



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

BACHARELADO EM ENGENHARIA DE SOFTWARE

Sistema de disponibilidade de aluguel de roupas

Alexandre Dantas dos Santos

Bergony Bandeira de Melo Silva

Natal - RN

Abril de 2022

Sumário

1. Introdução	3
2. Arquitetura, implementação e execução	3
2.1 Arquitetura	3
2.2 Implementação	6
2.3 Execução	14
3. Conclusão	16

1 - Introdução

O objetivo deste trabalho é aplicar os conhecimentos adquiridos em sala de aula na utilização da biblioteca Java RMI para simular a comunicação entre um cliente e um servidor. A aplicação consiste em simular uma rede de estabelecimentos de aluguel de roupas, onde o cliente fornece informações como estilo de roupa e as datas de início e término do aluguel. O servidor, por sua vez, retorna uma lista de roupas disponíveis, juntamente com as respectivas lojas, para o período de tempo selecionado pelo cliente, bem como o valor total do aluguel.

O trabalho descreve a arquitetura do sistema, a implementação do projeto, a execução e conclui com uma descrição geral do assunto tratado nos tópicos abordados.

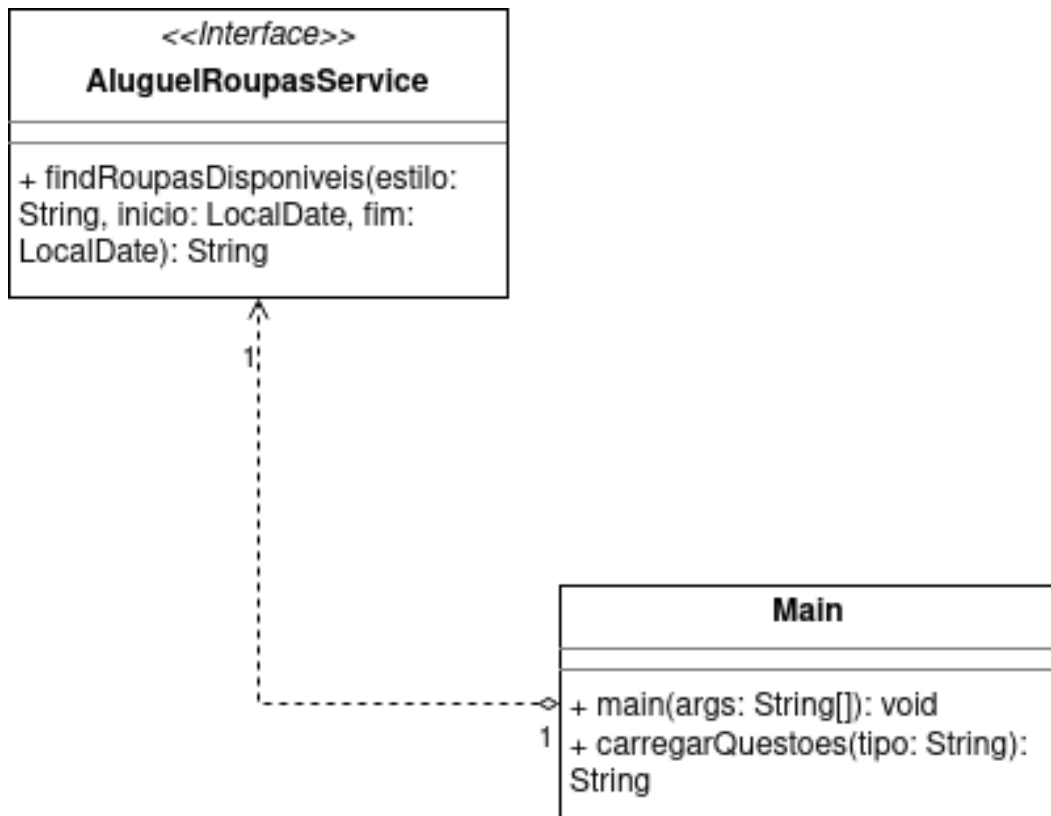
2 - Arquitetura, implementação e execução

2.1 - Arquitetura

O projeto foi desenvolvido utilizando-se do modelo de arquitetura cliente-servidor. O projeto também foi construído utilizando-se da comunicação síncrona como principal estratégia de execução da aplicação, pois entendeu-se ser a melhor abordagem para o problema apresentado. A seguir, serão apresentados os detalhes gerais da arquitetura do projeto, bem como o seu processo de desenvolvimento e execução.

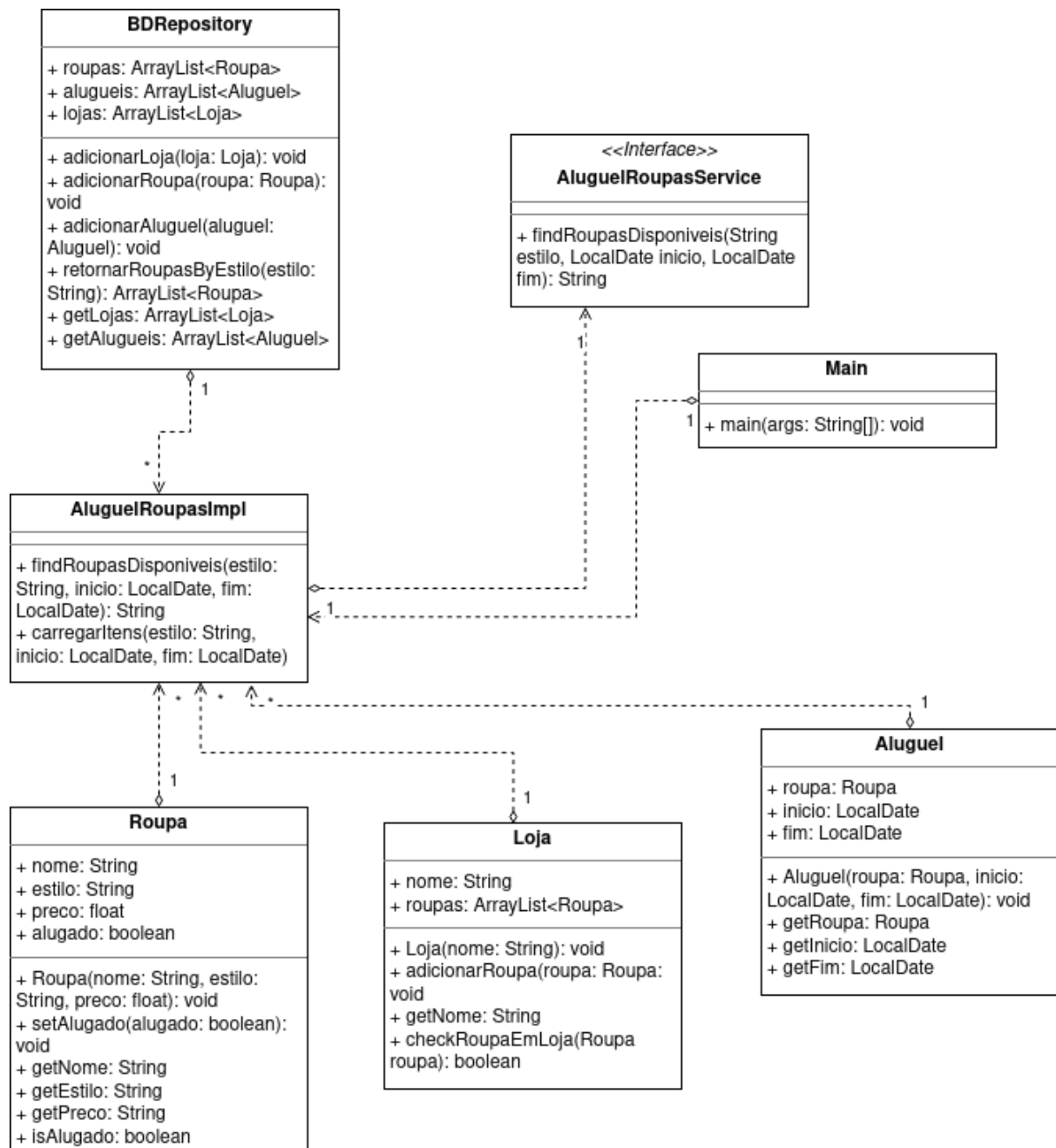
A arquitetura do projeto pode ser interpretada de duas maneiras: a partir da perspectiva do cliente e a partir da perspectiva do servidor. A seguir, será apresentada a visão do cliente.

Na figura 1, temos a representação da arquitetura do ponto de vista do cliente. Nesta visão temos a presença da classe: *Main*, responsável pelas chamadas de comunicação junto a aplicação *Server*. Há também a presença da interface: *AluguelRoupasService*, interface remota responsável pela declaração dos métodos que o cliente pode invocar.



(Figura 1: diagrama de classes do cliente)

Já na figura 2, temos uma representação do diagrama de classes do servidor. Podemos observar na definição desse projeto, a existência da classe *Main*, necessária para inicialização do servidor e da interface *AluguelRoupasService*, que também encontra-se presente no projeto *Server*. Esta interface estabelece o contrato de comunicação entre o cliente e o servidor e estende a classe *Remote* presente no pacote: *java.rmi.Remote*.



(Figura 2: diagrama de classes do servidor)

O projeto também possui as classes de domínio: *Loja*, *Roupa* e *Aluguel*, que representam as entidades atuantes durante a execução da aplicação. Há também a presença da classe: *BDRRepository*, entidade responsável pelo armazenamento dos objetos/dados utilizados na sessão. Por fim, temos também a classe: *AluguelRoupasImpl*. Esta classe, que estende *UnicastRemoteObject* do pacote: *java.rmi.server.UnicastRemoteObject* também implementa a interface: *AluguelRoupasService*.

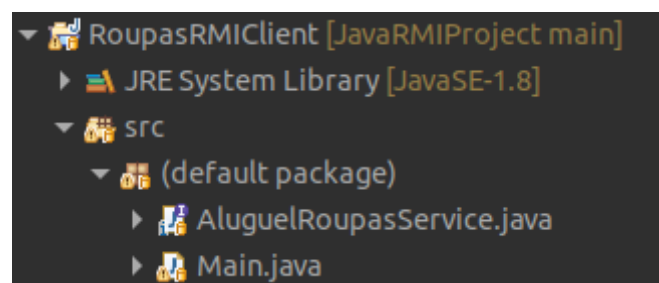
2.2 - Implementação

Quanto à implementação, o projeto foi desenvolvido com base em dois subprojetos que representam o cliente e o servidor, nomeados como **RoupasRMICliente** e **RoupasRMIServer**. O projeto foi construído utilizando a versão 8 do Java, em conjunto com a tecnologia Java RMI, para chamada remota dos objetos.

No Servidor, é implementada a lógica de negócio da aplicação. O servidor responde aos pedidos do cliente, apresentando as informações dos tipos de trajes desejados de acordo com a necessidade do usuário. Internamente, o servidor também cria e manipula as instâncias dos objetos necessários para o funcionamento da aplicação.

Já o Cliente, implementa a fase de interação com o usuário, realizando o consumo das funções do servidor através de uma interface. Os clientes selecionam um tipo de traje desejado, e também selecionam o período desejado para consulta da disponibilidade do aluguel.

A estrutura do projeto do *Client* encontra-se detalhada na figura 3. Este projeto possui uma estrutura bem simples, onde temos localizadas a classe: *Main* e a interface remota *AluguelRoupasService*.



(Figura 3: estrutura do projeto do cliente)

Neste ponto, a interface *AluguelRoupasService* realiza a declaração do método: *findRoupasDisponiveis()*, método este necessário para estabelecimento da comunicação entre cliente e servidor. O código dessa interface encontra-se presente na figura 4.

```

public interface AluguelRoupasService extends Remote {
    String findRoupasDisponiveis(String estilo, LocalDate inicio, LocalDate fim) throws RemoteException;
}

```

(Figura 4: interface AluguelRoupasService.java)

Como pode-se observar na figura 4, a interface possui um método declarado que recebe o estilo de traje desejado pelo cliente e as datas de início e fim para aluguel do traje.

Já a classe: *Main* possui dois métodos utilizados na interface de acesso ao usuário: *main()* e *carregarQuestoes()*. O método *carregarQuestoes()* é apresentado na figura 5.

```

public static String carregarQuestoes(String tipo) {
    Scanner in2 = new Scanner(System.in);
    String date="";
    System.out.print("Informe a data " + tipo + " para o aluguel - (Formato: DD/MM/AAAA): ");
    date = in2.nextLine();
    return date;
}

```

(Figura 5: método carregarQuestoes())

A função deste método é atuar como suporte da classe para instanciação das perguntas referentes a seleção de datas, evitando assim a repetição de código redundante.

Na figura 6 vemos parte da implementação do método: *main()*.

```

Registry registry = LocateRegistry.getRegistry("localhost");

AluguelRoupasService client = (AluguelRoupasService) registry.lookup("Roupas");
LocalDate start, end;

int option = 100;
String inicio, fim, saida;

```

(Figura 6: método main() - parte inicial)

Neste trecho são declaradas as variáveis para utilização no menu de perguntas do usuário: *option*, *início*, *fim* e *saída* e são apresentados também a declaração do objeto: *Registry*, útil para estabelecimento da conexão com o *server*, e também o comando: *registry.lookup*, útil para acesso ao nome lógico do serviço.

Na figura 7, temos a continuação da apresentação do método *main()*.

```
try (Scanner in = new Scanner(System.in)) {
    while (option != 0) {

        System.out.println("Caro cliente, informe a opção desejada para aluguel:");
        System.out.println("1 - Aluguel de roupas esportivas.");
        System.out.println("2 - Aluguel de roupas tradicionais.");
        System.out.println("3 - Aluguel de roupas para festas.");
        System.out.println("0 - Sair.");
        System.out.print("Opção desejada: ");
        option = in.nextInt();

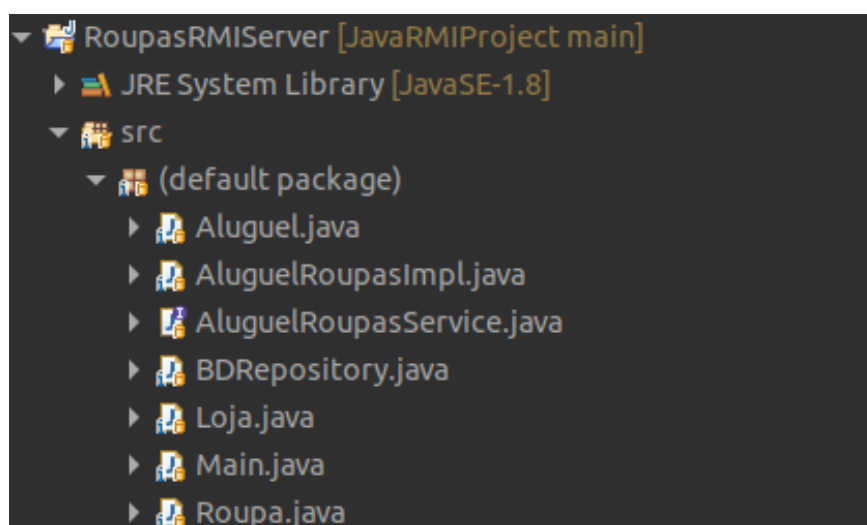
        switch (option) {

            case 1:
                inicio = carregarQuestoes("início");
                fim = carregarQuestoes("fim");
                start = LocalDate.parse(inicio, DateTimeFormatter.ofPattern("dd/MM/yyyy"));
                end = LocalDate.parse(fim, DateTimeFormatter.ofPattern("dd/MM/yyyy"));
                saida = client.findRoupasDisponiveis("Esportivo", start, end);
                System.out.println(saida);
                break;
        }
    }
}
```

(Figura 7: método *main()* - parte principal)

Este método contém o esqueleto do menu de perguntas apresentado para o usuário. Onde são exibidas as opções de seleção disponíveis para o usuário. Cada opção de escolha é trabalhada com a estrutura *switch(case)*, e o método remoto *findRoupasDisponiveis()* é chamado passando o tipo de traje e o período desejado para aluguel. Observar também a chamada para o método: *carregarQuestoes()* descrito anteriormente.

Já o nosso *server* possui a seguinte estrutura:



(Figura 8: estrutura do projeto do servidor)

Este projeto possui uma estrutura mais complexa, onde temos a existência de diversas classes e da interface remota *AluguelRoupasService*.

A classe *Main* atua no estabelecimento de inicialização do servidor, executando os comandos úteis para registro do serviço remoto. Seu detalhamento encontra-se na figura 9.

```
public class Main {  
  
    public static void main(String[] args) throws RemoteException, AlreadyBoundException {  
  
        Registry registry = LocateRegistry.createRegistry(1099);  
        AluguelRoupasService server = new AluguelRoupasImpl();  
        registry.rebind("Roupas", (Remote) server);  
        System.out.println("Servidor RMI pronto!");  
    }  
}
```

(Figura 9: estrutura da classe Main no servidor)

Esta classe possui apenas o método *main()* declarado em seu corpo, e pode lançar também as exceções: *RemoteException* e *AlreadyBoundException*. Por fim, é exibida no console a mensagem: “Servidor RMI pronto”.

Dentro do projeto do servidor, temos também as classes: *Loja*, *Roupa* e *Aluguel*. Tais classes representam o domínio da aplicação e mapeiam os atributos e métodos *getters* e *setters* dos objetos criados. O corpo da classe: *Roupa* encontra-se presente na figura 10.

```
public class Roupa {  
  
    String nome;  
    String estilo;  
    float preco;  
    boolean alugado;  
  
    public Roupa(String nome, String estilo, float preco) {  
        super();  
        this.nome = nome;  
        this.estilo = estilo;  
        this.preco = preco;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public String getEstilo() {  
        return estilo;  
    }  
  
    public float getPreco() {  
        return preco;  
    }  
  
    public boolean isAlugado() {  
        return alugado;  
    }  
  
    public void setAlugado(boolean alugado) {  
        this.alugado = alugado;  
    }  
}
```

(Figura 10: estrutura da classe Roupa)

Outra classe bastante importante para o projeto é a classe: *BDRepository*. Tal classe possui um papel bastante útil pois permite o armazenamento da lista de objetos criados no projeto. Seu detalhamento é mostrado na figura abaixo:

```
public class BDRepository {

    public ArrayList<Roupa> roupas = new ArrayList<>();
    public ArrayList<Loja> lojas = new ArrayList<>();
    public ArrayList<Aluguel> alugueis = new ArrayList<Aluguel>();

    public void adicionarRoupa(Roupa roupa) {

        roupas.add(roupa);
    }

    public void adicionarLoja(Loja loja) {

        lojas.add(loja);
    }

    public void adicionarAluguel(Aluguel aluguel) {

        alugueis.add(aluguel);
    }

    public ArrayList<Roupa> retornarRoupasByEstilo(String estilo) {
        ArrayList<Roupa> roupasReturn = new ArrayList<>();

        for (Roupa roupa : roupas) {
            if (roupa.getEstilo().equals(estilo)) {
                roupasReturn.add(roupa);
            }
        }

        return roupasReturn;
    }
}
```

(Figura 11: estrutura da classe BDRepository)

Observar no detalhamento de parte da classe, os atributos: roupas, lojas, e alugueis do tipo: *ArrayList()*. Estas listas são úteis no armazenamento dos objetos criados junto às classes de domínio. Entre os métodos existentes, destaca-se também o método *retornarRoupasByEstilo()*, que é utilizado na consulta das roupas existentes para um estilo/tipo específico.

Dentro ainda do *Server*, temos a classe: *AluguelRoupasImpl* que executa toda a lógica de negócio do servidor. Esta classe possui dois métodos: O *findRoupasDisponiveis()* e o método *carregarItens()*.

O método: *findRoupasDisponiveis()*, apresentado na figura 12, sobrescreve o método de mesmo nome declarado na interface remota, e executa também o método *carregarItens()* que irá realizar todo o processamento de execução do servidor.

```
@Override
public String findRoupasDisponiveis(String estilo, LocalDate inicio, LocalDate fim) throws RemoteException {
    String msg = carregarItens(estilo, inicio, fim);
    return msg;
}
```

(Figura 12: método findRoupasDisponiveis())

O método *carregarItens()* é apresentado a partir da figura 13. Em razão do seu tamanho, seu detalhamento será mostrado por partes.

```
public String carregarItens(String estilo, LocalDate inicio, LocalDate fim) {

    BDRepository rp = new BDRepository();

    // Criação das lojas
    Loja loja1 = new Loja("Alguém Veste");
    Loja loja2 = new Loja("Ponto Xique");
    Loja loja3 = new Loja("Natal Rigor");

    // Criação das roupas
    Roupas roupa1 = new Roupas("Short preto básico para corrida", "Esportivo", 15);
    Roupas roupa2 = new Roupas("Caça legging", "Esportivo", 50);
    Roupas roupa3 = new Roupas("Camisa musculação", "Esportivo", 30);
    Roupas roupa4 = new Roupas("Camisa social padrão", "Tradicional", 40);
    Roupas roupa5 = new Roupas("Vestido tradicional", "Tradicional", 70);
    Roupas roupa6 = new Roupas("Calça Jeans padrão", "Tradicional", 30);
    Roupas roupa7 = new Roupas("Terno completo", "Festa", 100);
    Roupas roupa8 = new Roupas("Vestido para noiva", "Festa", 200);
    Roupas roupa9 = new Roupas("Smoking tradicional", "Festa", 150);

    // Criando localdates para aluguéis
    LocalDate localDate1 = LocalDate.of(2023, 03, 01);
    LocalDate localDate2 = LocalDate.of(2023, 03, 15);

    // Registrando aluguéis
    Aluguel aluguel1 = new Aluguel(roupa1, localDate1, localDate2);
    Aluguel aluguel2 = new Aluguel(roupa2, localDate1, localDate2);
}
```

(Figura 13: método carregarItens() - início)

O trecho apresentado mostra o caminho percorrido durante a execução do método. Podemos ver destacadamente a inicialização dos objetos necessários: lojas, roupas e aluguéis, mapeados em memória para execução da aplicação. Tais objetos são também adicionados a uma instância da classe: *BDRepository*.

Abaixo temos a continuação do detalhamento da execução do método: `carregarItens()`.

```
// Adicionando lojas ao repositório
rp.adicionarLoja(loja1);
rp.adicionarLoja(loja2);
rp.adicionarLoja(loja3);

// Adicionando aluguéis ao repositório
rp.adicionarAluguel(aluguel1);
rp.adicionarAluguel(aluguel2);

ArrayList<Roupa> roupas = new ArrayList<>();
roupas = rp.retornarRoupasByEstilo(estilo);

ArrayList<Loja> lojas = new ArrayList<>();
lojas = rp.getLojas();

ArrayList<Aluguel> alugueis = new ArrayList<>();
alugueis = rp.getAlugueis();

long dias = ChronoUnit.DAYS.between(inicio, fim);

if (dias == 0)
    dias = 1;

// Percorrendo as listas para identificar as roupas alugadas.
for (Aluguel aluguel : alugueis) {
    for (Roupa roupa : roupas) {
        if (aluguel.getRoupa().equals(roupa)) {
            if (!(aluguel.getInicio().isAfter(fim) || aluguel.getFim().isBefore(inicio))) {
                roupa.setAlugado(true);
            }
        }
    }
}
```

(Figura 14: método `carregarItens()` - detalhamento)

Nesta lógica temos o detalhamento do trecho de código responsável pela identificação das roupas alugadas. Os objetos recém criados são adicionados a instância do *BDRRepository* e em seguida três listas são criadas recebendo seus valores. Observar que no caso das roupas, apenas as roupas do estilo informado são carregadas. É calculado também a diferença entre o número de dias do período de aluguel informado pelo usuário. Tal cálculo é útil no cálculo final do aluguel da roupa junto à loja. O último trecho serve para identificar as roupas alugadas. Caso alguma das roupas do estilo informado possua registro de aluguel dentro do período informado, aquela roupa será definida como alugada (Observar o método booleano: `roupa.setalugado(true)`).

Por fim, na figura 15 é apresentado o trecho final do método: *carregarItens()*.

```
String msg = "\nSeguem as roupas encontradas para o estilo selecionado: " + estilo + ". \n";

for (Loja loja : lojas) {
    for (Roupa roupa : roupas) {
        if (loja.checkRoupaEmLoja(roupa)) {
            if (!roupa.isAlugado()) {
                msg = msg + "Produto: " + roupa.getNome() + " de preço (diária): " + roupa.getPreco()
                    + " reais, localizado na loja: " + loja.getNome() + ". (Valor total da locação: R$"
                    + (roupa.getPreco() * dias) + ")" + " \n";
            } else {
                msg = msg + "Produto: " + roupa.getNome() + " de preço (diária): " + roupa.getPreco()
                    + " reais, localizado na loja: " + loja.getNome()
                    + ". (Roupa alugada no período. Indisponível para locação)" + "\n";
            }
        }
    }
}

return msg;
```

(Figura 15: método *carregarItens()* - final)

Este método retorna em uma string várias informações úteis para o cliente. São apresentadas as informações de valor do produto e sua localização (loja), além do seu preço unitário e valor total de aluguel. Para os produtos alugados (ver trecho: *!roupa.isAlugado()*) é apresentada a informação de que o traje está indisponível para aluguel. O método interno: *checkRoupaEmLoja()* checa se aquele traje (da coleção de roupas) está presente na loja especificada.

2.3 - Execução

Ao acionar a execução do *Server*, é exibida apenas uma mensagem informando que o servidor está em execução. Podemos ver isto na figura 16.

A terminal window with a dark background. The text 'Servidor RMI pronto!' is displayed in a light-colored monospace font at the top of the window.

(Figura 16: representação da execução do server)

Ao executar o *Client*, é exibido o menu de interação junto ao cliente. Inicialmente, será apresentado para o usuário as opções de seleção do tipo de traje, conforme descrito na figura 17.

A terminal window with a dark background. The text is as follows:
Caro cliente, informe a opção desejada para aluguel:
1 - Aluguel de roupas esportivas.
2 - Aluguel de roupas tradicionais.
3 - Aluguel de roupas para festas.
0 - Sair.
Opção desejada:

(Figura 17: tela de seleção do tipo de traje)

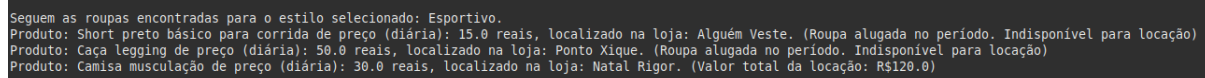
Caso o usuário selecione a opção: “0” é dada a opção para este sair da aplicação. Já as opções: 1, 2 e 3 correspondem aos tipos de trajes desejados. Neste caso, correspondente às respectivas categorias: Esportivo, tradicional e festa.

Após seleção do tipo de traje desejado, será apresentado também junto ao cliente, as opções de data início e fim para aluguel da roupa. Tal situação é descrita na figura 18.

A terminal window with a dark background. The text is as follows:
Caro cliente, informe a opção desejada para aluguel:
1 - Aluguel de roupas esportivas.
2 - Aluguel de roupas tradicionais.
3 - Aluguel de roupas para festas.
0 - Sair.
Opção desejada: 1
Informe a data início para o aluguel - (Formato: DD/MM/AAAA): 01/03/2023
Informe a data fim para o aluguel - (Formato: DD/MM/AAAA): 05/03/2023

(Figura 18: tela de seleção do período desejado)

Após seleção do período desejado, o *server* irá apresentar os trajes disponíveis dentro do período informado, conforme descrito na figura 19.



```
Seguem as roupas encontradas para o estilo selecionado: Esportivo.  
Produto: Short preto básico para corrida de preço (diária): 15.0 reais, localizado na loja: Alguém Veste. (Roupa alugada no período. Indisponível para locação)  
Produto: Caça leggings de preço (diária): 50.0 reais, localizado na loja: Ponto Xique. (Roupa alugada no período. Indisponível para locação)  
Produto: Camisa musculação de preço (diária): 30.0 reais, localizado na loja: Natal Rigor. (Valor total da locação: R$120.0)
```

(Figura 19: tela de apresentação dos trajes)

O sistema apresentará a informação da localização do traje (loja), o preço unitário do item e o seu valor total para locação (Baseado nas datas informadas). Para os produtos indisponíveis, será apresentada também a informação de que o produto se encontra indisponível para locação no período informado.

Por fim, ainda é apresentado mais uma vez junto ao usuário, o menu principal de interação para possibilidade de seleção de uma nova consulta.

3 - Conclusão

O projeto aqui desenvolvido proporcionou um melhor aprofundamento sobre o entendimento dos sistemas distribuídos, em especial sobre o uso do Java RMI. Graças ao projeto, foi permitido também uma boa compreensão sobre os conceitos de comunicação síncrona e assíncrona dentro dos projetos de sistemas distribuídos. Todavia, algumas dificuldades foram apresentadas, como desconhecimento inicial acerca do funcionamento do Java RMI, e dificuldades na diferenciação entre os modelos síncrono e assíncrono.

Outro ponto importante, é a falta de armazenamento entre as sessões de execução do programa. Não obstante, embora seja possível implementar o sistema para seu funcionamento em conjunto com um banco de dados, optamos por não fazê-lo, por entendermos que tal ajuste “fugiria” um pouco do escopo definido para este trabalho.