

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE  
INSTITUTO METRÓPOLE DIGITAL  
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO  
LINGUAGEM DE PROGRAMAÇÃO II

RELATÓRIO: PROJETO – BOT PATRIMÔNIO TELEGRAM

CARMEM STEFANIE DA SILVA CAVALCANTE  
RAFAELLA SILVA GOMES  
VINICIUS RIBEIRO BULCÃO

NATAL  
NOVEMBRO 2019

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE  
INSTITUTO METRÓPOLE DIGITAL  
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO  
LINGUAGEM DE PROGRAMAÇÃO II

RELATÓRIO: PROJETO – BOT PATRIMÔNIO TELEGRAM

Relatório do projeto único referente à metade da nota para a segunda unidade da disciplina de Sistemas Operacionais do curso de Bacharelado em Tecnologia da Informação do Instituto Metrópole Digital da Universidade Federal do Rio Grande do Norte.

Grupo formado por:  
Noé Fernandes Carvalho Pessoa  
Pablo Emanuell Targino  
Rafaella Silva Gome

NATAL  
OUTUBRO 2019

## Sumário

1.Introdução.....	4
2.Modularização.....	4
2.1.Bibliotecas.....	4
2.2.Código.....	5
3.Implementação.....	6
4.Execução do programa.....	7
5.Discussão sobre o desenvolvimento.....	7
6.Trechos de código – Capturas de tela.....	7
7.Link.....	11

# 1. Introdução

Na terceira unidade da disciplina de Linguagem de Programação II, teremos como projeto final o desenvolvimento de um programa em linguagem Java, que consiste na implementação de um sistema de gerenciamento de patrimônio, onde deve ser criado e desenvolvido um *bot* (diminutivo de *robot*), utilizando a plataforma Telegram.

Nosso *bot*, identificado no sistema (Telegram) como `@goodsManagerBot`, terá como principal função permitir que o usuário seja capaz de cadastrar, consultar, atualizar e listar bens patrimoniais, bem como os setores e categorias relacionados a estes. Assim, devemos criar classes correspondentes a cada função executada pelo usuário, onde cada uma deve conter seus atributos próprios (nome, descrição e código) e dois atributos compartilhados (localização e categoria). Onde estes últimos apresentam atributos individuais, sendo eles: Localização (nome e descrição) e Categoria (nome, descrição e código). Além disso, devemos implementar métodos de busca, alteração, listagem de itens e geração de relatório.

Um ponto importante do nosso projeto é levar em conta a aplicação dos conhecimentos específicos vistos em aula. Dentre eles, a utilização de organização de pacotes, utilização de bibliotecas externas, herança, polimorfismo, classe abstrata e tratamento de exceções utilizando classes. Porém, alguns conceitos foram deixados de fora.

## 2. Modularização

Quanto a implementação do nosso sistema podemos começar pela modularização, sendo assim, nossos arquivos estão organizados da seguinte forma (Ver Figura 1):

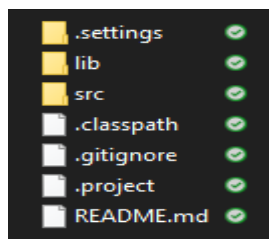


Figura 1: Modularização.

Nossa atenção deve ser voltada para duas pastas em particular, `lib` e `src`. Nelas encontraremos as bibliotecas externas que serão usadas e nossos códigos (classes, pacotes e arquivos de configuração), respectivamente.

### 2.1. Bibliotecas

Começando pela pasta `lib`, usamos quatro bibliotecas externas, conforme pode ser visto na Figura 2.

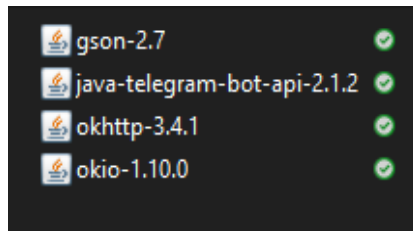


Figura 2: Bibliotecas.

A biblioteca “gson” é utilizada para converter objetos do tipo Java em representação JSON, uma vez que este, além de simples, também é baseado em subconjuntos do *JavaScript* e não depende de idiomas. Logo, isto faz do JSON uma linguagem de intercâmbio de dados ideal.

A biblioteca “java-telegram-bot-api” é o que nos permitirá fazer a interação com a API do Telegram por meio de suporte aos métodos da própria API. Dentre as funcionalidades disponíveis, esta biblioteca nos permite criar o *bot* passando o *token* que será recebido do @BotFather, enviar mensagens e atualizar informações. Em conjunto com esta biblioteca, utilizaremos a “OkHttp” para operações de rede com a criação do *bot*, onde esta precisa da biblioteca “Okio” para usar os recursos de entrada e saída (I/O) e os *buffers* de redimensionamento.

## 2.2. Código

Quanto à pasta “src” teremos algumas outras pastas, conforme podemos ver na Figura 3 abaixo:

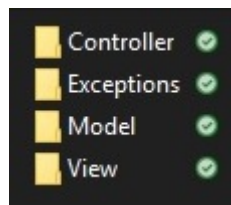


Figura 3: Conteúdo da pasta “src”.

A pasta “Controller” contém os arquivos de classes abstratas *Categories*, *Controller*, *Goods* e *Locations*. Já a pasta “Exceptions” contém as classes *EmptyList* e *OffTheList*, que herdam de *Exception* e são responsáveis pelo tratamento de exceções. Quanto a pasta “Model”, ela inclui os arquivos correspondentes às classes *Category*, *Good*, *Location* e *STATE*, onde este último inclui os status. Por fim, a pasta “View” contém a *Main* onde é criado o *bot* e realiza as funções de cadastro, busca e listagem de bens.

### 3. Implementação

No que diz respeito à implementação, a primeira parte consiste da criação das classes e seus atributos. Para efeitos de visualização, podemos ilustrar de acordo com a Figura 4, abaixo:

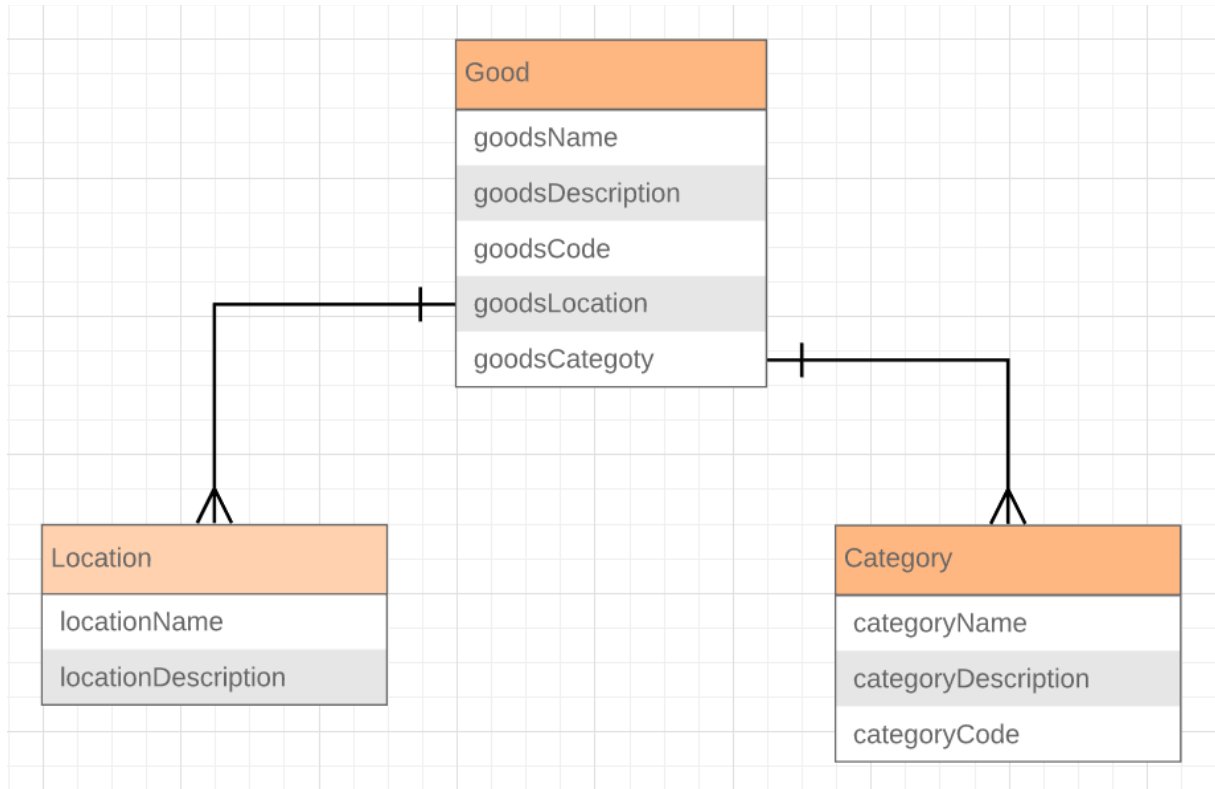


Figura 4: Organização das classes e seus atributos.

Como a parte de criação de classes é algo mais simples, podemos seguir para as funcionalidades da *Main*. Nela criamos inicialmente o *bot* e seus objetos, bem como as informações de acesso e com isso podemos fazer a comunicação entre usuário e *bot*. Em seguida criamos controladores para gerenciar os dados correspondentes a cada classe, criamos um objeto para executar o comando no Telegram para obter as mensagens e uma lista para armazená-las.

Por fim, teremos um laço *for* que analisará cada ação recebida por mensagem. Dentre as ações possíveis teremos:

- “castrar\_bem”;
- “cadastrar\_localizacao”;
- “cadastrar\_categoria”;
- “listar\_bens”;

- “listar\_localizacoes”;
- “listar\_categorias”;
- “listar\_bens\_por\_localizacao”;
- “buscar\_bem\_por\_codigo”;
- ”buscar\_bem\_por\_nome”;
- “buscar\_bem\_por\_descricao”;
- “trocar\_localizacao\_de\_bem”;

Cada vez que uma ação é chamada, ela recebe uma atualização de “STATE” que deverá direcioná-la até o local onde esta deverá ser efetivamente executada. Nos casos em que a ação exige uma busca, antes de ser executada, é feita uma checagem por meio de tratamento de exceções, para verificar se de fato o dado buscado existe e em seguida a ação é executada normalmente conforme solicitado.

## **4. Execução do programa**

Para verificar as funcionalidades do programa na prática é preciso ter uma conta válida no Telegram. Após se conectar ao aplicativo podemos contatar o nosso bot através do usuário @goodsManagerBot. Iniciada a interação com o bot por meio da mensagem “/start”, é possível realizar ações enviando mensagens programadas.

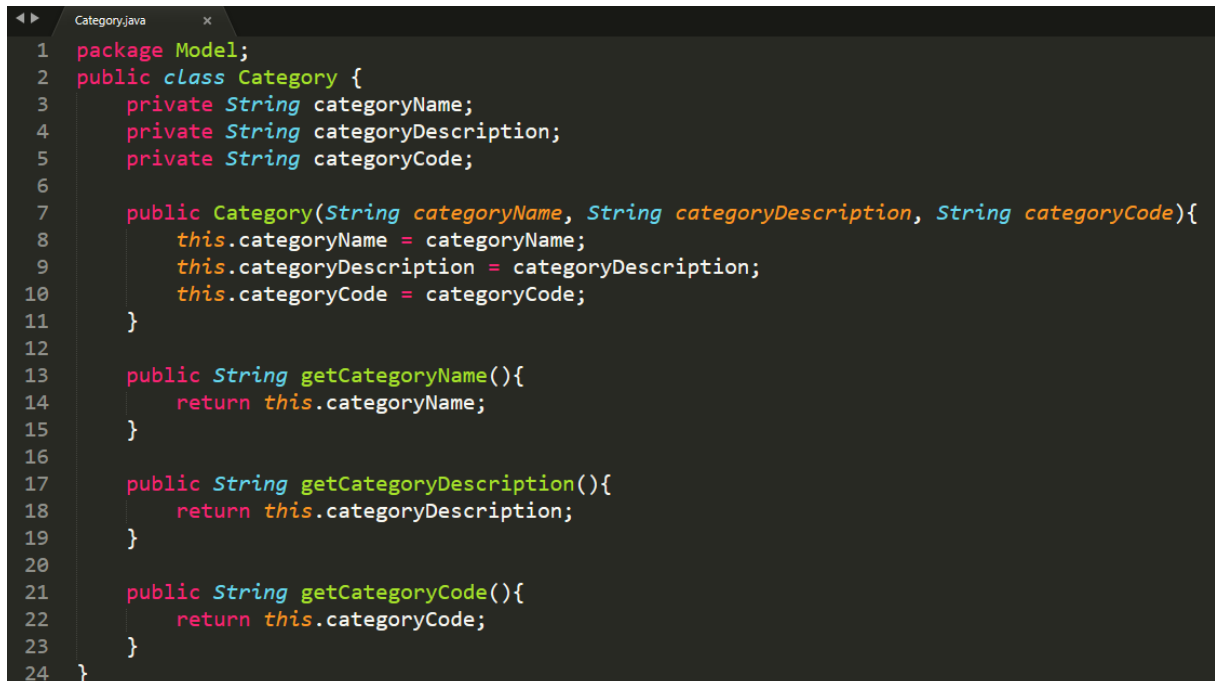
## **5. Discussão sobre o desenvolvimento**

De maneira unânime, o maior problema enfrentado durante o projeto foi a utilização da API do Telegram e suas bibliotecas. Inicialmente tivemos dificuldades para entender o funcionamento da troca de mensagens entre usuário e *bot* e a chamada das ações. Uma solução encontrada foi a criação de uma máquina de estado, por meio de um Enum chamado STATE.

Passado este obstáculo, a criação das classes foi relativamente simples de implementar, bem como o tratamento de exceções, herança e polimorfismo. Houve certa dificuldade quanto a busca e listagem, mas logo foi contornado.

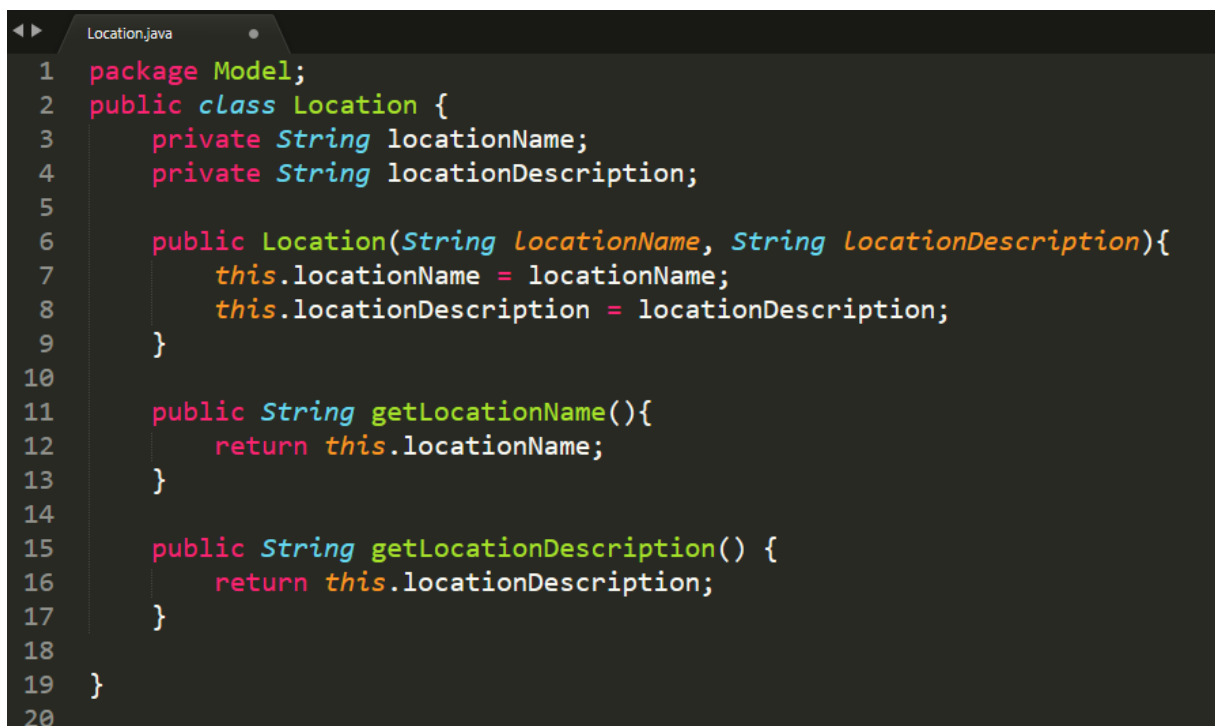
## **6. Trechos de código – Capturas de tela**

Abaixo teremos algumas capturas de tela para mostrar um pouco da estrutura do código. Devemos ressaltar alguns pontos importantes, como as classes, a herança, classe abstrata, utilização da biblioteca e importação de pacotes, criação do bot e execução de ações recebidas via mensagem.



```
1 package Model;
2 public class Category {
3     private String categoryName;
4     private String categoryDescription;
5     private String categoryCode;
6
7     public Category(String categoryName, String categoryDescription, String categoryCode){
8         this.categoryName = categoryName;
9         this.categoryDescription = categoryDescription;
10        this.categoryCode = categoryCode;
11    }
12
13    public String getCategoryName(){
14        return this.categoryName;
15    }
16
17    public String getCategoryDescription(){
18        return this.categoryDescription;
19    }
20
21    public String getCategoryCode(){
22        return this.categoryCode;
23    }
24 }
```

Figura 5: Classe Category



```
1 package Model;
2 public class Location {
3     private String locationName;
4     private String locationDescription;
5
6     public Location(String locationName, String locationDescription){
7         this.locationName = locationName;
8         this.locationDescription = locationDescription;
9     }
10
11    public String getLocationName(){
12        return this.locationName;
13    }
14
15    public String getLocationDescription() {
16        return this.locationDescription;
17    }
18
19 }
20
```

Figura 6: Classe Location



```

1 package Model;
2 public class Good {
3     private String goodsName;
4     private String goodsDescription;
5     private String goodsCode;
6     private String goodsLocation;
7     private String goodsCategory;
8
9     public Good(String goodsName, String goodsDescription, String goodsCode,
10                String goodsLocation, String goodsCategory){
11         this.goodsName = goodsName;
12         this.goodsDescription = goodsDescription;
13         this.goodsCode = goodsCode;
14         this.goodsLocation = goodsLocation;
15         this.goodsCategory = goodsCategory;
16     }
17
18     public String getGoodsName(){
19         return this.goodsName;
20     }
21
22     public String getGoodsDescription(){
23         return this.goodsDescription;
24     }
25
26     public String getGoodsCode(){
27         return this.goodsCode;
28     }
29
30     public String getGoodsLocation() {
31         return this.goodsLocation;
32     }
33
34     public String getGoodsCategory() {
35         return this.goodsCategory;
36     }
37 }

```

Figura 7: Classe Good

```

1 package Model;
2
3 public enum STATE {
4     NULL,
5     WAITING_GOOD_NAME, WAITING_GOOD_DESCRIPTION, WAITING_GOOD_CODE,
6     WAITING_LOCAL_NAME, WAITING_LOCAL_DESCRIPTION,
7     WAITING_CATEGORY_NAME, WAITING_CATEGORY_DESCRIPTION, WAITING_CATEGORY_CODE,
8     LIST_LOCATIONS, LIST_CATEGORIES, LIST_GOODS_BY_LOCATION,
9     WAITING_LOCATION, WAITING_CATEGORY,
10    WAITING_GOOD_BY_CODE, WAITING_GOOD_BY_NAME, WAITING_GOOD_BY_DESCRIPTION,
11    WAITING_GOOD_BY_CODE_FOR_EXCHANGE,
12    WAITING_NEW_LOCATION;
13 }

```

Figura 8: Enum STATE

```

1 package Controller;
2
3 import java.util.ArrayList;
4
5 import Exceptions.EmptyList;
6 import Exceptions.OffTheList;
7
8
9 public abstract class Controller {
10
11     //public abstract void registerLocation(String attributeOne, String attributeTwo);
12
13     public abstract ArrayList<?> list();
14
15     public abstract boolean findByName(String searchName) throws OffTheList;
16
17     public abstract void sizeOfList() throws EmptyList;
18
19 }

```

Figura 9: Classe abstrata Controller

```

27
28 public class Main {
29     public static void main(String[] args) {
30
31         /*
32          * Declaração de variáveis que serão utilizadas para receber os valores
33          * que o usuário digita.
34          */
35         STATE status = STATE.NULL;
36         String name = "null";
37         String description = "null";
38         String code = "null";
39         String location = "null";
40         String category = "null";
41         Good searchGood = null;
42         boolean search = false;
43
44         //Criação do objeto bot com as informações de acesso
45         TelegramBot bot = TelegramBotAdapter.build("909350681:AAHgxlxlR67oZtaC6EwyBURPEbMjVILUCA");
46
47         /*
48          * GetUpdatesResponse: objeto responsável por receber as mensagens
49          * SendResponse: objeto responsável por gerenciar o envio de respostas
50          * BaseResponse: responsável por gerenciar o envio de ações do chat
51          */
52         GetUpdatesResponse updatesResponse;
53         //SendResponse sendResponse;
54         //BaseResponse baseResponse;
55
56         //controle de off-set, isto é, a partir deste ID serão lido as mensagens pendentes na fila
57         int m = 0;
58

```

Figura 10: Início da Main com a declaração das variáveis, criação do bot e seus objetos.

```

82 while (true){
83
84     updatesResponse = bot.execute(new GetUpdates().limit(100).offset(m));
85
86     List<Update> updates = updatesResponse.updates();
87
88     /**
89     * Cada input do usuário através do bot resulta em uma ação, esse
90     * for funciona para analisar essas ações e realizar a ação
91     * necessária.
92     */
93     for (Update update : updates) {
94
95         //atualização do off-set
96         m = update.updateId()+1;
97
98
99         /**
100        * Comando para cadastrar o BEM, move o STATE para onde se recebe o nome do bem
101        * a ser cadastrado
102        */
103        if(update.message().text().equals("/cadastrar_bem")) {
104            bot.execute(new SendMessage(update.message().chat().id(), "Digite o nome do bem: "));
105            status = STATE.WAITING_GOOD_NAME;
106        }

```

Figura 11: Continuação da Main com o objeto "updatesResponse" que executa comando no Telegram para receber mensagens, em seguida temos a criação de uma lista que armazena as mensagens e por fim o início do laço for que analisa cada ação solicitada por meio de mensagem.

```

EmptyList.java
1 package Exceptions;
2
3 public class EmptyList extends Exception {
4
5     /**
6     *
7     */
8     private static final Long serialVersionUID = 1L;
9
10    public EmptyList(String message) {
11        super(message);
12    }
13 }
14

OffTheList.java
1 package Exceptions;
2
3 public class OffTheList extends Exception{
4
5     /**
6     *
7     */
8     private static final Long serialVersionUID = 1L;
9
10    public OffTheList(String message) {
11        super(message);
12    }
13 }
14

```

Figura 12: Classes de exceção EmptyList e OffTheList.

## 7. Link

[Repositório GitHub - Telegram bot with java](#)