PROJETO POO

Ricardo Reis, 200262024

Rodrigo Nogueira, 200262002

2020/2021 Professor Fábio Varanda

Contents

1. 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.

2. Classes Implementadas

2.1 Restaurante.java

```java
package restaurant;

/**
 *
 * @author Ricardo Reis     200262024 200262024@estudantes.ips.pt
 *         Rodrigo Nogueira 200262002 200262002@estudantes.ips.pt
 */
public class Restaurant {

    public static void main(String[] args){
        RestaurantManagement.startProgram();

    }
}
```

Fig. 1 - Classe Restaurant

```java
package restaurant;

/**
 *
 * @author Ricardo Reis     200262024 200262024@estudantes.ips.pt
 *         Rodrigo Nogueira 200262002 200262002@estudantes.ips.pt
 */

public class RestaurantManagement {
    private static Management restaurante;

    public static void startProgram(){

        RestauranteFileHandler saveFiles = new RestauranteFileHandler();

        if(saveFiles.readSerializedFile("savedata.bin") == null)
            restaurante = new Management();
        else
            restaurante = saveFiles.readSerializedFile("savedata.bin");

        restaurante.menu();
        saveFiles.saveFile(restaurante, "savedata.bin");
    }
}
```

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.

```java
public void saveFile(Management management, String filename) {
    try {
        File destination = makeAbsoluteFilename(filename);
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(destination));
        oos.writeObject(management);
        oos.flush();
        oos.close();

    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Fig. 3 - Método para guardar no ficheiro binário

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

```java
public Management readSerializedFile(String filename) {
    Management management;

    try {
        File destination = makeAbsoluteFilename(filename);
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(destination));
        management = (Management) ois.readObject();
        ois.close();

        return management;

    } catch (IOException | ClassNotFoundException e) {
        return null;
    }
}
```

Fig. 4 - Ler o ficheiro binário

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

2.2 Table.java

```
public class Table implements Serializable{

    private int tableNumber;
    private Order order;
    private boolean occupied;

    public Table(int number) {
        if(number > -1)
            this.tableNumber = number;
        else
            this.tableNumber = -1;

        order = new Order();
        occupied = false;
    }
```

Fig. 5 - Classe Table

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

2.3 Order.java

```
public class Order implements Serializable{
    private ArrayList<Item> itemList;
    private orderState state;

    private LocalDateTime openHour;
    private LocalDateTime closeHour; // = LocalDateTime.now();

    public Order(){
        itemList = new ArrayList<>();
        state = orderState.OPEN;
    }
```

Fig. 6 - Classe Order

```
public enum orderState {
    OPEN, PREPARATION, SERVED, CLOSED;
}
```

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 iguala-se a data/hora atual à variável de fecho(*closeHour*).

2.4 Item.java

```
public class Item implements Serializable{
    private Product product;
    private int quantity;

    public Item() {
        product = null;
        quantity = 0;
    }
```

Fig. 8 - Classe Item

Nesta classe define-se a quantidade de cada produto.

2.5 Product.java

```
public abstract class Product implements Serializable{

    private String name;
    private double price;
    private double iva;

    public Product(){
        this.price = 0;
        this.name = "Um produto qualquer";
        iva = 0.23;
    }
```

Fig. 9 - Classe Product

A 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.

```
public class Drink extends Product implements Serializable{

    private double capacity;
    private boolean hasAlcohol;

    public Drink() {
        this.capacity = 0.33;
        this.hasAlcohol = false;
    }
```

Fig. 10 - Classe Drink

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

2.6 Dish.java

```
public class Dish extends Product implements Serializable{
    private String description;

    public Dish(String name, double price, double iva, String description) {
        super(name, price, iva);

        if(description != null || !description.trim().equals(""))
            this.description = description.trim();
    }
```

Fig. 11 - Classe Dish

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

```
public class Snack extends Product implements Serializable{
    private int quantity;
    private boolean isSpicy;

    public Snack(String name, double price, double iva, int quantity, boolean isSpicy) {
        super(name, price, iva);

        if(quantity > 0)
            this.quantity = quantity;
        else
            this.quantity = 0;

        this.isSpicy = isSpicy;
    }
```

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.

```java
public class Sweet extends Product implements Serializable{

    private String description;
    private boolean madeInRestaurant;

    public Sweet(String name, double price, double iva, String description, boolean madeInRestaurant) {
        super(name, price, iva);

        if (description != null || !description.trim().equals(""))
            this.description = description.trim();

        this.madeInRestaurant = madeInRestaurant;
    }
```

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.

2.7 History.java

```
private void sortOrder(){
    switch(orderList.size()){
        case 1:
            return;
    }

    // Caso haja uma ordenação no percorrer do ciclo, provavelmente vai
    //haver mais. O ciclo de ordenação só pára quando estiver false
    boolean toOrder;

    // Pega num pedido temporariamente para o mover para um indice diferente
    do{
        Order temporaryOrder = new Order();
        toOrder = false;
        try{
            for(int i = 0; i < orderList.size(); i++){
                if(orderList.get(i+1) == null)
                    break;
                else if(orderList.get(i).getOpenHour().compareTo( orderList.get(i+1).getOpenHour() ) < 0 ){
                    temporaryOrder = orderList.get(i+1);

                    orderList.set(i+1, orderList.get(i));
                    orderList.set(i, temporaryOrder);
                    toOrder = true;
                }
            }
        }catch(IndexOutOfBoundsException e){

        }
    }while(toOrder);
}
```

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.

2.8 Management.java

2.8.1 addProduct()

```
while(true){
    try{
        Menu.mainMenuProducts();
        option = scan.getInt("Que tipo de produto quer adicionar? ");

        if(option < 0 || option > 4)
            throw new InvalidInputArgumentException("ERRO! Opção inválida!");
        else if(option == 0)
            return;

        //Verifica se o nome não está em branco ou se já existe um produto com o nome introduzido
        do{
            try{
                productName = scan.getString("Nome");
                if("0".equals(productName))
                    return;
                else if(productName == null || productName.trim().equals(""))
                    throw new InvalidInputArgumentException("ERRO: Nome não pode ficar em branco!");
                else{
                    checkProductDuplicates(productName);
                    break;
                }

            }catch(InvalidInputArgumentException e){
                System.err.println(e.getMessage());
            }
        }while(true);
```

Fig. 15 - Método addProduct() – PARTE 1

Na classe principal para adicionar produtos é utilizado o método addProduct, sempre que o utilizador escolhe o tipo de produto que pretende adicionar é perguntado o nome e em seguinda é verificado se já existe esse mesmo produto baseado no nome.

```
    //Verifica se o preço não é zero ou número negativo
    do{
        try{
            productPrice = scan.getDouble("Preço");
            if(productPrice == 0)
                throw new InvalidInputArgumentException("ERRO: Preço não pode ser zero!");
            else if(productPrice < 0)
                throw new InvalidInputArgumentException("ERRO: Preço não pode ser negativo!");
            break;
        }catch(InvalidInputArgumentException e){
            System.err.println(e.getMessage());
        }
    }while(true);

    do{
        try{
            productIva = scan.getDouble("IVA (0-100%)");
            if(productIva > 100)
                throw new InvalidInputArgumentException("ERRO: IVA não pode exceder os 100%!");
            else if(productIva < 0)
                throw new InvalidInputArgumentException("ERRO: IVA não pode ser negativo!");
            break;
        }catch(InvalidInputArgumentException e){
            System.err.println(e.getMessage());
        }
    }while(true);

    break;

}catch(InvalidInputArgumentException e){
    System.out.println(e.getMessage());
}
```

Fig. 16 - Método addProduct() - PARTE 2

Se o utilizador introduzir um novo nome para um produto é pedido o preço e iva.

Este três atributos são comuns a todos os produtos no programa.

```
switch(option) {
    case 1:
        addDrink(productName, productPrice, productIva);
        break;
    case 2:
        addSweet(productName, productPrice, productIva);
        break;
    case 3:
        addDish(productName, productPrice, productIva);
        break;
    case 4:
        addSnack(productName, productPrice, productIva);
}
```

Fig. 17 - Menu de distribuição

Este excerto de código iniciar o método respetivo para a criação de cada Produto.

2.8.2 addDrink()

```
private void addDrink(String drinkName, double drinkPrice, double productIva) {
    Drink drink = new Drink();
    InputReader scan = new InputReader();

    drink.setName(drinkName);
    drink.setPrice(drinkPrice);
    drink.setIva(productIva);

    while (true)
        try{
            drink.setCapacity(scan.getDouble("Capacidade da bebida (L)"));
            break;
        }catch(InvalidInputArgumentException e){
            System.err.println(e.getMessage());
        }

    while (true)
        try{
            drink.setHasAlcohol(scan.getString("Bebida alcoólica? (s/n)").toLowerCase().charAt(0));
            break;
        }catch(InvalidInputArgumentException e){
            System.err.println(e.getMessage());
        }

    productList.add(drink);
    System.out.println("++ Bebida adicionada com sucesso! ++");

}
```

Fig. 18 - Método addDrink()

Os métodos respetivos a cada Produto regem-se pelo mesmo padrão onde é introduzido o valor dos atributos em comum(nome, preço, iva) e em seguida é pedido ao utilizador os dados pertencentes às variáveis das devidas Subclasses do *Product* antes de serem adicionadas ao *arraylist* dos Produtos.

2.8.3 removeProduct()

```
private void removeProduct(){
    int i;
    Menu.listProducts(productList);
    InputReader scan = new InputReader();

    while(true){
        try{
            i = scan.getInt("Introduza o número do produto que deseja remover");
            if(i == -1)
                return;
            else if(i <= productList.size() && i >=1)
                break;
            throw new InvalidInputArgumentException("ERRO: Indique o número do produto apresentado na lista!");

        }catch(InvalidInputArgumentException e){
            System.err.println(e.getMessage());

        }

    }

    --i;
    for(Iterator<Product> it = productList.iterator(); it.hasNext(); )
        if ( it.next().getName().equals( productList.get(i).getName() ) ) {

            System.out.println("--" + productList.get(i).getName() + " removido com sucesso!");
            it.remove();

            break;
        }
}
```

Fig. 19 - Método removeProduct()

É mostrado uma lista numerada de produtos introduzidos previamente, bastando apenas ser necessário inserir o número correspondente ao produto.

```
=== Top Wings === Versão 0.6.0 07/06/2021 22:41
* 1 - Adicionar produto
* 2 - Remover produto
* 3 - Listar produtos
* 4 - Reservar uma nova mesa
* 5 - Editar uma mesa
* 6 - Mostrar histórico de pedidos
* 0 - Sair da aplicação
**************
Escolha uma opção: > 2
1) Bebida -> Nome: Danke | Preco: 1.0 (com IVA: 1.01) | IVA: 1 | Capacidade: 0.34 L | Bebida álcoolica
2) Doce   -> Nome: Kit Kat | Preco: 2.0 (com IVA: 2.22) | IVA: 11 | Chocolate |
3) Prato  -> Nome: Lasanha Vegetariana | Preco: 6.0 (com IVA: 6.72) | IVA: 12 | O prato perfeito depois de um longo dia a gerenciar um restaurante!
Introduza o número do produto que deseja remover>
```

Fig. 20 - Exemplo da remoção de produtos

2.8.4 bookTable()

```
private void bookTable(){
    try{
        if(!checkForUnoccupiedTables())
            throw new InvalidInputArgumentException("ERRO: Não há mesas disponíveis neste momento, feche um pedido.");
    }catch(InvalidInputArgumentException e){
        System.err.println(e.getMessage());
        return;
    }

    InputReader scan = new InputReader();
    int tableNumber;
    char tableOption;

    while(true){
        Menu.listTables(tableList);

        try{
            tableNumber = scan.getInt("Introduza o número da mesa que deseja reservar");
            --tableNumber;
            if(tableNumber == -1)
                return;
            else if(tableList[tableNumber].isOccupied() == true)
                throw new InvalidInputArgumentException("ERRO: Mesa já está reservada!");
            else if(tableNumber<= tableList.length && tableNumber >= 0 && !tableList[tableNumber].isOccupied())
                break;
        }catch(ArrayIndexOutOfBoundsException OutOfBounds){
            System.err.println("ERRO: Indique o número da mesa apresentado na lista!");
        }catch(InvalidInputArgumentException e){
            System.err.println(e.getMessage());
        }
    }
```

Fig. 21 - Método bookTable() - PARTE 1

É verificado se existem mesas disponiveis para uma reserva, caso existam é demostrado uma lista com as mesas do restaurante e é pedido ao utilizador que mesa, que não esteja já ocupada, deseja efetuar uma reserva.

```
for(int m = 0; m <= tableList.length; m++){
    if (tableList[tableNumber].equals(tableList[m])) {

        do{
            tableOption = scan.getString("Reserva-se a mesa para (a)gora ou para mais (t)arde?").toLowerCase().charAt(0);
        }while(tableOption != 'a' && tableOption != 't');

        Order temporaryOrder = new Order();
        switch(tableOption){
            case 'a':
                temporaryOrder.openOrder(LocalDateTime.now());
                tableList[tableNumber].setOccupied(); // Se a mesa é reservada para agora, é porque vai estar ocupada agora
                tableList[tableNumber].setOrder(temporaryOrder);
                editTable(tableList[tableNumber]);
                break;
```

Fig. 22 - Método bookTable() - PARTE 2

Esta parte do método pergunta ao utilizador se pretende reservar a mesa para uma data futura ou no momento. Se reserva para o momento o programa abre o pedido.

```
do{
    try{
        hasErrors = false;
        dateInput = scan.getString("Introduza a data (DD-MM-AAAA)").trim(); // 2-12-2021

        if(dateInput.length() <= 10){
            if(dateInput.contains("-"))
                formattedDate = dateInput.split("-");

            else if(dateInput.contains("/"))
                formattedDate = dateInput.split("/");

            else if(dateInput.contains(" "))
                formattedDate = dateInput.split(" ");
        }
```

Fig. 23 - Data para reversa futura

Caso seja feita uma reserva para uma data futura, o programa irá pedir pela data que se pretende reservar assim como as hora e minutos com as respetivas validações.

2.8.5 editTable()

```
private void editTable(Table table){
    int productNumber;
    char getChar=' ';
    InputReader scan = new InputReader();

    if(!checkTableOrderState(table) || table == null)
        return;

    while(true){
        Item item = new Item();
        try{
            Menu.listProducts(productList);

            productNumber = scan.getInt("Introduza o número dos produtos que deseja adicionar (0 - Sair)");
            --productNumber;

            if(productNumber == -1)
                return;
            else if(productNumber < productList.size() && productNumber >= 0)
                item.setProduct(productList.get(productNumber));
            else
                throw new InvalidInputArgumentException("ERRO: Indique o número do produto apresentado na lista!");

            while(true){
                try{
                    productNumber = scan.getInt("Introduza a quantidade");

                    if(productNumber == 0)
                        return;
                    else if(productNumber < 0)
                        throw new InvalidInputArgumentException("ERRO: Quantidade não pode ser negativa!");
```

Fig. 24 - Método editTable()

Neste método é apresentado mais uma vez a lista de produtos para adicionar um pedido da respetiva mesa, para tal, é auxiliado pelo método *selectTable* que é utilizado para receber a mesa como parametro.

```
case 5:
    editTable(selectTable());
    break;
```

Fig. 25 - Utilização do selectTable() no menu principal

Também é utilizado como auxilio o método checkTableOrderState para caso a data atual não for superior à data do pedido da mesa, ser possível fechar o pedido ou para serem adicionados mais produtos, passar o estado do pedido para servido e fechar o pedido, sendo este introduzido no histórico.