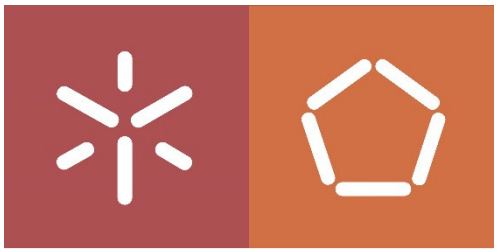

Universidade do Minho
Escola de Engenharia

Projeto prático de Programação Orientada aos Objetos

Trabalho realizado por:

Grupo 36

Ana Filipa Cruz Pinto - 96862
Joel Vieira Barros - 94692
Sandro Duarte - 94731

-INTRODUÇÃO-

Como parte da componente experimental da unidade curricular Programação Orientada aos Objetos, foi proposta a criação de um sistema que monitorize e registe a informação sobre o consumo energético das habitações de uma comunidade. O presente relatório visa sobre a elaboração desse trabalho.

De forma a criar o sistema pedido, foi-nos dada a informação que, em cada casa, existem um conjunto muito alargado de dispositivos. Estes, são todos controlados a partir do programa que precisa ser criado. Cada dispositivo é inteligente, no sentido que será possível ligá-lo e desligá-lo, e permite registar o seu consumo energético. Chamemos, por simplificação, a estes dispositivos de *SmartDevice*.

Cada um destes *SmartDevices* é identificado por um código único, definido pelo fabricante. Além do código, cada *SmartDevice* tem também um custo de instalação que é pago, independentemente da sua utilização ou não. Adicionalmente, é sabido que cada casa possui vários *SmartDevices*, que podem ser agrupados por divisão. Por exemplo, na sala de estar podemos ter uma coleção de dispositivos que queremos monitorizar e controlar, sendo que um utilizador poder ligar ou desligar todos os dispositivos de uma divisão da casa. Os *SmartDevice* disponíveis são lâmpadas, colunas e câmaras inteligentes.

Numa segunda dimensão, cada casa tem, num determinado momento, um contrato com uma empresa comercializadora de energia. Como tal, existem, no mercado, diversos fornecedores de energia e cada um deles tem uma definição de preço a que vende a energia, de acordo com os seus critérios comerciais. Com isto, queremos dizer que, no contexto deste exercício, não podem existir casas sem um fornecedor de energia associado.

Assim, o principal objetivo deste trabalho é a criação de um programa que gere uma espécie de simulação do valor de consumo de energia de várias casas, pondo em prática o conteúdo lecionado durante a unidade curricular. De notar que este trabalho será elaborado em Java.

-CLASSE SMARTDEVICE-

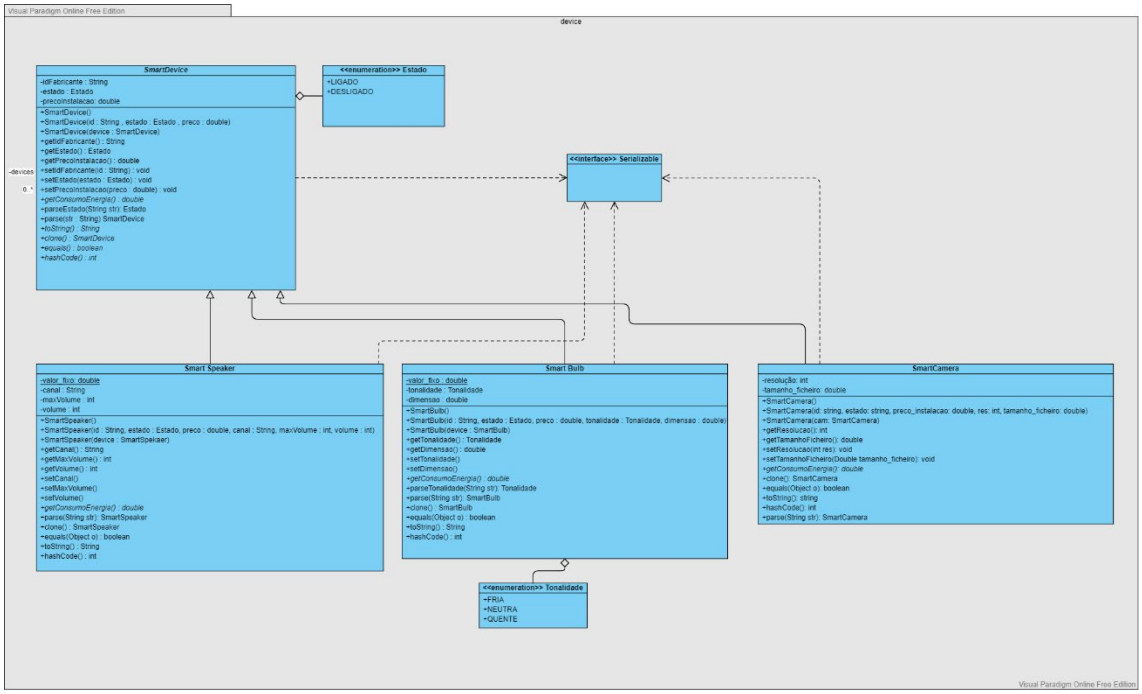

Figura 1. Esboço UML da classe SmartDevice

Começamos por desenhar uma hierarquia de classes, contendo os três tipos de dispositivos que as casas podem conter.

A superclasse ***SmartDevice*** foi concebida de modo a obtermos uma maior abstração, de forma a conseguir compatibilidade entre os tipos diferentes de aparelhos. Foram, assim, criadas coleções, agregando os diversos dispositivos. A classe foi considerada abstrata, pois o método ***getConsumoEnergia*** é algo comum entre as três subclasses, mas cada uma delas implementa-a de maneira diferente.

Outros métodos como ***toString***, ***equals***, ***clone*** e ***hashCode*** foram considerados ***abstract,*** de modo a forçar as subclasses a implementá-los. Estes métodos são de extrema importância, uma vez que são utilizados pelas coleções. Caso não sejam definidos para cada caso específico, os métodos utilizados são os herdados de ***Object***, sendo que estes não garantem a funcionalidade desejada.

Por sua vez, as classes ***SmartSpeaker***, ***SmartBulb*** e ***SmartCamera*** são subclasses de ***SmartDevice,*** pois são uma especialização da superclasse.

Note-se que todas as classes, referidas até ao momento, implementam a interface ***Serializable*** para poderem ser persistidas em memória.

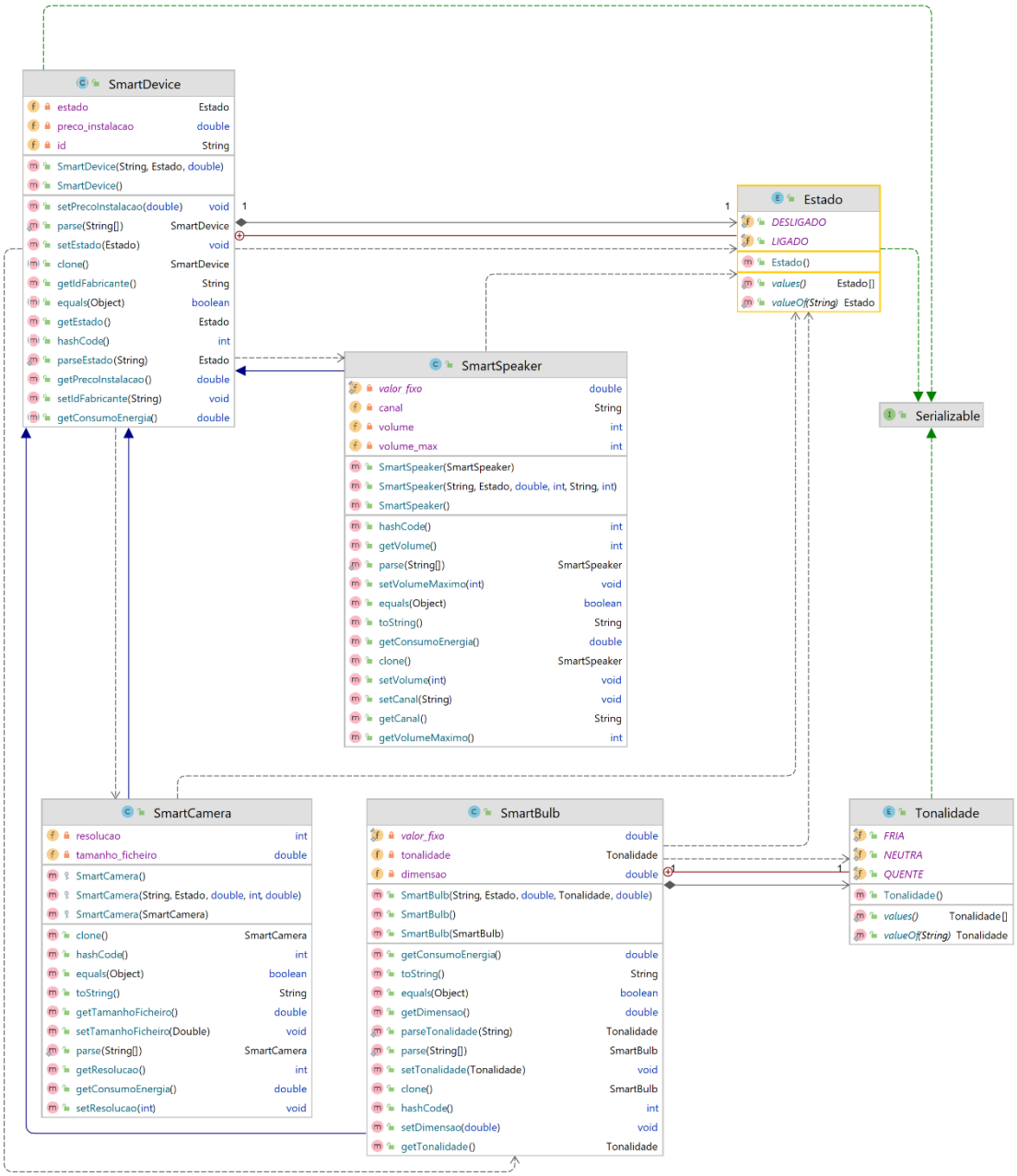

Figura 2. UML SmartDevice final

Todas as classes contam com métodos ***get*** e ***set*** para cada uma das suas variáveis de instância, de modo que estas possam ser alteradas. Além disso, as classes também têm métodos de classe estáticos, nomeadamente ***parse***, de forma a conseguir obter uma instância de cada objeto através de uma linha de texto devidamente formatada.

De modo a obedecer ao enunciado proposto, foi utilizada uma variável de classe ***valor_fixo*** na ***SmartBulb*** e ***SmartSpeaker*** .

-CLASSE CASA E DIVISAO-

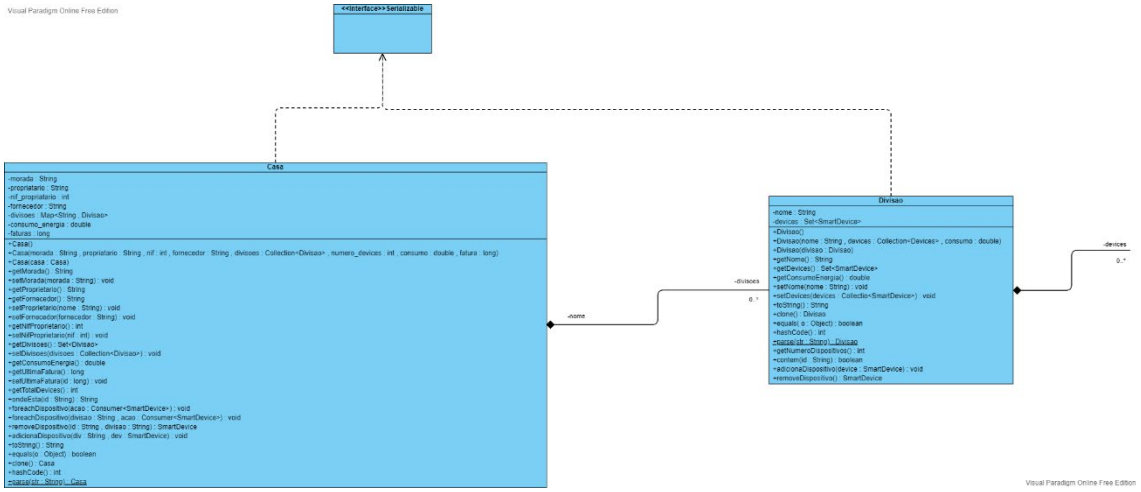

Figura 3. Esboço da classe Divisao e Casa

De modo a agrupar dispositivos por divisão, criamos uma classe chamada ***Divisao***. Esta será responsável pelos aparelhos de uma determinada área da casa. Qualquer alteração a ser feita num objeto nela contida terá de passar pela ***API*** da mesma. Assim, cada ***Divisao*** utiliza 0 ou mais ***SmartDevice***, por composição, agrupados num conjunto. De notar que a ordem neste contexto não é relevante.

A classe ***Casa*** guarda um mapeamento de divisões indexada pelo nome de cada zona, sendo que estas também são utilizadas por composição. Isto é garantido utilizando uma ***deep copy***, através do método ***clone***, de cada um dos elementos aquando da sua adição ou consulta. A única altura em que uma cópia não é devolvida, é na remoção de um dispositivo, pois este é "apagado" e já não pertence à instância da divisão.

Utilizamos este método de remoção, de forma a atualizar os dispositivos. Removemos, alteramos e depois voltamos a inserir os dispositivos, garantindo assim o encapsulamento dos dados. Para além disso, as classes implementam ***Serializable*** , para persistirem na memória.

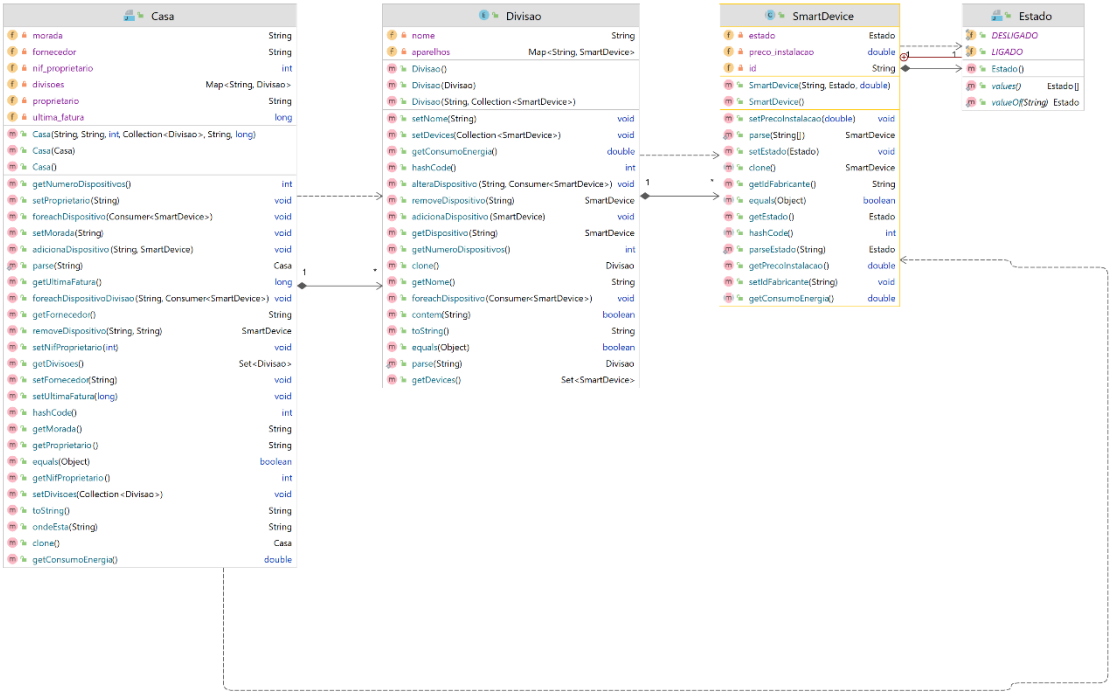

Figura 4. UML final Casa e Divisao

Em seguida, foi adicionada uma variável de instância, ***ultima_fatura***, de modo a facilitar a implementação da *query* para encontrar a casa com maior despesa.

-CLASSE FORNECEDORENERGIA E FATURA-

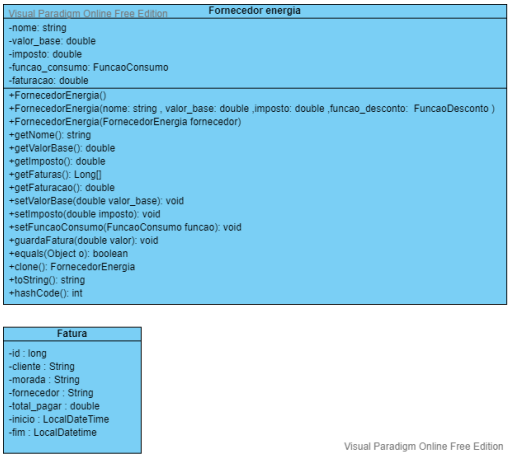

Figura 5. Esboço UML FornecedorEnergia e Fatura

As classes ***FornecedorEnergia*** e ***Fatura*** regem-se pela mesma lógica das anteriores. Aqui, o destaque vai para a variável de instância ***funcao_consumo***. Esta possibilita que cada fornecedor calcule o valor a pagar de cada casa.

Criamos, para isso, uma classe chamada ***FuncaoConsumo*** com um único método ***calculaTotalPagar***, que recebe como parâmetros uma ***Casa*** e uma ***FornecedorEnergia***. Assim, ao calcular o valor, podemos utilizar todas as informações disponíveis de cada casa e fornecedor. Podemos ainda ter cálculos que utilizam número de dispositivos para aplicar descontos, condições sobre consumos totais, etc. Cada fornecedor define a função que deseja.

No entanto, o entrave aqui é o facto de termos de criar uma classe para cada tipo de função. A nossa opção de resposta centrou-se apenas na criação da ***FuncaoPadrao*** para mostrar este mecanismo.

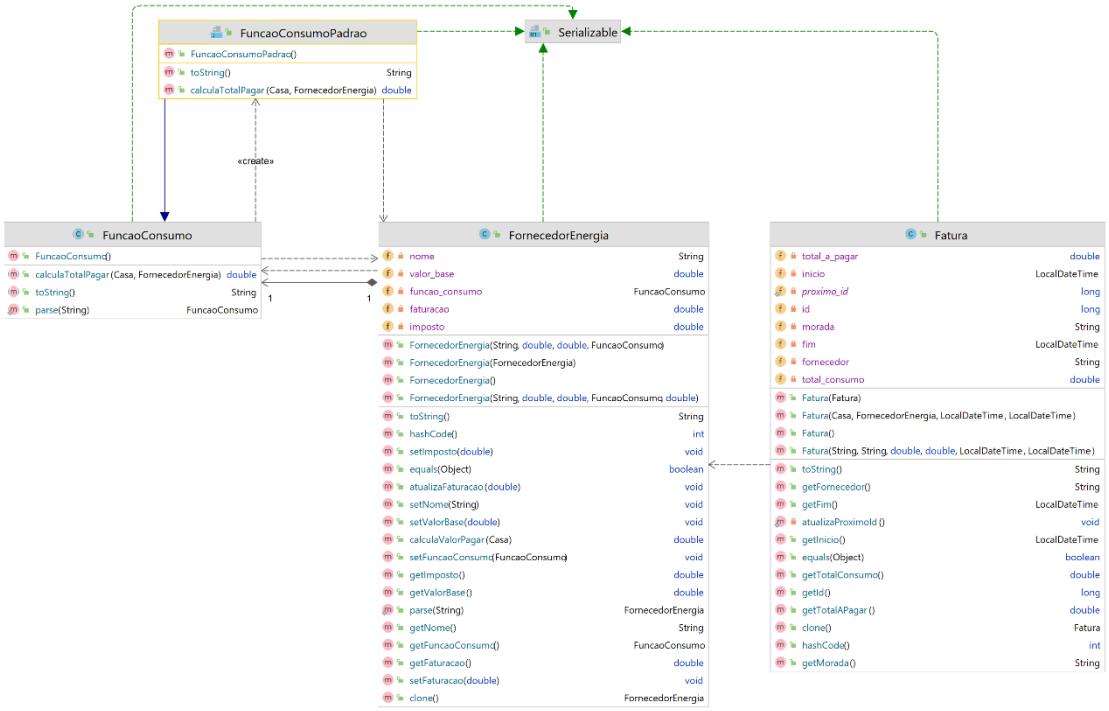

Figura 6. UML final Fornecedor, FuncaoConsumo e Fatura

A classe ***Fatura*** serve para armazenar informação sobre o consumo de uma casa. Essa classe usa um construtor que recebe uma ***Casa***, ***FornecedorEnergia***, data de início e fim de período de faturação, para que possa ser criada com as informações corretas.

-MVC-

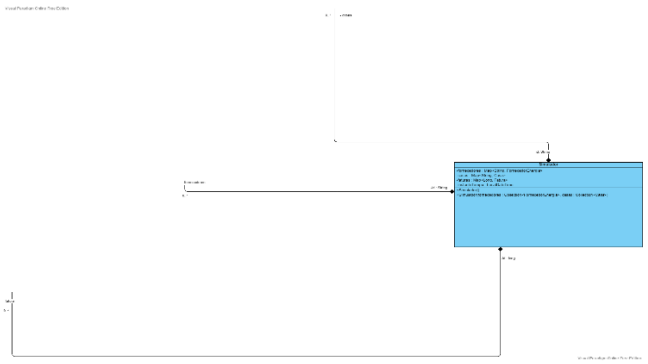

Figura 7. Esboço MVC UML

Decidimos, pois, utilizar o método ***MVC*** para podermos ter uma melhor organização no nosso trabalho prático, de forma a fomentar boas práticas de programação orientada a objetos e garantir que as regras base (encapsulamento, polimorfismo, integridade de dados, etc.) sejam cumpridas. Desta forma, o modelo vai guardar a implementação de todos os tipos de dados a ser usados pela aplicação. Por sua vez, o controlador garante que o modelo seja manipulado de forma correta, assegurando a resposta desejada, e a vista interage com o utilizador, mostrando os resultados da utilização da aplicação.

-MODELO-

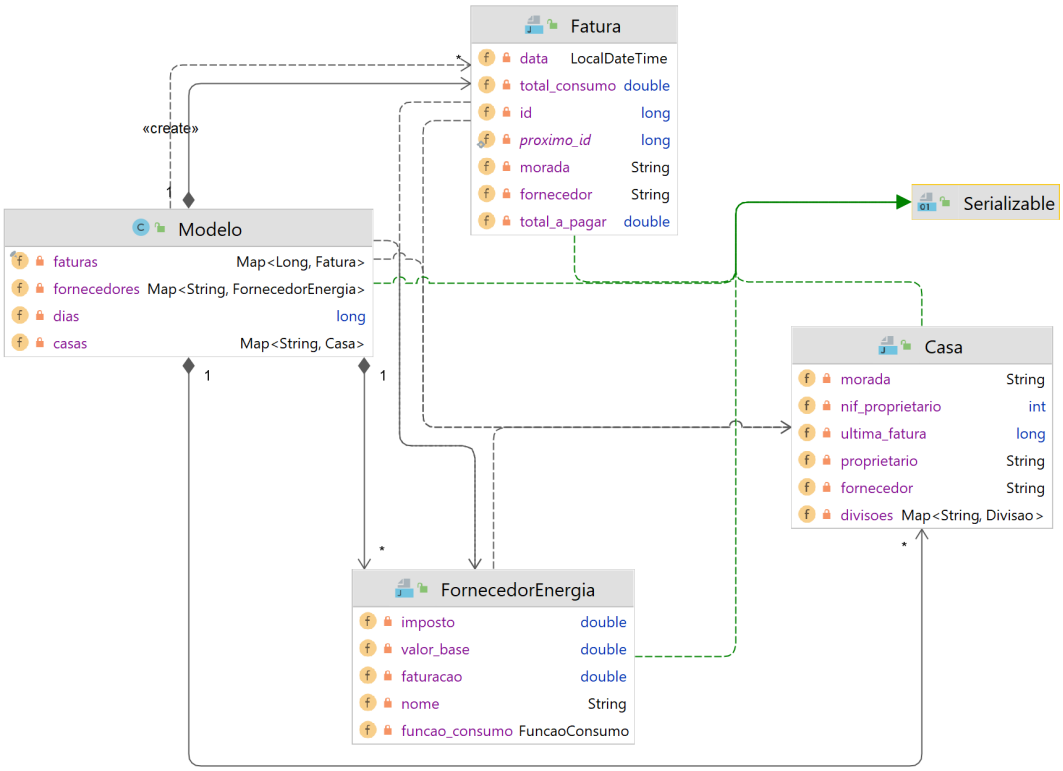

Figura 8. UML final do Modelo

No modelo são guardadas todas as instâncias a ser criadas em mapeamentos da natureza identificador e objeto. No caso dos fornecedores, o identificador é o seu nome. Nas casas, é a sua morada. Por fim, nas faturas é o numérico de fatura.

Aqui, tentamos que os métodos fossem os mais abstratos possíveis, garantindo o polimorfismo e a reutilização do código.

-CONTROLADOR-

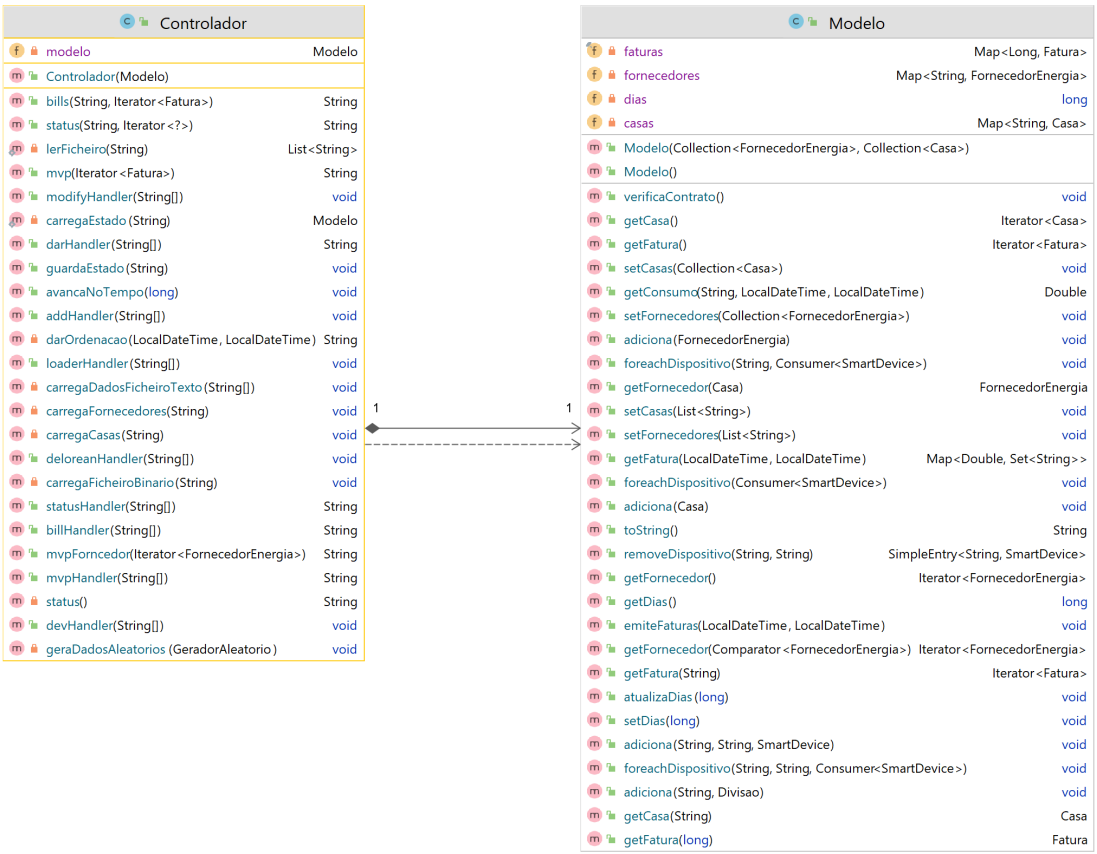

Figura 9. UML final do Controlador

Sendo o ***Modelo*** apenas a camada computacional, chegou a vez de criar o ***Controlador*** que vai atribuir carga semântica aos métodos. Por exemplo, ligar e desligar todos os aparelhos utilizam o mesmo método do ***Modelo***, ***foreachDispositivo***, mas com argumentos diferentes. Este tipo de reutilização acontece frequentemente e, por diversas razões, não foi explorado ao máximo. Existem, assim, algumas repetições de código desnecessárias como ***foreachDispositivo*** e ***foreachDispositivoDivisao*** que certamente poderiam ser uma única função.

-VIEW-

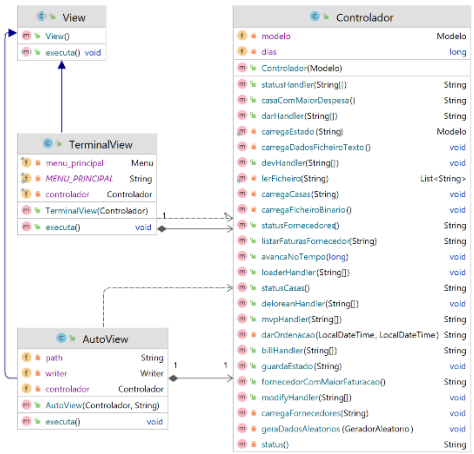

Figura 10. UML final da View

Foi considerado um modelo hierárquico para a ***`View`***, de forma a conseguir ter duas vistas distintas: Uma para ser a interação gráfica e outra para ser o modo autónomo, tal como proposto no enunciado. A variável ***`controlador`*** foi repetida em cada classe, para não existir a necessidade de clonar a instância sempre que a quiséssemos aceder.

-UTILIZAÇÃO DA APLICAÇÃO-

```
   <<<Sir Manuel>>>
-------------------------------------------------
| ld -bin <path>                : Carregar dados (ficheiro binário)
| ld -txt <forncedores> <casas> : Carregar dados (ficheiros de texto)
| ld -rand                    : Carregar com dados aleatórios
| mod -allon                  : Ligar todos os dispositivos
| mod -allof                  : Desligar todos os dispositivos
| mod -divon <casa> <divisao> : Ligar todos os aparelhos de uma divisão de uma casa
| mod -divof <casa> <divisao> : Desligar todos os aparelhos de uma divisão de uma casa
| dev -on <id>                : Liga um dispositivo
| dev -off <id>               : Desliga um dispositivo
| delorean <n>                : Regressa <n> dias ao futuro
| add -f <nome> <preco_base> <imposto> <funcao> : Adiciona um fornecedor
| add -h <morada> <prop> <nif> <fornecedor> : Adiciona uma Casa
| add -dev <morada> <divisao> <str_formatada> : Adiciona um SmartDevice
| add -div <morada> <str_formatada : Adiciona uma Divisao
| stat -all                   : Apresenta o estado do simulador
| stat -h                     : Apresenta o estado das casas
| stat -f                     : Apresenta o estado dos fornecedores
| mvp -h                      : Apresenta a casa com maior consumo
| mvp -f                      : Apresenta o fornecedor com maior faturação
| bills <fornecedor>                      : Listar faturas do fornecedor
| dar <data inicio> <data fim>            : "Dar" uma ordenação dos maiores consumidores
| dev -on <casa> <id>                     : Liga um dispositivo
| dev -off <casa> <id>                    : Desliga um dispositivo
| dev -price <casa> <id> <preco>          : Muda o preco de instalação
| dev -tone <casa> <id> <tone>            : Muda a tonalidade lâmpada
| dev -dim <casa> <id> <dim>              : Muda a dimensão da lâmpada
| dev -res <casa> <id> <res>              : Muda a resolução da câmara
| dev -file <casa> <id> <tamanho>         : Muda o tamanho do ficheiro
| dev -vol <casa> <id> <vol>              : Muda o volume da coluna
| dev -max <casa> <id> <max>              : Muda o volume máximo da coluna
| dev -cha <casa> <id> <canal>            : Muda o canal da coluna
-------------------------------------------------
Fornecedor -> nome;preco_base;imposto;funcao
Casa -> morada;proprietario;fornecedor;nif{Divisao[Device1 Device2|Divisao[...
Divisao -> nome[device1 device2...
Device -> tipo:id;estado;preco;...
```

Figura 11. Manual de utilização da aplicação

Ao invocar a aplicação sem argumentos, é iniciado o modo de interação com o utilizador. Utilizar um dos comandos acima permite executar as funcionalidades descritas.

Ao invocar a aplicação com um argumento, ficheiro de *input*, é iniciado o modo automático, que realiza os comandos escritos num ficheiro de texto. O resultado das operações é colocado no log.txt da pasta *input*.

-CONCLUSÃO-

Finda a descrição do processo de criação do sistema de monitorização e registo da informação sobre o consumo energético das habitações de uma comunidade, concluímos que a utilização de classes e a sua prévia planificação são cruciais no desenvolvimento de *software*. O uso de linguagem Java permitiu-

nos explorar o paradigma de Programação Orientada a Objetos. Conseguimos, portanto, concluir que a distinção entre a utilização de classes por composição ou agregação é importante na conceção e preservação da integridade dos dados de uma aplicação.

Para além disso, lidamos com as diversas coleções de Java, constatando a sua enorme utilidade e versatilidade na manipulação de dados. Este foi um aspeto que consideramos interessante, uma vez que possibilita total liberdade de *design* algorítmico, retirando preocupações com a sua implementação.

Numa última fase de desenvolvimento do projeto, descobrimos uma filosofia de programação bastante poderosa, o MVC, que nos permitiu modelar e estruturar uma aplicação de uma maneira não espectável. Esta é uma temática que queremos continuar a explorar, visto que não tivemos tempo necessário para a aprofundar devidamente.

Depois de elaborado o projeto, notamos que existem ainda, no entanto, várias situações particulares que continuam a merecer a nossa atenção. Elas são:

- O modo automático reconhece todos os comandos, mas produz resultados errados;
- Os valores de consumo são irreais, pois nenhum membro do grupo teve a disponibilidade de investigar métodos de contagem de consumo elétrico e também não achamos proveitoso despender tempo nessa tarefa. Privilegiamos os conceitos inerentes da Programação Orientada a Objetos na implementação dos mecanismos.

Em suma, o projeto desenvolvido encontra-se funcional, com vários testes unitários que conseguem comprovar a funcionalidade desejada e proposta.