Programação Orientada aos Objetos

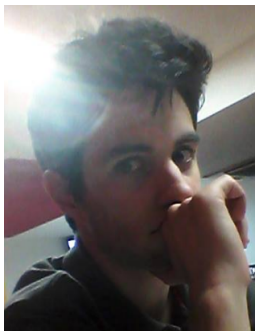
LCC Universidade do Minho

Ano Letivo 2015/2016

ImOObiliária — Trabalho prático

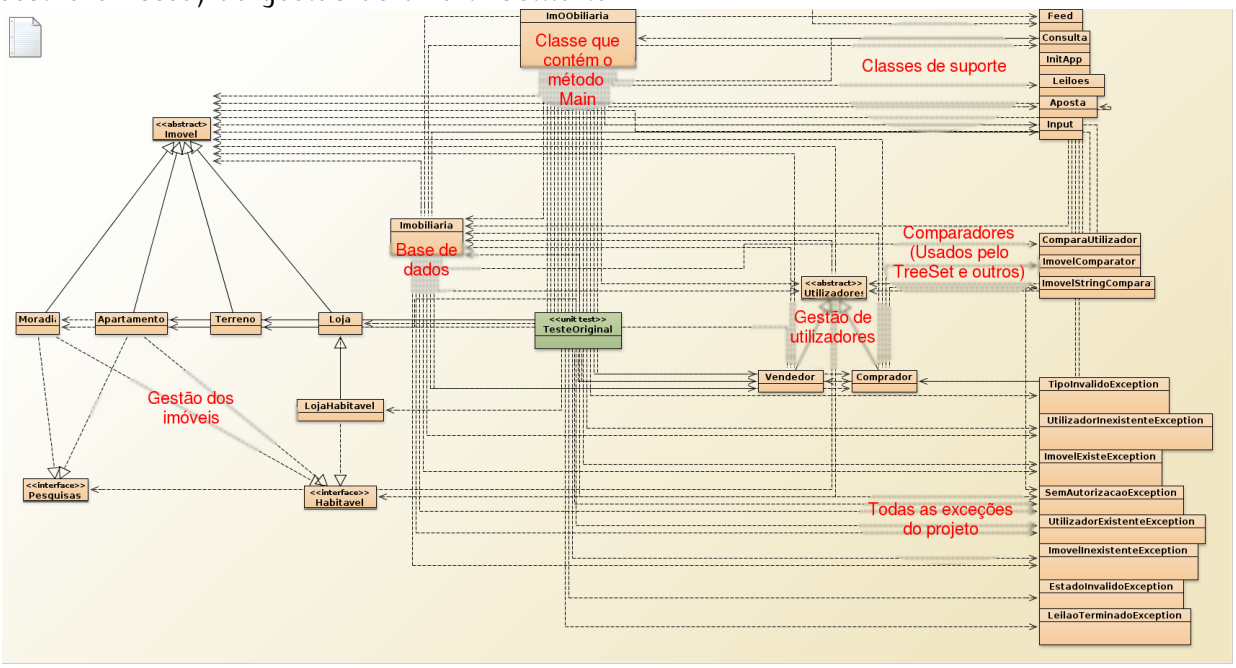
Grupo 5 62134 LCC Sérgio Oliveira
66698 LCC Carlos Silva

19 de Maio de 2016




1 Introdução

Nós somos o Sérgio Oliveira e o Carlos Silva, do grupo 18 da cadeira de Programação Orientada aos Objetos. Temos como objetivo construir uma interface (gráfica ou terminal, a escolha é nossa) de gestão de uma imobiliária.



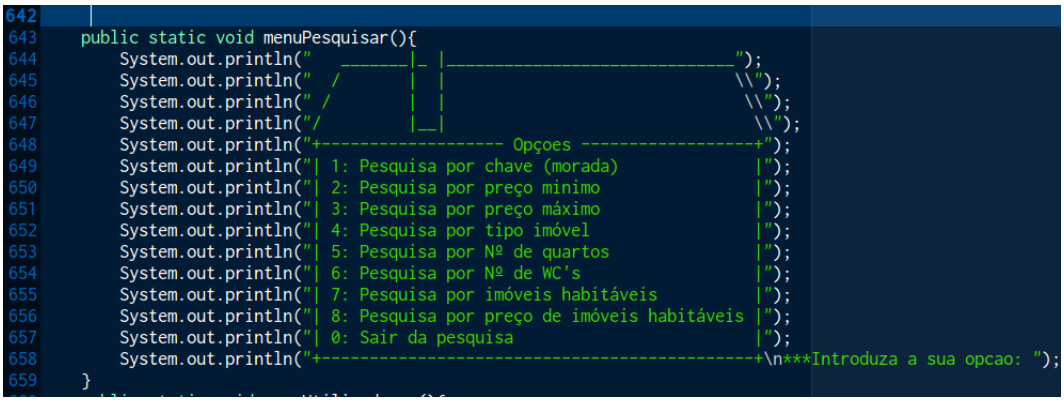
2 Primeiras classes

JAVA API

Vamos começar com a parte mais obvia de qualquer projeto.
Para conseguirmos compilar este programa, recorremos a algumas bibliotecas da Interface de programação de aplicações do JAVA.
Essas bibliotecas são as seguintes:

```
java.util;
java.io;
java.lang;
```

Fizemos grande parte deste projeto no programa BlueJ mas devido a algumas limitações deste programa, começamos a tomar outros rumos.



A interface situada na nossa classe principal com a main foi criada em Sublime Text (com a ajuda dos cursores múltiplos) em conjunto com o BlueJ e foi também utilizado ocasionalmente o IntelliJ IDEA.

Também recorremos à "Classe que abstrai a utilização da classe Scanner", criada pelo professor F. Mário Martins e a uma espécie de Feed de notícias com recurso a uma Queue que foi tirado de um tutorial.

Sem mais demoras, passaremos a uma das principais partes do nosso projeto, os imóveis.

Imóvel

"Imóvel"é a primeira classe que começamos a fazer e é uma das duas classes abstratas do nosso programa, para conseguirmos criar imóveis mais específicos, neste caso, para dividi-los em diversas categorias para determinarmos variáveis extra.

A classe é composta pelas variáveis estado, tipoImóvel, morada, precoPedido, precoMinimo e visitas.

O clone do imóvel é ligeiramente diferente dos clones de classes normais, sendo ele:

```
...
 abstract Imovel clone();
...
```

Também temos dois tipos de toString() diferentes (um para os vendedores e outro para os compradores, que mais tarde falaremos deles), para que possámos ocultar a informação do preço mínimo ao comprador.

```
public String toString(){
...
}
```

Também existe a opção de retornar, gravar e incrementar as visitas aos nossos imóveis para uso na classe Consulta, para que quando um utilizador qualquer aceder a um dado imóvel, o sistema consiga registar esse acesso.

Imovel está também povoado pelo típico equals() muito predefinido.

```
...
public boolean equals(Object o){
...
}
...
```

Como já está praticamente tudo falado da nossa classe abstrata, iremos então descer na nossa herança para elementos menos abstratos.

"Subclasses de Imóvel"

Comecemos com a subclasse Moradia. Esta foi a primeira subclasse que fizemos do abstract Imóvel. Como é habitual nas subclasses, fazemos construtores importando sempre as variáveis importantes da nossa classe abstrata através do método super(), subindo na herança para a nossa superclasse.

A subclasse Apartamento traz um tipo de variável um pouco habitual nas subclasses que fizemos que é neste caso o boolean. Esta subclasse também será importante para outra classe que falaremos daqui a pouco.

Tanto a classe Moradia como a classe Apartamento têm uma String em comum (Tipo). No entanto, os tipos das duas classes diferem uma da outra. Enquanto que a classe Moradia só aceita como tipos: Isolada,Germinada,Banda e Gaveto

```
if (!tipo.equals("Isolada")&&!tipo.equals("Germinada")&&
!tipo.equals("Banda")&&!tipo.equals("Gaveto"))
throw new TipoInvalidoException("Tipo de imóvel inválido");
```

A subclasse Apartamento só aceita estes seguintes tipos: Simples,Duplex,Triplex

```
if (!tipo.equals("Simples")&&!tipo.equals("Duplex")&&!tipo.equals("Triplex"))
throw new TipoInvalidoException("Tipo de imóvel inválido");
```

Nota: Embora fora das classes, se estas condições não forem aceites, elas irão gerar erros que mais tarde iremos falar.

A subclasse Terreno segue o mesmo príncipio das de cima, sendo o nome da String ligeiramente diferente, apropriadoPara, tal como a subclasse Loja que tem tiponegocio como nome da String.

Aqui entra a importância da Apartamento, pois uma classe Loja pode ser habitável. Ora, podemos fazer uma subclasse de Loja, ficando assim com uma subclasse LojaHabitavel, em que o principal objetivo é ligar dados do apartamento a uma dada loja.

Todas estas subclasses tem métodos em comum. Para facilitar a vida e para organização de código, nós criámos duas interfaces (Pesquisa e Habitável).

Pesquisas irá dar jeito para pesquisar o número de quartos ou o número de WC's de uma dada Moradia ou de um dado Apartamento.

Habitável por outro lado dá-nos jeito para saber se a nossa loja será habitável ou não e também retornar o preço pedido de um dado Imóvel (sendo Terreno a única exceção, a classe não tem a variável preçoPedido).

Utilizadores

Tivemos que implementar 3 outras classes para criar um cenário de gestão de utilizadores.

Temos uma classe abstrata chamada Utilizador que tem outras duas subclasses ligadas a ela, sendo elas Vendedor e Comprador.

No projeto ImOObiliaria, temos 3 estados que um utilizador pode usar:

- Utilizador não registado
- Vendedor registado
- Comprador registado

O facto de um utilizador estar registado ou não depende de um simples boolean.

Esse boolean diz-nos se o utilizador em questão já passou pelo estado de registo ou não.

> NOTA: quando construímos um utilizador (vendedor ou comprador) vazio, a flag registado começa sempre com o valor "false".

Temos metido um apontador para a nossa principal "base de dados"(se é que podemos chamar assim). É também uma declaração importante na nossa superclasse que se deixássemos de lado, provavelmente não funcionaria da maneira que queriamos (é de notar que os principais métodos de procura de imóveis estão quase todas nos utilizadores).
Esse apontador é feito usando:

```
...
    private Imobiliaria imob;
...
```

que nos permite então aceder sem problemas aos métodos principais da nossa classe Imobiliaria.
O resto das variáveis que usamos na superclasse Utilizadores é bastante normal e o básico para o funcionamento do projeto ImOObiliaria.
As subclasses tem pouco para mostrar em código, mas são extremamente precisas para que haja correta gestão dos dados.

Vendedor

Um vendedor registado consegue ter permissões avaçandas ao programa, nomeadamente aceder ao menu exclusivo dos vendedores e a algumas opções do menu dos leilões.
A classe Vendedor não contém grandes operações. Tem em vez disso dois portfólios distintos: um para armazenar os imóveis que estão em venda na base de dados e outro para registar os imóveis que já foram vendidos.
O vendedor também tem a possibilidade de remover diretamente um imóvel da base de dados, utilizando para isso o apontador para a classe Imobiliaria.

Comprador

Este estado também tem as suas únicas permissões não tão avançadas como um vendedor. Um comprador pode aceder ao menu exclusivo dos compradores, onde a criação, edição ou a eliminação de todos os imóveis não são permitidos.
Um comprador também consegue ter acesso aos leilões. No entanto dar início a um dado leilão ou encerrar um leilão não são operações permitidas. O comprador, ao contrário do vendedor, não tem nenhum portfólio de imóveis em venda ou vendidos mas sim uma lista de imóveis da escolha do comprador que possa gostar mais.
A classe comprador é das únicas classes que recorre à ordenação de listas, recorrendo para isso às classes Comparator que estão situadas no projeto.

```
... TreeSet <Imovel> (new ImovelStringComparator ()); ...
```

Ambas as classes recorrem ao método clone() para no caso de querermos registar informação de um dado utilizador sem alterar absolutamente nada do objeto original.

3 O motor do projeto

Imobiliaria (Base de dados)

Esta classe é a classe que provavelmente tem o maior número de métodos do nosso projeto (a grande responsável pela criação de dados).
As variáveis de instância desta classe são as que fazem o armazenamento de grande parte dos dados dos imóveis, utilizadores e um feed de notícias .

Quase todos os nossos getters recorrem a uma lista ou map auxiliares, de forma a que retornemos corretamente os dados enquanto que os setters são práticamente normais.

A classe faz-nos a gestão do inicio e do encerramento da sessão atual, não continuando se a base de dados dos utilizadores não tiver conteúdo ou se nenhuma sessão estiver ativa respetivamente.
Ao iniciarmos um programa, nós temos que criar um ponto em que haja um tipo de utilizador que não seja abstrato (pois não podemos instanciar uma classe abstrata) nem esteja registado.
Para isso recorremos a uma maneira de simular o que queriamos, criando um método init que inicia um Comprador sem qualquer informação mas que esteja ligado à nossa base de dados, em que a flag registado seja false para não ter permissões aos nossos menus especiais.

```
public void init(){atual = new Comprador ("","","","","",this,false);}
```

Ao terminar a sessão, a classe cria um ponto de situação em que o utilizador não tem informação e não está registado no sistema (boolean registado = false;).

```
public void fechaSessao(){... }
```

Os imóveis poderão ser registados, editados, listados ou removidos através desta classe, assim como os utilizadores também poderão ser registados e listados, mas não editados ou removidos.

O algoritmo do mapeamento de imóveis e do top de visitas dos imóveis também foi inserido nesta classe e recorre a apontadores dos dois tipos de utilizadores, para que possámos relacionar as visitas que os compradores ou os vendedores fazem aos imóveis, ou relacionar os vendedores e os seus respetivos imóveis em venda/reservados/vendidos.

Dois dos métodos mais importantes desta classe são os que nos permite a função de carregamento ou de salvaguarda de objetos, ou seja, guardar o estado inteiro do nosso programa. Para isso usamos então as classes predefinidas de JAVA ObjectInputStream e ObjectOutputStream.

ImOObiliaria (Main)

Este é o nosso arranque, a classe que tem o controlo dos principais menus de gestão do utilizador, de pesquisa, de vendedor e comprador, do leilão e do mapeamento de imóveis mais o top das consultas de imóveis do programa.

ImOObiliaria é a grande responsável por listar dados. No entanto, insereImovel é dos únicos métodos nesta função que utiliza o algoritmo de criação de dados em vez de listar, sendo a implementação desta função nesta classe um pouco estranha.

4 Outras classes importantes

Leilões

Aposta

Consulta
Tempo

5 Quando o código rebenta...

Exceptions

6 Conclusão

Principais dificuldades que não foram superadas neste projeto:

UTF-8 (utilizaÃ§Ã$o de caract§Ã$res especiais).
Leilões (comprador apostar de x em x minutos).
Bug estupido do double mostrar 0.0 em vez do valor devido.
Top X de imóveis não foi inteiramente implementado.

Dificuldades superadas:
fazer catch de uma exception no lugar errado...
Clone() (não dava o apontador do que queriamos)
Mapeamento de imóveis dava todos os imóveis para todos os vendedores.
Comprar um imóvel não alterava o estado do imóvel para reservado.

Este trabalho foi extremamente produtivo e só me trouxe vantagens para conhecer o mundo dos Exceptions, visto que começamos a "atinar"com eles.