Universidade do Minho

Programação Orientada aos Objetos

Trabalho Prático

Grupo 5

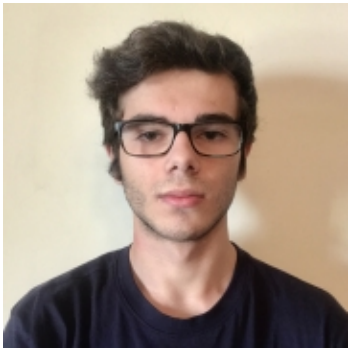
Ano letivo 2021/2022

a96106, Miguel Silva Pinto

a97755, Orlando José da Cunha Palmeira

a97613, Pedro Miguel Castilho Martins

Miguel Pinto



Orlando Palmeira

Pedro Martins

Introdução

Este relatório descreve a elaboração de um sistema que monitoriza e regista informação sobre o consumo energético das habitações de uma comunidade.
Em cada habitação existem dispositivos inteligentes e um contrato com um fornecedor de energia. Com estas informações, o programa é capaz de fazer a gestão dos consumos das casas e calcular todos os custos associados ao consumo.

Arquitetura e estrutura de classes

- A classe *SmartDevice*

```
public abstract class SmartDevice implements Serializable {

    private String id;
    private boolean on;
    private double totalConsumption;
    private LocalDateTime last_change;
```

Figura 1 - Atributos (estrutura) da classe *SmartDevice*

Esta é uma classe abstrata que representa o comportamento genérico dos dispositivos inteligentes, permitindo ligar/desligar um dispositivo, atualizar o seu consumo total e obter informação do seu consumo diário.
Descrição dos atributos da classe:

- *id* - Identificador único de um dispositivo.
- *on* - Valor booleano que indica se um dispositivo se encontra ligado ou desligado.
- *totalConsumption* - Indica o consumo total (em kWh) de um dispositivo.
- *last_change* - Indica a data e hora da última vez que um dispositivo foi ligado, desligado ou quando o seu consumo total foi atualizado.

De forma a permitir que o programa possa evoluir sem grandes constrangimentos e sem necessidade de adição e/ou alteração do código, decidimos que *SmartDevice* seria uma classe abstrata de modo a permitir a compatibilidade entre as suas subclasses (*SmartBulb*, *SmartSpeaker* e *SmartCamera*) e a criação de novas classes que representem novos dispositivos (por exemplo, uma nova classe *SmartTV*) sem interferir, por exemplo, no código da *CasaInteligente* que tem uma estrutura de dados para armazenar os *SmartDevice's.*
Também decidimos guardar em cada dispositivo a data e hora em que o seu estado foi alterado (isto é, quando foi ligado/desligado, quando se mudou a tonalidade de uma lâmpada ou quando se aumenta/diminui o volume de um *speaker*) uma vez que facilita muito o cálculo do consumo de cada dispositivo dado que sabemos o tempo que ele esteve naquele estado desde que lhe foi feita uma alteração na sua configuração.

- A classe *SmartBulb*

```
public class SmartBulb extends SmartDevice {
    public static final int WARM = 2;
    public static final int NEUTRAL = 1;
    public static final int COLD = 0;

    private int tone;
    private double dimension;
```

Figura 2 - Atributos (estrutura) da classe *SmartBulb*

Esta é uma subclasse da classe *SmartDevice* e especializa o comportamento de um dispositivo para uma lâmpada inteligente, permitindo saber e alterar sua tonalidade bem como saber qual é a sua dimensão.
Descrição dos atributos da classe:

- Constantes *WARM, NEUTRAL* e COLD: valores numéricos que identificam as tonalidades que uma lâmpada pode ter.
- *tone*: Indica a tonalidade atual da lâmpada.
- *dimension*: Indica a dimensão da lâmpada.

- A classe *SmartSpeaker*

```
public class SmartSpeaker extends SmartDevice {
    public static final int MAX = 20;

    private int volume;
    private String channel;
    private String marca;
```

Figura 3 - Atributos (estrutura) da classe *SmartSpeaker*

Esta é uma subclasse da classe *SmartDevice* e especializa o comportamento de um dispositivo para um *speaker* inteligente, permitindo saber e alterar o seu volume do som, o seu canal bem como saber qual é a sua marca.
Descrição dos atributos da classe:

- Constante *MAX*: indica o volume máximo a que um speaker pode tocar.
- *volume*: Indica o volume atual a que o *speaker* está a tocar.
- *channel*: Indica o canal em que o *speaker* está sintonizado.
- *marca*: Indica a marca do *speaker*.

- A classe *SmartCamera*

```
public class SmartCamera extends SmartDevice{
    private int resX;
    private int resY;
    private double sizeOfFile;
```

Figura 4 - Atributos (estrutura) da classe *SmartSpeaker*

Esta é uma subclasse da classe *SmartDevice* e especializa o comportamento de um dispositivo para uma câmara inteligente, permitindo saber a sua resolução da imagem bem como o tamanho de um ficheiro gerado.
Descrição dos atributos da classe:

- *resX*: indica a resolução horizontal da imagem.
- *resY*: indica a resolução vertical da imagem.
- *sizeOfFile:* indica o tamanho do ficheiro gerado pela câmara.

- A classe *CasaInteligente*

```
public class CasaInteligente implements Serializable {

    private String morada;
    private Map<String, SmartDevice> devices;
    private Map<String, Set<String>> locations;
    private Pessoa proprietario;
    private String fornecedor;
```

Figura 5 - Atributos (estrutura) da classe *CasaInteligente*

A classe *CasaInteligente* agrega vários dispositivos e divide-os por várias repartições. Uma casa permite ligar e desligar dispositivos bem como alterar algumas das suas configurações (o canal de um *speaker*, por exemplo) e também permite obter e atualizar o consumo total de todos os seus dispositivos.
Descrição dos atributos da classe:

- *morada*: Indica a morada desta casa
- *devices*: Um *Map* que armazena os dispositivos indexado pelos id's dos mesmos.
- *locations*: Um *Map* que armazena conjuntos de id's de dispositivos indexado pelo nome das repartições onde os dispositivos se encontram.
- *proprietario*: Um objeto da classe *Pessoa* (explicada posteriormente) que contém informações do proprietário da casa.
- *fornecedor:* Indica o nome do fornecedor da casa.

A *CasaInteligente* tem apenas o nome do seu fornecedor e não um objeto do tipo *EnergyProvider* uma vez que, para cumprir as regras do encapsulamento, teríamos de clonar o fornecedor sempre que se criava uma casa e isso traria problemas quando, por exemplo, na simulação se alterar o preço de um fornecedor e todas as casas com esse fornecedor tenham de ter conhecimento dessa alteração.

- A classe *Pessoa*

```
public class Pessoa implements Serializable {
    private String nome;
    private int nif;
```

Figura 6 - Atributos (estrutura) da classe *Pessoa*

A classe *Pessoa* serve para representar os proprietários das casas.
Descrição dos atributos da classe:

- *nome*: Indica o nome de uma pessoa.
- *nif*: Indica o nif de uma pessoa.

- A classe *EnergyProvider*

```
public class EnergyProvider implements Serializable {
    private String name;
    private double price_kwh;
    private double tax;
```

Figura 7 - Atributos (estrutura) da classe *EnergyProvider*

A classe *EnergyProvider* representa um fornecedor de energia. Esta permite calcular os custos dos consumos das casas bem como gerar as respetivas faturas.
Descrição dos atributos da classe:

- *name*: Indica o nome do fornecedor.
- *price_kwh*: Indica o preço aplicado por um fornecedor a cada kWh de energia gasto.
- *tax*: Indica o fator multiplicativo dos impostos de um fornecedor.

- A classe *Fatura*

```
public class Fatura implements Serializable {
    private static int last_id = 0;

    private EnergyProvider provider;
    private CasaInteligente casa;
    private LocalDateTime start;
    private LocalDateTime end;
    private double montante;
    private int id;
```

Figura 8 - Atributos (estrutura) da classe *Fatura*

A classe *Fatura* armazena informações sobre a cobrança de um fornecedor de energia no consumo de uma certa casa.
Descrição dos atributos da classe:

- *last_id*: Esta variável da classe indica o último id atribuído na criação de uma instância de *Fatura*.
- *provider*: Armazena o fornecedor que emitiu a fatura.
- *casa*: Armazena a casa à qual a fatura foi atribuída.
- *start* e *end*: Indicam a data de início e fim do período de faturação, respetivamente.
- *montante*: Valor cobrado na fatura.
- *id*: Id atribuído à fatura.

- A classe *Simulator*

```
public class Simulator implements Serializable{

    private int last_house_id;
    private Map<Integer, CasaInteligente> houses;
    private Map<String,EnergyProvider> energyProviders;
    private Map<String, List<Fatura>> billsPerProvider;
    private Map<String, Double> profitPerProvider;
    private List<CasaInteligente> consumptionOrder;
```

Figura 9 - Atributos (estrutura) da classe *Simulator*

A classe *Simulator* é a mais importante do programa. É ela que agrega todas informações, executa todas as alterações que o utilizador pretenda (ligar/desligar dispositivos nas casas, alterar preços/impostos dos fornecedores, emitir faturas, etc…) bem como efetua todos os cálculos relativos a uma simulação.
Descrição dos atributos da classe:

- *last_house_id*: Variável utilizada para atribuir id's automaticamente às casas que se inserem na simulação.
- *houses*: *Map* de casas indexado automaticamente pela variável referida anteriormente.
- *energyProviders*: Map de fornecedores indexado pelos nomes dos mesmos.
- *billsPerProvider*: Map de faturas emitidas por cada fornecedor indexado pelos nomes dos fornecedores (o seu valor é obtido apenas no fim da execução da simulação).
- *profitPerProvider*: Map que armazena o lucro obtido por cada fornecedor indexado pelos nomes dos fornecedores (o seu valor é obtido apenas no fim da execução da simulação).
- *consumptionOrder*: Lista de casas ordenada de forma decrescente segundo o consumo total de cada uma no fim da simulação (o seu valor é obtido apenas no fim da execução da simulação).

As variáveis de instância *billsPerProvider, profitPerProvider e consumptionOrder* existem apenas para aumentar a performance na obtenção dos seus resultados caso estes sejam solicitados mais do que uma vez. Por exemplo, nas estatísticas da simulação queremos saber a ordem de consumidores e o maior consumidor. Estes resultados podem ser obtidos através da lista em *consumptionOrder* com apenas um cálculo.
Esta classe poderia ter como variável de instância uma coleção de comandos. No entanto, optámos por não fazê-lo uma vez que achamos bem que um *Simulator* possa executar uma simulação mais do que uma vez e com comandos diferentes.
Assim, o *Simulator* recebe a coleção de comandos como argumento na função *startSimulation*.

- A classe *Command*

```
public class Command{

    private LocalDateTime executionDateTime;
    private Consumer<Simulator> command;
    private boolean flag;
```

Figura 10 - Atributos (estrutura) da classe *Command*

A classe *Command* representa um tratamento a ser aplicado à simulação. Pode representar, por exemplo, um pedido para desligar um dispositivo numa casa.
Descrição dos atributos da classe:

- *executionDateTime*: Indica a data e hora em que o comando vai ser executado.
- *command*: Armazena um *consumer* que é aplicado à simulação.
- *flag*: Valor booleano que indica se um comando pode ser executado durante a simulação (por exemplo, ligar um dispositivo) ou apenas no fim da simulação (por exemplo, alterar o preço aplicado por um fornecedor).

Por forma a facilitar a criação de novos comandos sem grandes impactos no código do programa, optámos por implementar esta classe com um *Consumer<Simulator>* que dá a esta classe uma configuração muito genérica.
Assim, qualquer expressão *lambda* que esteja na variável command pode ser executada sobre um *Simulator* e, deste modo, podemos ter uma infinidade de comandos diferentes.

- A classe *Interface*

```
public class Interface {

    private Simulator sim;
```

Figura 11 - Atributos (estrutura) da classe *Interface*

A classe *Interface* tem a responsabilidade de implementar a componente visual do programa (menus), comunicar com o utilizador bem como enviar os comandos à simulação.
Descrição dos atributos da classe:

- *sim*: A simulação utilizada no programa.

A *Interface* armazena uma simulação uma vez que a maioria dos seus métodos necessita de informações constantes na simulação (por exemplo, a existência de casas, existência de fornecedores, etc…).

- A classe *InvalidCommandException*

```
public class InvalidCommandException extends Exception {
    public InvalidCommandException(String message) {
        super(message);
    }
}
```

Figura 12 - Atributos (estrutura) da classe InvalidCommandException

A classe *InvalidCommandException* serve para sinalizar quando se está a tentar criar um comando inválido.
Esta exceção é essencialmente utilizada no *parsing* de strings que representam comandos e essas strings têm um formato incorreto a representar algum comando.

Diagrama de Classes (o diagrama estará disponível na pasta do trabalho caso não seja visível neste PDF)

Descrição de Funcionalidades

Menu Principal

No arranque do programa, o utilizador encontra o seguinte menu:

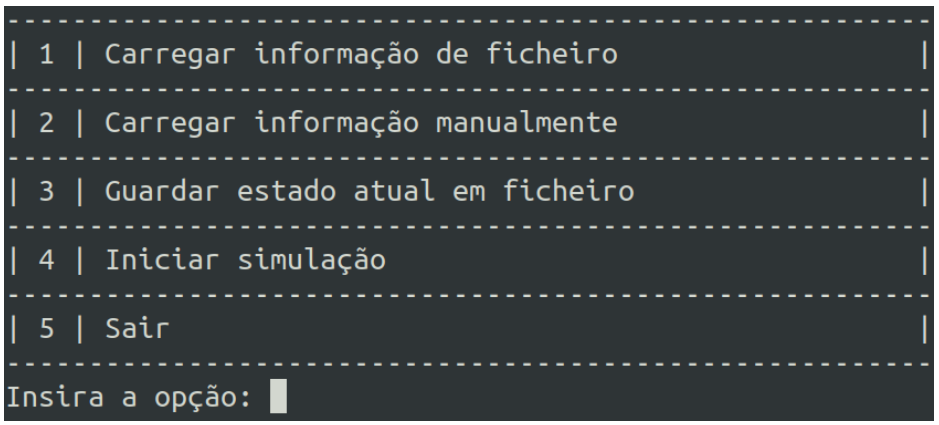


Figura 13 - Menu principal

Descrição de cada uma das opções:

- A primeira opção permite ao utilizador inserir o caminho para um ficheiro de objetos que armazene um estado do programa que tenha sido previamente criado.
- A segunda opção serve para o utilizador inserir as casas, dispositivos, fornecedores, etc… manualmente pelo teclado (funcionalidade explicada a seguir).
- A terceira opção serve para o utilizador guardar o estado atual que está carregado em memória num ficheiro de objetos para uso futuro.
- A quarta opção permite ao utilizador iniciar a simulação (funcionalidade explicada a seguir)
- A quinta opção serve apenas para encerrar o programa.

Carregamento manual de informação

Quando o utilizador seleciona a segunda opção do menu principal, encontra o seguinte menu:

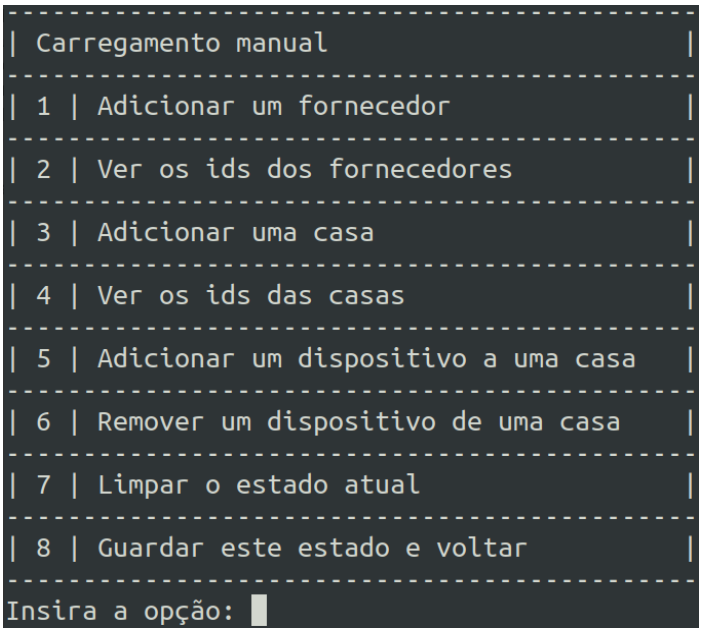


Figura 14 - Menu de carregamento manual de informação

Descrição de cada uma das opções:

- A primeira opção serve para adicionar um novo fornecedor e, quando é selecionada, o programa vai solicitar o nome do fornecedor e o preço/imposto por ele aplicado.
- A segunda opção permite ao utilizador consultar os id's dos fornecedores de energia já criados no programa (pode haver a necessidade de o utilizador saber que informação já se encontra inserida no programa).
- A terceira opção serve para adicionar uma nova casa e, quando é selecionada, o programa irá solicitar algumas informações, tais como: nome e NIF do seu proprietário, qual o seu fornecedor, qual a morada e também irá solicitar a criação de dispositivos a serem colocados na casa e nas respectivas repartições.
- A quarta opção serve para o utilizador saber os id's das casas já criadas.
- A quinta opção permite ao utilizador adicionar um dispositivo numa casa mesmo após a criação da mesma caso tenha mudado de ideias e pretenda ter mais dispositivos naquela casa (o modo como o programa interage para a criação de dispositivos é descrito a seguir). Quando selecionada, o programa irá solicitar algumas informações sobre o dispositivo, tais como: o seu tipo (*SmartBulb*, *SmartCamera*, *SmartSpeaker*), a repartição onde será colocado (se a mesma não existir, o programa cria-a automaticamente) e algumas informações específicas dependendo do tipo do dispositivo (por exemplo, a tonalidade quando no caso das lâmpadas, ou o volume de som no caso dos *speakers*).
- A sexta opção serve para remover um dispositivo de uma casa já criada caso o utilizador assim pretenda.
- A sétima opção serve para eliminar todos os dados registados no programa.
- A oitava opção guarda a informação que o utilizador carregou na memória e volta ao menu principal.

Criação de novos dispositivos

Quando um utilizador pretende adicionar um dispositivo numa casa, irá ser apresentado o seguinte menu:

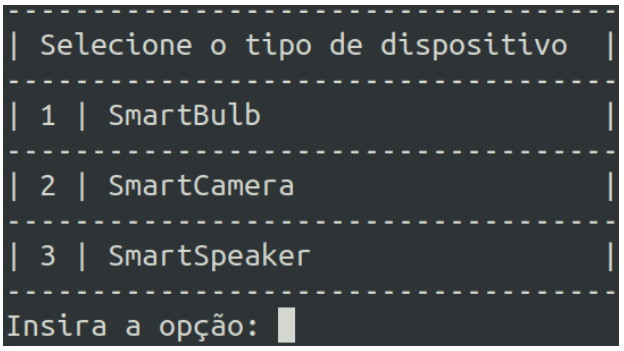


Figura 15 - Menu de criação de um dispositivo

Descrição de cada uma das opções:

- A primeira opção permite criar uma lâmpada e, quando é selecionada, o programa irá pedir o id a atribuir, se a lâmpada inicia ligada quando é criada, a tonalidade e a dimensão (em cm).
- A segunda opção permite criar uma câmara e, quando é selecionada, o programa irá pedir o id, se a câmara inicia ligada quando é criada, a sua resolução e o tamanho do ficheiro que gera.
- A terceira opção serve para criar um *speaker* e, quando é selecionada, o programa irá solicitar o seu id, se o *speaker* se encontra ligado quando é criado, o volume a que está a funcionar, o canal em que está sintonizado e a sua marca.

Execução da simulação

Quando o utilizador já inseriu todos os dados necessários e a simulação está pronta, quando é selecionada a quarta opção do menu principal, irá aparecer o seguinte menu:

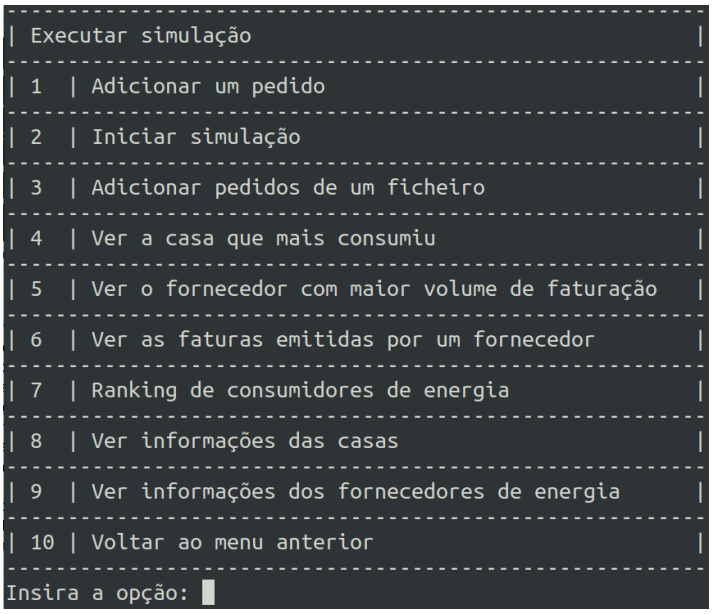


Figura 16 - Menu de execução da simulação

Descrição de cada uma das opções:

- A primeira opção serve para o utilizador criar uma alteração ao estado da simulação durante a sua execução (por exemplo, mudar o fornecedor de uma casa, ligar/desligar dispositivos, alterar preços de um fornecedor, etc…; esta funcionalidade é descrita mais detalhadamente a seguir).
- A segunda opção vai executar uma simulação, solicitando ao utilizador a data e hora de início e fim da simulação. Após isso, a execução é efetuada e os resultados estão prontos para serem visualizados nas opções seguintes.
- A terceira opção permite ao utilizador automatizar a simulação permitindo carregar para o programa um ficheiro de texto com comandos que permitem fazer alterações ao estado da simulação (esta funcionalidade é descrita com mais detalhe a seguir).
- A quarta opção permite ao utilizador saber qual a casa que mais consumiu energia na última simulação executada.
- A quinta opção permite ao utilizador saber qual o fornecedor de energia que teve mais lucro na última simulação executada.
- A sexta opção permite saber quais foram as faturas emitidas por um certo fornecedor.
- A sétima opção apresenta uma ordenação decrescente das casas segundo o seu consumo total na última simulação executada.
- As oitava e nona opções permitem visualizar o estado atual das casas e dos fornecedores para auxiliar o utilizador a decidir que alterações quer fazer na execução da simulação.

Inserção de comandos manualmente

Ao selecionar a primeira opção do menu de execução da simulação, irá ser apresentado o seguinte menu:

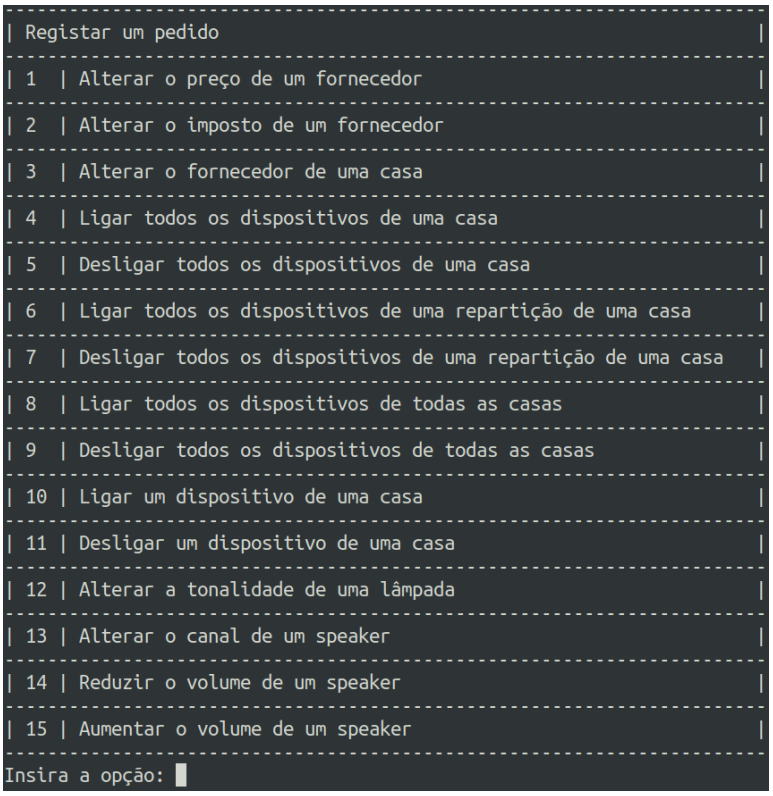


Figura 17 - Menu de inserção de comandos manualmente

Quando uma das opções do menu acima for selecionada, o programa irá solicitar informações acerca da entidade (fornecedor, casa, dispositivo) que irá sofrer as alterações que o respectivo comando executa e a data e hora em que o mesmo é executado.
Após a criação do comando, o mesmo é registado e executado quando a simulação for iniciada.
É importante referir que, quando um comando possui informação inválida, o programa irá avisar que a informação inserida não é válida (por exemplo, se uma casa não existe).

Inserção automática de comandos

Ao selecionar a terceira opção do menu de execução da simulação, o programa irá solicitar o caminho para o ficheiro de texto com os comandos a serem executados na simulação. Descrição dos comandos possíveis de serem enviados ao programa:

- `YYYY-MM-DD HH:mm,id_fornecedor,alteraPreco,novo_preco`
  Este comando serve para alterar o preço aplicado por um fornecedor dado o seu ID e o novo valor do preço.

- `YYYY-MM-DD HH:mm,id_fornecedor,alteraImposto,novo_imposto`
  Este comando serve para alterar o imposto aplicado por um fornecedor dado o seu ID e o novo imposto.

- `YYYY-MM-DD HH:mm,id_casa,alteraFornecedor,id_novo_fornecedor`
  Este comando serve para alterar o fornecedor associado a uma casa dado o ID da casa e o ID do novo fornecedor.

- `YYYY-MM-DD HH:mm,id_casa,ligaTodosNaCasa`
  Este comando serve para ligar todos os dispositivos de uma casa dado o ID da casa.

- `YYYY-MM-DD HH:mm,id_casa,desligaTodosNaCasa`
  Este comando serve para desligar todos os dispositivos de uma casa dado o ID da casa.

- `YYYY-MM-DD HH:mm,ligaTudoEmTodasAsCasas`
  Este comando serve para ligar todos os dispositivos de todas as casas que estão dentro do programa.

- `YYYY-MM-DD HH:mm,desligaTudoEmTodasAsCasas`
  Este comando serve para desligar todos os dispositivos de todas as casas que estão dentro do programa.

- `YYYY-MM-DD HH:mm,id_casa,ligaTodosNaReparticao,nome_reparticao`
  Este comando serve para ligar todos os dispositivos de uma dada repartição de uma casa dado o ID da casa e o nome da repartição.

- `YYYY-MM-DD HH:mm,id_casa,desligaTodosNaReparticao,nome_reparticao`
  Este comando serve para desligar todos os dispositivos de uma dada repartição de uma casa dado o ID da casa e o nome da repartição

- `YYYY-MM-DD HH:mm,id_casa,ligaDispositivoNaCasa,id_dispositivo`
  Este comando serve para ligar um dispositivo de uma casa dado o ID da casa e o ID do dispositivo a ser ligado.

- `YYYY-MM-DD HH:mm,id_casa,desligaDispositivoNaCasa,id_dispositivo`
  Este comando serve para desligar um dispositivo de uma casa dado o ID da casa e o ID do dispositivo a ser ligado.

- `YYYY-MM-DD HH:mm,id_casa,alteraTonalidade,id_lampada,nova_tonalidade`
  Este comando serve para alterar a tonalidade de uma lâmpada dado o ID da casa e o ID da lâmpada.

- `YYYY-MM-DD HH:mm,id_casa,alteraCanal,id_speaker,novo_canal`
  Este comando serve para alterar o canal a que um *speaker* está sintonizado dado o ID da casa e o ID do *speaker*.

- `YYYY-MM-DD HH:mm,id_casa,aumentaVolume,id_speaker`
  Este comando serve para aumentar o volume a que um *speaker* está a funcionar dado o ID da casa e o ID do *speaker*.

- `YYYY-MM-DD HH:mm,id_casa,diminuiVolume,id_speaker`
  Este comando serve para diminuir o volume a que um *speaker* está a funcionar dado o ID da casa e o ID do *speaker*.

É importante referir que, se o ficheiro de comandos tiver algum comando inválido, a execução do mesmo não é efetuada (o comando é ignorado).

Conclusões

Com a realização deste trabalho conseguimos ver como se aplica na prática alguns dos muitos conceitos abordados na programação orientada aos objetos (encapsulamento, abstração, herança, polimorfismo, etc…), bem como nos permitiu conhecer melhor e usar as funcionalidades e recursos das diversas API's do Java.
Achamos também que o tema proposto foi bastante interessante uma vez que implicou a criação de várias entidades e o estabelecimento de relações entre elas, o que nos obrigou a ter um cuidado redobrado na sua elaboração.
Finalmente, é importante referir que conseguimos implementar todas as funcionalidades solicitadas no enunciado, pelo que achamos que a realização deste projeto foi bem sucedida.