Projeto Prático

Programação Orientada a Objetos

Luís Pereira
A96681

Rui Oliveira
A95254

Tiago Pereira
A95104

Projeto desenvolvido para a Licenciatura em Engenharia Informática

Departamento de Informática
Universidade do Minho

maio 2022

Conteúdo

1 Introdução

O presente relatório é referente ao projeto prático, desenvolvido em Java, da Unidade Curricular de Programação Orientada a Objetos da Licenciatura em Engenharia Informática do Departamento da Informática da Escola de Engenharia da Universidade do Minho, para o ano de 2022.

O objetivo do projeto consistia em implementar uma aplicação que permitisse simular os custos energéticos relacionados com várias casas inteligentes. Entre os principais requisitos do mesmo estava a possibilidade de carregamento de dados a partir de ficheiros, incluindo automatização da simulação.

O projeto apresentado foi realizado pelo grupo número 10, constituído por Luís Manuel Fernandes Pereira (A96681), Rui Pedro Esteves Vasques Correia de Oliveira (A95254), e Tiago Miguel Moreira Bacelar Pereira (A95104).

Serão enunciadas as principais decisões tomadas pelo grupo na conceção do trabalho, analisando as suas consequências e eventuais vantagens e desvantagens; assim como será explicada em grande detalhe a arquitetura da aplicação desenvolvida.

2 Aplicação Desenvolvida

2.1 Interface

A aplicação desenvolvida tem uma interface pelo terminal. A interação com esta é feita através de comandos. Caso se queira saber todos os comandos que se encontram disponíveis, roda-se o comando SEND HELP (os comandos são *case insensitive*).

```
thehousingsimulator> SEND HELP
==========================================================================
==========================================================================
==                                                                      ==
==                  PROJETO DE POO: Grupo 10                            ==
==                           LEI                                        ==
==                           2022                                       ==
==========================================================================
==========================================================================

This project was developed by Luis Pereira (A96681), Rui Oliveira (A95254) and Tiago Pereira (A96681).

To use it, simply run one of the available command. Below is the list of all main commands available.

============================
= Commands:                =
============================

+ SUPPLIER LIST ALL: Lists all the suppliers in the simulation
+ SUPPLIER GET NAME=<name>: Lists all the suppliers with the given name
+ SUPPLIER GET ID=<id>: Returns the supplier with the given id
+ SUPPLIER UPDATE ID=<id>> NAME=<name> BASE_VALUE=<base value> TAX=<tax> FORMULA=<formula>: Updates a supplier
+ SUPPLIER ADD NAME=<name> BASE_VALUE=<base value> TAX=<tax> FORMULA=<formula>: Adds a supplier
+ SUPPLIER GET MOST SELLER: Gets the supplier with the highest sales volume


+ ENTITY LIST ALL: Lists all the entities of the simulation
+ ENTITY GET NAME=<name>: Lists all the entities with the given name
+ ENTITY DELETE ID=<id>>: Deletes the entity with the given id
```

Figura 1: Comando SEND HELP (excerto)

Os dados são mostrados em formato de tabela quando mais do que um objeto é apresentado, ou uma lista de atributos para um objeto singular, como mostrado nas figuras a baixo.

2.2 Funcionalidades

Todas as funcionalidades requeridas para a nota máxima de 20 valores foram implementadas. Assim, e para as listar, o programa desenvolvido suporta

- Inserção / atualização de dispositivos / casas / fornecedores na simulação, incluindo mudanças de fornecedores
- Avançar / recuar no tempo, emitir e consultar faturas
- Responder a estatísticas relativamente ao estado da simulação
- Guardar / carregar a simulação a partir de ficheiros, bem como carregar ficheiro de automatização da simulação

As funcionalidades de estatísticas podem ser feitas utilizando os seguintes comandos:

- *SUPPLIER GET MOST SELLER* - obtêm o melhor vendedor dos distribuidores de energia.
- *HOUSE SPENT MOST START=Nº END=Nº* - obtêm a casa que mais dinheiro gastou num intervalo
- *HOUSE GET BY ENERGY CONSUMPTION START=Nº END=Nº* - obtêm as casas ordenadas por consumo de energia.
- *RECEIPT GET SUPPLIER=Id* - obtêm todos os recibos emitidos por um distribuidor de energia.

A forma como estas foram implementadas é descrita na secção 3. A seguir mostram-se exemplos da utilização da aplicação.

Figura 2: Vista inicial do programa

```
thehousingsimulator> HOUSE GET ID=5
HOUSE INFO
============================
Id: 5
Name: HOUSE3
Owner's Name: OWNER3
Owner's NIF: 33
Energy Supplier Id: 2
Energy Supplier Name: SUPPLIER3
Energy Supplier base value: 9999.0
Energy Supplier tax: 0.9900000095367432
Energy Supplier formula: (Base+Consumption*999999999999)*(1+Tax)

+-----------------+
| Room  | DeviceId |
+-----------------+
| room3 | 8        |
| room3 | 17       |
| room3 | 29       |
| room1 | 13       |
| room1 | 22       |
| room1 | 34       |
| room2 | 9        |
| room2 | 18       |
| room2 | 30       |
+-----------------+
```

Figura 3: Vista singular de uma casa

```
thehousingsimulator> DEVICE LIST ALL
+-------------------------------------+
| Id | Name      | On   | House | Room  |
+-------------------------------------+
| 34 | speaker32 | true | 5     | room1 |
| 22 | cam32     | true | 5     | room1 |
| 13 | bulb32    | true | 5     | room1 |
| 32 | speaker23 | true | 4     | room1 |
| 20 | cam23     | true | 4     | room1 |
| 11 | bulb23    | true | 4     | room1 |
| 18 | cam21     | true | 5     | room2 |
| 9  | bulb21    | true | 5     | room2 |
| 30 | speaker21 | true | 5     | room2 |
| 19 | cam22     | true | 3     | room3 |
| 10 | bulb22    | true | 3     | room3 |
| 31 | speaker22 | true | 3     | room3 |
| 6  | bulb11    | true | 3     | room1 |
| 27 | speaker11 | true | 3     | room1 |
| 15 | cam11     | true | 3     | room1 |
| 16 | cam12     | true | 4     | room2 |
| 7  | bulb12    | true | 4     | room2 |
| 28 | speaker12 | true | 4     | room2 |
| 17 | cam13     | true | 5     | room3 |
| 8  | bulb13    | true | 5     | room3 |
| 29 | speaker13 | true | 5     | room3 |
| 35 | speaker33 | true | 3     | room2 |
| 23 | cam33     | true | 3     | room2 |
| 14 | bulb33    | true | 3     | room2 |
| 33 | speaker31 | true | 4     | room3 |
| 21 | cam31     | true | 4     | room3 |
| 12 | bulb31    | true | 4     | room3 |
+-------------------------------------+
```

Figura 4: Vista de tabela (dados carregados automaticamente)

3 Arquitetura

3.1 Model-View-Controller

A decisão mais impactante no projeto foi a escolha da utilização da arquitetura *Model-View-Controller*, de forma a separar a camada lógica da aplicação da camada de apresentação. A cada uma destas camadas foi atribuído um *package* único.

Dada a maior complexidade do diagrama de classes devido ao maior número destas proveniente da arquitetura MVC, apresentam-se em anexo 2 diagramas distintos: 1 corresponde ao diagrama do *núcleo* do trabalho, e outro corresponde ao diagrama do módulo usado para processamento de anotações (algo que será explicado mais adiante).

Em seguida, cada uma das camadas é explicada em profundidade.

3.2 Maven

Para facilitar o desenvolvimento do projeto, escolheu utilizar a ferramenta Maven para auxiliar na compilação e gestão das dependências da aplicação. Para compilar e rodar o projeto utilizam-se os comandos (a partir da diretoria base do projeto):

```
cd Annotations
mvn clean install
cd ..
mvn clean install
mvn exec:java -Dexec.mainClass="com.housingsimulator.Main"
```

4 Modelo

4.1 Entidades

Para facilitar a identificação dos várias intervenientes do problema, foi criado um tipo genérico, que os engloba e identifica unicamente. Esse tipo genérico corresponde à classe AbstractEntity. A identificação das entidades é feita através de dois atributos: por um lado o nome, que é uma String cujo objetivo é facilitar a identificação da entidade por parte do utilizador da aplicação, e um identificador inteiro (id), que é único para cada entidade.

Para garantir a unicidade dos identificadores, foi criada a variável de classe currentIndex, que corresponde ao identificador a atribuir da próxima vez que um construtor seja invocado. Quando é criada uma nova entidade, esta variável é incrementada.

Desta forma, garante-se uma indexação única de cada um dos intervenientes no problema, que viria a ser útil sobretudo na camada de apresentação e para a interpretação de comandos do utilizador, permitindo a generalização de muitos deles.

4.2 Dispositivos

Todos os dispositivos são entidades do programa e, genericamente, correspondem a objetos da classe SmartDevice, que implementa a funcionalidade comum a todos os dispositivos, nomeadamente o ato de os ligar / desligar; obrigando todas as suas subclasses a definir o método energyOutput, que calcula o consumo energético do dispositivo por unidade de tempo.

Esta abordagem permite uma maior flexibilidade na criação de fórmulas de consumo de cada dispositivo. Existem vários tipos diferentes de dispositivos (facilmente se criam mais, dada a modularidade da arquitetura).

Um aspeto a destacar é a arquitetura das colunas inteligentes (*SmartSpeaker*). Estes objetos têm um atributo, a sua marca (*SpeakerBrand*), que determina, em parte, o seu consumo energético. Para este atributo foi utilizada uma estratégia de composição, devido ao facto de alterações de consumo de uma determinada marca não deverem afetar as colunas já criadas. A título de exemplo, e para ajudar a compreender esta decisão, se a marca A de colunas decide alterar as colunas que produz para gastarem 100W em vez de 50W, todas as colunas produzidas antes desta alteração continuam, naturalmente, a consumir 50W.

4.3 Fornecedores de energia

Os fornecedores de energia correspondem a instâncias da classe *EnergySupplier*. Esta possui variáveis de instância que caracterizam o fornecedor, nomeadamente o valor base do custo da eletricidade e o imposto aplicado. Estes atributos corresponderem a variáveis de instância dá mais flexibilidade ao utilizador, uma vez que se se tratassem de variáveis de classe o seu valor seria o mesmo para todos os fornecedores, e esta solução permite ter fornecedores com valores distintos. Naturalmente, e pelo menos em Portugal, todos os fornecedores aplicam a mesma taxa de imposto aos seus clientes.

4.3.1 Cálculo de custo

A fórmula para o cálculo do preço a pagar depende do fornecedor. Deste modo, não é viável ter um método único que faça esse cálculo - no extremo, poderia haver vários métodos diferentes que corresponderiam a todas as fórmulas admissíveis, das quais cada instância escolheria qual utilizar.

Por esse motivo, para o cálculo do custo, permitiu-se que o fornecedor recebese a fórmula matemática que utiliza para o cálculo dos preços. Para a interpretação e cálculo do valor a partir dessa fórmula recorreu-se à biblioteca EvalEx. A fórmula é guardada no fornecedor sobre a forma de uma *String*, para permitir que esta seja serializada automaticamente ao gravar os dados do fornecedor para ficheiro; dado que a classe Expression da biblioteca não implementa a interface Serializable .

4.4 Casas

As casas foram implementadas na classe SmartHouse. Esta classe guarda os dispositivos que pertencem à casa por uma estratégia de composição, uma vez que os dispositivos pertencem à casa, indexados pelo seu identificador de entidade para mais fácil acesso.

A casa também tem um conjunto de divisões, identificadas pelo seu nome. Para cada divisão, é guardado o conjunto dos identificadores de todos os dispositivos que se encontram nessa mesma divisão.

Finalmente, existem os dados do dono da casa, e o fornecedor de energia afeto a essa casa, guardado segundo uma estratégia de agregação, pois alterações nos custos associados a um fornecedor devem propagar-se a todos os seus clientes.

4.5 Faturas

A classe Receipt implementa as faturas emitidas pelos fornecedores de energia. Uma fatura guarda o fornecedor de energia que a emitiu e a casa a que se refere numa estratégia de composição, dado que uma fatura deve capturar o estado destes no momento em que foi emitida, pelo que alterações futuras a estas entidades não devem estar aqui refletidas.

Para além disso, uma fatura tem também o intervalo de faturação (guardado como dois doubles, que é o tipo de dados associado a tempo na aplicação), assim como uma associação entre dispositivos e o seu consumo energético - dispositivos estes indexados por uma String igual a "<id do dispositivo>:<nome do dispositivo>", de forma a dar ao utilizador informação mais detalhada sobre o seu consumo. Naturalmente, o preço total da fatura encontra-se também guardado numa variável de instância.

4.6 Simulador

A simulação das várias casas ao longo do tempo é gerida pela classe Simulator. Esta classe, em vez de atualizar incrementalmente o estado inicial através de regras pré-definidas até ao momento alvo, recebe diretamente o método que retorna qualquer estado futuro. Desta forma, é possível obter o estado da simulação em qualquer momento futuro, sem a necessidade de calcular todos os momentos intermédios. Esta abordagem também permite tratar o tempo como uma medida contínua (usando o tipo de dados double), em vez de discreta (int).

De forma a modificar a simulação, perturbando a evolução descrita pela função inicial, são introduzidos eventos, objetos da classe Event. Cada evento funciona como um novo método inicial para a porção de tempo a seguir ao momento em que o evento é introduzido (chamemos a este método de *derivador*). Alguns exemplos de eventos são trocar o estado de uma lâmpada ou redefinir o nome de uma casa. Estas ações podem ou não depender do estado imediatamente anterior da simulação. No entanto, para evitar percorrer recursivamente todos os eventos anteriores até ao primeiro de cada vez que queremos o estado num certo momento, o passo de inicialização do evento, em que este tem

acesso ao estado imediatamente anterior, e a geração de um novo estado pelo método derivador, são feitos em separado, garantindo que cada evento guarda em variáveis de instância a informação que necessita do estado anterior da simulação.

Com esta definição de evento, não temos de nos limitar apenas a adicionar eventos no final da simulação. Podemos adicionar um evento entre quaisquer outros dois eventos, desde que os eventos seguintes sejam reinicializados. De forma a evitar reinicializações desnecessárias, podemos ainda tornar a inicialização preguiçosa.

Com o método descrito até agora, cada evento que queira atualizar apenas uma pequena parte do estado (por exemplo, acender uma das várias lâmpadas de uma de várias casas) tem de guardar o estado na totalidade para o poder retornar quando o seu método derivador for chamado, apesar da maior parte do estado (todos os outras lâmpadas e casas) permanecer inalterada. A solução encontrada foi dividir o estado da simulação em várias entidades independentes, que possam ser atualizadas independentemente. Desta forma, um evento pode afetar apenas algumas das entidades da simulação, diminuindo o uso de memória e o tempo de processamento da simulação. Cada entidade é referida por um índice inteiro, podendo ser referenciada noutros lugares da simulação por esse índice.

4.7 Simulação

Tendo em conta as definições apresentadas anteriormente, resta-nos definir qual deve ser o estado da simulação e como dividir esse estado em entidades independentes.

O estado da simulação é representado pela classe Simulation. Esta classe contém uma lista de fornecedores, uma lista de marcas de SmartSpeakers e a lista das casas, cada qual dividida em divisões e SmartDevices. Como cada casa contém vários SmartDevices, não os podemos separar em entidades separadas. Por esta razão, cada casa é uma entidade da simulação, e não os seus SmartDevices. A lista de marcas e a lista de fornecedores podem ser divididas nos seus elementos, pois estes são independentes entre si e da restante simulação, sendo cada elemento uma entidade da simulação.

4.8 Modelo

A classe Model serve como agregadora das duas classes descritas anteriormente. Pode ser, por isso, vista como o *facade* do modelo da aplicação, que será passado para todos os controladores da aplicação.

Para além disso, esta classe define também métodos para controlar o tempo atual da simulação, bem como para a obtenção do estado da mesma em qualquer instante. A classe guarda ainda todas as faturas emitidas, que são calculadas para cada casa de cada vez que é feito um avanço no tempo da simulação. Caso o tempo seja recuado, as faturas futuras são apagadas e as faturas que foram interrompidas, caso existam, são recalculadas para o novo intervalo de tempo.

4.9 Ficheiros de estado e automatização de comandos

Ao trabalhar com os sistemas explicados anteriormente, o grupo achou que seria proveitoso implementar formas de automatizar a execução de vários comandos no programa, bem como carregar automaticamente uma simulação já criada. Com este propósito, foram criados dois mecanismos: a serialização e os scripts.

Os scripts são ficheiros de texto que contém uma lista de comandos executados sequencialmente quando é chamado o comando SCRIPT EXEC <filename>. No entanto, o grupo achou a sua utilidade algo limitada pelo facto de as entidades na simulação serem referenciadas por id, que, sendo atribuído automaticamente por ordem de criação da entidade, implica que os scripts precisem de algum grau de conhecimento prévio das entidades na simulação. Isto poderia ser resolvido utilizando parâmetros no script ou permitindo o encadeamento de comandos, dando o output de um ao input de outro, ideias que o grupo certamente perseguiria com um mais relaxado prazo de entrega.

4.10 Exceções

Durante o desenvolvimento do modelo, houve necessidade da criação de exceções para assinalar erros durante a execução do programa. Estes erros podem dividir-se em dois tipos: aqueles que devem obrigatoriamente ser tratados, por corresponderem a situações anómalas que não conseguem ser previstas ou evitadas pelo programador; e aqueles que não necessitam de ser tratados, por apenas serem provocadas em situações previsíveis.

A título de exemplo, um erro a processar uma fórmula de um fornecedor de energia não é um erro previsível ou até evitável facilmente, pelo que a exceção correspondente tem de ser tratada por quem a recebe. Por outro lado, se se tentar aceder a uma entidade através de um identificador não existente, gera-se uma exceção; no entanto, não faz sentido ser obrigatório tratá-la, pois há casos onde o programador tem a certeza que não será lançada e, nesses casos, se for, é preferível deixar passar o controlo à *JVM*, por se tratar de uma situação indefinida programaticamente.

Assim sendo, todas as exceções que devem ser tratadas herdam da classe IOException, e as outras da classe RuntimeException, uma vez que o Java não obriga a tratar as exceções deste tipo.

5 Vistas

Para o projeto foram desenvolvidas várias vistas, uma por cada tipo de dados a mostrar no ecrã .

As vistas correspondem normalmente a 4 métodos: 2 deles genéricos e aplicáveis a todas, que correspondem a métodos que mostram ao utilizador que houve um erro ao processar um comando, e mostrar um aviso (por exemplo, tratar uma lâmpada como sendo uma câmara). Os restantes dois métodos imprimem para o ecrã o objeto que lhes é específico (uma casa, uma lâmpada,

.etc) e imprimem uma lista (para haver a noção de ordem de impressão) de objetos.

Para facilitar a impressão de dados em forma de tabela, criou-se a classe TablePrinter, que possui um método com número variável de argumentos que imprime uma tabela para o ecrã a partir das suas colunas, que é utilizada em praticamente todas as vistas. Também possui um método auxiliar que converte listas em listas de objetos, de forma a poder generalizar os tipos de dados que podem ser impressos no ecrã.

6 Controladores

Os controladores desenvolvidos serviram para ligar ambas as interfaces desenvolvidos ao modelo lógico e aplicacional. Foram desenvolvidos vários controladores distintos, um para cada entidade relevante na simulação, bem como outros controladores relacionados com carregamento de ficheiros e gestão de tempo da simulação. Assim sendo, os controladores expõem a funcionalidade da aplicação às vistas.

A forma como os controladores foram usados e ligados uns aos outros é detalhada de seguida.

6.1 Processamento de anotações

Para que o processamento dos comandos seja efetuado de forma eficiente, e para reduzir o esforço necessário para introduzir novos comandos à aplicação, foram desenvolvidas anotações, que em tempo de compilação geram uma classe - o MainController - que é responsável por gerir a execução dos comandos inseridos pelo utilizador [1] e pelo carregamento de todos os controladores necessários. O processsador de anotações desenvolvido encontra-se no package annotations.

De um modo geral, todos os controladores que pretendam ser carregados no início de execução devem ser anotados com @API. Quando o processador deteta esta classe, adiciona-a ao conjunto de controladores carregados, inicializando-o (todos os controladores devem herdar de uma classe Controller comum) no método gerado automaticamente loadControllers.

Para cada comando que se pretenda adicionar, o método que o implementa deve estar anotado com @Endpoint. Esta anotação tem um parâmetro, denominado regex, que, como o próprio nome indica, corresponde à expressão regular que determina o formato do comando a adicionar. Quando o processador encontra esta anotação, gera o código que permite verificar se o comando inserido corresponde a esta instrução e, caso afirmativo, faz a sua chamada com os argumentos convertidos para o tipo correto [2].

[1] Naturalmente, isto só é válido para a interface por terminal.

[2] De notar que em métodos anotados como @Endpoint não é permitido receber parâmetros de tipos primitivos, como int, por limitações da API de processamento de anotações. Os objetos usados devem ter um construtor que recebe uma só String

Todos estes métodos devem ou retornar uma String ou não retornar nada nem lançar exceções. Caso o retorno seja uma String, o processamento de comandos é feito recursivamente. Isto foi utilizado para implementação dos *scripts*.

7 Testes unitários

De forma a assegurar o correto funcionamento do projeto, com ênfase no modelo da aplicação, foram desenvolvidos testes unitários em paralelo com o desenvolvido do código do programa. O objetivo destes testes é, por um lado, ajudar a garantir a correção do programa, motivo pelo qual o grupo tentou garantir que a cobertura dos testes fosse praticamente total, objetivo que foi cumprido no *package* do modelo. Por outro lado, também foram criados testes unitários para avaliar se as decisões arquiteturais relativas a composição/agregação de classes foram respeitadas.

Tal como abordado nas aulas práticas, utilizou-se a biblioteca JUnit para o desenvolvimento dos testes. Estes encontram-se na diretoria *tests/*.

8 Conclusão

Assim sendo, o grupo considera que cumpriu com todos os requisitos exigidos para o cumprimento do projeto, bem como adicionou mais funcionalidade não exigida, de forma a destacar o projeto dos restantes.

Naturalmente, e como em qualquer projeto desenvolvido, há sempre aspetos que poderiam ser melhorados. Neste caso, e a título de exemplo, o grupo poderia

- Criar uma separação ainda maior entre tipos de dados e lógica de negócios, criando um conjunto de interfaces representativas do modelo de dados da aplicação, que poderiam ser passadas às vistas, facilitando a chamada dos seus métodos
- Desenvolvimento de um mecanismo de interação com o utilizador mais evoluído (interface gráfica). O grupo tentou implementar esta interface, mas o prazo de entrega não permitiu desenvolver a funcionalidade com a qualidade pelo grupo exigida para ser integrada com o restante trabalho
- Permitir exportar os resultados da simulação para um formato de documento *standard*, como PDF
- Permitir a comparação de diferentes cenários na simulação, ou seja, criar diferentes ramos de eventos que pudessem ser comparados para ver qual o mais eficiente
- Adicionar completar automaticamente comandos e permitir rodar os comandos anteriores (comportamento semelhante a uma *shell bash*

Para concluir, o projeto foi bem sucedido, todos os elementos do grupo tiveram uma atitude bastante positiva para com este e perante os normais problemas que surgem em engenharia de *software*, o que é visível na qualidade da aplicação desenvolvida.

9 Anexos

Aqui apresentam-se os diagramas de classes do projeto principal e do módulo de anotações. Uma cópia destes ficheiros foi enviada juntamente com o projeto, incluindo versão em texto e em imagem (formato .svg). Os diagramas foram gerados com o Mermaid.

9.1 Diagrama de classes

9.1.1 Projeto

Confrontar diagramas/projeto na submissão.

9.1.2 Anotações

```
classDiagram
direction BT
class API
class Endpoint {
  + regex() String
}
class EndpointProcessor {
  + EndpointProcessor()
  + init(ProcessingEnvironment) void
  + processClasses(RoundEnvironment) String
  + process(Set~TypeElement~, RoundEnvironment) boolean
  + processMethods(RoundEnvironment) String
  + methodCall(Element) String
  + error(Element, String, Object[]?) void
  + SourceVersion supportedSourceVersion
  + Set~String~ supportedAnnotationTypes
}
class IllegalAnnotationException {
  + IllegalAnnotationException(String)
}
class Main {
  + Main()
  + main(String[]?) void
}
```