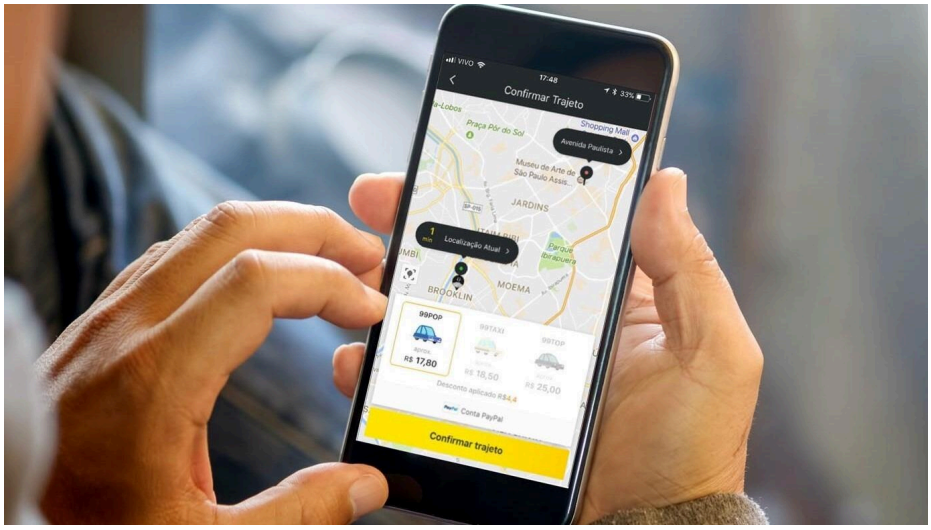


Sistema de Mobilidade Urbana (*Ride-Sharing*)

LUÍS GUSTAVO FERREIRA NUNES - 251012313

MATEUS ALVES ARAÚJO - 251013624

OSMAR JUNIO MENDES CARMO DOS SANTOS - 251037831

WANDA MARIA WRIGHT DOS SANTOS - 232006921

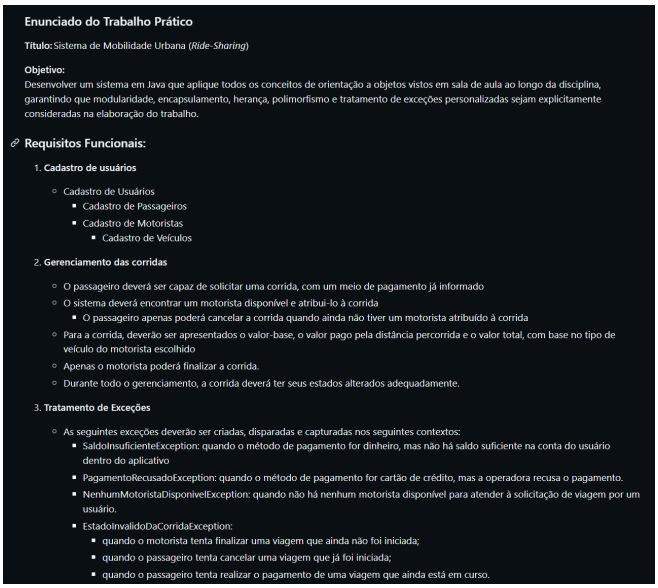
UNIVERSIDADE DE BRASÍLIA

ORIENTAÇÃO POR OBJETOS

— INTRODUÇÃO —

O objetivo desse projeto é construir um sistema com a lógica de um aplicativo de corridas, parecido com Uber, 99 e Cabify, usando todo conhecimento adquirido ao longo da disciplina. A função principal desse sistema é conectar passageiros que precisam de uma corrida à motoristas disponíveis.

Ao longo deste relatório iremos explicar como desenvolvemos passo-a-passo cada tópico do enunciado do trabalho principal, publicado pelo professor no GitHub.



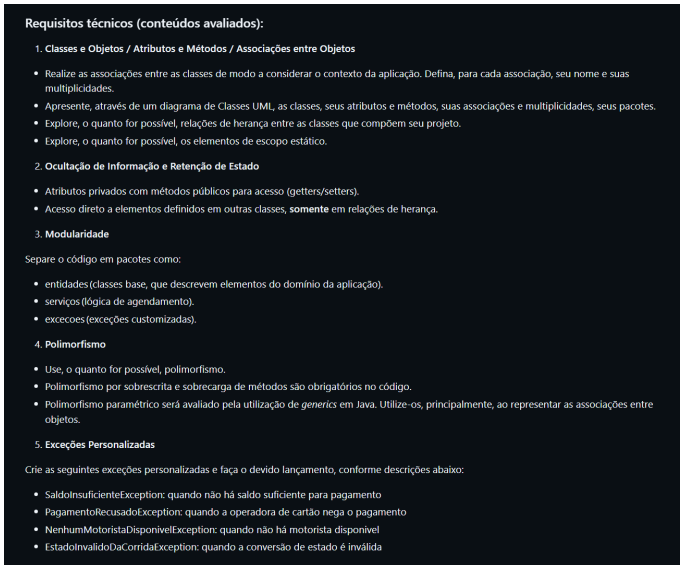
Enunciado do Trabalho Prático

Título: Sistema de Mobilidade Urbana (*Ride-Sharing*)

Objetivo:
Desenvolver um sistema em Java que aplique todos os conceitos de orientação a objetos vistos em sala de aula ao longo da disciplina, garantindo que modularidade, encapsulamento, herança, polimorfismo e tratamento de exceções personalizadas sejam explicitamente consideradas na elaboração do trabalho.

Requisitos Funcionais:

1. **Cadastro de usuários**
   - Cadastro de Usuários
     - Cadastro de Passageiros
     - Cadastro de Motoristas
       - Cadastro de Veículos
2. **Gerenciamento das corridas**
   - O passageiro deverá ser capaz de solicitar uma corrida, com um meio de pagamento já informado
   - O sistema deverá encontrar um motorista disponível e atribui-lo à corrida
     - O passageiro apenas poderá cancelar a corrida quando ainda não tiver um motorista atribuído à corrida
   - Para a corrida, deverão ser apresentados o valor-base, o valor pago pela distância percorrida e o valor total, com base no tipo de veículo do motorista escolhido
   - Apenas o motorista poderá finalizar a corrida.
   - Durante todo o gerenciamento, a corrida deverá ter seus estados alterados adequadamente.
3. **Tratamento de Exceções**
   - As seguintes exceções deverão ser criadas, disparadas e capturadas nos seguintes contextos:
     - SaldoInsuficienteException: quando o método de pagamento for dinheiro, mas não há saldo suficiente na conta do usuário dentro do aplicativo
     - PagamentoRecusadoException: quando o método de pagamento for cartão de crédito, mas a operadora recusa o pagamento.
     - NenhumMotoristaDisponivelException: quando não há nenhum motorista disponível para atender à solicitação de viagem por um usuário.
     - EstadoInvalidoDaCorridaException:
       - quando o motorista tenta finalizar uma viagem que ainda não foi iniciada;
       - quando o passageiro tenta cancelar uma viagem que já foi iniciada;
       - quando o passageiro tenta realizar o pagamento de uma viagem que ainda está em curso.



Requisitos técnicos (conteúdos avaliados):

1. **Classes e Objetos / Atributos e Métodos / Associações entre Objetos**

- Realize as associações entre as classes de modo a considerar o contexto da aplicação. Defina, para cada associação, seu nome e suas multiplicidades.
- Apresente, através de um diagrama de Classes UML, as classes, seus atributos e métodos, suas associações e multiplicidades, seus pacotes.
- Explore, o quanto for possível, relações de herança entre as classes que compõem seu projeto.
- Explore, o quanto for possível, os elementos de escopo estático.

2. **Ocultação de Informação e Retenção de Estado**

- Atributos privados com métodos públicos para acesso (getters/setters).
- Acesso direto a elementos definidos em outras classes, **somente** em relações de herança.

3. **Modularidade**

Separe o código em pacotes como:

- entidades (classes base, que descrevem elementos do domínio da aplicação).
- serviços (lógica de agendamento).
- excecoes (exceções customizadas).

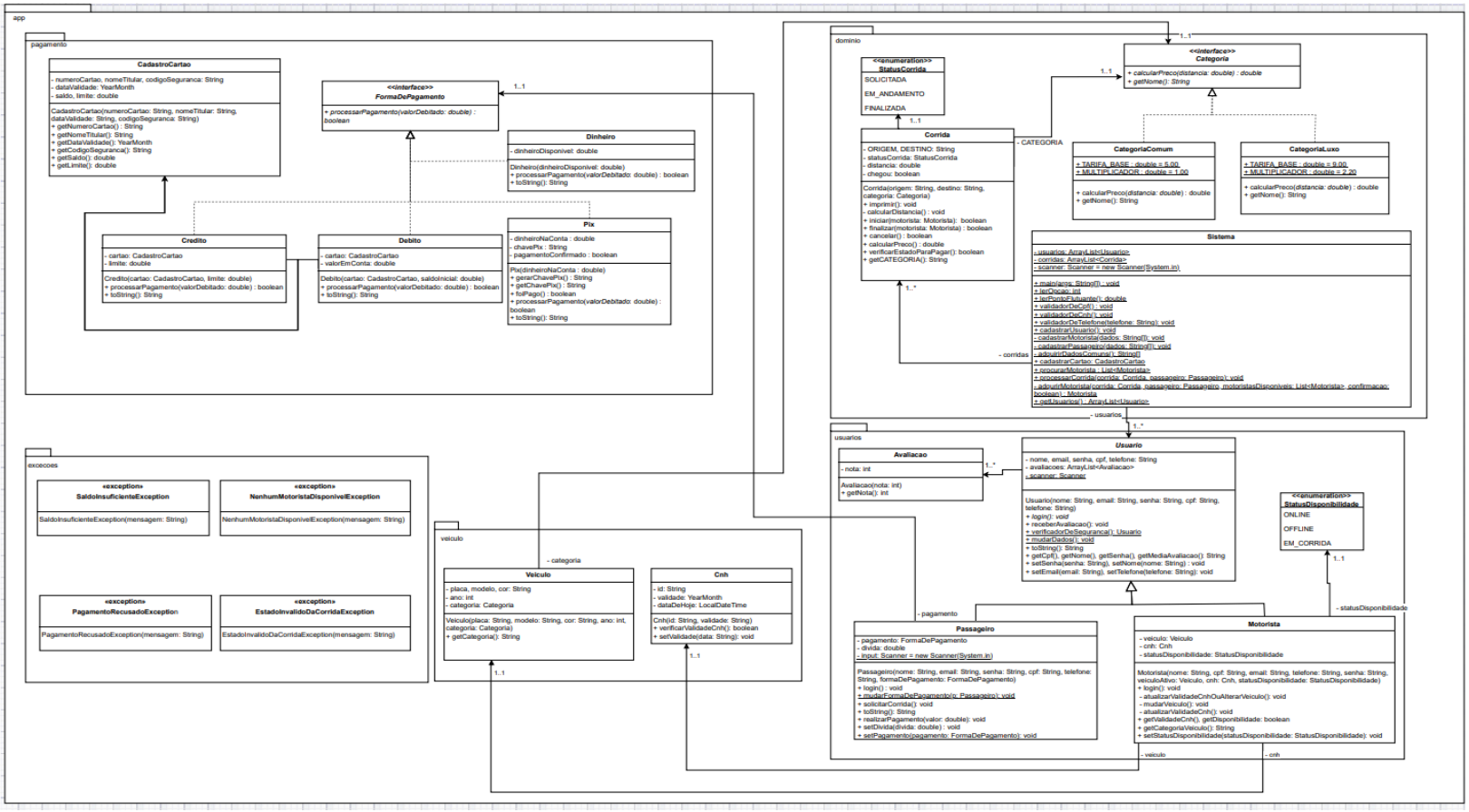
4. **Polimorfismo**

- Use, o quanto for possível, polimorfismo.
- Polimorfismo por sobrescrita e sobrecarga de métodos são obrigatórios no código.
- Polimorfismo paramétrico será avaliado pela utilização de *generics* em Java. Utilize-os, principalmente, ao representar as associações entre objetos.

5. **Exceções Personalizadas**

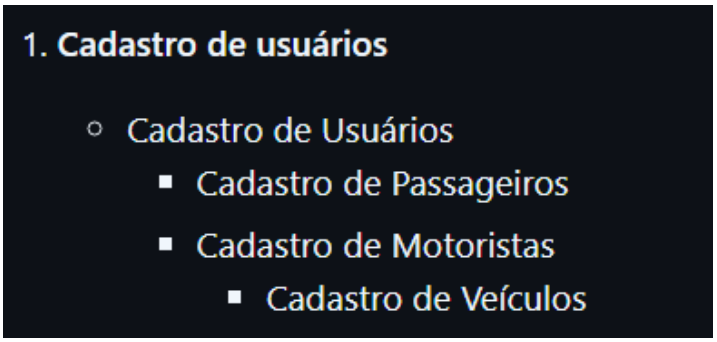
Crie as seguintes exceções personalizadas e faça o devido lançamento, conforme descrições abaixo:

- SaldoInsuficienteException: quando não há saldo suficiente para pagamento
- PagamentoRecusadoException: quando a operadora de cartão nega o pagamento
- NenhumMotoristaDisponivelException: quando não há motorista disponível
- EstadoInvalidoDaCorridaException: quando a conversão de estado é inválida



— UML DO PROJETO —

— CADASTRO DE USUÁRIOS —



Essa parte do projeto se destina a esclarecer sobre as principais funcionalidades das classes do pacote "usuários" e como elas se comportam umas com as outras e fora do pacote junto de outras classes.

Usuário: classe abstrata e classe mãe de Passageiro e Motorista, possui atributos comuns para ambos e métodos setters e getter para uso e/ou mudança de estado.

Passageiro: inicia a corrida, sendo o motor inicial para a aplicação. Se ocorrer o ciclo completo de uma corrida, o passageiro poderá avaliar o motorista ao fim do processo de pagamento.

Motorista: responsável pela disponibilidade da corrida, o motorista possui dados específicos como CNH, Status de Disponibilidade e Categoria de Veículo. Para iniciar o processo de viagem, o motorista deve possuir CNH dentro da data de validade bem como um Veículo e com o Status Online. Assim como passageiro, ao fim do ciclo completo de uma corrida, o motorista poderá avaliar seu passageiro.

StatusDisponibilidade: é encarregado de tratar da enumeração de "tags" que servem para definir o status atual do corrida; Similar a relação que a classe "StatusCorrida" tem com a classe "Corrida", por exemplo.

Avaliação: responsável por definir o método avaliativo que tanto passageiro quanto motorista possui.

```java
package app.usuarios;

public class Usuario {
    private String nome,email,senha,cpf,telefone;
    private Avaliacao avaliação;

    public Usuario(String nome, String email, String senha, String cpf, String telefone, Avaliacao avaliação) {
        this.nome = nome;
        this.email = email;
        this.senha = senha;
        this.cpf = cpf;
        this.telefone = telefone;
        this.avaliação=avaliação;

    };
    public void login(){
        System.out.println("Insira o cpf:");

    }
}
```

A classe Usuário define atributos tais como nome, cpf, email, telefone e senha que serão utilizados para o cadastro de um Usuário, seja ele motorista ou passageiro, devido a relação de herança que existe de Usuário → Passageiro e Usuário → Motorista.

Usuário possui apenas um construtor e métodos get e setters para seus atributos, os quais futuramente serão usados para a atualização de novos dados que o Usuário solicite.

```java
package app.usuarios;

public class Avaliacao {  3 usages  unknown +2
    private int nota;  2 usages

    public Avaliacao(int nota) { this.nota = nota; }
    public int getNota() {  1 usage  unknown
        return nota;
    }
```

A classe avaliação possui somente um atributo que é o de nota, o qual é do tipo inteiro. E seus métodos Construtor e getNota(); Dentro da classe Usuario, há o método "receberAvaliacao()," o qual é responsável pela lógica de negócio, onde o cliente poderá avaliar de uma nota que varia de 0 a 5, os dados coletados lá, futuramente, serão utilizados no cálculo de média avaliativa no método "getMediaAvaliacao()."

```java
package app.usuarios;

public enum StatusDisponibilidade {
    ONLINE, OFFLINE, EM_CORRIDA
}
```

Como supracitado acima, a função desta classe á de enumerar "tags" que serão utilizadas para definir o status atual do motorista. ex.: StatusDisponibilidade.ONLINE ao iniciar a corrida. (o construtor de motorista exige um objeto de "StatusDisponibilidade" além da própria classe Motorista possuir método set para"StatusDisponibilidade")

```java
public class Passageiro extends Usuario {
    private FormaDePagamento pagamento;
    private double divida;  6 usages
```

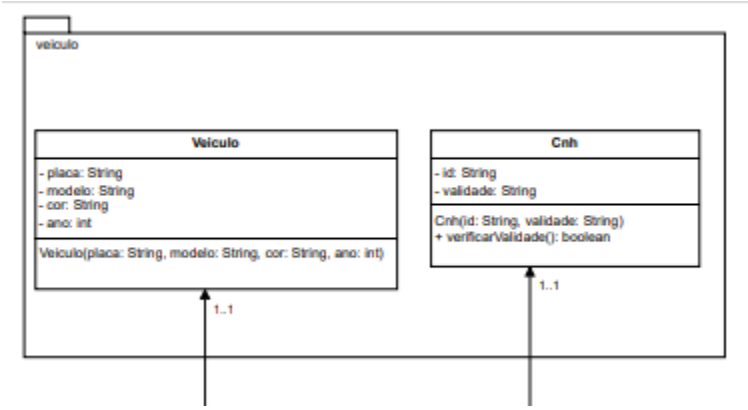
A classe Passageiro possui dois atributos próprios: pagamento do tipo "FormaDePagamento" - relação de atribuição - e dívida, responsável pelo armazenamento de fatura caso o pagamento de uma corrida fique como pendente. A classe possui um método de login() o qual determina o menu que será apresentado ao Passageiro, onde lá haverá opções de inicialização de corrida, mudança de forma de pagamento e dados pessoais, informações da conta e pagamento de dívida caso haja uma.

```java
public class Motorista extends Usuario {
    private Veiculo veiculo;  3 usages
    private Cnh cnh;  3 usages
    private StatusDisponibilidade statusD
```

A classe motorista possui atributos específicos dela como Veículo, CNH e StatusDisponibilidade. Além de contar com método de login() próprio, responsável por mostrar seu menu e também com a possibilidade de mudar seus dados pessoais, data de validade da CNH e veículo.

— VEÍCULO —

Todo motorista precisa obrigatoriamente cadastrar um veículo e uma CNH válida para poder começar suas atividades no aplicativo.



Nessa parte da UML, cadastramos duas classes referentes ao pacote veículo. A classe "veiculo" possui 4 atributos privados: placa, modelo, cor e ano. Privamos esses atributos pois não queremos que outras classes tenham acesso direto a eles. Os métodos implementados são para cadastro de veículos, e acesso aos atributos.

A classe Cnh possui dois atributos privados: id, que é o número individual gerado para toda CNH, a data de validade, e uma variável para absorver a data do dia do cadastro. O método cnh, para cadastro de CNH, e o método "verificarValidade", para saber se a cnh cadastrada está em dia, e se o número id é válido.

Essa parte do código é o cadastro da classe veículo, uma classe pública. O método "veiculo" é público pois é a forma com a qual o usuário vai ter acesso aos atributos para cadastrar as informações.

Logo após os métodos get são implementados, para que outras classes consigam ter acesso aos atributos dessa classe caso necessário.

```java
package app.veiculo;

public class Veiculo {
    private final String placa;
    private final String modelo;
    private final String cor;
    private final int ano;

    public Veiculo(String placa, String modelo, String cor, int ano) {

        this.placa = placa;
        this.modelo = modelo;
        this.cor = cor;
        this.ano = ano;
    }
    public String getPlaca() {
        return placa;
    }
    public String getModelo() {
        return modelo;
    }
    public String getCor() {
        return cor;
    }

    public int getAno() {
        return ano;
    }
```

```java
package app.veiculo;

import java.time.LocalDateTime;
import java.time.YearMonth;
import java.time.format.DateTimeFormatter;

public class Cnh {
    private final String id;
    private final YearMonth validade;
    private final LocalDateTime dataDeHoje = LocalDateTime.now();

    public Cnh(String id, String validade) {
        this.id = id;
        this.validade = YearMonth.parse(validade, DateTimeFormatter.ofPattern("MM/yyyy"));
    }
    public Boolean VerificarValidadeCnh () {
        if (this.id == null || this.id.trim().isEmpty()) {
            return false;
        }
        if (this.validade == null) {
            return false;
        }
        if ((dataDeHoje.getYear() > validade.getYear()) || (dataDeHoje.getYear() == validade.getYear()) && (dataDeHoje.getMonthValue() >= validade.getMonthValue())) {
            return false;
        }
        return this.id.length() == 9;

    }
```

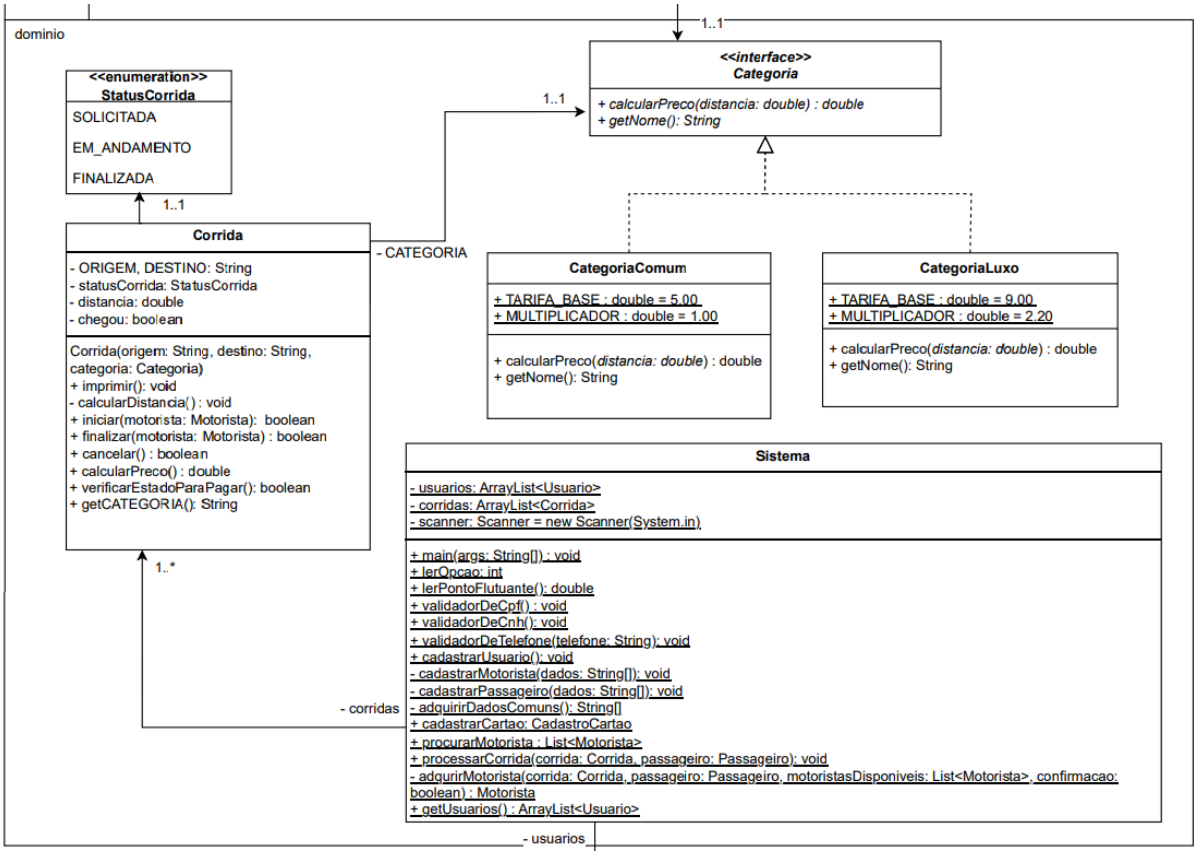
Essa parte do código é para cadastro da classe Cnh, uma classe pública. O método Cnh é público pois é com ele que o usuário irá fazer o cadastro da sua CNH. Possui um método que verifica se o número discado tanto no id, quanto na validade são válidos, e confere se não está vencida comparando a validade com a data do dia do cadastro.

— GERENCIAMENTO DAS CORRIDAS —

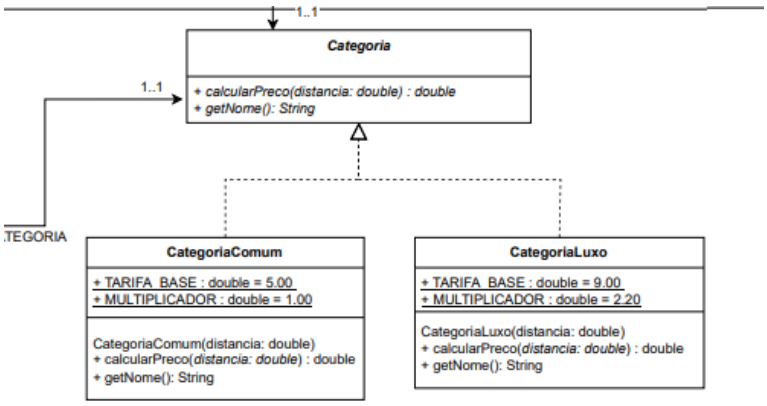
2. **Gerenciamento das corridas**

- O passageiro deverá ser capaz de solicitar uma corrida, com um meio de pagamento já informado
- O sistema deverá encontrar um motorista disponível e atribui-lo à corrida
  - O passageiro apenas poderá cancelar a corrida quando ainda não tiver um motorista atribuído à corrida
- Para a corrida, deverão ser apresentados o valor-base, o valor pago pela distância percorrida e o valor total, com base no tipo de veículo do motorista escolhido
- Apenas o motorista poderá finalizar a corrida.
- Durante todo o gerenciamento, a corrida deverá ter seus estados alterados adequadamente.

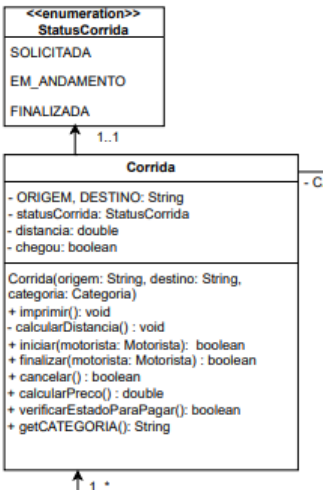
Esse trecho do enunciado explica bem diretamente o que essa parte deveria fazer. É nesse trecho em que o objetivo principal do aplicativo será aplicado.



No pacote dominio são cadastradas as classes ligadas ao funcionamento do aplicativo. Nesse pacote há a classe das corridas feitas no aplicativo,as classes das categorias das corridas e a classe "Sistema" que cadastra e acessa os Usuários e processa as corridas, além de realizar leitura de dados.



Essa parte do código é das categorias elas ditam qual o tipo de veículo é o da corrida e o preço que será a corrida. É usada uma interface chamada “Categoria” com dois métodos, um para calcular o preço com base no tipo de categoria e um método para que se saiba o tipo de categoria. Ambas as implementações de Categoria têm uma tarifa base e multiplicador constantes, eles são usados para calcular o preço da corrida somando a tarifa base à distância vezes o multiplicador, a distância é fornecida pela classe “Corrida”.



Essa é a classe Corrida, uma das classes centrais do programa, ela possui uma enumeração que representa seus 3 estados possíveis: Solicitada, Em andamento e Finalizada. Toda corrida possui uma origem , um destino e uma categoria fornecidos pelo Passageiro que a iniciou, a distância, um número entre 1 e 1000 quilômetros gerado pelo método “calcularDistância”, o atributo “chegou” verifica se o veículo alcançou o destino e sempre é iniciado como falso. O método “calcularPreco” chama o método de mesmo nome da interface “Categoria” para adquirir o preço. o método getCATEGORIA retorna o tipo de categoria como um texto(String).

```java
public Corrida(String origem, String destino, Categoria categoria ) {
    this.ORIGEM = origem;
    this.DESTINO = destino;
    this.CATEGORIA = categoria;
    calcularDistancia();
    statusCorrida = StatusCorrida.SOLICITADA;
}

public void imprimir() {
    System.out.println("------ Dados da Corrida ------");
    System.out.println("Origem: " + ORIGEM);
    System.out.println("Destino: " + DESTINO);
    System.out.println("Distância: " + distancia);
    System.out.println("Categoria: " + CATEGORIA.getNome());
    System.out.println("Preço: " + calcularPreco());
}
```

O método construtor de Corrida atribui a origem, destino e categoria da corrida ao seus respectivos atributos e gera uma distância e coloca a corrida como Solicitada, o método “imprimir” escreve na tela seus dados.

```java
public boolean cancelar() {
    try {
        if (StatusCorrida.SOLICITADA != statusCorrida) {
            throw new EstadoInvalidoDaCorridaException("Não é possível cancelar uma corrida em andamento ou finalizada.");
        }
    }catch (EstadoInvalidoDaCorridaException e) {
        System.out.println(e.getMessage());
        return false;
    }
    statusCorrida = StatusCorrida.FINALIZADA;
    System.out.println("Corrida Cancelada.");
    return true;
}
```

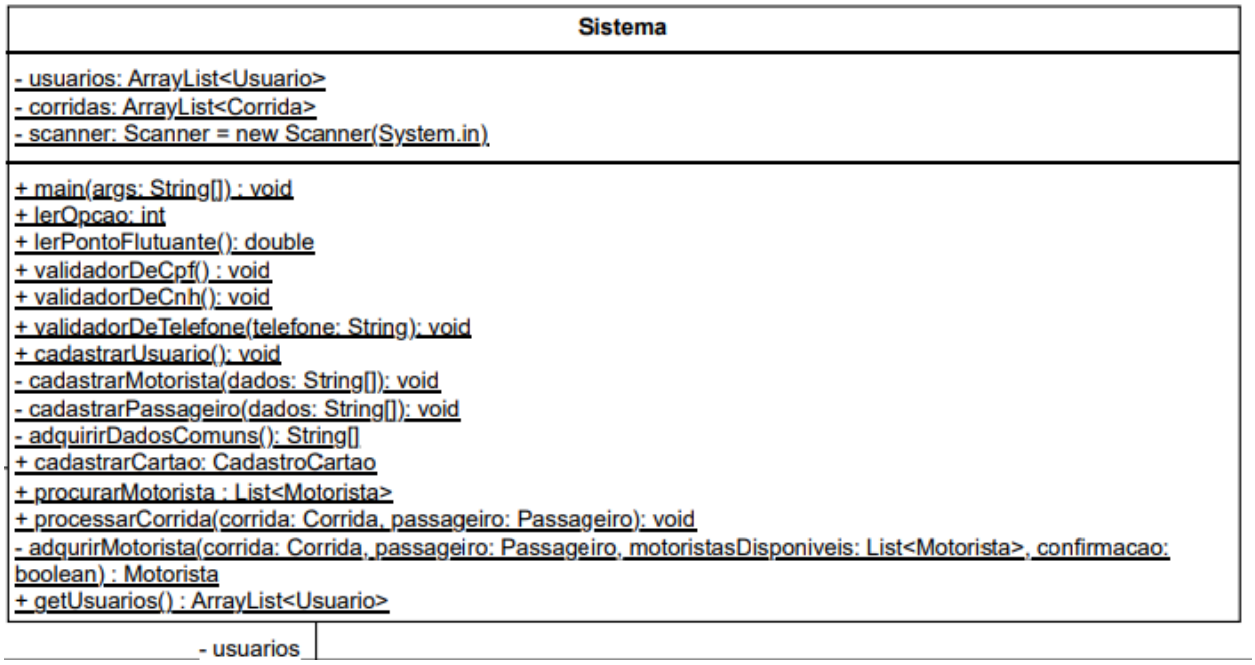
O método “cancelar” verifica se a corrida está no estado de Solicitada, se não estiver nesse estado é lançada uma exceção do tipo EstadoInvalidoDaCorridaException, é imprimida a mensagem “Não é possível cancelar uma corrida em andamento ou finalizada” e o método retorna falso, caso contrário a corrida é finalizada e o método retorna verdadeiro

```java
public void iniciar(Motorista motorista) {
    statusCorrida = StatusCorrida.EM_ANDAMENTO;
    motorista.setStatusDisponibilidade(StatusDisponibilidade.EM_CORRIDA);
    System.out.println("Corrida Iniciada.");
    new Thread(() -> {
        try {
            Thread.sleep(15000); // 15 segundos
            this.chegou = true;
            System.out.println("\nO veiculo chegou a:" + DESTINO);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("A corrida foi interrompida.");
        }
    }).start();
}

public boolean finalizar(Motorista motorista) {
    try {
        if (!chegou) {
            throw new EstadoInvalidoDaCorridaException("\nO veiculo não chegou em " + DESTINO);
        }
    } catch (EstadoInvalidoDaCorridaException e) {
        System.out.println(e.getMessage());
        return false;
    }
    statusCorrida = StatusCorrida.FINALIZADA;
    motorista.setStatusDisponibilidade(StatusDisponibilidade.ONLINE);
    System.out.println("Corrida Finalizada.");
    return true;
}
```

Quando o “método” iniciar é chamado a corrida muda seu estado para em andamento e o motorista do veículo tem seu estado mudado para em corrida. O programa espera 15 segundos para que o veículo chegue ao destino e o atributo “chegou” recebe verdadeiro. Se o Thread for interrompido é mostrado na tela que a corrida foi interrompida. Quando o motorista chama o método “finalizar” é verificado que se o veículo chegou ao destino, se não ainda não chegou é lançada uma exceção do tipo EstadoInvalidoDaCorridaException e o motorista é notificado que não pode finalizar uma corrida em andamento e o método retorna falso, caso contrário a corrida muda seu estado para finalizada e o motorista volta a estar disponível para outras corridas e o método retorna verdadeiro

```java
public boolean verificarEstadoParaPagar() {
    try {
        if (StatusCorrida.FINALIZADA != statusCorrida) {
            throw new EstadoInvalidoDaCorridaException("Espere a corrida terminar para pagar.");
        }

    }catch (EstadoInvalidoDaCorridaException e) {
        System.out.println(e.getMessage());
        return false;
    }
    return true;
}
```

Por fim o método “verificarEstadoParaPagar” verifica se o motorista finalizou para que o passageiro possa realizar seu pagamento, se isso não tiver ocorrido é lançada uma exceção do tipo EstadoInvalidoDaCorridaException, o passageiro é notificado que a corrida não foi terminada e o método retorna false, caso contrário o método retorna verdadeiro o que significa que o passageiro pode pagar a corrida.



A classe “Sistema” é a classe com o método main e por isso é a classe principal do programa. A classe possui uma lista com todos os usuários e outra com todas as corridas e uma scanner para leitura de dados o método “lerOpcao” é usado para ler valores inteiros com tratamento de exceção caso o valor não seja válido, o mesmo ocorre no método “lerPontoFlutuante”, mas para valores decimais. O método “getUsuarios” fornece a lista de todos os usuários.

```
public static void validadorDeCnh(String cnh) {
    if (!cnh.matches("[0-9]+")) {
        throw new InputMismatchException("CNH deve conter apenas números.");
    }
    if (cnh.length() != 9) {
        throw new InputMismatchException("CNH deve ter 9 dígitos.");
    }
}

public static void validadorDeTelefone(String telefone) {
    if (!telefone.matches("[0-9]+")) {
        throw new InputMismatchException("Telefone deve conter apenas números.");
    }
    if (telefone.length() < 12 || telefone.length() > 13) {
        throw new InputMismatchException("Telefone deve ter 12 ou 13 dígitos.");
    }
}
```

Os métodos “validadorDeCnh” e “validadorDeTelefone” funcionam de maneira similar, ambos verificam se os valores são apenas números, para a CNH é preciso ter 9 dígitos e para o telefone 12 ou 13 dígitos.

```java
public static void validadorDeCpf(String cpf) {
    for (Usuario usuario : usuarios) {
        if (usuario.getCpf().equals(cpf)) {
            throw new InputMismatchException("CPF já cadastrado.");
        }
    }
    cpf = cpf.replaceAll("[^0-9]", "");

    if (cpf.length() != 11) {
        throw new InputMismatchException("CPF deve ter 11 dígitos.");
    }

    if (cpf.matches("(\\d)\\1{10}")) {
        throw new InputMismatchException("CPF inválido.");
    }

    char dig10, dig11;
    int sm, i, r, num, peso;

    try {
        // Cálculo do 1º Dígito Verificador
        sm = 0;
        peso = 10;
        for (i = 0; i < 9; i++) {
            num = cpf.charAt(i) - 48;
            sm = sm + (num * peso);
            peso = peso - 1;
        }

        r = 11 - (sm % 11);
        if ((r == 10) || (r == 11)) {
            dig10 = '0';
        } else {
            dig10 = (char) (r + 48);
        }
        // Cálculo do 2º Dígito Verificador
        sm = 0;
        peso = 11;
        for (i = 0; i < 10; i++) {
            num = cpf.charAt(i) - 48;
            sm = sm + (num * peso);
            peso = peso - 1;
        }

        r = 11 - (sm % 11);
        if ((r == 10) || (r == 11)) {
            dig11 = '0';
        } else {
            dig11 = (char) (r + 48);
        }

        if (!((dig10 == cpf.charAt(9)) && (dig11 == cpf.charAt(10)))) {
            throw new InputMismatchException("CPF inválido.");
        }
    } catch (InputMismatchException erro) {
        throw new InputMismatchException("CPF inválido.");
    }
}
```

O Método “validadorDeCpf” verifica se o cpf pode ser cadastrado. Primeiro é verificado se aquele cpf não pertence a nenhum usuário cadastrado, se pertecer ele lança uma exceção. Se nenhuma exceção for lançada, são retirados todos os caracteres não numéricos do cpf e verifica se o cpf tem 11 dígitos, se ele não possuir essa quantidade é lançada uma exceção. Em seguida, é verificado se o cpf não é composto de caracteres todos iguais, pois valores como 000.000.000-00 passam na forma mas não constam em sites do governo, caso o cpf seja desse tipo é lançada uma exceção. Em seguida é feito o cálculo dos dígitos verificadores, após calcular os dígitos o programa verifica se os dígitos calculados batem com os dois últimos dígitos do cpf, caso sejam diferentes o método lança uma exceção. Se nenhuma exceção for lançada então é porque o cpf é válido.

```java
public static void main(String[] args) {
    int opcao;

    do {
        System.out.println("{----------------- Ride-Sharing    FGA0158 -----------------}");
        System.out.println("1. Cadastrar Usuário");
        System.out.println("2. Login");
        System.out.println("3. Sair");
        System.out.print("Escolha uma opção: ");
        opcao = lerOpcao();
        System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        System.out.println("==============================================================");
        switch (opcao) {
            case 1:
                cadastrarUsuario();
                break;
            case 2:
                Usuario.verificadorDeSeguranca().login();
                break;
            case 3:
                System.out.println("Saindo do sistema...");
                break;
            default:
                System.out.println("Opção inválida. Tente novamente.");
        }
    } while (opcao != 3);

    scanner.close();
}
```

O método main abre o menu principal do sistema com a possibilidade de escolher 3 opções: cadastrar usuário, fazer login em um usuário e sair do sistema e caso a opção escolhida não seja uma das fornecidas o programa notifica o usuário e pede outra opção.

```java
public static void cadastrarUsuario() {

    String[] dados;
    dados = adquirirDadosComuns();
    int tipoDeUsuario;
    System.out.println("1. Passageiro");
    System.out.println("2. Motorista");
    System.out.print(" Digite o tipo de usuário: ");
    tipoDeUsuario = lerOpcao();

    switch (tipoDeUsuario) {

        case 1:
            cadastrarPassageiro(dados);
            break;
        case 2:
            cadastrarMotorista(dados);
            break;
    }
    System.out.println("Usuário cadastrado com sucesso!");
    System.out.println("================================================================");
}
```

O método “cadastrarUsuario” como o nome diz é responsável por cadastrar os Usuários ele primeiro pede o CPF, telefone, email, nome e senha do usuário através do método “adquirirDadosComuns”, depois de adquiridos é perguntado o tipo de Usuário a ser cadastrado, Passageiro ou Motorista e então é chamado o método para cadastrar o tipo de usuário correspondente

```java
private static String[] adquirirDadosComuns() {
    String cpf;
    while (true) {
        try {
            System.out.print("Digite o CPF(Somente números): ");
            cpf = scanner.next();
            scanner.nextLine(); // Limpeza de buffer
            validadorDeCpf(cpf);
            break;
        } catch (InputMismatchException e) {
            System.out.println(e.getMessage() + " Tente novamente.");
        }
    }
    String nome;
    System.out.print("Digite o nome: ");
    nome = scanner.next();
    scanner.nextLine(); // Limpeza de buffer
    String email;
    System.out.print("Digite o email: ");
    email = scanner.next();
    scanner.nextLine(); // Limpeza de buffer
    String telefone;
    while (true) {
        try {
            System.out.print("Digite o telefone: ");
            telefone = scanner.next();
            validadorDeTelefone(telefone);
            break;
        } catch (InputMismatchException e) {
            System.out.println(e.getMessage() + " Tente novamente.");
        }
    }
    String senha, senhaDeConfirmacao;
    do {
        System.out.print("Digite a senha: ");
        senha = scanner.next();
        scanner.nextLine(); // Limpeza de buffer
        System.out.print("Confirme a senha: ");
        senhaDeConfirmacao = scanner.next();
        scanner.nextLine(); // Limpeza de buffer
        if (!senha.equals(senhaDeConfirmacao)) {
            System.out.println("As senhas não coincidem. Tente novamente.");
        }
    }while (!senha.equals(senhaDeConfirmacao));
    System.out.println("================================================================");
    return new String[]{nome, email, senha, cpf, telefone};
}
```

O método “adquirirDadosComuns” primeiro pede o CPF e verifica se ele é válido, caso não seja será pedido outro CPF até que seja digitado um CPF válido. Em seguida, são pedidos o nomes, o email, o telefone que é verificado sé é válido e por fim é preciso digitar a senha do Usuário e confirmar a senha, se as senhas forem iguais o método retorna um vetor de Strings.

```java
private static void cadastrarMotorista(String[] dados) {
    System.out.println("--------------------------------------------------------------------");
    String numeroCnh;
    while (true) {
        try {
            System.out.print("Digite o número da CNH: ");
            numeroCnh = scanner.next();
            validadorDeCnh(numeroCnh);
            break;
        } catch (InputMismatchException e) {
            System.out.println(e.getMessage() + " Tente novamente.");
        }
    }
    Cnh cnh = null;
    while (cnh == null) {
        try {
            System.out.print("Digite a data de validade da CNH (MM/yyyy): ");
            String validade = scanner.next();
            cnh = new Cnh(numeroCnh, validade);
        } catch (DateTimeParseException e) {
            System.out.println("Formato de data inválido. Use o formato MM/yyyy.");
        }
    }

    System.out.print("Digite a cor do veículo: ");
    String cor = scanner.next();
    System.out.print("Digite o modelo do veículo: ");
    String modelo = scanner.next();
    System.out.print("Digite o ano do veículo: ");
    int ano = lerOpcao();
    System.out.print("Digite a placa do veículo: ");
    String placa = scanner.next();

    Categoria categoria = null;
    int escolha;
    do {
        System.out.println("Digite a categoria do veículo (Comum ou Luxo): ");
        System.out.println("1. Comum");
        System.out.println("2. Luxo");
        System.out.print("Escolha uma opção: ");
        escolha = lerOpcao();
        switch (escolha) {
            case 1:
                categoria = new CategoriaComum();
                break;
            case 2:
                categoria = new CategoriaLuxo();
                break;
            default:
                System.out.println("Opção inválida.");
        }

    }while (escolha != 1 && escolha != 2);

    Veiculo veiculo = new Veiculo(placa, modelo, cor, ano, categoria);
```

```java
    System.out.println("--------------------------------------------------------------------");
    int disponivel;
    StatusDisponibilidade disponibilidade = null;
    do {
        System.out.println("O motorista pode começar a trabalhar agora?");
        System.out.println("1. Sim");
        System.out.println("2. Não");
        System.out.print("Escolha uma opção: ");
        disponivel = lerOpcao();

        switch (disponivel) {
            case 1:
                disponibilidade = StatusDisponibilidade.ONLINE;
                break;
            case 2:
                disponibilidade = StatusDisponibilidade.OFFLINE;
                break;
            default:
                System.out.println("Opção inválida.");
        }
    }while(disponivel != 1 && disponivel != 2);
    System.out.println("--------------------------------------------------------------------");
    usuarios.add(new Motorista(dados[0], dados[1], dados[2], dados[3], dados[4], veiculo, cnh, disponibilidade));
}
```

O método “CadastraMotorista” pode ser dividido em duas partes a primeira que cadastra a CNH e o veículo e a segunda que pergunta se ele já pode começar a trabalhar. Na primeira parte é digitado o ID da CNH que é verificado para saber se é válido, caso seja válido é pedido a data de validade da CNH no formato: MM/AAAA, caso o formato seja inválido o código notifica o usuário e pede a data no formato válido, se o formato for válido é criado um objeto do tipo Cnh. Em seguida, são pedidos a cor, ano, modelo e placa do veículo e após isso é digitado a qual categoria de corrida o veículo pertence, Luxo ou Comum e por fim é instanciado um objeto do tipo Veiculo. Na segunda parte é perguntado se o motorista pode trabalhar agora, se a resposta for sim então o motorista é instanciado e adicionada à lista de usuários com seus dados, veículo e cnh e com estado de disponibilidade online, senão o estado de disponibilidade é offline.

```java
private static void cadastrarPassageiro(String[] dados) {
    FormaDePagamento pagamento = null;
    int formaDePagamento;
    System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++");
    do {
        System.out.println("1. Credito");
        System.out.println("2. Dinheiro");
        System.out.println("3. Pix");
        System.out.println("4. Debito");
        System.out.print("Escolha a forma de pagamento que irá usar: ");
        formaDePagamento = lerOpcao();
        switch (formaDePagamento) {
            case 1:
                CadastroCartao cartaoCredito = cadastrarCartao();
                System.out.print("Digite o limite do Cartão: ");
                double limite = lerPontoFlutuante();
                pagamento = new Credito(cartaoCredito, limite);
                break;
            case 2:
                System.out.print("Digite o dinheiro disponível: ");
                double dinheiro = lerPontoFlutuante();
                pagamento = new Dinheiro(dinheiro);
                break;
            case 3:
                System.out.print("Digite o valor na conta: ");
                double valorNaConta = lerPontoFlutuante();
                pagamento = new Pix(valorNaConta);
                break;
            case 4:
                CadastroCartao cartaoDebito = cadastrarCartao();
                System.out.println("Digite o seu saldo: ");
                double saldo = lerPontoFlutuante();
                pagamento = new Debito(cartaoDebito, saldo);
                break;
            default:
                System.out.println("Opção de pagamento inválida.");
                break;
        }
    }while (formaDePagamento != 1 && formaDePagamento != 2 && formaDePagamento != 3 && formaDePagamento != 4);
    scanner.nextLine();
    System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++");
    usuarios.add(new Passageiro(dados[0], dados[1], dados[2], dados[3], dados[4], pagamento));
}
```

O método “cadastrarPassageiro” adquire a forma de pagamento que o passageiro irá usar, se o passageiro escolher qualquer tipo de cartão é usado o método “cadastrarCartao” para adquirir o número, validade, código de segurança e nome do titular do cartão,em seguida é pedido o saldo se for débito e limite se for crédito, se o passageiro escolher dinheiro ou pix é apenas pedido quanto dinheiro ou saldo ele tem. Por fim, um novo objeto do tipo Passageiro é adicionado à lista de usuários com os dados fornecidos anteriormente e com a forma de pagamento escolhida.

```java
private static List<Motorista> procurarMotoristas(String categoria) {
    List<Motorista> motoristasOnline;
    motoristasOnline = usuarios.stream()
            .filter(u -> u instanceof Motorista)
            .map(u -> (Motorista) u)
            .filter(m -> m.getDisponibilidade() && m.getValidadeCnh() && m.getCategoriaVeiculo().equals(categoria))
            .toList();
    if (motoristasOnline.isEmpty()) {
        throw new NenhumMotoristaDisponivelException("\nNenhum motorista disponível no momento.");
    }
    return motoristasOnline;
}
```

O método “procurarMotorista” varre toda a lista de usuários em busca de instância da classe Motorista, essas instâncias devem representar motoristas que estejam ONLINE, com CNH não vencida e que possuam um veículo da mesma categoria da corrida. Os elementos da lista de usuários que cumpram os requisitos são colocados em uma lista de motoristas, se a lista estiver vazia, ou seja, não há nenhum motorista disponível é lançada uma exceção do tipo NenhumMotoristaDisponivelException, caso contrário o método retorna a lista de motoristas.

```java
private static Motorista adquirirMotorista(Corrida corrida, Passageiro passageiro, List<Motorista> motoristasDisponiveis)
    int opcao;
    Motorista motorista = null;
    boolean confirmacao = false;
    for (Motorista i : motoristasDisponiveis) {
        System.out.println("============================================================");
        System.out.println("Encontramos o motorista: " + i.getNome());
        System.out.println(i.getMediaAvaliacao());
        System.out.println("============================================================");
        do {
            System.out.println("Aceita o motorista?");
            System.out.println("1. Sim");
            System.out.println("2. Não");
            System.out.println("3. Cancelar corrida");
            System.out.print("Escolha uma opção: ");
            opcao = lerOpcao();
            System.out.println("============================================================");
            switch (opcao) {
                case 1:
                    System.out.println("O Cliente " + passageiro.getNome() + " está solicitando uma corrida!");
                    System.out.println(passageiro.getMediaAvaliacao());
                    System.out.println("============================================================");
                    corrida.imprimir();
                    int opcao2;
                    do {
                        System.out.println("Aceita o passageiro?");
                        System.out.println("1. Sim");
                        System.out.println("2. Não");
                        System.out.print("Escolha uma opção: ");
                        opcao2 = lerOpcao();
                        switch (opcao2) {
                            case 1:
                                confirmacao = true;
                                motorista = i;
                                i.setStatusDisponibilidade(StatusDisponibilidade.EM_CORRIDA);
                                break;
                            case 2:
                                System.out.println("O motorista recusou a corrida.");
                                break;
                            default:
                                System.out.println("Opção inválida.");
                        }

                    } while (opcao2 != 1 && opcao2 != 2);
                    break;
                case 2:
                    break;
                case 3:
                    System.out.println("Corrida cancelada.");
                    corrida.cancelar();
                    return null;
                default:
                    System.out.println("Opção inválida.");
            }
        }while (opcao != 1 && opcao != 2);
        System.out.println("============================================================");
        if (confirmacao) {
            break;
        }
    }
    try {
        if (motorista == null) {
            throw new NenhumMotoristaDisponivelException("Não há mais nenhum motorista disponível.");
        }
    } catch (NenhumMotoristaDisponivelException e) {
        System.out.println(e.getMessage());
        return null;
    }
    System.out.println("____________________________________________________________");
    return motorista;
}
```

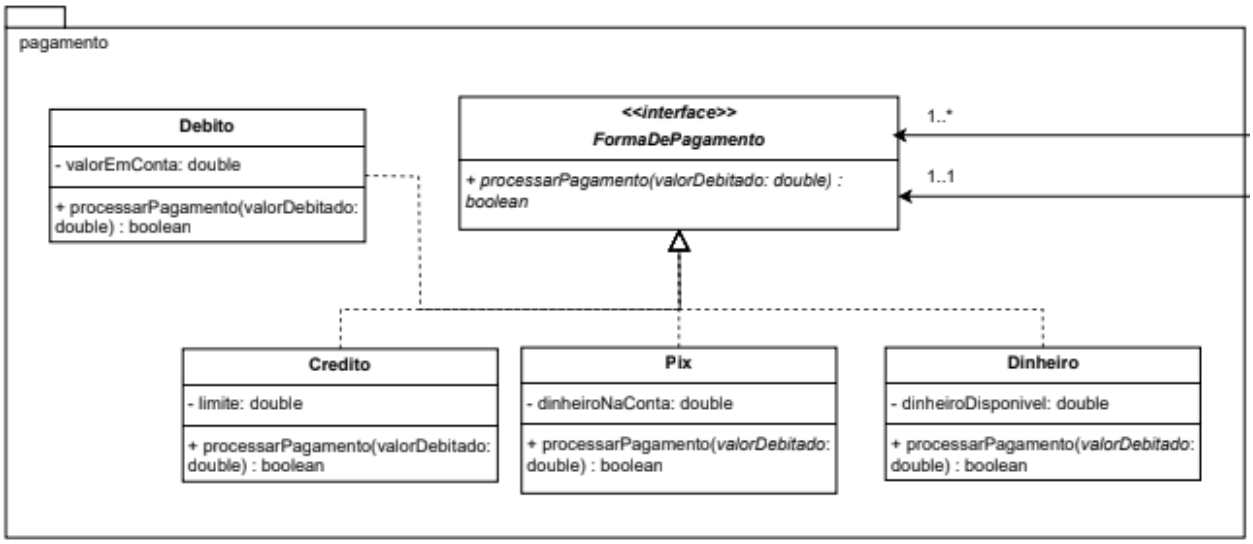
O método “adquirirMotorista” é responsável por selecionar um motorista dentre a lista de motoristas disponíveis. O programa notifica o passageiro que achou determinado motorista e mostra sua avaliação o passageiro tem então a opção de aceitar tal motorista ou cancelar a corrida, se cancelar a corrida a corrida é cancelada e o método retorna null, se escolher não, é escolhido outro motorista da lista, se escolher sim o programa pergunta ao motorista se ele aceita o passageiro que solicitou a corrida, mostrando a nota do passageiro, se o motorista aceitar o passageiro aquele motorista passa a ter o estado de em corrida e o método retorna aquele motorista, caso contrário o programa avisa o passageiro que o motorista recusou a corrida e fornece outro motorista. Caso não haja mais motoristas disponíveis, ou seja, a lista de motoristas foi completamente percorrida, o método lança uma exceção do tipo NenhumMotoristaDisponivelException e retorna null;

```java
public static void processarCorrida(Corrida corrida, Passageiro passageiro) {
    corridas.add(corrida);
    corrida.imprimir();
    int opcao;
    List<Motorista> motoristasDisponiveis;
    try {
        motoristasDisponiveis = procurarMotoristas(corrida.getCATEGORIA());
    } catch (NenhumMotoristaDisponivelException e) {
        System.out.println(e.getMessage());
        return;
    }
    Motorista motorista = adquirirMotorista(corrida,passageiro, motoristasDisponiveis);
    if (motorista == null) return;
    System.out.println("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++");
    corrida.iniciar(motorista);
   while (true) {
       System.out.println("1. Cancelar");
       System.out.println("2. Finalizar");
       System.out.println("3. Realizar Pagamento");
       System.out.print("Escolha uma opção: ");
       opcao = lerOpcao();
       switch (opcao) {
           case 1:
               if (corrida.cancelar()) {
                   return;
               }
               break;
           case 2:
               if (corrida.finalizar(motorista)) {
                   motorista.receberAvaliacao();
                   passageiro.receberAvaliacao();
               }
               break;
           case 3:
               if (corrida.verificarEstadoParaPagar()) {
                   passageiro.realizarPagamento(corrida.calcularPreco());
                   return;
               }
               break;
           default:
               System.out.println("Opção inválida.");
       }
   }
}
```

O método “processarCorrida” começa adicionando a corrida à lista de corridas do sistema e escreve os dados da corrida na tela, depois são pegos os motoristas disponíveis através do método “procurarMotoristas” com tratamento da exceção lançada por esse método se essa exceção for lançada o método é interrompido. Depois é escolhido um motorista da lista e se não foi possível escolher um motorista o método é interrompido. Após adquirir um motorista, a corrida é iniciada com as opções de cancelar, finalizar e realizar o pagamento da corrida cada opção chama o respectivo método da classe Corrida, se o Estado da Corrida for válido, quando se escolhe cancelar o método é interrompido, quando se finaliza é iniciada a avaliação dos participantes da corrida e quando se escolhe pagar é chamado o método do passageiro para realizar pagamento com o preço da corrida e o método finaliza.

— PAGAMENTO —

O passageiro deve cadastrar ao menos um método de pagamento. O sistema deve suportar diferentes formas de pagamento, como cartão de crédito, PIX ou dinheiro. Cada método de pagamento terá uma lógica distinta para processar a cobrança no final da corrida.



O pacote “pagamento” possui uma classe principal e outras 4 subclasses que herdam o método “processarPagamento”, que utiliza o polimorfismo por sobrescrita. Cada uma das subclasses são diferentes formas de pagamentos.

Implementamos uma classe chamada “CadastroCartao” para cadastrar as informações do cartão, e após o cadastro o cliente escolhe se usará como cartão de crédito ou cartão de débito. Essa classe cadastra número, código de segurança, validade e nome do titular do cartão. Ela também tem acesso direto ao limite ou saldo do cartão cadastrado.

```java
package app.pagamento;

public class CadastroCartao {
    private String numeroCartao;
    private String nomeTitular;
    private String dataValidade;
    private String codigoSeguranca;
    private double saldo;
    private double limite;

    public CadastroCartao(String numeroCartao, String nomeTitular, String dataValidade, String codigoSeguranca) {
        this.numeroCartao = numeroCartao;
        this.nomeTitular = nomeTitular;
        this.dataValidade = dataValidade;
        this.codigoSeguranca = codigoSeguranca;
        this.saldo = saldo;
        this.limite = limite;

    }

    public String getNumeroCartao() {
        return numeroCartao;
    }

    public String getNomeTitular() {
        return nomeTitular;
    }

    public String getDataValidade() {
        return dataValidade;
    }

    public String getCodigoSeguranca() {
        return codigoSeguranca;
    }

    public double getSaldo() {
        return saldo;
    }
    public double getLimite() {
        return limite;
    }
```

```java
package app.pagamento;

public class Credito implements FormaDePagamento {
    private CadastroCartao Cartao;
    private double Limite;

    public Credito(CadastroCartao cartao) {
        this.Cartao = cartao;
        this.Limite = cartao.getLimite();
    }

    @Override
    public boolean processarPagamento(double valorDebitado) {
        if (valorDebitado <= Limite) {
            Limite -= valorDebitado;
            System.out.println("Pagamento realizado com sucesso");
            return true;
        }
        else {
            System.out.println("Pagamento recusado: saldo insuficiente.");
            return false;
        }
    }
}
```

A forma de pagamento por cartão de crédito foi implementada nessa classe “Credito”, que absorve um atributo do tipo “CadastroCartao”, onde o atributo limite adquire o valor do limite do cartão cadastrado. Logo em seguida ele aplica o método herdado pela classe “FormaDePagamento”, onde ele realiza a operação de cobrança. Caso o valor da corrida seja menor ou igual ao limite de crédito do cartão, logo a operação é realizada com sucesso. Caso contrário, falha.

```java
package app.pagamento;

public class Debito implements FormaDePagamento {
    private CadastroCartao Cartao;
    private double ValorEmConta;

    public Debito(CadastroCartao cartao, double saldoInicial) {
        this.Cartao = cartao;
        this.ValorEmConta = cartao.getSaldo();
    }

    @Override
    public boolean processarPagamento(double valorDebitado) {
        if (valorDebitado <= ValorEmConta) {
            ValorEmConta -= valorDebitado;
            System.out.println("Pagamento realizado com sucesso!");
            return true;
        }
        else {
            System.out.println("Pagamento recusado: saldo insuficiente.");
            return false;
        }
    }
}
```

A forma de pagamento por cartão de Débito foi implementada nessa classe “Débito”, que possui exatamente a mesma lógica que o crédito, a única coisa que vai mudar é a forma que vai cobrar. Ao invés de cobrar do limite do cartão, ele cobrará do valor disponível na conta do cartão.

```java
package app.pagamento;

public class Pix implements FormaDePagamento {
    private double DinheironaConta;
    private String chavePix;
    private boolean pagamentoConfirmado = false;

    public Pix(double dinheironaConta) {
        DinheironaConta = dinheironaConta;
        gerarChavePix();
    }

    // Gera uma chave PIX aleatória
    public String gerarChavePix() {
        if (this.chavePix == null || this.chavePix.isEmpty()) {
            this.chavePix = java.util.UUID.randomUUID().toString().replace("-", "");
        }
        return this.chavePix;
    }

    public String getChavePix() {
        return this.chavePix;
    }

    // Informa se o último pagamento via PIX foi confirmado
    public boolean foiPago() {
        return this.pagamentoConfirmado;
    }

    @Override
    public boolean processarPagamento(double valorDebitado) {

        if (valorDebitado <= DinheironaConta) {
            DinheironaConta -= valorDebitado;
            pagamentoConfirmado = true;
            System.out.println("Pagamento realizado com sucesso");
            return true;
        }
        pagamentoConfirmado = false;
        System.out.println("Pagamento recusado: saldo insuficiente.");
        return false;
    }
}
```

A forma de pagamento por PIX é um pouco mais trabalhosa, porém muito útil. Essa classe possui um método para gerar uma chave pix aleatória para o usuário, onde o próprio passageiro deve depositar o valor pelo seu aplicativo do banco, usando a chave. Após gerar a chave, ele faz a conferência do pagamento, caso o dinheiro tenha sido debitado ele vai retornar sucesso. Caso contrário, o usuário terá de tentar novamente, ou outra forma de pagamento.

Já o pagamento por dinheiro físico é mais simples, pois o pagamento é administrado diretamente entre o motorista e o passageiro. Essa classe apenas entrega o valor pro usuário, e ao final da corrida o motorista informa se o pagamento foi concluído ou não.

```java
package app.pagamento;

public class Dinheiro implements FormaDePagamento {
    private double DinheiroDisponivel;

    public Dinheiro(double dinheiroDisponivel) {
        DinheiroDisponivel = dinheiroDisponivel;
    }

    @Override
    public boolean processarPagamento(double valorDebitado) {
        if (valorDebitado <= DinheiroDisponivel) {
            System.out.println("Pagamento concluído");
            return true;
        }
        else {
            System.out.println("Pagamento da corrida pendente!");
            return false;
        }
}
}
```

— EXCEÇÕES —

O código possui 4 exceções personalizadas, todas podem ocorrer apenas em tempo de execução, elas são:

- **EstadoInvalidoDaCorridaException - Ocorre quando se tenta realizar uma ação da corrida em um estado incorreto dela, como realizar pagamento de uma corrida que não terminou;**
- **NenhumMotoristaDisponivelException - Ocorre quando não há nenhum motorista válido disponível e que aceitou a corrida do cliente ou que o cliente tenha aceitado;**
- **PagamentoRecusadoException - Ocorre quando um cartão não aceita realizar um pagamento seja por algum erro no processamento ou por falta de saldo ou limite;**
- **SaldoInsuficienteException - Ocorre quando um passageiro tenta realizar um pagamento em dinheiro ou pix, mas não possui dinheiro ou saldo suficiente;**

— CONCLUSÃO —

A ideia pela qual gira em torno desse trabalho é essencialmente importante para mostrar a aplicação prática que uma Programação Orientada a Objetos tem a nos oferecer. De modo que com a extração de seus conceitos - associação, herança e polimorfismo - somos capazes de desenvolver nossa própria aplicação e a também amadurecer nossa visão profissional como programadores. Aprender POO nos faz perceber certas nuances das quais jamais podemos nos desprender, o nível de atenção assim como o nível de praticidade que uma linguagem Orientada a Objetos propõe é extremamente importante para projetos que como esse exigem um certo nível de complexidade até mesmo nos mais simples dos detalhes. Por fim, conclui-se que com esse projeto formos capazes de aprender e a principalmente extrair todos os nossos conhecimentos extraídos ao longo desse semestre em POO, visto que o nosso sistema atende aos requisitos de realização de corridas e cadastro de usuários.