

Universidade do Minho

Escola de Ciências

UNIVERSIDADE DO MINHO

LICENCIATURA EM CIÊNCIAS DA COMPUTAÇÃO

Programação Orientada aos Objetos - Trabalho

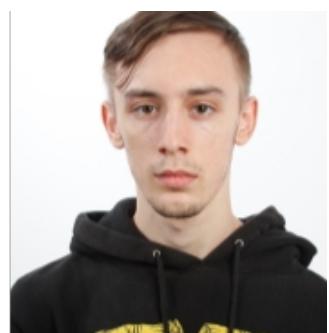
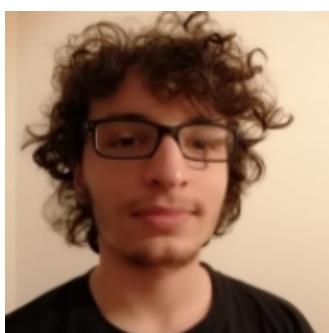
Prático

Grupo 19

André Lucena Ferreira (A94956) Carlos Machado(A96936)

Gonçalo Sousa (A97485)

Ano Letivo 2021/2022



Conteúdo

| | | |
|----------|---------------------------------------|-----------|
| 1 | Introdução | 4 |
| 2 | Model | 5 |
| 2.1 | CommunityApp | 5 |
| 2.1.1 | Command | 5 |
| 2.2 | Community | 6 |
| 2.2.1 | Provider | 6 |
| 2.2.2 | Invoice | 7 |
| 2.2.3 | SmartHouse | 8 |
| 2.3 | Exceptions | 9 |
| 3 | Controller | 10 |
| 3.1 | Controller | 10 |
| 3.1.1 | Parser | 10 |
| 3.1.2 | ParseActions | 11 |
| 3.1.3 | Métodos que fazem a ligação | 11 |
| 4 | View | 12 |
| 4.1 | Síntese | 12 |
| 4.2 | Menu | 12 |
| 4.2.1 | Handler | 13 |
| 4.2.2 | PreCondition | 13 |
| 4.3 | View | 13 |
| 4.3.1 | Menus da View | 13 |

| | |
|------------------------------|-----------|
| 5 Main | 15 |
| 6 Apresentação | 16 |
| 7 Diagrama de Classes | 17 |
| 8 Conclusão | 18 |

Capítulo 1

Introdução

O objetivo deste projeto foi a criação de uma aplicação capaz de simular as dinâmicas de um conjunto de casas (comunidade) como monitorizar e registar o consumo energético das mesmas. A aplicação é capaz de carregar um ficheiro *log* que contém uma série de instruções para criação de casas dispositivos e fornecedores, seja ele em texto ou em binário.

Capítulo 2

Model

Síntese

O *Model* é a pedra basilar do programa. Nele, temos toda a lógica estrutural, de armazenamento e de cálculo. O *Model* é completamente independente das outras estruturas, *View* e *Controller*, e essas estruturas não conhecem as especificidades das classes que este contém.

2.1 CommunityApp

Esta classe serve como *Facade* do resto do *Model*. Desse modo, a *CommunityApp* também tem como responsabilidade traduzir entre tipos de dados genéricos e não estruturados, entregues pelo *Controller*, para estruturas e classes específicas do *Model*, isto é, da classe *Community*, como, por exemplo, do tipo *SmartHouse*.

Esta classe difere da *Controller* por ter conhecimento de quais as classes integrantes da estrutura interna do *Model*, nomeadamente da classe *Community*, que compõe como privada.

```
private Community community;
```

Potencialmente, seria possível compor mais comunidades num Mapa dentro de *CommunityApp*, tendo então capacidade para mais que uma comunidade de cada vez.

A classe também tem como variável de instância um inteiro *idDevice*. A classe responsabiliza-se então por atribuir os *id*'s aos dispositivos que entram, identificadores que não serão inseridos pelo utilizador.

2.1.1 Command

Esta classe abstracta, implementada segundo o padrão do mesmo nome, auxilia a *CommunityApp* nos casos em que a alteração dos dados da simulação só deve ocorrer após a faturação do ciclo atual, ao compor uma lista de elementos desta classe e executando-os quando suposto.

```
private List<Command> command;
```

A classe abstrata em si contém a data em que a ação deve ser executada. A única declaração abstrata da classe é o método *execute*, que deve ser implementado por cada *Command* dependendo do necessário. Estas implementações referem-se diretamente a funções da classe *Community* que é passado como parâmetro do método aquando da sua invocação. Devido à sua implementação genérica, este método é passível de atirar 7 das 9 exceções específicas do programa, que irão ser omitidas nos exemplos aqui discriminados.

```
public LocalDate executionDate;  
public abstract void execute(Community community);
```

Cada uma das implementações desta classe tem como variáveis de instância os parâmetros da função que irá chamar. As implementações no projeto são:

- setBaseConsumptionCommand;
- setProviderAlgorthmCommand; setProviderDiscountFactorCommand;
- setSmartHouseProviderCommand;
- turnDivisionCommand; turnSmartDeviceCommand;

Como a *CommunityApp* não invoca por si os métodos da *Community* que cada comando utiliza, para cada um destes comandos, a classe *CommunityApp* tem como público um método que os adiciona à sua lista de comandos a executar. Assim, para o *Controller*, a única maneira de fazer este tipo de alterações é adicioná-los à lista de execução futura, garantindo o pedido no enunciado.

2.2 Community

A classe *Community* funciona como cerne do *Model*, representando uma comunidade energética. Esta classe compõe dois mapas, um de *Provider* e outro de *SmartHouse*. A chave para cada um é, respetivamente, o nome e a morada. A restante variável de instância é a data atual da simulação.

```
private Map<String , Provider> providerMap ;  
private Map<String , SmartHouse> smartHouseMap ;  
private LocalDate currentDate ;
```

Quando se avança a data, o *Community* pede a cada *SmartHouse* que peça a emissão das faturas aos seus *Provider*. A *Community* em si não acede às faturas, sendo estas responsabilidade da classe *Provider*

2.2.1 Provider

A classe *Provider* representa um Fornecedor de energia. Como variáveis de classe, possuem a informação do valor base por KWh e do fator de impostos ambos definidos à partida e, por isso declarados como *static*. Para além disso dispõe de nome, fator de desconto, e compõe *Invoice*'s através de um mapa que relaciona *SmartHouse* com *Sets* desta classe. Por fim, também agrupa uma implementação não abstrata de *DailyCostAlgorithm* como o algoritmo que utiliza para calcular o custo diário. A ordem natural desta classe é pela faturação total seguida pelo nome do fornecedor.

```

private static double baseValueKWH = 2.4, taxFactor = 0.23;
private String name;
private Double discountFactor;
private Map<String , Set<Invoice>> invoiceMap;
private DailyCostAlgorithm dailyCostAlgorithm;

```

A classe *Provider*, apesar de ter um mapa cujas chaves são as moradas de *SmartHouse*, não guarda as casas que lhe estão associadas a um dado instante. Esse mapa não corresponde ao tempo atual do programa mas sim qualquer fatura que tenha sido emitida. Isto é relevante porque as *SmartHouse*'s podem trocar de *Provider* mas isso não leva a que os *Provider* eliminem as faturas já emitidas para essa casa. Assim, podem existir chaves cujas *SmartHouse*'s tenham trocado de fornecedor.

DailyCostAlgorithm

Esta classe abstrata funciona como maneira de definir os algoritmos de custo diário de um *Provider*, segundo tanto o padrão *Strategy*. Para tal, a classe implementa a interface *BiFunctor*, recebendo um *Provider* e uma *SmartHouse* e devolvendo o custo como *Double*.

```

public abstract class DailyCostAlgorithm
implements BiFunction<Provider , SmartHouse , Double>

```

Para além disso, cada implementação da classe abstrata concretiza o padrão *Singleton*, ao ter como variável de classe a única instância, mantendo como *private* tanto essa variável como o seu construtor. Para conseguir referência da única instância, utiliza-se o método *getInstance*.

As implementações presentes no programa são as dos exemplos do enunciado do Trabalho Prático. Adicionar novas consiste apenas em criar um novo algoritmo e adicionar casos onde for necessário, nomeadamente num menu e num método da classe *CommunityApp*: *setProviderAlgorithm*.

```

private static DailyCostAlgorithmOne singletonAlgorithm = null;
private DailyCostAlgorithmOne() { }
public static DailyCostAlgorithmOne getInstance() {
    if (singletonAlgorithm == null)
        singletonAlgorithm = new DailyCostAlgorithmOne();
    return singletonAlgorithm;
}

```

2.2.2 Invoice

Esta classe representa as faturas emitidas. Tem como variáveis de instância *start* e *end*, delimitadoras do período de tempo sobre o qual versa a fatura. Além disso contém *consumption* e *cost* que instanciam respetivamente o consumo da casa para a qual a fatura foi emitida e o valor a pagar incluindo os custos de instalação relevantes. Por fim, guarda também a morada da casa para a qual foi emitida e o nome do fornecedor que a emitiu. A ordem natural das Faturas é dada pelo consumo, seguida pela ordem dos períodos e, por fim, pela casa a que pertencem.

```

public class Invoice
implements Comparable<Invoice>, Serializable { ... }

```

2.2.3 SmartHouse

Esta classe representa as casas inteligentes. Tem como variáveis de classe a sua morada, identificadora única das casas, nome e NIF do seu proprietário e uma variável auxiliar para o cálculo do custo de instalação. Para mais compõe dois mapas, um para guardar *SmartDevice*'s que associa *id*'s de dispositivos às divisões a que pertencem, isto para evitar ter que procurar os dispositivos nas divisões, e outro que associa nomes de divisões a *Division*'s, para garantir acessos constantes a *Division*'s. Finalmente agrupa também os *Invoice*'s que para si foram emitidos.

```
private String name;
private String nif;
private List<Invoice> invoices;
private Map<String, String> devices;
private Map<String, Division> divisions;
```

Como auxiliar, também utiliza uma variável de instância que guarda os custos de instalação dos *SmartDevice*'s que são instalados. Após a emissão de uma fatura, este valor é lhe somado e seguidamente posto a zero. Essa emissão de faturas é pedida às *SmartHouse*'s pela *Community*. Para tal, a *SmartHouse* acede ao seu *Provider* que, por ser agregado, regista todas as mudanças que lhe são feitas ao longo da execução do programa.

Division

Esta classe simboliza uma divisão de uma casa em particular. Possui como variáveis de instância o seu nome e um mapa composto que relaciona *id*'s de dispositivos a *SmartDevice*'s, de forma a proporcionar procura de dispositivos de complexidade constante. Desta forma, os *SmartDevice*'s são responsabilidade das *Division*'s dentro de cada *SmartHouse*. Por conveniência, as divisões nunca têm um nome que comece por um número para não confundir com os *id*'s dos dispositivos.

```
private String name;
private Map<String, SmartDevice> devices;
```

SmartDevice

Classe abstrata que define o comportamento de um dispositivo genérico através da declaração de variáveis de instância e da declaração de métodos que todos os objetos do tipo abstrato *SmartDevice* devem implementar. Tais como *id*, informação sobre se se encontra ligado ou desligado, custo de instalação e consumo base para além de métodos como *turnOn/Off* e *getInstallationCost()*.

```
private final String id;
private boolean on;
private double installationCost;
private double baseConsumption;
```

As implementações que herdam de *SmartDevice* no programa são os pedidos no enunciado: *SmartBulb*, que tem tom e diâmetro; *SmartSpeaker*, que tem volume, marca e rádio a tocar; e *SmartCamera*, que tem resolução e dimensão. Adicionar um novo *SmartDevice* consiste em adicionar uma nova classe e depois funções para auxiliar a sua inserção ao longo de toda a estrutura do programa.

SerializableComparator<T>

Esta Interface serve para utilizar em vez de *Comparator<T>* como interface funcional para garantir que é possível escrever para ficheiro esses comparadores, se guardados.

```
public interface SerializableComparator<T>
extends Comparator<T>, Serializable
```

2.3 Exceptions

Para melhor detalhar falhas no programa, implementaram-se diversas *Exceptions* que são atiradas sempre que conveniente. O *Model* não trata nenhuma delas, deixando essa responsabilidade para quem invoca os seus métodos. As *Exceptions* implementadas são:

- NoProvidersException;
- NoHouseInPeriodException;
- AddressDoesntExistException; AddressAlreadyExistsException;
- ProviderDoesntExistException; ProviderAlreadyExistsException;
- DivisionDoesntExistException; DivisionAlreadyExistsException;
- DeviceDoesntExistException;

Capítulo 3

Controller

Síntese

A principal função do *Controller* é estabelecer uma ponte entre o *Model* (referido no capítulo anterior) e a *View* (explicada no capítulo seguinte). Deste modo, o *Controller* satisfaaz os pedidos da *View* através de chamadas ao *Model*, nomeadamente à classe *CommunityApp*. Este comportamento permite, posteriormente, facilitar mudanças na *View* sem afetar o *Model* e vice-versa.

3.1 Controller

Toda a semântica do *Controller* é feita nesta classe. Devido à *Facade* que *CommunityApp* representa, o *Controller* não conhece as estruturas do *Community*, o que potencia ainda mais a independênci mas faz com que a invocação dos métodos de *CommunityApp* tenham de ser com os tipos mais genéricos.

Para tal, a classe *Controller* guarda como variável de instância uma referência ao *Model*.

```
private CommunityApp model;
```

3.1.1 Parser

Uma das tarefas instruídas ao *Controller* foi realizar o *parse* do *input*, removendo essa lógica do *View*. O *parsing* é feito através de um método chamado *parser*. De modo a facilitar novas implementações de novos dispositivos e ou outras funcionalidades, decidimos não usar um switch e sim um mapa. Esse mapa é criado por um método chamado *createClassMap* que através de um array de strings, mapeia para cada string um método *create + nome da string*, esse método deve ser implementado no controller e garante a parse/criação de Strings que vão ser enviadas para o *Model*. Voltando ao método *parser*, o mesmo irá partir a linha do *input* em duas, ver o primeiro elemento que irá corresponder ao nome da classe, verificar se a mesma está mapeada e invocar o método *create* através de um método *invoke*.

Variáveis de instância

O *Controller* possui 4 variáveis de instância, o *Model* e outras 3 que têm como objetivo simplificar a criação das diversas estruturas nas funções *create*. A *addressGenerate* irá gerar um endereço a partir do zero e sempre que for acrescentada uma nova *SmartHouse* é incrementado por um. A *lastDivision* e a *lastAddress* servem para compor a lógica de adição de casas, pois quando queremos adicionar uma divisão ela precisa de um endereço de uma casa. O mesmo ocorre para os *smartDevices*, um *smartDevice* precisa de uma casa e de uma divisão para ser adicionado. Resolvemos adicionar pelas últimas de modo a criar uma estruturação semelhante à do log de texto.

```
private Integer addressGenerate;
private String lastDivision, lastAddress;
private CommunityApp model;
```

3.1.2 ParseActions

Outra tarefa do *Controller* é a de receber linhas de texto que contêm as ações automáticas e de decidir, para cada uma, qual a função do *Model* a invocar. Para tal, implementa a função *parseActions*. Para expandir as ações possíveis, as alterações apenas tinham de ser feitas nesta função. As ações automáticas implementadas são aquelas que alteram detalhes da simulação, nenhuma adiciona novos dispositivos.

Cada ação segue uma estrutura tal e qual a exemplificada no enunciado do Trabalho Prático. O diagrama seguinte emula a árvore para construir cada linha de ação, onde cada entrada é separada por uma vírgula seguida de um espaço. Em cada entrada, parênteses indicam o formato. Se não existirem, o método procura exatamente aquele texto.

```
data(string em formato yyyy.mm.dd),
moradaCasa(string),
idDispositivo(string),
setOn
setOff
alteraConsumoBase,
valorConsumo(double)
nomeFornecedor(string)
nomeDivisao(string),
divSetOn
divSetOff
nomeFornecedor(string),
alteraValorDesconto,
novoValor(double)
alteraAlgoritmo,
numeroAlgoritmo(int)
```

3.1.3 Métodos que fazem a ligação

Os restantes métodos permitem comunicar com o *Model*, como avançar ciclos, adicionar elementos à simulação, verificar aspectos sobre o *Model*, entre outros.

Capítulo 4

View

4.1 Síntese

A *View* corresponde ao nível de interação com o utilizador com o auxilio de menus, que permitem realizar operações IO como imprimir entidades do programa ou ler um *log* de texto/objeto de modo a facilitar a adição manual caso queiramos algo mais extenso. A mesma comunica com o *Controller* quando quer requisitar algo do *Model*. É importante salientar que a *View* é a única capaz de realizar input/output, inclusive para erros.

4.2 Menu

A classe *Menu* auxilia a *View* proporcionando uma forma de montar instâncias de menus com opções e com métodos que implementam cada uma das opções. Para esse efeito, cada *Menu* tem um nome; uma lista das opções de texto; uma lista de instâncias que implementam a interface *Handler*; e uma lista de instâncias que implementam a interface *PreCondition*. As listas estão coordenadas: cada posição nas entradas das opções refere-se exatamente à mesma posição na lista de *Handler*'s e na lista de *PreCondition*'s. Esta abordagem é inspirada numa solução apresentada nas aulas teóricas.

```
private static Scanner is = new Scanner(System.in);
private String name;
private List<String> options;
private List<Handler> handlers;
private List<PreCondition> preConditions;
```

A utilização de cada *Menu*, após a sua criação, faz-se a partir do método *execute*, que o imprime, recebe as opções do utilizador e depois executa o que tem como definido para cada uma das opções escolhidas.

Esta solução permite modularidade na criação de *Menu*'s e também a possibilidade

4.2.1 Handler

A interface *Handler* é utilizada como uma interface funcional, usada pelo método *execute*. O método recebe uma lista de *String* como argumentos, normalmente utilizado para referenciar moradas ou nomes de fornecedores. O método também devolve um *int* que é utilizado na *View* para saber se o menu se deve repetir ou terminar.

```
public interface Handler extends Serializable {  
    public int execute(List<String> args);  
}
```

4.2.2 PreCondition

A interface *PreCondition* é utilizada como uma interface funcional pelo método *validate*. O método devolve um *Boolean*, utilizado para invalidar certas opções se não for possível a sua utilização. Esta vertente não é extensivamente explorada, preferindo-se ao invés a validação ao longo do *Handler*.

```
public interface PreCondition extends Serializable {  
    public boolean validate();  
}
```

4.3 View

Esta classe implementa a semântica da *View*. Para tal agrega um mapa que relaciona o nome de *Menu* com o próprio *Menu* e compõe um *Controller*. O construtor da *View* adiciona os *Menu*'s todos. Isto podia ser feito ou de fora, da *Main* por exemplo, ou lidos de um ficheiro. A *View* éposta em execução a partir do método *run*, que recebe uma data inicial para a execução e depois executa o *Menu startMenu*.

4.3.1 Menus da View

A *View* implementa os seus próprios *Menu*'s em métodos seus. Para os colocar em execução, é necessário saber qual o seu nome e aceder ao mapa de *Menu*'s. De seguida, estão os *Menu*'s

StartMenu

Menu inicial do programa. Serve para poder carregar estados iniciais.

SimulationMenu

Menu principal da simulação. Pode-se carregar ações automáticas, aceder ao menu de alterar detalhes, avançar dias, aceder ao menu de impressão ou gravar o estado num ficheiro binário.

AutomaticSimulationMenu

Menu de controlo da simulação automática. Permite avançar a simulação e aceder ao menu de impressão.

PrintMenu

Menu de impressão do estado do programa. Contém os métodos pedidos no enunciado sobre o estado atual do programa, como o Fornecedor com maior volume de faturaçāo ou a Ordenação dos maiores consumidores de energia.

AlterSimulationDetailsMenu

Menu para alterar os detalhes da simulação. Acrescentar elementos é uma ação instantânea enquanto que alterar detalhes de elementos já existentes só se efetua no próximo ciclo de faturaçāo.

AlterSimulationDetailsHouseMenu

Menu específico para alterar os detalhes de uma casa.

AlterSimulationDetailsProviderMenu

Menu específico para alterar os detalhes de um fornecedor.

AlterProviderAlgorithmMenu

Menu específico para escolher, de entre os algoritmos implementados, qual deles o fornecedor deve utilizar.

AddSmartDeviceMenu

Menu específico para adicionar, de entre os dispositivos implementados, qual deles adicionar à simulação.

Capítulo 5

Main

A classe *Main* é o ponto de execução do programa. Esta classe cria uma instância de cada uma das partes da estrutura: *CommunityApp* para o *Model*; *Controller* para o *Controller*, que recebe o anterior; *View* para a *View*, que recebe o anterior. Para dar início, executa o método *run* da *View*.

Capítulo 6

Apresentação

Para a apresentação, temos 4 ficheiros específicos: "*apresentacao.obj*" é o ficheiro de objetos obtido depois de aplicar as ações automáticas em "*prebuilt.txt*" ao estado inicial em "*log.txt*", o ficheiro de início fornecido junto com o enunciado. "*automatic.txt*" é um ficheiro de ações automáticas para utilizar durante a apresentação.

Capítulo 7

Diagrama de Classes

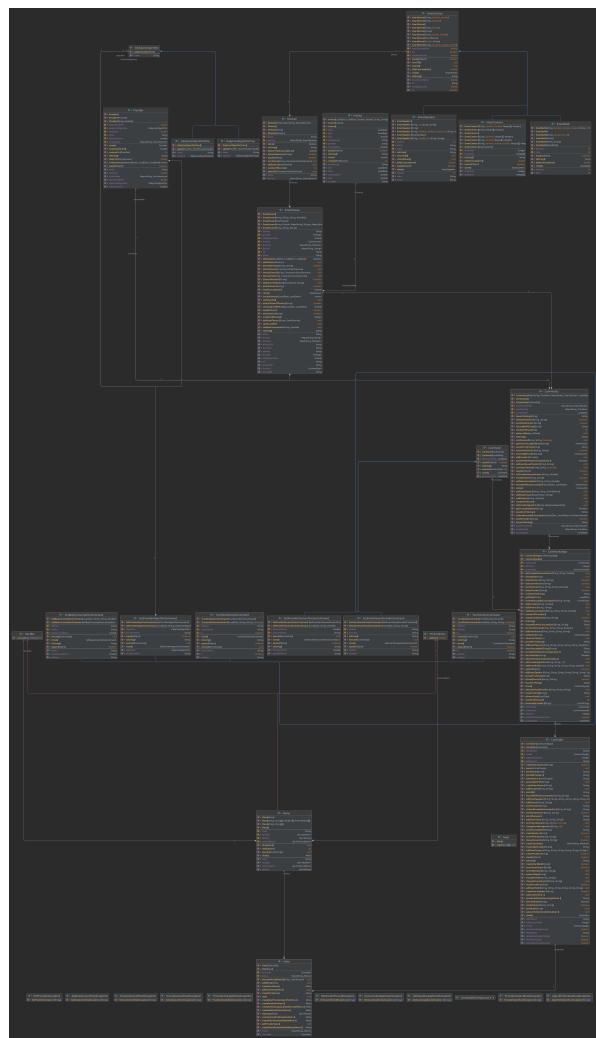


Figura 7.1: Diagrama de classes do programa, gerado pelo *IntelliJ*

Capítulo 8

Conclusão

Em suma, tendo em consideração a estrutura explicitada anteriormente, consideramos que fomos capazes de cumprir todos os objetivos que nos foram apresentados de uma forma modular e eficiente, não perdendo, no entanto a capacidade de expandir o projeto caso necessário. Cremos que respondemos ao desafio de implementar todas as facetas da aplicação pedida de forma sensata e cumprindo com as regras e convenções que advém do paradigma de programação orientada a objetos e que viemos, ao longo do semestre a aletradar.