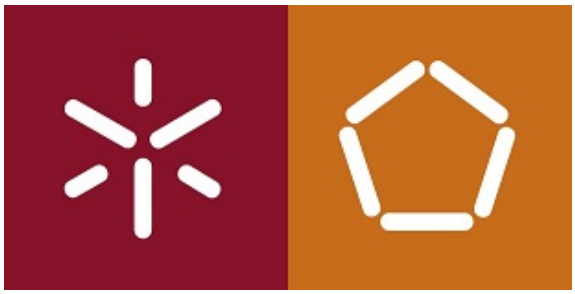


Universidade do Minho
Departamento de Informática

Sistema de Gestão de Recomendações

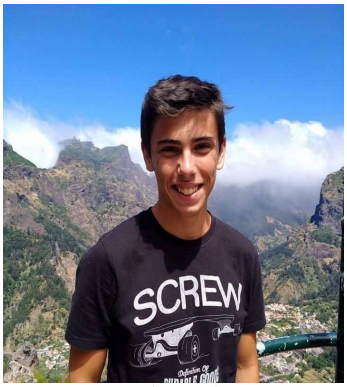
Laborátorio de Informática 3

Grupo 24

19 de Junho, 2021


Henrique Costa
(a93325)


Marco Esperança
(a93291)



José Pedro
(a93163)

Índice

Lista de Figuras

1. Introdução

Este trabalho foi realizado no âmbito da unidade curricular Laboratórios de Informática 3 e teve como objectivo a implementação de um sistema de gestão de recomendações, utilizando a linguagem de programação **Java**. Para a realização de tal trabalho, utilizamos conceitos aprendidos em tal unidade curricular, assim como em Programação Orientada aos Objetos. Dentre as ferramentas e mecanismos mais utilizados, é importante referir a utilização da *Java Collections Framework*.

2. Catálogos

2.1 Sobre o package *Catalogs*

O package *Catalogs*, que está localizado dentro do *model*, possui as classes correspondentes à "forma com que armazenamos as informações dos ficheiros lidos", assim como a validação dos mesmos. Assim, podemos separar em 2 grupos as classes deste *package*, de acordo com suas funcionalidades:

- `Catalog`
- `Businesses`
- `Reviews`
- `Users`
- `RowBusinesses`
- `RowReviews`
- `RowUsers`

Este primeiro grupo contém as classes de armazenamento de informação. Enquanto:

- `Parser`
- `SavedBusinesses`
- `SavedReviews`
- `SavedUsers`

... Correspondem às classes de leitura e validação de ficheiros. Nas próximas secções serão abordados com mais detalhes estes grupos.

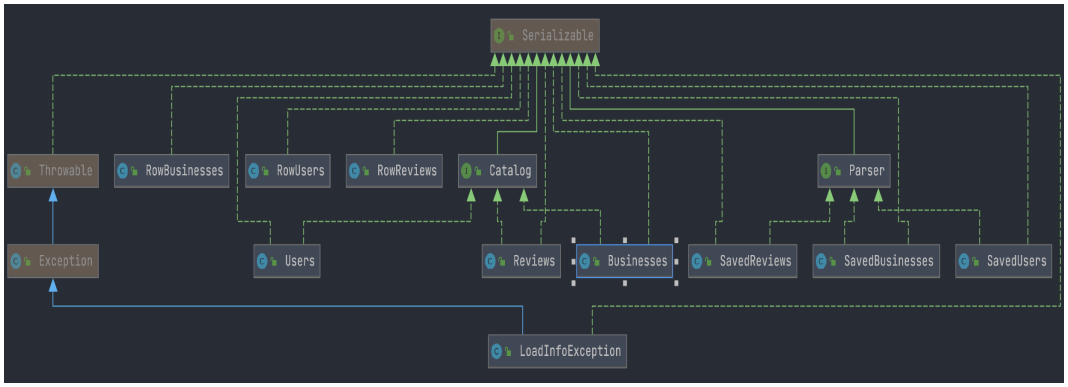


Figura 2.1: O package Catalogs

2.2 Classes de armazenamento

Os catálogos em si são nomeadamente compostos pelas classes **Businesses**, **Reviews**, **Users**. Numa fase inicial do projeto e, por critérios de expansibilidade, optamos por criar uma interface chamada **Catalogs**, de tal modo que, se por ventura tivermos de acrescentar um novo catálogo, o mesmo deveria implementar as funcionalidades e características básicas de um catálogo. Com o avanço do projeto, acabamos por não acrescentar nenhuma componente em tal interface mencionada, reconhecendo assim sua aparente "inutilidade"de momento. Porém, a base para tal ideia já está feita. Segue então um diagrama que representa esta relação:

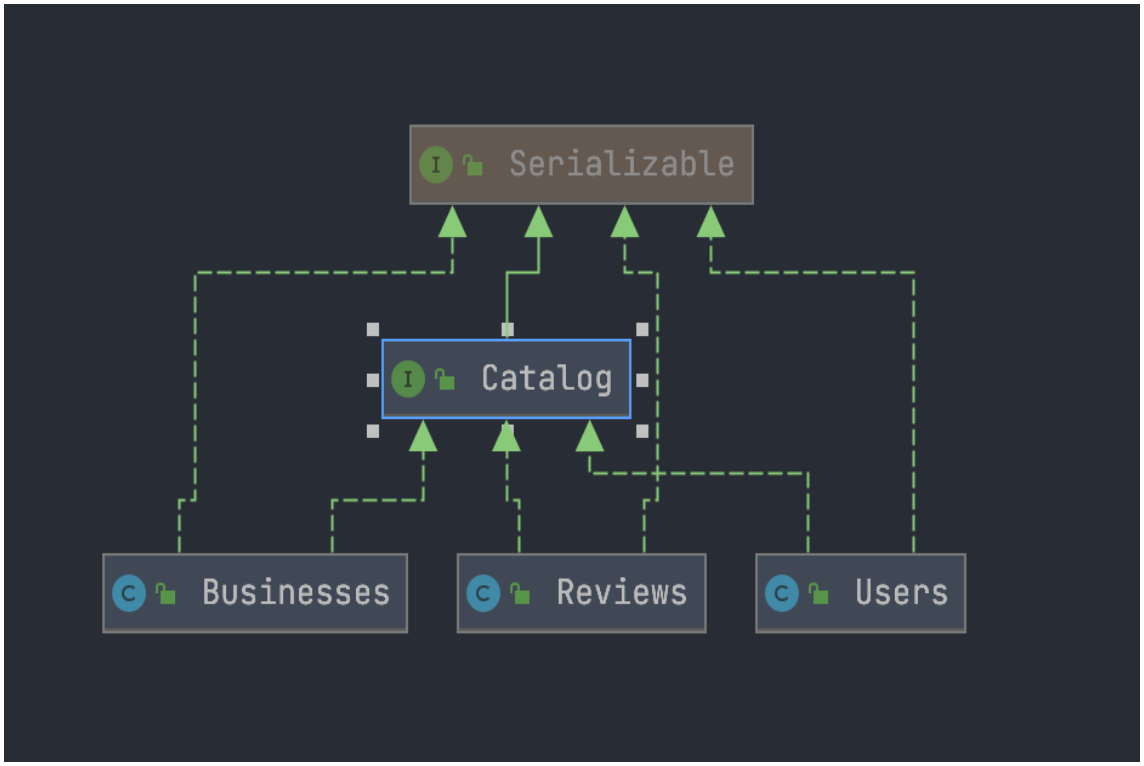


Figura 2.2: Classes de armazenamento

Por sua vez, a estrutura dos catálogos é muito similar. Cada catálogo possui como variável de instância uma "espécie de tabela", que é onde a informação dos ficheiros é armazenada. Optamos por representar tal tabela através de um `HashMap`. Isto porque temos uma relação implícita de (**chave, valor**), onde as **chaves** correspondem ao **id** de cada tipo de catálogo. Toma-se por exempllo o catálogo de Businesses, teremos então um `HashMap` composto de chaves que são os `businessesId` e valores que são `RowBusinesses`.

Falemos então das classes de **Row's**. Cada catálogo possui sua *Row*, que corresponde aos valores associados a cada chave da tabela anteriormente mencionada. A estrutura das classes que representam as *Row's* são de certa forma similares: possuem como variáveis de instância as componentes de cada linha da tabela de um catálogo. Tomando como exemplo o catálogo de *Business*, temos então as seguintes variáveis de instância para a classe `RowBusinesses`:

Para além das variáveis de instância, as classes das Row's possuem os habituais métodos `getters()`, `setters()`, `clone()`, `toString()`, `equals()` e `hashcode()`.

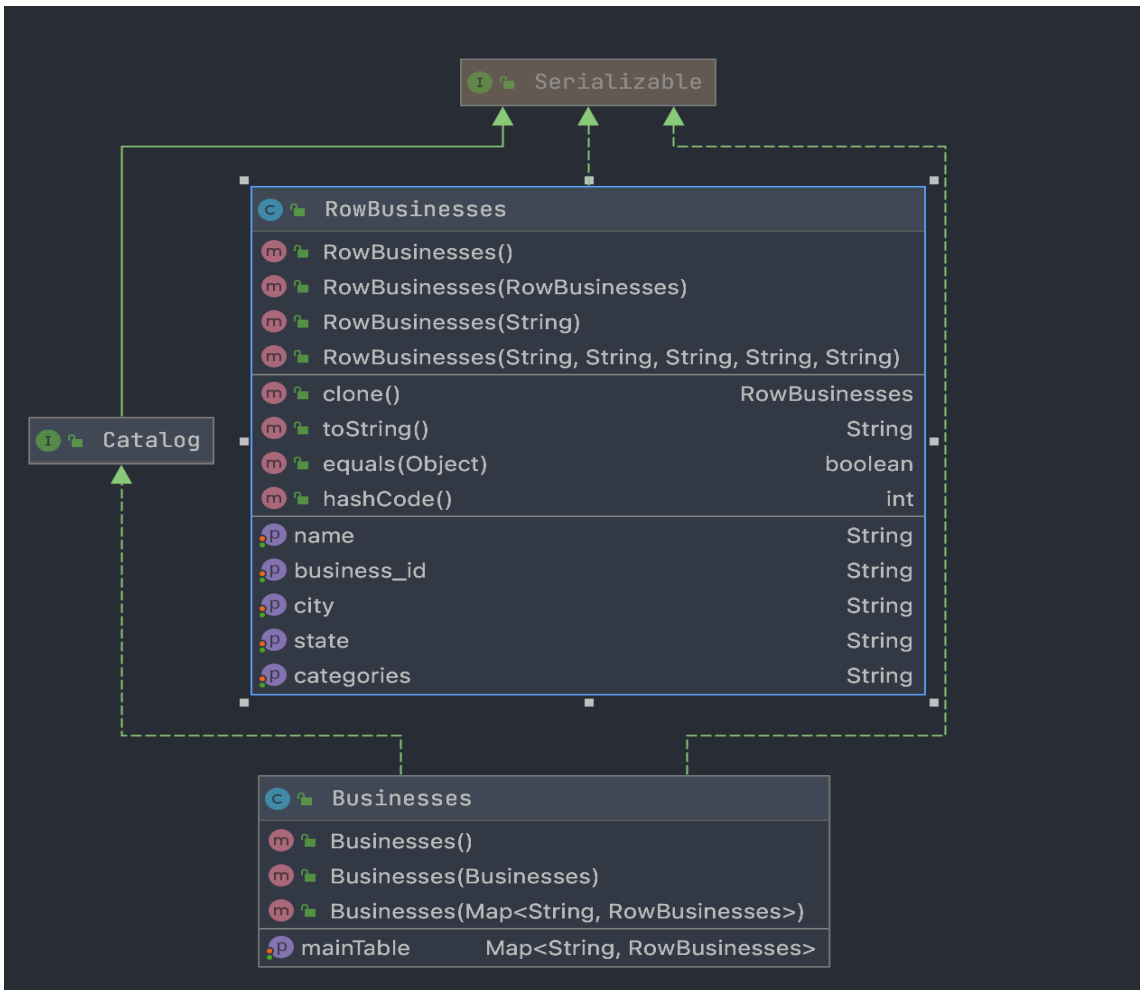


Figura 2.3: Relação Catálogo-Row

2.3 Classes de leitura e validação

Por forma a ler os ficheiros de texto, decidimos criar uma classe específica para cada tipo de ficheiro com a mesma estrutura. Todas continham uma variável de instância que contém o objeto a ser lido, sendo inicializada com o construtor vazio da classe final e posteriormente era completada informação a partir da implementação do método `parser`, da interface `Parser`. A vantagem de ter estas classes especializadas é que por um lado torna o *parser* mais independente da classe que se pretende obter e proporciona uma maior flexibilidade no tipo de ficheiro a ler. Por exemplo, um ficheiro que contenha informação sobre mais que um modelo de dados. Outra estratégia também seria válida, como fazer cada catálogo implementar um *parser*. Mas decidimos abordar esta questão, não pelo modelo de dados, mas sim pelo tipo de ficheiro a ser lido.

A validação feita não é a ideal. Fez-se a verificação quanto ao número de campos do ficheiro. No entanto, facilmente é possível extender o método `parser` para tornar a validação mais restrita e rigorosa.

3. Queries

De forma a possibilitar uma melhor organização, decidimos criar um *package* para as *queries*. Este *package* contém uma classe para cada *row* das queries, uma classe **Queries** que contém o código das *queries*, bem como algumas classes de *comparators*. Na classe **Queries** todas as *queries* vão receber o `Model`, sendo que cada uma tem uma *table* com a informação necessária para armazenar o resultado.

As várias tabelas (que são variáveis de instâncias da classe `Queries`) são constitúidas nomeadamente por *HashMaps*, de forma similar às tabelas citadas no capítulo 2. A diferença encontra-se nos valores associados às chaves das tabelas. Tais valores são as *Row's*, e cada *query* produz uma tabela com *Row's* diferentes. Assim, criamos uma classe de *Row* para cada *query*. Após a realização de uma *query*, o seu resultado fica armazenado na respetiva tabela (variável de instância da classe `Queries`). Segue-se então um diagrama, mostrando a constituição das classes `Queries` e `RowQuery1` como exemplo:

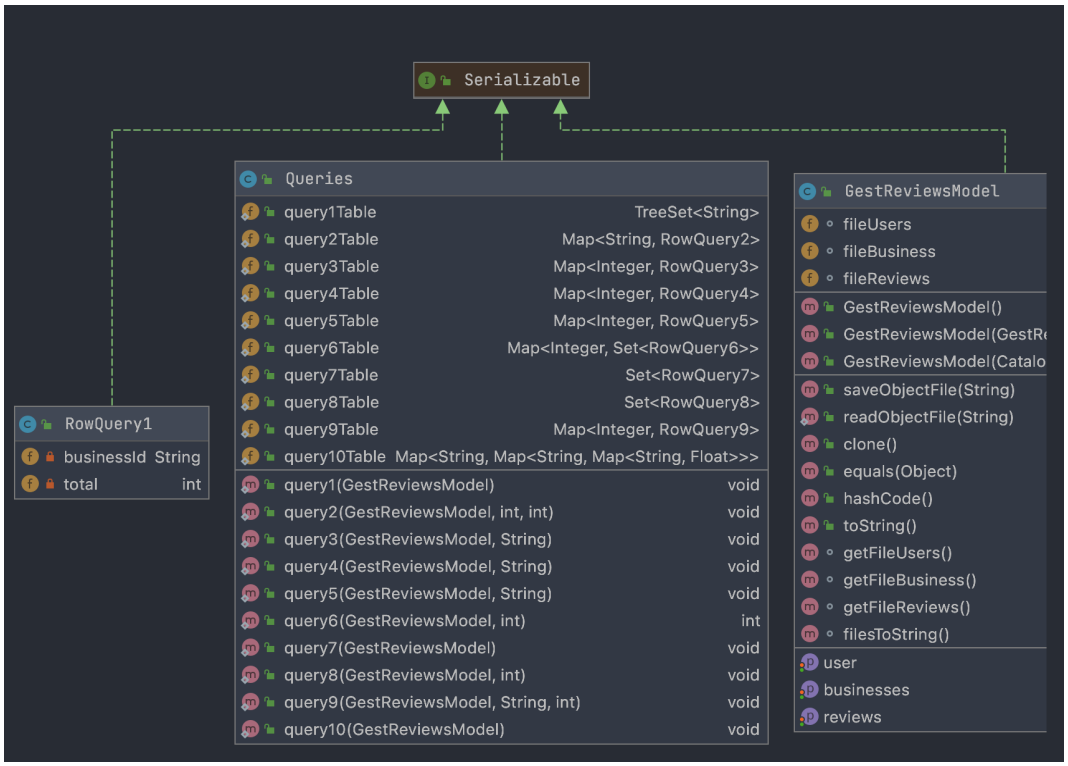


Figura 3.1: Relação Row-Queries-Model

4. Testes de perfomance

Foram realizadas 5 medições para cada comando/funcionalidade, sendo os resultados apresentados na tabela a média aritmética das mesmas (arrendodadas às centésimas quando necessário).

Processador	# *Cores*	Frequência(GHz)	RAM(GB)
Intel Core i7	6	2.6	16

Figura 4.1: Características do computador usado nos testes de desempenho

Comando/Funcionalidade	Tempo de execução(s)
Carregamento de users_full.csv, business_full.csv e reviews_1M.csv	22
1	6
2 (Ano: 2008, Mês: 07)	0.86
3 (id user: ak0TdVmGKo4pwqdJSTLwWw)	0.38
4 (id negócio: tXvdYGvlEceDljN8gt2_3Q)	0.4
5 (id user: ak0TdVmGKo4pwqdJSTLwWw)	0.4
6 (id negócio: tXvdYGvlEceDljN8gt2_3Q)	0.4
7 (id negócio: tXvdYGvlEceDljN8gt2_3Q)	15.2
8 (nº utilizadores: 5)	1.77
9 (id negócio: tXvdYGvlEceDljN8gt2_3Q, quantidade users: 5)	0.37
10 (id negócio: tXvdYGvlEceDljN8gt2_3Q, quantidade users: 5)	0.76

Figura 4.2: Tabela dos tempos de execução dos comandos/funcionalidades

5. Conclusão

Notamos uma grande diferença entre a realização deste projeto utilizando a linguagem de programação **C** e a linguagem **Java**. Em **Java** encontramos uma maior flexibilidade e ausência de preocupação no que respeita à gestão de memória, bem como os mecanismos em si das *queries* que foram facilitados pela *Java Collections Framework*. Reconhecemos não termos implementado da melhor forma as regras e bons hábitos de programar orientado aos objetos, porém conseguimos implementar todas as funcionalidades citadas no enunciado, concluindo assim o trabalho de forma completa.

A. Diagrama de Classes

Dada a dimensão do diagrama de classes final, optamos por incluir, ao longo deste relatório, pequenos diagramas relativos a apenas algumas das classes/*packages*. Segue-se o diagrama que inclui os principais *packages* utilizados pela aplicação.

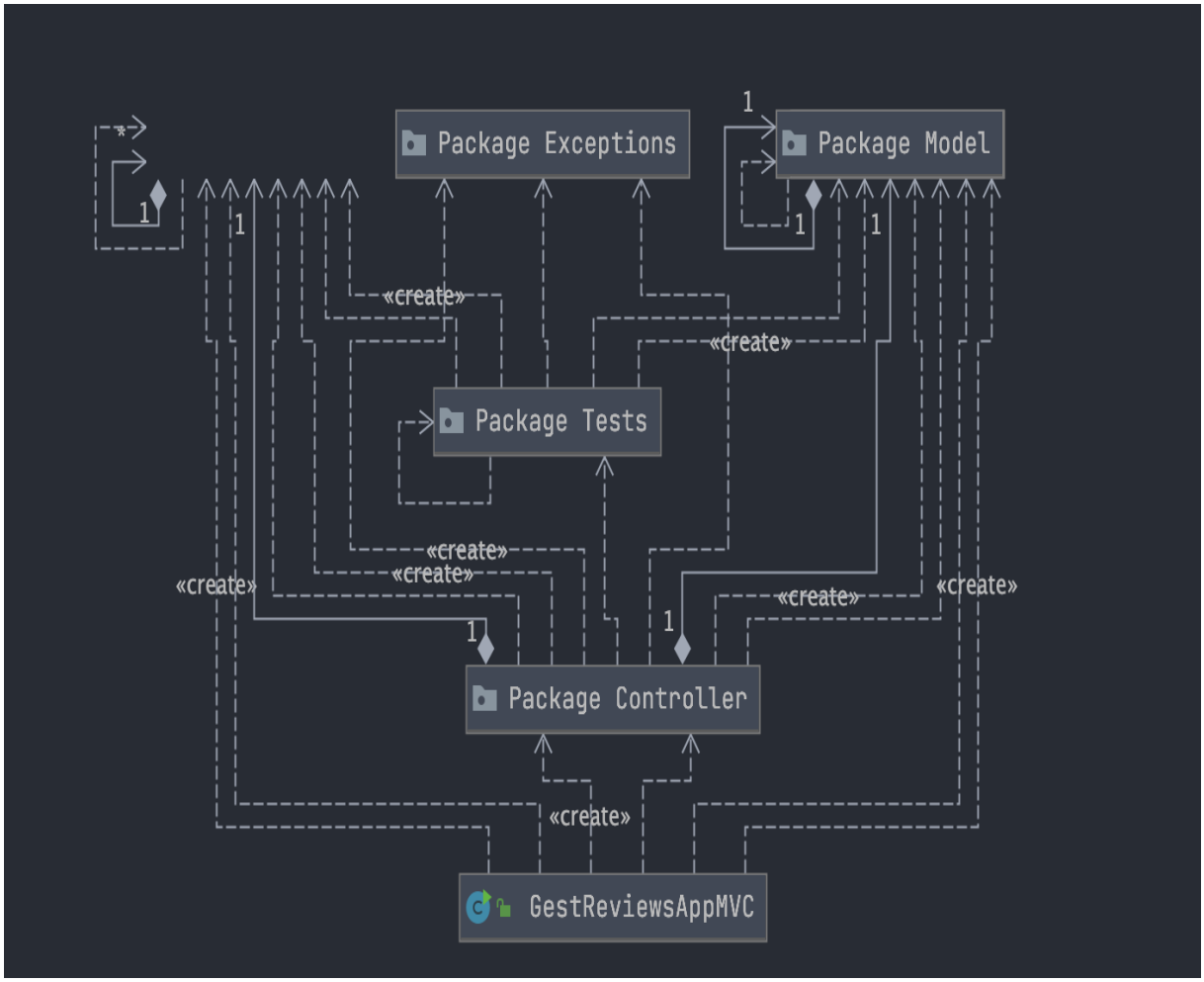


Figura A.1: Diagrama geral do Sistema de Gestão de Recomendações

De forma a compreender mais detalhadamente a organização arquitetural do *Model* de seguida apresenta-se o seu diagrama.

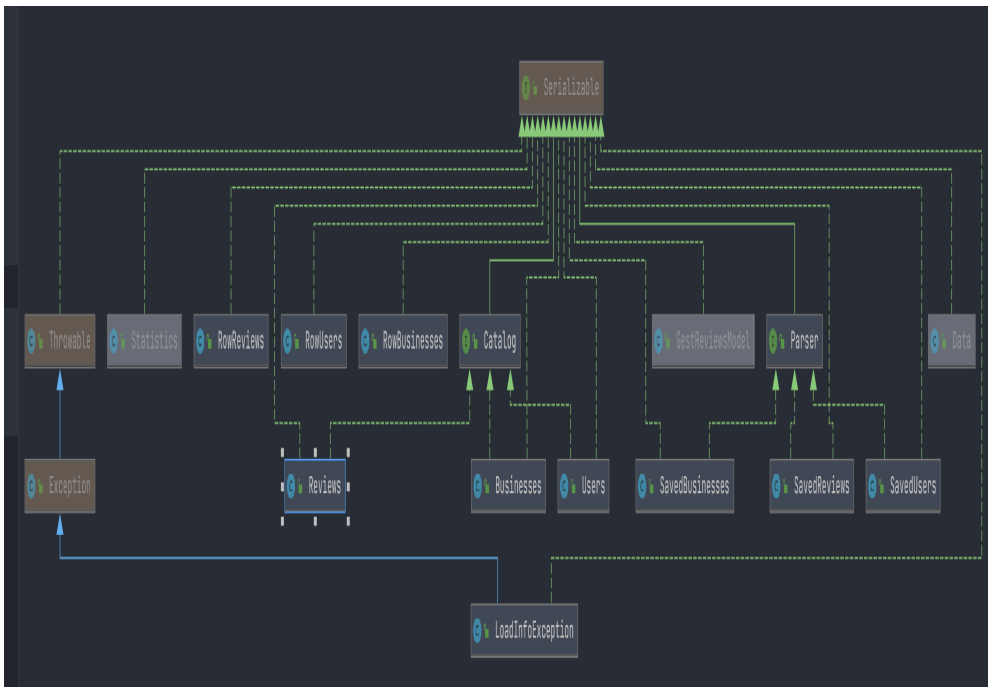


Figura A.2: Diagrama de Classes Do *Model*