Ricardo Reis, 200262024

Rodrigo Nogueira, 200262002

2020/2021  Professor Fábio Varanda

Projeto POO

# Introdução

O objetivo deste relatório é descrever a aplicação desenvolvida para a gestão de um restaurante, utilizando a linguagem *Java* e o paradigma de programação orientada por objetos (*POO*). Foi pedido uma aplicação que permita aplicar soluções para os conceitos aprendidos no decorrer da unidade curricular, os quais permitem satisfazer um número de princípios fundamentais da Engenharia de Software, nomeadamente: a modularidade conceptual e de implementação, a correção de erros e eficiência.

# Classes Implementadas

## Restaurante.java

Graphical user interface, text, application, email

Description automatically generated

Fig. 1 - Classe Restaurant

Text

Description automatically generated

Fig. 2 - Classe RestaurantManagement

A classe *Restaurant*(a *main*) chama um método estático da classe *RestaurantManagement* para não ser necessário criado um objeto desta mesma classe, uma vez que o método serve para verificar se o ficheiro binário já existe. Se esse ficheiro já existir, vai ser lido, caso contrário é criado o objeto da classe *Management* e inicializa o programa.

Graphical user interface, text, application, email

Description automatically generated

Fig. 3 - Método para guardar no ficheiro binário

Graphical user interface, text, application, email

Description automatically generatedEste é o método dentro da classe RestaurantFileHandler que permite ao programa guardar os dados do restaurante em um ficheiro binário.

Fig. 4 - Ler o ficheiro binário

Este método também está dentro da classe *RestaurantFileHandler* que permite ao programa carregar dados do ficheiro binário.

## Table.java

Graphical user interface, text, application

Description automatically generated

Fig. 5 - Classe Table

Nesta classe é definido o número da mesa, o estado de ocupação da mesma e contém apenas um pedido.

## Order.java

Text

Description automatically generated

Fig. 6 - Classe Order

Text

Description automatically generated with low confidence

Fig. 7 - Estados do pedido

A classe *Order* é composta por um conjunto de Itens, o seu estado(Fig. 7), a data/hora em que o pedido é aberto e a data/hora de fecho do mesmo. Sempre que um pedido é fechado é registada a data/hora atual.

## Item.java

Text

Description automatically generated

Fig. 8 - Classe Item

Nesta classe define-se a quantidade de cada produto.

## Product.java

Graphical user interface, text, application

Description automatically generated

´

Fig. 9 - Classe Product

Graphical user interface, text, application

Description automatically generatedA classe *Product* é uma classe abstrata pois existem quatro tipos de produtos e cada um com a própria classe, métodos e atributos. Cada Subclasse derivante herda os atributos: nome, preço e *iva*. O iva é sempre introduzido como número inteiro mas convertido para decimal para facilitar os cálculos.

Fig. 10 - Classe Drink

Na Subclasse *Drink* é definida a capacidade da bebida, tendo um mínimo de 0,33L , e se é alcoólica.

## Dish.java

Graphical user interface, text, application

Description automatically generated

Fig. 11 - Classe Dish

Graphical user interface, text, application, email

Description automatically generatedA classe *Dish* contém a descrição do prato, pois o nome, preço e iva são atributos da classe Product.

Fig. 12 - Classe Snack

Na classe *Snack* é definida a quantidade apesar de estre atributo não afetar o preço total do pedido. Apenas a quantidade definida no Item irá influenciar o preço final.

Graphical user interface, text, application

Description automatically generated

Fig. 13 - Classe Sweet

A classe Sweet assim como a classe *Dish* também tem uma descrição, adicionalmente existe um boolean que indica se o doce é confecionado ou não no próprio restaurante.

## History.java

Graphical user interface, text, application, email

Description automatically generated

Fig. 14 - Método do histórico para ordenar pedidos do mais recente para o mais antigo

A classe *History* tem uma *arraylist* de pedidos(*Order*) como atríbuto e como método o *sortOrder* que é chamado cada vez que é adicionado um novo pedido.

No ciclo do-while existe um objeto da classe *Order*(*temporaryOrder*) que é utilizado para transporte de pedidos entre os índices do orderList e uma variável booleana que por defeito é falsa e convertida para verdadeira sempre que é feita uma ordenação dos índices para sinalizar o recomeço do ciclo pois pode existir mais índices por ordenar.

No ciclo que percorre o arraylist da lista de pedidos(*for-loop*) é verificado se o índice após ao atual é nulo, ou seja, chegou-se ao final do ciclo, se for verdade é lançada uma exeção que dá por terminado o final do ciclo. Em contraste no else-if é comparada a data/hora de abertura do pedido atual com a data/hora do pedido do índice seguinte, caso o pedido atual seja mais antigo que o pedido seguinte, que é retornado um número inteiro negativo, é feita a troca do objeto do pedido do índice atual pelo objeto do índice seguinte.