Universidade do Minho
Escola de Engenharia

Smart Community

Trabalho Prático
Programação Orientada aos Objetos

Grupo 3

Nuno Guilherme Cruz Varela - a96455

Gabriela Santos Ferreira da Cunha - a97393

Miguel de Sousa Braga - a97698

a96455
Guilherme Varela

a97393
Gabriela Cunha

a97698
Miguel Braga

1 Introdução

Este trabalho consiste na implementação de um sistema que monitorize e registe a informação sobre o consumo energético das habitações de uma comunidade. Cada casa possui um conjunto de *Smart Devices*, dispositivos inteligentes identificados por um código único, que podem ser ligados ou desligados a partir deste programa. Estes dispositivos têm um custo de instalação associado e podem ser agrupados por divisão da casa, sendo também possível monitorizar e controlar os dispositivos apenas de uma determinada divisão.

2 Arquitetura de classes

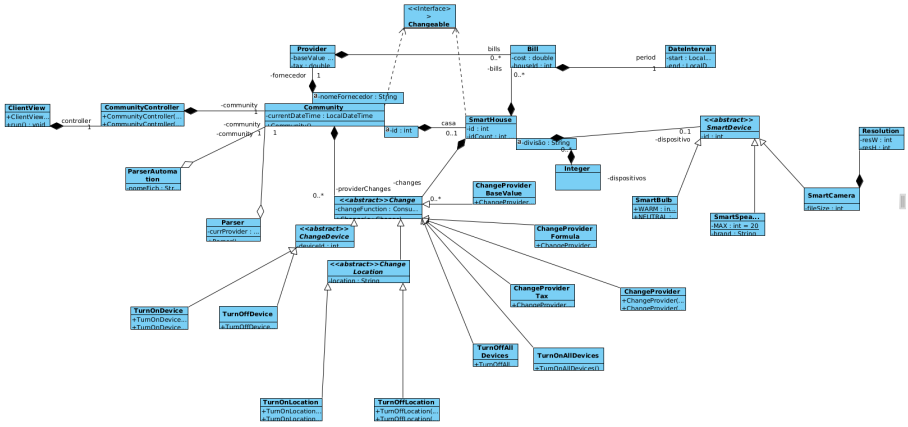

Figura 1: Esboço do diagrama de classes

2.1 SmartDevice

```java
public abstract class SmartDevice implements Serializable {
    private final int id;
    private boolean onState;
    private int costOfInstallation;
    private double baseConsumption;

    private static int idCount = 1;
```

Figura 2: Variáveis de instância da classe *SmartDevice*

Neste momento, temos disponíveis vários tipos de *SmartDevices* na Casa Inteligente: *SmartBulb*, *SmartCamera* e *SmartSpeaker*.

Contudo, é nosso desejo poder adicionar novos dispositivos mais tarde, com a implementação de novas funcionalidades. Para isso, elaboramos o nosso programa de modo a ser simples de acomodar mais entidades do tipo *SmartDevice*. Essas novas entidades serão adicionadas à hierarquia dos *SmartDevices*, sendo que o *SmartDevice* é uma classe abstrata que define algumas variáveis de instância, como o id (autoincrementado), estado (ligado/desligado), custo de instalação, consumo base, pertencentes a todos os tipos de dispositivos, e algumas funcionalidades, como o cálculo do consumo.

2.2 SmartHouse

```
public class SmartHouse implements Comparable<SmartHouse>, Serializable, Changeable {
    private int id;
    private String address;
    private String ownerName;
    private int ownerNif;
    private String providerName;
    private double currentConsumption;
    private double totalKwh;
    private double accumCost;
    private Map<String, Set<Integer>> locations;
    private Map<Integer, SmartDevice> devices;
    private Set<Bill> bills;
    private List<Change> changes;
    private LocalDateTime lastBill;
    private LocalDateTime lastChange;

    private static int idCount = 1;
```

Figura 3: Variáveis de instância da classe *SmartHouse*

Dentro das variáveis base de uma dada casa encontra-se o id (auto-incrementado), o endereço, o nome do dono e o respetivo NIF. Qualquer uma destas variáveis poderia servir como chave da nossa casa. No entanto, decidimos adicionar um id gerido por nós como chave, de modo a termos mais controlo - evitando chaves demasiado longas ou chaves que pudessem colidir entre várias casas.

No nosso programa uma casa é uma composição de vários SmartDevices, agrupados por divisões. Os *SmartDevices* encontram-se armazenados num *HashMap* que mapeia o id do dispositivo para o próprio dispositivo. Escolhemos esta estrutura para facilitar as pesquisas por id dos dispositivos. Para o agrupamento em divisões, escolhemos também um *HashMap*, mas desta vez com uma *String* como chave a representar o nome da divisão e um conjunto (*HashSet*) com os ids dos dispositivos como valor de cada entrada.

Por outro lado, a casa apresenta informações sobre o fornecedor a ela associado (nome) bem como o conjunto das faturas emitidas para essa casa, pelos diferentes fornecedores, nos diferentes intervalos de simulação. Este conjunto é representado num *TreeSet* porque nos interessa a noção de ordem para mais tarde conseguirmos procurar faturas dentro de um certo intervalo.

A Casa inteligente é um dos pontos centrais do nosso programa na medida em que é lá que se fazem as alterações mais importantes (em *runtime*) do nosso programa, tais como alterar dispositivos, fornecedores, emitir faturas, etc. Como tal, é a classe mais complexa do nosso programa, apresentando outras variáveis de controlo do estado de simulação, únicas a cada casa.

2.3 Provider

```
public class Provider implements Serializable{
    private String name;
    private double baseValue;
    private double tax;
    private double currentCost;
    private String formula;
    private Set<Bill> bills;
```

Figura 4: Variáveis de instância da classe *Provider*

A classe *Provider* representa os comercializadores de energia da nossa aplicação. Estão a ser guardados nesta classe o nome, que vai servir como identificador do prestador de serviço, o valor base, a taxa e a fórmula associados a este provedor. Decidimos, ainda, utilizar uma estrutura para guardar todas as faturas emitidas por este, de modo a responder a certas estatísticas sobre o programa. A variável *currentCost* representa o volume de faturação do fornecedor, algo que nos será útil na construção da *query* que pede o fornecedor com maior faturação e, portanto, torna o funcionamento desta mais eficiente.

2.4 Community

```
public class Community implements Serializable, Changeable {
    private Map<Integer, SmartHouse> houses;
    private Map<String, Provider> providers;
    private LocalDateTime currentDateTime;
    private List<Change> providerChanges;
```

Figura 5: Variáveis de instância da classe *Community*

A comunidade é a classe que vai albergar todas as casas e fornecedores existentes no programa. Como podemos observar na figura 5, utilizamos um *HashMap* para guardar as casas, em que a chave corresponde ao seu id e o valor associado é a própria casa. É feito o mesmo para os fornecedores, mas, desta vez, a chave corresponde ao seu nome. Optamos por estas estruturas visto que nos oferecem tempos de acesso constantes aos dados que vamos precisar no decorrer do programa. Para além disto, a comunidade guarda uma lista das alterações a fazer nos fornecedores, algo que nos será util para quando avançamos o tempo.

Esta classe contém todos os métodos responsáveis pelas devidas alterações fornecidas pelo utilizador a aplicar na estrutura das casas e dos utilizadores com vista a alterar o estado da comunidade, assim como os que tratam de percorrer as estruturas para responder às estatísticas pedidas.

2.5 ClientView

```
public class ClientView {
    private CommunityController controller;
    private Scanner sc;
```

Figura 6: Variáveis de instância da classe *ClientView*

Faz parte do conjunto das boas práticas de qualquer linguagem de programação separar a funcionalidade que interage com o cliente da funcionalidade responsável por realizar os pedidos do clientes. Assim, criamos a classe *ClientView* que trata das operações de *Input/Output* com o utilizador.

Uma das primeiras funcionalidades que esta permite é a escolha do modo de leitura dos dados necessários para a simulação. Estes dados podem ser carregados através de um ficheiro de texto, de um ficheiro binário ou de um menu. O carregamento a partir de ficheiro de texto é assegurado pela classe *Parser*, enquanto que o carregamento a partir de ficheiro binário é realizado dentro do controlador da comunidade.

Esta classe é também responsável por inquirir o utilizador quanto às suas opções de simular o decorrer do programa. Dentro desta, podemos escolher entre simular a execução do programa por menu ou por automatização. Os menus (existentes nesta classe) permitem ao utilizador alterar dispositivos, fornecedores, avançar o tempo, mostrar faturas, casas e fornecedores, assim como pedir resultados das estatísticas e salvar o estado atual em ficheiro.

2.5.1 Tratamento de exceções

Como nem tudo pode correr bem a meio da execução do nosso programa - o utilizador pode inserir valores inválidos -, temos de ter um mecanismo responsável por lançar e tratar erros em *runtime*. O uso de exceções revelou-se essencial neste caso, pois permite que os erros sejam detetados no modelo e avaliados (apanhados) na vista. Uma exceção a esta regra tem a ver com as mudanças que podemos operar aos dispositivos de uma casa. Como estas mudanças são realizadas por métodos que serão executados a meio da execução, temos de ser capazes de detetar eventuais erros nas mudanças antes de estas chegarem ao modelo. É essa uma das funcionalidades do controlador.

2.6 CommunityController

Figura 7: Variáveis de instância da classe *CommunityController*

A classe *CommunityController* fornece à *view* um serviço de interligação com o modelo (a comunidade), encaminhando pedidos da *view* para a comunidade e para a *view* a respetiva resposta da comunidade.

Esta classe fornece outros serviços como começar a leitura de ficheiros ou guardar o estado atual da comunidade em ficheiro. Para além disso, como a vista não conhece (nem deve conhecer) os tipos de dados do modelo, esta classe é também responsável por criar os tipos de dados adequados (*SmartDevice*, *Provider*, *SmartHouse*, etc), enviando-os para a comunidade por composição.

Assim, consideramos que para além de funcionar como controlador, esta classe serve também como fachada da comunidade, na medida em que fornece uma API para interação com a comunidade.

3 Funcionalidades do programa

3.1 Requisitos base de gestão de entidades

O nosso programa cumpre a implementação de todos os requisitos base referidos no enunciado: criar dispositivos, casas e fornecedores de energia, ligar e desligar dispositivos e, ainda, avançar o tempo e calcular o consumo e gerar as respetivas faturas.

Assim, é possível no menu principal do nosso projeto adicionar novas casas e fornecedores de energia à comunidade e novos dispositivos a uma dada casa fornecida pelo utilizador (através do ID). Para o avanço do tempo, apenas é necessário indicar o número de horas relativas ao avanço solicitado e as faturas são automaticamente geradas, estas que podem ser consultadas a qualquer momento na opção "Mostrar faturas" do menu e são sempre relativas ao último avanço do tempo feito pelo utilizador.

3.1.1 Criação de dispositivos, casas e fornecedores

No início da execução, o utilizador pode carregar dados de uma comunidade através de um ficheiro de texto ou binário ou criá-la a partir de um menu. Assim como no menu de criação da comunidade, também é possível, posteriormente, alterar os dados da comunidade carregada - adicionar novos dispositivos a uma dada casa, criar novos fornecedores (com nome, fórmula, taxa de imposto e valor base fornecidos pelo utilizador) e novas casas.

3.1.2 Ligar/desligar dispositivos

No requisito de ligar/desligar dispositivos, decidimos adicionar as opções de o fazer apenas para um dispositivo específico, para uma divisão inteira ou para toda a casa inteligente.

3.1.3 Avanço do tempo

O avanço do tempo do nosso programa é feito em horas, pelo que é perguntado ao utilizador a quantidade de horas que quer simular. Após o avanço do tempo, são aplicadas as alterações nas casas e fornecedores, no caso de existirem, e são emitidas as respetivas faturas para cada casa.

3.2 Estatísticas sobre o estado do programa

De modo a atacar as *queries* propostas, decidimos implementá-las priveligiando a sua eficiência e operacionalidade. Explicaremos, em seguida, as estratégias utilizadas para cada estatística.

3.2.1 Casa que mais gastou naquele período

Esta estatística sobre o estado do programa pede-nos a casa que mais gastou no último período de simulação (após o avanço do tempo). Para tal, decidimos implementar a ordem natural das faturas como sendo por ordem da data em que foram emitidas. Isto permite-nos ter um *TreeSet* de faturas ordenado em cada casa, pelo que nos basta apenas pegar na última fatura desta árvore e verificar se a data de emissão desta coincide com a data atual da comunidade, o que significa que foi emitida no último avanço do tempo. O maior esforço desta *query* é o facto de termos de percorrer toda a estrutura que contém as casas, mas é algo inevitável, uma vez que necessitamos do maior consumo de cada casa.

3.2.2 Comercializador com maior volume de faturação

De modo a satisfazer esta estatística, decidimos adicionar uma variável de instância à classe *Provider*, com o objetivo de guardar ao longo do programa a respetiva faturação acumulada. Esta implementação evita que tenhamos de percorrer a lista de faturas associada a cada fornecedor sempre que queremos calcular a sua faturação. Com isto, basta fazer uma travessia na estrutura dos fornecedores e verificar qual tem o maior valor associado a esta variável.

3.2.3 Listar as faturas de um comercializador

Esta estatística foi, a nosso ver, a mais simples, sendo que para a implementar apenas foi necessário percorrer a lista de faturas associada ao fornecedor em questão de modo a listar todas as faturas emitidas por este.

3.2.4 Ordenação dos maiores consumidores de energia durante um período a determinar

Para responder à ordenação, começamos por criar um *TreeSet* com um comparador específico que calcula o consumo das casas no período fornecido como parâmetro. Para calcular este consumo num certo período, percorremos a lista de faturas associada a cada casa e verificamos as que estão no intervalo fornecido. Após a criação desta árvore, vamos percorrer a estrutura da comunidade que guarda as casas e adicioná-las a esta nova estrutura, de modo a ficarem ordenadas. Finalmente, já temos os maiores consumidores ordenados por energia e devolvemos esta ordenação.

3.3 Alteração dos operadores e dispositivos de uma casa

Tal como explicitado nos pontos 5, 6 e 7 do enunciado do trabalho prático, podemos operar mudanças a meio da simulação, seja no fornecedor ou nos dispositivos de uma casa, seja nos próprios valores dos fornecedores. Um ponto importante refere que, nos requisitos base deste programa, as alterações só poderão ser feitas quando se pedir o avançar do tempo, sendo criado um novo período de simulação. Por isso, decidimos que, tanto na comunidade como nas casas da comunidade, teríamos uma estrutura de dados que guardasse as mudanças que seriam pedidas pelo utilizador no menu. Mais tarde, no início da simulação seguinte, aplicaríamos essas mudanças na entidade correspondente.

3.3.1 A hierarquia *Change*

```
public abstract class Change implements Serializable {
    private Consumer<? extends Changeable> changeFunction;
```

Figura 8: Variáveis de instância da classe *Change*

As nossas mudanças seguem uma hierarquia que diferencia as mudanças aplicadas a dispositivos das mudanças aplicadas aos fornecedores, tornando fácil o acolhimento de novas mudanças num futuro *update*. Todas as mudanças são subclasses da classe *Change*, que adiciona uma varíavel de instância (um *consumer*) e vários métodos de construção e de controlo do *consumer* guardado. Este *consumer* é instanciado no construtor de cada uma das subclasses Mudança, através de uma expressão lambda adequada. Acreditamos que isto fornece à nossa aplicação uma maior flexibilidade pois para adicionar novas mudanças apenas teremos que criar uma nova expressão lambda. Este *consumer* é depois aplicado na classe correspondente (que implemente a interface *Changeable*) através do método "accept", existente na API "java.util.function".

3.3.2 A interface *Changeable*

```
public interface Changeable {
    public void applyChanges();
    public void addChange(Change c);

}
```

Figura 9: Interface *Changeable*

Esta interface é usada para definir que objetos são capazes de sofrer alterações no seu estado a meio da execução do programa.

Neste momento, apenas os fornecedores, os dispositivos e as casas podem sofrer essas alterações. No entanto, como os fornecedores pertencem à comunidade e os dispositivos às casas, decidimos que seriam as classes *Community* e *SmartHouse* a implementar esta interface, garantindo assim um meio termo entre modularidade e eficiência.

Tal como podemos ver na figura acima, esta interface estabelece um contrato em como estes objetos têm de implementar os métodos "applyChanges" e "addChange", dando a entender a necessidade de uma estrutura de dados que guarde as mudanças dentro da classe que implemente esta interface. Como nem todos os dispositivos nem todos os fornecedores são alterados em cada intervalo de simulação, seria um desperdício de espaço e eficiência estar a guardar uma estrutura (vazia) de *Changes* em instâncias que não sofressem alterações nesse período.

3.4 Automatização

3.4.1 Objetivos

Até agora, temos trabalhado com um método de simulação totalmente por menu, em que o utilizador define as mudanças a aplicar no início do próximo período de simulação.

Não estando no enunciado, temos trabalhado até agora com avanços no tempo em horas, de modo a simplificar as unidades, estando mais de acordo com a noção que temos da realidade. No entanto, nesta fase, fomos mais além, procurando que a nossa simulação adquirisse um maior grau de realismo e semelhança com a realidade.

Para isso, elaboramos um interpretador (*Parser*) de comandos que simulem o decorrer da simulação numa comunidade. Temos o mesmo tipo de mudanças referido anteriormente, no entanto, estas mudanças são aplicadas na hora, tendo impacto no consumo da casa até serem desligadas. Quanto às mudanças nos fornecedores, estas também podem ocorrer a qualquer momento, com a diferença de que forçam a emissão de uma fatura para todas as casas que sofram com a alteração no valor (valor base, imposto ou fórmula) do fornecedor. Temos por isso, intervalos de simulação diferentes para cada uma das casas, não deixando de haver momentos em que podemos definir a emissão de faturas para todas as casas da comunidade (comando "endOfSimulation").

3.4.2 Implementação

A estratégia abordada para permitir alterações no estado dos dispositivos sem emissão de faturas para a casa que sofre a mudança foi dividir um intervalo de simulação em intervalos mais pequenos. Cria-se um novo subintervalo sempre que há uma alteração de um qualquer dispositivo da casa (ligar ou desligar).

Criamos, por isso, variáveis adicionais de controlo na classe *SmartHouse* (*currentConsumption, totalKwh e lastChange*). A variável *currentConsumption* guarda o consumo (em Kw) da casa, sendo atualizada sempre que se liga ou desliga um dispositivo. Deste modo, evita-se realizar a soma dos consumos de cada dispositivo, sempre que queremos saber este consumo. A variável *totalKwh* acumula o consumo (em Kwh) desde a emissão da última fatura, sendo atualizada no início de cada subintervalo, com base no tamanho do intervalo (em horas) e no consumo durante esse intervalo (em Kw). A última variável (*lastChange*) guarda a data da última mudança, permitindo calcular assim o tempo desde a última mudança se soubermos a data atual.

A classe *ParserAutomation* é responsável por ler os comandos e acionar na comunidade a mensagem correspondente. Neste caso, não usamos o padrão MVC, interagindo diretamente com o modelo (comunidade), por não estarmos propriamente perante uma situação de *input/output*, não havendo necessidade de tratar - para já - as respostas da comunidade.

4 Conclusão

Neste trabalho, fomos introduzidos de uma forma bastante prática ao paradigma da programação orientada aos objetos e, sobretudo, aos novos padrões de desenvolvimento de aplicações neste paradigma.

Assim, aprofundamos os conceitos de encapsulamento, modularidade e abstração de dados, tão importantes para garantir uma evolução saudável do nosso programa no sentido de permitir acrescentar novas funcionalidades sem necessitar de alterar muito o código já existente. Aprendemos que um dos ingredientes para garantir a longevidade de uma aplicação é gastar um pouco de tempo no planeamento, algo por vezes difícil de realizar. Nesse sentido, ao elaborar o esboço inicial do diagrama de classes e, mais tarde, o final, pudemos entender a importância de alguns objetos existentes no java como as classes abstratas, as interfaces e as próprias estruturas de dados.

Por fim, fazemos uma análise bastante positiva ao trabalho realizado. Temos a noção de que há aspetos a melhorar, no entanto acreditamos que o trabalho e o esforço empenhado valeram pelo conhecimento adquirido.