

Desenvolvimento de Sistemas de Informação Distribuídos (ACH2147)

Sistema de informações sobre componentes (parts) usando Remote Method Invocation (RMI) e Java

Maria Eduarda Rodrigues Garcia 11796621
Mirela Mei 11208392

Junho 2023

1 Resumo

O objetivo do trabalho consiste em realizar um sistema de informações sobre peças, distribuído em múltiplos servidores - tendo cada um implementado um repositório de informações sobre peças. Cada servidor, nesse caso, possui um repositório, fornecendo acesso às peças armazenadas neste. O usuário, portanto, pode conectar-se a um cliente, acessar um servidor à sua escolha, realizar ações dentro deste escopo e acessar outros servidores a partir de subpeças de peças existentes no servidor em que se encontra

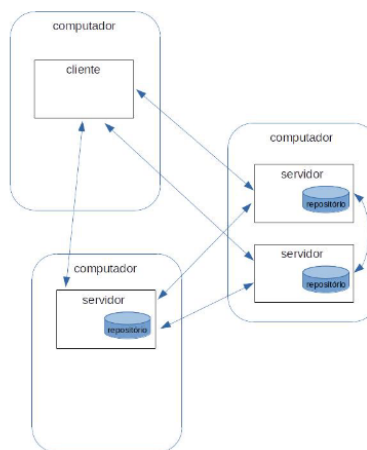


Figura 1: Esquematização do sistema distribuído.

2 Estrutura do projeto

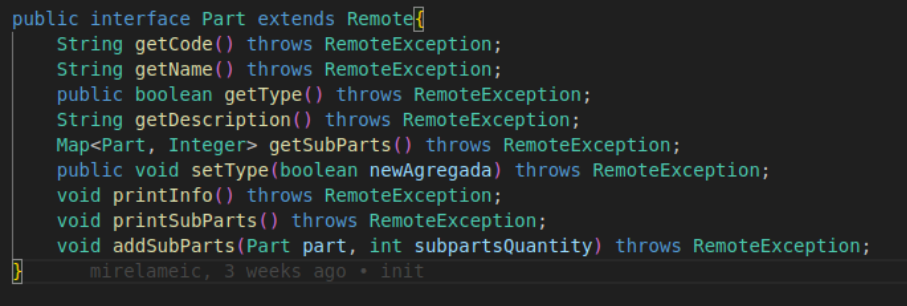
2.1 Estrutura de classes

O projeto foi dividido em duas principais pastas:

- bin: pasta com arquivos .class
- src: pasta com arquivos .java

E em sete classes:

- **Part:** Interface chamada "Part" que estende a interface "Remote" da biblioteca Java RMI. Ela define métodos remotos que podem ser invocados em objetos distribuídos. A interface possui métodos para obter o código, nome e descrição de uma parte, bem como uma lista de seus subcomponentes. Essa interface permite a comunicação remota entre objetos que implementam a interface "Part" e os clientes que desejam acessar esses objetos remotamente. Os métodos declarados nessa interface são:



```
public interface Part extends Remote {
    String getCode() throws RemoteException;
    String getName() throws RemoteException;
    public boolean getType() throws RemoteException;
    String getDescription() throws RemoteException;
    Map<Part, Integer> getSubParts() throws RemoteException;
    public void setType(boolean newAgregada) throws RemoteException;
    void printInfo() throws RemoteException;
    void printSubParts() throws RemoteException;
    void addSubParts(Part part, int subpartsQuantity) throws RemoteException;
}
```

Figura 2: Captura de tela do terminal.

- `getCode`: retorna o atributo code de um objeto Part.
- `getName`: retorna o atributo name de um objeto Part.
- `getType`: retorna o atributo agregada de um objeto Part.
- `getDescription`: retorna o atributo description de um objeto Part.
- `getServer`: retorna o atributo server de um objeto Part.
- `getSubparts`: retorna o atributo subParts de um objeto Part.
- `setType`: atualiza o valor do atributo agregada de um objeto Part.
- `printInfo`: imprime os atributos de um objeto Part, inclusive a qual repositório pertence e quantas subpeças têm.
- `printSubParts`: imprime as instâncias do objeto Subpart relacionados à peça corrente.
- `addSubParts`: adiciona subpeças (objetos Subpart) à peça corrente.

- **PartImpl:** Implementa os métodos definidos na interface "Part" e retorna os valores dos campos correspondentes quando esses métodos são invocados.
 - **code:** Uma string que representa o código único da peça.
 - **name:** Uma string que representa o nome da peça.
 - **agregada:** Um booleano que indica se a peça é agregada (true) ou primitiva (false).
 - **description:** Uma string que representa a descrição da peça.
 - **server:** Uma string que representa o nome do repositório da peça.
 - **subParts:** É um HashMap que mapeia objetos Part (subpeças) a inteiros (quantidade). É usado para armazenar as subpeças da peça atual, juntamente com suas quantidades.
- **PartRepository:** Interface que estende a interface "Remote" da biblioteca Java RMI. Ela define métodos remotos que podem ser invocados para manipular um repositório de partes. A interface possui métodos para adicionar uma parte, obter uma parte com base em um código e obter todas as partes armazenadas no repositório. Essa interface permite a comunicação remota para gerenciar um repositório de partes através do Java RMI. Os métodos declarados nessa interface são:

```
public interface PartRepository extends Remote{
    void addPart(Part part) throws RemoteException;
    Part getPart(String code) throws RemoteException;
    String getName() throws RemoteException;
    List<Part> getAllParts() throws RemoteException;
    Integer getPartsQuantity() throws RemoteException;
}
```

Figura 3: Captura de tela do terminal.

- **addPart:** adiciona instância do objeto Part à PartRepository (repositório) corrente.
- **getPart:** dado um código, retorna o atributo nome da peça (objeto Part) buscado.
- **getName:** retorna o atributo nome do repositório (objeto PartRepository).
- **getAllParts:** retorna todas as instâncias de Part armazenadas em PartRepository.
- **getPartsQuantity:** retorna a quantidade de peças existente no repositório.

- **PartRepositoryImpl:** Implementa os métodos definidos na interface "PartRepository" e retorna os valores dos campos correspondentes quando esses métodos são invocados.
 - **name:** Uma string que representa o nome do repositório de peças.
 - **parts:** É um HashMap que mapeia strings (códigos de peças) a objetos Part (peças). É usado para armazenar todas as peças do repositório.
- **UserInterface:** Responsável por interagir com o usuário, exibir mensagens no console e obter comandos de entrada do usuário. Ela fornece métodos simples para exibir mensagens, ler comandos e lidar com erros de exceção. A interface implementa os métodos:

```
public class UserInterface {
    private static Scanner scanner;

    public UserInterface() {
        scanner = new Scanner(System.in);
    }

    public static String getUserCommand() {
        System.out.print(">> ");
        return scanner.nextLine().trim();
    }

    public static void displayMessage(String message) {
        System.out.println(message);
    }

    public static void displayError(String errorMessage, Exception exception) {
        System.err.println(errorMessage);
        System.err.println(exception.getMessage());
    }

    public static void printLine() {
        System.out.println();
    }
}
```

Figura 4: Captura de tela do terminal.

- **getUserCommand:** recebe o comando enviado pelo usuário.
- **displayMessage:** apresenta na tela uma mensagem através do comando `System.out.println`.
- **displayError:** apresenta na tela uma mensagem de erro através do comando `System.err.println`.
- **printLine:** pula uma linha através do comando `System.out.println`.

- **Server:** Representa um servidor que cria e disponibiliza um repositório de partes remotamente através do Java RMI.

```
public class Server {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("Usage: Server <server_name> <port>");
            System.exit(1);
        }

        String serverName = args[0];
        int port = Integer.parseInt(args[1]);

        try {
            PartRepository repository = new PartRepositoryImpl(serverName);
            Registry registry = LocateRegistry.createRegistry(port);
            Naming.rebind(serverName, repository);

            UserInterface.displayMessage("Server bound");
        } catch (Exception e) {
            UserInterface.displayError("Server Exception.", e);
        }
    }
}
```

Figura 5: Captura de tela do terminal.

Essa classe chamada **Server** é responsável por iniciar um servidor de objetos remotos e vincular o objeto remoto ao registro, utilizando a tecnologia RMI (Remote Method Invocation) em Java. O método **main** é o ponto de entrada do programa e é executado quando o programa é iniciado. Ele recebe argumentos da linha de comando, que são usados para configurar o servidor.

Dentro do bloco **try**, três ações principais são realizadas:

- É criada uma instância da classe **PartRepositoryImpl**, que implementa a interface **PartRepository**. Essa classe representa o repositório de partes do servidor e é responsável por armazenar e gerenciar as partes.
- É criado um registro RMI (**Registry**) na porta especificada pelo argumento **port**. O registro é usado para vincular objetos remotos aos seus nomes e permitir que os clientes possam localizá-los.
- O objeto **repository** é vinculado ao nome **serverName** no registro RMI, usando o método **rebind**. Isso significa que o objeto **repository** estará disponível para os clientes que procurarem por esse nome no registro.

- **Client:** Representa um cliente que se conecta a um repositório de partes remoto e executa operações como adicionar partes, recuperar partes

específicas e obter todas as partes do repositório. Essa classe instancia a interface de usuário **UserInterface** e apresenta todas as opções disponíveis, realizando a comunicação completa com o cliente e realizando as chamadas de métodos relacionados aos direcionamentos dados. Também apresenta mensagens de erro conforme a inconsistência dos dados inseridos. Possui três atributos principais:

- **currentRepository**: É uma instância da interface **PartRepository**, que representa um repositório de peças. É usado para armazenar o repositório atualmente conectado pelo cliente.
- **currentPart**: É uma instância da interface **Part**, que representa uma peça. É usado para armazenar a peça atualmente selecionada pelo cliente.
- **currentSubParts**: É um **HashMap** que mapeia objetos **Part** (peças) a inteiros (quantidade de subpeças). É usado para armazenar todas as subpeças que serão alocadas quando uma nova **Part** for adicionada pelo método **addp**.

2.2 Descrição das escolhas de estrutura de dados

A utilização dessas estruturas de dados no contexto do programa tem as seguintes vantagens/motivos:

- **HashMap**: O **HashMap** é usado para armazenar coleções de pares chave-valor. Nesse caso, é usado para mapear códigos de peças a objetos **Part** (no **PartRepositoryImpl**) e mapear objetos **Part** a quantidades (no **PartImpl**). Isso permite uma busca eficiente de peças com base em seus códigos, bem como a associação de quantidades às subpeças. Além disso, a estrutura **HashMap** permite uma implementação mais simples e direta das operações de adição, remoção e busca de elementos.
- **List**: A interface **List** é usada para representar coleções ordenadas de elementos. No contexto do programa, é usada para representar a lista de todas as peças em um repositório. A vantagem de usar uma **List** é que ela permite acessar, percorrer e manipular as peças em ordem específica, o que pode ser necessário em certas operações do programa.
- **String**: A classe **String** é usada para armazenar informações textuais, como códigos de peças, nomes e descrições. Strings são amplamente utilizadas em programas Java e fornecem métodos convenientes para manipulação de texto.
- **boolean**: O tipo primitivo **boolean** é usado para representar valores lógicos (verdadeiro ou falso) para indicar se uma peça é agregada ou primitiva. É uma escolha apropriada quando se trata de uma propriedade binária simples.

- **int:** O tipo primitivo `int` é usado para representar quantidades, como a quantidade de subpeças em uma peça. É apropriado para representar números inteiros não negativos.
- **UUID:** A classe `UUID` (Universally Unique Identifier) é usada para gerar códigos únicos para identificar peças. É útil garantir que cada peça tenha um código exclusivo e é amplamente adotada para essa finalidade.
- **Interface:** O uso de interfaces, como `Part` e `PartRepository`, permite a criação de abstrações e polimorfismo no programa. Isso facilita a manipulação e o gerenciamento de peças e repositórios, pois as interfaces fornecem um contrato que as classes concretas devem seguir. Essa abstração permite que diferentes implementações de peças e repositórios sejam usadas de forma intercambiável, tornando o código mais flexível e modular.

3 Execução do projeto

3.1 Para compilar o projeto

Dentro da pasta `distributed-parts`, executar o comando

```
$ javac -d bin src/*.java
```

3.2 Para executar o projeto

Dentro da pasta `distributed-parts`, execute os seguintes comandos:

```
$ cd bin; rmiregistry
```

Abra outro terminal. Troque `<servername>` pelo novo nome do servidor que será criado e `<port>` pelo número da porta que deseja conectar:

```
$ java -cp bin Server <server_name> <port>
```

Abra outro terminal:

```
$ java -cp bin Client
```

3.3 Para usufruir do sistema

Ao executar um servidor do tipo `Client`, o usuário recebe uma lista dos repositórios (do tipo `Server`) disponíveis, e deve digitar o nome daquele que deseja conectar-se, como no exemplo abaixo:

```
→ distributed-parts git:(master) X java -cp bin Client server1
Available Servers:
server1
server2
Choose one server
>> server1
Connected to repository: server1
>> □
```

Figura 6: Captura de tela do terminal.

No caso da captura de tela acima, o servidor escolhido foi o **server1**, e o usuário possui as seguintes possibilidades de comando:

- **bind** Faz o cliente se conectar a outro servidor e muda o repositório corrente. Este comando recebe o nome de um repositório e obtém do serviço de nomes uma referência para esse repositório, que passa a ser o repositório corrente.
- **listp** Lista as peças do repositório corrente.
- **lists** Exibe todos os repositórios existentes.
- **getp** Busca uma peça por código. A busca é efetuada no repositório corrente. Se encontrada, a peça passa a ser a nova peça corrente.
- **showp** Mostra atributos da peça corrente.
- **showsub** Mostra atributos da lista de subpeças corrente.
- **clearlist** Esvazia a lista de sub-peças corrente.
- **addsubpart** Adiciona à lista de sub-peças corrente N unidades da peça corrente.
- **addp** Adiciona uma peça ao repositório corrente. A lista de sub-peças corrente é usada como lista de subcomponentes diretos da nova peça. (É só para isto que existe a lista de sub-peças corrente).
- **showinfo** Mostra o nome e a quantidade de peças do repositório corrente.
- **quit** Encerra a execução do cliente.

Inicialmente, o usuário pode adicionar peças no repositório em que se encontra utilizando o comando **addp** e, para verificá-las, utilizar o comando **listp**:


```
Connected to repository: server1
>> addp
Enter the name of the new part:
>> peca1
Enter the description of the new part:
>> descricao1
New part added with code: 4cd67ee6-fadc-4fc4-a12e-8cccd26cd35e
>> addp
Enter the name of the new part:
>> peca2
Enter the description of the new part:
>> descricao2
New part added with code: feda276a-e2aa-4846-b157-1f329b14e580
>> listp

All Parts:

Part: peca1
Code: 4cd67ee6-fadc-4fc4-a12e-8cccd26cd35e
Agregada: false
Description: descricao1
Server: server1

Part: peca2
Code: feda276a-e2aa-4846-b157-1f329b14e580
Agregada: false
Description: descricao2
Server: server1
```

Figura 7: Captura de tela do terminal.

Ao estar conectado a um determinado servidor, o usuário pode trocar sua conexão para outro, utilizando o método `bind` e em seguida o repositório que se deseja acessar, e, dentro desse novo repositório, executar qualquer uma das ações disponíveis, como listar as peças do repositório corrente, como apresenta a captura abaixo:

```

>> bind server2
Connected to repository: server2
>> listp
No parts yet.
>> addp
Enter the name of the new part:
>> parte1server2
Enter the description of the new part:
>> descricao1
New part added with code: dd9b39e5-9d49-49d7-b8a0-9156faec66c2
>> addp
Enter the name of the new part:
>> parte2server2
Enter the description of the new part:
>> descricao2
New part added with code: cc6b9838-0a7b-47af-a548-35ab357ba08f
>> listp

All Parts:

Part: parte1server2
Code: dd9b39e5-9d49-49d7-b8a0-9156faec66c2
Agregada: false
Description: descricao1
Server: server2

Part: parte2server2
Code: cc6b9838-0a7b-47af-a548-35ab357ba08f
Agregada: false
Description: descricao2
Server: server2

```

Figura 8: Captura de tela do terminal.

A imagem demonstra o acesso ao servidor 2 e o cadastro de novas peças dentro deste, com acesso às suas informações através do comando `listp`. O usuário pode realizar o bind de forma mais assertiva utilizando o comando `lists`, que exibe todos os repositórios disponíveis para conexão, conforme figura abaixo:

```

>> lists
Available Servers:
server1
server2
>> bind server2
Connected to repository: server2
>> 

```

Figura 9: Captura de tela do terminal.

O usuário também pode buscar uma peça dentro do repositório que se encontra com o comando `getp`, enviando o código desta, que tem acesso ao executar o comando `listp`.

```
>> getp dd9b39e5-9d49-49d7-b8a0-9156faec66c2
Part retrieved: parte1server2
>> 
```

Figura 10: Captura de tela do terminal.

Dessa forma, recebe como retorno o nome da peça pesquisada, e torna a peça em questão uma peça corrente. O usuário também pode acessar informações do repositório em que se encontra através do comando **showinfo**.

```
Connected to repository: server3
>> showinfo
Current Repository: server3
Parts Quantity: 0
>> addp
Enter the name of the new part:
>> peca1server3
Enter the description of the new part:
>> descricao1
New part added with code: 714d473f-f7c4-4e7a-9146-26bd48fe1b31
>> addp
Enter the name of the new part:
>> peca2server3
Enter the description of the new part:
>> descricao2
New part added with code: 38fe8fbd-787c-47d9-a666-5a9718ec83dc
>> showinfo
Current Repository: server3
Parts Quantity: 2
```

Figura 11: Captura de tela do terminal.

Na captura acima, pode-se observar que as informações retornadas são o nome e a quantidade de peças contidas no repositório. Ao rodar o comando pela primeira vez, por estar vazio, a quantidade de peças retorna zero. Após adicionar uma peça, entretanto, retorna um.

Para exibir as informações sobre uma peça específica, basta digitar o comando **getp** com seu respectivo id e, em seguida, o comando **showp**.

```
>> listp

All Parts:

Part: peca1
Code: 4cd67ee6-fadc-4fc4-a12e-8cccd26cd35e
Agregada: false
Description: descricao1
Server: server1

Part: peca2
Code: fed276a-e2aa-4846-b157-1f329b14e580
Agregada: false
Description: descricao2
Server: server1

>> getp 4cd67ee6-fadc-4fc4-a12e-8cccd26cd35e
Part retrieved: peca1
>> showp
Current Part details:
Part: peca1
Code: 4cd67ee6-fadc-4fc4-a12e-8cccd26cd35e
Agregada: false
Description: descricao1
Server: server1
```

Figura 12: Captura de tela do terminal.

Os detalhes da parte apresentados são o nome da parte, seu código, se é agregada ou não (primitiva, caso o valor seja **false**, sua descrição e seu repositório. O atributo agregada se relaciona com a existência de subpartes, enquanto seu oposto, a não existência de subpartes, se relaciona com o atributo primitiva.

Para adicionar uma subpeça à uma peça, é necessário, a priori, ter uma peça corrente (**showp** para verificar), que será a subpeça adicionada. Depois, rodar o comando **addsubpart**, e inserir a quantidade de peças que deseja adicionar. Após isso, essa peça será adicionada a uma lista de subpeças correntes do **Client**, podendo ser adicionadas quantas subpeças forem necessárias a essa lista. Entretanto, para efetivamente adicionar subpeças a uma peça, depois de popular a lista de subpças, deve-se rodar o comando **addp**, que irá criar uma nova peça e adicionar todas as subpeças a ela de forma automática.

É importante ressaltar que, ao imprimir as informações das peças (com o comando **listp**), a lista de subpeças será exibida em conjunto com as outras informações, bem como o novo valor do atributo agregada, que agora aparece como verdadeiro, conforme a captura abaixo demonstra.

```

>> addsubpart
Insert how many subparts you would like to add
>> 5
Sub-part added.
>> addp
Enter the name of the new part:
>> peca3
Enter the description of the new part:
>> descricao3
New part added with code: 02e612aa-f129-46ac-8049-db0fd3f4df63
>> listp

All Parts:

Part: peca1
Code: 4cd67ee6-fadc-4fc4-a12e-8cccd26cd35e
Agregada: false
Description: descricao1
Server: server1

Part: peca2
Code: fed276a-e2aa-4846-b157-1f329b14e580
Agregada: false
Description: descricao2
Server: server1

```

Figura 13: Captura de tela do terminal.

```

Part: peca3
Code: 02e612aa-f129-46ac-8049-db0fd3f4df63
Agregada: true
Description: descricao3
Server: server1

SubParts de peca3:
Part: peca1
Code: 4cd67ee6-fadc-4fc4-a12e-8cccd26cd35e
Agregada: false
Description: descricao1
Server: server1

Quantity: 5

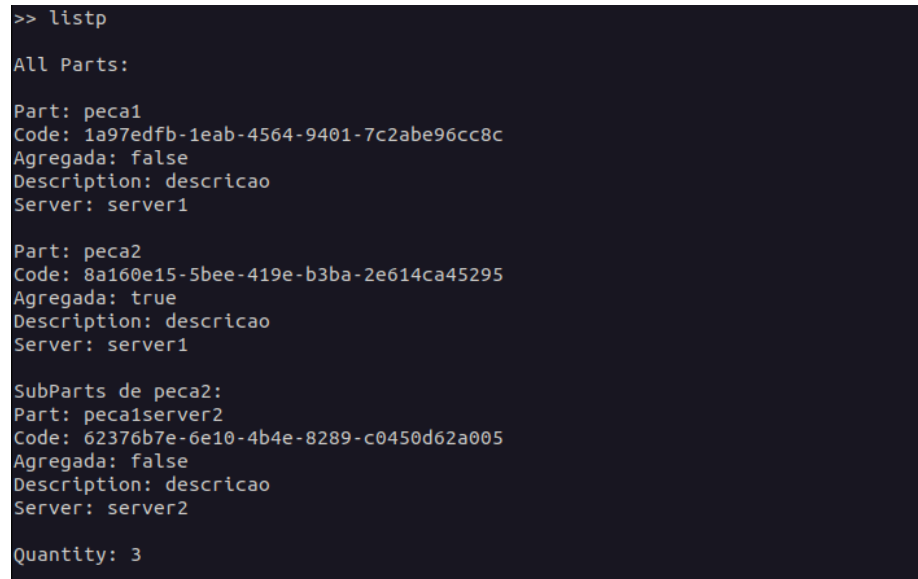
```

Figura 14: Captura de tela do terminal.

Nota-se que essa é a implementação principal do programa, uma vez que o usuário consegue navegar entre os diferentes **servers**, pegar as peças com o comando **getp** e adicioná-las como subpeças de outros **servers**, indo de encontro com o intuito de ter um sistema distribuído de **Parts**.

A captura de tela abaixo demonstra a inserção de uma subpeça a uma peça, ambas localizadas em repositórios diferentes. Ao final, ao rodar o comando **listp**, note que é mostrada a peça 2 (server1), com três subpeças do tipo peça 1 localizadas no server 2.

Para tal, basta: se conectar a um servidor e adicionar uma peça (que tornará peça corrente); se conectar a outro servidor e adicionar uma subpeça neste (que referencia `currentPart`, ou seja, a última peça criada); rodar o comando `showsub` para verificar se a subpeça foi adicionada corretamente; criar outra peça nesse segundo servidor e executar o comando `listp`, que apresentará as informações desta, inclusive de suas subpeças, podendo ser de diferentes servidores.



```
>> listp

All Parts:

Part: peca1
Code: 1a97edfb-1eab-4564-9401-7c2abe96cc8c
Agregada: false
Description: descricao
Server: server1

Part: peca2
Code: 8a160e15-5bee-419e-b3ba-2e614ca45295
Agregada: true
Description: descricao
Server: server1

SubParts de peca2:
Part: peca1server2
Code: 62376b7e-6e10-4b4e-8289-c0450d62a005
Agregada: false
Description: descricao
Server: server2

Quantity: 3
```

Figura 15: Captura de tela do terminal.

Há ainda o comando `showsub`, que mostra a lista de subpeças corrente, e o comando `clearlist`, que esvazia essa lista, caso o usuário não queira que a próxima peça adicionada tenha essa lista de subpeças, ou queira que a lista seja diferente.

```

>> showsub
Current SubParts:
Part: peca1server2
Code: 6745964a-2e98-4b6c-9dfd-644d936747db
Agregada: false
Description: descricao
Server: server2

Quantity: 3

Part: peca1
Code: b7d28045-f3a3-4e4c-90eb-e0db6bc858b9
Agregada: false
Description: descricao
Server: server1

Quantity: 9

>> clearlist
Sub-parts list cleared.
>> showsub
No subparts yet.
>> 

```

Figura 16: Captura de tela do terminal.

A captura abaixo, por sua vez, demonstra a criação de outra subpeça, que no caso é a p2, e define a quantidade como sete. Ao executar o comando `showsub`, tem-se uma lista composta por todas as subpeças adicionadas. Após a utilização do comando `clearlist`, essa lista é esvaziada, e ao tentar apresentá-la novamente com o comando `showsub`, o retorno é "No subparts available".

```

>> getp dc6dbe7c-3a58-4cc3-8f18-2beac565aae0
Part retrieved: p2
>> addsubpart
Insert how many subparts you would like to add
>> 7
Sub-part added.
>> showsub
Current SubParts:
Part: p2
Code: dc6dbe7c-3a58-4cc3-8f18-2beac565aae0
Description: desc2
Quantity: 7

Part: p1
Code: a7075400-fe7e-404d-a205-e935eff6767d
Description: desc1
Quantity: 4

>> clearlist
Sub-parts list cleared.
>> showsub
No subparts available.
>> 

```

Figura 17: Captura de tela do terminal.

O último comando disponível é `quit`, que encerra a conexão com o repositório, conforme demonstra a captura abaixo.



```
>> quit
Client terminated.
→ distributed-parts git:(master) X
```

Figura 18: Captura de tela do terminal.

3.4 Informações adicionais da máquina

```
$ lsb_release -a
```

```
No LSB modules are available. Distributor ID: Ubuntu
Description: Ubuntu 22.04.2 LTS
Release: 22.04
Codename: jammy
```

```
$ java -version
```

```
openjdk version "11.0.19" 2023-04-18 OpenJDK Runtime Environment
(build 11.0.19+7-post-Ubuntu-0ubuntu122.04.1) OpenJDK 64-Bit Server VM
(build 11.0.19+7-post-Ubuntu-0ubuntu122.04.1, mixed mode, sharing)
```

3.5 Código-fonte

O repositório com o código-fonte pode ser acessado na plataforma Github pelo link: <https://github.com/mirelameic/distributed-parts>.