Relatório do Trabalho Prático

Programação Orientada Aos Objetos

Grupo 18

Guilherme Sampaio, a96766

João Longo, a95536

Licenciatura em Engenharia Informática

21 de maio de 2022

Conteúdo

1 Introdução **1**

2 Estrutura da Aplicação **1**

3 Descrição da Aplicação **1**

3.1 SmartDevice: SmartSpeakers, SmartCameras e SmartBulbs 2

3.2 Room 2

3.3 PriceFormulas e EnergyProvider 3

3.4 Bill 4

3.5 House 4

3.6 Simulation e FileLoader 4

3.7 Controller e View 5

4 Funcionalidades Implementadas **5**

5 Conclusão **6**

1 Introdução

No âmbito da unidade curricular Programação Orientada aos Objetos, este relatório visa apresentar as estratégias e métodos empregues na realização do projeto, revelar complicações encontradas, assim como a descrição da aplicação. Deste modo, este relatório está dividido em 4 partes de onde 3 delas estão completamente implementadas.

2 Estrutura da Aplicação

Neste projeto, decidimos seguir com a estrutura *Model View Controller* (MVC). Ou seja, existem três camadas distintas, porém quando interligadas permitem o funcionamento da aplicação.

- O modelo é a ponto entre as camadas *View* e *Controller*, consiste na parte lógica da aplicação, que gere os dados e o seu comportamento sendo representado pelas classes: SmartDevice, SmartBulb, SmartSpeaker, SmartCamera, Resolution, House, Room, Simulation, Bill e EnergyProvider.
- A *View* faz o tratamento da representação de dados (texto escrito para o ecrã, tabelas ou diagramas) ao solicitar os dados ao *Model*. No projeto é representado pela classe View.
- A camada de controle (*Controller*) faz a mediação de entrada e saída, comandando a *View* e o emphModel para serem alterados de forma apropriada conforme o utilizador ordenou. É representado pelas classes FileLoader, Controller e PriceFormulas.

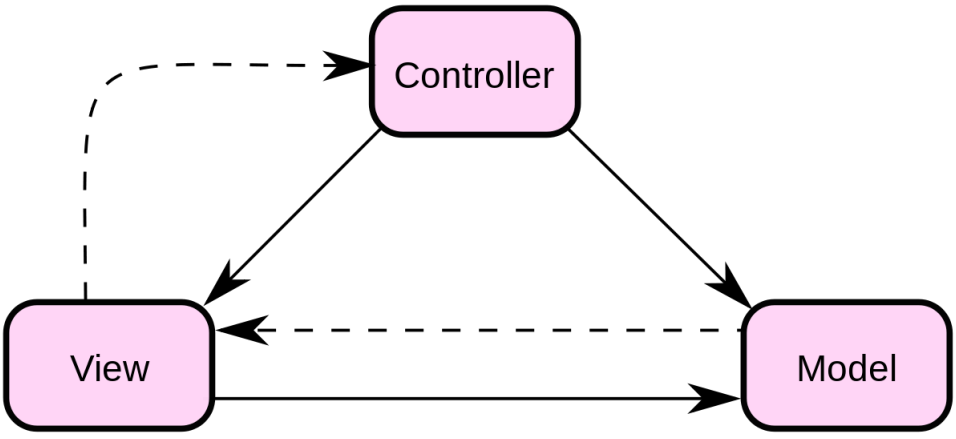

Figura 1: Arquitetura da Aplicação

3 Descrição da Aplicação

Neste capítulo todas as classes serão explicadas, isto é, qual a sua utilidade e o porquê da escolha de certas variáveis de classe, assim como a estratégia (composição/agregação) das mesmas.

3.1 SmartDevice: SmartSpeakers, SmartCameras e SmartBulbs

A classe abstrata SmartDevice é a base para o projeto inteiro. Através dela, foram criados três tipos diferentes de dispositivos, os SmartSpeakers, as SmartCameras e as SmartBulbs. Todos eles têm o comportamento base de SmartDevice, porém cada um tem algo que os distingue. Nos SmartSpeaker's podemos controlar o volume, nas SmartCamera's podemos modificar a sua resolução (classe Resolution) e nas SmartBulb's podemos alterar o tamanho e cor da lâmpada. Óbvio que estas propriedades refletem uma alteração no preço do consumo de cada dispositivo.

```
// Classe SmartDevice
private int deviceId;
private String deviceName;
private State deviceState;
private double baseCost;
```

```
// Classe SmartCamera extends SmartDevice
private Resolution resolution;
private int fileSize;
```

```
// Classe SmartBulb extends SmartDevice
public enum Tone
{
    NEUTRAL,
    COLD,
    WARM
}
private Tone color;
private double size;
```

```
// Classe SmartSpeaker extends SmartDevice
public static final int MAX = 100;
private int speakerVolume;
private String channel;
private String speakerBrand;
```

O valor do consumo diário de cada dispositivo é dado através de uma fórmula fixa (*hardcoded*) e em função dos seus atributos de classe.

3.2 Room

Como o objetivo do trabalho é controlar um determinado número de casas, a existência de divisões (Rooms) é inevitável.

```
// Classe Room
private String name;
private HashMap<Integer, SmartDevice> devices;
```

Uma divisão possui apenas um identificador (*String:name*) e um *Map* onde os dispositivos inteligentes são armazenados conforme o seu identificador numerário. Escusado será dizer que estes são os principais componentes das nossas casas.

3.3 PriceFormulas e EnergyProvider

```
private String nameId;
private double baseCost;
private double taxMargin;
private String formula = new PriceFormulas(FICHEIRO_DE_FORMULAS).
    getRandomFormula();
private List<Bill> bills;
```

Um dos requisitos do trabalho é a existência de comercializadores de energia (EnergyProvider). Estes possuem como variáveis o seu nome, o seu custo base por kilowatt de energia, a taxa de imposto, uma fórmula para o cálculo do valor final e uma lista que contém as faturas das casas que o têm como comercializador. A maior dificuldade na realização desta classe foi na geração e introdução de fórmulas, pois eu queria ter uma maneira de poder atribuir uma fórmula qualquer a um comercializador sem ter de criar uma função.

Para ultrapassar este problema decidi que ia gerar um ficheiro com fórmulas aleatórias em função de 3 variáveis:

- x: Representa o custo base por kilowatt de energia.
- y: Representa o valor do consumo energético diário do dispositivo.
- z: Representa o valor da taxa de imposto.

```
(Exemplo "formulas.txt")
x + y + z
2 * x + y + 2 * z
x + 3 * y * z
2 * x + 7 * z + y
```

Assim com a ajuda da classe PriceFormulas e do seu método $getRandomFormula()$ é possível atribuir a um consumidor uma *String* que representa a fórmula.

Mas como podemos retirar um valor numerário de uma *String* ? É aí que entra o método *String::replaceAll* que nos deixa substituir as variáveis por números e a biblioteca externa *org.mariuszgromada.math.mxparser* que nos permite criar objetos *Expression* que interpretam uma expressão matemática em formato de *String*. Deste modo, para obter um valor teríamos o seguinte processo:

```
"x + y + z" (formula base)
"1.5 + 4 + 10" (substituicao dos valores)
Expression e = new Expression("1.5 + 4 + 10") (transformacao em
    expressao)
e.calculate() (obtencao do valor final)
```

3.4 Bill

A classe Bill é a nossa maneira de representar uma fatura de um período de tempo. Está associado a uma casa (pelo seu identificador numerário - NIF) e são guardadas em cada comercializador de energia como dito acima.

```
// Classe Bill
private long deviceNum; --> numero total de dispositivos
private double powerUsed; --> total energia consumida
private int houseOwner; --> dono da casa
private LocalDate fromDate; --> inicio da faturacao
private LocalDate issueDate; --> fim da faturacao
private double totalCost; --> total a pagar
```

3.5 House

A classe House serve para armazenar as diferentes divisões da casa e consequentemente todos os dispositivos, para além disso, também possui o nome do seu dono, o número de identificação fiscal do mesmo e o provedor de energia.

```
// Classe House
private String ownerName;
private int ownerNIF;
private EnergyProvider energyProvider;
private final List<Room> rooms;
```

3.6 Simulation e FileLoader

```
// Classe Simulation
private final Map<String, EnergyProvider> energyProviders;
private final Map<Integer, House> houses;
```

A classe Simulation é a mais importante do projeto todo, pois é a que armazena tanto as casas (House) e os provedores de energia (EnergyProviders). Possui uma particularidade em relação aos objetos do tipo EnergyProviders, como podemos observar, esta classe possui uma estrutura que armazena os provedores. Estes objetos são partilhados com as casas, pois as casas possuem um provedor, assim, optei por guardar elementos desta classe usando uma estratégia de agregação, pois, deste modo, é possível alterar dados dos provedores situados em *energyProviders* e ver os frutos dessa alteração refletida nas casas.

Para além disso, como maneira de introduzir dados ao programa, optei por criar um *parser* de ficheiros de texto que gera duas listas, uma para os EnergyProvider e outra para as House.

```
// Classe FileLoader
private List<EnergyProvider> energyProviders;
private List<House> houses;
```

Ambas estruturas são convertidas em *Map* no construtor da classe Simulation para poderem ser utilizados.

3.7 Controller e View

Esta classe gere o fluxo da aplicação. É responsável por gerir os diversos menus do programa (gerados pela classe View), carregar dados e interagir com as outras classes do programa pertencentes ao *Model*. Tem um método principal, Controller:run, que inicializa e corre a aplicação inteira.

Atualmente, os menus disponíveis são:

```
(Main menu)

[1] Edit data. -> Permite editar dados, como por exemplo ligar e
   desligar dispositivos.

[2] Simulate passage of time. -> Permite simular a passagem do tempo.

[3] List stored data. -> Apresenta diversas opcoes de listagem de dados
[4] Execute queries. -> Executa queries feitas aos dados.

[5] Save current state. -> Guarda o estado atual da execucao do
   programa.

[6] Quit from program.
```

```
(Load menu)

[?] Load the data:
    [1] From binary file. -> Carregar ficheiros de dados binarios.
    [2] From text file.   -> Carregar ficheiros de texto.
    [3] Quit from program.
```

Cada opção do menu principal possui, porém, sub-menus que dão ao utilizador uma melhor experiência. Como, por exemplo:

```
(Edit menu)

[*] Edit Menu
    [?] Choose an element to edit:
        [1] Energy provider.
        [2] Switch house devices.
        [3] House.
```

4 Funcionalidades Implementadas

Neste projeto foram propostos quatro requisitos no que toca à implementação de funcionalidades:

- 1. Requisitos base de gestão das entidades.
- 2. Efetuar estatísticas sobre o programa.
- 3. Alterar os operadores e os dispositivos de uma casa.
- 4. Automatizar a simulação.

Destes todos o único não cumprido é o requisito número 4. De resto, esta aplicação apresenta as seguintes funcionalidades:

- 1. Simular passagem no tempo.
- 2. Editar os dados de provedores de energia (fórmula, valor base e valor de taxa).
- 3. Alterar o provedor de energia de uma casa.
- 4. Ligar e desligar os dispositivos de uma casa ou divisão.
- 5. Carregar dados via ficheiro binário ou de texto.
- 6. Guardar o estado atual de execução num ficheiro binário.
- 7. Executar as *queries* propostas.

Para além disso também podemos listar os dados de quatro maneiras diferentes: listar apenas as casas, as casas e as divisões, listar toda a informação sobre uma casa, listar os provedores de energia e listar as faturas de um certo provedor de energia.

5 Conclusão

Tendo em conta tudo o que foi exposto até agora, pensamos que temos um trabalho sólido. Acreditamos que conseguimos responder a todos os principais problemas propostos de uma boa maneira implementando as regras de encapsulamento (exceção ao solução do EnergyProvider referida acima). Para além disso graças à arquitetura da aplicação o projeto consegue crescer controladamente sem ficar demasiado confuso.