Relatório POO

IMOOBILIÁRIA



GRUPO 36

FÁBIO LUÍS BAIÃO DA SILVA, A75662
JOÃO DA CUNHA COELHO, A74859
LUÍS MIGUEL MOREIRA FERNANDES, A74748

Índice

Breve descrição do enunciado

É proposto aos alunos de POO o desenvolvimento de uma aplicação que permita fazer a gestão de imóveis. Pretende-se que a aplicação desenvolvida dê suporte a todo o ciclo de vida de um imóvel numa agência imobiliária. O processo deve abranger desde a criação do imóvel no sistema (sob a forma de anúncio), até ao registo da venda. Será necessária a existência de dois tipos distintos de utilizadores, nomeadamente, os vendedores e os compradores.

Cada perfil de utilizador deve apenas conseguir aceder às informações e funcionalidades respetivas.

- Os compradores poderão:

– Pesquisar imóveis dado um conjunto de características ou o identificador do mesmo, não necessitando de estar obrigatoriamente registados na aplicação;

– Marcar um imóvel como favorito, sendo, neste caso, necessário estar registado e autenticado no sistema.

- Os vendedores poderão:

– Inserir, consultar e remover anúncios de imóveis, bem assim como alterar o seu estado (em venda, reservado, vendido).

– Aceder a informação atualizada sobre as estatísticas respetivas (número de anúncios criados, número de visualizações dos anúncios, número de vendas de imóveis).

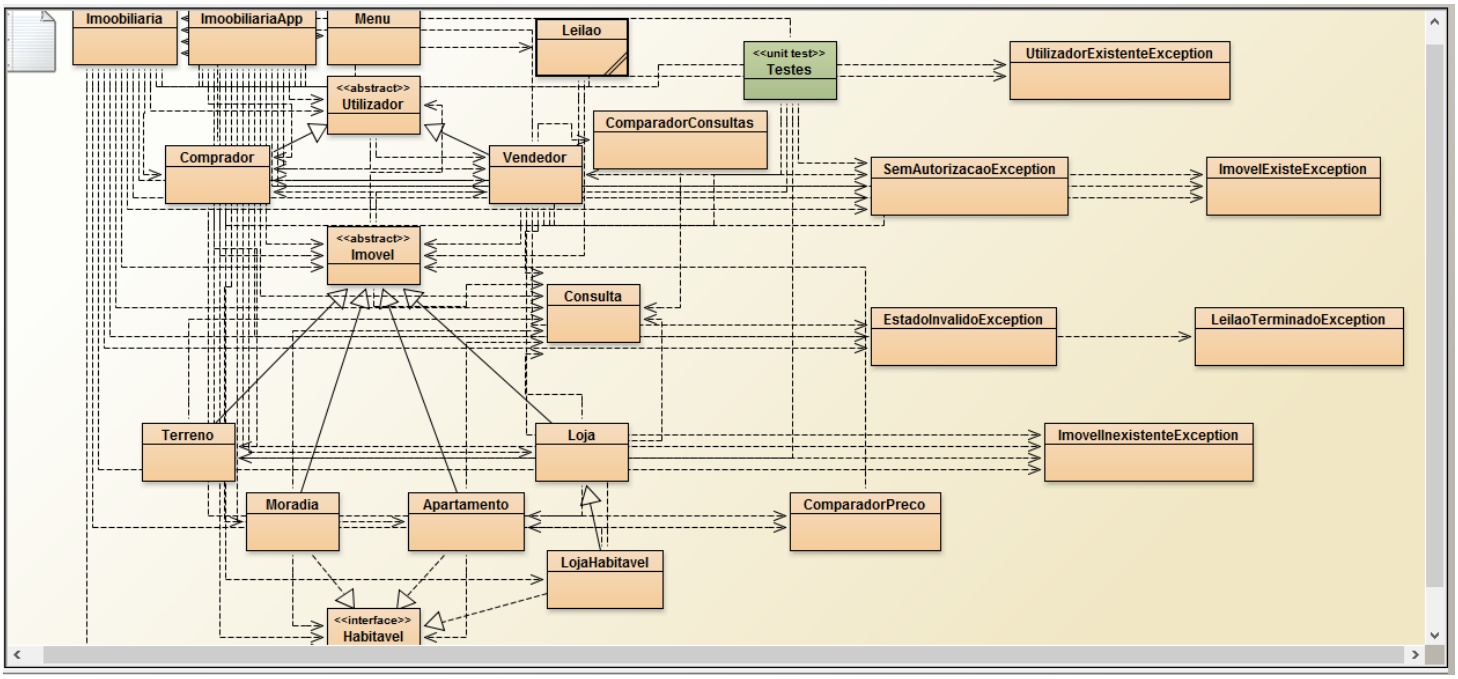
Para a gestão da Imoobiliária foram especificados métodos a implementar, como requisitos básicos do programa.

Foi também proposta a criação de um sistema de leilões: a qualquer momento um vendedor pode colocar um seu imóvel em leilão. O processo de leiloar um imóvel consiste na definição de um período (em horas) durante o qual o imóvel está em venda ao público. Durante esse período, os compradores podem fazer propostas cada vez mais altas, até ao momento do fecho do leilão.

Nesse momento, caso o preço da última oferta seja superior ao preço mínimo que o proprietário aceita pelo imóvel, este passa ao estado de reservado, até que a compra seja efetivada.

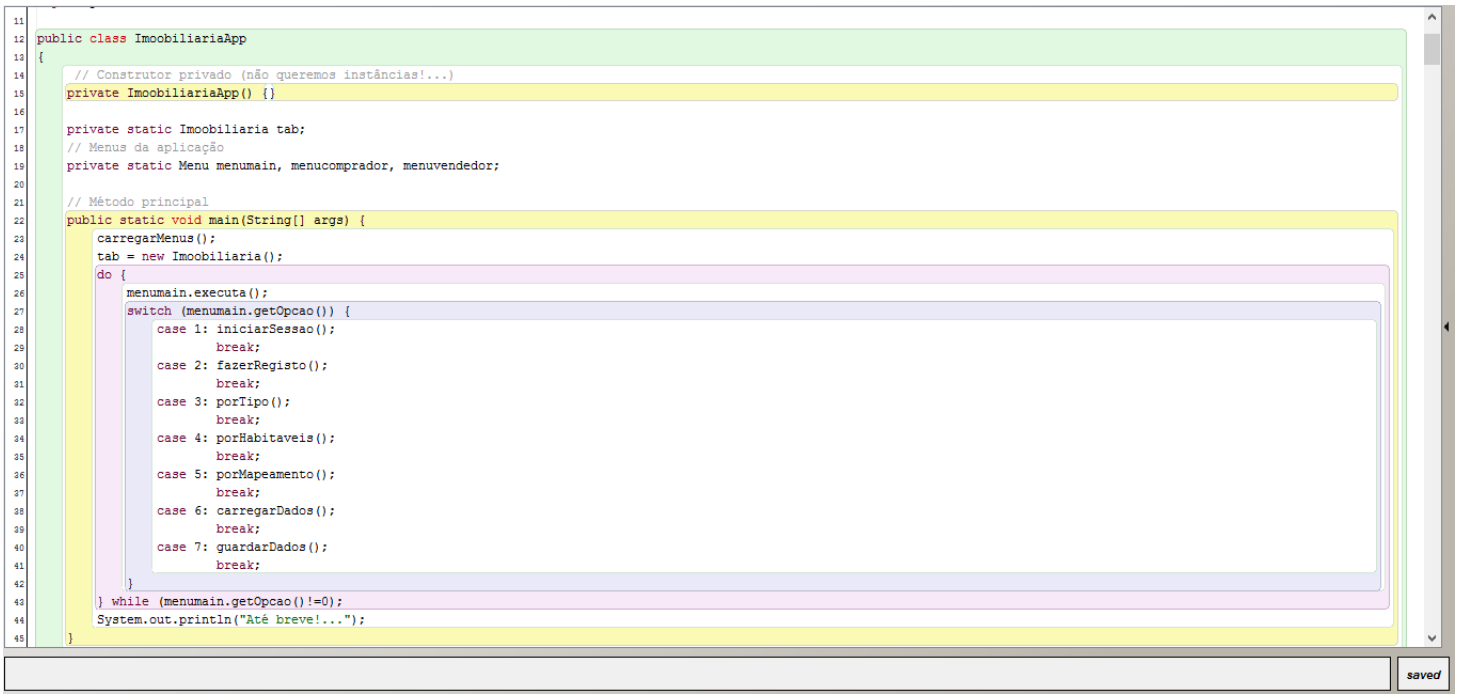
Descrição da arquitetura de classes utilizada e das decisões tomadas

Para dar resposta ao problema proposto, optamos pela seguinte arquitetura de classes:



ImoobiliariaApp

- Responsável pela interação com o utilizador da App.
- Input/Output existe apenas nesta classe, sem contarmos com as exceções, também presentes em métodos de outras classes.
- Estabelece ligação com a classe Menu, gerindo a disposição dos diferentes menus.
- Aqui são chamados os métodos de requisito mínimo apresentados pelo enunciado, que dão resposta às várias faculdades da aplicação, selecionáveis pelo utilizador.
- As únicas variáveis presentes nesta classe são uma instância da classe Imoobiliária, imutável, e três da classe Menu, correspondentes aos diferentes menus.



```
public class ImoobiliariaApp
{
    // Construtor privado (não queremos instâncias!...)
    private ImoobiliariaApp() {}

    private static Imoobiliaria tab;
    // Menus da aplicação
    private static Menu menumain, menucomprador, menuvendedor;

    // Método principal
    public static void main(String[] args) {
        carregarMenus();
        tab = new Imoobiliaria();
        do {
            menumain.executa();
            switch (menumain.getOpcao()) {
                case 1: iniciarSessao();
                        break;
                case 2: fazerRegisto();
                        break;
                case 3: porTipo();
                        break;
                case 4: porHabitaveis();
                        break;
                case 5: porMapeamento();
                        break;
                case 6: carregarDados();
                        break;
                case 7: guardarDados();
                        break;
            }
        } while (menumain.getOpcao()!=0);
        System.out.println("Até breve!...");
    }
```

saved

Menu

- Permite formar diferentes menus dado que possui como atributo uma lista de Strings onde são colocadas as opções a aparecer no menu. O segundo atributo é um inteiro onde é guardada a opção do utilizador.
- Esta classe é responsável essencialmente por permitir e gerir a apresentação no ecrã dos menus.

Imoobiliaria

- É a classe principal, apesar de a main estar na ImoobiliariaApp.
- Aqui foram implementados os métodos requisitados no enunciado.
- Optou-se por definir como variáveis de instância, para facilitar e acelerar a pesquisa, um Map<String, Utilizador>, onde são incluídos todos os utilizadores adicionados ao sistema, sendo a String o seu email de registo, um Map<String, Imovel>, onde são registados os imóveis introduzidos nos dados, com a key a ser o Id do imóvel, e um Utilizador, instância desta classe onde se guarda o utilizador com sessão iniciada.

```
public class Imoobiliaria implements Serializable
{
    //Variáveis de instância
    private Map<String, Utilizador> utilizadores;
    private Map<String, Imovel> imoveis;
    private Utilizador online;
```

- O método initApp() carrega um ficheiro, inicialmente vazio, onde são registados os dados aquando da execução da App. Com este método, juntamente com o responsável por guardar os dados, é possível, numa próxima utilização da App, recuperar o estado em que ficou.
- O registarUtilizador(Utilizador u) verifica, através do email de u, ou seja, pesquisa no keySet do Map, se este já existe na lista de utilizadores. Caso não exista insere-o, caso contrário lança uma exceção.
- No método iniciaSessao (String email, String password) inicialmente verifica-se se o utilizador já está registado, isto é, se foi encontrada alguma correspondência para a chave email. Se não for, retorna uma exceção, se for prossegue para a validação da password. Só se esta for igual à que consta nas variáveis do utilizador é que se consuma o início de sessão. Caso contrário lança uma exceção do mesmo tipo da já mencionada.
- O fechaSessao() simplesmente põe o Utilizador online a null, indicando que nenhum utilizador está em sessão.
- O método registaImovel(Imovel im), depois de verificar se o utilizador em sessão se encontra online e é um vendedor, mandando uma exceção caso contrário, verifica se o imóvel já se encontra registado (exceção em caso afirmativo). Só se não estiver no Map de imóveis é que este será adicionado, sendo também adicionado por um método da classe Vendedor ao Map de imóveis em venda ou vendidos (consoante o estado) do vendedor em questão.
- O método getConsultas() tem por base o getConsultas() da classe Vendedor. Este cria uma lista ordenada por datas das consultas aos imóveis, em venda e vendidos, do vendedor em questão (online).
- O método setEstado (String idImovel, String estado) começa por verificar se o imóvel se encontra já registado, em seguida verifica se existe um Utilizador online e se é um vendedor e por fim verifica se o estado que recebe é válido, retornando a exceção correspondente caso uma destas situações falhe. Se tudo se verificar, altera o estado do imóvel identificado pelo idImovel, recorrendo ao setEstado da classe Vendedor.

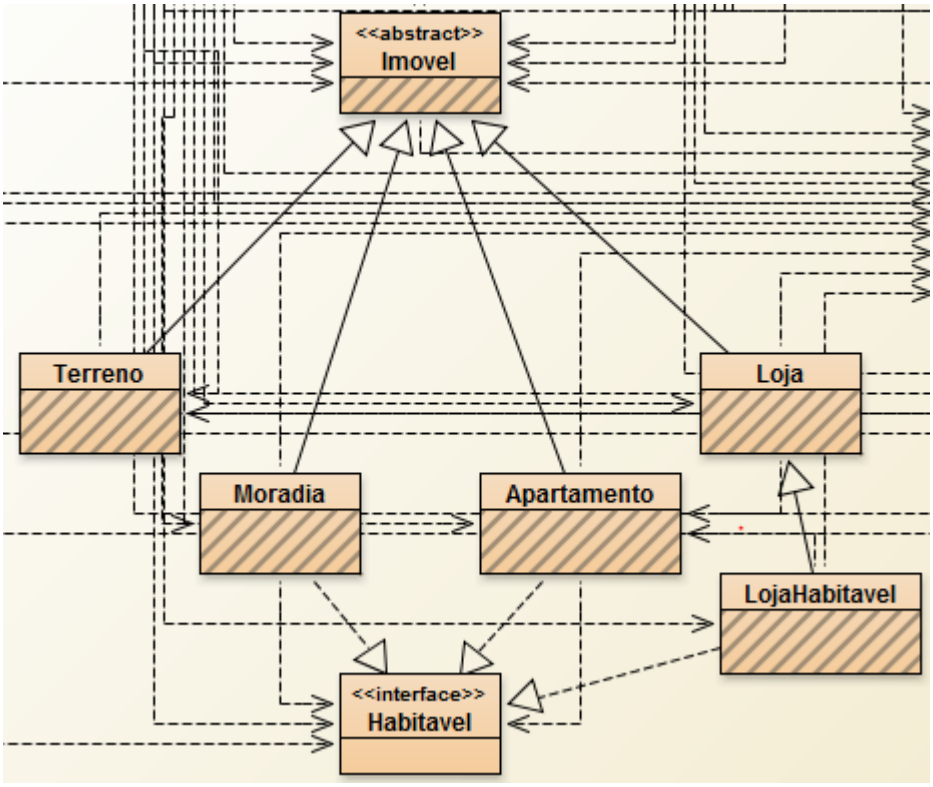
```
/**
* Alterar o estado de um imóvel, de acordo com as acções feitas sobre ele
* @param idImovel
* @param estado
*/
public void setEstado (String idImovel, String estado) throws ImovelInexistenteException, SemAutoriza
    if (!imoveis.containsKey(idImovel)){
        throw new ImovelInexistenteException ("Imovel não existe");
    }
    if (online == null || !(online instanceof Vendedor)){
        throw new SemAutorizacaoException ("Utilizador sem autorização");
    }
    if (!estado.equals("vendido") || !estado.equals("reservado") || !estado.equals("em venda")){
        throw new EstadoInvalidoException ("Estado inválido");
    }

    ((Vendedor) online).setEstado (idImovel, estado);
}
```

- O método Set<String> getTopImoveis (int n) recorre ao método de mesmo nome da classe Vendedor, no caso de o Utilizador online estar registado e ser vendedor, o qual cria um Set com os imóveis cuja lista de consultas possui dimensão superior a n.
- O método List<Imovel> getImovel (String classe, int preco) começa por apontar o email do Utilizador em sessão, caso exista, para que este possa ser registado na lista de consultas dos imóveis do tipo classe com preço pedido até preco €. É, portanto, percorrido o Map dos imóveis e vê-se se este respeita a classe e o preço necessários.
- O método List<Habitavel> getHabitaveis (int preco) começa por apontar o email do Utilizador em sessão, caso exista, para que este possa ser registado na lista de consultas dos imóveis habitáveis com preço pedido até preco €. É, portanto, percorrido o Map dos imóveis e vê-se se este implementa a interface habitável e se o preço é inferior a preco €.
- O método Map<Imovel, Vendedor> getMapeamentoImoveis () também começa por apontar o email do Utilizador em sessão, caso exista. Em seguida, usa este email para obter a lista de imóveis de cada utilizador que seja vendedor, navegando por essas listas e construindo um Map entre o Imóvel e o vendedor.
- O método setFavorito (String idImovel) começa por verificar se o imóvel se encontra já registado e, em seguida verifica se existe um Utilizador online e se é um comprador. Se isto se verificar, então recorre ao setFavorito da classe Comprador e insere este imóvel na lista dos favoritos.
- O método TreeSet<Imovel> getFavoritos () verifica inicialmente se há um Utilizador em sessão e se é um comprador, lançando uma exceção caso isto não se verifique. Depois, recorre ao email do comprador para obter a sua lista de favoritos e ordena por preço os imóveis constituintes recorrendo a um Comparator.
- O método saveApp (String nomeFicheiro) permite guardar os dados introduzidos pelo utilizador num ficheiro, de modo a preservar o estado da App.
- Nesta classe é também criada uma instância da classe Leilao e são definidos os métodos:

- iniciaLeilao (Imovel im, int horas), que começa por verificar se há um Utilizador em sessão e se é um vendedor, lançando uma exceção em caso contrário.
- adicionaComprador (String idComprador, double limite, double incrementos, double minutos), que caso o leilão não esteja terminado, adicionado um comprador ao Leilão, recorrendo ao método de mesmo nome da classe Leilao.
- encerraLeilao (), que dá por encerrado um leilão, retornando o comprador vencedor.

Imovel

- Trata-se de uma classe abstrata, uma vez que como temos vários tipos de imóveis o método clone() não pode ser instanciado, tem de ser definido para cada caso. Cada subclasse de Imóvel implementa o seu próprio clone().
- Um imóvel possui como atributos: rua, precoPedido, precoMinimo, estado, id e Lista de consultas.



Moradia

- Estende a classe Imóvel e implementa a interface habitável.
- Uma moradia possui como atributos, para além dos comuns com todos os imóveis: tipo, área de implantação, área total coberta, área do terreno envolvente, número de quartos e wc's e o número da porta.

Terreno

- Estende a classe Imóvel.
- Um terreno possui como atributos, para além dos comuns com todos os imóveis: diâmetro das canalizações (em milímetros), kWh máximo suportados pela rede elétrica, se instalados, bem como se existe acesso à rede de esgotos.

Loja

- Estende a classe Imóvel.
- Dada a existência de lojas com parte habitacional, optou-se por tornar a classe Loja abstrata, criando uma subclasse LojaHabitavel que a estende e que implementa a interface Habitavel.
- Uma loja possui como atributos, para além dos comuns com todos os imóveis: a área, se possuem, ou não, WC, qual o tipo de negócio viável na loja e o número da porta.
- Se for LojaHabitavel, para além destes atributos possui ainda uma instância de um Apartamento.

```
import java.util.List;
import java.io.Serializable;
public class LojaHabitavel extends Loja implements Habitavel, Serializable
{
    // variáveis de instância
    private Apartamento apartamento;

    /**
     * Construtor para objetos da classe LojaHabitavel
     */
    public LojaHabitavel (){
        super ();
        apartamento = new Apartamento();
    }

    public LojaHabitavel (String rua, double precoPedido, double precoMinimo,
```

Apartamento

- Estende a classe Imóvel e implementa a interface Habitavel.

- Um apartamento possui como atributos, para além dos comuns com todos os imóveis: o tipo, a área total, o número de quartos e WC's, o número da porta e o andar, e se possui, ou não, garagem.

Utilizador

- É uma classe abstrata à semelhança da Imovel.
- Os atributos de um utilizador já foram mencionados anteriormente.

Comprador

- É uma das classes que estende a Utilizador.
- Os atributos específicos de um comprador também já foram mencionados.
- A propósito dos leilões, são definidas variáveis na classe Comprador relativas à licitação de um comprador, como são o limite, os incrementos entre licitações, o intervalo de tempo entre elas, os minutos restantes do leilão e o preço final.

Vendedor

- É uma das classes que estende a Utilizador.
- Os atributos específicos de um vendedor também já foram mencionados.
- Grande parte dos métodos da classe Vendedor intervêm nos métodos da classe Imoobiliaria, sendo até homónimos muitos deles, pelo que a sua explicação já foi adiantada.
- A propósito dos leilões, é definido o método iniciaLeilao(Imovel im, int horas) que, caso o imóvel a leiloar não pertença à lista de imóveis para venda do vendedor, lança uma exceção.

Consulta

- Classe que representa uma consulta a um imóvel.
- Como atributos possui a data da consulta e o email do utilizador que consultou o imóvel.

ComparadorConsultas

- Implementa um Comparator.
- Permite ordenar, por data de ocorrência, a lista onde são reunidas as consultas de vários imóveis.

```
import java.util.Comparator;

public class ComparadorConsultas implements Comparator<Consulta>
{
    public int compare (Consulta m ,Consulta a) {
        if (m.getData().compareTo(a.getData())>0){
            return 1;
        }
        if (m.getData().compareTo(a.getData())==0){
            return 0;
        }
        else return -1;
    }
}
```

ComparadorPreco

- Implementa um Comparator.
- Permite ordenar, por preço, o Set onde são reunidos os imóveis favoritos de um comprador.

```
import java.util.Comparator;

public class ComparadorPreco implements Comparator<Imovel>
{
    public int compare(Imovel i1, Imovel i2){
        double p1 = i1.getPrecoPedido();
        double p2 = i2.getPrecoPedido();
        if (p1 < p2){
            return -1;
        }
        if (p1 == p2){
            return 0;
        }
        return 1;
    }
}
```

Leilao

- Para responder ao desafio dos leilões, criou-se esta classe, cujas variáveis de instância são o tempo do leilão, o imóvel a ser leiloado, a licitação vencedora até ao momento, o comprador vencedor e uma lista com os licitadores.

- O método adicionaComprador (Comprador comprador) adiciona um Comprador à lista de licitadores.
- O método encerraLeilao()faz um ciclo minuto a minuto e em cada iteração reduz um minuto ao tempo que falta para a próxima licitação de cada comprador. Se esse tempo já estiver a zero verifica se está em condições de fazer uma licitação.

Diferentes classes de Exceções

- UtilizadorExistenteException, SemAutorizaçãoException, LeilaoTerminadoException, ImovelExisteException, ImovelInexistenteException, EstadoInvalidoException.

```
1  import java.lang.Exception;
2
3  public class SemAutorizacaoException extends Exception{
4      public SemAutorizacaoException(String msg){
5          super(msg);
6      }
7  }
```

Testes

- Classe de testes.

```
int s = imo.getImovel("Terreno", Integer.MAX_VALUE).size();
assertTrue(s>0);
Set<String> ids = imo.getTopImoveis(0);
assertTrue(ids.contains(t.getId()));
assertTrue(imo.getMapeamentoImoveis().keySet().contains(t));
try {
    assertTrue(imo.getConsultas().size()>0);
} catch(Exception e) {
    fail();
}
```

Descrição da aplicação desenvolvida

```
 *** IMOOBILIÁRIA ***

0 - Sair
1 - Iniciar Sessão
2 - Fazer Registo
3 - Consultar imovéis por tipo
4 - Consultar imóveis habitáveis
5 - Consultar imóveis e respetivos vendedores
6 - Carregar Dados
7 - Guardar Dados

Opção:
```

1. Menu inicial.

```
--- Iniciar Sessão ---

Introduza a sua conta de utilizador (email): vendedor1
Introduza a password: 12345
```

2. Iniciar sessão.

```
--- Imóveis Habitáveis ---

Preço Máximo: 100000

Imóvel: md2_1
 Localização: Rua M
 Preço: 65000.0

Imóvel: ap3_1
 Localização: Rua J
 Preço: 23000.0

Imóvel: ljh4_1
 Localização: Rua H
 Preço: 29000.0

Imóvel: ap1_1
 Localização: Rua D
 Preço: 10000.0

Imóvel: ap1_2
 Localização: Rua A
 Preço: 17000.0
```

3. Imóveis habitáveis.

```
--- Marcar Imóvel como Favorito ---

Identifique o imóvel a adicionar aos favoritos: ap1_2
Adicionado!!
```

4. Marcar imóvel como favorito.

```
--- Imóveis Favoritos ---

Imóvel: ap1_1
Localização: Rua D
Preço: 10000.0

Imóvel: ap1_2
Localização: Rua A
Preço: 17000.0

Imóvel: ap3_1
Localização: Rua J
Preço: 23000.0
```

5. Imóveis favoritos.

```
--- Entrar no Leilão ---

Defina um limite de licitação: 9000
Quanto está disposto a incrementar? 500
Defina um intervalo entre licitações: 10
Adicionado ao Leilão
```

6. Entrar no Leilão.

```
--- Gerir Leilões ---

Criar(1) ou encerrar(2) leilão? 1
Indique o id do imóvel que pretende leiloar: ap1_1
Quantas horas pretende deixar o leilão aberto? 8
Leilão iniciado (à espera de compradores)
```

7. Gerir Leilões.

```
--- Gerir Leilões ---

Criar(1) ou encerrar(2) leilão? 2
Leilão encerrado
Vencedor: Patrícia
 Preço: 9000.0
```

```
*** IMOOBILIÁRIA ***

0 - Terminar Sessão
1 - Consultar lista de imóveis favoritos
2 - Adicionar imóvel aos favoritos
3 - Consultar imovéis por tipo
4 - Consultar imóveis habitáveis
5 - Consultar imóveis e respetivos vendedores
6 - Juntar-se ao leilão

Opção:
```

8. Menu Compradores.

```
*** IMOOBILIÁRIA ***

0 - Terminar Sessão
1 - Registar imóvel para venda
2 - Alterar estado de um imóvel
3 - Ver as 10 últimas consultas a imóveis para venda
4 - Lista dos imóveis mais consultados
5 - Consultar imovéis por tipo
6 - Consultar imóveis habitáveis
7 - Consultar imóveis e respetivos vendedores
8 - Gerir leilão

Opção:
```

9. Menu Vendedores.

```
---- Mapeamento Imóveis-Vendedor ---

Imóvel: md2_2
 Localização: Rua A
 Preço: 110000.0
 Vendedor: Manuel

Imóvel: tr2_1
 Localização: Rua N
 Preço: 47000.0
 Vendedor: Manuel

Imóvel: md2_1
 Localização: Rua M
 Preço: 65000.0
 Vendedor: Manuel

Imóvel: lj3_1
 Localização: Rua O
 Preço: 9000.0
 Vendedor: João

Imóvel: lj3_2
 Localização: Rua A
 Preço: 14000.0
 Vendedor: João

Imóvel: lj4_1
 Localização: Rua E
 Preço: 16000.0
 Vendedor: Helena
```

11. Mapeamento Imóveis-Vendedores.

```
--- Tipo de imóvel ---

 1. Moradia
 2. Terreno
 3. Apartamento
 4. Loja
 5. LojaHabitavel
Opção: 3
Preço Máximo: 100000

Imóvel: ap3_1
Localização: Rua J
Preço: 23000.0

Imóvel: ap1_1
Localização: Rua D
Preço: 10000.0

Imóvel: ap1_2
Localização: Rua A
Preço: 17000.0
```

10. Tipo de imóvel.

```
--- Imóveis mais Consultados ---

Introduza o número minimo de consultas: 1

Imóveis mais consultados:
Imóvel: ap1_1
Imóvel: ap1_2
```

12. Imóveis mais consultados.

Discussão sobre como seria possível incluir novos tipos de imóveis na aplicação

- Para incluir novos tipos de imóveis, seria necessária criar classes para esses tipos, as quais seriam subclasse da abstract Imovel, isto é, teriam “extends Imovel” no cabeçalho da classe.
- Nestas classes dos novos tipos de imóveis seria necessário definir o método clone(), dado que este não está definido em Imovel por ser dependente da subclasse/tipo de imóvel.
- Cada tipo de imóvel tem as suas próprias características logo um novo tipo de imóveis implicaria variáveis de instância específicas da classe.
- Nos construtores da classe, seria usado o construtor de superclasse super(), para inicializar as variáveis comuns à classe Imovel, sendo as restantes inicializadas normalmente.