

Programação Orientada aos Objetos

LCC Universidade do Minho

Ano Letivo 2015/2016

ImOObiliária — Trabalho prático

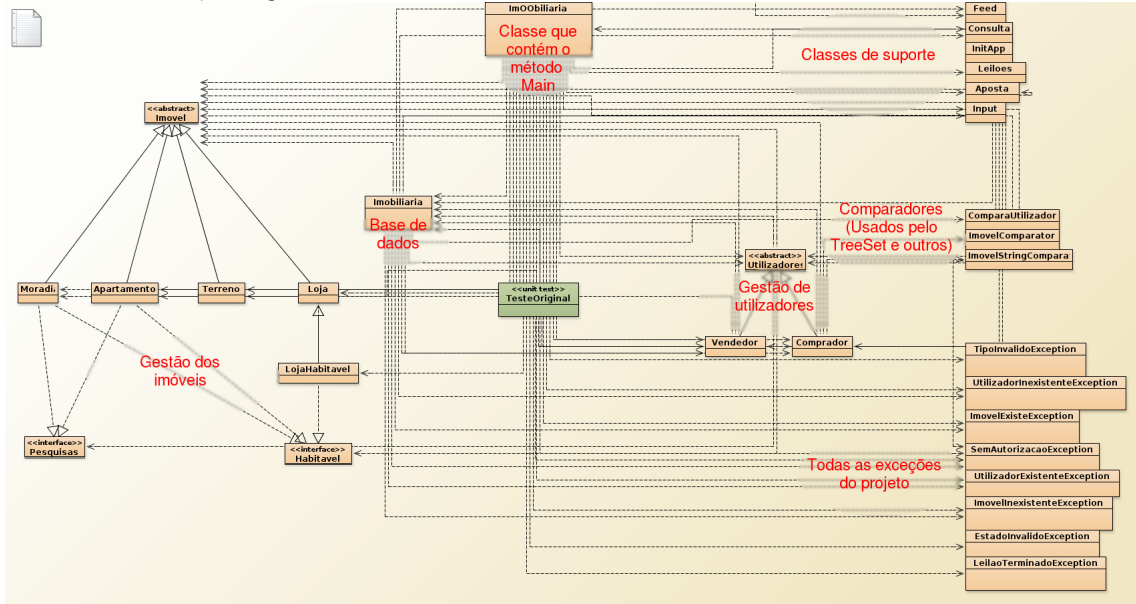
Grupo 5 62134 LCC Sérgio Oliveira
66698 LCC Carlos Silva

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.



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"

Começamos 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 princípio 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 5 estados que um utilizador pode usar:

- Utilizador não registado
- Vendedor
- Vendedor registado
- Comprador
- 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. 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 queríamos (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 registrado

Comprador

3 No coração do projeto...

Imobiliária

ImOObiliaria (Main)

4 Toca a licitar...

Leilões

Aposta

Consulta
Tempo

5 Quando o código rebenta...

Exceptions

6 Conclusão

Principais dificuldades que não foram superadas neste projeto:

UTF-8 (utiliza caracteres especiais).

Leilões (comprador apostar de x em x minutos).

Bug estúpido 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 queríamos)

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.