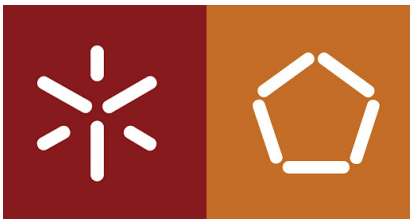

Universidade do Minho

Licenciatura em Engenharia Informática

Programação Orientada aos Objectos
Projecto prático
Grupo 16

Rodrigo Rodrigues (A93201) Rui Monteiro (A93179)
Gonçalo Moreira (A73591)

Ano Letivo 2021/2022

1 Introdução

Este trabalho prático surge no âmbito da Unidade Curricular de Programação Orientada aos Objetos perante a proposta de construir um sistema que monitorize e registe a informação sobre o consumo energético das habitações de uma comunidade.

Este relatório serve o propósito de rever detalhadamente a nossa implementação de software para o sistema proposto segundo um Paradigma Orientado aos Objetos e da linguagem de programação JAVA, fazendo recurso à abstração e hierarquia de Classes para atingir código modular e reutilizável.

Nas seguintes secções serão abordados tópicos como as estruturas de dados escolhidas para armazenar as diferentes informações necessárias ao funcionamento do programa, as razões para essas escolhas, e as estratégias seguidas para elaborar a aplicação.

2 Arquitetura

A aplicação foi desenvolvida tendo em conta o modelo de desenvolvimento MVC (Model - View - Controller). Desta forma, é possível distinguir três pilares fulcrais neste trabalho: o *Model* que agrega todas as classes que são utilizadas no armazenamneto das estruturas de dados, e a parte algorítmica do programa; a *View* que efetua todas as operações correspondentes à comunicação do utilizador; o *Controller* serve como um meio de comunicação entre *Model* e *View*, na medida em que o fluxo tem de passar sempre primeiro pelo *Controller*.

Para promover reutilização de código, procuramos criar hierarquia de classes, definindo a classe abstrata *SmartDevice* cujas variáveis de instância são partilhadas pelos diferentes tipos de dispositivos (classes *SmartBulb*, *SmartSpeaker*, *SmartCamera*).

3 Classes

3.1 Model

3.1.1 SmartDevice

```
private String id;
private boolean on;
private LocalDateTime timeOfTurningOn;
private double installationCost;

public SmartDevice () {
    this.id = UUID.randomUUID().toString();
    Random random = new Random();
    this.on = random.nextBoolean();;
    this.timeOfTurningOn = LocalDateTime.now();
    this.installationCost = 4.99;
}
```

As variáveis de instância desta classes são partilhadas pelos restantes dispositivos inteligentes do sistema (lâmpadas, colunas, cameras). Esta classe é abstrata pelo que não se permite a criação de instâncias de *SmartDevice*, apenas de *SmartBulb*, *SmartSpeaker* e *SmartCamera*.

- **id**: identificador de um dispositivo
- **on**: estado do dispositivo (on/off)
- **timeOfTurningOn**: instante em que o dispositivo foi ligado
- **installationCost**: custo de instalação do dispositivo

Quando um dispositivo é utilizado o construtor por omissão para criar uma nova instância de um dispositivo, o estado **ON** é calculado por uma função *random*, visto que o utilizador não o indicou. Por sua vez, o instante em que o dispositivo foi ligado é definido para o instante de criação e o custo padrão de instalação é de 4.99 u.m..

3.1.2 SmartBulb

```
private int tone;
private int diameter;
private double consumption;
```

- **tone**: tonalidade da lâmpada (WARM, NEUTRAL, COLD)
- **diameter**: diametro da lâmpada
- **consumption**: consumo diário do dispositivo

```
public double determineConsumption(){
    double multiplier = 1.0;

    switch (this.tone){
        case NEUTRAL:
            multiplier = 1.0;
            break;
        case COLD:
            multiplier = 1.11;
            break;
        case WARM:
            multiplier = 1.07;
            break;
    }
    return (consumption * multiplier);
}
```

3.1.3 SmartSpeaker

```
private String radio;
private String brand;
private double volume;
private double consumption;
```

- **radio**: canal que está a tocar
- **brand**: marca das colunas
- **volume**: valor do volume
- **consumption**: consumo diário das colunas

```
public double determineConsumption() {
    double multiplier = 1.0;

    if(this.volume > 10 && this.volume < 50) {
        multiplier = 1.05;
    } else {
        multiplier = 1.09;
    }

    return (this.consumption * multiplier);
}
```

3.1.4 SmartCamera

```
private int resolutionX;
private int resolutionY;
private double fileSize;
private double consumption;
```

- **resolutionX / resolutionY**: valores (x,y) da resolução da camera
- **fileSize**: tamanho dos ficheiros gerados
- **consumption**: consumo diário da camera

```
public double determineConsumption(){
    return (this.fileSize / 1000.0) * ((this.resolutionY + this.resolutionY) /
        1000.0);
}
```

3.1.5 Request

Como referido no enunciado, existem certos pedidos que podem ser feitos mas que apenas devem ter efeito na ronda seguinte de simulação, isto é, quando se faz avançar o tempo. Exemplos disso são:

1. Casa solicita mudança de fornecedor de energia
2. Casa opta por ligar e desligar dispositivos
3. Fornecedores alteram os valores

Desta forma, foi criada a classe *Request* que contém informação acerca do pedido em causa, tendo um construtor para cada um dos pedidos mencionados acima. Quando um utilizador faz um pedido, é criada uma instância de *Request* com o devido identificado do tipo de pedido e este é adicionado a uma lista de pedidos em espera que serão processados após se fazer avançar a data (terão efeito na ronda de simulação seguinte).

```
private String type;
private int nif;
private double tax;
private double baseValue;
private String oldSupplier;
private String newSupplier;
private List<SmartDevice> devices;
```

3.1.6 Invoice

Esta classe contém informação acerca de uma fatura de energia, tal como o nome indica.

```
private LocalDateTime start;
private LocalDateTime end;
private int NIF;
private String houseOwner;
private String supplier;
private double consumption;
private double cost;
```

3.1.7 Model

Nesta classe agrupamos e armazenamos todos os itens necessários para o funcionamento do sistema. O *Model* funciona como uma espécie de base de dados onde se guarda informação sobre as casas, respetivas divisões e dispositivos, os fornecedores, os dispositivos que foram criados mas ainda não foram adicionados a uma casa, os pedidos feitos pelos utilizadores que ficam em espera até à ronda de simulação seguinte e, por fim, as datas das rondas de simulação efetuadas até ao momento.

```
private Map<String, SmartDevice> devices;
private Map<Integer, SmartHouse> houses;
private Map<String, Supplier> suppliers;
private List<SmartDevice> freeDevices;
private List<Request> requests;
private List<LocalDateTime> dates;
```

Desta forma, quando se guarda o estado do programa através da opção **SAVE** do menu do utilizador, é possível guardar informação sobre todas as simulações que decorreram anteriormente, bem como as suas estatísticas, as casas/dispositivos/fornecedores presentes no sistema, e ainda a data em que o programa estava quando foi guardado, podendo então partir dessa data.

4 Controller

No *Controller*, é feita a mediação entre os pedidos do utilizador e o *Model*.

5 View

Na classe da *View* encontram-se métodos para exibição de menus e/ou informação relativa a queries feitas pelo utilizador.

6 Manual de utilização e Funcionalidades do programa

Ao ser inicializada a aplicação é apresentado o menu principal que oferece navegação baseada nas linhas de comando e permite ao utilizador executar diversas funcionalidades.

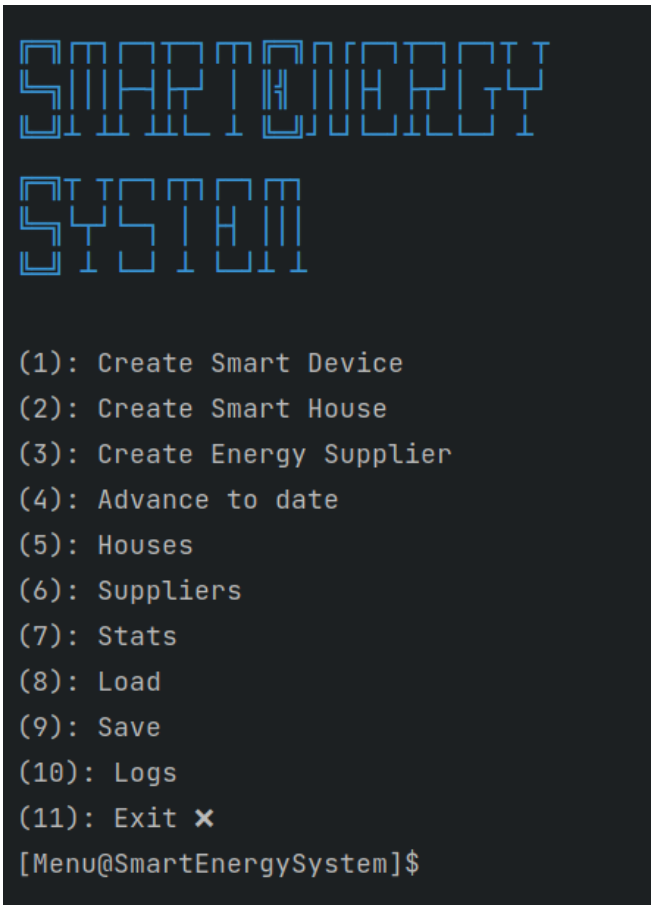

Figura 1: Menu inicial

6.1 Criação de dispositivos, casas e fornecedores

As três primeiras opções são relativas à criação de dispositivos, casas e fornecedores de energia conforme os parâmetros escolhidos pelo utilizador.

6.2 Avançar o tempo (realizar simulação)

O utilizador pode solicitar ao programa que avance no tempo para uma determinada data futura. O programa avança realizando a simulação. No final de cada simulação, os fornecedores emitem as faturas aos seus clientes com os valores de consumo e o respetivo preço. Estas faturas são exibidas ao utilizador através de um conjunto de páginas, sendo possível avançar/recuar/saltar páginas.

Abaixo segue-se um exemplo da página 11 de um livro de faturas resultantes de uma simulação.

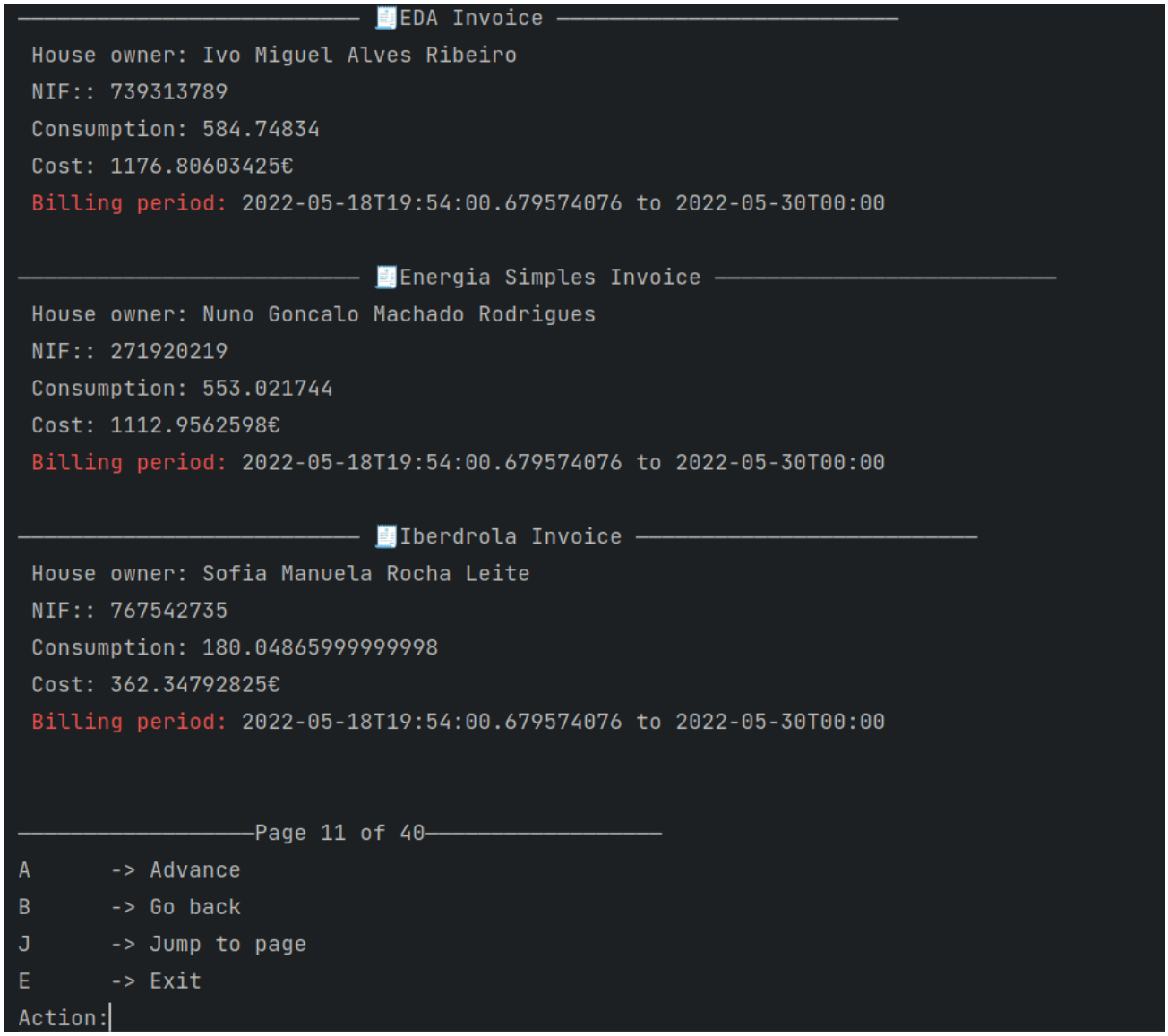

Figura 2: Faturas emitidas no final de uma simulação

Adicionalmente, no final da exibição das faturas é apresentado no ecrã uma estatística relativa à **casa que teve mais custos** na ronda de simulação em causa. Finalmente, o programa informa que os **pedidos** que foram feitos pelo utilizador (ligar/desligar dispositivos) ou por algum fornecedor(mudar preços), estão a ser **processados** e terão efeito na próxima ronda de simulação. Por exemplo, se um utilizador requisitou a mudança de fornecedor (EDP -> Enat), na próxima faturação já iremos ver uma fatura da Enat referente à sua casa com os valores atualizados.

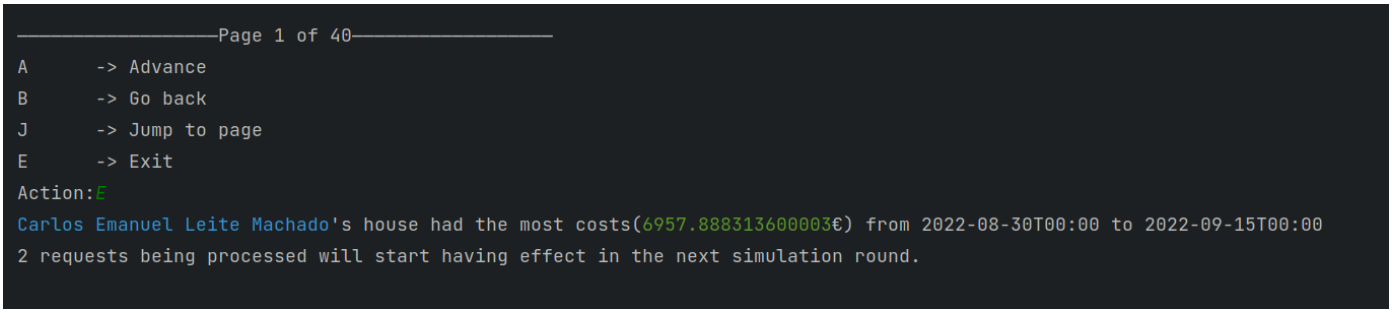

Figura 3: Estatística de casa com mais custos e número de pedidos a serem processados

6.3 Casas

Ao selecionar a opção 5 (**Houses**) é possível obter uma paginação de todas as casas existentes com alguma informação básica de cada uma (nome e NIF do dono e fornecedor).

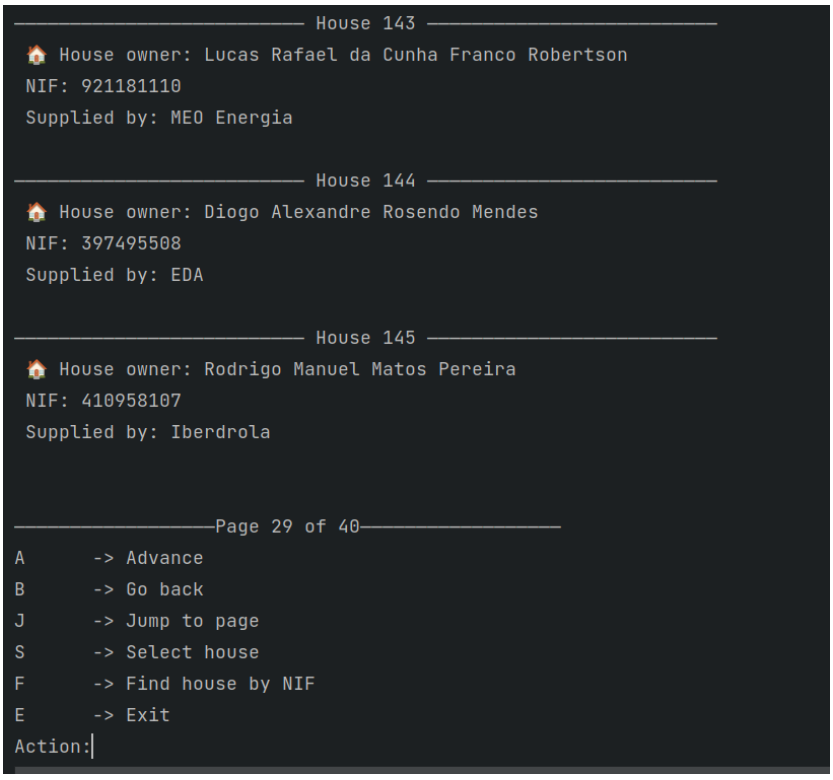

Figura 4: Paginação das casas

Nesta paginação é possível selecionar casas pelo seu número na página ou pelo seu próprio NIF. Eis as opções possíveis a realizar sobre uma casa inteligente:

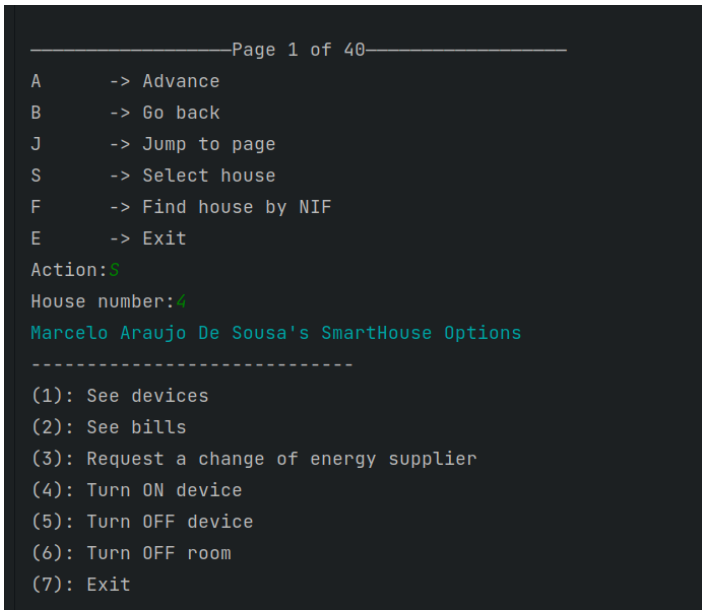

Figura 5: Opções possíveis sobre uma casa

Por exemplo, é possível ver as divisões da casa e respetivos dispositivos.

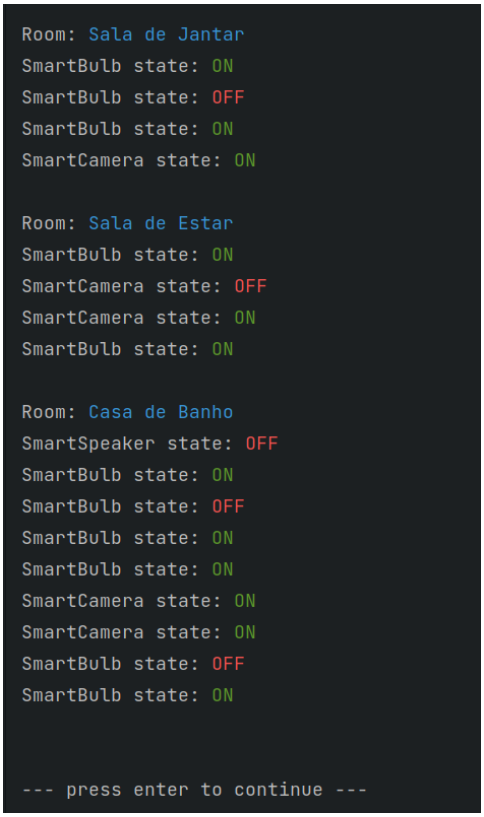

Figura 6: Ver dispositivos de uma casa

Exemplo de pedido para ligar um dispositivo da casa.

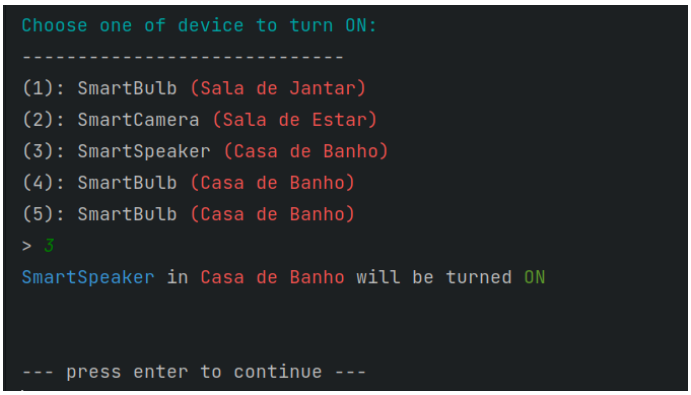

Figura 7: Pedido para ligar um dispositivo.

6.4 Fornecedores

À semelhança das casas, também se pode consultar os fornecedores disponíveis no sistema e realizar operações sobre os mesmos.

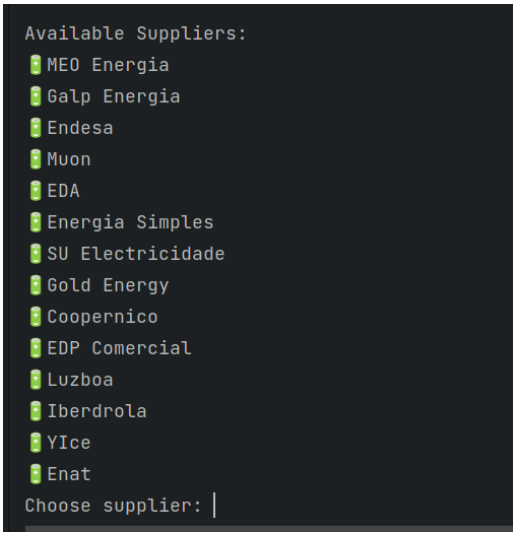

Figura 8: Lista de fornecedores disponíveis.

É possível consultar os clientes dos fornecedores, bem como requistar a mudança dos valores (preços/impostos).

6.5 Stats

Na secção de estatísticas, é possível:

1. Qual o comercializador com maior volume de facturação

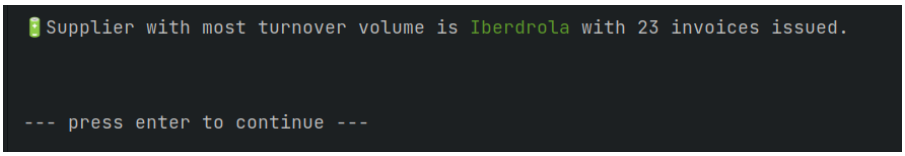

Figura 9: Estatística de maior volume de faturação.

2. Listar as faturas emitidas por um comercializador
3. Dar uma ordenação dos maiores consumidores de energia durante um periodo a determinar

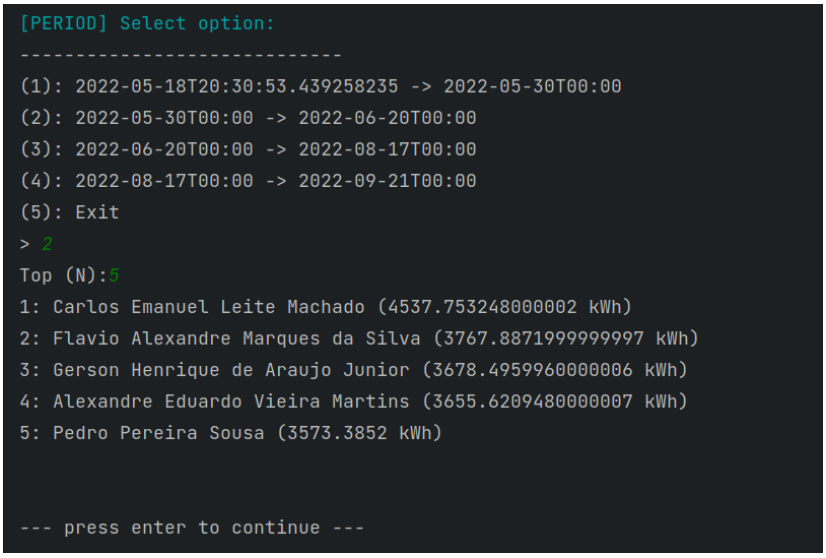

Figura 10: Ordenação dos maiores consumidores

6.6 Gravar/Recuperar Estado do programa

A opção 8 (**Save**) permite efetuar a gravação do estado do programa e informação relevante das entidades e a opção 9 (**Load**) permite recuperar o estado que foi gravado.

6.7 Carregar ficheiro de logs

A opção 10 (**Logs**) permite efetuar a leitura e carregamento da informação presente nos ficheiros de logs.

7 Diagrama de Classes

Na figura abaixo apresenta-se o diagrama de classes do *Model*, também em anexo com alta resolução.

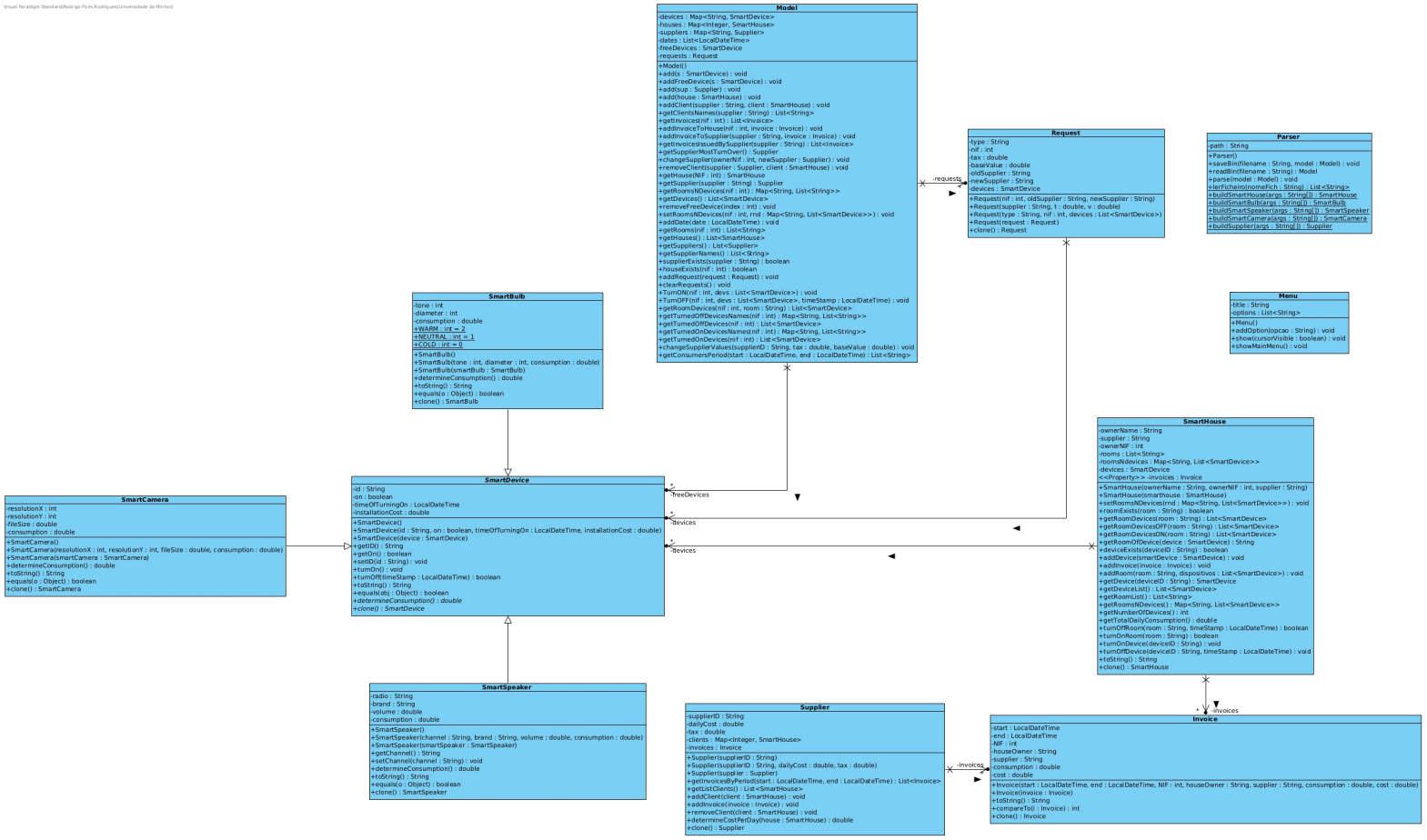

Figura 11: Diagrama de classes do *Model*

8 Conclusão

Este projeto tinha como objetivos principais a aplicação de conceitos lecionados na Unidade Curricular de Programação Orientada aos Objetos, nomeadamente Modularidade, encapsulamento de dados, reutilização de código, boas práticas e MVC. A utilização de modularidade permite a introdução de novas entidades e funcionalidades no sistema com relativa facilidade, e a utilização de clones permite o encapsulamento de dados no sistema. Por sua vez, o modelo MVC permite um fluxo organizado entre Utilizador e Base de Dados.
Todos os os requisitos necessários para a resolução foram correspondidos.