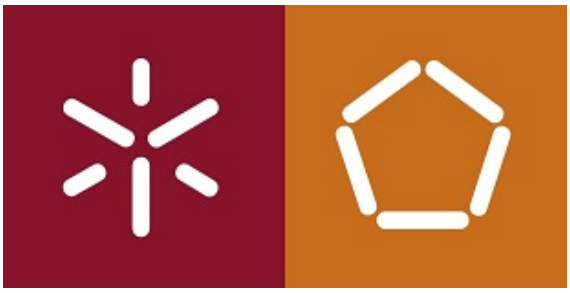

Universidade do Minho
Departamento de Informática

Smart House

Programação Orientada aos Objetos

Grupo 25

21 de maio de 2022

Guilherme Lima Barros Gomes Fernandes (a93216)


Mariana Rodrigues (a93229)


Jéssica Macedo Fernandes (a93318)

Conteúdo

Capítulo 1

Introdução

No âmbito da Unidade Curricular de Programação Orientada a Objetos, foi-nos proposto desenvolver um projeto cujo objetivo principal fosse o de monitorizar e registar informações acerca o consumo energético de habitações de uma certa comunidade, utilizando o paradigma da programação orientada a objectos.

O desenvolvimento deste projeto teve sempre em mente os objetivos do Paradigma Orientado aos Objetos e da linguagem de programação *Java*, nomeadamente, a modularidade e reutilização do código, a abstração e a hierarquia de classes.

Capítulo 2

Implementação

De modo a conseguirmos obter uma melhor organização deste projeto, começamos por definir a arquitetura usar, **MVC (Model View Controller)**. Desta forma, foi nos possível obter um desenvolvimento mais rápido e escalável, sendo que, através desta arquitetura qualquer alteração ou adição feita ao *Model* não implicaria uma mudança no *Controller e na View*.

2.1 Descrição da arquitetura de classes

Como outrora mencionada, a aplicação foi concebida tendo em mente o modelo de desenvolvimento *MVC*. Assim sendo, organizamos este projeto em três grandes packages:

- **Model:** Encontra-se responsável por todas as classes que são utilizadas no armazenamento de estruturas de dados e pela parte algorítmica do programa.
- **Controller:** Encontra-se responsável por controlar todo o funcionamento do programa.
- **View:** Encontra-se responsável por todas as classes que oferecem resposta visual aos pedidos do utilizador.

2.2 Diagrama de Classes

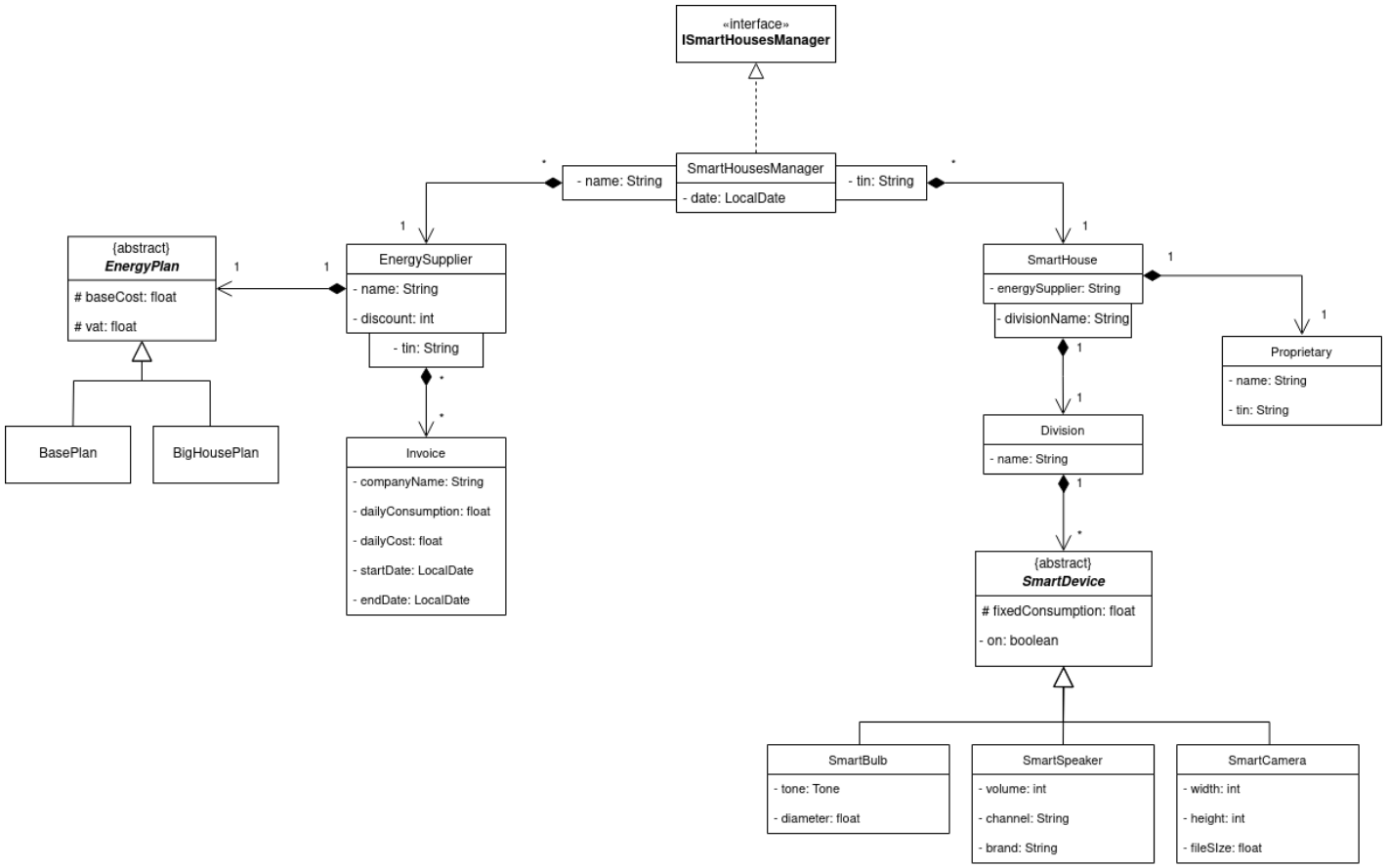

Figura 2.1: Diagrama de Classes

2.3 Model

Como já mencionado, o propósito do **model** é a gestão dos dados.

2.3.1 SmartDevice

Visto termos de ser capazes de representar um dispositivo de uma casa, criamos a classe **SmartDevice**. Tendo em vista a hierarquia de classes, decidimos tornar esta classe **abstract**. Deste modo, conseguimos a generalizacação de um dispositivo independentemente do seu tipo (lâmpada, câmara ou coluna) e permitimos assim, a partilha de algumas variáveis de instância entre as diversas classes.

```
public abstract class SmartDevice implements Serializable {
    protected final float fixedConsumption;
    private boolean on;
    ...
}
```

Como podemos observar, um **SmartDevice** terá sempre um custo fixo (**fixedConsumption**) e um booleano que nós indicara se o dispositivo se encontra ou não ligado (**on**).

Neste projeto teremos três tipos diferentes de *smart devices*: uma câmara, uma coluna e uma lâmpada.

```
public class SmartCamera extends SmartDevice implements Serializable {
    private final int width;
    private final int height;
    private final float fileSize;
    ...
}

public class SmartSpeaker extends SmartDevice implements Serializable {
    public static final int MAX = 100;
    public static final int MIN = 0;

    private int volume;
    private String channel;
    private String brand;
    ...
}

public class SmartBulb extends SmartDevice implements Serializable {
    private Tone tone;
    private float diameter;
    ...
}
```

Falemos especificamente, ainda, sobre a classe **SmartBulb** que implementa, como o nome sugere, a lâmpada. Um dos atributos da lâmpada é o seu tom, que por sua vez pode tomar os seguintes valores: **WARM**, **NEUTRAL** ou **COLD**. Estes valores são responsáveis por indicar o fator de energia gasto pela lâmpada de acordo com o modo (tom) em que se encontra, quando ligada.

```
public enum Tone implements Serializable {
    WARM(1),
    NEUTRAL(2),
    COLD(3);
    ...
}
```

2.3.2 Proprietary

A classe **Proprietary** é responsável por representar um proprietário. Esta classe é muito simples, possuindo apenas dois atributos, o nome do proprietário da casa, e o seu número de identificação fiscal. Aqui, é de salientar que a identificação fiscal de cada um dos proprietários é única.

```
public class Proprietary implements Serializable {
    private final String name;
    private final String tin;
    ...
}
```

2.3.3 EnergySupplier

A classe **EnergySupplier** é responsável por representar um fornecedor de energia. Para tal, este terá um nome associado, um plano de energia, um determinado desconto e todas as faturas aplicadas aos proprietários.

```
public class EnergySupplier implements Serializable {
    private final String name;
    private EnergyPlan energyPlan;
    private int discount;
    private final HashMap<String, List<Invoice>> invoicesByProprietaryTin;
    ...
}
```

EnergyPlan

De modo a ser possível um fornecedor de energia conter um plano de energia, desenvolvemos a classe **EnergyPlan**.

Com o intuito de permiter existirem diversos planos de energia, esta classe é *abstrata*. Obrigando a que qualquer plano de energia tivesse, pelo menos, um preço base(que dependerá do tipo de plano), um imposto(correspondente à variável **vat**) e um método que calcula-se o custo da energia.

```
public abstract class EnergyPlan implements Serializable {
    protected static float baseCost;
    protected static float vat;
    ...
    public abstract float energyCost(int numDevices, float devicesConsumption);
    ...
}
```

De modo a termos diferentes tipos de planos, decidimos então ter dois tipos de planos disponíveis. Um plano *standard*(**BasePlan**), e um para casas consideradas maiores(**BigHousePlan**), no sentido de terem mais dispositivos consumidores de energia. As classes que implementam estes dois planos não têm nenhuma instância para além das da classe abstrata que extendem.

2.3.4 Invoice

Esta classe encontra-se responsável por representar uma fatura, esta é constituída pelo nome da companhia de energia responsável pela prestação de serviço, o consumo total da casa por dia, o total a pagar, e as datas de início e fim do período de consumo.

```
    public class Invoice implements Serializable {
        private final String companyName;
        private final float dailyConsumption;
```

```
        private final float cost;
        private final LocalDate startDate;
        private final LocalDate endDate;
        ...
    }
```

2.3.5 SmartHouse

Esta classe encontra-se responsável por representar uma casa. Esta será constituído por um proprietário bem como o nome da companhia atualmente contratada para prestar os serviços de energia e um conjunto de divisões por onde os *devices* serão distribuídos.

```
public class SmartHouse implements Serializable {
    private final Proprietary proprietary;
    private final Map<String, Division> divisionsByName;
    private String energySupplier;
    ...
}
```

2.3.6 Division

Uma divisão conterá apenas a lista de dispositivos associados à própria, e um nome que permitirá aceder com facilidade ao seu conteúdo. Isto será explicado melhor quando virmos a implemntação da classe **SmartHouse**.

```
public class Division implements Serializable {
    private final String name;
    private final List<SmartDevice> smartDevices;
    ...
}
```

2.3.7 SmartHouseManager

Esta classe encontra-se responsável por agrupar todas as classes anteriores. Deste modo, conseguimos monitorizar e modificar todos os proprietários, todas as *smartHouses* e todos os *energySuppliers*.

```
public class SmartHousesManager implements Serializable, ISmartHousesManager {
    private final Map<String, SmartHouse> smartHousesByTIN;
    private final Map<String, EnergySupplier> energySuppliers;
    private LocalDate date;
    ...
}
```

Para finalizar, criou-se a interface **ISmartHousesManager** de modo a que mais a frente, em vez de se utilizar uma classe diretamente utilizasse uma interface, permitindo assim a utilização de diferentes implementações.

É de notar que todas as classes aqui apresentadas extendem **Serializable**,necessário para as poder gravar num ficheiro de objetos.

2.4 Controller

O **Controller** é responsável por unir o **Model** e a **View**, chamando os métodos necessários do primeiro, de modo a satisfazer os pedidos do utilizador.

Para tal, este encontra-se dividido:

- **Parser:** Responsável pela leitura dos logs e dos eventos.
- **State:** Classe principal, que controla todas as ações necessárias.

```
public class State implements IState {
    private ISmartHousesManager smartHousesManager;
    ...
}
```

- **IState:** Interface principal do **Controller**

2.5 View

A **View** é responsável por decidir o que o utilizador vê e como o vê, recebendo os pedidos do mesmo.

O módulo **IO** encontra-se responsável por enviar os pedidos do utilizador ao **IState**. Para tal, foi essencial a criação da classe **MenuCatalog**.

```
public class MenuCatalog<T> {
  private final List<Menu<T>> menus;
  private int option;
  ...
  public void run(T that) {...}
  ...
}
```

Esta classe contém uma lista de todos os menus possíveis a serem executados, assim foi possível a divisão de todas as opções que o utilizador pode executar por tópicos. De forma sucinta, isto representa um menu, que, selecionada uma opção, nos levará a outros menus com mais opções.

```
public class Menu<T> {
  private final List<OptionCommand<T>> options;
  private final String description;
  private int option;
  private boolean stop;
  ...
 }
```

Cada **Menu** será responsável pela execução de uma lista de **OptionCommand**s guardadas por este. Este conterá uma breve descrição do que será responsável, a última opção que terá sido executada e um *boolean*, este servirá para informar se após a primeira seleção de uma das opções, deverá terminar ou continuar com a execução desse mesmo menu, até o utilizador selecionar a opção de saída.

```
public final class OptionCommand<T> {
  private final String command;
  private final Consumer<T> function;
  private final boolean stop;
  ...
}
```

Uma **OptionCommand** é um objeto que guarda um **Consumer**, posteriormente aplicado ao objeto dado inicialmente no **MenuCatalog** aquando a execução do método **run**.
Conterá também uma pequena descrição da sua utilidade e um *boolean* **stop** que serve apenas para informar o **Menu**, que o chamou, que deverá ser terminado, isto é, que deverá voltar ao menu principal.
Com isto foi-nos possível uma maior eficácia em termos de organização, de remoção e de adição de novas funcionalidades.

Capítulo 3

Ilustração das funcionalidades

Ao ser inicializado o programa, é apresentado ao utilizador o menu principal, proporcionando-lhe a oportunidade de escolher uma das diversas opções disponíveis de forma a conseguir interagir com a aplicação.

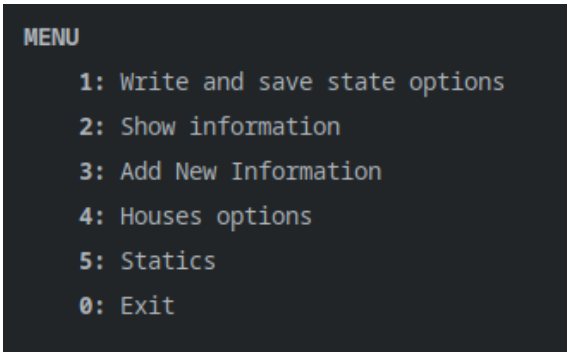

Figura 3.1: Menu Principal

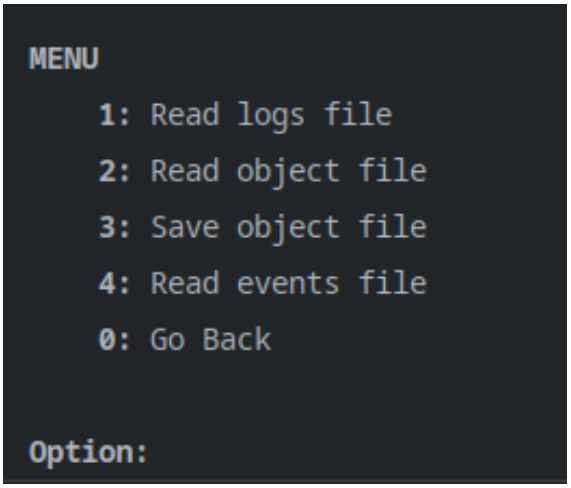

Figura 3.2: Menu Write and save state options

```
MENU
    1: Show all proprietaries
    2: Show all proprietary's devices
    3: Show all proprietary's devices and division
    4: Show device
    5: Show all energy plans
    6: Show all energy suplliers's name
    7: Show date
    0: Go Back
```

Figura 3.3: Menu Show information

```
MENU
    1: Add new proprietary and smart house
    2: Add new devices
    3: Add new energy supplier
    0: Go Back

Option:
```

Figura 3.4: Menu Add New Information

```
MENU
    1: SmartBulb
    2: SmartCamera
    3: SmartSpeaker
    0: Go Back

Option:
```

Figura 3.5: Menu Add New Devices

```
MENU
    1: Control date
    2: Turn ON/OFF smart devices
    3: Change energy plan
    4: Change Energy supllier discount
    5: Control devices
    0: Go Back

Option:
```

Figura 3.6: Menu Houses options

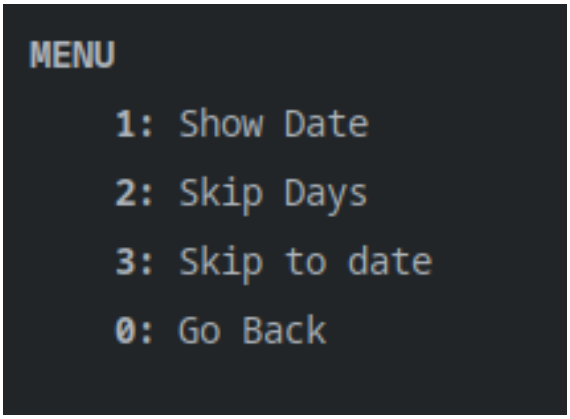

Figura 3.7: Menu Control date

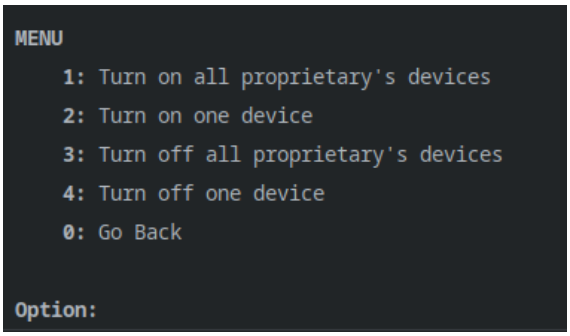

Figura 3.8: Menu Turn ON/OFF smart devices

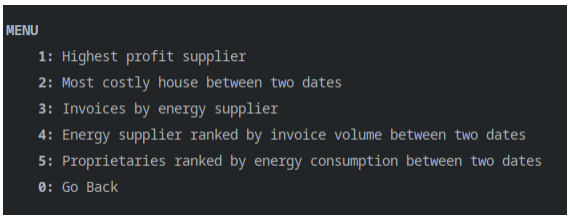

Figura 3.9: Menu Statics

Capítulo 4

Conclusão

Através da realização deste trabalho prático foi nos possível consolidar e aprofundar melhor a matéria lecionada nas aulas teórica, nomeadamente o paradigma da programação orientada a objetos e a arquitetura **MVC (Model Controller View)**,

Para finalizar, achamos que cumprimos com os objetivos propostos. Todavia, entendemos que existem pequenos pontos que poderiam vir a ser melhorados e novas funcionalidades a acrescentar.

No entanto, consideramos que realizamos um excelente trabalho, conseguindo obter resultados bastante satisfatórios.