ImOObiliária

Relatório do trabalho prático de Programação Orientada aos Objetos

Universidade do Minho
Escola de Engenharia

Grupo 17

Ana Paula Carvalho
(A61855)

João Pires Barreira
(A73831)

Rafael Braga
(A61799)

1 Enunciado do projeto

No âmbito da unidade curricular de Programação Orientada aos Objetos, foi-nos proposto o desenvolvimento de uma aplicação que permitisse efetuar a gestão dos imóveis de uma imobiliária, desde a criação de um imóvel no sistema, até à sua venda. Para isso, tivemos de implementar as classes relativas aos próprios imóveis e aos vários atores existentes.

1.1 Utilizadores

Representam entidades que partilham informações como email, nome, password, morada e data de nascimento. Estão subdivididos em dois grupos com características comuns entre si: *vendedores* e *compradores*.

1.1.1 Vendedores

Representam as entidades responsáveis pela gestão dos anúncios de imóveis para venda. Estes utilizadores têm ainda um *portfólio de imóveis em venda* e um *histórico de imóveis vendidos*, que os distinguem dos restantes utilizadores.

1.1.2 Compradores

São entidades que representam as pessoas que irão fazer a compra dos imóveis. Caso se encontrem registados, terão a possibilidade de definir a sua *lista de imóveis favoritos*.

1.2 Imóveis

Representam as entidades à venda no sistema. Os imóveis distinguem-se em *moradias*, *apartamentos*, *lojas* e *terrenos*. Cada imóvel, independentemente do seu tipo, tem a si associado uma *morada*, um *preço* e um *preço mínimo* aceite pelo vendedor.

1.2.1 Moradia

É um imóvel indicado para habitação familiar. Cada moradia deverá ser caracterizada por *tipo* (i.e. *isolada*, *geminada*, *em banda* ou *gaveto*), *área de implantação*, *área total coberta*, *número de quartos* e *WCs* e *número de porta*.

1.2.2 Apartamento

Um imóvel inserido num prédio que não possui jardim. Tal como as moradias, os apartamentos são específicados por *tipo* (i.e. *Simples*, *Duplex* ou *Triplex*), por *número de quartos* e *WCs*, *número de porta* e *andar*. Tem ainda informação quanto à existência ou não de garagem.

1.2.3 Loja

Representa um espaço destinado a um negócio. Contém informações relativamente à sua *área*, se *possui WC* ou não, qual o *tipo de negócio viável* e o *número de porta*. Existem, no entanto, algumas lojas específicas que possuem uma *parte habitacional*, sendo esta guardada sob a forma de um *apartamento* anexo.

1.2.4 Terreno

Espaço disponível para construção. Um terreno é discriminado como sendo apropriado para a construção de uma habitação ou um armazém. São ainda considerados dados relativamente ao diâmetro (em milímetros) das canalizações, à potência suportada pela rede elétrica e à existência ou não de acesso à rede de esgotos.

1.3 Leilões

Aos vendedores, é dada a possibilidade de colocar um dos seus imóveis em leilão. O processo de leiloar um imóvel consiste na estipulação de um período (em horas) durante o qual o imóvel estará à venda ao público. Durante esse período, os compradores podem fazer licitações cada vez mais altas até ao momento de encerramento do leilão. Após o encerramento, caso o preço da última oferta seja superior ao preço mínimo que o proprietário aceita pelo imóvel, este passa ao estado reservado até que se efetue a compra.

Cada comprador pode declarar a intenção de participar num leilão, qual o valor máximo que está disposto a dar, qual o valor dos incrementos a executar e o intervalo entre licitações. No final do leilão é indicado o comprador vencedor e o valor da venda do imóvel.

2 Arquitetura de classes

A nossa aplicação encontra-se dividida em 17 classes, além da package das exceções. Todas as classes implementam métodos comuns, nomeadamente os métodos *equals, hashCode e toString*. Ambas fornecem também, pelo menos, construtores por partes e construtores de cópia. Assim, todas implementam o método *clone* para garantirem o encapsulamento. As classes guardam os seus dados em coleções o mais genéricas possível de modo a aumentar a extensibilidade da aplicação.

2.1 Main, Paginacao, Menu e Input

Classe responsáveis pela aplicação *Imoobiliaria*. As funcionalidades destas classes serão descritas mais aprofundamente na seccção *Aplicação e ilustração das funcionalidades*.

2.2 Imoobiliaria

Classe principal dos dados da aplicação. Esta classe contém o registo de todos os utilizadores (quer vendedores, quer compradores) registados numa imobiliária. Contém também todos os imóveis anunciados pelos vendedores. Esta classe interage com todas as classes de dados da aplicação e contém funções para a manipulação das mesmas.

- **Variáveis de instância**:
  - **utilizadores** – mapeamento entre os utilizadores e os respetivos emails (i.e. Map<String, Utilizador>). Claramente, para o controlo dos utilizadores a coleção mais adequada é um Map. Optou-se por guardar os dados num *Map* e não numa coleção mais específica, aumentando a extensibilidade da aplicação.
  - **imoveis** – lista de todos os imóveis (i.e. List<Imovel>). Os imóveis são guardados numa *List* dado à fácil manipulação e eficiência desta. Novamente, os imóveis não foram guardados numa coleção mais específica de modo a aumentar a extensibilidade.
  - **utilizador** – utilizador com sessão iniciada.
  - **leilao** – último leilão.
- **Métodos mais relevantes**:

- **getConsultas()** – Cria-se um comparador de forma a ordenar as datas por ordem cronológica. De seguida, para cada imóvel verifica-se se o vendedor com sessão iniciada (caso não seja um vendedor, lança-se a exceção SemAutorizacao) tem esse imóvel na sua lista de imóveis em venda e, em caso afirmativo, adicionam-se as consultas feitas a esse imóvel ao TreeSet das consultas, que fica ordenado pelo comparador. Finalmente, adicionam-se as últimas 10 consultas de cada imóvel ao ArrayList que depois é devolvido no return.
- **getTopImoveis()** – Através de um stream() e de um filter(), filtram-se os imóveis com um número de consultas superior ao parâmetro n e, de seguida, com o auxílio de um map() adicionam-se ao TreeSet do return os seus códigos de identificação.
- **getImovel()** – Para todos os imóveis, verifica-se se correspondem a uma dada classe recebida como parâmetro e se o preco é inferior ao preço recebido como parâmetro. Em caso afirmativo, adiciona-se uma nova consulta com a data atual e o email do utilizador com sessão iniciada. Por fim, retorna-se a um ArrayList com o resultado da pesquisa.
- **getHabitaveis()** – Para cada imóvel, verifica-se se é do tipo Habitavel e se o seu preço está abaixo do preço máximo recebido como parâmetro. Em caso afirmativo, adiciona-se uma nova consulta com a data atual e o email do utilizador com sessão iniciada. Por fim, retorna-se um ArrayList com o resultado da pesquisa.
- **getMapeamentoImoveis()** – Para cada imóvel e para cada vendedor, verifica-se se este contém o imóvel na sua lista de imóveis em venda. Em caso afirmativo, adiciona-se ao Map do return, uma cópia desse mesmo vendedor. Por fim, retorna-se o Map que faz corresponder a cada imóvel o seu respetivo vendedor.
- **getFavoritos()** – Primeiro cria-se um comparador para ordenar os imóveis por preço. Se o utilizador for um comprador (caso não o seja, é lançada a exceção SemAutorizacao), vai-se à sua lista de favoritos e adiciona-se uma cópia de cada um dos imóveis favoritos ao TreeSet do return ordenado pelo comparador. Por fim, devolve-se o TreeSet.
- **setEstado()** – Primeiro verifica-se se o utilizador com sessão iniciado é um vendedor - se não o for é lançada a exceção SemAutorizacao. De

seguida, verifica-se se o vendedor contém na sua lista de imóveis em venda o imóvel cujo código de identificação é passado como parâmetro - caso não contenha, é lançada a exceção ImovelInexistente. Além disso, é validado o estado recebido como parâmetro - caso não seja válido é lançada a exceção EstadoInvalido. Após o tratamento das exceções, procura-se o imóvel cujo código de identificação é passado ao método e altera-se o seu estado para o pretendido. Por fim, caso se tenha alterado o estado do imóvel para "vendido", é adicionado o imóvel à lista de imóveis vendidos do vendedor.

- **setFavorito()** – Primeiro verifica-se se o utilizador com sessão iniciada é um comprador - caso não o seja é lançada a exceção SemAutorizacao. De seguida, verifica-se se o código de identificação recebido como parâmetro corresponde a algum dos imóveis no campo "imoveis" da Imoobiliaria - caso não exista, é lançada a exceção ImovelInexistente. Em caso afirmativo, faz-se uma cópia desse imóvel e adiciona-se esse imóvel à lista de favoritos do comprador.
- **registarUtilizador()** – Primeiro verifica-se se a lista de utilizadores já contém o utilizador que se pretende registar - caso já contenha, é lançada a exceção UtilizadorExistente. Caso contrário é adicionado o utilizador passado como parâmetro à lista de utilizadores da Imoobiliaria.
- **iniciaSessao()** – Primeiro verifica-se se o utilizador está registado (i.e. se o seu email consta da lista de utilizadores da Imoobiliaria) e se a password recebida como parâmetro correponde à que consta da lista dos utilizadores. Caso algum destes casos não se verifique, é lançada a exceção SemAutorizacao. Se o utilizador estiver registado, é adicionado ao campo "utilizador" da Imoobiliaria.
- **registaImovel()** – Primeiro verifica-se se o imóvel já existe na lista de imóveis da Imoobiliaria - caso já exista é lançada a exceção ImovelExiste. De seguida, verifica-se se o utilizador com sessão iniciada é um vendedor - caso não o seja, é lançada a exceção SemAutorizacao. Depois percorre-se todos os utilizadores à procura do vendedor com sessão iniciada e adiciona-se este novo imóvel à sua lista de imóveis em venda. Por fim, são atualizados os dados do vendedor na lista de utilizadores da Imoobiliaria, de forma a que contenha esta alteração no seu portfólio.

- **iniciaLeilao()** – Primeiro verifica-se se o utilizador com sessão iniciada é um vendedor - caso não seja é lançada a exceção SemAutorizacao. Inicia-se o leilão e percorre-se a lista de utilizadores de forma a encontrar todos os compradores que participem no leilão (i.e. os que têm o campo "participaLeilao" a true) e a adicioná-los à lista de compradores do leilão.
- **adicionaComprador()** – Primeiro verifica-se se o utilizador com sessão iniciada é um vendedor - caso não seja é lançada a exceção SemAutorizacao. Caso o leilão não exista ou já tenha terminado é lançada a exceção LeilaoTerminado. Depois, caso o comprador exista na lista dos utilizadores da Imoobiliaria, é adicionado o comprador ao leilão.
- **encerraLeilao()** – Primeiro verifica-se se o utilizador com sessão iniciada é um vendedor - caso não seja é lançada a exceção SemAutorizacao. Caso o leilão não exista (i.e. seja igual a "null") é lançada a exceção LeilaoTerminado. De seguida, cria-se um cópia do comprador que venceu o leilão e muda-se o estado do imóvel leiloado para reservado. Por fim, retorna-se a cópia do comprador vencedor.
- **gravaObj()** – Grava os campos da instância Imoobiliaria num ficheiro objeto.

2.3 Utilizador

- **Variáveis de instância**:
  - **email** – endereço de email do utilizador.
  - **nome** – nome do utilizador.
  - **password** – palavra-passe do utilizador.
  - **morada** – morada do utilizador.
  - **dataNascimento** – data de nascimento do utilizador, implementada com o tipo *GregorianCalendar*.
- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.3.1 Vendedor

Além de herdar os métodos e as variáveis de instância da superclasse *Utilizador* esta classe possui duas coleções do tipo *List*.

- **Variáveis de instância**:
  - **portfolio** – lista de imóveis em venda (i.e. List<Imovel>).
  - **historico** – lista de imóveis vendidos (i.e. List<Imovel>).
- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), os métodos *equals()*, *toString()*, *clone()* e *hashCode()*, e métodos simples de consulta e inserção no portfólio/histórico.

2.3.2 Comprador

Além de herdar os métodos e as variáveis de instância da superclasse *Utilizador*:

- **Variáveis de instância**:
  - **participaLeilao** – se participa ou não nos leilões.
  - **limite** – quantia máxima que o comprador está disposto a dar num leilão.
  - **incrementos** – valor de incremento entre as licitações nos leilões.
  - **minutos** – tempo entre as licitações dos leilões.
  - **favoritos** – conjunto de imóveis favoritos (i.e. *Set<Imovel>*).
- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), os métodos *equals()*, *toString()*, *clone()* e *hashCode()*, e métodos simples de consulta e inserção no portfólio/histórico.

2.4 Imovel

- **Variáveis de instância**:
  - **id** – código de identificação do imóvel.
  - **rua** – rua onde se localiza o imóvel.

  - **preco** – preço do imóvel.
  - **precoMin** – preço mínimo estipulado pelo vendedor.
  - **consultas** – lista de consultas ao imóvel (i.e. *List<Consulta>*).
  - **estado** – estado do imóvel (i.e. *emVenda*, *vendido*, *reservado*).

- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.4.1 Moradia

Além de herdar os métodos e as variáveis de instância da superclasse *Imovel* e de implementar a interface *Habitavel*:

- **Variáveis de instância**:
  - **tipo** – tipo da moradia (i.e. *isolada*, *geminada*, *em banda* ou *gaveto*), implementado com um *enum TipoMoradia*.
  - **areaImplantacao** – área de implantação da moradia.
  - **areaTotal** – área total coberta da moradia.
  - **areaTerreno** – área do terreno envolvente.
  - **numQuartos** – número de quartos.
  - **numWCs** – número de casas de banho.
  - **numPorta** – número da porta.

- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.4.2 Apartamento

Além de herdar os métodos e as variáveis de instância da superclasse *Imovel* e de implementar a interface *Habitavel*:

- **Variáveis de instância**:
  - **tipo** – tipo do apartamento (i.e. *simplex*, *duplex* ou *triplex*), implementado com um *enum TipoApartamento*.
  - **areaTotal** – área total do apartamento.

  - **numQuartos** – número de quartos.
  - **numWCs** – número de casas de banho.
  - **numPorta** – número da porta.
  - **numAndar** – número do andar.
  - **possuiGaragem** – se possui ou não garagem.

- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.4.3 Terreno

Além de herdar os métodos e as variáveis de instância da superclasse *Imovel*:

- **Variáveis de instância**:
  - **tipo** – tipo do terreno (i.e. *habitação* ou *armazém*), implementado com um *enum TipoTerreno*.
  - **diametro** – diâmetro das canalizações.
  - **kWh** – capacidade máxima da rede elétrica.
  - **existeAcesso** – se existe ou não acesso à rede de esgotos.

- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.4.4 Loja

Além de herdar os métodos e as variáveis de instância da superclasse *Imovel*:

- **Variáveis de instância**:
  - **area** – área da loja.
  - **possuiWC** – se possui ou não WC.
  - **tipo** – tipo de negócio viável na loja (como não foram discriminadas as várias opções a este respeito, utilizámos uma *String* em vez de um *enum*).
  - **numPorta** – número da porta.

- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.4.5 LojaHabitavel

Além de herdar os métodos e as variáveis de instância das superclasses *Imovel* e *Loja*, e de implementar (ao contrário da superclasse *Loja*) a interface *Habitavel*:

- **Variáveis de instância**:
  - **apartamento** – variável do tipo *Apartamento* que representa a parte habitável das lojas do tipo habitável.
- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.5 Habitavel

Interface que, no contexto do projeto, serve para sinalizar os imóveis que são do tipo habitável e que, a nível da nossa implementação, apenas possui os métodos (*getters* e *setters* das variáveis de instância) comuns às classes que partilham desta interface.

2.6 Consulta

- **Variáveis de instância**:
  - **email** – endereço de email do utilizador que efetuou a consulta.
  - **data** – data em que foi realizada a consulta.
- **Métodos mais relevantes**: apenas tem definidos os *getters* e os *setters* das variáveis de instância, os construtores (vazio, por partes e por cópia), e os métodos *equals()*, *toString()*, *clone()* e *hashCode()*.

2.7 Leilao

- **Variáveis de instância**:
  - **compradores** – lista dos compradores que participam no leilão.

  - **montantes** – é um mapeamento entre todos os compradores do leilão e seus respetivos montantes licitados.
  - **imovel** – imóvel a ser leiloado.
  - **horas** – duração do leilão.
  - **montante** – valor máximo licitado no leilão.
  - **inicio** – contem as horas do início do leilão.

- **Métodos mais relevantes**:
  - **addComprador()** – adiciona um comprador à lista de compradores registados e calcula o seu montante.
  - **getvencedor()** – devolve o email do maior licitador.
  - **leilaoTerminado()** – verifica se um leilão já terminou.

2.8 ImoobiliariaExceptions (package)

Optou-se por criar uma pasta para guardar as excessões presentes na classe *Imoobiliaria* por uma questão de organização. Esta *package* inclui uma classe para cada uma das seis exceções existentes na aplicação (*UtilizadorExistente*, *SemAutorizacao*, *ImovelExiste*, *EstadoInvalido*, *ImovelInexistente* e *LeilaoTerminado*). Em cada uma dessas classes, optámos por implementar dois construtores: um vazio e outro que recebe uma mensagem de erro. Ambos os construtores chamam os da sua superclasse *Exception* pré-definida em Java.

3 Aplicação e ilustração das funcionalidades

A interface com o utilizador é realizada através das classes *Main, Input, Menu e Paginacao*. As classes *Input e Menu* são classes auxiliares que servem, respetivamente, para a leitura de vários tipos de dados do teclado escondendo os seus vários tipos de erro e criação de menus. A classe *Paginacao* é responsável pela paginação de uma lista se *Strings*. Esta classe é útil para certos métodos desta aplicação onde se pretende listar enormes quantidades de dados. Assim, é possível navegar nestes em pequenas páginas. Os principais métodos desta classe permitem navegar para a página seguinte ou página anterior de uma lista, bem como navegar para um índice válido de uma página desta.

A classe é a classe principal da aplicação. A classe é responsável pela leitura dos dados préviamente guardados em ficheiros de objetos da classe *Imoobiliaria*. A leitura e escrita dos dados é guardada por defeito no ficheiro *imoobiliaria.dat*, no entanto o utilizador pode alterar o nome deste passando o nome como parâmetro na execução da aplicação. Sempre que ocorre uma mudança da classe *Imoobiliaria* os dados são guardados no ficheiro correspondente. A aplicação contém as seguintes funcionalidades :

- **Registar um utilizador.**
- **Efetuar o login de um utilizador préviamnte resgistado.**
- **Todos os utilizadores:** Pesquisar qualquer tipo de imóveis e obter o mapeamento entre todos os imóveis e respetivos vendedores.
- **Compradores:** Pesquisar qualquer tipo de imóveis e obter o mapeamento entre todos os imóveis e respetivos vendedores. Podem adicionar imóveis ao seu conjunto de imóveis favoritos, bem como consultar este conjunto sempre que pretenderem.
- **Vendedores:** Pesquisar qualquer tipo de imóveis e obter o mapeamento entre todos os imóveis e respetivos vendedores. Podem registar imóveis, adicionando-os aos seus portfólios, bem como alterar o estado destes. Podem obter a lista dos seus imóveis mais consultados e obter a lista das 10 últimas consultas realizadas aos seus imóveis. Aos vendedores é possível consultar os seus portfólios e os seus históricos de imóveis vendidos. Finalmente, os vendedores podem criar leilões de um dado imóvel que possuam para venda e adicionar compradores a este.

A aplicação é robusta pois não permite comportamentos que podem comprometer a execução do programa. A interface do utilizador impede o acesso a certas operações não permitidas, embora a classe *Imoobiliaria* possua vários mecanismos para esta funcionalidade.

4 Outras possíveis funcionalidades

As classes foram criadas de modo a fornecerem uma grande extensibilidade. Para tal, os dados foram guardados com os tipos mais genéricos possível. Assim

podem-se acrescentar vários subtipos sem que a aplicação mude de comportamento. Como os diferentes tipos de imóveis são guardados como o seu supertipo *Imovel*, caso se pretenda acrescentar outro tipo de imóvel, esse novo tipo apenas teria que estender a classe *Imovel* e implementar os seus tipos específicos. O mesmo acontece para o tipo *Utilizador*. Outra consideração tomada consistiu em guardar os dados em coleções o mais abstratas possível. Assim a aplicação pode converter estas coleções para coleções mais específicas conforme lhe convenha sem haver algum impacto nas classes responsáveis pelo armazenamento dos dados.

5 Conclusão

Com a realização deste trabalho concluímos que o bom encapsulamento dos dados aumenta em muito a segurança de uma aplicação. O uso de super classes e o armazenamento de super tipos em vez de tipos mais específicos é muito útil para a extensibilidade do programa. Deste modo podem-se fácilmente definir e acrescentar vários subtipos sem ter que se mudar tudo o que já foi criado. Da mesma maneira, o uso das coleções mais genéricas de *Java* aumenta em muito a reutilização de código. As interfaces são um mecanismo interessante para reunir propriedades comuns entre classes distintas. Assim, é possível agrupar vários tipos de classes que sao completamente diferentes mas que contêm certas propriedades comuns. Neste trabalho tentámos usufruir das novas ferramentas provenientes de *Java 8*. Estas fornecem bastante eficiêcia e legibilidade de código já que permitem realizar operações complicadas em poucas instruções.