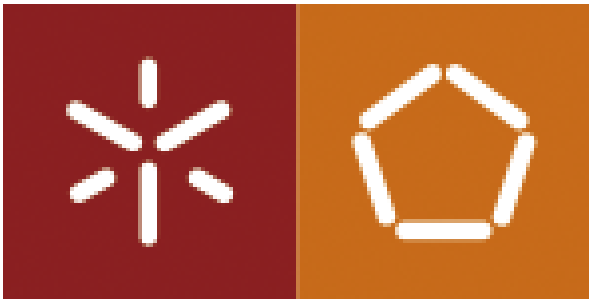

Universidade do Minho

Programação Orientada aos Objetos

Relatório do Trabalho Prático

Artur Jorge Castro Leite

A97027

Afonso Miguel Rodrigues Magalhães

A95250

Nuno Gonçalo Machado Rodrigues

A90439

Índice

1) Introdução

Foi proposta a realização de uma aplicação Java capaz de monitorizar o consumo energético de um conjunto de casas. Para além disso, este programa deverá ter a capacidade de controlar certos dispositivos: os *SmartDevices*.

Durante o desenvolvimento do projeto, visamos sempre manter os princípios fundamentais do JAVA, nomeadamente:

- Encapsulamento, através da criação de métodos que permitem aceder e modificar atributos de qualquer classe;
- Hierarquia, através da utilização de superclasses que englobam métodos comuns a determinadas classes;
- Polimorfismo, que resulta da invocação dos mesmos métodos para subclasses distintas.

2) Arquitetura das classes

2.1) Menus de Interação

Na realização do projeto, optamos por criar duas classes responsáveis pela interação entre o utilizador e a parte computacional da aplicação. As classes utilizadas são: Menu e Programa.

Nestas classes, estão implementados também métodos que permitem popular as Casas, Dispositivos, Fornecedores, etc, de acordo com os inputs do usuário.

2.2) Casa

A classe Casa é onde estão definidas as diversas informações relevantes à casa. Eis as variáveis de instância presentes nesta classe:

- **`String nome`**- nome do proprietário;
- **`String nif`**- número de identificação fiscal do proprietário;
- **`Map<Integer, SmartDevice> devices`**- mapeamento entre ID do dispositivo e *SmartDevice* correspondente;
- **`Map<String, Set <Integer>> rooms`-** mapeamento entre divisória e coleção dos IDs dos *SmartDevices* dessa divisória;
- **`List <Fatura> faturas`-** lista de faturas emitidas;
- **`String fornecedor`-** nome da empresa fornecedora de energia.

Através desta classe, conseguimos implementar alguns métodos essenciais:

- Obter listas de todos os dispositivos da casa, podendo também obter lista de dispositivos de cada divisão;
- Obter lista de divisões da casa;
- Definir o estado dos dispositivos;
- Testar se existem certos dispositivos;

Estas implementação são cruciais para o resto do projeto, pois através do encapsulamento, quaisquer classes que necessitem de aceder às informações referentes aos Dispositivos ou aos Consumos por exemplo, serão utilizadas estas funções definidas.

2.3) Fornecedor

Na classe Fornecedor apenas temos presente uma variável de instância:

- **`String name`-** nome da empresa Fornecedora;
- **`List <Fatura>`-** tal como na classe casa, lista de faturas emitidas;
- **`Double Desconto`-** desconto aplicado no custo.

A principal funcionalidade desta classe advém do método que implementamos que é capaz de criar Faturas. É através destas Faturas que somos capazes de armazenar novas informações referentes a custos e consumos que são essenciais nos processos de organização de casas e fornecedores de acordo a estes parâmetros.

2.4) SmartDevice

A classe SmartDevice é a classe abstrata que complementa a estrutura dos outros dispositivos. As variáveis de instância presentes são:

- **`Boolean on`-** estado do dispositivo, ligado ou Desligado;
- **`Int id`-** número de identificação do aparelho;
- **`Double consume`-** consumo base de cada dispositivo.

Para além disso, é também nesta classe que existe a definição dos seguintes métodos abstractos:

- `Public abstract SmartDevice clone();`
- `Public abstract double consumption();`

Esta classe, sendo abstrata, vai servir como extensão para as classes que definem cada um dos diferentes dispositivos que vamos utilizar (SmartBulb, SmartCamera, SmartSpeaker). Devido a tal, todas estas classes serão obrigadas a implementar uma definição do cálculo do consumo, que varia de dispositivo para dispositivo. Eis as classes acima mencionadas:

2.4.1) SmartBulb

Na classe SmartBulb estão definidas as informações pedidas no enunciado que definem este *SmartDevice.* As variáveis de instância presentes são:

- **`Tones tone`-** tipo Tones criado por nós. Variável onde estão definidos os tons possíveis das lâmpadas inteligentes (podendo estas variar entre normal, frio e quente) juntamente com o consumo médio de casa tom.
- **`Int dimension`-** dimensão da lâmpada.

Nesta classe também está presente a fórmula necessária para o cálculo do consumo energético para as lâmpadas:

$$Caso\ on: 1 * Consumo\ Base * Consumo\ Médio\ do\ Tom$$

$$Caso\ off: 0 * Consumo\ Base * Consumo\ Médio\ do\ Tom$$

2.4.2) SmartCamera

Na classe SmartCamera estão definidas as informações pedidas no enunciado que definem este *SmartDevice.* As variáveis de instância presentes são:

- **Int resolutionX-** resolução da câmara no eixo X;
- **Int resolutionY-** resolução da câmara no eixo Y;
- **Int fileDim-** tamanho do ficheiro aonde são guardados os eventos.

Nesta classe também está presente a fórmula necessária para o cálculo do consumo energético para as câmaras:

$$Consumo\ Base + resolutionX * 0.001 + resolutionY * 0.001 \ \ + FileDim * 0.0001$$

2.4.3) SmartSpeaker

Na classe SmartSpeaker estão definidas as informações pedidas no enunciado que definem este *SmartDevice.* As variáveis de instância presentes são:

- **Int volume-** volume da coluna;
- **String RadioStation-** estação de rádio que está a tocar;
- **String brand-** marca da coluna.

Nesta classe também está presente a fórmula necessária para o cálculo do consumo energético para as colunas:

$$Caso\ on: 1 * ConsumoBase + Volume * 0.001$$

$$Caso\ off: 0 * ConsumoBase + Volume * 0.001$$

2.5) EstadoPrograma

A classe EstadoPrograma é onde estão presentes diversos processos cruciais. Para tal, estão definidas as seguintes variáveis da classe:

- **Map <String,Casa> casas-** mapeamento entre NIF e a casa;
- **Map <String, Fornecedor> fornecedores-** mapeamento entre o nome e o Fornecedor;
- **Double custoEnergia-** custo constante de energia (0.15);
- **Double imposto-** imposto base (0.06).

Estão definidas ainda as seguintes variáveis de instância:

- **Queue <Consumer<EstadoPrograma>> pedidos-** fila de espera que guarda os pedidos do utilizador para serem executados quando é avançado o tempo;
- **LocalDate data_atual-** data atual da simulação do programa.

É nesta classe onde estão definidos os métodos que realizam os requisitos definidos no enunciado. A sua implementação está explicada mais à frente.

2.6) Fatura

Classe onde estão definidas as informações que precisam de ser guardadas nas Faturas. Estão também definidos os métodos de obtenção dessas mesmas informações. Eis as variáveis de instância presentes nesta classe:

- **Int codigoFatura-** código atribuído a cada fatura para facilitar a sua procura;
- **String fornecedor-** empresa que fornecedora de energia;
- **String nifCliente-** número de identificação fiscal da respetiva casa a qual é emitida a fatura;

- **Double custo-** custo monetário do consumo no período abrangido pela fatura;
- **Double consumo-** consumo energético no período abrangido pela fatura;
- **LocalDate inicioPeriodo-** data inicial do período abrangido;
- **LocalDate fimPeriodo-** data final do período abrangido.

Está ainda definida também uma variável de classe:

- **Int next_codigoFatura (1) -** Código da próxima fatura, que vai ser incrementado sempre que uma nova é emitida.

2.7) Classes de Exceções

As seguintes classes implementadas descrevem o comportamento do programa quando este se depara com um erro nelas descrito:

- **AlreadyExistDeviceException-** comportamento do programa caso exista uma tentativa de inserir um dispositivo que já existe;
- CasaInexistenteException- comportamento do programa caso exista uma tentativa de aceder a uma casa que não existe;
- **DataInvalidaInexistenteException-** comportamento do programa caso o utilizador tentar introduzir uma data invalida;
- **DeviceInexistenteException-** comportamento do programa caso exista uma tentativa de aceder a um dispositivo que não existe;
- **ExisteCasaException-** comportamento do programa caso exista uma tentativa de inserir uma casa que já exista;
- **ExisteFornecedorException-** comportamento do programa caso exista uma tentativa de inserir um fornecedor que já exista;
- **FornecedorErradoException-** comportamento quando um é pedido a um fornecedor emitir uma fatura de uma casa que não possui contrato com este;
- **FornecedorInexistenteException-** comportamento do programa caso exista uma tentativa de aceder a um fornecedor que não existe;
- **RoomAlreadyExistsException-** comportamento do programa caso exista uma tentativa de adicionar uma divisória já existente na casa;
- **RoomInexistenteException-** comportamento do programa caso exista uma tentativa de aceder a uma divisória inexistente na casa.
- **TipoDeviceErradoException**- comportamento do programa caso o tipo a ser comparado não bate certo com o esperado;

3) Estatísticas do estado do programa

3.1) Ordenação dos maiores consumidores de energia

```
Public List <Casa> maiorConsumidorPeriodo(LocalDate inicio, LocalDate fim, int N){
        Comparator<Casa> comp= Comparator.comparingDouble
        (c-> c.consumoPeriodo(inicio,fim));
        Return this.casas.values()
                .stream()
```

```
            .sorted(comp.reversed())
            .limit(N)
            .map(Casa::clone())
            .collect(Collectors.toList());
```

Analisando o excerto, conseguimos aferir que realizamos uma comparação entre os valores de consumo de cada casa num certo período de tempo. Após isso, organizamos essa lista de forma decrescente. Aplicamos um limite nessa lista, para garantir que apenas analisamos os top N elementos. Obtemos a cópia das casas.

3.2) Comercializador com maior volume de faturação

```
Public String getFornecedorMaiorFaturacao(){
       Comparator<Map.Entry<String, Fornecedor>> comp= (f1,f2) ->{
              Double faturacao1= f1.getValue().faturacao();
              Double faturacao2= f2.getValue().faturacao();
              Return Double.compare (faturacao1, faturacao2);
       };
       Return this.fornecedores.entrySet().stream().max(comp).map(Map.Entry::getKey)
       .orElse("Não existe nenhum fornecedor");
```

Analisando o excerto de código, conseguimos analisar uma definição de uma comparação que analisa os parâmetros de faturação de dois fornecedores diferentes. Depois aplicamos essa comparação na lista de fornecedores pretendida. Como pretendemos obter o nome dos fornecedores, analisamos as Keys dos maps, garantindo assim que retornamos o nome.

3.3) Lista de Faturas por comercializador

```
Public List <Fatura> getFaturasFornecedor (String nome)
throws FornecedorInexistenteException {
       If(!this.fornecedores.containsKey(nome) throw new
       FornecedorInexistenteException("Não existe fornecedor: " + nome);
 Return
 this.fornecedores.get(nome).getFaturas().stream().map(Fatura::clone)
 .collect(Collectors.toList());
 }
```

Ao analisarmos o excerto de código, conseguimos aferir que introduzimos uma exceção que lida com a possibilidade de o utilizador tentar analisar um fornecedor inexistente. Após isso, simplesmente recolhemos as faturas dos fornecedores cujo nome coincide com o introduzido pelo usuário.

3.4) Casa que mais gastou naquele período

```
Public Optional <Casa> getCasaMaisGastadora() {
        Return this.casas.values().stream().max(Comparator.comparingDouble c->{
                List <Fatura> faturas = c.getFaturas();
                Return faturas.stream().mapToDouble (Fatura::GetConsumo).sum();
        } ).map(Casa::clone);
 }
```

Analisando o excerto do código que implementamos, conseguimos analisar que definimos um método de comparação que compara o gasto total entre duas casas, através da obtenção da soma de todos os custos presentes nas faturas totais de cada habitação. Depois disso retornamos a cópia da casa obtida com a maior despesa total.

A seguinte estatística foi implementada como método suplementar. Não pertence aos requisitos do enunciado.

3.5) Dispositivos mais usados

```
Public List <String> podiumDeviceMaisUsado(){
        Return this.casas.values()
          .stream()
          .flatMap(c-> c.getListDevices()
               .stream()
               .map(d-> d.getClass().getSimpleName()))
          .collect(Collectors.groupingBy(d -> d, Collectors.counting ()))
          .entrySet().stream()
          .sorted((e1, e2) -> (int) (e2.getValue() – e1.getValue()))
          .map(Map.Entry::getKey).limit(3).collect(Collectors.toList));
```

Esta estatística foi implementada com o objetivo de abranger as opções do usuário no que toca em análise do programa. Para tal, implementamos um flatMap que junta todas as listas de devices. De seguida, colecionamos os valores após uma contagem, organizando a lista de ordem crescente.

4) Diagrama de Classes

Figura 1- Diagrama de Classes do Programa, gerado pelo IntelIJ

5) Conclusão

Em termos gerais, acreditamos que fomos capazes de realizar um bom projeto e que respondemos de forma correta ao problema proposto, gerindo de forma eficiente as relações entre as classes e implementando métodos que facilitam o funcionamento geral do programa. Apesar disso, não fomos capazes de implementar uma forma de automatizar a simulação pois optamos por garantir que os restantes parâmetros do projeto estavam bem conseguidos ao invés de incluir esta simulação. Apesar disso, acreditamos que valeu a pena pois sentimos que, tal como mencionamos em cima, o projeto foi bem conseguido.