste tipo. O segundo caso utiliza-se para implementar interfaces com menus e sub-menus. O método `adicionarAlunoATurma()` (que falta implementar) será um *event handler* deste tipo.

Assim, a utilização de um menu consiste nos seguintes passos:

1. Criar o menu.

   Como se pode ver pela lista de construtores da classe (ver Figura 1.4) é possível criar menus com ou sem título e com ou sem a lista de opções a apresentar. Um exemplo de criação de um menu pode ser visto no método `menuPrincipal`, da classe `TextUI`:

```
private void menuPrincipal() {
      Menu menu = new Menu(new String[]{
                  "Operações sobre Alunos",
                  "Operações sobre Turmas",
                  "Adicionar Aluno a Turma",
                  "Remover Aluno de Turma",
                  "Listar Alunos de Turma"
      });
```

   Neste caso, está a utilizar-se o construtor que recebe apenas um array de opções, o título não está a ser definido. Assim sendo, é utilizado o título genérico "*** Menu ***", tal como pode ser observado na Figura 1.5.

2. Definir pré-condições.

   O método `setPreCondition(int i, PreCondition b)` permite (re)definir a pré-condição da `i`-ésima opção do menu[3]. A linha seguinte do método `menuPrincipal()` garante que a terceira opção do menu (adicionar aluno a turma) só está disponível se existirem alunos e turmas na aplicação:

   ```
   // Registar pré-condições das transições
   menu.setPreCondition(3,
       ()->this.model.haAlunos() && this.model.haTurmas());
   ```

   Note a utilização de uma expressão lambda para definir a pré-condição. Note, ainda, a utilização do *Model* para obter informações sobre os dados.

3. Definir *event handlers*.

   De modo análogo às pré-condições, o método `setHandler(int i, Handler h)` permite definir os *event handlers* das opções do menu. Também no método `menuPrincipal()` da classe `TextUI`, podemos ver a definição de todos os *event handlers* do menu principal:

   ```
   // Registar os handlers das transições
   menu.setHandler(1, ()->gestaoDeAlunos());
   menu.setHandler(2, ()->gestaoDeTurmas());
   menu.setHandler(3, ()->adicionarAlunoATurma());
   menu.setHandler(4, ()->removerAlunoDeTurma());
   menu.setHandler(5, ()->listarAlunosDaTurma());
   ```

   Neste caso, escolher a opção 1 causa a execução do método `gestaoDeAlunos()`, escolher a opção 2, o método `gestaoDeTurmas()`, etc. Alguns destes métodos estão já definidos, outros estão por definir.

   Na prática, ao definir os *event handlers*, estamos a definir de que modo o utilizador pode navegar na aplicação.

4. Executar o menu.

   A última linha do método `menuPrincipal()` executa o menu, utilizando, para isso, o método `run()`:

   ```
   // Executar o menu
   menu.run();
   }
   ```

[3] A primeira opção do menu está na posição 1.

```
Bem vindo ao Sistema de Gestão de Turmas!

 *** Menu ***
1 - Operações sobre Alunos
2 - Operações sobre Turmas
3 - ---
4 - Remover Aluno de Turma
5 - Listar Alunos de Turma
0 - Sair
Opção:
```

Figura 1.5: Construtores da classe `Menu`

O método apresenta, ou não, as opções, dependendo dos valores das respectivas pré-condições, e executa os *event handlers*, em resposta às escolhas do utilizador.

A Figura 1.5 mostra o menu que é apresentado quando se inicia a aplicação. Note que a opção 3 não é disponibilizada. Tal deve-se à sua pré-condição ter falhado (nesta versão da aplicação não existe ainda persistência de dados, pelo que inicialmente não há nem alunos nem turmas). Note, ainda, que foi acrescentada a opção 0 (zero), correspondente a sair do menu. O menu será executado até que o utilizador opte por sair.

Para os casos em que se pretenda executar um menu apenas uma vez, a classe `Menu` disponibiliza o método `runOnce()`.

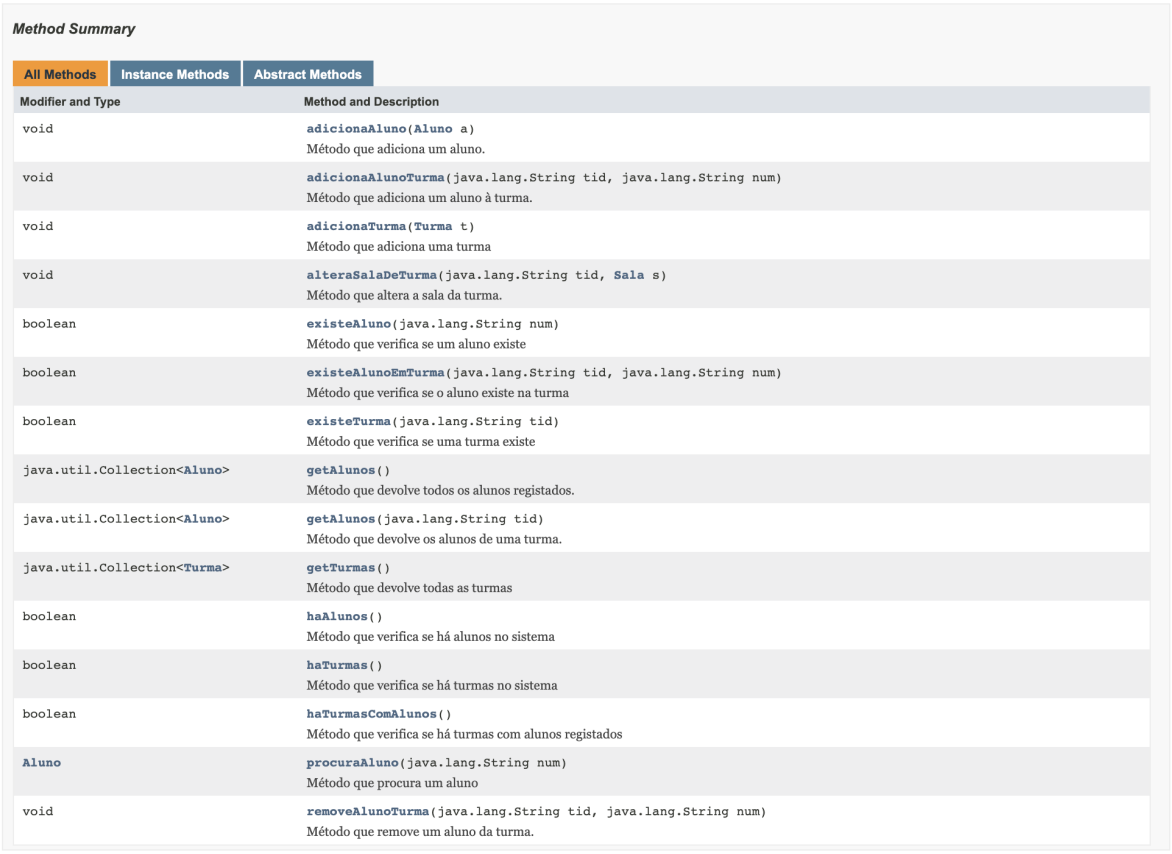
Vale a pena referir que a definição de pré-condições e *event handlers* pode ser realizada por qualquer ordem.

Finalmente, o método `void option(String name, PreCondition p, Handler h)`, não ilustrado aqui, permite adicionar uma opção ao menu, com a respectiva pré-condição e *event handler*. Este método é útil no caso em que o menu é criado sem qualquer opção.

1.4.2 Camada de negócio

A camada de negócio fornece à camada de apresentação a API definida na interface `ITurmasFacade` (ver Figure 1.6).

Nesta versão do projecto a camada de negócio está completamente implementada, embora existam diversos aspectos a melhorar (ver exercícios na Secção 1.5).



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 1.6: API da lógica de negócio — métodos da interface `ITurmasFacade`

A classe `TurmasFacade` implementa a interface `ITurmasFacade`, assumindo, como já visto, o papel de *Model* no padrão *Model-Delegate*, da camada de apresentação.

A classe `TurmasFacade` trabalha com dois `Map` que obtém da camada de dados:

```
public class TurmasFacade implements ITurmasFacade {

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

    public TurmasFacade() {
        this.turmas = TurmaDAO.getInstance();
        this.alunos = AlunoDAO.getInstance();
    }
```

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

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

1.4.3 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*). Neste caso, e como se pode ver no *package* `uminho.dss.turmas3l.data`, a camada contém duas destas classes: `AlunoDAO` e `TurmaDAO`; referentes a alunos e turmas, respectivamente.

Na versão do projecto disponibilizada, as classes não implementam ainda a persistência, limitando-se a construir os `Map` necessários a guardar os dados em memória (ver métodos `getInstance()`).

1.5 Exercícios

1. Estude o código de modo a familiarizar-se com as diferentes camadas da aplicação. Desenhe um diagrama de classes para facilitar a compreensão da arquitectura utilizada.

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.

3. A camada de interface está incompleta, faltando implementar diversas componentes da interface. Para a completar, implemente os métodos `gestaoDeAlunos()` e `gestaoDeTurmas()`, sabendo que deverão definir menus que implementem o comportamento descrito na Figura 1.7.

   Os métodos, `gestaoDeAlunos()` e `gestaoDeTurmas()` correspondem aos estados "Gestão de Alunos" e "Gestão de Turmas", respectivamente. Note que os métodos correspondentes aos estados "Adicionar Aluno", "Consultar Aluno", "Listar Alunos", "Adicionar Turma", "Listar Turmas" e "Mudar Sala à Turma", já estão implementados. O que necessita fazer é definir os menus, com as opções, pré-condições e *event handlers* adequados.

   Teste a aplicação para verificar que funciona correctamente. De modo particular, verifique se a interface obedece ao que está definido na Figura 1.7.

4. Implemente, agora, o tratamento de erros através da utilização de excepções. Terá que acrescentar excepções na camada de negócio, tratando-as na camada de interface.

   Verifique, mais uma vez, que tudo funciona correctamente.

5. A implementação usual da persistência é realizada com recurso a uma Base da Dados (tema a abordar no próximo semestre, numa outra Unidade Curricular).

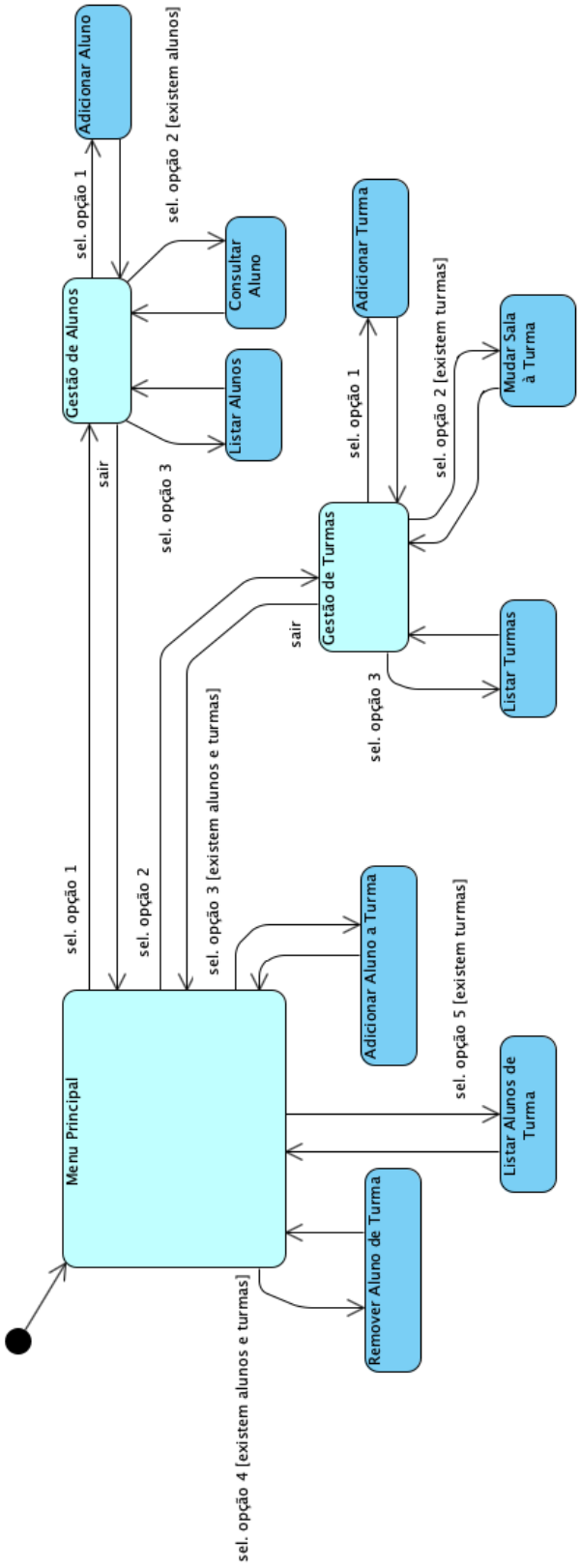


Figura 1.7: Mapa de navegação pretendido para a aplicação

Por agora, relembre o que estudou em POO sobre escrita e leitura de ficheiros em modo binário (com recurso às *object streams* do Java) e implemente a persistência dos dados gravando e lendo os `Map` em ficheiro.

O método `getInstance()` deverá ler o `Map` de alunos/turmas (dependendo do DAO) e terá que adicionar um outro método, a cada um dos DAO, para realizar a gravação. Os dados deverão ser gravados antes de sair do programa.