FÁBIO LUIZ RIBEIRO GUIMARÃES
2203474

ATIVIDADE EXERCÍCIO SOLID

Atividade da Disciplina de **Microservices & Serverless Architecture** Prof. Roan Monteiro

São Paulo
2023

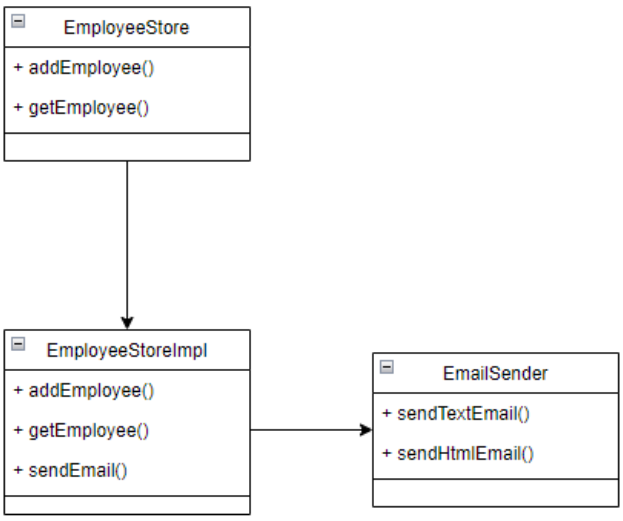
Sumário

Sumário

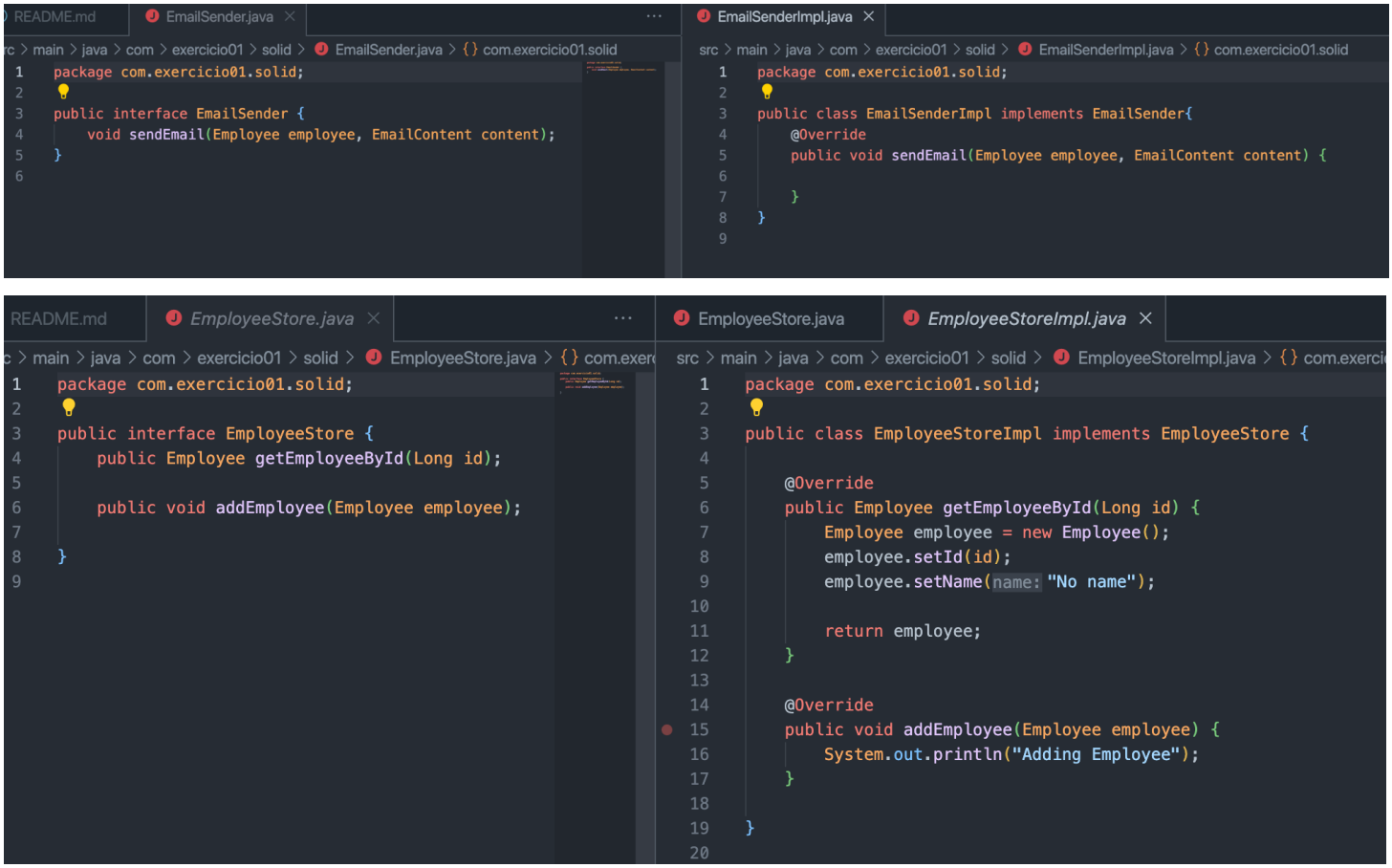
Exercício 01

Para fixar o conceito de SRP (Princípio da responsabilidade única), vamos imaginar que temos uma interface EmployeeStore.java e sua implementação EmployeeStoreImpl.java. Existem 3 métodos, buscar e adicionar employees e enviar email para eles. Podemos reparar que esse design não está legal, e suponhamos que o conteúdo pode ser dois tipos de conteúdo, HTML e texto. Se você reparar, o parâmetro de conteúdo no método somente suporta texto. Nesse caso, o que faremos? Criaremos um método sendHtmlEmail()? E cada vez que tivermos conteúdo diferente vamos adicionando um método? Não seria prudente, certo? Baseado no que foi aprendido em sala de aula, faça um Design (UML) e implementação Java que contemple o Princípio da Responsabilidade Única.

UML



IMPLEMENTAÇÃO



```java
package com.exercicio01.solid;

public interface EmailSender {
    void sendEmail(Employee employee, EmailContent content);
}
```

```java
package com.exercicio01.solid;

public class EmailSenderImpl implements EmailSender{
    @Override
    public void sendEmail(Employee employee, EmailContent content) {

    }
}
```

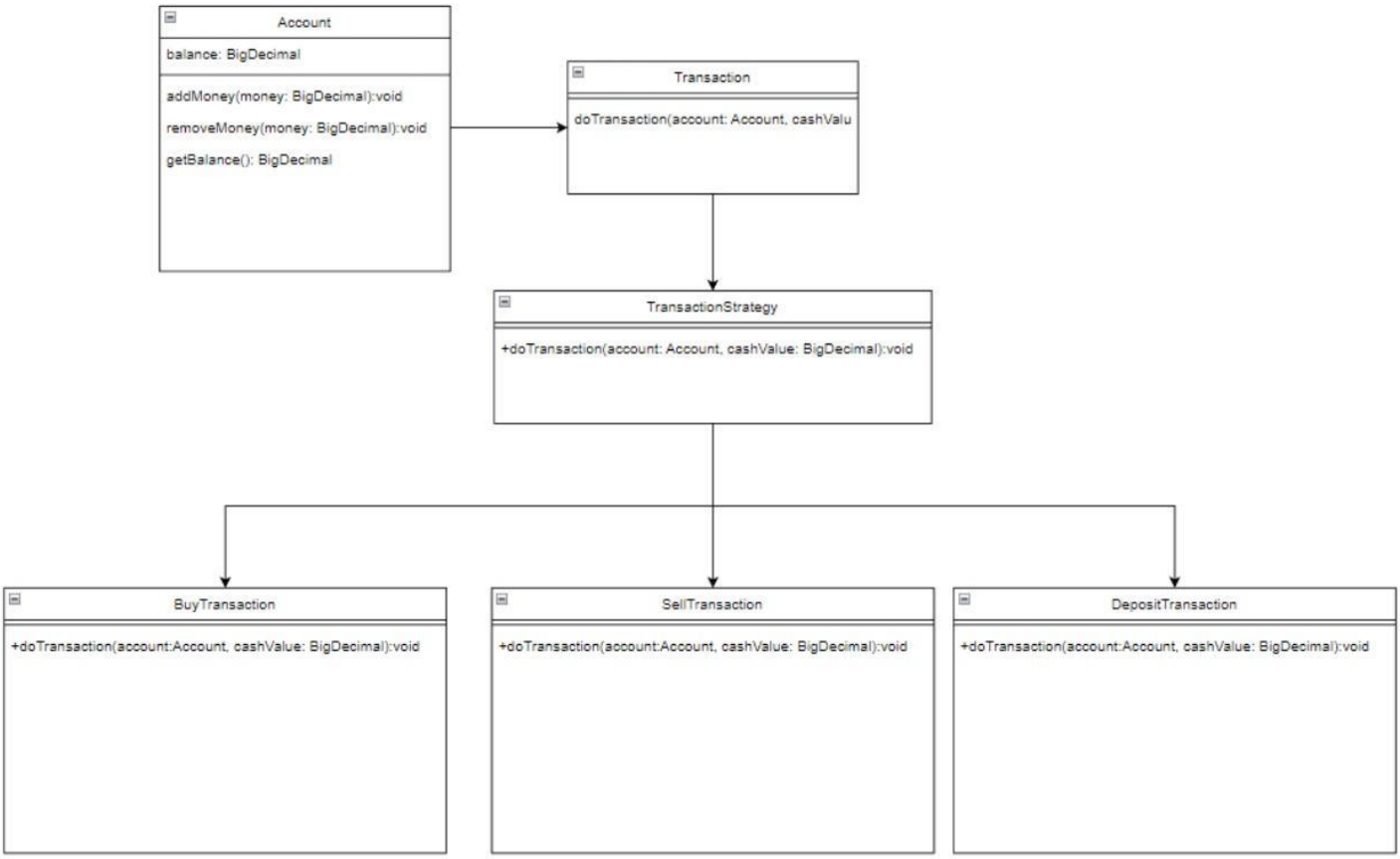
```java
package com.exercicio01.solid;

public interface EmployeeStore {
    public Employee getEmployeeById(Long id);

    public void addEmployee(Employee employee);

}
```

```java
package com.exercicio01.solid;

public class EmployeeStoreImpl implements EmployeeStore {

    @Override
    public Employee getEmployeeById(Long id) {
        Employee employee = new Employee();
        employee.setId(id);
        employee.setName(name: "No name");

        return employee;
    }

    @Override
    public void addEmployee(Employee employee) {
        System.out.println("Adding Employee");
    }

}
```

Exercício 02

Imagina que trabalhamos no banco e precisamos implementar as operações de transações que são compra (buy), venda(sell), depósito (deposit) e saque (withdrawal). A compra e o saque são movimentos que debitam o valor da sua conta, já a venda e o depósito são os que creditam. O design da classe foi feito sem respeitar muito os padrões de orientação a objeto e o OCP. Nós temos a classe Account.java com as operações e que está bem definida, esta não precisa ser alterada. Já a classe Transaction.java esta com problema de design e não se aplica ao OCP, logo precisaremos refatorar o código e aplicar uma melhor solução para a solução abaixo. Desenvolva um melhor design (UML) e implementação Java para a solução baseada no que aprendemos na sala de aula para OCP.

UML



A interface TransactionStrategy define o método doTransaction() que será implementado pelas classes que representam as diferentes operações de transação. A classe BuyTransaction implementa a interface TransactionStrategy e define a lógica para realizar uma compra, aplicando a taxa de 15% sobre o valor e removendo o valor da conta do cliente. A classe SellTransaction também implementa a interface TransactionStrategy e define a lógica para realizar uma venda, aplicando a taxa de 10% sobre o valor e adicionando o valor à conta do cliente. A classe DepositTransaction também implementa a interface TransactionStrategy e define a lógica para realizar um depósito, aplicando a taxa de 5% sobre o valor e adicionando o valor à conta do cliente. A classe WithdrawalTransaction também implementa a interface TransactionStrategy e define a lógica para realizar um saque, aplicando a taxa de 20% sobre o valor e removendo o valor da conta do cliente.

IMPLEMENTAÇÃO

README.md | Account.java

src > main > java > com > exercicio02 > solid > Account.java > {} com.exercicio02.solid

```java
package com.exercicio02.solid;

import java.math.BigDecimal;

public class Account {
    private BigDecimal balance = new BigDecimal("1000");

    public void addMoney(BigDecimal money) {
        this.balance = this.balance.add(money);
    }

    public void removeMoney(BigDecimal money) {
        this.balance = this.balance.subtract(money);
    }

    public BigDecimal getBalance() {
        return balance;
    }

    public void doTransaction(TransactionStrategy strategy, BigDecimal cashValue) {
        strategy.doTransaction(this, cashValue);
    }
}
```

EmailSenderImpl.java | TransactionStrategy.java

src > main > java > com > exercicio02 > solid > TransactionStrategy.java > {} com.exercicio

```java
package com.exercicio02.solid;

import java.math.BigDecimal;

public interface TransactionStrategy {
    void doTransaction(Account account, BigDecimal cashValue);
}
```

README.md | BuyTransaction.java

src > main > java > com > exercicio02 > solid > BuyTransaction.java > {} com.exercicio02.solid

```java
package com.exercicio02.solid;

import java.math.BigDecimal;
import java.math.RoundingMode;

public class BuyTransaction implements TransactionStrategy {

    @Override
    public void doTransaction(Account account, BigDecimal cashValue) {
        BigDecimal taxPercentageValue = cashValue
                .multiply(BigDecimal.valueOf(0.15))
                .divide(new BigDecimal(100), RoundingMode.HALF_DOWN);

        cashValue = cashValue.add(taxPercentageValue);
        account.removeMoney(cashValue);
    }
}
```

EmailSenderImpl.java | SellTransaction.java

src > main > java > com > exercicio02 > solid > SellTransaction.java > {} com.exercicio02.solid

```java
package com.exercicio02.solid;

import java.math.BigDecimal;
import java.math.RoundingMode;

public class SellTransaction implements TransactionStrategy {

    @Override
    public void doTransaction(Account account, BigDecimal cashValue) {
        BigDecimal taxPercentageValue = cashValue
                .multiply(BigDecimal.valueOf(0.1))
                .divide(new BigDecimal(100), RoundingMode.HALF_DOWN);

        cashValue = cashValue.add(taxPercentageValue.negate());
        account.addMoney(cashValue);
    }
}
```

EXPLORADOR

BSOLID

src | IIMPLEMENTACAO

- main
- java / com
  - exercicio01
  - exercicio02 / solid
    - Account.java
    - BuyTransaction.java
    - DepositTransaction.java
    - Program.java
    - SellTransaction.java
    - TransactionStrategy.java
    - UML_Exercicio_02.jpeg
    - WithdrawalTransaction.java

README.md | DepositTransaction.java

src > main > java > com > exercicio02 > solid > DepositTransaction.java > {} com.exercicio02.solid

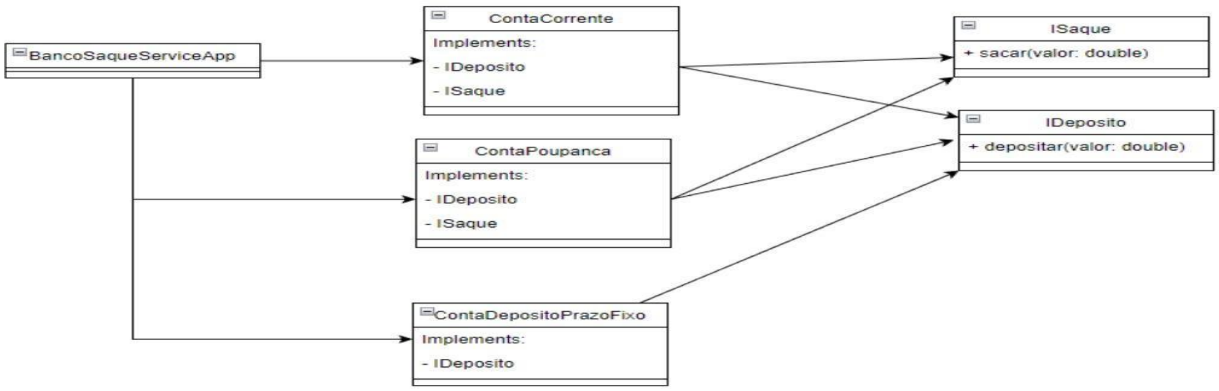
```java
package com.exercicio02.solid;

import java.math.BigDecimal;
import java.math.RoundingMode;

public class DepositTransaction implements TransactionStrategy {
    @Override
    public void doTransaction(Account account, BigDecimal cashValue) {
        BigDecimal taxPercentageValue = cashValue
                .multiply(BigDecimal.valueOf(0.05))
                .divide(new BigDecimal(100), RoundingMode.HALF_DOWN);

        cashValue = cashValue.add(taxPercentageValue);
        account.addMoney(cashValue);
    }
}
```

Exercício 03

Para esse exercício, imagine ainda que você trabalha para o banco e que exista um problema de design que para se aplicar o LSP, será necessário aplicar o OCP antes.

O diagrama a seguir mostra como está o sistema para ContaCorrente e ContaPoupanca. Esse diagrama representa o Design atual da aplicação, e que não está correto, de acordo com as práticas do SOLID. Abaixo estarei listando os problemas: O BancoSaqueServiceApp tem a instância das duas implementações, logo ela está ciente de duas implementações concretas, logo, o BancoSaqueServiceApp precisaria ser alterado toda vez que um novo tipo de conta fosse introduzido. Utilizando o princípio de Aberto e Fechado, podemos fazer o código mais pragmático e resolver o primeiro problema de uma forma simples. Criamos uma classe abstrata Conta no qual ContaCorrente e ContaPoupanca herdam. Pronto, menos um problema, temos então BancoSaqueServiceApp não dependendo mais de uma classe concreta, pois agora depende de uma abstração. Com o design abaixo, BancoSaqueServiceApp fica aberto para extensão para novos tipos de conta mas fechado para modificação. Logo ele é obrigado a implementar o método de sacar sem precisar, afinal é uma conta de depósito com prazo fixo. Com isso, fica claro para você que este design não está correto? BancoSaqueServiceApp e um cliente da classe conta, ela espera que todas contas e seus subtipos tenham um comportamento permitindo o saque, regra de negócio principal para a classe de serviço. Mas essa nova classe não tem essa função, o que acaba acontecendo por violar os princípios de substituição de Liskov. De acordo com esse princípio, qual seria o melhor design (UML) e implementação Java de código para esse problema?

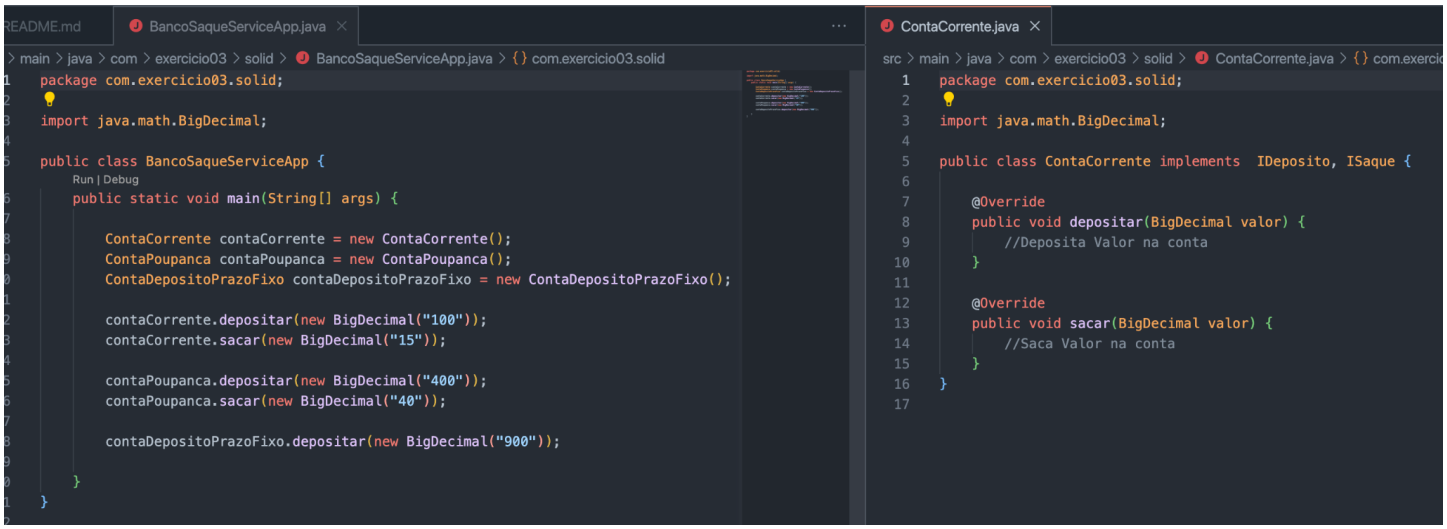
UML



Para corrigir a violação do Princípio de Substituição de Liskov, podemos dividir a classe abstrata "Conta" em duas interfaces, uma para operações de depósito e outra para operações de saque. Vamos chamar essas interfaces de "IDeposito" e "ISaque". Dessa forma, as classes "ContaCorrente" e "ContaPoupanca" podem implementar

ambas as interfaces, enquanto "ContaDepositoPrazoFixo" implementará apenas a interface "IDeposito".

IMPLEMENTAÇÃO



```java
package com.exercicio03.solid;

import java.math.BigDecimal;

public class BancoSaqueServiceApp {
    Run | Debug
    public static void main(String[] args) {

        ContaCorrente contaCorrente = new ContaCorrente();
        ContaPoupanca contaPoupanca = new ContaPoupanca();
        ContaDepositoPrazoFixo contaDepositoPrazoFixo = new ContaDepositoPrazoFixo();

        contaCorrente.depositar(new BigDecimal("100"));
        contaCorrente.sacar(new BigDecimal("15"));

        contaPoupanca.depositar(new BigDecimal("400"));
        contaPoupanca.sacar(new BigDecimal("40"));

        contaDepositoPrazoFixo.depositar(new BigDecimal("900"));

    }
}
```

```java
package com.exercicio03.solid;

import java.math.BigDecimal;

public class ContaCorrente implements  IDeposito, ISaque {

    @Override
    public void depositar(BigDecimal valor) {
        //Deposita Valor na conta
    }

    @Override
    public void sacar(BigDecimal valor) {
        //Saca Valor na conta
    }
}
```

```java
package com.exercicio03.solid;

import java.math.BigDecimal;

public class ContaDepositoPrazoFixo implements IDeposito {
    @Override
    public void depositar(BigDecimal valor) {
        //Deposita Valor na conta
    }
}
```

```java
package com.exercicio03.solid;

import java.math.BigDecimal;

public class ContaPoupanca implements  IDeposito, ISaque {
    @Override
    public void depositar(BigDecimal valor) {
        //Deposita Valor na conta
    }

    @Override
    public void sacar(BigDecimal valor) {
        //Saca Valor na conta
    }
}
```

```java
package com.exercicio03.solid;

import java.math.BigDecimal;

public interface ISaque {
    void sacar(BigDecimal valor);

}
```

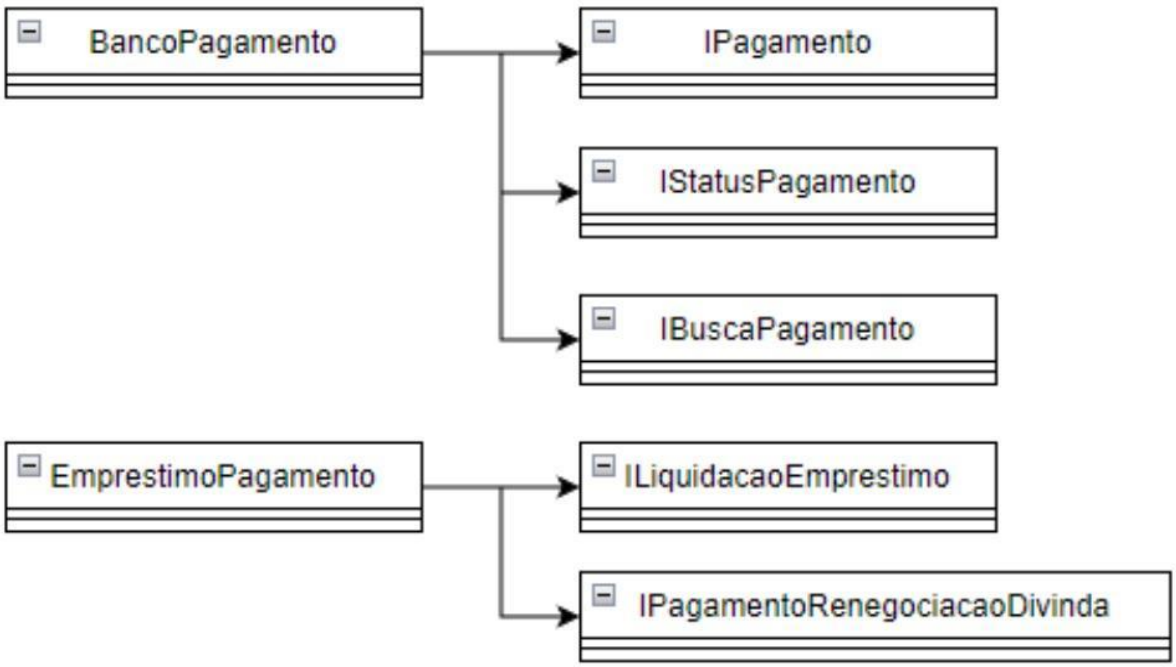
```java
package com.exercicio03.solid;

import java.math.BigDecimal;

public interface IDeposito {
    void depositar(BigDecimal valor);
}
```

Exercício 04

Considerando o princípio de segregação de interfaces, imagina que você trabalha em uma empresa financeira, na área de Pagamentos do Banco. Você é o engenheiro sênior responsável por um novo banco e irá começar a fazer o desenvolvimento de um

dos produtos do banco no qual uma equipe de programadores seniores começaram mas sem o conhecimento do SOLID. Quando você chega na empresa, se depara com uma interface Pagamentos com 3 métodos(vide diagrama abaixo) além de uma classe que a implementa BancoPagamento. A priori, a implementação dos três métodos (iniciar Pagamentos, status e buscar Pagamentos nao fere o princípio. Mas alguém teve a brilhante ideia de adicionar uma nova classe EmprestimoPagamento. Com isso foi adicionado na interface Pagamento dois outros métodos, são eles o iniciarLiquidacaoEmprestimo e iniciarPagamentoRenegociacaoDivida (vide diagrama abaixo). O problema é que para essa nova classe, ela não aceita iniciarPagamentos pois somente deve ser utilizado para operações de Banco. Assim como Liquidação de Empréstimo e Renegociação de Dívida deve pertencer apenas a nova classe criada. Baseado na aula sobre SOLID como podemos fazer um novo design e implementação em Java de classe que atenda o princípio de segregação de interface.

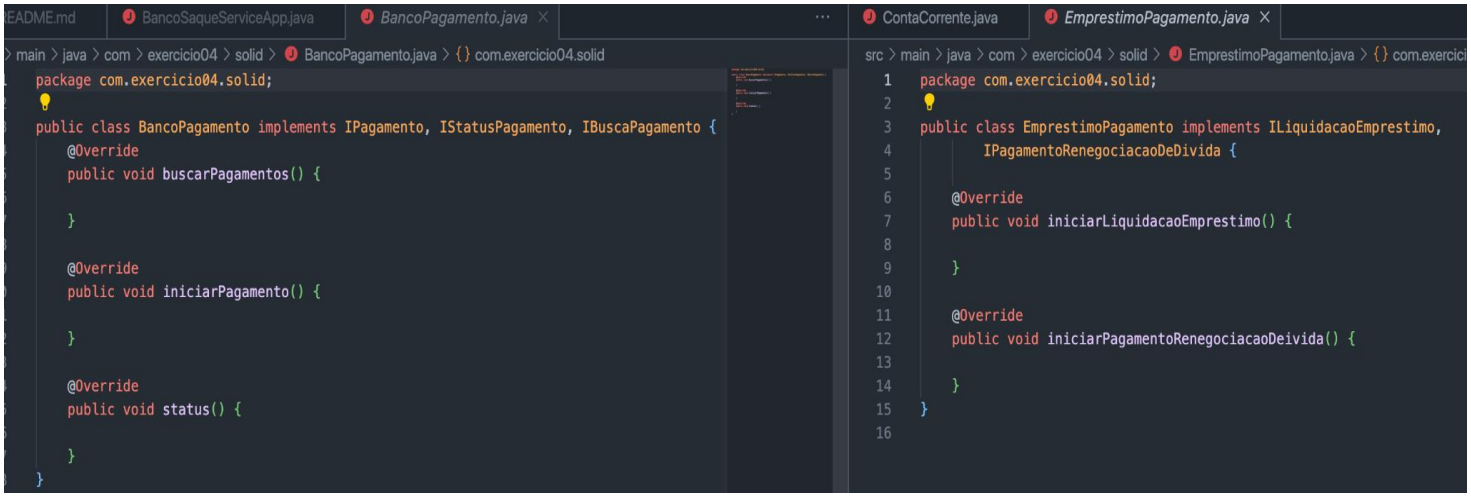
UML

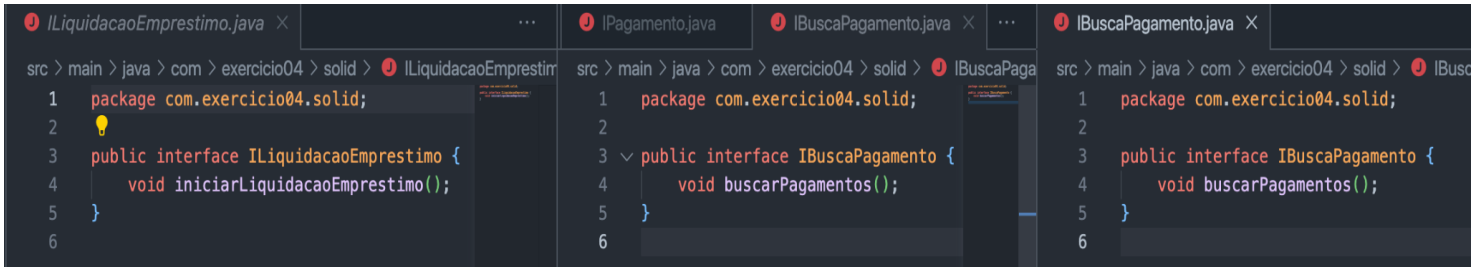


Para atender ao princípio de segregação de interfaces, é necessário dividir a interface "Pagamento" em várias interfaces menores e mais específicas, de modo que cada classe implemente apenas as interfaces necessárias para suas funcionalidades. Criaremos quatro interfaces: IPagamento, IStatusPagamento, ILiquidacaoEmprestimo e

IPagamentoRenegociacaoDivida. Assim, a classe BancoPagamento implementará as três primeiras interfaces (IPagamento, IStatusPagamento e IBuscaPagamento), enquanto a classe EmprestimoPagamento implementará as duas últimas (ILiquidacaoEmprestimo e IPagamentoRenegociacaoDivida).

IMPLEMENTAÇÃO



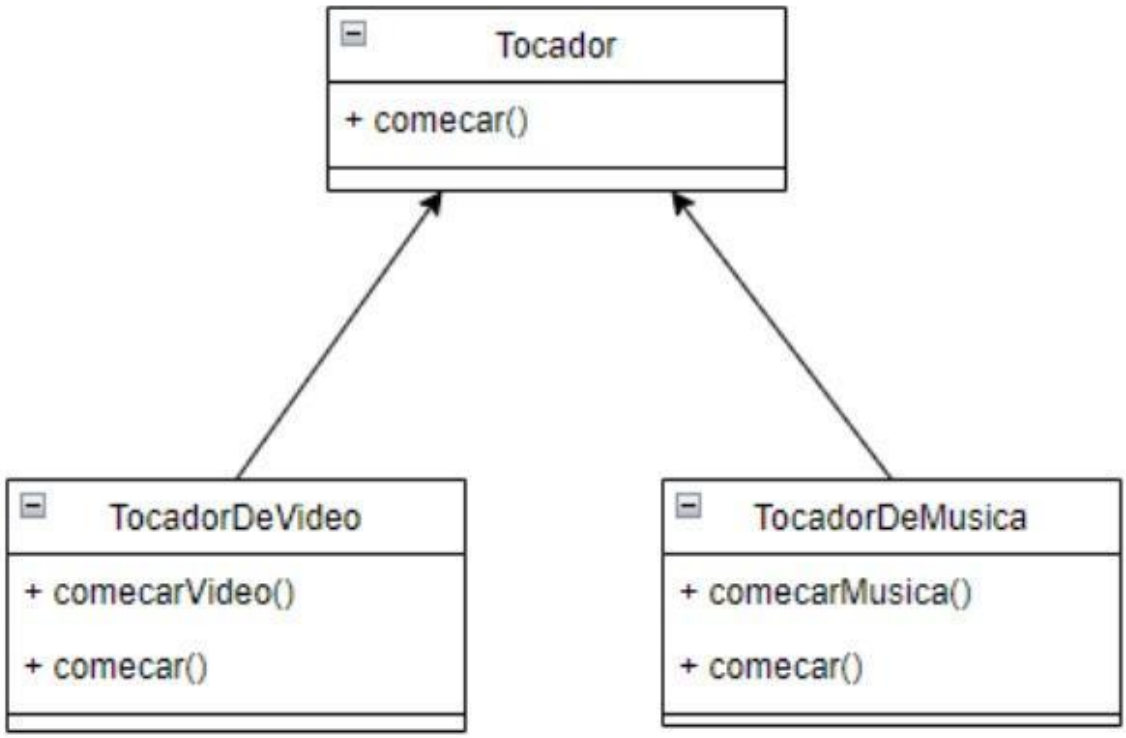


Exercício 05

Imagine agora que tu foi trabalhar numa empresa inovadora que é concorrente do Spotify e Netflix, seria uma multi plataforma de Streaming de músicas e vídeos. Imagina que a empresa se chama MultiPlayStream e você é o engenheiro/arquiteto

responsável em implementar uma das funcionalidades. Acontece que seu programador junior, não conhecendo muito de programação, fez o seguinte design abaixo. Crie o diagrama de classe e a implementação em Java que corrija o design e considere o princípio de segregação de interfaces.

UML



O princípio de segregação de interfaces sugere que as interfaces devem ser divididas em conjuntos de métodos coesos e relacionados, de forma que as classes que implementam as interfaces não precisem implementar métodos que não são relevantes para elas. Com base nesse princípio, podemos criar duas interfaces separadas, uma para cada método, e então implementá-las em suas respectivas classes.

IMPLEMENTAÇÃO

TocadorDeMusica.java

```java
package com.exercicio05.solid;

public class TocadorDeMusica implements Tocador{

    @Override
    public void comecar() {
        comecarMusica();
    }

    private void comecarMusica() {
    }

}
```

ContaCorrente.java

```java
package com.exercicio03.solid;

import java.math.BigDecimal;

public class ContaCorrente implements  IDeposito, ISaque {

    @Override
    public void depositar(BigDecimal valor) {
        //Deposita Valor na conta
    }

    @Override
    public void sacar(BigDecimal valor) {
        //Saca Valor na conta
    }
}
```

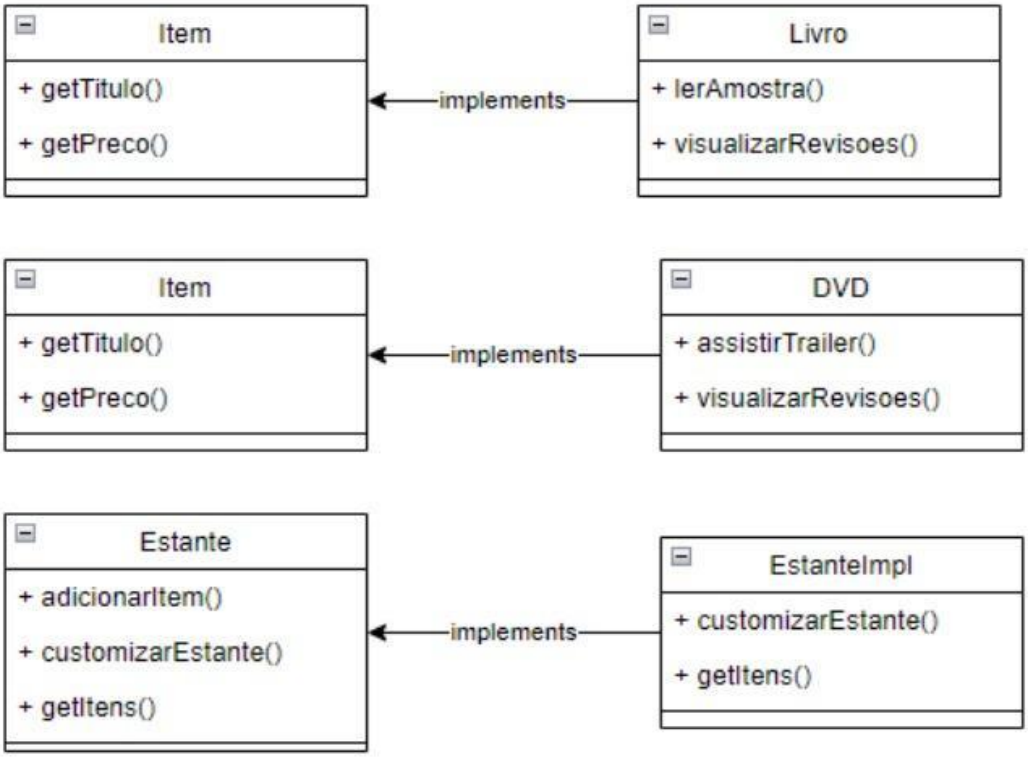
TocadorDeVideo.java

```java
package com.exercicio05.solid;

public class TocadorDeVideo implements Tocador{

    @Override
    public void comecar() {
        comecarVideo();
    }

    private void comecarVideo() {
    }

}
```

Exercício 06

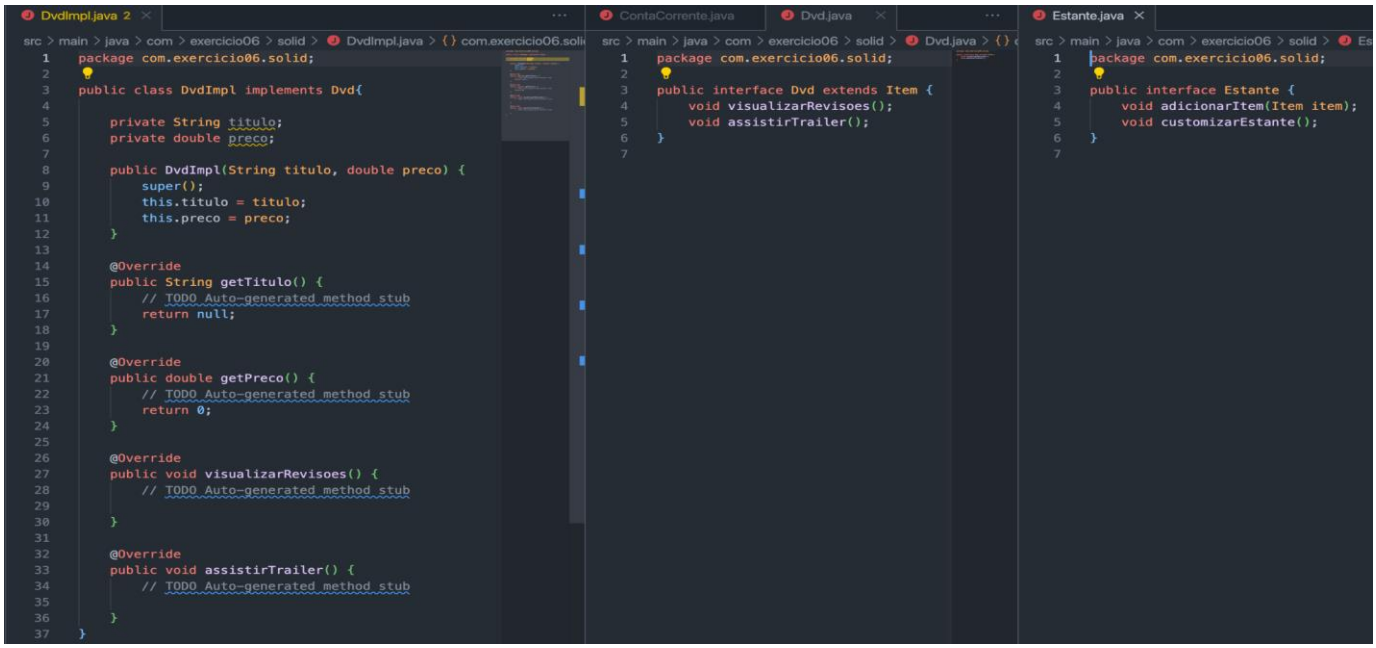
Baseado no que vimos sobre o princípio da inversão de dependência, imagina que trabalhamos numa livraria e precisamos desenvolver um sistema onde cada Estante pode ter uma coleção de Livros e DVDs. Os livros possuem dois métodos, visualizarRevisoes() e lerAmostra(), já o objeto onde fica o livro e DVDs devem ter adicionar Livro ou DVDs e customizarEstante(). Já o objeto DVD tem visualizarRevisoes() e assistirTrailer(). Aplicando o princípio em questão, como que poderíamos ter o design e implementação Java dessas classes?

UML



Para aplicar o princípio da inversão de dependência, devemos criar interfaces que definem os comportamentos que cada classe deve ter. Dessa forma, as classes podem depender das interfaces, em vez de depender diretamente de outras classes.

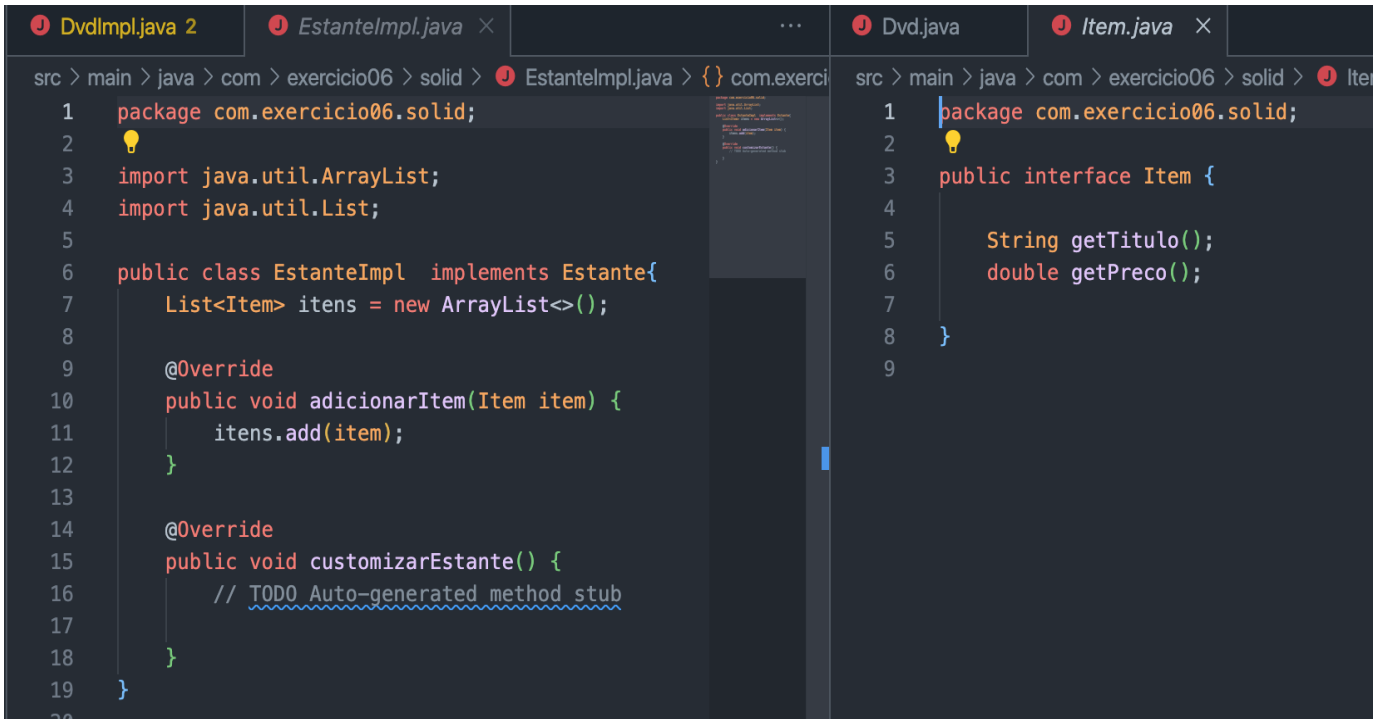
IMPLEMENTAÇÃO



```java
package com.exercicio06.solid;

public class DvdImpl implements Dvd{

    private String titulo;
    private double preco;

    public DvdImpl(String titulo, double preco) {
        super();
        this.titulo = titulo;
        this.preco = preco;
    }

    @Override
    public String getTitulo() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public double getPreco() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public void visualizarRevisoes() {
        // TODO Auto-generated method stub

    }

    @Override
    public void assistirTrailer() {
        // TODO Auto-generated method stub

    }
}
```

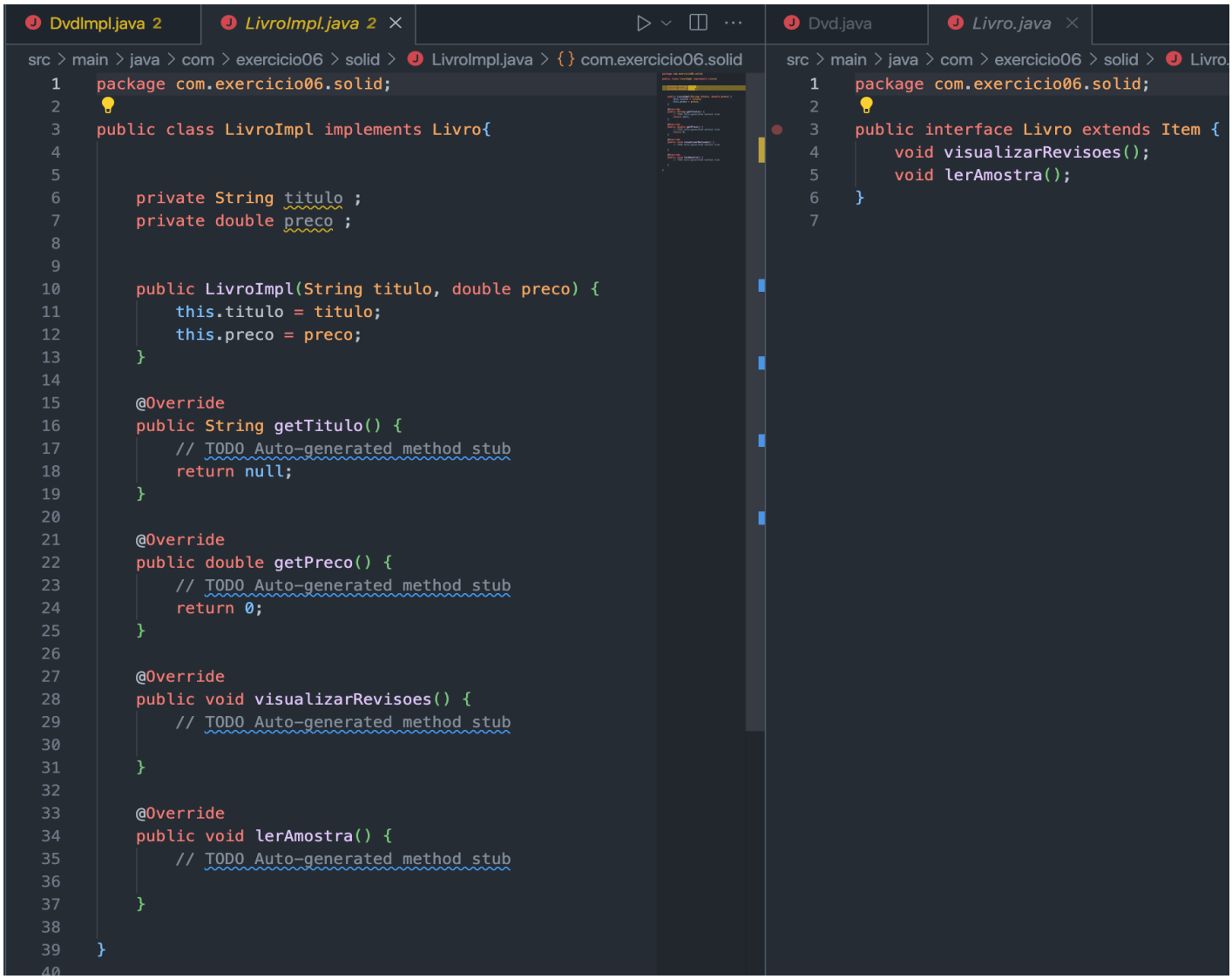
```java
package com.exercicio06.solid;

public interface Dvd extends Item {
    void visualizarRevisoes();
    void assistirTrailer();
}
```

```java
package com.exercicio06.solid;

public interface Estante {
    void adicionarItem(Item item);
    void customizarEstante();
}
```



```java
package com.exercicio06.solid;

import java.util.ArrayList;
import java.util.List;

public class EstanteImpl  implements Estante{
    List<Item> itens = new ArrayList<>();

    @Override
    public void adicionarItem(Item item) {
        itens.add(item);
    }

    @Override
    public void customizarEstante() {
        // TODO Auto-generated method stub

    }
}
```

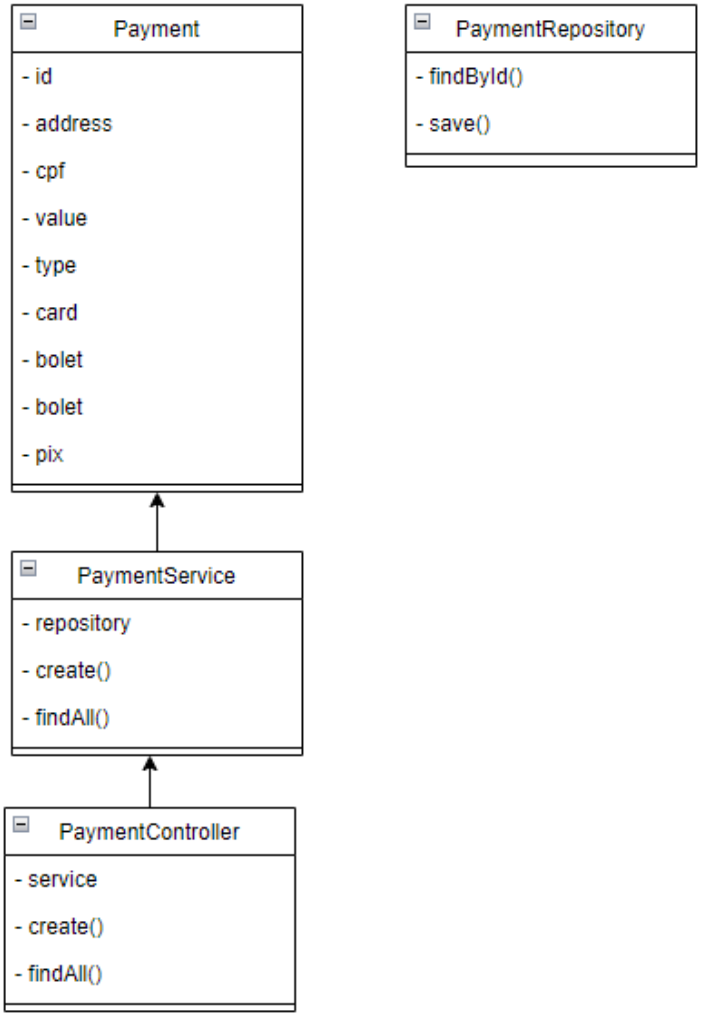
```java
package com.exercicio06.solid;

public interface Item {

    String getTitulo();
    double getPreco();

}
```



Exercício 07

DESAFIO EXTRA: Você agora é responsável em criar um novo projeto que irá se chamar pagamento-service. Para esse pagamento você pode aceitar apenas cartões de crédito, PIX e Boleto. Para o pagar é necessário enviar endereço, cpf, valor. Mas se a compra for no cartão de crédito a vista ou parcelado, caso seja parcelado enviar quantidade de parcelas, dados do cartão de crédito (nome impresso no cartão, número do cartão, data de validade e CVV (3 dígitos atrás do cartão)). Caso seja o boleto deverá enviar o codigo de barras e caso seja o pix a chave PIX. Com isso criar um PagamentoController com o método POST para submeter um pagamento e um GET para buscar todos os pagamentos. Em seguida criar um PagamentoService e um PagamentoRepository (esse último os dados precisam ser persistidos na base de dados H2 que é in memory). Não se esqueça de aplicar SOLID.

UML



IMPLEMENTAÇÃO

DvdImpl.java 2 | CashPaymentInfo.java 1

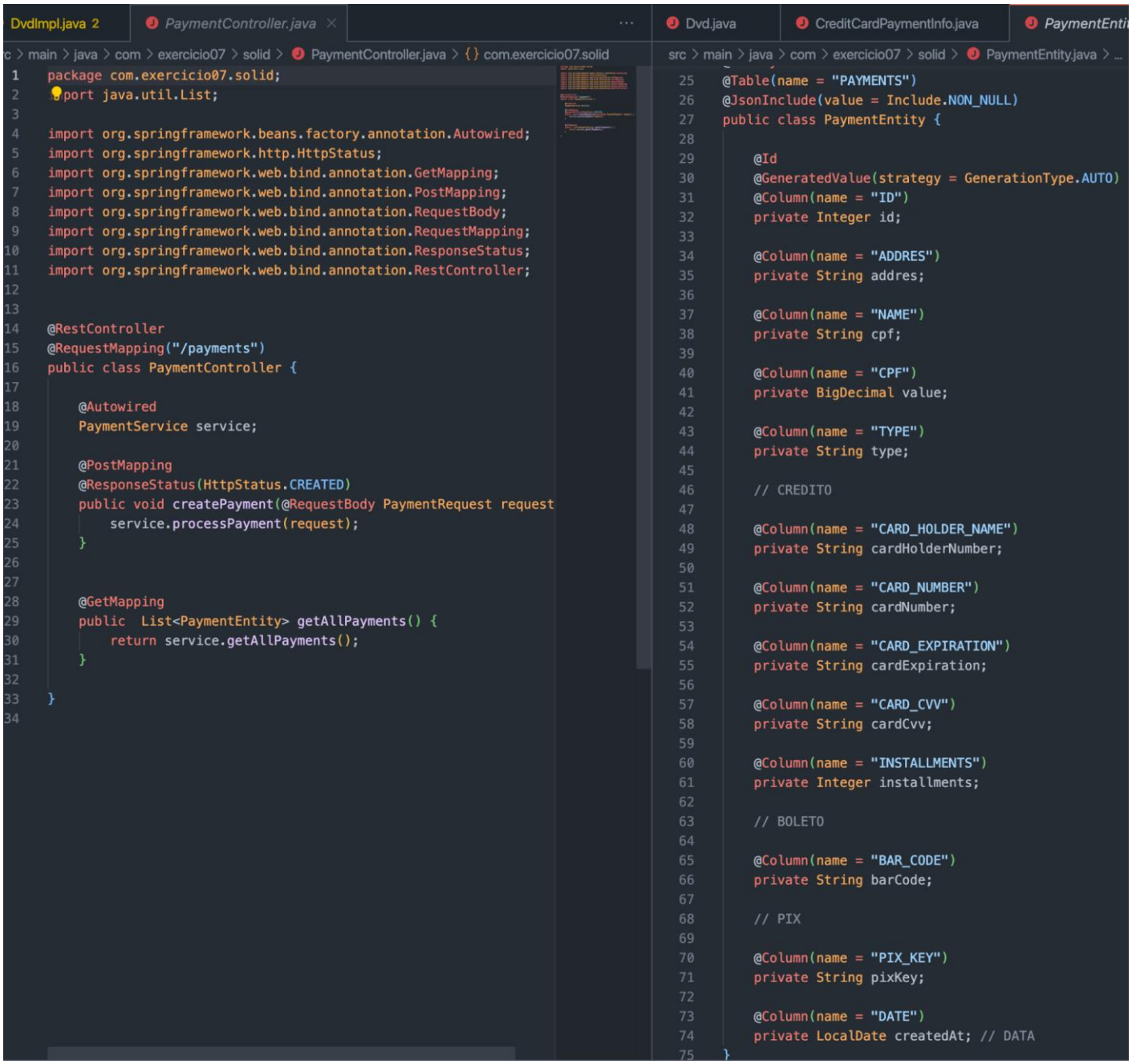
c > main > java > com > exercicio07 > solid > CashPaymentInfo.java > { } com.exercic

```java
package com.exercicio07.solid;
import com.fasterxml.jackson.annotation.JsonProperty;

import java.time.LocalDateTime;

public class CashPaymentInfo {
    @JsonProperty("pixKey")
    private String pixKey;
    @JsonProperty("barCode")
    private String barCode;

    public String getPixKey() {
        return pixKey;
    }

    public void setPixKey(String pixKey) {
        this.pixKey = pixKey;
    }

    public String getBarCode() {
        return barCode;
    }

    public void setBarCode(String barCode) {
        this.barCode = barCode;
    }
}

```

Dvd.java | CreditCardPaymentInfo.java

src > main > java > com > exercicio07 > solid > CreditCardPaymentInfo.java > { } com.

```java
public class CreditCardPaymentInfo {
    @JsonProperty("cardHolderName")
    private String cardHolderName;
    @JsonProperty("cardNumber")
    private String cardNumber;
    @JsonProperty("expirationDate")
    private String expirationDate;
    @JsonProperty("cvv")
    private String cvv;
    @JsonProperty("installments")
    private Integer installments;

    public String getCardHolderName() {
        return cardHolderName;
    }

    public void setCardHolderName(String cardHolderName) {
        this.cardHolderName = cardHolderName;
    }

    public String getCardNumber() {
        return cardNumber;
    }

    public void setCardNumber(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public String getExpirationDate() {
        return expirationDate;
    }

    public void setExpirationDate(String expirationDate) {
        this.expirationDate = expirationDate;
    }

    public String getCvv() {
        return cvv;
    }

    public void setCvv(String cvv) {
        this.cvv = cvv;
    }

    public Integer getInstallments() {
        return installments;
    }

    public void setInstallments(Integer installments) {
        this.installments = installments;
    }
```



Como a implementação do exercício 07 é extensa, apresentei apenas algumas class o conteúdo e fontes desse e dos demais exercicios estao disponiveis e compartilhados em: https://github.com/guima147/JobSOLID.git

BIBLIOGRAFIA

Gitbook Tutorial java. Disponível em: https://glysns.gitbook.io/java-basico/sintaxe/documentacao. Acesso em 18.Abr.2023

W3schools tutoriais java. disponível em : https://www.w3schools.com/java/ . Acesso em 18.Abr.2023