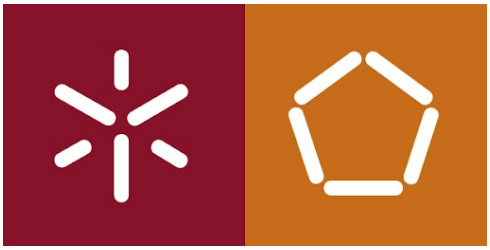

Universidade do Minho

Licenciatura em Engenharia Informática

Smart Management System

Programação Orientada a Objetos

Leonardo Marreiros (a89537)

15 de abril de 2022

Conteúdo

Lista de Figuras

1 Introdução

O presente relatório foi elaborado no âmbito da unidade curricular *Programação Orientada a Objetos* (POO), ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho, e tem como objetivo a criação de uma plataforma onde os utilizadores possam fazer o monitoramento e gestão dos dispositivos inteligentes de casas bem como a capacidade de gerar simulações do valor do consumo de energia de várias casas.

1.1 Descrição do Problema

O projeto proposto implica desenvolver uma plataforma que permita a criação de casas, dispositivos e fornecedores de energia. Com estes, deve ser possível ligar e desligar quer um dispositivos específico como todos os dispositivos de uma dada divisão de uma casa. Além disso, cada casa tem associado um fornecedor de energia e cada dispositivo inteligente tem inerentemente um consumo energético associado. Com isto, deve ser possível calcular uma simulação do consumo de energia e montante monetário a pagar de cada casa durante um número determinado de dias. Como é óbvio, apenas dispositivos ligados consomem energia. É pressuposto que o estado dos dispositivos mantém-se o mesmo durante todos os dias da simulação. Deve ser possível alterar o estado dos dispositivos assim como o seu fornecedor de energia. Igualmente, os fornecedores de energia podem alterar os valores de energia (valor base de custo diário e imposto). No entanto, essas ações só se materializam após o próximo período de simulação, isto é, esse evento permite além de fazer cáclulos mudar a informação que seja necessário. Finalmente, pretende-se que seja possível inquirir sobre um conjunto de estatísticas sobre o programa como: a casa mais gastadora em termos monetários, o fornededor com maior volume de receitas, as faturas emitidas por um dado fornecedor e as casas mais consumidoras de energia num dado período.

De forma mais clara e sucinta os requisitos do programa são os seguintes:

1. Criar casas, dispositivos inteligentes e forncecedores de energia;
2. Avançar para uma determinada data;
3. Calcular o consumo de cada casa assumindo que durante esses dias os dispositivos estiveram sempre no mesmo estado (ligados ou desligados);
4. Criar faturas com informação sobre o período, o consumo e o custo;
5. Solicitar a mudança de fornecedor de energia por parte de uma casa;
6. Ligar e desligar dispositivos, tanto específicos como todos os que se encontrem numa dada divisão de uma casa;
7. Alterar os valores de fornecedores de energia;
8. Determinar qual é a casa que mais gastou até à data;

9. Determinar o fornecedor de energia com maior volume de faturação;
10. Listar as faturas emitidas por um fornecedor;
11. Determinar os maiores consumidores de energia num dado período.
12. Guardar e carregar um estado da aplicação;

2 Diagrama de Classes

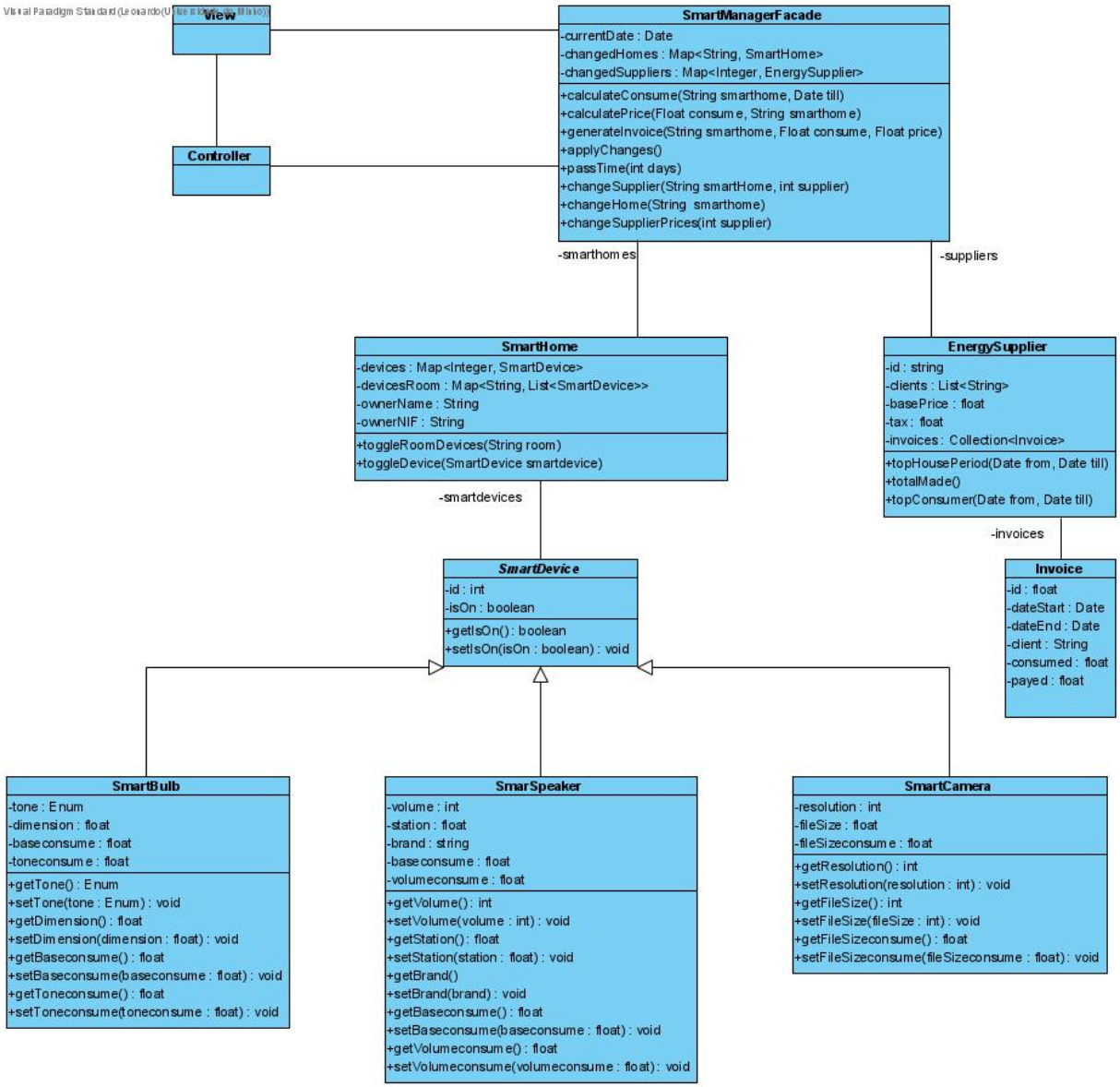

Figura 1: Diagrama de classes

Numa primeira fase foi analisado o problema e desenvolvido um diagrama de classes que se revelou ser a solução ótima que acabou por ser implementada. De seguida descreve-se cada uma das classes que se encontram na solução implementada.

2.1 Classes

2.1.1 *SmartDevice*

A classe *SmartDevice* é uma classe abstrata que tem as seguintes variáveis de instância:

```
private int id;
private boolean isOn;
```

O id e isOn (dispositivo ligado ou desligado) são atributos essenciais (informação comum) a todos os dispositivos inteligentes (*SmartBulbs*, *SmartSpeakers* e *SmartCameras*) pelo que se considerou que seria mais útil e intuitivo elaborar esta classe abstrata.

2.1.2 *SmartBulb*

A classe *SmartBulb*, extensão da classe *SmartDevice*, apresenta outras variáveis de instância que complementam a informação necessária para identificar uma lâmpada inteligente:

```
private Tone tone;
private float dimension;
private float baseConsume;
private float toneConsume;
```

A variável tone identifica a tonalidade em que uma lâmpada inteligente pode ser ligada. Para isto foi criada uma classe enum que apenas contém os valores *COLD, NEUTRAL, WARM*.

A variável dimension representa a dimensão de uma lâmpada em milímetros.

As variáveis baseConsume e toneConsume representam o consumo diário de uma lâmpada. A primeira é um valor fixo e a segunda é um fator de consumo adicional mediante o tipo de luz que está a ser emitido.

O consumo diário de uma lâmpada é dado por:

$$Consumo_{diário} = (baseConsume + toneConsume * (Tone)) \quad (1)$$

2.1.3 *SmartSpeaker*

Tal como a classe anterior, a classe *SmartSpeaker*, extensão da classe *SmartDevice*, apresenta outras variáveis de instância únicas a uma coluna inteligente:

```
private int volume;
private float station;
private String brand;
private float baseConsume;
private float volumeConsume;
```

As variáveis station e brand representam a estação de rádio online em que a coluna está e a sua marca, respetivamente.

O volume da coluna, representado por volume é medido em decibéis e é utilizado na fórmula do consumo energético uma vez que maior volume requer maior potência.

As variáveis baseConsume e volumeConsume são utilizadas para calcular o consumo diário de uma coluna. A primeira é um valor fixo e a segunda é um fator de consumo adicional mediante o volume está a ser emitido.

O consumo diário de uma coluna é dado por:

$$Consumo_{diário} = (baseConsume + volumeConsume * (volume)) \tag{2}$$

2.1.4 *SmartCamera*

Igualmente, sendo um dispositivo inteligente, a classe *SmartCamera* estende a classe *SmartDevice* e apresenta outras variáveis de instância únicas a uma câmara inteligente:

```
private int resolution;
private float fileSize;
private float fileSizeConsume;
```

As variáveis resolution e fileSize representam as características da câmara em termos de resolução de imagem e tamanho dos ficheiros gravados.

A variável fileSizeConsume representa o coeficiente pelo qual o tamanho dos ficheiros gerados será multiplicado para determinar o consumo energético diário.

O consumo diário de uma câmara é dado por:

$$Consumo_{diário} = (fileSizeConsume * fileSize * (resolution/1080)) \tag{3}$$

2.1.5 *SmartHome*

A classe *SmartHome* representa uma casa que apresenta as seguintes variáveis de instância necessárias para a sua identificação:

```
private Map<Integer, SmartDevice> devicesAll;
private Map<String, List<Integer>> devicesRoom;
private String ownerName;
private String ownerNIF;
```

Cada casa é identificada pelo NIF do proprietário e pelo seu nome nas variváreis ownerNIF e ownerName respetivamente.

Cada casa pode ter uma coleção de dispositivos inteligentes de qualquer dos tipos anteriores. Para isso existe a variável devicesAll que representa um hashmap de todos os dispositivos existentes nessa casa tendo como chave o identificador único de cada dispositivo.

Numa casa existem várias divisões, divisões essas que podem conter dispositivos inteligentes. Para isso existe a variável devicesRoom que representa um hashmap que tem como chave o nome da divisão e como valor uma lista de identificadores de dispositivos inteligentes presentes nessa divisão.

2.1.6 *Invoice*

No final de cada perído de simulação são geradas faturas com informações acerca do período que passou. A classe *Invoice* representa uma fatura e conta com as seguintes variáveis de instância:

```
private LocalDate dateStart;
private LocalDate dateEnd;
private String client;
private float consumed;
private float payed;
```

Cada fatura inclui as seguintes informações: data de ínicio do período de faturação (dateStart), data de fim do período de faturação (dateEnd), cliente a que a fatura se refere, neste caso o NIF do proprietário da casa (client), total de energia consumido nesse período em killowatts (consumed), e total pago pela energia consumida, em euros (payed).

2.1.7 *EnergySupplier*

A classe *EnergySupplier* representa um fornecedor de energia e contém as seguintes variáveis de instância:

```
private String name;
private List<String> clients;
private Map<String, Invoice> invoices;
private float basePriceRate;
private float tax;
```

Um fornecedor de energia tem uma designação (name) e contém informação acerca dos clientes que atualmente recebem a sua energia na forma de uma lista com os identificadores dos proprietários de cada casa, isto é, o seu NIF (clients).

Além disso, são guardadas todas as faturas geradas por um fornecedor de energia na forma de um hashmap cuja chave é o número da fatura e o valor a fatura correspondente (invoices).

Finalmente, um forncedor de energia tem um preço base para o killowatt de energia na variável basePriceRate e uma taxa de imposto associada, tax.

2.1.8 SmartManagerFacade

Esta classe representa a classe principal da aplicação, envolvendo as principais classes apresentadas anteriormente de forma a cumprir os requisitos. Tem as seguintes variáveis de instância:

```
private LocalDate currentDate;
private Map<String, SmartHome> smarthomes;
private Map<String, SmartHome> changedsmarthomes;
private Map<Integer, EnergySupplier> suppliers;
private Map<Integer, EnergySupplier> changedsupplier;
```

A variável currentDate representa o dia atual na simulação, isto é, é atualizada de cada vez que é "passado tempo"nas simulações. Por exemplo, se a aplicação é iniciada a 12/04/2022, após um período de simulação de 10 dias esta variável será 22/04/2022.

Para armazenar as casas e os fornecedores de energia foram utilizados os maps smarthomes e suppliers respetivamente. A chave do primeiro mapa é o NIF do proprietário da casa, sendo assim foi assumido que cada pessoa apenas pode ter uma casa e que não existem NIF repetidos. No segundo, a chave é um identificador único de cada fornecedor de energia.

Na medida em que apenas devem ser materializadas eventuais mudanças quer ao estado dos dispositivos das casas, quer ao fornecedor de energia de uma casa ou aos valores de energia de cada fornecedor após um período de simulação, foi necessária a criação de dois maps para guardarem as mudanças que ocorreram entre períodos de simulação, changedsmarthomes e changedsupplier. No fim de uma simulação caso estes mapas contenham dados, são alterados os mapas smarthomes e suppliers com os dados alterados correspondentes.

2.1.9 Exceções

De forma a melhor controlar fluxos excecionais no programa foram desenvolvidas um conjunto de classes *Exception*.

InexistentInvoicesException Criada para o caso de ainda não terem sido geradas faturas pelo que é impossível calcular estatísticas como a casa mais dispendiosa ou o fornecedor com maior faturação. Acontece antes de terem sido geradas simulações no programa.

InvalidDeviceException Lançada quando um dispositivo inquirido não existe. Surge quando o utilizador tenta trocar o estado de um dispositivo específico que não existe.

InvalidHomeException Lançada quando uma casa inquirida não existe. Surge em qualquer funcionalidade que envolva o *input* do utilizador acerca de uma casa em específico nos casos em que o *input* inserido não corresponda a nenhuma casa presente no sistema.

InvalidRoomException Lançada quando a divisão de uma casa inquirida não existe. Surge quando o utilizador escolhe desligar ou ligar todos os dispositivos de uma dada divisão que não exista na casa em particular.

InvalidSupplierException Lançada quando o forncecedor de energia inquirido não existe. Surge quando uma funcionalidade que requira o *input* do utilizador acerca do identificador de um fornecedor de energia resulte num identificador que não existe atualmente no sistema.

SameSupplierException Lançada quando é requerida a mudança de fornecedor de energia de um cliente para o seu fornecedor de energia atual.

3 Arquitetura

O programa segue a arquitetura *Model View Controller* (MVC) estando portanto organizado em três camadas:

- A camada de dados (modelo) é constituída pelas classes de dispositivos inteligentes, casas, faturas e fornecedores de energia. Estas classes convergem na classe *SmartManagerFacade* onde são concretizadas todas as funcionalidades da lógica de negócio.
- A camada de interação com o utilizador (vista) é composta unicamente pela classe *View* que está encarregue de apresentar menus, pedir *inputs* e receber estruturas genéricas de dados para apresentar o resultado das funcionalidades ao utilizador.
- A camada de controlo do fluxo do programa (controlador) é composto pela classe *Controller* que funciona como uma ponte entre o modelo e a vista. Acede às funcionalidades do modelo e utiliza o seu resultado num correspondente método da vista para apresentar ao utilizador.

Todo o projeto basea-se no ideal de encapsulamento e modularidade. Isto verifica-se no facto de não haver relações entre classes que não o necessitam, por exemplo, um fornecedor de energia, apesar de ter uma lista de clientes (casas), não depende da classe das casas, não contém uma lista de *Smarthome* mas sim uma lista de *strings* cujo significado de cada corresponde ao identificador único de cada casa. Além disso, também é feita a cópia profunda de objetos em situações que o requiram, de forma a não violar o encapsulamento de dados.

4 Funcionalidades

4.1 *Setup* inicial dos dados

De forma a evitar que os utilizadores tenham de fazer o *setup* da informação de cada vez que usam o programa, a criação de casas, dispositivos e fornecedores de energia é feita em ficheiro JSON. Foram criados dois ficheiros com um *setup* exemplo de informação: EnergySuppliers e SmartHomes, que têm a seguinte estrutura:

```
{
    "ownerName" : "John Doe",
    "ownerNIF" : "282731629",
    "devices" : [
        {
            "type" : "SmartBulb",
            "id" : "1",
            "isOn" : "false",
            "tone" : "WARM",
            "dimension" : "62",
            "baseConsume" : "0.045",
            "toneConsume" : "0.016"
        },
        {
            "type" : "SmartSpeaker",
            "id" : "2",
            "isOn" : "true",
            "volume" : "86",
            "station" : "92.9",
            "brand" : "JBL",
            "baseConsume" : "1.4",
            "volumeConsume" : "0.01"
        },
        {
            "type" : "SmartCamera",
            "id" : "3",
            "isOn" : "false",
            "resolution" : "360",
            "fileSize" : "0.5",
            "fileSizeConsume" : "0.96"
        }
    ],
    "rooms" : [
        {"bedroom" : ["1"]},
        {"kitchen" : ["2"]},
        {"living-room" : ["3"]},
    ]
}
```

Listing 1: Representação textual de um objeto de tipo *SmartHome* no ficheiro SmartHomes

```
{
    "name" : "EDP",
    "basePriceRate" : "0.755",
    "tax" : "0.13",
    "clients" : ["282731629"]
}
```

Listing 2: Representação textual de um objeto de tipo *EnergySupplier* no ficheiro EnergySupplier

Quando é iniciado o programa é carregada a configuração destes ficheiros nas estruturas adequadas. O parse dos ficheiros é feito nas classes *JsonEnergySuppliersDataReader* e

JsonSmartHomesDataReader. Para isso foi importada a biblioteca *json-simple*.

4.2 Listar faturas emitidas por um fornecedor

Esta funcionalidade permite listar todas as faturas geradas por um fornecedor dado o ID desse fornecedor. Para isso, primeiro vai-se buscar o fornecedor com o ID correspondente (caso exista) e depois gera-se um Map¡String, String¿ baseado na lista de faturas desse fornecedor cuja chave é o número de fatura e o valor é a fatura (em formato *string*).

4.3 Casa mais dispendiosa por período

Esta funcionalidade permite determinar qual das casas do sistema gastou mais em energia num dado período de simulação válido. Para isto cada fornecedor determina dentro dos seus clientes qual aquele que gastou mais nesse período e depois é comparado entre fornecedores este resultado, determinando qual o cliente com maior gasto absoluto.

4.4 Fornecedor com maior volume de faturação

Esta funcionalidade permite determinar qual o fornecedor que ganhou mais dinheiro dos seus clientes até à data. Para isto é determinado o total feito por cada fornecedor a partir da soma dos valores das suas faturas e depois comparado este resultado entre fornecedores para determinar qual o com maior volume de faturação.

4.5 Ordenação das casas mais consumidoras num determinado período

Esta funcionalidade permite obter uma lista ordenada das casas mais consumidoras de energia (em kW) num determinado período de simulação válido. Isto é conseguido ao obter primeiramente para cada fornecedor uma ordenação dos clientes mais consumidores nesse período, concatenando este resultado com as ordenações resultantes dos outros fornecedores e organizando o resultado por ordem crescente.

4.6 Consultar o estado atual das casas

Esta funcionalidade permite consultar o estado atual de uma casa em particular (fornecendo o NIF do proprietário da casa) ou de todas as casas. Isto é conseguido facilmente utilizando os métodos *toString* desenvolvidos para cada classe que envolve uma casa (*SmartHome, SmartDevice, SmartBulb, SmartCamera, SmartSpeaker*) e apresentando esse resultado.

4.7 Consultar o estado atual dos fornecedores de energia

Esta funcionalidade permite consultar o estado atual de um fornecedor de energia em particular (fornecendo o ID do fornecedor de energia) ou de todas os fornecedores de energia. Isto é conseguido facilmente utilizando o método *toString* desenvolvido para a classe referente a um fornecedor de energia (*EnergySupplier*) e apresentando esse resultado.

4.8 Trocar o estado de dispositivos numa casa

Esta funcionalidade permite trocar o estado (ligar ou desligar) de um dispositivo em específico de uma casa (fornecendo o ID do dispositivo e o NIF da casa) ou de todos os dispositivos de uma divisão de uma casa (fornecendo o nome da divisão e o NIF da casa). Isto é conseguido fazendo primeiramente uma cópia profunda da casa requerida pelo utilizador, com isto, é depois trocado o estado do(s) dispositivo(s) de acordo com o pretendido pelo utilizador. Esta cópia agora mudada é inserida então no mapa de casas mudadas presente na classe *SmartManagerFacade*. Esta cópia profunda é efetuada pois caso contrário o estado do(s) dispositivo(s) alterados seria logo materializado que não é o objetivo do programa, isto é, estas mudanças apenas podem ser visíveis após um período de simulação.

4.9 Trocar os valores de energia de um fornecedor

Esta funcionalidade permite trocar os valores de energia (preço base e imposto) de fornecedor (dado o ID do fornecedor). Isto é conseguido guardando uma cópia do fornecedor de energia requerido com o valor a alterar diferente do original no mapa de fornecedores de energia mudados presente na classe *SmartManagerFacade*. Estas mudanças serão depois aplicadas após um período de simulação.

4.10 Alterar o fornecedor de energia de uma casa

Esta funcionalidade permite trocar qual o fornecedor de energia de uma dada casa (fornecendo o NIF da casa e o ID do novo fornecedor). Isto é conseguido com uma série de passos simples: primeiro é feita uma cópia da lista de clientes do fornecedor de energia atual da casa requirida, depois é removido esse cliente da lista, a seguir é feita uma cópia da lista de clientes do fornecedor de energia para o qual o cliente vai trocar e é inserido esse cliente na lista. Finalmente, são colocados no mapa de fornecedores de energia mudados presente na classe *SmartManagerFacade* as cópias destes fornecedores de energia com as listas de clientes atualizadas. Novamente, estas mudanças serão depois aplicadas após um período de simulação.

4.11 Avançar para uma determinada data

Esta é a funcionalidade principal do programa e permite fazer uma simulação, avançando para uma determinada data obtendo as faturas e eventuais mudanças de informação pendentes. Isto envolve uma série de passos: para cada casa é calculado o consumo energético durante os dias de simulação, com este valor é calculado o total a pagar dependendo do fornecedor de energia correspondente de cada casa. Com esta informação é então possível gerar as faturas. De seguida são efetuadas as eventuais mudanças pendentes, isto é, caso os mapas de mudanças em casas ou fornecedores de energia contenham dados, então os valores contidos nestes mapas substituem os valores atuais. No caso de haver valores nestes mapas, depois de efetivadas as concretizações, é dado um *reset* aos mapas de mudanças. É desta forma materializada as alterações de informação pendentes. Além disso, é também atualizada a data atual na simulação para a nova data que foi inserida.

4.12 Salvaguarda do estado da aplicação

Através da utilização de *Serializable* é possível a qualquer momento guardar em ficheiro a informação existente em memória. Igualmente, é possível carregar estes ficheiros de forma a recuperar a informação presente nesse estado.

5 Conclusão

Face ao problema apresentado e analisando criticamente a solução implementada é possível concluir que todas as tarefas com exceção da automatização da simulação foram concluídas, conseguindo atingir os objetivos definidos. As soluções encontradas para os problemas foram pensadas de forma a serem facilmente adaptadas a situações futuras, por exemplo a criação da classe abstrata *SmartDevice* existe com o intuito de facilitar a inserção de novos tipos de dispositivos inteligentes.

O problema de materializar as ações de mudanças no estado dos dados apenas quando se faz o avançar do tempo foi ultrapassado eficientemente com a criação de cópias dos objetos e a criação de dois mapas de mudanças pendentes.

Em suma, não obstante as potenciais melhorias que poderiam ser feitas no programa, nomeadamente, a utilização um *TreeMap* ao invés de um *HashMap* para a estrutura de dados de faturas nos fornecedores de energia possibilitaria a organização cronológica das faturas por data, é justo considerar que o projeto cumpre todos os requisitos pedidos.