Universidade Federal
do Espírito Santo

GUIA DE RESPOSTAS

S.O.L.I.D. E DESIGN PATTERNS

CADERNO DE EXERCÍCIOS

PRATIQUE E EVOLUA

CLAYTON VIEIRA FRAGA FILHO

A imagem da capa foi com auxílio da tecnologia de IA da OpenAI's DALL·E

Guia de Respostas

S.O.L.I.D. e Design Patterns: caderno de exercícios: pratique e evolua

Clayton Vieira Fraga Filho

Universidade Federal do Espírito Santo

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Central da Universidade Federal do Espírito Santo, ES, Brasil)

Fraga Filho, Clayton Vieira, 1978-
F811p Guia de Respostas: S.O.L.I.D. e Design Patterns: caderno de exercícios: pratique e evolua [recurso eletrônico] / Clayton Vieira Fraga Filho. – Alegre, ES : UFES, 2024.
474 p. : il.

ISBN: 978-65-01-07208-1

1. Programação (Computadores). 2. Software – Desenvolvimento. 3. Java (Linguagem de programação de computador). 4. UML (Computação). I. Título.

CDU: 004.42

Elaborado por Manuela Barbosa Garcia – CRB-6 ES-000980/O

Apresentação

Bem-vindos ao "Guia de Respostas” do "Caderno de Exercícios: Pratique e evolua seu aprendizado". Este material foi desenvolvido para complementar seu estudo dos princípios SOLID e padrões de projeto em software.

Neste guia, você encontrará respostas detalhadas para cada exercício, elucidando as práticas recomendadas no desenvolvimento de software, com explicações que evidenciam a aplicação dos princípios SOLID e dos diversos padrões de design abordados. Esperamos que essa abordagem possa fortalecer seu aprendizado como cada conceito teórico é implementado na prática.

Recomendo que faça uso deste caderno de respostas em conjunto com a prática ativa e o diálogo com colegas e professores. Utilize este material como uma referência contínua durante seus estudos.

Agradecimento especial ao discente Gabriel dos Santos Souza, que realizou, sob minha supervisão, toda a revisão técnica e a resolução de parte dos exercícios.

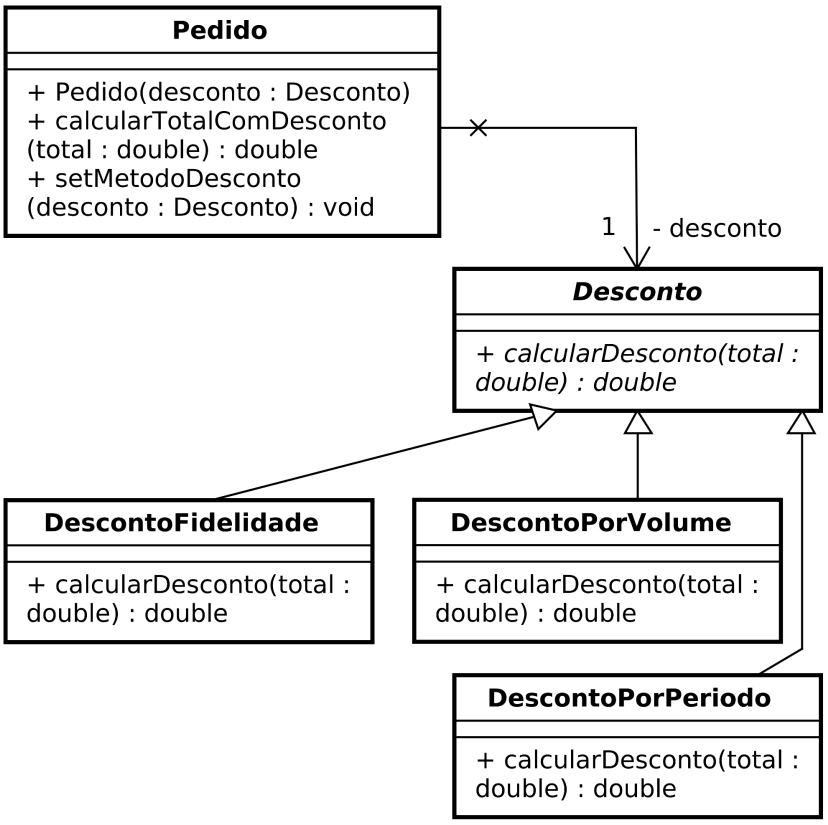
Desejo a todos um aprofundamento proveitoso nos estudos e na prática dos princípios SOLID e dos padrões de projeto.

Prof. Clayton Vieira Fraga Filho

Sumário

Solução do Exercício 1

No diagrama de classes abaixo, a classe abstrata Desconto que contém o método abstrato calcularDesconto(double total). As classes DescontoFidelidade, DescontoPorVolume e DescontoPorPeriodo estendem a classe Desconto, portanto com a obrigação de implementar o método calcularDesconto. Já a classe Pedido possui um atributo tipo Desconto, no seu construtor é esperado um parâmetro do tipo abstrato Desconto, a classe Pedido também contém os métodos calcularTotalComDesconto e setMetodoDesconto.



Agora, veja abaixo a implementação em código Java. Inicialmente é codificado a classe abstrata Desconto e as classes específicas que a estendem. Para fins deste exercício foi utilizado 10% para desconto de fidelidade, 5% para desconto por volume e 7% para desconto por período.

```
public abstract class Desconto {
  public abstract double calcularDesconto(double total);
}
```

```
public class DescontoFidelidade extends Desconto {
  @Override
  public double calcularDesconto(double total) {
    return total * 0.10;
  }
}

public class DescontoPorVolume extends Desconto {
  @Override
  public double calcularDesconto(double total) {
    return total * 0.05;
  }
}

public class DescontoPorPeriodo extends Desconto {
  @Override
  public double calcularDesconto(double total) {
    return total * 0.07;
  }
}
```

Em seguida é codificado a classe Pedido, observe que o método setMetodoDesconto espera um parâmetro tipo Desconto, a utilização deste método possibilita a alternância de método de desconto em tempo de execução do sistema.

```
public class Pedido {
  private Desconto desconto;

  public Pedido(Desconto desconto) {
    this.desconto = desconto;
  }

  public double calcularTotalComDesconto(double total) {
    double descontoAplicado = desconto.calcularDesconto(this.total);
```

```
        return this.total - descontoAplicado;
    }

    public void setMetodoDesconto(Desconto desconto){
        this.desconto = desconto;
    }
}
```

Por fim, é criado a classe Principal para demonstrar o uso da solução. Em seu código, são instanciados objetos das classes DescontoFidelidade, DescontoPorVolume e DescontoPorPeriodo, em seguida são criados os pedidos A e B, cada um deles já com um tipo de desconto passado como parâmetro no construtor. Logo é utilizado o método calcularTotalComDesconto e exibido o resultado no console.

Ainda na classe Principal, demonstramos o uso da alternância de métodos de desconto em tempo de execução, que acontence quando o pedido B invoca o método setMetodoDesconto(descontoPorPeríodo) e então chama o método calcularTotalComDesconto novamente.

```
public class Principal {
    public static void main(String[] args) {
        Desconto descontoFidelidade = new DescontoFidelidade();
        Desconto descontoPorVolume = new DescontoPorVolume();
        Desconto descontoPorPeriodo = new DescontoPorPeriodo();

        Pedido pedidoA = new Pedido(descontoFidelidade);
        Pedido pedidoB = new Pedido(descontoPorVolume);

        double totalPedidoA = 100.0;
        double totalPedidoB = 200.0;

        double totalComDescontoPedidoA = pedidoA.calcularTotalComDesconto(totalPedidoA);
        double totalComDescontoPedidoB = pedidoB.calcularTotalComDesconto(totalPedidoB);
```

```
        System.out.println("Total do Pedido A com desconto de fidelidade: " +
totalComDescontoPedidoA);
        System.out.println("Total do Pedido B com desconto por volume: " +
totalComDescontoPedidoB);

        pedidoB.setMetodoDesconto(descontoPorPeriodo);
        totalComDescontoPedidoB =
pedidoB.calcularTotalComDesconto(totalComDescontoPedidoB);

        System.out.println("Total do Pedido B com desconto por período: " +
totalComDescontoPedidoB);
    }
}
```

Esta abordagem permite adicionar novos tipos de desconto sem modificar a classe Pedido, seguindo o Princípio Aberto/Fechado.

Solução do Exercício 2

No sistema de registro de logs para uma aplicação, é necessário garantir que apenas uma instância do objeto de registro de logs seja criada durante a execução do programa, evitando problemas de concorrência de acesso ao arquivo de log e desperdício de recursos. Para solucionar essa questão, o padrão de projeto Singleton deve ser implementado, pois assegura a criação de apenas uma instância da classe de registro de logs, denominada Logger, foi projetada. Ela contém um método para gravar informações de log em um arquivo específico. Ao implementar o Singleton, a classe terá um método estático getInstance(), característica marcante deste padrão, para acessar a única instância da classe.

Veja o código Java da classe Logger a seguir.

```
public class Logger {
    private static Logger instancia;
    private File logFile;
```

```
  private Logger() {
    logFile = new File("log.txt");
  }

  public static Logger getInstancia() {
    if (instancia == null) {
      instancia = new Logger();
    }
    return instancia;
  }

  public void log(String message) {
    try {
      FileWriter writer = new FileWriter(logFile, true);
      PrintWriter printer = new PrintWriter(writer);
      printer.println(message);
      printer.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

O construtor da classe Logger cria um arquivo único para log chamado log.txt. Já no método log é realizado a inserção da mensagem log no arquivo.

Utilizaremos as classes Pedido e DescontoFidelidade da solução do exercício 1 como exemplo para o registro de log em várias classes distintas, observe a seguir a adição do método privado registraLog nas classes citadas.

```
public class DescontoFidelidade extends Desconto {
  @Override
  public double calcularDesconto(double total) {
```

```
        registraLog("Desconto fidelidade aplicado.");
        return total * 0.10;
    }

    private void registraLog(String mensagem){
        Logger logger = Logger.getInstancia();

        logger.log(mensagem);
    }
}

public class Pedido {
    private Desconto desconto;
    public Pedido(Desconto desconto) {
        this.desconto = desconto;
    }
    public double calcularTotalComDesconto(double total) {
        double descontoAplicado = desconto.calcularDesconto(this.total);
        registraLog("Desconto valor " + descontoAplicado +" foi aplicado no pedido: " + this);
        return this.total - descontoAplicado;
    }
    public void setMetodoDesconto(Desconto desconto){
        this.desconto = desconto;
        registraLog("Método de desconto alterado");
    }

    private void registraLog(String mensagem){
        Logger logger = Logger.getInstancia();

        logger.log(mensagem);
    }
}
```

Agora, para demonstrar o uso da solução é criado a classe Principal, que inicialmente obtém a instancia de Logger e registra a inicialização do sistema. Em seguida são criadas instâncias da classe DescontoFidelidade e Pedido e utilizado o método calcularTotalComDesconto.

```
public class Principal {
  public static void main(String[] args) {
    Logger logger = Logger.getInstancia();

    logger.log("Iniciando aplicação...");

    Desconto descontoFidelidade = new DescontoFidelidade();
    Pedido pedido = new Pedido(descontoFidelidade);
    double valorPedido = 100;

    pedido.calcularTotalComDesconto(valorPedido);
  }
}
```

O resultado esperado no arquivo log.txt após a execução desta implementação está a seguir.

Inicializando aplicação...
Desconto fidelidade aplicado.
Desconto valor 10.0 aplicado no pedido {Pedido}

Solução do Exercício 3

Para solucionar esta questão de monitoramento de ações, foi adotada uma abordagem que segue os princípios SOLID, visando criar um sistema flexível e extensível. A ideia principal é permitir que investidores se inscrevam para receber notificações sobre alterações no preço ou na data de cotação de ações específicas, sem que seja necessário modificar a classe que representa a ação sempre que novos tipos de investidores ou métodos de notificação forem adicionados.

Inicialmente foi criada a interface INotificacao que define o contrato para enviar notificações. Ela possui um único método, enviarNotificacao, que recebe um observador e uma mensagem como parâmetros. Essa abstração permite que diferentes formas de notificação sejam implementadas, como e-mail, SMS, entre outras, sem afetar o código das classes que utilizam o serviço de notificação.

As implementações concretas, NotificacaoEmailImpl e NotificacaoSMSImpl, são responsáveis por fornecer a lógica específica para enviar notificações por e-mail e SMS, respectivamente. Ambas implementam o método enviarNotificacao da interface INotificacao, garantindo que sigam o contrato estabelecido e possam ser utilizadas de forma intercambiável pelo serviço de gerenciamento de notificações.

A classe GerenciadorNotificacaoService desempenha o papel de gerenciar o envio de notificações para os observadores. Ela mantém uma lista de diferentes métodos de notificação disponíveis e oferece um método dispararNotificacao, que itera sobre esses métodos e envia a mensagem para cada um deles. Essa abstração permite uma fácil extensão do sistema para suportar novos métodos de notificação no futuro, sem a necessidade de modificar a lógica de notificação existente.

Portanto esta implementação se torna um módulo de notificação, veja o código Java a seguir.

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

public interface INotificacao {
  void enviarNotificacao(IObservador observador, String mensagem);
}

public class NotificacaoEmailImpl implements INotificacao {
  @Override
  public void enviarNotificacao(IObservador observador, String mensagem) {
    System.out.println("Enviando e-mail: " + mensagem);
    // Lógica para enviar e-mail
  }
}
```

```
public class NotificacaoSMSImpl implements INotificacao {
  @Override
  public void enviarNotificacao(IObservador observador, String mensagem) {
    System.out.println("Enviando SMS: " + mensagem);
    // Lógica para enviar SMS
  }
}

public class GerenciadorNotificacaoService {
  private List<INotificacao> metodosNotificacao;

  public GerenciadorNotificacaoService(){
    this.metodosNotificacao = new ArrayList<>();
  }

  void dispararNotificacao(IObservador observador, String mensagem) {
    for (INotificacao metodo : metodosNotificacao) {
      metodo.enviarNotificacao(observador, mensagem);
    }
  }
}
```

Agora, será codificada a interface IObservador define um contrato para os observadores que recebem notificações sobre as ações. Ela possui um único método, notificar, que recebe um objeto observável como parâmetro. Essa abstração permite que diferentes tipos de observadores possam ser criados e recebam notificações de mudanças em objetos observáveis, como instâncias da classe Acao.

A classe InvestidorImpl implementa o comportamento específico de um observador, que é um investidor interessado em receber notificações sobre as ações. Ela mantém informações sobre o nome do investidor e implementa o método notificar, conforme exigido pela interface IObservador. Além disso, ela oferece um método realizaInscricao, que permite que um investidor se inscreva para receber notificações sobre uma ação específica.

Internamente, ele chama o método adicionarObservador da classe Acao para registrar o investidor como um observador da ação. Essa classe demonstra a flexibilidade do sistema, permitindo que diferentes tipos de observadores possam se inscrever para receber notificações sobre objetos observáveis no sistema.

O código Java é apresentado abaixo:

```
public interface IObservador {
  void notificar(IObservavel observavel);
}

public class InvestidorImpl implements IObservador {
  private String nome;

  public InvestidorImpl(String nome) {
    this.nome = nome;
  }

  @Override
  public void realizaInscricao(Acao acao) {
    acao.adicionarObservador(this);
  }
}
```

Em seguida, é criada a classe Acao que representa uma ação no sistema. Ela mantém o estado da ação, incluindo código, nome, preço e data de cotação, e oferece métodos para adicionar e remover observadores interessados na ação. Quando há alterações no estado da ação, como mudanças no preço ou na data, a classe Acao notifica os observadores registrados por meio do serviço de gerenciamento de notificações, garantindo a separação de responsabilidades e o cumprimento do princípio de responsabilidade única (SRP).

A implementação em código Java da classe Acao está a seguir.

```
public class Acao {
  private String codigo;
```

```
    private String nome;
    private double preco;
    private String dataCotacao;
    private List<IObservador> observadores;
    private GerenciadorNotificacaoService notificadorService;

    public Acao(String codigo, String nome, double preco, String data) {
        this.codigo = codigo;
        this.nome = nome;
        this.preco = preco;
        this.dataCotacao = data;
        this.observadores = new ArrayList<>();
        this.notificadorService = new GerenciadorNotificacaoService();
    }

    public void setPreco(double preco) {
        this.preco = preco;
        notificarObservadores("Notificação Ação " + this.nome + " teve seu preço alterado para " +
this.preco + " na data " + this.dataCotacao);
    }

    public void setData(String data) {
        this.data = data;
        notificarObservadores("Notificação Ação " + this.nome + " teve sua data de cotacao
alterada para " + this.preco + " na data " + this.dataCotacao);
    }

    @Override
    public void adicionarObservador(IObservador observador) {
        observadores.add(observador);
    }
```

```
    @Override
    public void removerObservador(IObservador observador) {
        observadores.remove(observador);
    }

    @Override
    private void notificarObservadores(String mensagem) {
        for (IObservador observador : observadores) {
            notificadorService.dispararNotificacao(observador, mensagem);
        }
    }
    //Getters e setters
}
```

Para demonstrar o uso do código fornecido, criaremos uma classe Principal com um método main. Esta classe será responsável por simular a interação entre investidores e ações, demonstrando como os investidores se inscrevem para receber notificações sobre as mudanças nas ações e como as notificações são enviadas quando ocorrem alterações.

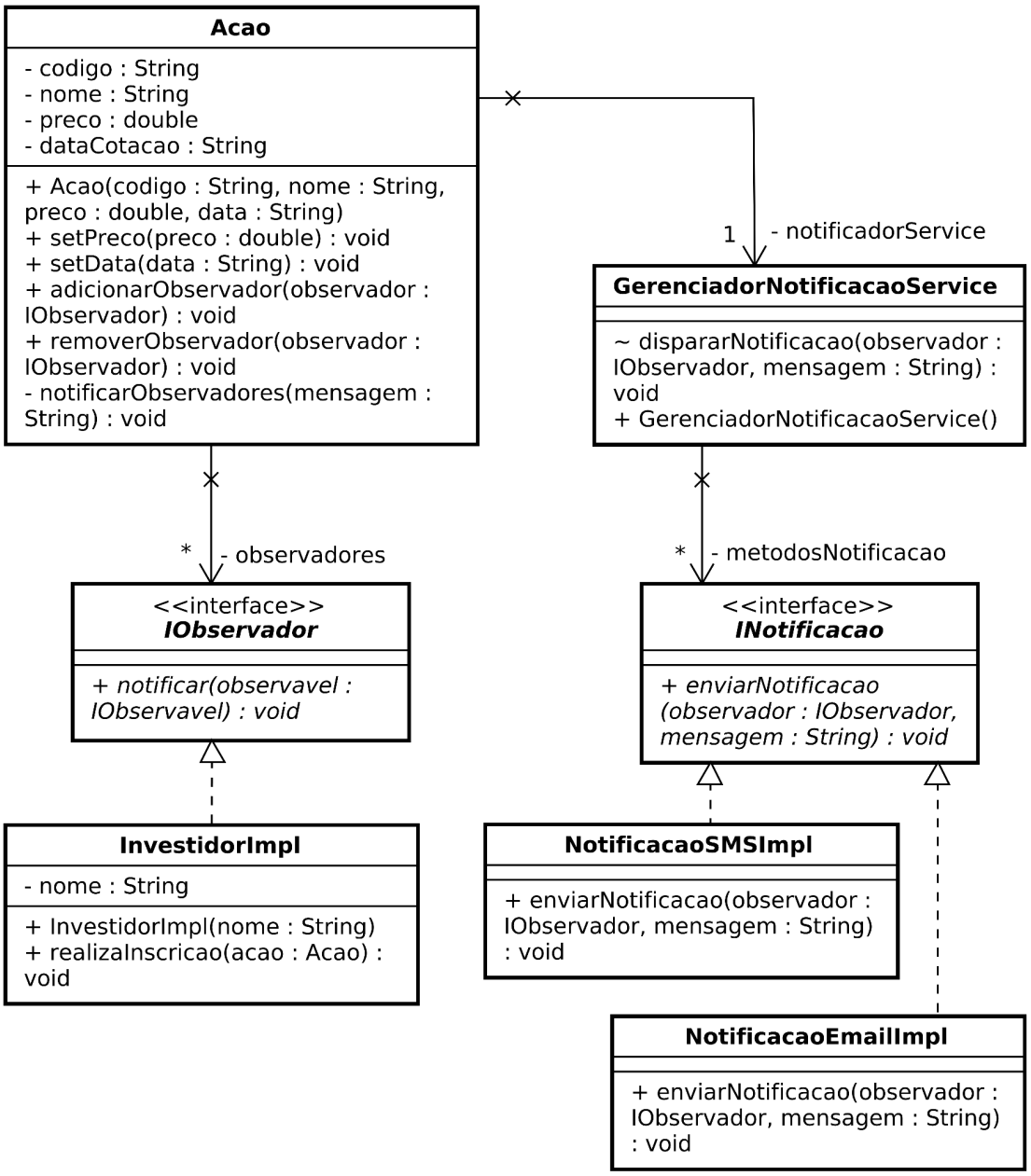
Segue o código Java da classe Principal:

```
public class Principal {
    public static void main(String[] args) {
        Acao acao = new Acao("AAPL", "Apple Inc.", 150.0, "2024-05-14");
        InvestidorImpl investidor1 = new InvestidorImpl("Investidor 1");
        InvestidorImpl investidor2 = new InvestidorImpl("Investidor 2");

        investidor1.realizaInscricao(acao);
        investidor2.realizaInscricao(acao);

        acao.setPreco(160.0);
        acao.setPreco(170.0);

        acao.setData("2024-05-15");
```

```
  }
}
```

Este código cria uma instância de uma ação (AAPL - Apple Inc.) e duas instâncias de investidores (Investidor 1 e Investidor 2). Em seguida, os investidores se inscrevem para receber notificações sobre a ação usando o método realizaInscricao. Após isso, são feitas algumas simulações de alterações no preço e na data de cotação da ação. Cada vez que uma alteração ocorre, os investidores recebem notificações correspondentes. Este exemplo demonstra como o sistema permite que investidores se inscrevam para receber notificações sobre ações e como as notificações são enviadas quando ocorrem mudanças nas ações.

Veja abaixo o diagrama de classes UML desta solução.



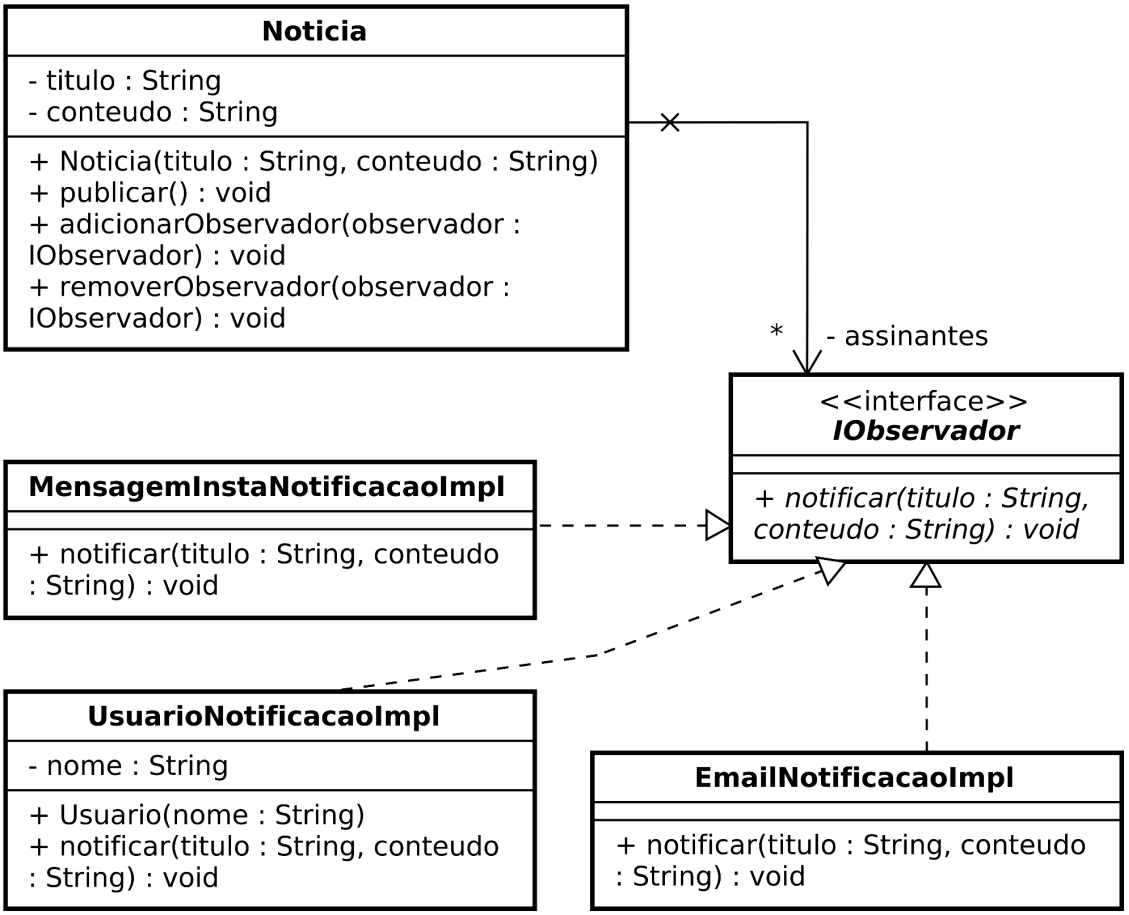
Solução do Exercício 4

Exercício resolvido no caderno.

Solução do Exercício 5

Para solucionar o problema de notificação de novas notícias aos assinantes, foi adotada uma abordagem que segue os princípios SOLID, visando criar um sistema flexível e extensível.

Observe abaixo o diagrama de classes com a solução proposta.



Inicialmente, foi definida uma interface IObservador que estabelece o contrato para os observadores que recebem notificações sobre as notícias. Essa interface possui um método notificar que recebe o título e o conteúdo da notícia como parâmetros.

Confira o código Java a seguir.

```
public interface IObservador {
  void notificar(String titulo, String conteudo);
}
```

A classe Noticia representa uma notícia a ser publicada. Ela mantém o título, o conteúdo e uma lista de observadores (assinantes) interessados em receber notificações. A classe oferece métodos para adicionar e remover observadores, além do método publicar, que notifica todos os observadores quando uma nova notícia é publicada.

```
public class Noticia {
  private String titulo;
```

```
  private String conteudo;
  private List<IObservador> assinantes;

  public Noticia(String titulo, String conteudo) {
    this.titulo = titulo;
    this.conteudo = conteudo;
    this.assinantes = new ArrayList<>();
  }

  public void publicar() {
    System.out.println("Nova notícia publicada: " + this.titulo);
    System.out.println("Conteúdo: " + this.conteudo);

    for (IObservador assinante : assinantes) {
      assinante.notificar(this.titulo, this.conteudo);
    }
  }

  public void adicionarObservador(IObservador observador) {
    assinantes.add(observador);
  }

  public void removerObservador(IObservador observador) {
    assinantes.remove(observador);
  }
}
```

Para cada tipo de observador, foi implementada uma classe correspondente que implementa a interface IObservador. A implementação UsuarioNotificacaoImpl notifica um usuário por meio de uma mensagem no console, enquanto EmailNotificacaoImpl e MensagemInstaNotificacaoImpl notificam por e-mail e mensagem instantânea, respectivamente.

```
public class UsuarioNotificacaoImpl implements IObservador {
  private String nome;

  public Usuario(String nome) {
```

```
        this.nome = nome;
    }

    @Override
    public void notificar(String titulo, String conteudo) {
        System.out.println("Usuário " + nome + " recebeu notificação: " + titulo);
    }
}

public class EmailNotificacaoImpl implements IObservador {
    @Override
    public void notificar(String titulo, String conteudo) {
        // Lógica para enviar email
        System.out.println("E-mail enviado com a notícia: " + titulo);
    }
}

public class MensagemInstaNotificacaoImpl implements IObservador {
    @Override
    public void notificar(String titulo, String conteudo) {
        // Lógica para enviar mensagem instantânea
        System.out.println("Mensagem instantânea enviada com a notícia: " + titulo);
    }
}
```

Já na classe Principal, é criada uma instância da Noticia com um título e conteúdo específicos. Em seguida, são instanciados os observadores para usuário, e-mail e mensagem instantânea, adicionados à lista de observadores da notícia e, finalmente, a notícia é publicada.

```
public class Principal {
    public static void main(String[] args) {
        Noticia noticia = new Noticia("Título da Notícia", "Conteúdo da notícia");
```

```
        IObservador notificacaousuario = new UsuarioNotificacaoImpl("João");
        IObservador notificacaoEmail = new EmailNotificacaoImpl();
        IObservador notificacaoMensagemInstantanea = new
MensagemInstaNotificacaoImpl();
        noticia.adicionarObservador(notificacaousuario);
        noticia.adicionarObservador(notificacaoEmail);
        noticia.adicionarObservador(notificacaoMensagemInstantanea);

        noticia.publicar();
    }
}
```

Solução do Exercício 6

Para solucionar o problema de cálculo do tempo de viagem com base no método de transporte selecionado, foi adotada uma abordagem que utiliza o padrão de projeto Strategy. Este padrão permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis. A escolha do algoritmo pode ser feita dinamicamente, com base no contexto ou nas preferências do usuário.

Inicialmente, foi criada uma interface ICalculoTempoViagemStrategy que define o contrato para os diferentes métodos de cálculo do tempo de viagem. Esta interface possui um método calcularTempoViagem, que recebe a distância a ser percorrida como parâmetro e retorna o tempo estimado de viagem.

```
public interface ICalculoTempoViagemStrategy {
    double calcularTempoViagem(double distancia);
}
```

Em seguida, foram implementadas as diferentes estratégias de cálculo do tempo de viagem para cada método de transporte. Cada classe concreta implementa a interface CalculoTempoViagem e fornece a lógica específica para o cálculo do tempo de viagem.

```
public class CaminhadaStrategyImpl implements ICalculoTempoViagemStrategy {
  @Override
  public double calcularTempoViagem(double distancia) {
    // Lógica para calcular o tempo de viagem a pé
    return distancia / 5.0;
  }
}

public class BicicletaStrategyImpl implements ICalculoTempoViagemStrategy {
  @Override
  public double calcularTempoViagem(double distancia) {
    // Lógica para calcular o tempo de viagem de bicicleta
    return distancia / 15.0;
  }
}

public class CarroStrategyImpl implements ICalculoTempoViagemStrategy {
  @Override
  public double calcularTempoViagem(double distancia) {
    // Lógica para calcular o tempo de viagem de carro
    return distancia / 60.0;
  }
}

public class OnibusStrategyImpl implements ICalculoTempoViagemStrategy {
  @Override
  public double calcularTempoViagem(double distancia) {
    // Lógica para calcular o tempo de viagem de ônibus
    return distancia / 30.0;
  }
}
```

A classe CalculadoraTempoViagem desempenha o papel de contexto, ou seja, é responsável por receber a estratégia selecionada e executar o cálculo do tempo de viagem com base nessa estratégia.

```
public class CalculadoraTempoViagem {
  private ICalculoTempoViagemStrategy estrategia;

  public void setEstrategia(ICalculoTempoViagemStrategy estrategia) {
    this.estrategia = estrategia;
  }

  public double calcularTempoViagem(double distancia) {
    return estrategia.calcularTempoViagem(distancia);
  }
}
```

Por fim, para demonstrar o uso da solução, criamos uma classe Principal com um método main. Neste método, instanciamos a CalculadoraTempoViagem e selecionamos uma estratégia de cálculo do tempo de viagem. Em seguida, chamamos o método calcularTempoViagem, passando a distância desejada como parâmetro.

```
public class Principal {
  public static void main(String[] args) {
    CalculadoraTempoViagem calculadora = new CalculadoraTempoViagem();

    calculadora.setEstrategia(new CaminhadaStrategyImpl());

    double tempoViagem = calculadora.calcularTempoViagem(10.0);

    System.out.println("Tempo de viagem a pé: " + tempoViagem + " horas");

    calculadora.setEstrategia(new BicicletaStrategyImpl());
    tempoViagem = calculadora.calcularTempoViagem(10.0);
    System.out.println("Tempo de viagem de bicicleta: " + tempoViagem + " horas");
```

```
        calculadora.setEstrategia(new CarroStrategyImpl());
        tempoViagem = calculadora.calcularTempoViagem(10.0);
        System.out.println("Tempo de viagem de carro: " + tempoViagem + " horas");

        calculadora.setEstrategia(new OnibusStrategyImpl());
        tempoViagem = calculadora.calcularTempoViagem(10.0);
        System.out.println("Tempo de viagem de ônibus: " + tempoViagem + " horas");
    }
}
```

Este código demonstra como podemos utilizar a CalculadoraTempoViagem para calcular o tempo de viagem utilizando diferentes estratégias de cálculo, sem a necessidade de modificar a classe CalculadoraTempoViagem além da alternância de estratégia em tempo de execução do sistema. Isso mantém o princípio Aberto/Fechado do padrão Strategy, permitindo que novas estratégias de cálculo do tempo de viagem sejam adicionadas facilmente no futuro. Cada nova estratégia pode ser implementada como uma nova classe que implementa a interface ICalculoTempoViagemStrategy.

Solução do Exercício 7

Exercício resolvido no caderno.

Solução do Exercício 8

Para resolver o problema de garantir que as configurações do sistema sejam consistentes em todo o código e evitar a criação desnecessária de múltiplas instâncias, será utilizado o padrão Singleton. Este padrão garante que apenas uma instância de uma classe seja criada e fornece um ponto de acesso global a essa instância.

Inicialmente, será criada uma classe ConfiguracaoSistema que implementa o padrão Singleton. Essa classe terá um método estático getInstance() que retorna a única instância da classe, e seu construtor será privado para evitar a criação de novas instâncias fora da própria classe.

```
public class ConfiguracaoSistema {
  private static ConfiguracaoSistema instance;
  private HashMap<String, String> configuracoes;

  private ConfiguracaoSistema() {
    this.configuracoes = new HashMap<>();
  }

  public static ConfiguracaoSistema getInstance() {
    if (instance == null) {
      instance = new ConfiguracaoSistema();
    }
    return instance;
  }

  public void setConfiguracao(String chave, String valor) {
    configuracoes.put(chave, valor);
  }

  public String getConfiguracao(String chave) {
    return configuracoes.get(chave);
  }
}
```

Agora, na classe Principal, será demonstrado o uso da instância única das configurações do sistema, onde exemplifica a utilização dos métodos setConfiguracao e getConfiguracao:

```
public class Principal {
  public static void main(String[] args) {
    ConfiguracaoSistema configuracao = ConfiguracaoSistema.getInstance();

    configuracao.setConfiguracao("idioma", "português");
    configuracao.setConfiguracao("tema", "escuro");
```

```
        String idioma = configuracao.getConfiguracao("idioma");
        String tema = configuracao.getConfiguracao("tema");

        System.out.println("Idioma: " + idioma);
        System.out.println("Tema: " + tema);
    }
}
```

Solução do Exercício 9

No código da implementação atual, a classe ValidacaoTransacao possui o método validar() que acessa diretamente os atributos dos objetos Criptoativo, Transacao, EnderecoOrigem, EnderecoDestino e RegraNegocio, violando a Lei de Demeter.

Para corrigir essa violação, é necessário encapsular a obtenção dos atributos dos objetos Criptoativo, Transacao, EnderecoOrigem, EnderecoDestino e RegraNegocio dentro das respectivas classes, criando métodos que retornem esses atributos de forma encapsulada. Em seguida, a classe ValidacaoTransacao deve utilizar esses métodos para obter os atributos dos objetos Criptoativo, Transacao, EnderecoOrigem, EnderecoDestino e RegraNegocio, em vez de acessá-los diretamente.

Abaixo está o código corrigido:

```
public class ValidacaoTransacao {
    private List<Transacao> transacoes;
    private ApiCriptoativo apiCriptoativo;
    private List<RegraNegocio> regrasNegocio;

    public ValidacaoTransacao(List<Transacao> transacoes, ApiCriptoativo apiCriptoativo,
List<RegraNegocio> regrasNegocio) {
        this.transacoes = transacoes;
        this.apiCriptoativo = apiCriptoativo;
        this.regrasNegocio = regrasNegocio;
```

```
    }

    public boolean validar() {
        for (Transacao transacao : transacoes) {
            Criptoativo criptoativo =
apiCriptoativo.obterCriptoativoPorId(transacao.getCriptoativoId());
            if (!criptoativo.ehValido()) {
                return false;
            }
            if (transacao.getSaldoOrigem() < transacao.getValor()) {
                return false;
            }
            if (!transacao.getEnderecoDestino().ehValido()) {
                return false;
            }
            for (RegraNegocio regraNegocio : regrasNegocio) {
                if (!regraNegocio.aplica(transacao)) {
                    return false;
                }
            }
        }
        return true;
    }
}

public class Criptoativo {
    private String id;
    private boolean valido;

    public Criptoativo(String id, boolean valido) {
        this.id = id;
        this.valido = valido;
```

```
    }

    public String getId() {
        return id;
    }

    public boolean ehValido() {
        return valido;
    }
}

public class Transacao {
    private String id;
    private String criptoativoId;
    private double valor;
    private EnderecoOrigem enderecoOrigem;
    private EnderecoDestino enderecoDestino;

    public Transacao(String id, String criptoativoId, double valor,
EnderecoOrigem enderecoOrigem, EnderecoDestino enderecoDestino) {
        this.id = id;
        this.criptoativoId = criptoativoId;
        this.valor = valor;
        this.enderecoOrigem = enderecoOrigem;
        this.enderecoDestino = enderecoDestino;
    }

    public String getId() {
        return id;
    }

    public String getCriptoativoId() {
```

```
        return criptoativoId;
    }

    public double getValor() {
        return valor;
    }

    public double getSaldoOrigem() {
        return enderecoOrigem.getSaldo();
    }

    public EnderecoDestino getEnderecoDestino() {
        return enderecoDestino;
    }
}

public class EnderecoOrigem {
    private String endereco;
    private double saldo;

    public EnderecoOrigem(String endereco, double saldo) {
        this.endereco = endereco;
        this.saldo = saldo;
    }

    public String getEndereco() {
        return endereco;
    }

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

```
}

public class EnderecoDestino {
    private String endereco;
    private boolean valido;

    public EnderecoDestino(String endereco, boolean valido) {
        this.endereco = endereco;
        this.valido = valido;
    }

    public String getEndereco() {
        return endereco;
    }

    public boolean ehValido() {
        return valido;
    }
}

public interface ApiCriptoativo {
    Criptoativo obterCriptoativoPorId(String id);
}

public interface RegraNegocio {
    boolean aplica(Transacao transacao);
}
```

No código acima, as classes `Criptoativo`, `Transacao`, `EnderecoOrigem`, `EnderecoDestino` e `RegraNegocio` possuem métodos que retornam os atributos dos objetos de forma encapsulada. A classe `ValidacaoTransacao` utiliza esses métodos para obter os atributos dos objetos `Criptoativo`, `Transacao`, `EnderecoOrigem`, `EnderecoDestino` e `RegraNegocio`, em vez de acessá-los diretamente, corrigindo assim a violação da Lei de Demeter.

Solução do Exercício 10

Para solucionar a questão de gerenciamento de estados de um tocador de música, foi adotada uma abordagem que segue os princípios SOLID, visando criar um sistema flexível e extensível. A ideia principal é permitir que a música esteja em um dos três estados: tocando, pausado e parado, e que as transições entre esses estados sejam gerenciadas de forma clara e modular.

Inicialmente, foi criada a classe Musica, que representa a música e contém informações sobre seu estado atual e seu nome. O construtor da classe Musica inicializa o estado da música como "Parado" utilizando a classe ParadoState. Além disso, a classe Musica possui métodos para tocar, pausar e parar a música, delegando essas ações para o objeto de estado atual.

```
public class Musica {
  private MusicaEstado estado;
  private String nome;

  public Musica(String nome) {
     this.nome = nome;
     setEstado(new ParadoState(this));
  }

  public void setEstado(MusicaEstado estado){
     this.estado = estado;
  }

  public String getNome(){
     return this.nome;
  }
}
```

A classe abstrata MusicaEstado define o comportamento padrão para os estados de uma música. Ela possui um atributo protegido musica e métodos para tocar, pausar e parar a

música, todos lançando uma exceção RuntimeErrorException. Isso garante que as subclasses especificarão o comportamento adequado para cada estado, se não for da sua atribuição, não haverá a subescrita, e se no momento da execução estes métodos forem chamado será lançado a exceção da classe base.

```
public abstract class MusicaEstado {
  protected Musica musica;

  public MusicaEstado(Musica musica){
     this.musica = musica;
  }

  public void tocar(Musica musica){
     throw new RuntimeErrorException("Não é possível tocar música.");
  }

  public void pausar(Musica musica){
     throw new RuntimeErrorException("Não é possível pausar música.");
  }

  public void parar(Musica musica){
     throw new RuntimeErrorException("Não é possível parar música.");
  }
}
```

A classe TocandoState estende MusicaEstado e representa o estado em que a música está tocando. No construtor, ela exibe uma mensagem indicando que a música está tocando. Os métodos pausar e parar mudam o estado da música para PausadoState e ParadoState, respectivamente.

```
public class TocandoState extends MusicaEstado {
  public TocandoState(Musica musica){
     super(musica);
     System.out.println("Música tocando.");
```

```
    }

    @Override
    public void pausar(Musica musica) {
        musica.setEstado(new PausadoState(musica));
    }

    @Override
    public void parar(Musica musica) {
        musica.setEstado(new ParadoState(musica));
    }
}
```

A classe PausadoState também estende MusicaEstado e representa o estado em que a música está pausada. No construtor, ela exibe uma mensagem indicando que a música está pausada. Os métodos tocar e parar mudam o estado da música para TocandoState e ParadoState, respectivamente.

```
public class PausadoState extends MusicaEstado {
    public PausadoState(Musica musica){
        super(musica);
        System.out.println("Música pausada.");
    }

    @Override
    public void tocar(Musica musica) {
        musica.setEstado(new TocandoState(musica));
    }

    @Override
    public void parar(Musica musica) {
        musica.setEstado(new ParadoState(musica));
    }
```

```
}
```

Por fim, a classe ParadoState estende MusicaEstado e representa o estado em que a música está parada. No construtor, ela exibe uma mensagem indicando que a música está parada. O método tocar muda o estado da música para TocandoState.

```
public class ParadoState extends MusicaEstado {
  public ParadoState(Musica musica){
    super(musica);
    System.out.println("Música parada.");
  }

  @Override
  public void tocar(Musica musica) {
    musica.setEstado(new TocandoState(musica));
  }
}
```

Para demonstrar o uso do código fornecido, criaremos uma classe Principal com um método main. Esta classe será responsável por simular a interação com a música, demonstrando como os estados mudam conforme as ações são executadas.

```
public class Principal {
  public static void main(String[] args) {
    try {
      Musica musica = new Musica("Minha Música Favorita");

      musica.tocar();
      musica.pausar();
      musica.tocar();
      musica.parar();
    } catch (Exception e) {
      System.out.println("ERRO: " + e.getMessage());
    }
```

```
    }
}
```

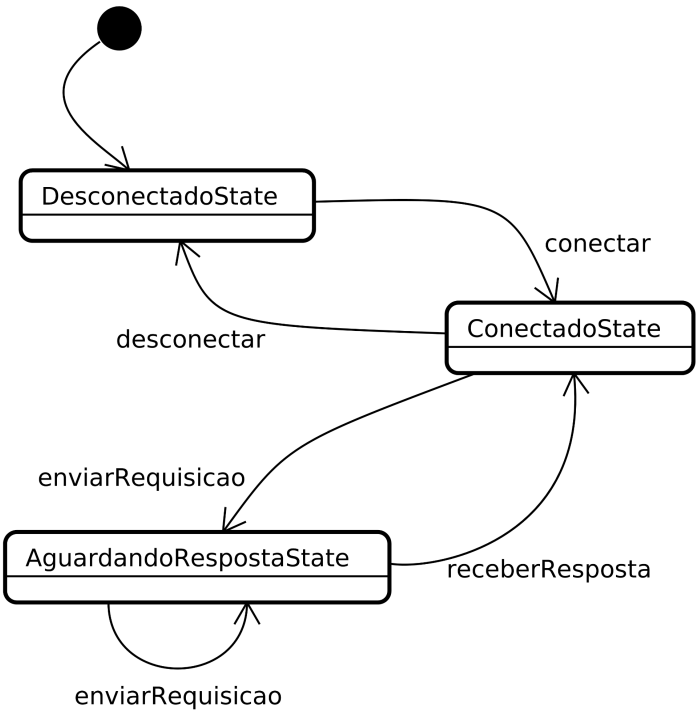
Esta solução demonstra como o sistema permite que uma música esteja em diferentes estados e como as ações disponíveis mudam de acordo com o estado atual. A implementação segue os princípios SOLID, especialmente o princípio de responsabilidade única (SRP) e o princípio aberto/fechado (OCP), garantindo que a adição de novos estados ou modificações nos estados existentes possam ser feitas sem alterar o código das classes que utilizam esses estados.

Solução do Exercício 11

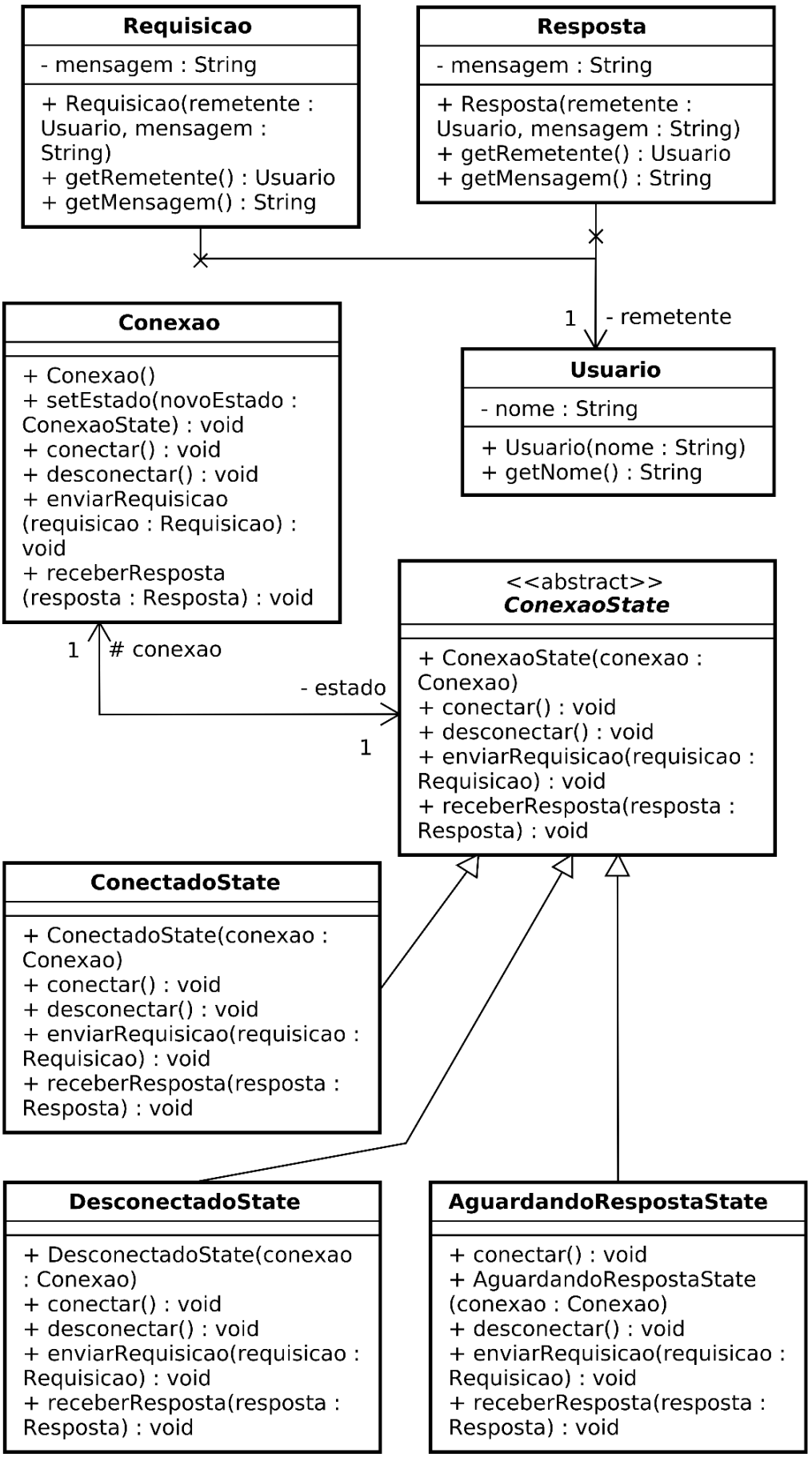
Exercício resolvido no caderno.

Solução do Exercício 12

Primeiramente, vamos entender melhor quais estados são permitidos por meio do diagrama de máquina de estado a seguir:



Analise o diagrama de classes a seguir com a solução proposta.


Requisicao
- mensagem : String
+ Requisicao(remetente : Usuario, mensagem : String)
+ getRemetente() : Usuario
+ getMensagem() : String
Resposta
- mensagem : String
+ Resposta(remetente : Usuario, mensagem : String)
+ getRemetente() : Usuario
+ getMensagem() : String
1
- remetente
Usuario
- nome : String
+ Usuario(nome : String)
+ getNome() : String
Conexao
+ Conexao()
+ setEstado(novoEstado : ConexaoState) : void
+ conectar() : void
+ desconectar() : void
+ enviarRequisicao (requisicao : Requisicao) : void
+ receberResposta (resposta : Resposta) : void
1
conexao
- estado
1
<<abstract>>
ConexaoState
+ ConexaoState(conexao : Conexao)
+ conectar() : void
+ desconectar() : void
+ enviarRequisicao(requisicao : Requisicao) : void
+ receberResposta(resposta : Resposta) : void
ConectadoState
+ ConectadoState(conexao : Conexao)
+ conectar() : void
+ desconectar() : void
+ enviarRequisicao(requisicao : Requisicao) : void
+ receberResposta(resposta : Resposta) : void
DesconectadoState
+ DesconectadoState(conexao : Conexao)
+ conectar() : void
+ desconectar() : void
+ enviarRequisicao(requisicao : Requisicao) : void
+ receberResposta(resposta : Resposta) : void
AguardandoRespostaState
+ conectar() : void
+ AguardandoRespostaState (conexao : Conexao)
+ desconectar() : void
+ enviarRequisicao(requisicao : Requisicao) : void
+ receberResposta(resposta : Resposta) : void

Inicialmente, iremos codificar a classe Usuario, que representará os usuários do sistema de mensagens. Cada usuário possui um nome que identifica exclusivamente o usuário.

```
public class Usuario {
  private String nome;

  public Usuario(String nome) {
    this.nome = nome;
  }

  public String getNome() {
    return nome;
  }
}
```

Em seguida, implementamos a classe Resposta, que tem como responsabilidade armazenar a resposta enviada por um usuário em relação a uma requisição. Cada resposta é composta pela mensagem da resposta e pelo remetente, que é o usuário que enviou a resposta.

```
public class Resposta {
  private String mensagem;
  private Usuario remetente;

  public Resposta(Usuario remetente, String mensagem) {
    this.remetente = remetente;
    this.mensagem = mensagem;
  }

  public Usuario getRemetente() {
    return remetente;
  }

  public String getMensagem() {
    return mensagem;
```

```
  }
}
```

Prosseguimos com a implementação da classe Requisicao, que é responsável por armazenar uma requisição enviada por um usuário. Cada requisição é composta pela mensagem da requisição e pelo remetente, que é o usuário que enviou a requisição.

```
public class Requisicao {
  private String mensagem;
  private Usuario remetente;

  public Requisicao(Usuario remetente, String mensagem) {
    this.remetente = remetente;
    this.mensagem = mensagem;
  }

  public Usuario getRemetente() {
    return remetente;
  }

  public String getMensagem() {
    return mensagem;
  }
}
```

Para a implementação do padrão State, criaremos a classe abstrata ConexaoState, que define os métodos que representam os estados possíveis de uma conexão.

```
public abstract class ConexaoState {
  protected Conexao conexao;

  public ConexaoState(Conexao conexao){
    this.conexao = conexao;
  }
```

```
  public void conectar(){
     throw new RuntimeErrorException("Não é possível conectar.");
  }
  public void desconectar(){
     throw new RuntimeErrorException("Não é possível desconectar.");
  }
  public void enviarRequisicao(Requisicao requisicao){
     throw new RuntimeErrorException("Não é possível enviar requisição.");
  }
  public void receberResposta(Resposta resposta){
     throw new RuntimeErrorException("Não é possível receber resposta.");
  }
}
```

Agora, implementaremos as classes de estado concretas. A classe DesconectadoState representa o estado em que a conexão está desconectada.

```
public class DesconectadoState extends ConexaoState {
  public DesconectadoState(Conexao conexao){
     super(conexao);
  }

  @Override
  public void conectar() {
     System.out.println("Conectando a conexão.");
     conexao.setEstado(new ConectadoState(conexao));
  }
}
```

A classe ConectadoState representa o estado em que a conexão está conectada e pronta para enviar requisições.

```
public class ConectadoState extends ConexaoState {
```

```
  public ConectadoState(Conexao conexao){
    super(conexao);
  }

  @Override
  public void desconectar() {
    System.out.println("Desconectando ...");
    conexao.setEstado(new DesconectadoState(conexao));
  }

  @Override
  public void enviarRequisicao(Requisicao requisicao) {
    System.out.println("Enviando requisição para o serviço de mensagens.");
    conexao.setEstado(new AguardandoRespostaState(conexao));
  }
}
```

Por fim, a classe AguardandoRespostaState representa o estado em que a conexão enviou uma requisição e aguarda uma resposta.

```
public class AguardandoRespostaState extends ConexaoState {
  public AguardandoRespostaState(Conexao conexao){
    super(conexao);
  }

  @Override
  public void enviarRequisicao(Requisicao requisicao) {
    System.out.println("Enviando novas requisições para o serviço de mensagens.");
  }

  @Override
  public void receberResposta(Resposta resposta) {
    System.out.println("Recebendo resposta do serviço de mensagens.");
```

```
        conexao.setEstado(new ConectadoState(conexao));
    }
}
```

Finalmente, implementaremos a classe Conexao, que gerencia o estado atual da conexão e delega as operações para o estado correspondente.

```
public class Conexao {
    private ConexaoState estado;

    public Conexao() {
        this.estado = new DesconectadoState(this);
    }

    public void setEstado(ConexaoState novoEstado) {
        this.estado = novoEstado;
    }

    public void conectar() {
        estado.conectar();
    }

    public void desconectar() {
        estado.desconectar();
    }

    public void enviarRequisicao(Requisicao requisicao) {
        estado.enviarRequisicao(requisicao);
    }

    public void receberResposta(Resposta resposta) {
        estado.receberResposta(resposta);
    }
```

```
}
```

Na classe Principal, vamos demonstrar o uso da implementação do padrão State para gerenciar uma conexão de mensagens. Nesta demonstração, criamos um usuário chamado "Alice" e uma conexão. Em seguida, conectamos a conexão e enviamos uma requisição, posteriormente recebemos a resposta.

```
public class Principal {
  public static void main(String[] args) {
     Usuario usuario = new Usuario("Alice");
     Conexao conexao = new Conexao();

     conexao.conectar();
     conexao.enviarRequisicao(new Requisicao(usuario, "Olá, tudo bem?"));
     conexao.receberResposta(new Resposta(usuario, "Olá, estou bem!"));

     conexao.desconectar();
  }
}
```

Solução do Exercício 13

O padrão de projeto usado aqui é o Chain of Responsibility (Cadeia de Responsabilidade). Ele permite que uma solicitação seja tratada por uma série de objetos, cada um deles podendo decidir se processa a solicitação ou a passa adiante para o próximo na cadeia.

Inicialmente, cria-se a classe Solicitacao, que representa uma solicitação feita pelo cliente. Ela contém uma descrição e um tipo.

```
public class Solicitacao {
  private String descricao;
  private String tipo;
  public Solicitacao(String descricao, String tipo) {
     this.descricao = descricao;
     this.tipo = tipo;
```

```
  }
  public String getDescricao() {
    return descricao;
  }
  public String getTipo() {
    return tipo;
  }
}
```

Em seguida, temos a classe abstrata Suporte, que define a estrutura da cadeia de responsabilidade. Ela possui um método para configurar o próximo elo na cadeia e métodos abstratos para tratar a solicitação e obter o tipo de solicitação.

```
public abstract class Suporte {
  protected Suporte proximoSuporte;
  public void setProximo(Suporte proximoSuporte) {
    this.proximoSuporte = proximoSuporte;
  }
  public abstract void tratarSolicitacao(Solicitacao solicitacao);
  public abstract String getTipoSolicitacao();
}
```

Depois, temos as classes concretas AtendenteSuporte, SupervisorSuporte e GerenteSuporte, que representam os diferentes níveis de suporte na cadeia. Cada uma implementa os métodos para tratar a solicitação e obter o tipo de solicitação correspondente.

```
public class AtendenteSuporte extends Suporte {
  @Override
  public void tratarSolicitacao(Solicitacao solicitacao) {
    if (solicitacao.getTipo().equals(getTipoSolicitacao())) {
      System.out.println("Atendente de Suporte tratando: " + solicitacao.getDescricao());
    } else if (proximoSuporte != null) {
      proximoSuporte.tratarSolicitacao(solicitacao);
    }
```

```
    }
    @Override
    public String getTipoSolicitacao() {
        return "DUVIDA_GERAL";
    }
}
public class SupervisorSuporte extends Suporte {
    @Override
    public void tratarSolicitacao(Solicitacao solicitacao) {
        if (solicitacao.getTipo().equals(getTipoSolicitacao())) {
            System.out.println("Especialista de Suporte tratando: " + solicitacao.getDescricao());
        } else if (proximoSuporte != null) {
            proximoSuporte.tratarSolicitacao(solicitacao);
        }
    }
    @Override
    public String getTipoSolicitacao() {
        return "RECLAMACAO_PRODUTO";
    }
}
public class GerenteSuporte extends Suporte {
    @Override
    public void tratarSolicitacao(Solicitacao solicitacao) {
        if (solicitacao.getTipo().equals(getTipoSolicitacao())) {
            System.out.println("Gerente de Suporte tratando: " + solicitacao.getDescricao());
        } else if (proximoSuporte != null) {
            proximoSuporte.tratarSolicitacao(solicitacao);
        }
    }
    @Override
    public String getTipoSolicitacao() {
        return "SOLICITACAO_DEVOLUCAO";
```

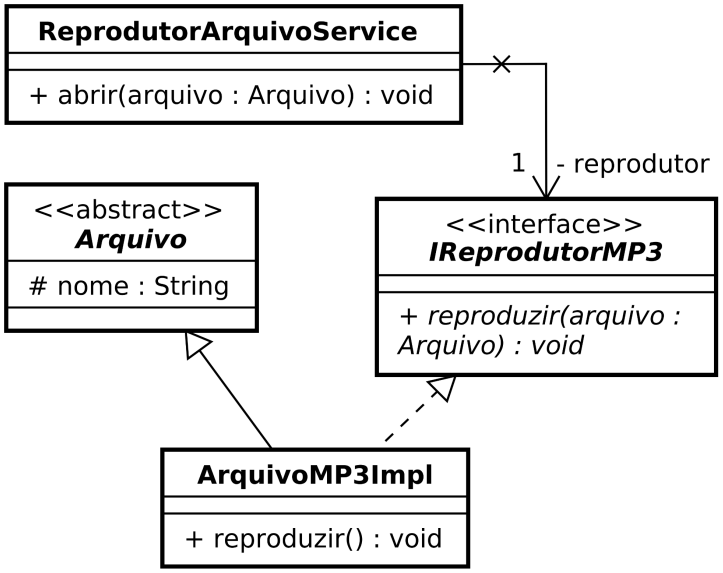
```
  }
}
```

Por fim, temos a classe Principal que configura a cadeia de responsabilidade definindo quem é o próximo elo na cadeia para cada objeto de suporte. Em seguida, cria solicitações e as encaminha para o primeiro objeto da cadeia (atendente). Cada objeto de suporte decide se trata a solicitação ou a passa adiante para o próximo na cadeia.

```
public class Principal {
  public static void main(String[] args) {
    // Cria os objetos de suporte
    Suporte atendente = new AtendenteSuporte();
    Suporte supervisor = new SupervisorSuporte();
    Suporte gerente = new GerenteSuporte();

    // Configura a cadeia de responsabilidade
    atendente.setProximo(supervisor);
    supervisor.setProximo(gerente);

    // Cria solicitações
    Solicitacao solicitacao1 =
new Solicitacao("Cliente com dúvida sobre produto", "DUVIDA_GERAL");
    Solicitacao solicitacao2 =
new Solicitacao("Cliente insatisfeito com a qualidade do produto", "RECLAMACAO_PRODUTO");
    Solicitacao solicitacao3 =
new Solicitacao("Cliente quer devolver produto", "SOLICITACAO_DEVOLUCAO");

    // Trata as solicitações
    atendente.tratarSolicitacao(solicitacao1);
    atendente.tratarSolicitacao(solicitacao2);
    atendente.tratarSolicitacao(solicitacao3);
  }
}
```

Solução do Exercício 14

Observa-se abaixo o diagrama de classes com a implementação atual.



Inicialmente, imagina-se que na implementação atual, a interface IReprodutorMP3 define o contrato para reproduzir arquivo MP3. Ela possui um único método, reproduzir, que recebe uma instância da classe Arquivo como parâmetro.

A classe Arquivo é uma classe abstrata que define como atributo o nome arquivo. Essa propriedade é protegida, permitindo que as subclasses acessem e modifiquem esse valor.

A implementação concreta, ArquivoMP3Impl, é uma subclasse de Arquivo que implementa a interface IReprodutorMP3.

A classe ReprodutorArquivoService desempenha o papel de gerenciar a reprodução de arquvios. Ela possui uma dependência da interface IReprodutorMP3, permitindo que qualquer implementação dessa interface possa ser utilizada. O método abrir recebe uma instância de Arquivo e delega a reprodução à implementação do reprodutor MP3, chamando o método reproduzir.

Dado contexto para resolver a questão, utilizaremos o padrão de projetos Adapter, uma vez que o enunciado deixa explícito que não deve ser alterada a interface original. A descrição e o código Java completo está a seguir.

Primeiramente, definimos a interface original IReprodutorMP3, que estabelece o contrato para reproduzir músicas no formato MP3.

```
public interface IReprodutorMP3{
  public void reproduzir(Arquivo arquivo);
}
```

Em seguida, criamos uma classe abstrata Arquivo que será a base para diferentes tipos de arquivos de música.

```
public abstract class Arquivo{
  protected String nome;
}
```

Implementamos a classe ArquivoMP3Impl, que estende Arquivo e implementa IReprodutorMP3, permitindo reproduzir arquivos MP3.

```
public class ArquivoMP3Impl extends Arquivo implements IReprodutorMP3{
  @Override
  public void reproduzir(){
     System.out.println("reproduzindo MP3");
  }
}
```

Implementamos as classes ArquivoMP4 e ArquivoVLC, que estendem Arquivo e possuem seus próprios métodos de reprodução.

```
public class ArquivoMP4 extends Arquivo{
  @Override
  public void reproduzir(){
     System.out.println("reproduzindo MP4");
  }
}
```

```
public class ArquivoVLC extends Arquivo{
  @Override
  public void reproduzir(){
     System.out.println("reproduzindo VLC");
  }
}
```

Para adaptar os formatos MP4 e VLC ao formato MP3, implementamos os adaptadores ReprodutorMP4Adapter e ReprodutorVLCAdapter, que implementam IReprodutorMP3.

```
public class ReprodutorMP4Adapter implements IReprodutorMP3{
  @Override
  public void reproduzir(Arquivo arquivo){
     Arquivo arquivoAdaptdao = adaptador(arquivo);
     //lógica para reproduzir
  }

  private Arquivo adaptador(Arquivo arquivo){
     //lógica para adaptação
  }
}

public class ReprodutorVLCAdapter implements IReprodutorMP3{
  @Override
  public void reproduzir(Arquivo arquivo){
     Arquivo arquivoAdaptdao = adaptador(arquivo);
     //lógica para reproduzir
  }

  private Arquivo adaptador(Arquivo arquivo){
     //lógica para adaptação
  }
```

```
}
```

Criamos a classe ReprodutorArquivoService que utiliza um IReprodutorMP3 para abrir arquivos, permitindo a definição dinâmica do reprodutor a ser usado.

```
public class ReprodutorArquivoService {
  private IReprodutorMP3 reprodutor;

  public void abrir(Arquivo arquivo){
     reprodutor.reproduzir(arquivo);
  }

  public void setReprodutor(IReprodutorMP3 reprodutor){
     this.reprodutor = reprodutor;
  }
}
```

Por fim, criamos a classe Principal para demonstrar a utilização do sistema de reprodução de músicas com suporte a arquivos MP3, MP4 e VLC.

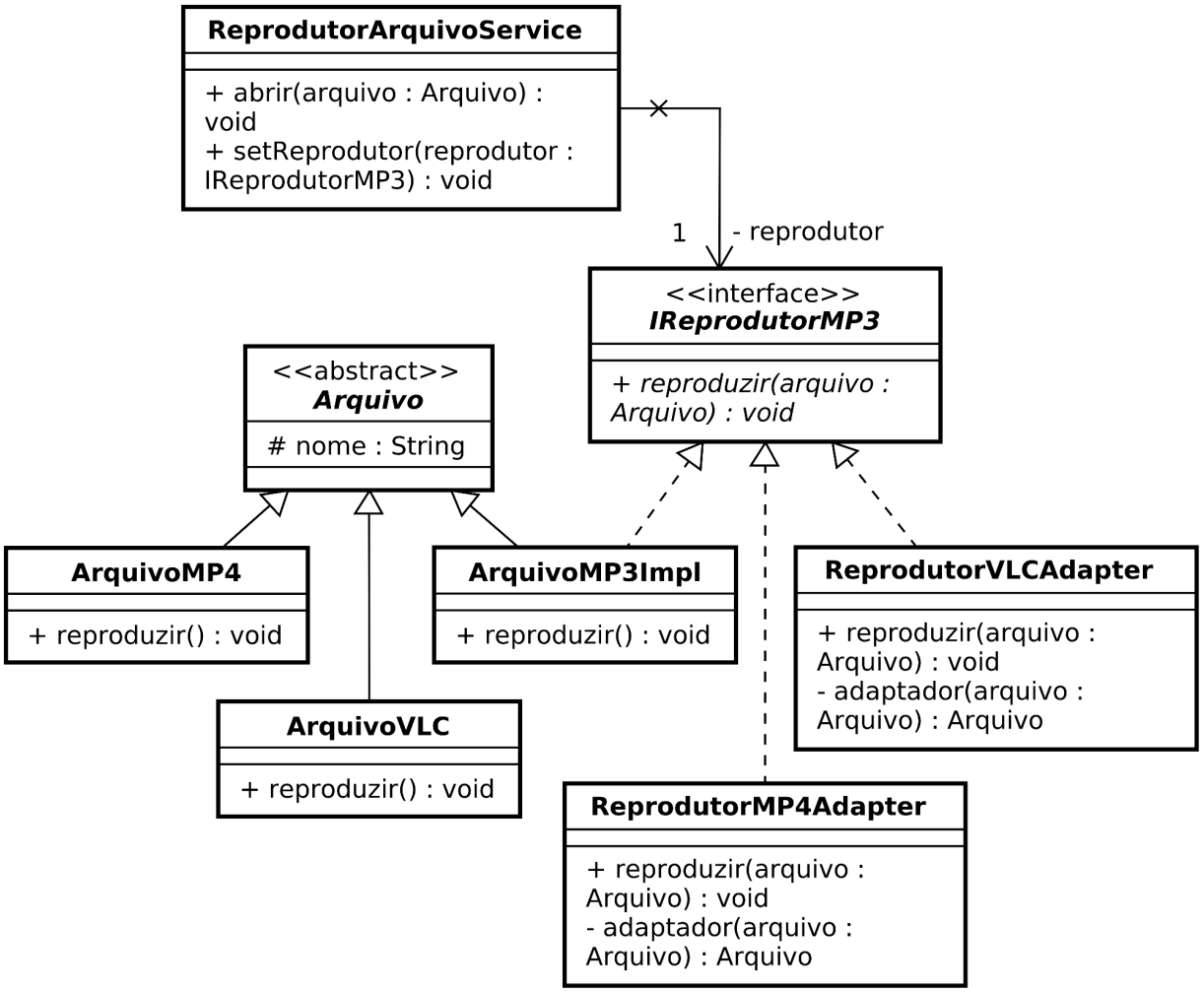
```
public class Principal {
  public static void main(String[] args) {
     ReprodutorArquivoService service = new ReprodutorArquivoService();

     Arquivo mp3 = new ArquivoMP3Impl();
     Arquivo mp4 = new ArquivoMP4();
     Arquivo vlc = new ArquivoVLC();

     service.setReprodutor(new ArquivoMP3Impl());
     service.abrir(mp3);

     service.setReprodutor(new ReprodutorMP4Adapter());
     service.abrir(mp4);
```

```
        service.setReprodutor(new ReprodutorVLCAdapter());
        service.abrir(vlc);
    }
}
```

Espera-se como saída no console as seguintes informações.

```
reproduzindo MP3
reproduzindo MP4
reproduzindo VLC
```

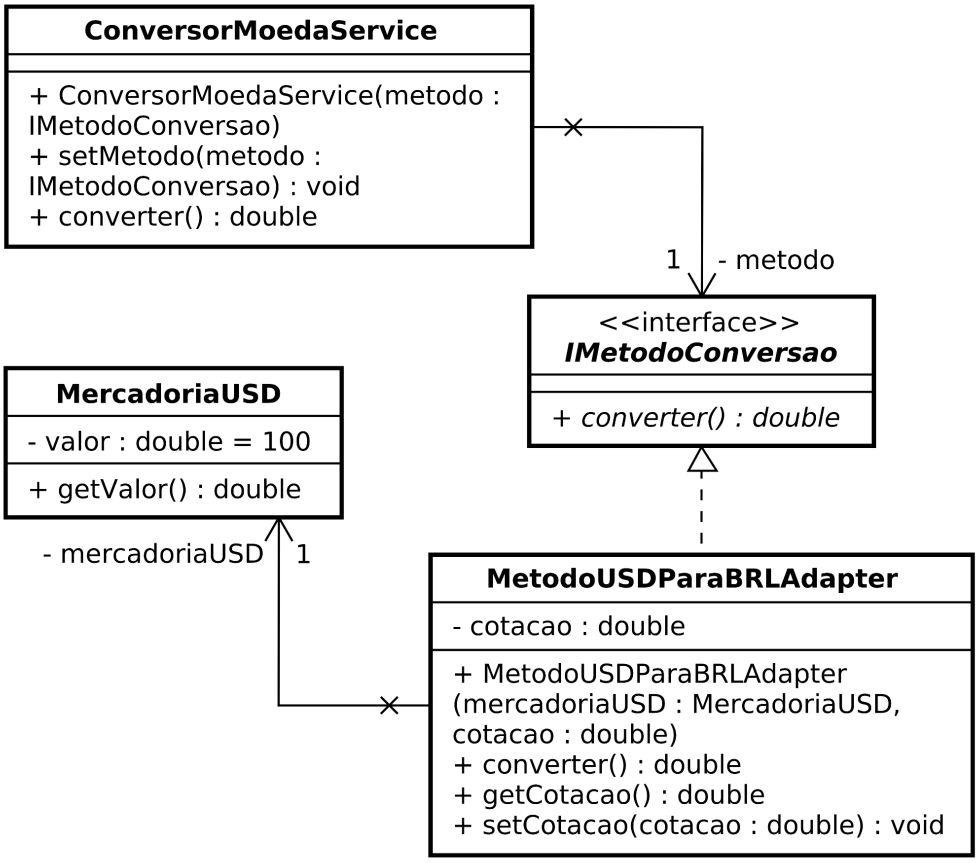
O diagrama de classes com a solução proposta está a seguir:



Solução do Exercício 15

Para fins de exercício, utilizaremos uma classe MercadoriaUSD para representar a mercadoria em dólares. Em um sistema real, esta classe poderia ser substituída por uma API externa que fornece os valores em USD.

Observe o diagrama de classes abaixo e em seguida a codificação em Java com textos explicativos.



Inicialmente, definimos a classe MercadoriaUSD que representa uma mercadoria com um valor fixo em dólares. Esta classe possui um atributo valor que armazena o valor da mercadoria que neste exercício inicializamos com 100, e um método getValor que retorna este valor.

```
public class MercadoriaUSD{
  private double valor = 100;

  public double getValor() {
     return valor;
```

```
  }
}
```

A interface IMetodoConversao define o contrato para os métodos de conversão de moedas. Ela declara um único método converter que deverá ser implementado pelas classes que fornecem diferentes métodos de conversão.

```
public interface IMetodoConversao {
  public double converter();
}
```

A classe ConversorMoedaService é responsável por realizar a conversão de moeda utilizando um objeto que implementa a interface IMetodoConversao. A classe possui um atributo metodo, um construtor que inicializa este atributo e métodos para definir e utilizar o método de conversão.

```
public class ConversorMoedaService {
  private IMetodoConversao metodo;

  public ConversorMoedaService(IMetodoConversao metodo) {
    this.metodo = metodo;
  }

  public void setMetodo(IMetodoConversao metodo) {
    this.metodo = metodo;
  }

  public double converter(){
    return metodo.converter();
  }
}
```

A classe MetodoUSDParaBRLAdapter adapta a mercadoria em USD para BRL aplicando uma taxa de câmbio. Ela implementa a interface IMetodoConversao, permitindo que seja utilizada pelo ConversorMoedaService.

```
public class MetodoUSDParaBRLAdapter implements IMetodoConversao{
```

```
  private MercadoriaUSD mercadoriaUSD;
  private double cotacao;

  public MetodoUSDParaBRLAdapter(MercadoriaUSD mercadoriaUSD, double cotacao) {
     this.mercadoriaUSD = mercadoriaUSD;
     this.cotacao = cotacao;
  }

  @Override
  public double converter(){
     return mercadoriaUSD.getValor() * cotacao;
  }

  public double getCotacao() {
     return cotacao;
  }

  public void setCotacao(double cotacao) {
     this.cotacao = cotacao;
  }
}
```

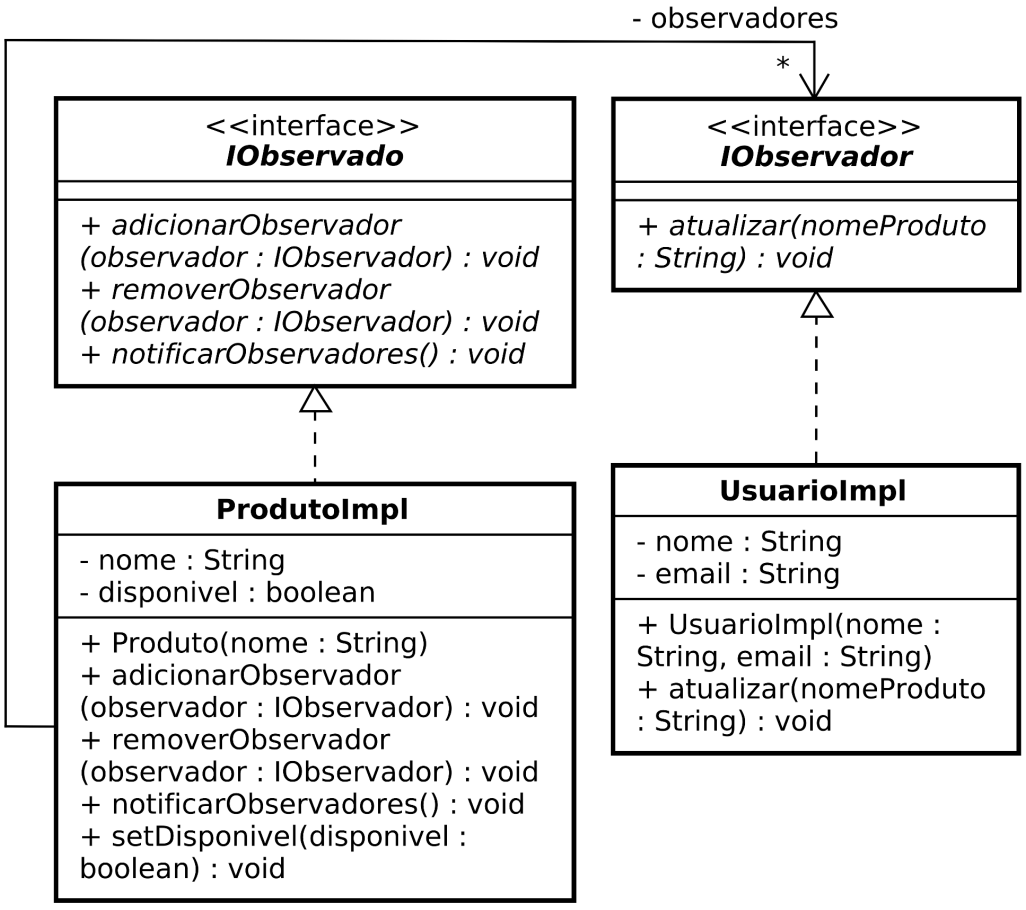
Por fim, é criada a classe Principal, responsável por demonstrar o uso do sistema de conversão de moedas. Ela cria uma instância de MercadoriaUSD para representar a mercadoria em dólares. Em seguida, cria um adaptador MetodoUSDParaBRLAdapter, passando a mercadoria e uma taxa de câmbio de 5 reais por dólar. Este adaptador é então utilizado para inicializar um ConversorMoedaService. O método converter do serviço é chamado para realizar a conversão, e o valor convertido é impresso no console, juntamente com o valor original em dólares.

```
public class Principal {
  public static void main(String[] args) {
     MercadoriaUSD mercadoriaUSD = new MercadoriaUSD();
     IMetodoConversao adaptador = new MetodoUSDParaBRLAdapter(mercadoriaUSD, 5);
```

```
        ConversorMoedaService service = new ConversorMoedaService(adaptador);

        double valorConvertido = service.converter();

        System.out.print("\nUSD $ " + mercadoriaUSD.getValor() + "\nBRL $ " + valorConvertido);
    }
}
```

Solução do Exercício 16

Para implementar o padrão Observer em um sistema de e-commerce, precisamos definir uma interface para os observadores e uma classe concreta para representar os usuários que serão notificados sobre a disponibilidade dos produtos. Analise o diagrama de classes abaixo:



Primeiramente, codificamos a interface IObservador, veja:

```
public interface IObservador {
  void atualizar(String nomeProduto);
}
```

Agora vamos implementar a classe UsuarioImpl que será o observador no padrão Observer.

```
public class UsuarioImpl implements IObservador {
  private String nome;
  private String email;

  public UsuarioImpl(String nome, String email) {
     this.nome = nome;
     this.email = email;
  }

  @Override
  public void atualizar(String nomeProduto) {
      System.out.println("Olá " + nome + ", o produto " + nomeProduto + " está disponível novamente no estoque!");
  }
}
```

Em seguida, definir a interface IObservado, que será implementada pela classe ProdutoImpl para gerenciar os observadores e notificá-los sobre mudanças de estado.

```
public interface IObservado {
  void adicionarObservador(IObservador observador);
  void removerObservador(IObservador observador);
  void notificarObservadores();
}
```

Agora vamos implementar a classe ProdutoImpl, que atua como o sujeito observado no padrão Observer. Esta classe é responsável por notificar os observadores (usuários) quando um produto específico estiver disponível novamente no estoque.

```
public class ProdutoImpl implements IObservado {
  private String nome;
  private boolean disponivel;
  private List<IObservador> observadores;

  public Produto(String nome) {
     this.nome = nome;
     this.disponivel = false;
     this.observadores = new ArrayList<>();
  }

  @Override
  public void adicionarObservador(IObservador observador) {
     observadores.add(observador);
  }

  @Override
  public void removerObservador(IObservador observador) {
     observadores.remove(observador);
  }

  @Override
  public void notificarObservadores() {
     for (IObservador observador : observadores) {
        observador.atualizar(this.nome);
     }
  }

  public void setDisponivel(boolean disponivel) {
     this.disponivel = disponivel;
     if (disponivel) {
        notificarObservadores();
```

```
    }
  }
}
```

Com a classe ProdutoImpl, temos a implementação do sujeito observado. Ele possui métodos para adicionar, remover e notificar observadores (usuários), bem como um método para atualizar o status de disponibilidade do produto.

```
public class Principal {
  public static void main(String[] args) {
    IObservado produto = new ProdutoImpl("PlayStation 5");

    IObservador usuario1 = new UsuarioImpl("João", "joao@example.com");
    IObservador usuario2 = new UsuarioImpl("Maria", "maria@example.com");

    produto.adicionarObservador(usuario1);
    produto.adicionarObservador(usuario2);

    System.out.println("Atualizando disponibilidade do produto...");
    produto.setDisponivel(true);
  }
}
```

Solução do Exercício 17

Para criar um sistema que utiliza o padrão de projeto Proxy para carregar dados de pedidos de forma eficiente, implementaremos uma estrutura que envolve várias classes. A ideia é carregar inicialmente apenas as informações básicas de um pedido e, somente quando necessário, carregar os detalhes dos itens do pedido. Este método é conhecido como lazy loading e é frequentemente utilizado em sistemas como o Hibernate. Vamos definir a estrutura do projeto a seguir.

Cerifique-se de criar os seguintes arquivos com o respectivo conteúdo na pasta resources do projeto Maven.

pedidos.csv

```
numero,data,nomeCliente,valorTotal
1001,2023-06-01,John Doe,450.00
1002,2023-06-02,Jane Smith,675.00
1003,2023-06-03,Bob Johnson,1200.00
```

itens.csv

```
numeroPedido,nomeItem,valorUnitario,quantidade,valorTotal
1001,Effective Java,45.00,2,90.00
1001,Clean Code,60.00,1,60.00
1001,Refactoring,75.00,4,300.00
1002,The Pragmatic Programmer,50.00,3,150.00
1002,Design Patterns,75.00,5,375.00
1003,Domain-Driven Design,85.00,2,170.00
1003,Continuous Delivery,80.00,5,400.00
1003,The Phoenix Project,105.00,6,630.00
```

Para resolver essa necessidade, começamos pela classe Pedido, que representa um pedido com informações básicas, como número, data, nome do cliente e valor total. Ela também possui uma lista de itens que será carregada sob demanda.

```
import java.util.Date;
import java.util.List;

public class Pedido {
  private int numero;
  private Date data;
  private String nomeCliente;
  private double valorTotal;
  private List<Item> itens;

  public Pedido(int numero, Date data, String nomeCliente, double valorTotal) {
    this.numero = numero;
    this.data = data;
```

```
    this.nomeCliente = nomeCliente;
    this.valorTotal = valorTotal;
}

public int getNumero() {
    return numero;
}

public Date getData() {
    return data;
}

public String getNomeCliente() {
    return nomeCliente;
}

public double getValorTotal() {
    return valorTotal;
}

public List<Item> getItens() {
    return itens;
}

public void setItens(List<Item> itens) {
    this.itens = itens;
}

@Override
public String toString() {
    return "Pedido{" +
            "numero=" + numero +
```

```
            ", data=" + data +
            ", nomeCliente='" + nomeCliente + '\'' +
            ", valorTotal=" + valorTotal +
            '}';
    }
}
```

Em seguida, definimos a classe Item, que representa um item individual de um pedido, com informações como número do pedido, nome do item, valor unitário, quantidade e valor total.

```
public class Item {
    private int numeroPedido;
    private String nomeItem;
    private double valorUnitario;
    private int quantidade;
    private double valorTotal;

    public Item(int numeroPedido, String nomeItem, double valorUnitario, int quantidade) {
        this.numeroPedido = numeroPedido;
        this.nomeItem = nomeItem;
        this.valorUnitario = valorUnitario;
        this.quantidade = quantidade;
        this.valorTotal = valorUnitario * quantidade;
    }

    public int getNumeroPedido() {
        return numeroPedido;
    }

    public String getNomeItem() {
        return nomeItem;
    }
```

```
  public double getValorUnitario() {
     return valorUnitario;
  }

  public int getQuantidade() {
     return quantidade;
  }

  public double getValorTotal() {
     return valorTotal;
  }

  @Override
  public String toString() {
     return "Item{" +
           "numeroPedido=" + numeroPedido +
           ", nomeItem='" + nomeItem + '\'' +
           ", valorUnitario=" + valorUnitario +
           ", quantidade=" + quantidade +
           ", valorTotal=" + valorTotal +
           '}';
  }
}
```

Para resolver o problema do carregamento dos pedidos, definimos a interface IPedidoRepository, que especifica as operações necessárias para buscar um pedido e seus itens.

```
import java.util.List;

public interface IPedidoRepository {
  Pedido buscarPedidoPorNumero(int numero);
  List<Item> buscarItensDoPedido(int numeroPedido);
```

}

Em seguida, implementamos a classe PedidoRepositoryImpl, que carrega os dados dos pedidos e dos itens a partir de arquivos CSV.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class PedidoRepositoryImpl implements IPedidoRepository {
  private static final String PEDIDOS_FILE = "resources/pedidos.csv";
  private static final String ITENS_FILE = "resources/itens.csv";

  @Override
  public Pedido buscarPedidoPorNumero(int numero) {
    try (BufferedReader br = new BufferedReader(new FileReader(PEDIDOS_FILE))) {
      String linha;
      br.readLine();

      while ((linha = br.readLine()) != null) {
        String[] dados = linha.split(",");
        int numeroPedido = Integer.parseInt(dados[0]);
        if (numeroPedido == numero) {
          Date data = new SimpleDateFormat("yyyy-MM-dd").parse(dados[1]);
          String nomeCliente = dados[2];
          double valorTotal = Double.parseDouble(dados[3]);
          return new Pedido(numeroPedido, data, nomeCliente, valorTotal);
        }
```

```
        }
    } catch (IOException | ParseException e) {
        e.printStackTrace();
    }
    return null;
}

@Override
public List<Item> buscarItensDoPedido(int numeroPedido) {
    List<Item> itens = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(ITENS_FILE))) {
        String linha;
        br.readLine();

        while ((linha = br.readLine()) != null) {
            String[] dados = linha.split(",");
            int numero = Integer.parseInt(dados[0]);
            if (numero == numeroPedido) {
                String nomeItem = dados[1];
                double valorUnitario = Double.parseDouble(dados[2]);
                int quantidade = Integer.parseInt(dados[3]);
                double valorTotal = Double.parseDouble(dados[4]);
                itens.add(new Item(numero, nomeItem, valorUnitario, quantidade));
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return itens;
}
}
```

Para implementar o padrão Proxy e realizar o lazy loading dos itens, criamos a classe PedidoRepositoryProxy, que encapsula a PedidoRepositoryImpl.

```
import java.util.List;

public class PedidoRepositoryProxy implements IPedidoRepository {
  private PedidoRepositoryImpl pedidoRepositoryImpl = new PedidoRepositoryImpl();

  @Override
  public Pedido buscarPedidoPorNumero(int numero) {
    Pedido pedido = pedidoRepositoryImpl.buscarPedidoPorNumero(numero);
    return new PedidoProxy(pedido);
  }

  @Override
  public List<Item> buscarItensDoPedido(int numeroPedido) {
    return pedidoRepositoryImpl.buscarItensDoPedido(numeroPedido);
  }

  private class PedidoProxy extends Pedido {
    private boolean itensCarregados = false;
    private Pedido pedidoReal;

    public PedidoProxy(Pedido pedidoReal) {
      super(pedidoReal.getNumero(), pedidoReal.getData(), pedidoReal.getNomeCliente(),
pedidoReal.getValorTotal());
      this.pedidoReal = pedidoReal;
    }

    @Override
    public List<Item> getItens() {
      if (!itensCarregados) {
```

```
            List<Item> itens =
pedidoRepositoryImpl.buscarItensDoPedido(pedidoReal.getNumero());
            pedidoReal.setItens(itens);
            itensCarregados = true;
        }
        return pedidoReal.getItens();
    }
  }
}
```

Por fim, a classe Main utiliza todas as classes definidas para demonstrar o funcionamento do sistema.

```
public class Main {
  public static void main(String[] args) {
    IPedidoRepository pedidoRepository = new PedidoRepositoryProxy();

    Pedido pedido = pedidoRepository.buscarPedidoPorNumero(1001);
    System.out.println(pedido);

    System.out.println("Itens do pedido:");
    for (Item item : pedido.getItens()) {
      System.out.println(item);
    }
  }
}
```

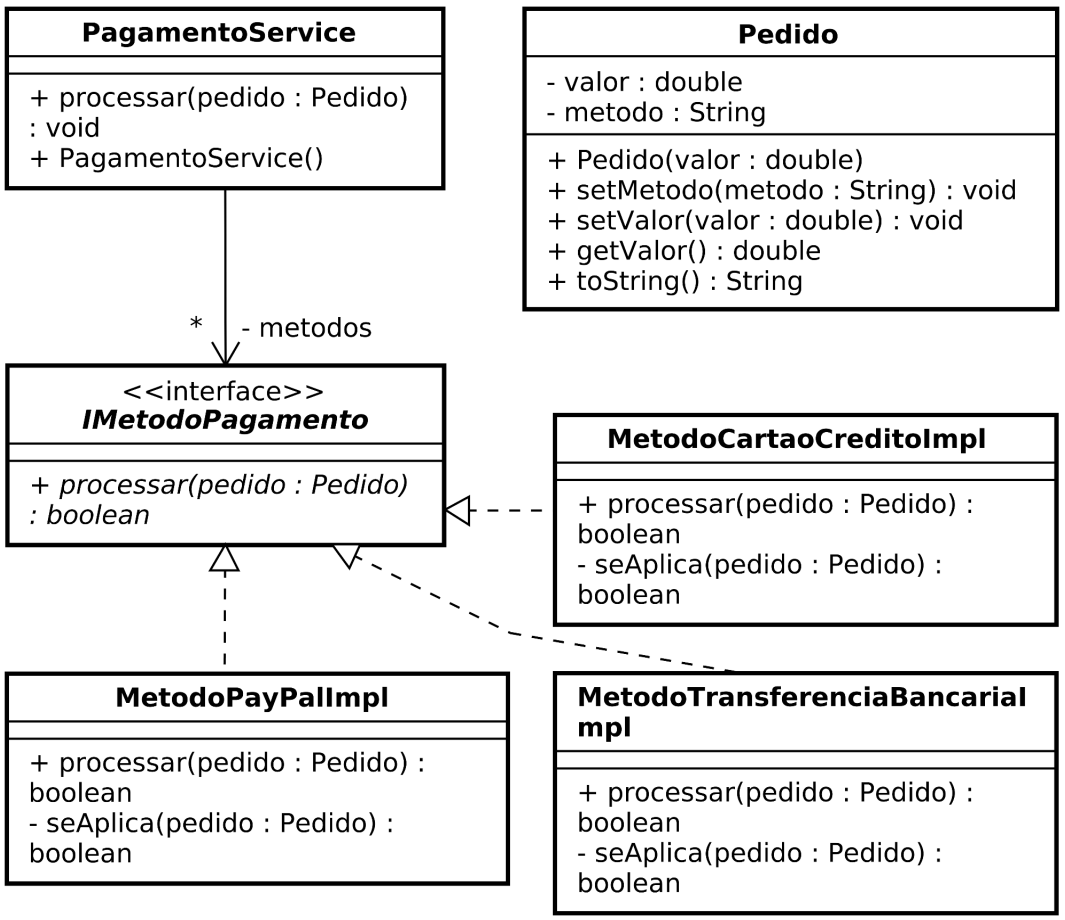
A saída esperada após a execução do método main é a seguinte:

Pedido{numero=1002, data=Fri Jun 02 00:00:00 BRT 2023, nomeCliente='Jane Smith', valorTotal=675.0}
Itens do pedido:
Item{numeroPedido=1002, nomeItem='The Pragmatic Programmer', valorUnitario=50.0, quantidade=3, valorTotal=150.0}

Item{numeroPedido=1002, nomeItem='Design Patterns', valorUnitario=75.0, quantidade=5, valorTotal=375.0}

Solução do Exercício 18

Analise a solução proposta no diagrama de classes abaixo.



Inicialmente definimos a classe entidade Pedido que possui como atributo o valor e ao fim do processamento de pagamento é realizado o atribução de uma string definindo qual método foi utilizado.

```
public class Pedido {
  private double valor;
  private String metodo;

  public Pedido(double valor) {
    this.valor = valor;
```

```
  }

  public void setMetodo(String metodo) {
    this.metodo = metodo;
  }

  public void setValor(double valor) {
    this.valor = valor;
  }

  public double getValor() {
    return valor;
  }

  @Override
  public String toString() {
    return "Pedido{" + "valor=" + valor + ", metodo=" + metodo + '}';
  }
}
```

Para solucionar a questão, utilizamos uma interface que define o contrato para todos os métodos de pagamento e classes concretas que implementam essa interface, cada uma com suas regras específicas de aplicação. Além disso, um serviço central de pagamento itera sobre os métodos disponíveis para processar o pagamento de um pedido. A seguir, apresentamos a implementação detalhada.

A interface IMetodoPagamento define um contrato para os métodos de pagamento. Esta interface declara um método processar que recebe um objeto Pedido e retorna um booleano indicando se o método de pagamento foi aplicado com sucesso.

```
public interface IMetodoPagamento {
  public boolean processar(Pedido pedido);
}
```

As classes MetodoCartaoCreditoImpl, MetodoPayPalImpl e MetodoTransferenciaBancariaImpl implementam a interface IMetodoPagamento. Estas classes possuem a responsabilidade de definir a lógica específica para processar o pagamento. O método processar verifica se as condições para aplicar este método são satisfeitas através do método seAplica, neste exercício a regra é conforme o valor do pedido. Se as condições forem atendidas, o método retorna verdadeiro e o pagamento é realizado. Caso contrário, retorna falso.

```
public class MetodoCartaoCreditoImpl implements IMetodoPagamento{
  @Override
  public boolean processar(Pedido pedido){
     if(seAplica(pedido)){
        pedido.setMetodo("Metodo Cartão de Credito");
        return true;
     }
     return false;
  }

  private boolean seAplica(Pedido pedido){
     return pedido != null && pedido.getValor() < 20;
  }
}

public class MetodoPayPalImpl implements IMetodoPagamento {
  @Override
  public boolean processar(Pedido pedido){
     if(seAplica(pedido)){
        pedido.setMetodo("Metodo PayPal");
        return true;
     }
     return false;
  }

  private boolean seAplica(Pedido pedido){
```

```
        return pedido != null && pedido.getValor() < 10;
    }
}

public class MetodoTransferenciaBancariaImpl implements IMetodoPagamento {
    @Override
    public boolean processar(Pedido pedido){
        if(seAplica(pedido)){
            pedido.setMetodo("Metodo Transferencia Bancaria");
            return true;
        }
        return false;
    }
    private boolean seAplica(Pedido pedido){
        return pedido != null && pedido.getValor() < 30;
    }
}
```

A classe PagamentoService gerencia os diferentes métodos de pagamento e processa os pedidos utilizando esses métodos. A classe armazena uma lista de métodos de pagamento e tenta processar o pedido com cada método até que um seja aplicado com sucesso.

```
public class PagamentoService {
    private ArrayList<IMetodoPagamento> metodos;

    public PagamentoService() {
        this.metodos = new ArrayList<>();
        this.metodos.add(new MetodoPayPalImpl());
        this.metodos.add(new MetodoCartaoCreditoImpl());
        this.metodos.add(new MetodoTransferenciaBancariaImpl());
    }

    public void processar(Pedido pedido){
```

```
        for(IMetodoPagamento elem: metodos){
            if(elem.processar(pedido))
                break;
        }
    }
}
```

Por fim, é criada a classe Principal para demonstrar o uso do sistema de pagamento. No método main, é criada uma instância de PagamentoService e uma instância de Pedido com valor de 20. O serviço de pagamento então processa o pedido, tentando aplicar os métodos de pagamento na ordem em que foram adicionados. O resultado, incluindo o método de pagamento aplicado ao pedido, é impresso no console.

```
public class Principal {
    public static void main(String[] args) {
        PagamentoService servicePagamento = new PagamentoService();
        Pedido pedido = new Pedido(20);

        servicePagamento.processar(pedido);

        System.out.print(pedido.toString());
    }
}
```

Solução do Exercício 19

O objetivo desta aplicação é gerenciar uma lista de produtos em estoque, permitindo realizar operações CRUD (Criar, Ler, Atualizar e Deletar) de forma separada da lógica de negócios e do acesso aos dados, facilitando assim a manutenção e evolução do sistema. Para isso, vamos desenvolver a aplicação utilizando padrões de projeto adequados.

Primeiro, definimos a classe Produto, que representa um produto no estoque. Esta classe possui atributos para armazenar o ID, nome, preço e quantidade do produto, e inclui construtores, getters e setters para esses atributos.

```
public class Produto {
  private int id;
  private String nome;
  private double preco;
  private int quantidade;

  // Construtores, getters e setters
  public Produto(int id, String nome, double preco, int quantidade) {
     this.id = id;
     this.nome = nome;
     this.preco = preco;
     this.quantidade = quantidade;
  }

  public int getId() {
     return id;
  }
  public void setId(int id) {
     this.id = id;
  }
  public String getNome() {
     return nome;
  }
  public void setNome(String nome) {
     this.nome = nome; }
  public double getPreco() {
     return preco;
  }
  public void setPreco(double preco) {
     this.preco = preco;
  }
  public int getQuantidade() {
```

```
        return quantidade;
    }
    public void setQuantidade(int quantidade) {
        this.quantidade = quantidade;
    }
}
```

Em seguida, criamos a interface IProdutoDAO, que define os métodos necessários para realizar operações CRUD. Esta interface permite a separação da lógica de acesso aos dados da lógica de negócios, facilitando mudanças na fonte de dados sem alterar a lógica de negócios conforme o requisito do enunciado.

```
public interface IProdutoDAO {
    void criar(Produto produto);
    Produto ler(int id);
    void atualizar(Produto produto);
    void deletar(int id);
}
```

Depois, implementamos a interface IProdutoDAO na classe ProdutoDAOImpl, que utiliza para fins de exercício, um HashMap para armazenar os produtos em memória. A implementação dos métodos criar, ler, atualizar e deletar permite manipular os dados dos produtos no estoque.

```
public class ProdutoDAOImpl implements IProdutoDAO {
    private Map<Integer, Produto> produtos = new HashMap<>();

    @Override
    public void criar(Produto produto) {
        produtos.put(produto.getId(), produto);
    }

    @Override
    public Produto ler(int id) {
        return produtos.get(id);
```

```
    }

    @Override
    public void atualizar(Produto produto) {
        produtos.put(produto.getId(), produto);
    }

    @Override
    public void deletar(int id) {
        produtos.remove(id);
    }
}
```

Para separar a lógica de negócios do acesso aos dados, criamos a classe ProdutoService, que utiliza a interface IProdutoDAO para realizar operações nos produtos. A ProdutoService contém métodos para criar, ler, atualizar e deletar produtos, delegando essas operações à implementação de IProdutoDAO.

```
public class ProdutoService {
    private IProdutoDAO produtoDAO;

    public ProdutoService(IProdutoDAO produtoDAO) {
        this.produtoDAO = produtoDAO;
    }

    public void criarProduto(Produto produto) {
        produtoDAO.criar(produto);
    }

    public Produto lerProduto(int id) {
        return produtoDAO.ler(id);
    }
```

```
  public void atualizarProduto(Produto produto) {
      produtoDAO.atualizar(produto);
  }

  public void deletarProduto(int id) {
      produtoDAO.deletar(id);
  }
}
```

Por fim, criamos a classe Principal para demonstrar o uso do sistema de gerenciamento de produtos. Na classe Principal, instanciamos um objeto ProdutoDAOImpl e um objeto ProdutoService para manipular os produtos. Adicionamos dois produtos, lemos um produto do estoque, atualizamos o preço de um produto e deletamos um produto, mostrando como o sistema gerencia as operações CRUD.

```
public class Principal {
  public static void main(String[] args) {
      IProdutoDAO produtoDAO = new ProdutoDAOImpl();

      ProdutoService produtoService = new ProdutoService(produtoDAO);

      Produto produto1 = new Produto(1, "Produto A", 10.0, 100);
      Produto produto2 = new Produto(2, "Produto B", 20.0, 200);

      produtoService.criarProduto(produto1);
      produtoService.criarProduto(produto2);

      Produto p = produtoService.lerProduto(1);
      System.out.println("Produto lido: " + p.getNome() + ", Preço: " + p.getPreco());

      produto1.setPreco(12.0);
      produtoService.atualizarProduto(produto1);
      p = produtoService.lerProduto(1);
```

```
        System.out.println("Produto atualizado: " + p.getNome() + ", Preço: " + p.getPreco());

        produtoService.deletarProduto(2);
        System.out.println("Produto 2 deletado");
    }
}
```

Solução do Exercício 20

O objetivo desta aplicação é gerenciar pedidos, itens e produtos em um sistema de gerenciamento. Cada pedido contém itens que se referem a produtos específicos. A aplicação deve permitir operações CRUD (Criar, Ler, Atualizar e Deletar) nos dados dos pedidos, itens e produtos.

Primeiramente, vamos definir a classe Produto, que representa um produto no estoque. Esta classe contém atributos para armazenar o ID, nome, preço e quantidade do produto. Também inclui construtores, getters e setters para esses atributos.

```
public class Produto {
    private int id;
    private String nome;
    private double preco;
    private int quantidade;

    // Construtores, getters e setters
    public Produto(int id, String nome, double preco, int quantidade) {
        this.id = id;
        this.nome = nome;
        this.preco = preco;
        this.quantidade = quantidade;
    }
    // Getters e setters
}
```

Em seguida, criamos a interface IProdutoDAO, que define os métodos necessários para realizar operações CRUD nos produtos. Isso separa a lógica de acesso aos dados da lógica de negócios, facilitando possíveis mudanças na fonte de dados no futuro.

```
public interface IProdutoDAO {
  void criar(Produto produto);
  Produto ler(int id);
  void atualizar(Produto produto);
  void deletar(int id);
}
```

Depois, implementamos a interface IProdutoDAO na classe ProdutoDAOImpl. Nesta implementação, utilizamos um HashMap para armazenar os produtos em memória. Os métodos criar, ler, atualizar e deletar permitem manipular os dados dos produtos no estoque.

```
import java.util.HashMap;
import java.util.Map;

public class ProdutoDAOImpl implements IProdutoDAO {
  private Map<Integer, Produto> produtos = new HashMap<>();

  @Override
  public void criar(Produto produto) {
     produtos.put(produto.getId(), produto);
  }

  @Override
  public Produto ler(int id) {
     return produtos.get(id);
  }

  @Override
  public void atualizar(Produto produto) {
     produtos.put(produto.getId(), produto);
```

```
    }

    @Override
    public void deletar(int id) {
        produtos.remove(id);
    }
}
```

Para separar a lógica de negócios do acesso aos dados, criamos a classe ProdutoService. Esta classe utiliza a interface IProdutoDAO para realizar operações nos produtos. Ela contém métodos para criar, ler, atualizar e deletar produtos, delegando essas operações à implementação de IProdutoDAO.

```
public class ProdutoService {
    private IProdutoDAO produtoDAO;

    public ProdutoService(IProdutoDAO produtoDAO) {
        this.produtoDAO = produtoDAO;
    }

    public void criarProduto(Produto produto) {
        produtoDAO.criar(produto);
    }

    public Produto lerProduto(int id) {
        return produtoDAO.ler(id);
    }

    public void atualizarProduto(Produto produto) {
        produtoDAO.atualizar(produto);
    }

    public void deletarProduto(int id) {
```

```
        produtoDAO.deletar(id);
    }
}
```

Por fim, criamos a classe Principal para demonstrar o uso do sistema de gerenciamento de produtos. Na classe Principal, instanciamos um objeto ProdutoDAOImpl e um objeto ProdutoService para manipular os produtos. Adicionamos dois produtos, lemos um produto do estoque, atualizamos o preço de um produto e deletamos um produto, mostrando como o sistema gerencia as operações CRUD.

```
public class Principal {
    public static void main(String[] args) {
        IProdutoDAO produtoDAO = new ProdutoDAOImpl();

        ProdutoService produtoService = new ProdutoService(produtoDAO);

        Produto produto1 = new Produto(1, "Produto A", 10.0, 100);
        Produto produto2 = new Produto(2, "Produto B", 20.0, 200);

        produtoService.criarProduto(produto1);
        produtoService.criarProduto(produto2);

        Produto p = produtoService.lerProduto(1);
        System.out.println("Produto lido: " + p.getNome() + ", Preço: " + p.getPreco());

        produto1.setPreco(12.0);
        produtoService.atualizarProduto(produto1);
        p = produtoService.lerProduto(1);
        System.out.println("Produto atualizado: " + p.getNome() + ", Preço: " + p.getPreco());

        produtoService.deletarProduto(2);
        System.out.println("Produto 2 deletado");
    }
```

```
}
```

Solução do Exercício 21

A aplicação de gestão de pedidos visa gerenciar pedidos, itens e produtos, permitindo operações CRUD nos dados dos pedidos. A separação das operações de acesso aos dados da lógica de negócios é crucial para facilitar a manutenção e evolução da solução. Para isso, será utilizado o padrão de projeto Repository, que abstrai o acesso aos dados de forma independente do tipo de banco de dados, proporcionando maior flexibilidade para futuras mudanças na fonte de dados.

Primeiramento, definimos a estrutura dos dados com as classes Pedido, Item e Produto. A classe Pedido representa um pedido e contém uma lista de itens. A classe Item representa um item do pedido e mantém uma referência ao produto associado. Por fim, a classe Produto representa um produto no estoque.

```
public class Produto {
  private int id;
  private String nome;
  private double preco;

  // Construtores, getters e setters
}

public class Item {
  private int id;
  private Produto produto;
  private int quantidade;

  // Construtores, getters e setters
}

public class Pedido {
  private int id;
  private List<Item> itens;
```

```
  // Construtores, getters e setters
}
```

Em seguida, criamos a interface PedidoRepository para definir os métodos necessários para manipular os pedidos no repositório. Esta interface permitirá separar a lógica de negócios da camada de acesso aos dados.

```
public interface PedidoRepository {
  void criar(Pedido pedido);
  Pedido ler(int id);
  void atualizar(Pedido pedido);
  void deletar(int id);
}
```

Agora, implementamos a interface PedidoRepository na classe PedidoRepositoryImpl. Esta implementação utilizará uma fonte de dados específica para armazenar os pedidos, permitindo realizar operações CRUD sobre eles.

```
public class PedidoRepositoryImpl implements PedidoRepository {
  private Map<Integer, Pedido> pedidos = new HashMap<>();

  @Override
  public void criar(Pedido pedido) {
    pedidos.put(pedido.getId(), pedido);
  }

  @Override
  public Pedido ler(int id) {
    return pedidos.get(id);
  }

  @Override
  public void atualizar(Pedido pedido) {
    pedidos.put(pedido.getId(), pedido);
```

```
  }

  @Override
  public void deletar(int id) {
      pedidos.remove(id);
  }
}
```

Com a estrutura de dados e o repositório definidos, podemos criar a classe PedidoService para separar a lógica de negócios das operações de acesso aos dados. Esta classe utilizará o PedidoRepository para realizar as operações CRUD nos pedidos.

```
public class PedidoService {
  private PedidoRepository pedidoRepository;

  public PedidoService(PedidoRepository pedidoRepository) {
      this.pedidoRepository = pedidoRepository;
  }

  public void criarPedido(Pedido pedido) {
      pedidoRepository.criar(pedido);
  }

  public Pedido lerPedido(int id) {
      return pedidoRepository.ler(id);
  }

  public void atualizarPedido(Pedido pedido) {
      pedidoRepository.atualizar(pedido);
  }

  public void deletarPedido(int id) {
      pedidoRepository.deletar(id);
```

```
  }
}
```

Por fim, na classe Principal, demonstramos o uso da aplicação de gestão de pedidos, instanciando um objeto PedidoRepositoryImpl e um objeto PedidoService. Realizamos a criação, leitura, atualização e exclusão de pedidos, mostrando como o sistema gerencia as operações CRUD.

```
public class Principal {
  public static void main(String[] args) {
    PedidoRepository pedidoRepository = new PedidoRepositoryImpl();
    PedidoService pedidoService = new PedidoService(pedidoRepository);

    Pedido pedido1 = new Pedido(1, new ArrayList<>());
    pedidoService.criarPedido(pedido1);

    Pedido pedidoRecuperado = pedidoService.lerPedido(1);
    System.out.println("Pedido lido: " + pedidoRecuperado);

    pedidoRecuperado.adicionarItem(new Item(1, produto1, 5));
    pedidoService.atualizarPedido(pedidoRecuperado);

    pedidoService.deletarPedido(1);
    System.out.println("Pedido 1 deletado");
  }
}
```

Essa estrutura da aplicação garante uma separação clara entre a lógica de negócios e o acesso aos dados, seguindo o padrão de projeto Repository para facilitar futuras mudanças na fonte de dados.

Solução do Exercício 22

Exercício resolvido no caderno.

Solução do Exercício 23

Primeiro, definimos a interface IComponente, que será implementada tanto por arquivos quanto por diretórios. Esta interface define um contrato com os métodos para obter o nome, obter o tamanho e imprimir a estrutura do componente.

```
public interface IComponente {
  String getNome();
  int getTamanho();
  void imprimirEstrutura(String caminho);
}
```

Em seguida, criamos a classe ArquivoImpl, que implementa a interface IComponente. A classe ArquivoImpl possui atributos para armazenar o nome e o tamanho em bytes do arquivo, além de implementar os métodos definidos na interface IComponente.

```
public class ArquivoImpl implements IComponente {
  private String nome;
  private int tamanhoEmBytes;

  public ArquivoImpl(String nome, int tamanhoEmBytes) {
    this.nome = nome;
    this.tamanhoEmBytes = tamanhoEmBytes;
  }

  @Override
  public String getNome() {
    return nome;
  }

  @Override
  public int getTamanho() {
    return tamanhoEmBytes;
  }
```

```
  @Override
  public void imprimirEstrutura(String caminho) {
    System.out.println(caminho + "- " + nome);
  }
}
```

Em seguida, criamos a classe DiretorioImpl, que também implementa a interface IComponente. A classe DiretorioImpl possui um nome e uma lista de componentes, que podem ser arquivos ou outros diretórios. Esta classe também implementa os métodos definidos na interface IComponente.

```
public class DiretorioImpl implements IComponente {
  private String nome;
  private List<IComponente> componentes;

  public DiretorioImpl(String nome) {
    this.nome = nome;
    this.componentes = new ArrayList<>();
  }

  public void adicionarComponente(IComponente componente) {
    componentes.add(componente);
  }

  @Override
  public String getNome() {
    return nome;
  }

  @Override
  public int getTamanho() {
    int tamanhoTotal = 0;
```

```
        for (IComponente componente : componentes) {
            tamanhoTotal += componente.getTamanho();
        }
        return tamanhoTotal;
    }

    @Override
    public void imprimirEstrutura(String caminho) {
        System.out.println(caminho + "> " + nome);
        for (IComponente componente : componentes) {
            componente.imprimirEstrutura(caminho + "  ");
        }
    }
}
```

Para demonstrar a extensibilidade do sistema, adicionamos a classe LinkSimbolicoImpl, que implementa a interface IComponente. A classe LinkSimbolicoImpl possui um nome e uma referência a outro componente. Esta classe também implementa os métodos definidos na interface IComponente.

```
public class LinkSimbolicoImpl implements IComponente {
    private String nome;
    private IComponente referencia;

    public LinkSimbolico(String nome, IComponente referencia) {
        this.nome = nome;
        this.referencia = referencia;
    }

    @Override
    public String getNome() {
        return nome;
    }
```

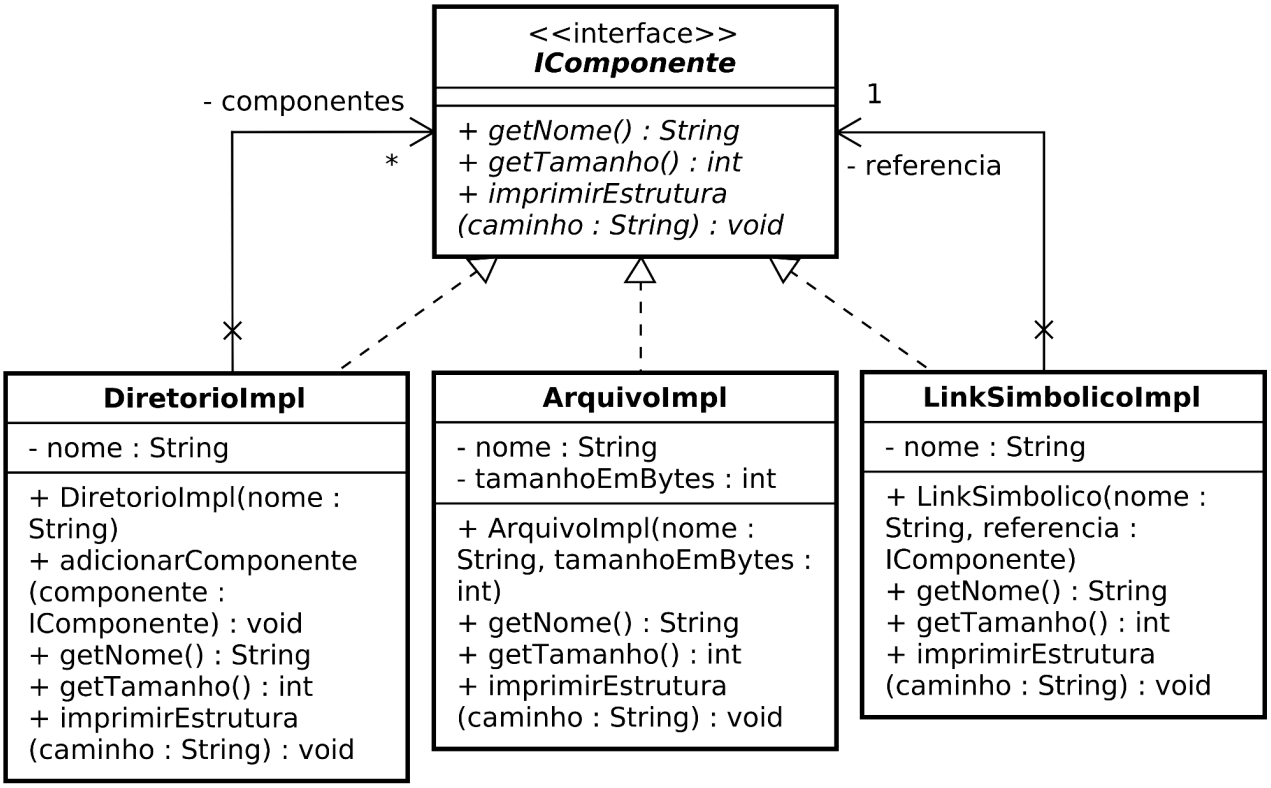
```
  @Override
  public int getTamanho() {
      return referencia.getTamanho();
  }

  @Override
  public void imprimirEstrutura(String caminho) {
      System.out.println(caminho + "-> " + nome + " (link para " + referencia.getNome() + ")");
  }
}
```

Por fim, criamos a classe Principal para demonstrar o uso do sistema de gerenciamento de arquivos. Nesta classe, instanciamos objetos de Arquivo, Diretorio e LinkSimbolico, e mostramos como adicionar componentes a um diretório, calcular o tamanho total de um diretório e imprimir a estrutura de diretórios.

```
public class Principal {
  public static void main(String[] args) {
      IComponente arquivo1 = new ArquivoImpl("Arquivo1.txt", 100);
      IComponente arquivo2 = new ArquivoImpl("Arquivo2.txt", 200);
      DiretorioImpl subDiretorio = new DiretorioImpl("SubDiretorio");
      subDiretorio.adicionarComponente(arquivo2);

      DiretorioImpl diretorioPrincipal = new DiretorioImpl("DiretorioPrincipal");
      diretorioPrincipal.adicionarComponente(arquivo1);
      diretorioPrincipal.adicionarComponente(subDiretorio);

      IComponenteImpl link = new LinkSimbolicoImpl("LinkParaArquivo1", arquivo1);
      diretorioPrincipal.adicionarComponente(link);

      System.out.println("Estrutura do diretório:");
      diretorioPrincipal.imprimirEstrutura("");
```

```
        System.out.println("Tamanho total do diretório: " + diretorioPrincipal.getTamanho() + "
bytes");
    }
}
```

Veja abaixo o diagrama de classes UML da solução proposta.



Este design aplica o padrão Composite ao sistema de gerenciamento de arquivos, permitindo a adição de novos tipos de componentes, como links simbólicos, sem modificar a classe DiretorioImpl. Além disso, a estrutura de diretórios é gerenciada de forma recursiva e o cálculo do tamanho total é simplificado. O uso de interfaces e classes concretas com convenções de nomenclatura apropriadas melhora a clareza e a extensibilidade do código.

Solução do Exercício 24

Primeiro, definimos a interface IComponente, que representa um componente genérico no sistema de gerenciamento de recursos, seja uma unidade ou um esquadrão. Esta interface possui métodos para obter o custo, obter a saúde e realizar um ataque.

```
public interface IComponente {
  int getCusto();
  int getSaude();
  void atacar();
}
```

Em seguida, criamos a classe UnidadeImpl, que implementa a interface IComponente. Esta classe representa uma unidade no jogo e possui atributos para armazenar o custo e a saúde da unidade, além de um método para realizar o ataque.

```
public class UnidadeImpl implements IComponente {
  private int custo;
  private int saude;

  public UnidadeImpl(int custo, int saude) {
    this.custo = custo;
    this.saude = saude;
  }

  @Override
  public int getCusto() {
    return custo;
  }

  @Override
  public int getSaude() {
    return saude;
  }

  @Override
  public void atacar() {
    System.out.println("Unidade atacando com força!");
  }
```

```
}
```

Depois, criamos a classe EsquadraoImpl, que também implementa a interface IComponente. Esta classe representa um esquadrão, que pode conter outros componentes (unidades e subesquadrões). O método atacar é recursivo e propaga a ação para todos os membros do esquadrão.

```
public class EsquadraoImpl implements IComponente {
  private List<IComponente> membros;

  public EsquadraoImpl() {
    this.membros = new ArrayList<>();
  }

  public void adicionarMembro(IComponente membro) {
    membros.add(membro);
  }

  @Override
  public int getCusto() {
    int custoTotal = 0;
    for (IComponente membro : membros) {
      custoTotal += membro.getCusto();
    }
    return custoTotal;
  }

  @Override
  public int getSaude() {
    int saudeTotal = 0;
    for (IComponente membro : membros) {
      saudeTotal += membro.getSaude();
    }
```

```
        return saudeTotal;
    }

    @Override
    public void atacar() {
        System.out.println("Esquadrão atacando!");
        for (IComponente membro : membros) {
            membro.atacar();
        }
    }
}
```

Para adicionar um novo tipo de componente ao sistema, como uma superunidade, podemos criar a classe SuperUnidadeImpl, que também implementa a interface IComponente. Esta classe possui atributos e métodos específicos para a superunidade.

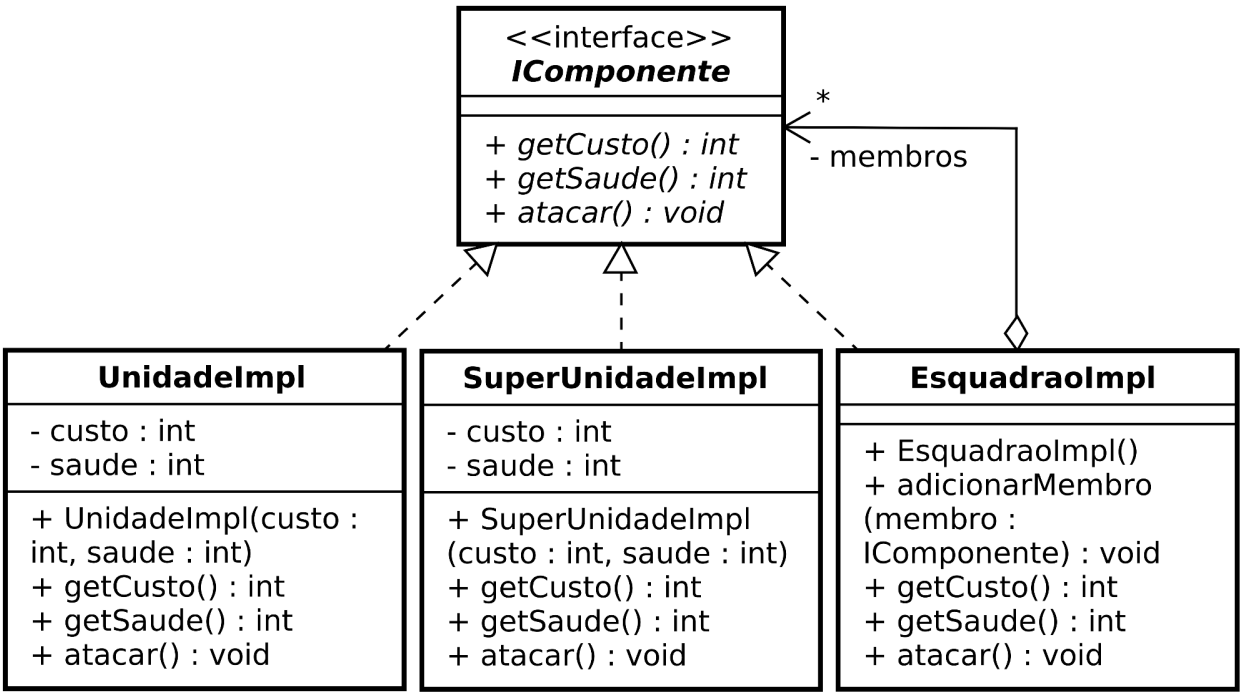
```
public class SuperUnidadeImpl implements IComponente {
    private int custo;
    private int saude;

    public SuperUnidadeImpl(int custo, int saude) {
        this.custo = custo;
        this.saude = saude;
    }

    @Override
    public int getCusto() {
        return custo;
    }

    @Override
    public int getSaude() {
        return saude;
```

```
    }

    @Override
    public void atacar() {
        System.out.println("SuperUnidade atacando com força máxima!");
    }
}
```

Por fim, criamos a classe Principal para demonstrar o uso do sistema de gerenciamento de recursos. Na classe Principal, instanciamos objetos UnidadeImpl, EsquadraoImpl e SuperUnidadeImpl, e manipulamos a estrutura de esquadrões e unidades.

```
public class Principal {
    public static void main(String[] args) {
        IComponente unidade1 = new UnidadeImpl(100, 50);
        IComponente unidade2 = new UnidadeImpl(150, 75);
        IComponente superUnidade = new SuperUnidadeImpl(300, 150);

        EsquadraoImpl esquadrao1 = new EsquadraoImpl();
        esquadrao1.adicionarMembro(unidade1);
        esquadrao1.adicionarMembro(unidade2);

        EsquadraoImpl esquadraoPrincipal = new EsquadraoImpl();
        esquadraoPrincipal.adicionarMembro(esquadrao1);
        esquadraoPrincipal.adicionarMembro(superUnidade);

        System.out.println("Custo total do esquadrão: " + esquadraoPrincipal.getCusto());
        System.out.println("Saúde total do esquadrão: " + esquadraoPrincipal.getSaude());

        System.out.println("Ataque do esquadrão:");
        esquadraoPrincipal.atacar();
    }
}
```

Veja a seguir o diagrama de classes da solução proposta.

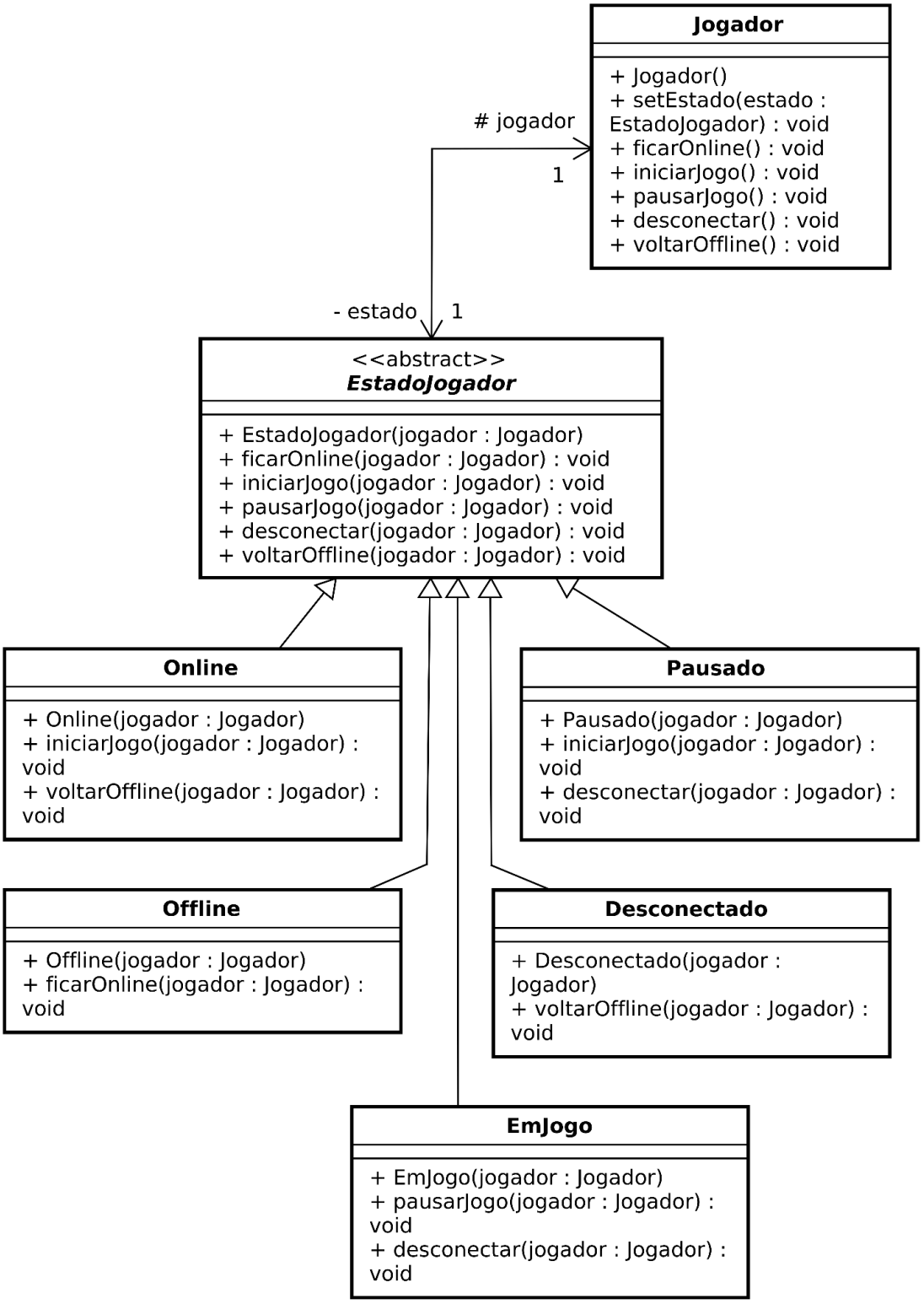


Este design aplica o padrão Composite ao sistema de gerenciamento de recursos para um jogo de estratégia, permitindo a adição de novos tipos de componentes, como superunidades, sem modificar a classe EsquadraoImpl. Além disso, a estrutura de esquadrões é gerenciada de forma recursiva e o cálculo do custo e da saúde totais é simplificado. O uso de interfaces e classes concretas com convenções de nomenclatura apropriadas melhora a clareza e a extensibilidade do código.

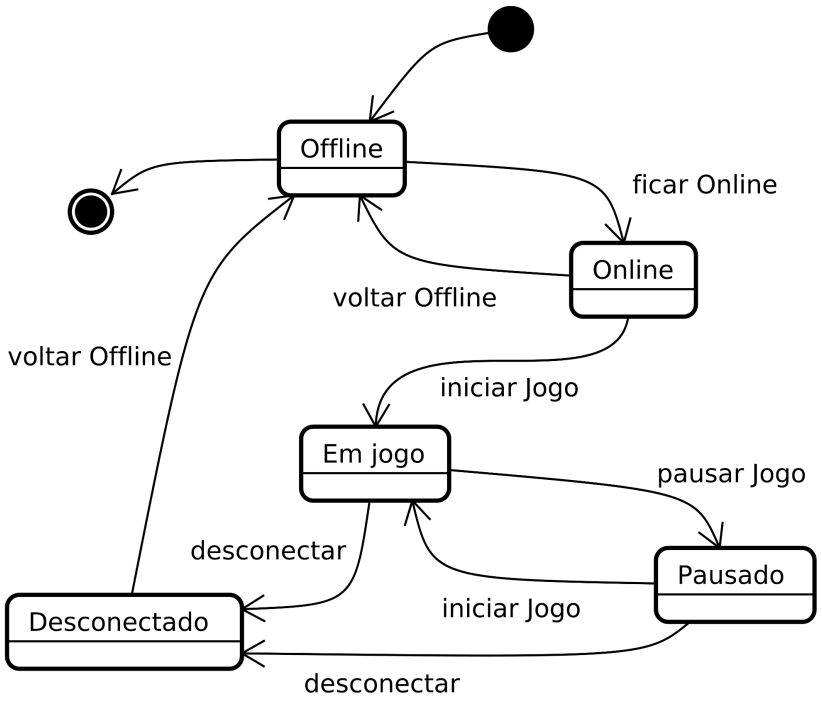
Solução do Exercício 25

Para resolver o problema proposto e seguir os princípios SOLID, implementamos o padrão de projeto State. Este padrão permite que um objeto altere seu comportamento quando seu estado interno muda. Utilizando este padrão, criamos diferentes classes para representar os diversos estados de um jogador, cada uma contendo a lógica correspondente a esse estado. Este design assegura que a classe Jogador não viole o Princípio de Responsabilidade Única nem o Princípio Aberto-Fechado.

A)


Jogador
+ Jogador()
+ setEstado(estado : EstadoJogador) : void
+ ficarOnline() : void
+ iniciarJogo() : void
+ pausarJogo() : void
+ desconectar() : void
+ voltarOffline() : void
# jogador
1
- estado
1
<<abstract>>
EstadoJogador
+ EstadoJogador(jogador : Jogador)
+ ficarOnline(jogador : Jogador) : void
+ iniciarJogo(jogador : Jogador) : void
+ pausarJogo(jogador : Jogador) : void
+ desconectar(jogador : Jogador) : void
+ voltarOffline(jogador : Jogador) : void
Online
+ Online(jogador : Jogador)
+ iniciarJogo(jogador : Jogador) : void
+ voltarOffline(jogador : Jogador) : void
Pausado
+ Pausado(jogador : Jogador)
+ iniciarJogo(jogador : Jogador) : void
+ desconectar(jogador : Jogador) : void
Offline
+ Offline(jogador : Jogador)
+ ficarOnline(jogador : Jogador) : void
Desconectado
+ Desconectado(jogador : Jogador)
+ voltarOffline(jogador : Jogador) : void
EmJogo
+ EmJogo(jogador : Jogador)
+ pausarJogo(jogador : Jogador) : void
+ desconectar(jogador : Jogador) : void

B)



C)

Primeiramente definimos a classe abstrata EstadoJogador, que contém métodos para as transições de estado possíveis, com um lançamento de exceção. Portanto, cada estado específico de um jogador implementa somente os métodos possíveis.

```
public abstract class EstadoJogador {
  protected Jogador jogador;

  public EstadoJogador(Jogador jogador){
     this.jogador = jogador;
  }

  public void ficarOnline(Jogador jogador){
     throw new RuntimeException("Não é possível realizar a operação de ficar online");
  }
  public void iniciarJogo(Jogador jogador){
     throw new RuntimeException("Não é possível realizar a operação de iniciar jogo");
  }
  public void pausarJogo(Jogador jogador){
```

```
        throw new RuntimeException("Não é possível realizar a operação de pausar jogo");
    }
    public void desconectar(Jogador jogador){
        throw new RuntimeException("Não é possível realizar a operação de desconectar");
    }
    public void voltarOffline(Jogador jogador){
        throw new RuntimeException("Não é possível realizar a operação de voltar Offline");
    }
}
```

Para resolver o problema dos estados, criamos a classe Jogador, que mantém uma referência ao estado atual do jogador. A classe Jogador delega as mudanças de estado aos métodos do objeto EstadoJogador atual.

```
public class Jogador {
    private EstadoJogador estado;

    public Jogador() {
        this.estado = new Offline(this); // Estado inicial é Offline
    }

    public void setEstado(EstadoJogador estado) {
        this.estado = estado;
    }

    public void ficarOnline() {
        estado.ficarOnline(this);
    }

    public void iniciarJogo() {
        estado.iniciarJogo(this);
    }
```

```
  public void pausarJogo() {
      estado.pausarJogo(this);
  }

  public void desconectar() {
      estado.desconectar(this);
  }

  public void voltarOffline() {
      estado.voltarOffline(this);
  }
}
```

Em seguida, criamos as classes que representam os diferentes estados de um jogador: Offline, Online, EmJogo, Pausado e Desconectado. Cada classe estende a classe abstrata EstadoJogador e define o comportamento específico para os métodos possíveis.

```
public class Offline extends EstadoJogador {
  public Offline(Jogador jogador){
      super(jogador);
  }

  @Override
  public void ficarOnline(Jogador jogador) {
      jogador.setEstado(new Online(jogador));
      System.out.println("Jogador agora está online.");
  }
}

public class Online extends EstadoJogador {
  public Online(Jogador jogador){
      super(jogador);
  }
```

```
    @Override
    public void iniciarJogo(Jogador jogador) {
        jogador.setEstado(new EmJogo(jogador));
        System.out.println("Jogador agora está em jogo.");
    }

    @Override
    public void voltarOffline(Jogador jogador) {
        jogador.setEstado(new Offline(jogador));
        System.out.println("Jogador agora está offline.");
    }
}

public class EmJogo extends EstadoJogador {
    public EmJogo(Jogador jogador){
        super(jogador);
    }

    @Override
    public void pausarJogo(Jogador jogador) {
        jogador.setEstado(new Pausado(jogador));
        System.out.println("Jogo pausado.");
    }

    @Override
    public void desconectar(Jogador jogador) {
        jogador.setEstado(new Desconectado(jogador));
        System.out.println("Jogador desconectado.");
    }
}
```

```
public class Pausado extends EstadoJogador {
  public Pausado(Jogador jogador){
     super(jogador);
  }

  @Override
  public void iniciarJogo(Jogador jogador) {
     jogador.setEstado(new EmJogo(jogador));
     System.out.println("Jogo retomado.");
  }

  @Override
  public void desconectar(Jogador jogador) {
     jogador.setEstado(new Desconectado(jogador));
     System.out.println("Jogador desconectado.");
  }
}

public class Desconectado extends EstadoJogador {
  public Desconectado(Jogador jogador){
     super(jogador);
  }

  @Override
  public void voltarOffline(Jogador jogador) {
     jogador.setEstado(new Offline(jogador));
     System.out.println("Jogador agora está offline.");
  }
}
```

D)

Na classe principal, demonstramos a criação de um jogador e as transições de estado possíveis.

```
public class Principal {
  public static void main(String[] args) {
    Jogador jogador = new Jogador();

    jogador.ficarOnline();
    jogador.iniciarJogo();
    jogador.pausarJogo();
    jogador.desconectar();
    jogador.voltarOffline();
  }
}
```

A saída esperada após a execução do método main é a seguinte:

```
Jogador agora está online.
Jogador agora está em jogo.
Jogo pausado.
Jogador desconectado.
Jogador agora está offline.
```

Solução do Exercício 26

O sistema de gestão de transações financeiras apresenta módulos distintos para transações de débito, crédito e transferência, cada um deles projetado separadamente sem qualquer tentativa de abstração ou reutilização de código para o processamento comum de transações, resultando em redundância a nível de design.

Para corrigir essa violação do Princípio DRY, será necessário criar uma abstração que represente o processamento comum de transações, evitando a duplicação de código e melhorando a manutenibilidade do sistema.

A solução proposta consiste na criação de uma classe abstrata Transação, que conterá os métodos comuns a todas as transações, como validação e registro. Essa classe abstrata será

herdada pelas classes Debito, Credito e Transferencia, que implementarão os métodos específicos de cada tipo de transação.

A classe Transação conterá os métodos processar(), validarTransacao() e registrarTransacao(), que serão herdados pelas classes filhas. O método processar() será responsável por validar e registrar a transação, enquanto os métodos validarTransacao() e registrarTransacao() conterão as regras e lógicas de negócio fundamentais para validação e registro de transações, respectivamente.

As classes Debito, Credito e Transferencia herdarão a classe Transação e implementarão os métodos específicos de cada tipo de transação, como realizarDebito(), realizarCredito() e realizarTransferencia(). Esses métodos conterão as regras e lógicas de negócio específicas para cada tipo de transação, mas não duplicarão o código comum de validação e registro de transações.

Com essa solução, o sistema de gestão de transações financeiras estará alinhado ao Princípio DRY, evitando a redundância de código e melhorando a manutenibilidade do sistema. Além disso, a abstração da classe Transação permitirá a criação de novos tipos de transações de forma mais fácil e rápida, uma vez que as regras e lógicas de negócio comuns já estarão implementadas.

Após a análise do problema apresentado, foi identificada a necessidade de se criar uma abstração para evitar a redundância de código e melhorar a manutenibilidade do sistema. Para isso, será criada uma classe abstrata chamada "Transacao" que conterá os métodos comuns a todas as transações, como validação e registro. Essa classe será herdada pelas classes "Debito", "Credito" e "Transferencia", que implementarão os métodos específicos de cada tipo de transação.

A classe "Transacao" conterá o método "processar()", que será responsável por validar e registrar a transação. Esse método será herdado pelas classes filhas e será utilizado para processar cada tipo de transação. O código da classe "Transacao" é apresentado abaixo:

```
public abstract class Transacao {
  public void processar() {
    if (validarTransacao()) {
```

```
            registrarTransacao();
        }
    }

    protected abstract boolean validarTransacao();

    protected abstract void registrarTransacao();
}
```

A classe "Debito" herdará a classe "Transacao" e implementará o método "validarTransacao()" para validar as transações de débito e o método "registrarTransacao()" para registrar as transações de débito. O código da classe "Debito" é apresentado abaixo:

```
public class Debito extends Transacao {
    @Override
    protected boolean validarTransacao() {
        if (getValor() <= 0) {
            throw new IllegalArgumentException("Valor da transação deve ser maior que zero.");
        }
        if (getSaldoConta() < getValor()) {
            throw new IllegalArgumentException("Saldo insuficiente para realizar a transação.");
        }
        return true;
    }

    @Override
    protected void registrarTransacao() {
        getContaOrigem().debitar(getValor());
        getContaDestino().creditar(getValor());
    }
}
```

A classe "Credito" herdará a classe "Transacao" e implementará o método "validarTransacao()" para validar as transações de crédito e o método "registrarTransacao()" para registrar as transações de crédito. O código da classe "Credito" é apresentado abaixo:

```
public class Credito extends Transacao {
    @Override
    protected boolean validarTransacao() {
        if (getValor() <= 0) {
            throw new IllegalArgumentException("Valor da transação deve ser maior que zero.");
        }
        return true;
    }

    @Override
    protected void registrarTransacao() {
        getContaOrigem().creditar(getValor());
        getContaDestino().debitar(getValor());
    }
}
```

A classe "Transferencia" herdará a classe "Transacao" e implementará o método "validarTransacao()" para validar as transações de transferência e o método "registrarTransacao()" para registrar as transações de transferência. O código da classe "Transferencia" é apresentado abaixo:

```
public class Transferencia extends Transacao {
    @Override
    protected boolean validarTransacao() {
        if (getValor() <= 0) {
            throw new IllegalArgumentException("Valor da transação deve ser maior que zero.");
        }
        if (getContaOrigem().equals(getContaDestino())) {
            throw new IllegalArgumentException("Conta de origem e conta de destino não podem ser iguais.");
```

```
        }
        if (getSaldoConta() < getValor()) {
            throw new IllegalArgumentException("Saldo insuficiente para realizar a transação.");
        }
        return true;
    }

    @Override
    protected void registrarTransacao() {
        getContaOrigem().debitar(getValor());
        getContaDestino().creditar(getValor());
    }
}
```

Com essa solução, o sistema de gestão de transações financeiras está alinhado ao Princípio DRY, evitando a redundância de código e melhorando a manutenibilidade do sistema.

Solução do Exercício 27

Para demonstrar o uso do padrão de projeto Composite na construção de formulários em uma interface gráfica com o usuário, criaremos agora um formulário para o registro de uma música. Utilizaremos classes para representar elementos de formulário como campos de texto, botões, campos de calendário, campos numéricos, botões de opção (radio buttons), caixas de seleção (checkboxes) e caixas de seleção múltipla (select). A classe ComponenteFormulario será a abstração base para todos os componentes, enquanto as classes concretas implementarão os comportamentos específicos de cada elemento. A classe Formulario atuará como o Composite, contendo e gerenciando os diversos componentes.

A classe abstrata ComponenteFormulario define o método renderizar, responsável por retornar o código HTML correspondente ao componente.

```
public abstract class ComponenteFormulario {
    public abstract String renderizar();
}
```

As classes concretas, como CampoTexto, Botao, CampoCalendario, CampoNumerico, BotaoOpcao, CaixaSelecao e CaixaSelecaoMultipla, herdam de ComponenteFormulario e implementam o método renderizar para produzir o HTML específico de cada elemento.

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

public class CampoTexto extends ComponenteFormulario {
  private String nome;
  private String placeholder;

  public CampoTexto(String nome, String placeholder) {
    this.nome = nome;
    this.placeholder = placeholder;
  }

  @Override
  public String renderizar() {
    return "<input type=\"text\" name=\"" + nome + "\" placeholder=\"" + placeholder + "\" />";
  }
}

public class Botao extends ComponenteFormulario {
  private String texto;

  public Botao(String texto) {
    this.texto = texto;
  }

  @Override
  public String renderizar() {
    return "<button>" + texto + "</button>";
  }
```

```
}

public class CampoCalendario extends ComponenteFormulario {
  private String nome;
  private String placeholder;

  public CampoCalendario(String nome, String placeholder) {
    this.nome = nome;
    this.placeholder = placeholder;
  }

  @Override
  public String renderizar() {
    return "<input type=\"date\" name=\"" + nome + "\" placeholder=\"" + placeholder + "\" />";
  }
}

public class CampoNumerico extends ComponenteFormulario {
  private String nome;
  private String placeholder;

  public CampoNumerico(String nome, String placeholder) {
    this.nome = nome;
    this.placeholder = placeholder;
  }

  @Override
  public String renderizar() {
    return "<input type=\"number\" name=\"" + nome + "\" placeholder=\"" + placeholder + "\" />";
  }
}
```

```
public class BotaoOpcao extends ComponenteFormulario {
  private String nome;
  private String valor;
  private String texto;

  public BotaoOpcao(String nome, String valor, String texto) {
    this.nome = nome;
    this.valor = valor;
    this.texto = texto;
  }

  @Override
  public String renderizar() {
    return "<input type=\"radio\" name=\"" + nome + "\" value=\"" + valor + "\" />" + texto;
  }
}

public class CaixaSelecao extends ComponenteFormulario {
  private String nome;
  private String texto;

  public CaixaSelecao(String nome, String texto) {
    this.nome = nome;
    this.texto = texto;
  }

  @Override
  public String renderizar() {
    return "<input type=\"checkbox\" name=\"" + nome + "\" />" + texto;
  }
}
```

```
public class CaixaSelecaoMultipla extends ComponenteFormulario {
  private String nome;
  private List<String> opcoes;

  public CaixaSelecaoMultipla(String nome, List<String> opcoes) {
    this.nome = nome;
    this.opcoes = opcoes;
  }

  @Override
  public String renderizar() {
    StringBuilder html = new StringBuilder();
    html.append("<select name=\"").append(nome).append("\">");
    for (String opcao : opcoes) {
      html.append("<option
value=\"").append(opcao).append("\">").append(opcao).append("</option>");
    }
    html.append("</select>");
    return html.toString();
  }
}
```

A classe Formulario atua como o Composite, mantendo uma lista de componentes (componentes) e fornecendo o método adicionarComponente para adicionar novos elementos à lista. O método renderizar de Formulario itera sobre todos os componentes, concatenando os resultados dos métodos renderizar individuais de cada componente em uma única string de HTML.

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

public class Formulario extends ComponenteFormulario {
  private List<ComponenteFormulario> componentes = new ArrayList<>();
```

```
  public void adicionarComponente(ComponenteFormulario componente) {
    componentes.add(componente);
  }

  @Override
  public String renderizar() {
    StringBuilder html = new StringBuilder();
    html.append("<form>");
    for (ComponenteFormulario componente : componentes) {
      html.append(componente.renderizar()).append("<br>");
    }
    html.append("</form>");
    return html.toString();
  }
}
```

No método main da classe Principal, criamos uma instância de Formulario e adicionamos vários componentes, como campos de texto, campos numéricos, campos de calendário, uma caixa de seleção múltipla para os gêneros musicais e botões de registro e cancelamento. Finalmente, chamamos o método renderizar de Formulario para obter e imprimir o HTML completo do formulário.

```
public class Principal {
  public static void main(String[] args) {
    Formulario formulario = new Formulario();

    List<String> generosMusicais = new ArrayList<>();
    generosMusicais.add("Rock");
    generosMusicais.add("Pop");
    generosMusicais.add("Jazz");
    generosMusicais.add("Clássica");
    generosMusicais.add("Eletrônica");
```

```
        formulario.adicionarComponente(new CampoTexto("titulo", "Digite o título da música"));
        formulario.adicionarComponente(new CampoTexto("artista", "Digite o nome do artista"));
        formulario.adicionarComponente(new CampoNumerico("duracao", "Digite a duração em
segundos"));
        formulario.adicionarComponente(new CampoCalendario("dataLancamento", "Selecione a
data de lançamento"));
        formulario.adicionarComponente(new CaixaSelecaoMultipla("genero", generosMusicais));
        formulario.adicionarComponente(new Botao("Registrar"));
        formulario.adicionarComponente(new Botao("Cancelar"));

        System.out.println(formulario.renderizar());
    }
}
```

Em outro exemplo de uso, no método main da classe Main, criamos uma instância de Formulario e adicionamos vários componentes relevantes para o registro de um usuário. Esses componentes incluem campos de texto para o nome completo, email, nome de usuário e senha; um campo de calendário para selecionar a data de nascimento; uma caixa de seleção para aceitar os termos e condições; e botões para registrar e cancelar a operação. Finalmente, chamamos o método renderizar de Formulario para obter e imprimir o HTML completo do formulário de registro de usuário.

```
public class Main {
    public static void main(String[] args) {
        Formulario formulario = new Formulario();

        formulario.adicionarComponente(new CampoTexto("nome", "Digite seu nome completo"));
        formulario.adicionarComponente(new CampoTexto("email", "Digite seu email"));
        formulario.adicionarComponente(new CampoTexto("usuario", "Escolha um nome de
usuário"));
        formulario.adicionarComponente(new CampoTexto("senha", "Digite sua senha"));
```

```
        formulario.adicionarComponente(new CampoCalendario("dataNascimento", "Selecione sua data de nascimento"));
        formulario.adicionarComponente(new CaixaSelecao("aceitarTermos", "Aceito os termos e condições"));
        formulario.adicionarComponente(new Botao("Registrar"));
        formulario.adicionarComponente(new Botao("Cancelar"));

        System.out.println(formulario.renderizar());
    }
}
```

Solução do Exercício 28

Para resolver a violação mencionada, é necessário garantir que a subclasse não altere a semântica fundamental da operação de remoção definida pela classe base. Em vez de redefinir completamente a estratégia de remoção, a GerenciadorTarefasComHistorico pode incorporar a funcionalidade de manter um histórico de maneira independente, sem alterar a operação de remoção esperada.

Código Corrigido

Classe Abstrata GerenciadorTarefas

```
import java.util.Stack;

public abstract class GerenciadorTarefas {
    protected Stack<ITarefa> pilhaTarefas = new Stack<>();

    public void push(ITarefa tarefa) {
        pilhaTarefas.push(tarefa);
    }

    public ITarefa pop() {
        return pilhaTarefas.pop();
    }
```

```
    public abstract void processar();
}
```

Subclasse GerenciadorTarefasComHistorico

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

public class GerenciadorTarefasComHistorico extends GerenciadorTarefas {
    private List<ITarefa> historicoTarefas = new ArrayList<>();

    public ITarefa popComHistorico() {
        ITarefa tarefa = super.pop();
        historicoTarefas.add(tarefa);
        return tarefa;
    }

    public List<ITarefa> getHistorico() {
        return historicoTarefas;
    }

    @Override
    public void processar() {
        while (!pilhaTarefas.isEmpty()) {
            ITarefa tarefa = popComHistorico();
            notificarTarefaProcessada(tarefa);
        }
    }

    private void notificarTarefaProcessada(ITarefa tarefa) {
        System.out.println("Tarefa " + tarefa.getDescricao() + " foi processada.");
    }
```

```
}
```

Neste código, a classe GerenciadorTarefasComHistorico mantém a funcionalidade adicional de manter um histórico das tarefas removidas, sem alterar o comportamento do método pop da classe base. A funcionalidade de histórico é incorporada através de um método separado popComHistorico, mantendo assim a aderência ao Princípio de Substituição de Liskov.

Classe Principal Main

```
public class Main {
    public static void main(String[] args) {
        GerenciadorTarefas gerenciadorComHistorico = new GerenciadorTarefasComHistorico();

        // Adicionando tarefas à pilha
        gerenciadorComHistorico.push(new Tarefa("Tarefa 1"));
        gerenciadorComHistorico.push(new Tarefa("Tarefa 2"));

        // Processando tarefas
        gerenciadorComHistorico.processar();

        // Exibindo histórico de tarefas processadas
        if (gerenciadorComHistorico instanceof GerenciadorTarefasComHistorico) {
            List<ITarefa> historico = ((GerenciadorTarefasComHistorico)
gerenciadorComHistorico).getHistorico();
            System.out.println("Histórico de tarefas processadas:");
            for (ITarefa tarefa : historico) {
                System.out.println(tarefa.getDescricao());
            }
        }
    }
}
```

Classe Tarefa

```
public class Tarefa implements ITarefa {
```

```
    private String descricao;

    public Tarefa(String descricao) {
        this.descricao = descricao;
    }

    @Override
    public String getDescricao() {
        return descricao;
    }
}
```

Neste exemplo, a classe Main inicializa uma instância de GerenciadorTarefasComHistorico e utiliza seus métodos para adicionar e processar tarefas. A funcionalidade de histórico é acessada através de um método separado, preservando a semântica esperada da operação de remoção definida na classe base. Isso ilustra a flexibilidade do design e como diferentes comportamentos de gerenciamento podem ser integrados e manipulados usando uma única interface de serviço, mantendo a consistência com o Princípio de Substituição de Liskov.

Solução do Exercício 29

No código atual, a classe `AutenticacaoService` possui os métodos `autenticar()` e `autorizar()` que acessam diretamente os atributos dos objetos `Usuario` e `Permissao`, violando a Lei de Demeter.

Para corrigir essa violação, é necessário encapsular a obtenção dos atributos dos objetos `Usuario` e `Permissao` dentro das respectivas classes, criando métodos que retornem esses atributos de forma encapsulada. Em seguida, a classe `AutenticacaoService` deve utilizar esses métodos para obter os atributos dos objetos `Usuario` e `Permissao`, em vez de acessá-los diretamente. Abaixo está o código corrigido:

```
public class AutenticacaoService {
    private List<Usuario> usuarios;
    private List<Permissao> permissoes;
```

```
    public AutenticacaoService(List<Usuario> usuarios, List<Permissao> permissoes) {
        this.usuarios = usuarios;
        this.permissoes = permissoes;
    }

    public boolean autenticar(String nomeUsuario, String senha) {
        for (Usuario usuario : usuarios) {
            if (usuario.autenticar(nomeUsuario, senha)) {
                return true;
            }
        }
        return false;
    }

    public boolean autorizar(String nomeUsuario, String recurso) {
        for (Permissao permissao : permissoes) {
            if (permissao.autorizar(nomeUsuario, recurso)) {
                return true;
            }
        }
        return false;
    }
}

public class Usuario {
    private String nome;
    private String senha;

    public Usuario(String nome, String senha) {
        this.nome = nome;
        this.senha = senha;
    }
```

```
    public String getNome() {
        return nome;
    }

    public boolean autenticar(String nome, String senha) {
        return this.nome.equals(nome) && this.senha.equals(senha);
    }
}

public class Permissao {
    private Usuario usuario;
    private String recurso;

    public Permissao(Usuario usuario, String recurso) {
        this.usuario = usuario;
        this.recurso = recurso;
    }

    public Usuario getUsuario() {
        return usuario;
    }

    public String getRecurso() {
        return recurso;
    }

    public boolean autorizar(String nomeUsuario, String recurso) {
        return usuario.getNome().equals(nomeUsuario) && this.recurso.equals(recurso);
    }
}
```

No código acima, as classes Usuario e Permissao possuem os métodos autenticar() e autorizar(), respectivamente, que retornam verdadeiro ou falso com base nas informações de autenticação e autorização do usuário. A classe AutenticacaoService utiliza esses métodos para obter as informações de autenticação e autorização dos objetos Usuario e Permissao, em vez de acessar diretamente os atributos dos objetos, corrigindo assim a violação da Lei de Demeter.

Solução do Exercício 30

Exercício resolvido no caderno.

Solução do Exercício 31

Vamos criar um exemplo funcional utilizando o padrão de projeto Abstract Factory para uma aplicação de parsing de HTML. O padrão Abstract Factory permite a criação de famílias de objetos relacionados sem especificar suas classes concretas. No contexto de parsing de HTML, esse padrão pode ser utilizado para criar objetos que representam diferentes elementos de documentos HTML, como parágrafos, tabelas, imagens, links e formulários. Isso permite que os desenvolvedores manipulem e extraiam informações de documentos HTML de forma estruturada e eficiente, sem precisar lidar diretamente com o código HTML bruto. A utilidade do Abstract Factory está em permitir a substituição de tecnologias no futuro sem alterar a lógica principal.

Para resolver essa necessidade, começamos com a criação da interface IElementoHTML que define o método renderizar para todos os elementos HTML.

```
public interface IElementoHTML {
  String renderizar();
}
```

Em seguida, implementamos as classes concretas para cada tipo de elemento HTML. Cada uma dessas classes implementa a interface IElementoHTML e o método renderizar.

```
public class TextoHTML implements IElementoHTML {
  private String conteudo;
```

```
    public TextoHTML(String conteudo) {
        this.conteudo = conteudo;
    }

    @Override
    public String renderizar() {
        return conteudo;
    }
}

public class ParagrafoHTML implements IElementoHTML {
    private String conteudo;

    public ParagrafoHTML(String conteudo) {
        this.conteudo = conteudo;
    }

    @Override
    public String renderizar() {
        return "<p>" + conteudo + "</p>";
    }
}

public class ImagemHTML implements IElementoHTML {
    private String url;
    private String alt;

    public ImagemHTML(String url, String alt) {
        this.url = url;
        this.alt = alt;
    }
```

```
    @Override
    public String renderizar() {
        return "<img src=\"" + url + "\" alt=\"" + alt + "\">";
    }
}

public class TabelaHTML implements IElementoHTML {
    private String[][] dados;

    public TabelaHTML(String[][] dados) {
        this.dados = dados;
    }

    @Override
    public String renderizar() {
        StringBuilder sb = new StringBuilder();
        sb.append("<table>");
        for (String[] linha : dados) {
            sb.append("<tr>");
            for (String celula : linha) {
                sb.append("<td>").append(celula).append("</td>");
            }
            sb.append("</tr>");
        }
        sb.append("</table>");
        return sb.toString();
    }
}

public class LinkHTML implements IElementoHTML {
    private String url;
    private String texto;
```

```
    public LinkHTML(String url, String texto) {
        this.url = url;
        this.texto = texto;
    }

    @Override
    public String renderizar() {
        return "<a href=\"" + url + "\">" + texto + "</a>";
    }
}

public class FormularioHTML implements IElementoHTML {
    private String acao;
    private String metodo;

    public FormularioHTML(String acao, String metodo) {
        this.acao = acao;
        this.metodo = metodo;
    }

    @Override
    public String renderizar() {
        return "<form action=\"" + acao + "\" method=\"" + metodo + "\">";
    }
}
```

Para resolver o problema da criação de elementos HTML de forma estruturada, definimos a interface FabricaElemento, que especifica as operações para criar diferentes tipos de elementos HTML.

```
public interface FabricaElemento {
    IElementoHTML criarTexto(String conteudo);
```

```
  IElementoHTML criarParagrafo(String conteudo);
  IElementoHTML criarImagem(String url, String alt);
  IElementoHTML criarTabela(String[][] dados);
  IElementoHTML criarLink(String url, String texto);
  IElementoHTML criarFormulario(String acao, String metodo);
}
```

Em seguida, implementamos a interface FabricaElemento na classe FabricaElementoHTML, que cria instâncias dos elementos HTML.

```
public class FabricaElementoHTML implements FabricaElemento {
  @Override
  public IElementoHTML criarTexto(String conteudo) {
     return new TextoHTML(conteudo);
  }

  @Override
  public IElementoHTML criarParagrafo(String conteudo) {
     return new ParagrafoHTML(conteudo);
  }

  @Override
  public IElementoHTML criarImagem(String url, String alt) {
     return new ImagemHTML(url, alt);
  }

  @Override
  public IElementoHTML criarTabela(String[][] dados) {
     return new TabelaHTML(dados);
  }

  @Override
  public IElementoHTML criarLink(String url, String texto) {
```

```
        return new LinkHTML(url, texto);
    }

    @Override
    public IElementoHTML criarFormulario(String acao, String metodo) {
        return new FormularioHTML(acao, metodo);
    }
}
```

Finalmente, a classe DocumentoHTML utiliza a fábrica abstrata para criar e manipular elementos HTML, oferecendo métodos para adicionar texto, parágrafos, imagens, tabelas, links e formulários ao documento.

```
public class DocumentoHTML {
    private FabricaElemento fabricaElementos;

    public DocumentoHTML(FabricaElemento fabricaElementos) {
        this.fabricaElementos = fabricaElementos;
    }

    public String adicionarTexto(String conteudo) {
        IElementoHTML texto = fabricaElementos.criarTexto(conteudo);
        return texto.renderizar();
    }

    public String adicionarParagrafo(String conteudo) {
        IElementoHTML paragrafo = fabricaElementos.criarParagrafo(conteudo);
        return paragrafo.renderizar();
    }

    public String adicionarImagem(String url, String alt) {
        IElementoHTML imagem = fabricaElementos.criarImagem(url, alt);
        return imagem.renderizar();
```

```
    }

    public String adicionarTabela(String[][] dados) {
        IElementoHTML tabela = fabricaElementos.criarTabela(dados);
        return tabela.renderizar();
    }

    public String adicionarLink(String url, String texto) {
        IElementoHTML link = fabricaElementos.criarLink(url, texto);
        return link.renderizar();
    }

    public String adicionarFormulario(String acao, String metodo) {
        IElementoHTML formulario = fabricaElementos.criarFormulario(acao, metodo);
        return formulario.renderizar();
    }
}
```

Por fim, a classe Principal demonstra o uso de todas essas classes. Ela cria uma instância de FabricaElementoHTML e a utiliza para criar um documento HTML com diferentes elementos. As strings HTML resultantes são então impressas no console, mostrando a estrutura HTML gerada pelo sistema.

```
public class Principal {
    public static void main(String[] args) {
        FabricaElemento fabrica = new FabricaElementoHTML();
        DocumentoHTML documento = new DocumentoHTML(fabrica);

        String paragrafo = documento.adicionarParagrafo("Este é um parágrafo de exemplo.");
        String imagem = documento.adicionarImagem("https://exemplo.com/imagem.jpg",
"Imagem de exemplo");
        String[][] dadosTabela = {
            {"Cabeçalho 1", "Cabeçalho 2"},
```

```
            {"Dado 1", "Dado 2"},
            {"Dado 3", "Dado 4"}
        };
        String tabela = documento.adicionarTabela(dadosTabela);
        String link = documento.adicionarLink("https://exemplo.com", "Clique aqui");
        String formulario = documento.adicionarFormulario("/submit", "post");

        System.out.println(paragrafo);
        System.out.println(imagem);
        System.out.println(tabela);
        System.out.println(link);
        System.out.println(formulario + "</form>");
    }
}
```

A saída esperada após a execução do método main é a seguinte:

```
<p>Este é um parágrafo de exemplo.</p>
<img src="https://exemplo.com/imagem.jpg" alt="Imagem de exemplo">
<table><tr><td>Cabeçalho 1</td><td>Cabeçalho 2</td></tr><tr><td>Dado 1</td><td>Dado 2</td></tr><tr><td>Dado 3</td><td>Dado 4</td></tr></table>
<a href="https://exemplo.com">Clique aqui</a>
<form action="/submit" method="post"></form>
```

Solução do Exercício 32

Exercício resolvido no caderno.

Solução do Exercício 33

Para resolver completamente a violação do princípio de responsabilidade única (SRP) na classe Pedido, criaremos uma nova classe chamada CalculadoraFreteService, que será responsável por delegar a chamada da estratégia de frete. A classe Pedido ficará apenas com a responsabilidade de gerenciar as informações do pedido.

Primeiro, vamos definir a interface IFrete, que declara o método calcular para o cálculo de frete:

```
public interface IFrete {
  double calcular();
}
```

Em seguida, criaremos classes concretas para cada tipo de frete, implementando a interface Frete:

```
public class FreteNormalImpl implements IFrete {
  @Override
  public double calcular() {
     return 10.0;
  }
}

public class FreteSedexImpl implements IFrete {
  @Override
  public double calcular() {
     return 20.0;
  }
}

public class FreteExpressoImpl implements IFrete {
  @Override
  public double calcular() {
     return 30.0;
  }
}
```

Agora, criamos a classe CalculadoraFreteService, que será responsável por delegar a chamada da estratégia de frete:

```
public class CalculadoraFreteService {
  private IFrete frete;

  public void setFrete(IFrete frete) {
```

```
        this.frete = frete;
    }

    public double calcularFrete() {
        if (frete == null) {
            throw new RuntimeException("Tipo de frete não definido");
        }
        return frete.calcular();
    }
}
```

Finalmente, a classe Pedido será simplificada, mantendo apenas as informações do pedido:

```
public class Pedido {
    private List<Produto> produtos;

    public Pedido(List<Produto> produtos) {
        this.produtos = produtos;
    }

    public List<Produto> getProdutos() {
        return produtos;
    }

    public void setProdutos(List<Produto> produtos) {
        this.produtos = produtos;
    }
}
```

Na classe Principal, demonstramos como configurar e calcular o frete usando diferentes estratégias, utilizando a CalculadoraFreteService:

```
public class Principal {
    public static void main(String[] args) {
```

```
    Pedido pedido = new Pedido(new ArrayList<>());
    CalculadoraFreteService calculadoraFreteService = new CalculadoraFreteService();

    calculadoraFreteService.setFrete(new FreteNormalImpl());
    System.out.println("Frete Normal: " + calculadoraFreteService.calcularFrete());

    calculadoraFreteService.setFrete(new FreteSedexImpl());
    System.out.println("Frete SEDEX: " + calculadoraFreteService.calcularFrete());

    calculadoraFreteService.setFrete(new FreteExpressoImpl());
    System.out.println("Frete Expresso: " + calculadoraFreteService.calcularFrete());
  }
}
```

Essa solução distribui as responsabilidades de maneira adequada. A classe Pedido apenas gerencia as informações do pedido, enquanto a CalculadoraFreteService lida com a lógica de cálculo de frete, utilizando diferentes estratégias. Assim, o código fica mais modular, fácil de manter e expandir.

Solução do Exercício 34

Para resolver o problema de duplicação de código e facilitar a adição de novos métodos de pagamento no sistema de e-commerce, podemos criar uma classe abstrata PagamentoTemplate usando o padrão Template Method. Esta classe conterá a estrutura do algoritmo de processamento de pagamento e delegará a implementação das etapas específicas às subclasses. A seguir, apresenta-se a solução proposta.

Primeiro, definimos a classe abstrata PagamentoTemplate. Esta classe contém o método template processarPagamento() que chama as etapas comuns do processamento de pagamento e delega a implementação de algumas etapas às subclasses.

```
public abstract class PagamentoTemplate {
  public final void processarPagamento(Pagamento pagamento) {
    validarPagamento(pagamento);
```

```
        iniciarTransacao(pagamento);
        executarTransacao(pagamento);
        finalizarTransacao(pagamento);
    }

    protected void validarPagamento(Pagamento pagamento) {
        System.out.println("Validando pagamento...");
    }

    protected void iniciarTransacao(Pagamento pagamento) {
        System.out.println("Iniciando transação...");
    }

    protected abstract void executarTransacao(Pagamento pagamento);

    protected void finalizarTransacao(Pagamento pagamento) {
        System.out.println("Finalizando transação...");
    }
}
```

Em seguida, criamos as subclasses PagamentoPayPal, PagamentoCartaoCredito e PagamentoCriptomoeda, que estendem PagamentoTemplate e implementam o método executarTransacao(), específico para cada tipo de pagamento.

```
public class PagamentoPayPal extends PagamentoTemplate {
    @Override
    protected void executarTransacao(Pagamento pagamento) {
        System.out.println("Executando transação PayPal...");
    }
}

public class PagamentoCartaoCredito extends PagamentoTemplate {
    @Override
```

```
  protected void executarTransacao(Pagamento pagamento) {
    System.out.println("Executando transação com Cartão de Crédito...");
  }
}

public class PagamentoCriptomoeda extends PagamentoTemplate {
  @Override
  protected void executarTransacao(Pagamento pagamento) {
    System.out.println("Executando transação com Criptomoeda...");
  }
}
```

Finalmente, criamos a classe Principal para demonstrar o uso do sistema de processamento de pagamentos. Na classe Principal, instanciamos objetos PagamentoPayPal, PagamentoCartaoCredito e PagamentoCriptomoeda e chamamos o método processarPagamento() para cada um.

```
public class Principal {
  public static void main(String[] args) {
    Pagamento pagamento = new Pagamento();

    PagamentoTemplate pagamentoPayPal = new PagamentoPayPal();
    pagamentoPayPal.processarPagamento(pagamento);

    PagamentoTemplate pagamentoCartaoCredito = new PagamentoCartaoCredito();
    pagamentoCartaoCredito.processarPagamento(pagamento);

    PagamentoTemplate pagamentoCriptomoeda = new PagamentoCriptomoeda();
    pagamentoCriptomoeda.processarPagamento(pagamento);
  }
}
```

Esta solução aplica o padrão Template Method para evitar a duplicação de código e facilita a adição de novos métodos de pagamento no futuro. A estrutura do algoritmo é definida na classe

base PagamentoTemplate, enquanto as subclasses fornecem implementações específicas para cada tipo de pagamento. Isso melhora a modularidade e a manutenção do código.

Solução do Exercício 35

Para resolver o problema de complexidade e violação do princípio de responsabilidade única na classe GerenciadorDocumento, podemos aplicar o Princípio OCP e criar uma interface IProcessaDocumento. Esta interface conterá um método executar(Documento documento), e cada classe específica, como ArmazenamentoDocumento, FormatacaoDocumento e ValidacaoDocumento, implementará esta interface.

Primeiro, definimos a interface IProcessaDocumento com o método executar(Documento documento).

```
public interface IProcessaDocumento {
  void executar(Documento documento);
}
```

Depois, criamos as classes específicas que implementam esta interface, cada uma focada em uma responsabilidade única: validação, formatação e armazenamento do documento.

```
public class ValidacaoDocumentoImpl implements IProcessaDocumento {
  @Override
  public void executar(Documento documento) {
     System.out.println("Validando documento...");
  }
}

public class FormatacaoDocumentoImpl implements IProcessaDocumento {
  @Override
  public void executar(Documento documento) {
     System.out.println("Formatando documento...");
  }
}
```

```
public class ArmazenamentoDocumentoImpl implements IProcessaDocumento {
  @Override
  public void executar(Documento documento) {
    System.out.println("Armazenando documento...");
  }
}
```

A classe entidade Documento representa o documento que está sendo processado. Pode ser uma classe simples com alguns atributos relevantes e responsabilidade única de manter informações do documento.

```
public class Documento {
  private String conteudo;

  public Documento(String conteudo) {
    this.conteudo = conteudo;
  }

  public String getConteudo() {
    return conteudo;
  }

  public void setConteudo(String conteudo) {
    this.conteudo = conteudo;
  }
}
```

Finalmente, modificamos a classe GerenciadorDocumentos para delegar as responsabilidades de processamento às classes específicas através da interface IProcessaDocumento.

```
public class GerenciadorDocumento {
  private Documento documento;
  private List<IProcessaDocumento> processadores = new ArrayList<>();
```

```
  public GerenciadorDocumento(Documento documento) {
    this.documento = documento;
  }

  public void adicionarProcessador(IProcessaDocumento processador) {
    processadores.add(processador);
  }

  public void processarDocumento() {
    for (IProcessaDocumento processador : processadores) {
      processador.executar(documento);
    }
  }
}
```

Para demonstrar o uso da solução, criamos a classe Principal que instancia um objeto da classe Documento, o GerenciadorDocumento e adiciona os processadores de todas as estratégias que foram codificadas nesta resolução. Após isso é chamado o método processarDocumento do gerenciador.

```
public class Principal {
  public static void main(String[] args) {
    Documento documento = new Documento("Conteúdo do documento");

    GerenciadorDocumento gerenciador = new GerenciadorDocumento(documento);
    gerenciador.adicionarProcessador(new ValidacaoDocumentoImpl());
    gerenciador.adicionarProcessador(new FormatacaoDocumentoImpl());
    gerenciador.adicionarProcessador(new ArmazenamentoDocumentoImpl());

    gerenciador.processarDocumento();
  }
}
```

Esta solução mantém a lógica de processamento em classes separadas, respeitando o princípio de responsabilidade única e evitando o antipadrão "God Object". A classe GerenciadorDocumentos agora delega as responsabilidades de processamento para as classes específicas, tornando o código mais modular e fácil de manter.

Solução do Exercício 36

Para resolver a violação das pré-condições ilustradas, é necessário reestruturar a relação entre a classe base AnaliseMusica e a subclasse AnaliseMusicaAvancada de modo a garantir a conformidade com o Princípio de Substituição de Liskov.

Inicialmente, a classe AnaliseMusica deve manter seu método validarDados inalterado, garantindo que apenas verifica se o número de reproduções é não negativo, o que estabelece uma base genérica para validação que se aplica a todos os tipos de análises de músicas.

A fim de lidar com a necessidade específica de AnaliseMusicaAvancada, onde uma análise mais detalhada é necessária para músicas com um número significativo de reproduções, recomenda-se a introdução de um novo método específico para essa classe. Este novo método, que poderia ser denominado validarDadosAvancados, adicionaria as verificações adicionais sem modificar o comportamento do método validarDados herdado da classe base. Assim, AnaliseMusicaAvancada mantém a funcionalidade original de validarDados e adiciona uma nova camada de validação por meio de validarDadosAvancados.

Classe base AnaliseMusica:

```
public abstract class AnaliseMusica {
  public final void analisar(int reproducoes) throws IllegalArgumentException {
    if (!validarDados(reproducoes)) {
      throw new IllegalArgumentException("Número de reproduções inválido.");
    }
    executarAnalise(reproducoes);
  }

  protected boolean validarDados(int reproducoes) {
    return reproducoes >= 0;
```

```
    }

    protected abstract void executarAnalise(int reproducoes);
}
```

Classe derivada AnaliseMusicaAvancada:

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class AnaliseMusicaAvancada extends AnaliseMusica {
    private LocalDate dataLancamento;

    public AnaliseMusicaAvancada(LocalDate dataLancamento) {
        this.dataLancamento = dataLancamento;
    }

    public void analisarAvancado(int reproducoes) throws IllegalArgumentException {
        if (!validarDados(reproducoes) || !validarDadosAvancados(reproducoes)) {
            throw new IllegalArgumentException("Número de reproduções inválido.");
        }
        executarAnalise(reproducoes);
    }

    private boolean validarDadosAvancados(int reproducoes) {
        return reproducoes > 100;
    }

    @Override
    protected void executarAnalise(int reproducoes) {
        double indicePopularidade = calcularIndicePopularidade(reproducoes);
        System.out.println("Realizando análise detalhada para música com " + reproducoes + "
reproduções.");
        System.out.println("Índice de popularidade: " + indicePopularidade);
```

```
    }

    private double calcularIndicePopularidade(int reproducoes) {
        long diasDesdeLancamento = ChronoUnit.DAYS.between(dataLancamento,
LocalDate.now());
        return reproducoes / (diasDesdeLancamento + 1.0) * 100.0; // Fator de escala para
normalizar com base no tempo
    }
}
```

Neste código, a classe AnaliseMusicaAvancada utiliza a data de lançamento para ajustar o número de reproduções pela proximidade do lançamento, calculando um índice de popularidade que leva em conta não apenas o volume de reproduções mas também o tempo desde o lançamento, oferecendo uma análise mais refinada e contextualizada das músicas. Além disso, a introdução do método validarDadosAvancados permite que a classe mantenha a conformidade com o Princípio de Substituição de Liskov, ao mesmo tempo, adiciona a verificação específica necessária para a análise avançada sem comprometer a generalidade da validação básica herdada da classe base.

Classe principal AplicacaoAnaliseMusica:

```
import java.time.LocalDate;
import java.util.Arrays;

public class AplicacaoAnaliseMusica {
    public static void main(String[] args) {
        AnaliseMusica analiseAvancada = new AnaliseMusicaAvancada(LocalDate.of(2020, 1, 1));
        // Supõe-se que a música tem 150 reproduções, o que é válido para a análise avançada
        try {
            analiseAvancada.analisar(150);
        } catch (IllegalArgumentException e) {
            System.err.println(e.getMessage());
        }

        // Exemplo de análise básica
```

```
        List<Integer> historicoReproducoes = Arrays.asList(120, 130, 125, 140, 135);
        AnaliseMusica analiseBasica = new AnaliseMusicaBasica(historicoReproducoes);
        // A análise básica não tem restrições adicionais além de não negatividade
        try {
            analiseBasica.analisar(130);
        } catch (IllegalArgumentException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Neste exemplo, AnaliseMusicaAvancada é inicializada com a data de lançamento, e uma análise é executada com 150 reproduções, suficiente para passar pela validação avançada que requer mais de 100 reproduções. Por outro lado, AnaliseMusicaBasica é inicializada com um histórico de reproduções para calcular a estabilidade da popularidade ao longo do tempo e também realiza sua análise sem restrições adicionais de número mínimo de reproduções.

Classe AnaliseMusicaBasica:

```
import java.util.List;

public class AnaliseMusicaBasica extends AnaliseMusica {
    private List<Integer> historicoReproducoes;

    public AnaliseMusicaBasica(List<Integer> historicoReproducoes) {
        this.historicoReproducoes = historicoReproducoes;
    }

    @Override
    protected void executarAnalise(int reproducoes) {
        double indiceEstabilidade = calcularIndiceEstabilidade();
        System.out.println("Realizando análise

básica para música com " + reproducoes + " reproduções.");
```

```
        System.out.println("Índice de estabilidade: " + indiceEstabilidade);
    }

    private double calcularIndiceEstabilidade() {
        if (historicoReproducoes.isEmpty()) return 0.0;
        double media = historicoReproducoes.stream().mapToDouble(a -> a).average().orElse(0.0);
        double desvioPadrao = Math.sqrt(historicoReproducoes.stream().mapToDouble(a ->
Math.pow(a - media, 2)).sum() / historicoReproducoes.size());
        return desvioPadrao / media * 100.0;  // Calcula o coeficiente de variação como índice de
estabilidade
    }
}
```

Neste código, a classe AnaliseMusicaBasica calcula o índice de estabilidade como o coeficiente de variação, ou seja, o desvio padrão dividido pela média, multiplicado por 100 para converter em porcentagem. Este índice quantifica a variação relativa no número de reproduções, oferecendo uma visão sobre quão consistentemente uma música manteve sua popularidade ao longo do tempo. A utilização de uma lista de histórico de reproduções permite essa análise sem alterar as premissas da validação inicial, mantendo a classe compatível e substituível com a classe base AnaliseMusica.

Esta abordagem reforça a flexibilidade do serviço de análise que está sendo projetado, onde diferentes subclasses podem ser desenvolvidas e utilizadas sem conflitos ou violações dos princípios de design orientado a objetos, garantindo a modularidade e extensibilidade do sistema de análise.

Solução do Exercício 37

Para resolver o problema de complexidade e violação do Princípio de Responsabilidade Única (SRP) no método de cálculo do preço da classe Produto, podemos aplicar o padrão Decorator. Este padrão nos permitirá compor dinamicamente comportamentos adicionais ao cálculo do preço, tornando o código mais modular e extensível.

Primeiro, definimos a interface ICalculadoraPreco que conterá o método calcularPreco.

```
public interface ICalculadoraPreco {
```

```
  double calcularPreco(Produto produto);
}
```

Depois, criamos a classe base CalculadoraPrecoBaseImpl que implementa a interface ICalculadoraPreco e retorna o preço base do produto.

```
public class CalculadoraPrecoBaseImpl implements ICalculadoraPreco {
  @Override
  public double calcularPreco(Produto produto) {
    return produto.getPrecoBase();
  }
}
```

A seguir, definimos as classes concretas para cada regra de cálculo que inclui desconto, taxa adicional e promoção, todas implementando a interface ICalculadoraPreco e recebendo uma instância dessa interface para compor a lógica de cálculo.

```
public class CalculadoraDescontoDecorator implements ICalculadoraPreco {
  private ICalculadoraPreco calculadoraPreco;

  public CalculadoraDescontoDecorator(ICalculadoraPreco calculadoraPreco) {
    this.calculadoraPreco = calculadoraPreco;
  }

  @Override
  public double calcularPreco(Produto produto) {
    double preco = calculadoraPreco.calcularPreco(produto);
    if(produto.getPorcentagemDesconto() > 0) {
      preco -= (produto.getPrecoBase() * produto.getPorcentagemDesconto() / 100);
    }
    return preco;
  }
}
```

```
public class CalculadoraTaxaDecorator implements ICalculadoraPreco {
  private ICalculadoraPreco calculadoraPreco;

  public CalculadoraTaxaDecorator(ICalculadoraPreco calculadoraPreco) {
     this.calculadoraPreco = calculadoraPreco;
  }

  @Override
  public double calcularPreco(Produto produto) {
     double preco = calculadoraPreco.calcularPreco(produto);
     if(produto.getTaxaAdicional() > 0) {
        preco += (produto.getPrecoBase() * produto.getTaxaAdicional() / 100);
     }
     return preco;
  }
}

public class CalculadoraPromocaoDecorator implements ICalculadoraPreco {
  private ICalculadoraPreco calculadoraPreco;

  public CalculadoraPromocaoDecorator(ICalculadoraPreco calculadoraPreco) {
     this.calculadoraPreco = calculadoraPreco;
  }

  @Override
  public double calcularPreco(Produto produto) {
     double preco = calculadoraPreco.calcularPreco(produto);
     if(produto.isPromocao()) {
        preco /= 2;
     }
     return preco;
  }
```

```
}
```

A classe Produto mantém apenas as informações básicas do produto.

```
public class Produto {
  private String descricao;
  private double precoBase;
  private double porcentagemDesconto;
  private double taxaAdicional;
  private boolean promocao;

    public Produto(String descricao, double precoBase, double porcentagemDesconto, double taxaAdicional, boolean promocao) {
      this.descricao = descricao;
      this.precoBase = precoBase;
      this.porcentagemDesconto = porcentagemDesconto;
      this.taxaAdicional = taxaAdicional;
      this.promocao = promocao;
  }

  public String getDescricao() {
      return descricao;
  }

  public double getPrecoBase() {
      return precoBase;
  }

  public double getPorcentagemDesconto() {
      return porcentagemDesconto;
  }

  public double getTaxaAdicional() {
      return taxaAdicional;
```

```
  }

  public boolean isPromocao() {
    return promocao;
  }
}
```

Agora, é criado a classe Principal onde demonstra o uso da solução, criando uma instancia de Produto com os parâmetros fictícios, em seguida cria uma instância de CalculadoraPrecoBase que representa o cálculo do preço base sem nenhuma modificação. Depois, a instancia de Produto é decorada com os diferentes cálculos. Cada novo decorador é construído em torno do decorador anterior, adicionando sua lógica de cálculo. Por fim, é chamado o método calcularPreco que aplica todas as regras ao preço base do produto.

```
public class Principal {
  public static void main(String[] args) {
    Produto produto = new Produto("Produto A", 100.0, 10.0, 5.0, true);

    ICalculadoraPreco calculadora = new CalculadoraPrecoBase();
    calculadora = new CalculadoraDescontoDecorator(calculadora);
    calculadora = new CalculadoraTaxaDecorator(calculadora);
    calculadora = new CalculadoraPromocaoDecorator(calculadora);

    double precoFinal = calculadora.calcularPreco(produto);
    System.out.println("Preço final: " + precoFinal);
  }
}
```

Solução do Exercício 38

Para resolver esse problema, devemos refatorar o código para eliminar a redundância e garantir a manutenibilidade do sistema. Uma solução possível seria utilizar a interface "IMusicaBusiness" para definir a busca e o acesso às músicas, e as classes "PlaylistService" e

"UsuarioService" devem utilizar uma implementação dessa interface, evitando assim a redundância de código e facilitando a manutenção do sistema.

Um exemplo de código refatorado, implementando a solução acima e seguindo as ideias da arquitetura limpa, é o seguinte:

```
public interface IMusicaBusiness {
    List<Musica> buscarMusicas(String termo);
    Musica obterMusica(int idMusica);
}

public class MusicaBusiness implements IMusicaBusiness {
    @Override
    public List<Musica> buscarMusicas(String termo) {
        // lógica de busca de músicas
    }

    @Override
    public Musica obterMusica(int idMusica) {
        // lógica de obtenção de música
    }
}

public class PlaylistService {
    private IMusicaBusiness musicaBusiness;

    public PlaylistService(IMusicaBusiness musicaBusiness) {
        this.musicaBusiness = musicaBusiness;
    }

    public void adicionarMusica(int idPlaylist, int idMusica) {
        // lógica de adição de música à playlist
    }
```

```
    public void removerMusica(int idPlaylist, int idMusica) {
        // lógica de remoção de música da playlist
    }

    public int calcularDuracaoTotal(int idPlaylist) {
        List<Musica> musicas = buscarMusicasDaPlaylist(idPlaylist);
        int duracaoTotal = 0;
        for (Musica musica : musicas) {
            duracaoTotal += musica.getDuracao();
        }
        return duracaoTotal;
    }

    public void gerarCapaPlaylist(int idPlaylist) {
        // lógica de geração da capa da playlist
    }

    public void compartilharPlaylist(int idPlaylist, int idUsuario) {
        // lógica de compartilhamento da playlist com outro usuário
    }

    private List<Musica> buscarMusicasDaPlaylist(int idPlaylist) {
        // lógica de busca das músicas da playlist
    }
}

public class UsuarioService {
    private IMusicaBusiness musicaBusiness;

    public UsuarioService(IMusicaBusiness musicaBusiness) {
        this.musicaBusiness = musicaBusiness;
```

```
    }

    public void criarUsuario(String nome, String email, String senha) {
        // lógica de criação de novo usuário
    }

    public void atualizarInformacoesUsuario(int idUsuario, String novoNome, String novoEmail) {
        // lógica de atualização das informações do usuário
    }

    public void adicionarAmigo(int idUsuario, int idAmigo) {
        // lógica de adição de amigo
    }

    public void seguirUsuario(int idUsuario, int idUsuarioSeguido) {
        // lógica de seguimento de outro usuário
    }

    public List<Musica> buscarMusicas(String termo) {
        return musicaBusiness.buscarMusicas(termo);
    }
}
```

Nesse exemplo de código refatorado, criamos uma interface "IMusicaBusiness" que possui os métodos "buscarMusicas()" e "obterMusica()", que são utilizados pelas classes "PlaylistService" e "UsuarioService". Dessa forma, evitamos a redundância de código e facilitamos a manutenção do sistema, pois qualquer alteração na busca ou no acesso às músicas deve ser feita apenas na implementação da interface "IMusicaBusiness". Além disso, cada classe tem uma responsabilidade única e bem definida, atendendo ao Princípio da Responsabilidade Única (SRP). Também seguimos as ideias da arquitetura limpa, separando as regras de negócio em camadas distintas, como a camada de serviço e a camada de negócio, e utilizando interfaces para definir as dependências entre as camadas.

Solução do Exercício 39

Para resolver o problema de carregar um arquivo do disco apenas quando necessário, podemos aplicar o padrão de projeto Proxy. O padrão Proxy nos permite criar um substituto ou marcador para outro objeto, controlando o acesso a ele. Neste caso, usaremos o padrão Proxy para atrasar a carga do arquivo até que ele seja realmente necessário.

Primeiro, definimos uma interface IArquivo que declara o método ler.

```
public interface IArquivo {
  public String ler();
}
```

Depois, modificamos a classe Arquivo para implementar IArquivo e representar o objeto real que realiza a leitura do disco.

```
public class ArquivoImpl implements IArquivo {
  private String conteudo;
  private String nome;

  public ArquivoImpl(String nome) {
    this.nome = nome;
    carregarDoDisco();
  }

  private void carregarDoDisco() {
    System.out.println("Carregando o arquivo " + nome);
    conteudo = "Conteúdo do arquivo";
  }

  @Override
  public String ler() {
    return conteudo;
  }
}
```

Em seguida, criamos a classe ArquivoProxy que também implementa IArquivo e representa o proxy para Arquivo. Este proxy irá atrasar a carga do arquivo até que o método ler seja chamado.

```
public class ArquivoProxy implements IArquivo {
  private ArquivoImpl arquivo;
  private String nome;

  public ArquivoProxy(String nome) {
    this.nome = nome;
  }

  @Override
  public String ler() {
    if (arquivo == null) {
      arquivo = new ArquivoImpl(nome);
    }
    return arquivo.ler();
  }
}
```

Agora, criamos a classe Principal para demonstrar o uso do sistema de arquivos com o proxy. Inicialmente instancia-se um objeto ArquivoProxy, que representa um proxy para o objeto Arquivo. Quando instanciamos ArquivoProxy, o arquivo não é carregado do disco imediatamente. Apenas quando o método ler é chamado, o proxy instancia o objeto Arquivo real e carrega o conteúdo do disco.

```
public class Principal {
  public static void main(String[] args) {
    IArquivo arquivo = new ArquivoProxy("meuarquivo.txt");

    System.out.println("Nome do arquivo: " + arquivo);

    System.out.println("Conteúdo do arquivo: " + arquivo.ler());
  }
```

```
}
```

O padrão Proxy é aplicado aqui para otimizar a operação de leitura do arquivo, adiando o carregamento dispendioso do disco até que seja realmente necessário. Isso nos permite criar objetos de arquivo de forma eficiente, sem gastar recursos do sistema até que a leitura do conteúdo seja necessária.

Esta abordagem mantém o código mais modular e eficiente, seguindo o Princípio de Responsabilidade Única (SRP) ao separar a lógica de carregamento do disco da lógica de acesso ao conteúdo, e o Princípio do Aberto/Fechado (OCP) ao permitir a adição de novas funcionalidades (como o proxy) sem modificar o código existente.

Solução do Exercício 40

Exercício resolvido no caderno.

Solução do Exercício 41

Para resolver o problema de criação de sanduíches usando os padrões de projeto Builder e Decorator, vamos criar um sistema em Java que permita montar sanduíches a partir de ingredientes disponíveis e adicionar ingredientes extras de forma flexível e extensível. O padrão Builder será usado para construir diferentes tipos de sanduíches com os ingredientes básicos específicos de cada tipo, enquanto o padrão Decorator será utilizado para adicionar ingredientes extras aos sanduíches já criados. Vamos implementar dois tipos de sanduíches: Americano e Beirute, e permitiremos a adição de vários ingredientes extras.

Primeiro, criamos a interface ISanduiche que declara os métodos getDescricao e getPreco.

```
public interface ISanduiche {
  public String getDescricao();
  public double getPreco();
}
```

Depois, criamos classes concretas que implementam a interface Sanduiche representando diferentes tipos de sanduíches. Vamos criar os exemplos: AmericanoImpl e BeiruteImpl.

```
public class AmericanoImpl implements ISanduiche {
```

```
  @Override
  public String getDescricao() {
     return "Americano (presunto, queijo, ovo, alface, tomate, maionese)";
  }

  @Override
  public double getPreco() {
     return 7.00;
  }
}

public class BeiruteImpl implements ISanduiche {
  @Override
  public String getDescricao() {
     return "Beirute (pão sírio, rosbife, queijo, alface, tomate, ovo frito)";
  }

  @Override
  public double getPreco() {
     return 10.00;
  }
}
```

Em seguida, criamos a classe abstrata SanduicheDecorator que implementa a interface Sanduiche e delega a responsabilidade para outro objeto Sanduiche.

```
public abstract class SanduicheDecorator implements ISanduiche {
  protected ISanduiche sanduiche;

  public SanduicheDecorator(ISanduiche sanduiche) {
     this.sanduiche = sanduiche;
  }
```

```
  @Override
  public String getDescricao() {
      return sanduiche.getDescricao();
  }

  @Override
  public double getPreco() {
      return sanduiche.getPreco();
  }
}
```

Depois, criamos classes concretas que estendem SanduicheDecorator para adicionar ingredientes específicos, como Ovo, Tomate, e Alface.

```
public class Ovo extends SanduicheDecorator {
  public Ovo(ISanduiche sanduiche) {
      super(sanduiche);
  }

  @Override
  public String getDescricao() {
      return sanduiche.getDescricao() + ", Ovo";
  }

  @Override
  public double getPreco() {
      return sanduiche.getPreco() + 1.50;
  }
}

public class Tomate extends SanduicheDecorator {
  public Tomate(ISanduiche sanduiche) {
      super(sanduiche);
  }
```

```
    @Override
    public String getDescricao() {
        return sanduiche.getDescricao() + ", Tomate";
    }

    @Override
    public double getPreco() {
        return sanduiche.getPreco() + 0.75;
    }
}

public class Alface extends SanduicheDecorator {
    public Alface(ISanduiche sanduiche) {
        super(sanduiche);
    }

    @Override
    public String getDescricao() {
        return sanduiche.getDescricao() + ", Alface";
    }

    @Override
    public double getPreco() {
        return sanduiche.getPreco() + 0.50;
    }
}
```

Agora, criamos a interface ISanduicheBuilder que define o método para construir sanduíches.

```
public interface ISanduicheBuilder {
    public ISanduiche build();
}
```

Depois, criamos classes concretas que implementam ISanduicheBuilder para construir tipos específicos de sanduíches, como AmericanoBuilder e BeiruteBuilder.

```
public class AmericanoBuilder implements ISanduicheBuilder {
  @Override
  public ISanduiche build() {
     return new AmericanoImpl();
  }
}

public class BeiruteBuilder implements ISanduicheBuilder {
  @Override
  public ISanduiche build() {
     return new BeiruteImpl();
  }
}
```

Finalmente, criamos a classe Principal para demonstrar o uso da solução proposta. Inicialmente criamos um objeto instância de AmericanoBuilder. Usamos esse builder para construir um objeto do tipo ISanduiche chamado americano. Em seguida, decoramos o sanduíche americano adicionando os ingredientes extras "Ovo" e "Tomate", utilizando os decoradores correspondentes. O sanduíche é primeiro decorado com Ovo e, depois, com Tomate. Em seguida, imprimimos a descrição e o preço total do sanduíche americano decorado.

Repetimos o processo para o sanduíche beirute. Criamos um objeto instância de BeiruteBuilder e usamos este builder para construir um objeto do tipo ISanduiche chamado beirute. Decoramos o sanduíche beirute adicionando os ingredientes extras "Alface" e "Tomate". Primeiro, o sanduíche é decorado com Alface e, depois, com Tomate. Finalmente, imprimimos a descrição e o preço total do sanduíche beirute decorado.

```
public class Principal {
  public static void main(String[] args) {
     ISanduicheBuilder americanoBuilder = new AmericanoBuilder();
     ISanduiche americano = americanoBuilder.build();
```

```
        americano = new Ovo(americano);
        americano = new Tomate(americano);
        System.out.println("Sanduíche: " + americano.getDescricao());
        System.out.println("Preço: R$ " + americano.getPreco());

        ISanduicheBuilder beiruteBuilder = new BeiruteBuilder();
        ISanduiche beirute = beiruteBuilder.build();
        beirute = new Alface(beirute);
        beirute = new Tomate(beirute);
        System.out.println("Sanduíche: " + beirute.getDescricao());
        System.out.println("Preço: R$ " + beirute.getPreco());
    }
}
```

Nesta solução, utilizamos os padrões de projeto Builder e Decorator para permitir a criação e modificação de sanduíches de maneira flexível e extensível. O padrão Builder é aplicado para construir sanduíches básicos com seus ingredientes específicos, enquanto o padrão Decorator é usado para adicionar ingredientes extras aos sanduíches já construídos, respeitando o princípio de responsabilidade única e o princípio aberto/fechado.

Solução do Exercício 42

a)

Para resolver o problema usando o padrão de projeto Composite, vamos criar uma estrutura hierárquica de componentes que pode representar tanto arquivos quanto pastas. No padrão Composite, os objetos são tratados de maneira uniforme, seja ele um arquivo ou uma pasta. Abaixo, temos a implementação detalhada em Java e um exemplo de uso que simula o cenário descrito no enunciado.

Primeiro, definimos a interface IComponente que declara operações comuns para objetos simples (arquivos) e compostos (pastas):

```
public interface IComponente {
    public void imprimir();
```

```
  public double getTamanho();
}
```

Em seguida, implementamos a classe ArquivoImpl, que representa arquivos individuais, e a classe PastaImpl, que pode conter outros arquivos e pastas.

```
public class ArquivoImpl implements IComponente {
  private String nome;
  private double tamanho;

  public ArquivoImpl(String nome, double tamanho) {
    this.nome = nome;
    this.tamanho = tamanho;
  }

  @Override
  public void imprimir() {
    System.out.println(nome + ", tipo: Arquivo, tamanho: " + tamanho);
  }

  @Override
  public double getTamanho() {
    return tamanho;
  }
}

public class PastaImpl implements IComponente {
  private String nome;
  private List<IComponente> componentes = new ArrayList<>();

  public PastaImpl(String nome) {
    this.nome = nome;
  }
```

```
  public void adicionar(IComponente componente) {
    componentes.add(componente);
  }

  @Override
  public void imprimir() {
    System.out.println("Nome da pasta: " + nome + ", tipo: Pasta, Conteúdo: ");
    for (IComponente componente : componentes) {
      componente.imprimir();
    }
  }

  @Override
  public double getTamanho() {
    double tamanhoTotal = 0;
    for (IComponente componente : componentes) {
      tamanhoTotal += componente.getTamanho();
    }
    return tamanhoTotal;
  }
}
```

b)

A seguir, criamos a classe Principal com o método main para demonstrar o uso do padrão Composite. Logo, criamos uma estrutura de pastas e arquivos conforme especificado no enunciado. São criadas instâncias de arquivos e pastas, e adicionamos arquivos às pastas usando o método adicionar. Depois, imprimimos a estrutura das pastas e seus tamanhos totais.

```
public class Principal {
  public static void main(String[] args) {
    IComponente prova1 = new ArquivoImpl("Prova1.docx", 16.1);
    IComponente prova2 = new ArquivoImpl("Prova2.docx", 36.1);
```

```
        IComponente prova3 = new ArquivoImpl("Prova3.docx", 164.1);

        PastaImpl projetoDeSistemas = new PastaImpl("Projeto de Sistemas de Software");
        projetoDeSistemas.adicionar(prova1);
        projetoDeSistemas.adicionar(prova2);

        PastaImpl pasta2013_2 = new PastaImpl("2013-2");
        pasta2013_2.adicionar(projetoDeSistemas);
        pasta2013_2.adicionar(prova3);

        System.out.println("<<");
        projetoDeSistemas.imprimir();
        System.out.println("Tamanho total: " + projetoDeSistemas.getTamanho() + ">>");

        System.out.println("<<");
        pasta2013_2.imprimir();
        System.out.println("Tamanho total: " + pasta2013_2.getTamanho() + ">>");
    }
}
```

A saída do console será a seguinte:

```
<<
Nome da pasta: Projeto de Sistemas de Software, tipo: Pasta, Conteúdo:
Prova1.docx, tipo: Arquivo, tamanho: 16.1
Prova2.docx, tipo: Arquivo, tamanho: 36.1
Tamanho total: 52.2>>
<<
Nome da pasta: 2013-2, tipo: Pasta, Conteúdo:
Nome da pasta: Projeto de Sistemas de Software, tipo: Pasta, Conteúdo:
```

Prova1.docx, tipo: Arquivo, tamanho: 16.1

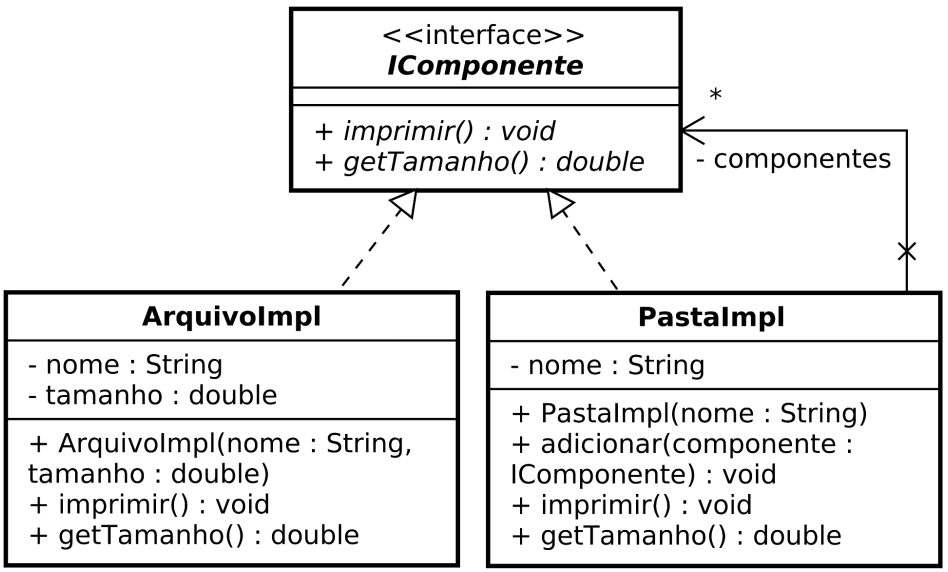
Prova2.docx, tipo: Arquivo, tamanho: 36.1

Prova3.docx, tipo: Arquivo, tamanho: 164.1

Tamanho total: 268.5>>

Neste exemplo, utilizamos o padrão Composite para representar pastas e arquivos. O padrão Composite permite tratar objetos individuais (arquivos) e composições de objetos (pastas) de maneira uniforme, facilitando a adição de novos componentes e a manutenção do código.

c)



Solução do Exercício 43

Para resolver o exercício utilizamos o padrão de projeto Adapter. Inicialmente, criamos a classe Composto, que representa a estrutura básica de um composto químico. Em seguida, implementamos a classe CompostoRico, que atua como um adaptador para enriquecer os compostos básicos com informações adicionais obtidas de um banco de dados químico.

Primeiro, definimos a classe Composto, que representa a estrutura básica de um composto químico e atua como a classe "Target" no contexto do padrão Adapter.

```
public class Composto {
```

```
  protected float pontoEbulicao;
  protected float pontoFusao;
  protected double pesoMolecular;
  protected String formulaMolecular;

  public void exibir() {
    System.out.println("\nComposto: Desconhecido ------");
  }
}
```

Em seguida, implementamos a classe CompostoRico, que atua como um adaptador para enriquecer os compostos básicos com informações adicionais obtidas de um banco de dados químico.

```
public class CompostoRico extends Composto {
  private String quimico;
  private BancoDeDadoQuimico banco;

  public CompostoRico(String quimico) {
    this.quimico = quimico;
  }

  @Override
  public void exibir() {
    banco = new BancoDeDadoQuimico();
    pontoEbulicao = banco.obterPontoCritico(quimico, "E");
    pontoFusao = banco.obterPontoCritico(quimico, "F");
    pesoMolecular = banco.obterPesoMolecular(quimico);
    formulaMolecular = banco.obterEstruturaMolecular(quimico);

    System.out.println("\nComposto: " + quimico + " ------");
    System.out.println(" Fórmula: " + formulaMolecular);
    System.out.println(" Peso: " + pesoMolecular);
```

```
    System.out.println(" Ponto de fusão: " + pontoFusao);
    System.out.println(" Ponto de ebulição: " + pontoEbulicao);
  }
}
```

A classe BancoDeDadosQuimico age como o "Adaptee", fornecendo informações detalhadas sobre os compostos químicos. Utilizamos métodos dessa classe para obter o ponto de fusão, o ponto de ebulição, a fórmula molecular e o peso molecular de cada composto.

```
public class BancoDeDadoQuimico {
  public float obterPontoCritico(String composto, String tipo) {
    if (tipo.equals("F")) {
      switch (composto.toLowerCase()) {
        case "agua": return 0.0f;
        case "benzeno": return 5.5f;
        case "etanol": return -114.1f;
        default: return 0f;
      }
    } else {
      switch (composto.toLowerCase()) {
        case "agua": return 100.0f;
        case "benzeno": return 80.1f;
        case "etanol": return 78.3f;
        default: return 0f;
      }
    }
  }

  public String obterEstruturaMolecular(String composto) {
    switch (composto.toLowerCase()) {
      case "agua": return "H2O";
      case "benzeno": return "C6H6";
      case "etanol": return "C2H5OH";
```

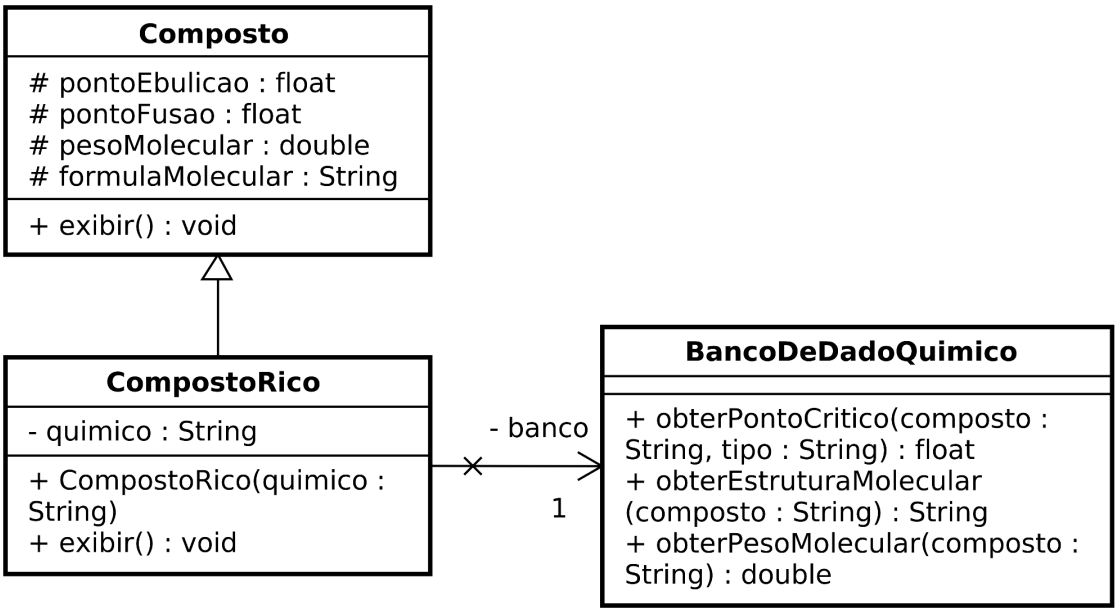
```
            default: return "";
        }
    }

    public double obterPesoMolecular(String composto) {
        switch (composto.toLowerCase()) {
            case "agua": return 18.015;
            case "benzeno": return 78.1134;
            case "etanol": return 46.0688;
            default: return 0d;
        }
    }
}
```

Por fim, criamos a classe Principal com o método main para demonstrar o uso do padrão Adapter, onde instanciamos objetos das classes Composto e CompostoRico, e chamamos o método exibir para cada um deles. A classe CompostoRico utiliza o BancoDeDadoQuimico para obter informações detalhadas sobre os compostos, agindo como um adaptador que transforma a interface da classe BancoDeDadoQuimico em uma interface que a classe Composto pode usar.

```
public class Principal {
    public static void main(String[] args) {
        Composto desconhecido = new Composto();
        desconhecido.exibir();

        Composto agua = new CompostoRico("Agua");
        agua.exibir();
        Composto benzeno = new CompostoRico("Benzeno");
        benzeno.exibir();
        Composto etanol = new CompostoRico("Etanol");
        etanol.exibir();
    }
```

}

Observe o diagrama de classes a seguir da solução proposta.



O padrão de projeto utilizado para resolver esse problema é o Adapter, que nos permite adaptar a interface de uma classe existente para outra interface esperada, sem modificar sua estrutura interna. Isso nos permite adicionar novas funcionalidades sem alterar o código existente, seguindo o princípio Aberto/Fechado (Open/Closed Principle).

Solução do Exercício 44

Para começar, definimos a interface IInvestidor, que representa o contrato para os observadores interessados em receber atualizações sobre as ações. Após isso, codificamos InvestidorImpl que implementa IInvestidor.

```
public interface IInvestidor {
  public void atualizar(Acao acao);
}
public class InvestidorImpl implements IInvestidor {
  private String nome;

  public InvestidorImpl(String nome) {
    this.nome = nome;
```

```
  }

  @Override
  public void atualizar(Acao acao) {
    System.out.printf("Notificado " + this.nome + " sobre a mudança no preço da ação " +
acao.getSimbolo() + " para" + acao.getPreco());
  }
}
```

Em seguida, temos a classe Acao, que representa uma ação genérica e implementa o mecanismo de anexar e notificar investidores:

```
public class Acao {
  private String simbolo;
  private double preco;
  private List<Investidor> investidores = new ArrayList<>();

  public Acao(String simbolo, double preco) {
    this.simbolo = simbolo;
    this.preco = preco;
  }

  public void anexar(Investidor investidor) {
    investidores.add(investidor);
  }

  public void notificarInvestidores() {
    for (Investidor investidor : investidores) {
      investidor.atualizar(this);
    }
  }

  public String getSimbolo() {
```

```
    return simbolo;
  }

  public double getPreco() {
    return preco;
  }

  public void setPreco(double preco) {
    this.preco = preco;
    notificarInvestidores();
  }
}
```

Logo após, temos a classe AcaoIBM, que estende a classe Acao para criar uma ação específica para a IBM:

```
public class AcaoIBM extends Acao {
  public AcaoIBM(String simbolo, double preco) {
    super(simbolo, preco);
  }
}
```

Por fim, a classe Principal cria uma ação para a IBM, instancia investidores e os anexa à ação. Em seguida, ela altera o preço da ação várias vezes para simular mudanças no mercado, notificando os investidores a cada alteração de preço.

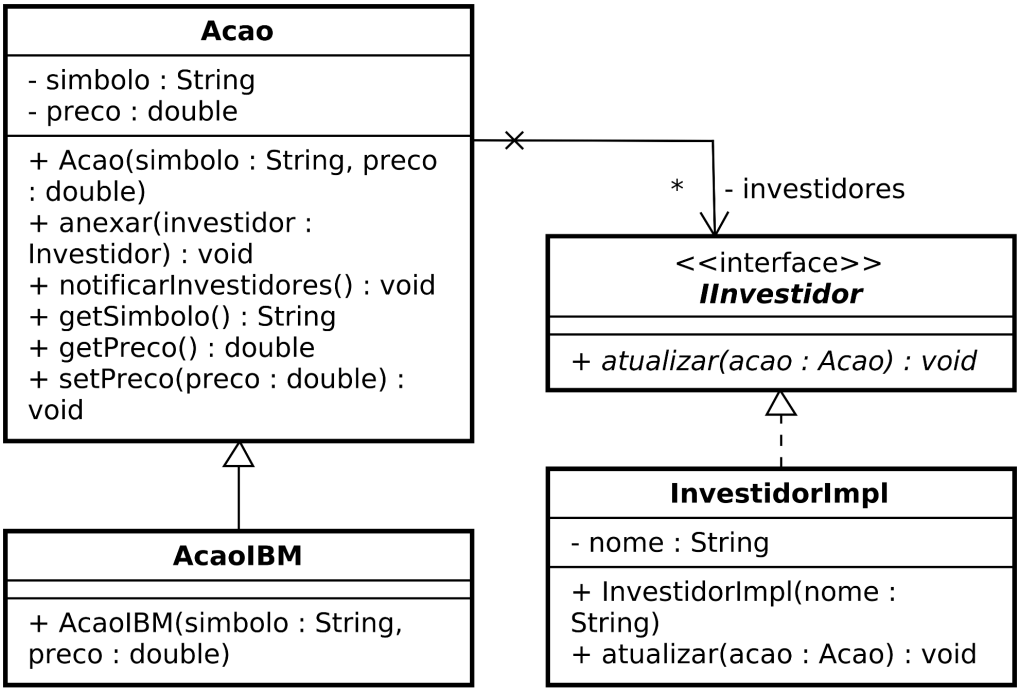
```
public class Principal {
  public static void main(String[] args) {
    AcaoIBM ibm = new AcaoIBM("IBM", 120.00);

    IInvestidor investidor1 = new InvestidorImpl("Sorros");
    IInvestidor investidor2 = new InvestidorImpl("Berkshire");
    ibm.anexar(investidor1);
    ibm.anexar(investidor2);
```

```
        ibm.setPreco(120.10);
        ibm.setPreco(121.00);
        ibm.setPreco(120.50);
        ibm.setPreco(120.75);
    }
}
```

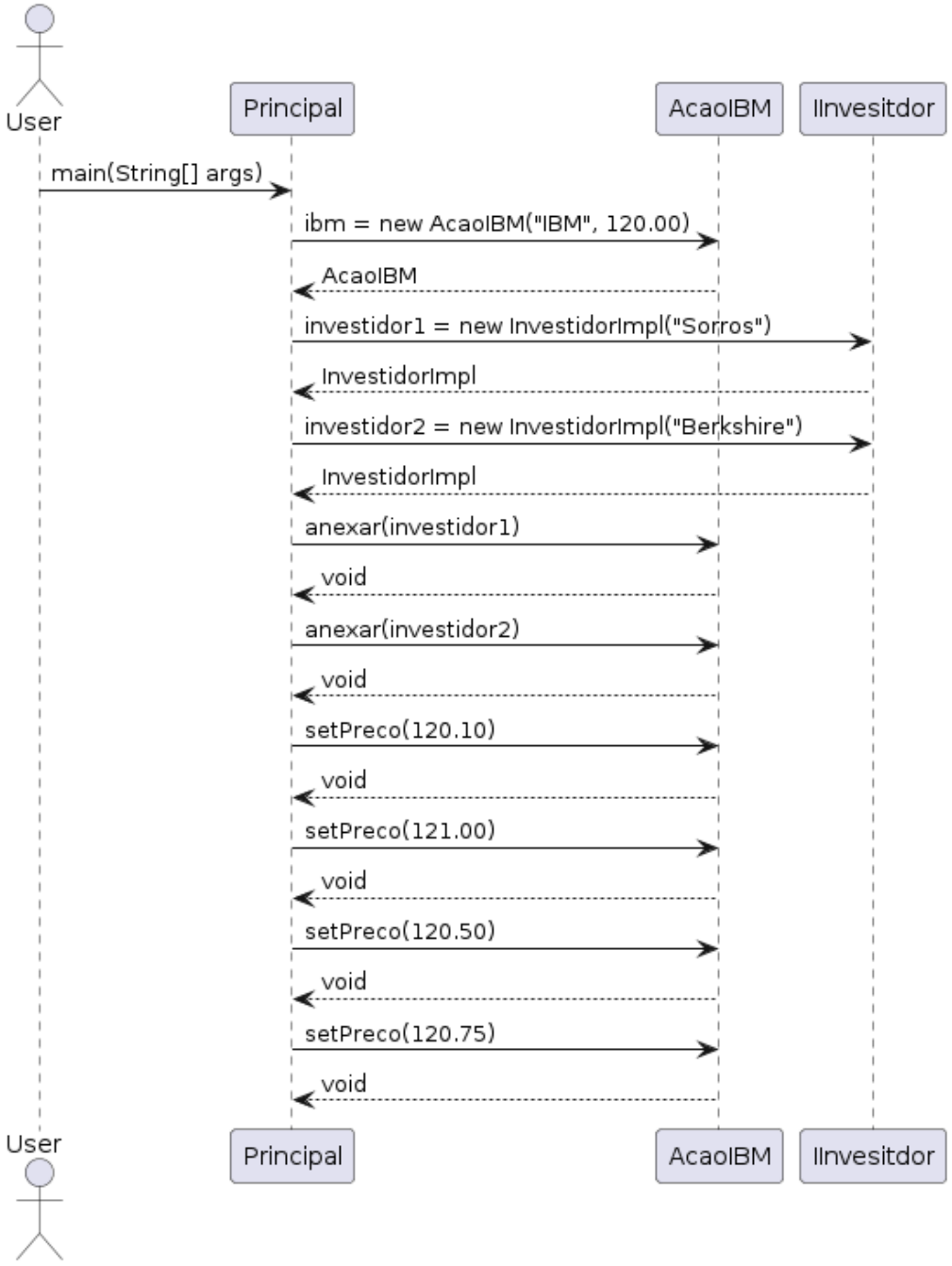
Espera-se que a saída no console seja a seguinte:

```
Notificado Sorros da mudança de IBM para 120.10
Notificado Berkshire da mudança de IBM para 120.10

Notificado Sorros da mudança de IBM para 121.00
Notificado Berkshire da mudança de IBM para 121.00

Notificado Sorros da mudança de IBM para 120.50
Notificado Berkshire da mudança de IBM para 120.50

Notificado Sorros da mudança de IBM para 120.75
Notificado Berkshire da mudança de IBM para 120.75
```

Confira a seguir o diagrama de classes da solução proposta.



Veja o fluxo conforme o diagrama de sequencia.



Solução do Exercício 45

Para resolver o exercício, primeiramente criamos classe abstrata ComponentePrograma que é classe base para todos os componentes da aplicação. Ela define os métodos adicionar, remover e exibir, que serão implementados pelas classes concretas. Esta classe abstrata permite que todas as subclasses compartilhem a mesma interface e possam ser tratadas de forma uniforme.

```
public abstract class ComponentePrograma {
  public abstract void adicionar(ComponentePrograma componente);
  public abstract void remover(ComponentePrograma componente);
  public abstract void exibir();
}
```

A classe Programa estende a ComponentePrograma e representa um programa completo. Ela mantém uma lista de componentes e implementa os métodos para adicionar, remover e exibir esses componentes. O método exibir mostra a estrutura do programa, listando todos os componentes contidos nele.

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

public class Programa extends ComponentePrograma {
  private List<ComponentePrograma> componentes = new ArrayList<>();
  private String nome;

  public Programa(String nome) {
    this.nome = nome;
  }

  @Override
  public void adicionar(ComponentePrograma componente) {
    componentes.add(componente);
  }

  @Override
  public void remover(ComponentePrograma componente) {
    componentes.remove(componente);
  }

  @Override
```

```
  public void exibir() {
    System.out.println("Programa: " + nome);
    for (ComponentePrograma componente : componentes) {
      componente.exibir();
    }
  }
}
```

A classe Metodo também estende a ComponentePrograma e representa um método dentro de uma classe Java. Similar à classe Programa, ela mantém uma lista de componentes e implementa os métodos adicionar, remover e exibir. O método exibir mostra a estrutura da função, listando todos os componentes contidos nela.

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

public class Metodo extends ComponentePrograma {
  private List<ComponentePrograma> componentes = new ArrayList<>();
  private String nome;

  public Metodo(String nome) {
    this.nome = nome;
  }

  @Override
  public void adicionar(ComponentePrograma componente) {
    componentes.add(componente);
  }

  @Override
  public void remover(ComponentePrograma componente) {
    componentes.remove(componente);
  }
```

```
  @Override
  public void exibir() {
      System.out.println("  Método: " + nome);
      for (ComponentePrograma componente : componentes) {
          componente.exibir();
      }
  }
}
```

A classe Classe representa uma classe dentro do programa. Ela também mantém uma lista de componentes e implementa os métodos adicionar, remover e exibir da super classe ComponentePrograma. O método exibir mostra a estrutura da classe, listando todos os componentes contidos nela.

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

public class Classe extends ComponentePrograma {
  private List<ComponentePrograma> componentes = new ArrayList<>();
  private String nome;

  public Classe(String nome) {
      this.nome = nome;
  }

  @Override
  public void adicionar(ComponentePrograma componente) {
      componentes.add(componente);
  }

  @Override
  public void remover(ComponentePrograma componente) {
```

```
        componentes.remove(componente);
    }

    @Override
    public void exibir() {
        System.out.println("Classe: " + nome);
        for (ComponentePrograma componente : componentes) {
            componente.exibir();
        }
    }
}
```

A classe BlocoCodigo representa um bloco de código específico dentro de um método. Ao contrário das outras classes, ela não mantém uma lista de componentes e não implementa os métodos adicionar e remover. O método exibir mostra o conteúdo do bloco de código.

```
public class BlocoCodigo extends ComponentePrograma {
    private String codigo;

    public BlocoCodigo(String codigo) {
        this.codigo = codigo;
    }

    @Override
    public void adicionar(ComponentePrograma componente) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void remover(ComponentePrograma componente) {
        throw new UnsupportedOperationException();
    }
```

```
  @Override
  public void exibir() {
      System.out.println("    Código: " + codigo);
  }
}
```

A classe Main demonstra o funcionamento do padrão Composite na construção de uma estrutura de programa. Ela cria instâncias de Programa, Classe, Metodo e BlocoCodigo, monta a estrutura hierárquica e chama o método exibir para imprimir a estrutura completa do programa.

```
public class Main {
  public static void main(String[] args) {
      ComponentePrograma programa = new Programa("MeuPrograma");

      ComponentePrograma classe = new Classe("MinhaClasse");

      ComponentePrograma metodo1 = new Metodo("meuMetodo1");
      ComponentePrograma metodo2 = new Metodo("meuMetodo2");

      ComponentePrograma bloco1 = new BlocoCodigo("int x = 10;");
      ComponentePrograma bloco2 = new BlocoCodigo("System.out.println(x);");

      metodo1.adicionar(bloco1);
      metodo2.adicionar(bloco2);

      classe.adicionar(metodo1);
      classe.adicionar(metodo2);

      programa.adicionar(classe);

      programa.exibir();
  }
```

```
}
```

A saída esperada após a execução do método main é a seguinte:

```
Programa: MeuPrograma
Classe: MinhaClasse
  Método: meuMetodo1
    Código: int x = 10;
  Método: meuMetodo2
    Código: System.out.println(x);
```

Em outro exemplo de demonstração de uso do padrão de projeto Composite na criação de classes em Java, utilizamos uma estrutura representativa da sintaxe de uma programa. Primeiramente, utilizamos classes para representar componentes como programas, classes, métodos e blocos de código. Em seguida, construímos um exemplo que demonstra a criação da classe MascaraService utilizando o padrão Composite.

Inicialmente, criamos uma instância da classe Programa chamada MascaraServiceProgram. Posteriormente, instanciamos a classe MascaraService e adicionamos um atributo denominado validadores. Definimos então o construtor MascaraService, cuja função é inicializar a lista de validadores e adicionar os validadores ValidadorMascaraCpf e ValidadorMascaraEndereco. Na sequência, implementamos o método processar, responsável por iterar sobre os validadores e aplicar as validações apropriadas.

Para estruturar a classe MascaraService, incluímos os componentes necessários, como atributos, construtor e métodos. Em seguida, adicionamos a classe MascaraService ao programa e exibimos a estrutura completa.

O método exibir do Programa tem a responsabilidade de imprimir a estrutura da classe MascaraService, demonstrando a aplicação do padrão Composite para construir e representar a estrutura sintática de uma classe Java.

```
public class Main {
  public static void main(String[] args) {
    Programa programa = new Programa("MascaraServiceProgram");
```

```
        // Criação da classe MascaraService
        Classe classeMascaraService = new Classe("MascaraService");

        // Atributos da classe
        BlocoCodigo atributoValidadores = new BlocoCodigo("private List<ValidadorMascara>
validadores;");

        // Construtor da classe
        Metodo construtorMascaraService = new Metodo("public MascaraService()");
        BlocoCodigo inicializacaoValidadores = new BlocoCodigo("this.validadores = new
ArrayList<>();");
        BlocoCodigo adicionaValidadorCpf = new BlocoCodigo("this.validadores.add(new
ValidadorMascaraCpf());");
        BlocoCodigo adicionaValidadorEndereco = new BlocoCodigo("this.validadores.add(new
ValidadorMascaraEndereco());");
        BlocoCodigo comentarioAdicionaOutrosValidadores = new BlocoCodigo("// Outros
validadores conforme necessário");

        construtorMascaraService.adicionar(inicializacaoValidadores);
        construtorMascaraService.adicionar(adicionaValidadorCpf);
        construtorMascaraService.adicionar(adicionaValidadorEndereco);
        construtorMascaraService.adicionar(comentarioAdicionaOutrosValidadores);

        // Método processar
        Metodo metodoProcessar = new Metodo("public String processar(String tipoDado, String
valor)");
        BlocoCodigo inicioMetodoProcessar = new BlocoCodigo("for (ValidadorMascara validador :
validadores) {");
        BlocoCodigo condicaoAplica = new BlocoCodigo("if (validador.aplica(tipoDado)) {");
        BlocoCodigo retornoValidacao = new BlocoCodigo("return validador.validar(valor);");
        BlocoCodigo fechaCondicao = new BlocoCodigo("}");
        BlocoCodigo fechaFor = new BlocoCodigo("}");
```

```
        BlocoCodigo retornoPadrao = new BlocoCodigo("return valor;");

        metodoProcessar.adicionar(inicioMetodoProcessar);
        metodoProcessar.adicionar(condicaoAplica);
        metodoProcessar.adicionar(retornoValidacao);
        metodoProcessar.adicionar(fechaCondicao);
        metodoProcessar.adicionar(fechaFor);
        metodoProcessar.adicionar(retornoPadrao);

        // Adiciona elementos à classe
        classeMascaraService.adicionar(atributoValidadores);
        classeMascaraService.adicionar(construtorMascaraService);
        classeMascaraService.adicionar(metodoProcessar);

        // Adiciona a classe ao programa
        programa.adicionar(classeMascaraService);

        // Exibe a estrutura do programa
        programa.exibir();
    }
}
```

A saída esperada após a execução do método main é a seguinte:

```
Programa: MascaraServiceProgram
Classe: MascaraService
  Código: private List<ValidadorMascara> validadores;
 Método: public MascaraService()
  Código: this.validadores = new ArrayList<>();
  Código: this.validadores.add(new ValidadorMascaraCpf());
  Código: this.validadores.add(new ValidadorMascaraEndereco());
  Código: // Outros validadores conforme necessário
 Método: public String processar(String tipoDado, String valor)
  Código: for (ValidadorMascara validador : validadores) {
```

Código: if (validador.aplica(tipoDado)) {

Código: return validador.validar(valor);

Código: }

Código: }

Código: return valor;

Solução do Exercício 46

Para solucionar a questão, primeiro definimos a classe BalanceadorDeCarga que implementa o padrão Singleton, contendo uma lista de servidores disponíveis e usa um gerador de números aleatórios para distribuir as solicitações de serviço entre eles. O método getInstancia() garante que apenas uma instância do balanceador de carga seja criada, utilizando a técnica de "double-checked locking" para suportar ambientes multithread. Esta classe garantirá que apenas uma instância do balanceador de carga seja criada e usada.

```
public class BalanceadorDeCarga {
  private static BalanceadorDeCarga instancia;
  private List<String> servidores = new ArrayList<>();
  private Random random = new Random();
  private static final Object locker = new Object();

  private BalanceadorDeCarga() {
    servidores.add("ServidorI");
    servidores.add("ServidorII");
    servidores.add("ServidorIII");
    servidores.add("ServidorIV");
    servidores.add("ServidorV");
  }

  public static BalanceadorDeCarga getInstancia() {
    if (instancia == null) {
      synchronized (locker) {
        if (instancia == null) {
```

```
                instancia = new BalanceadorDeCarga();
            }
        }
    }
    return instancia;
  }

  public String getServidor() {
    int r = random.nextInt(servidores.size());
    return servidores.get(r);
  }
}
```

Em seguida, criamos a classe Principal com o método main para demonstrar o uso do BalanceadorDeCarga. Portanto, criamos múltiplas referências ao BalanceadorDeCarga e verificamos se todas apontam para a mesma instância. Em seguida, distribuímos várias solicitações de serviço entre os servidores e imprimimos no console o servidor que atende cada solicitação.

```
public class Principal {
  public static void main(String[] args) {
    BalanceadorDeCarga b1 = BalanceadorDeCarga.getInstancia();
    BalanceadorDeCarga b2 = BalanceadorDeCarga.getInstancia();
    BalanceadorDeCarga b3 = BalanceadorDeCarga.getInstancia();
    BalanceadorDeCarga b4 = BalanceadorDeCarga.getInstancia();

    if (b1 == b2 && b2 == b3 && b3 == b4) {
      System.out.println("Mesma instância\n");
    }

    BalanceadorDeCarga balancer = BalanceadorDeCarga.getInstancia();
    for (int i = 0; i < 15; i++) {
      String servidor = balancer.getServidor();
```

```
            System.out.println("Distribuir solicitação para: " + servidor);
        }
    }
}
```

Solução do Exercício 47

Para solucionar a questão, primeiro definimos a classe abstrata Pagina, que representa uma página em um documento. Cada tipo de página será uma subclasse desta classe abstrata.

```
public abstract class Pagina {
    public abstract void exibirDetalhes();
}
```

Agora, criamos várias subclasses de Pagina, cada uma representando um tipo específico de página.

```
public class PaginaHabilidade extends Pagina {
    @Override
    public void exibirDetalhes() {
        System.out.println("Página de Habilidades");
    }
}

public class PaginaEducacao extends Pagina {
    @Override
    public void exibirDetalhes() {
        System.out.println("Página de Educação");
    }
}

public class PaginaExperiencia extends Pagina {
    @Override
    public void exibirDetalhes() {
```

```
        System.out.println("Página de Experiência");
    }
}

public class PaginaIntroducao extends Pagina {
    @Override
    public void exibirDetalhes() {
        System.out.println("Página de Introdução");
    }
}

public class PaginaResultado extends Pagina {
    @Override
    public void exibirDetalhes() {
        System.out.println("Página de Resultados");
    }
}

public class PaginaConclusao extends Pagina {
    @Override
    public void exibirDetalhes() {
        System.out.println("Página de Conclusão");
    }
}

public class PaginaResumo extends Pagina {
    @Override
    public void exibirDetalhes() {
        System.out.println("Página de Resumo");
    }
}
```

```
public class PaginaBibliografia extends Pagina {
  @Override
  public void exibirDetalhes() {
    System.out.println("Página de Bibliografia");
  }
}
```

Em seguida, definimos a classe abstrata Documento, que representa um documento e mantém uma lista de páginas. As subclasses desta classe implementarão o método criarPaginas para adicionar páginas específicas ao documento.

```
public abstract class Documento {
  protected List<Pagina> paginas = new ArrayList<>();

  public Documento() {
    this.criarPaginas();
  }

  public List<Pagina> getPaginas() {
    return paginas;
  }

  public abstract void criarPaginas();
}
```

Agora, criamos subclasses de Documento para tipos específicos de documentos, como currículos e relatórios.

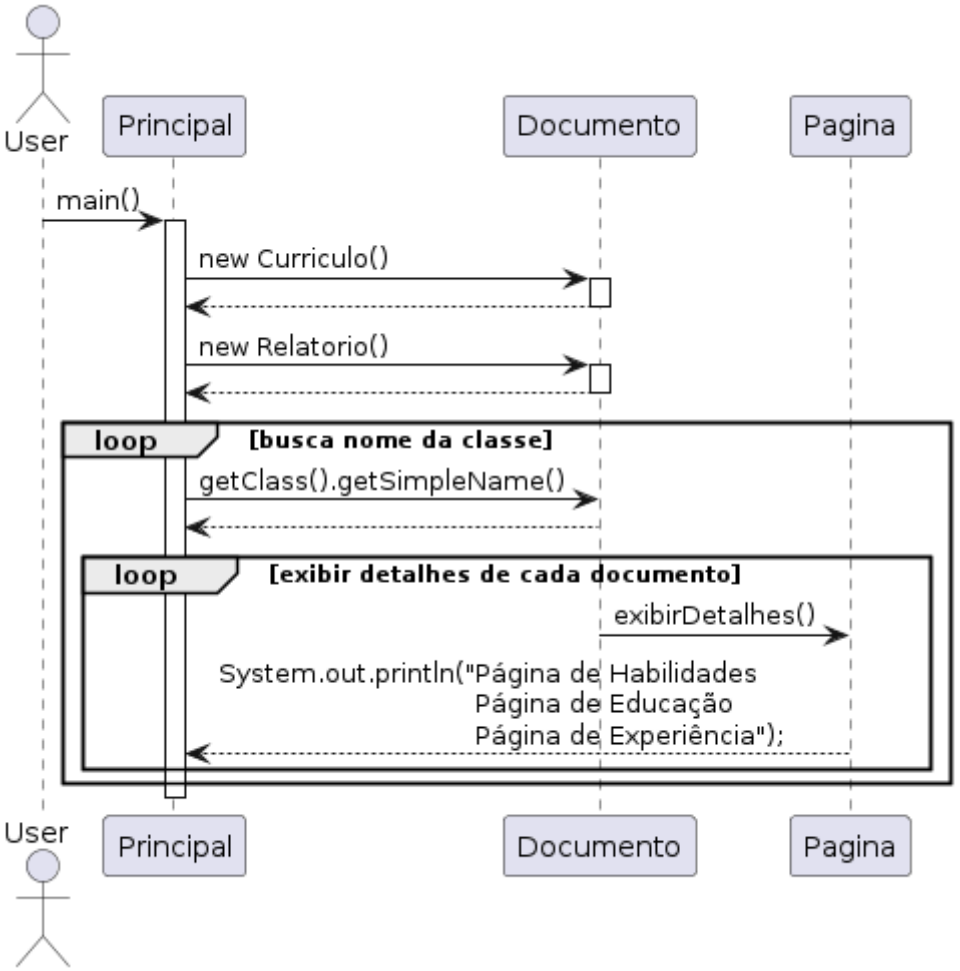
```
public class Curriculo extends Documento {
  @Override
  public void criarPaginas() {
    paginas.add(new PaginaHabilidades());
    paginas.add(new PaginaEducacao());
    paginas.add(new PaginaExperiencia());
```

```
  }
}

public class Relatorio extends Documento {
  @Override
  public void criarPaginas() {
    paginas.add(new PaginaIntroducao());
    paginas.add(new PaginaResultados());
    paginas.add(new PaginaConclusao());
    paginas.add(new PaginaResumo());
    paginas.add(new PaginaBibliografia());
  }
}
```

Por fim, implementamos a classe principal para demonstrar o uso do padrão Factory Method.

```
public class Principal {
  public static void main(String[] args) {
    Documento[] documentos = new Documento[2];
    documentos[0] = new Curriculo();
    documentos[1] = new Relatorio();

    for (Documento documento : documentos) {
      System.out.println("\n" + documento.getClass().getSimpleName() + " --");
      for (Pagina pagina : documento.getPaginas()) {
        pagina.exibirDetalhes();
      }
    }
  }
}
```

Observe o diagrama de sequência abaixo demonstrando o fluxo de execução do sistema a partir da classe Principal.



Este código demonstra como o padrão de projeto Factory Method permite a criação de páginas específicas para diferentes tipos de documentos sem acoplar o código às classes concretas das páginas.

Solução do Exercício 48

Para solucionar a questão, primeiro defini-se a classe abstrata AcessorDeDado, que define o método template executar e declara os métodos abstratos que devem ser implementados pelas subclasses.

```
public abstract class AcessorDeDado {
  public abstract void conectar();
  public abstract void selecionar();
  public abstract void processar(int quantidade);
  public abstract void desconectar();
```

```
  public void executar(int quantidade) {
      conectar();
      selecionar();
      processar(quantidade);
      desconectar();
  }
}
```

Agora, criamos a classe concreta Categoria, que herda de AcessorDeDados e implementa os métodos abstratos.

```
public class Categoria extends AcessorDeDado {
  private List<String> categorias;

  @Override
  public void conectar() {
      categorias = new ArrayList<>();
  }

  @Override
  public void selecionar() {
      categorias.add("Vermelho");
      categorias.add("Verde");
      categorias.add("Azul");
      categorias.add("Amarelo");
      categorias.add("Roxo");
      categorias.add("Branco");
      categorias.add("Preto");
  }

  @Override
  public void processar(int quantidade) {
```

```
        System.out.println("Categorias ---- ");
        for (int i = 0; i < quantidade; i++) {
            System.out.println(categorias.get(i));
        }
        System.out.println();
    }

    @Override
    public void desconectar() {
        categorias.clear();
    }
}
```

Em seguida, criamos a classe concreta Produto, que também herda de AcessorDeDado e implementa os métodos abstratos.

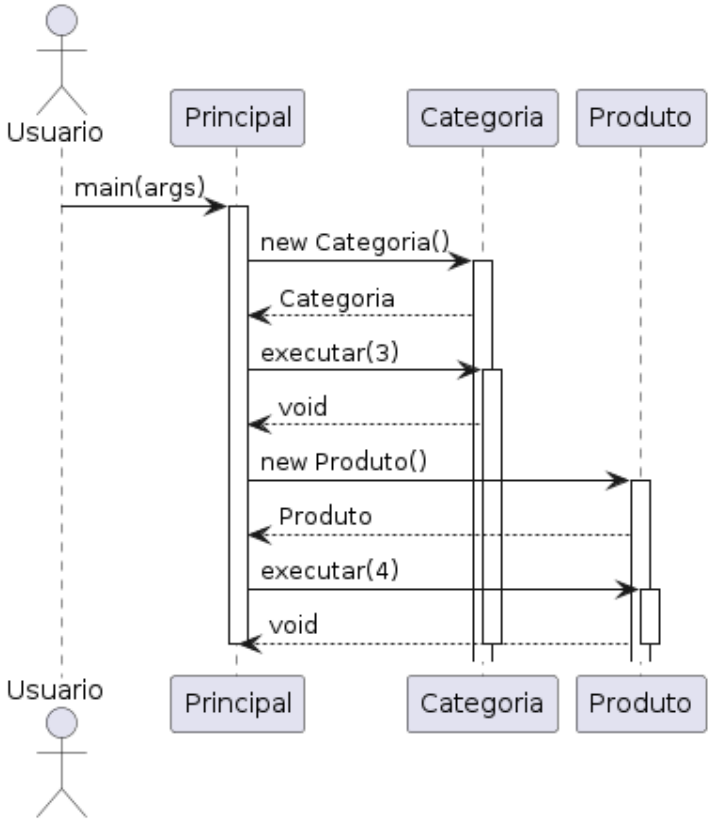
```
public class Produto extends AcessorDeDado {
    private List<String> produtos;

    @Override
    public void conectar() {
        produtos = new ArrayList<>();
    }

    @Override
    public void selecionar() {
        produtos.add("Carro");
        produtos.add("Bicicleta");
        produtos.add("Barco");
        produtos.add("Caminhão");
        produtos.add("Motocicleta");
        produtos.add("Patins");
        produtos.add("Carrinho de bebê");
```

```
  }

  @Override
  public void processar(int quantidade) {
      System.out.println("Produtos ---- ");
      for (int i = 0; i < quantidade; i++) {
          System.out.println(produtos.get(i));
      }
      System.out.println();
  }

  @Override
  public void desconectar() {
      produtos.clear();
  }
}
```

Finalmente, implementamos a classe principal para demonstrar o uso do padrão Template Method.

```
public class Principal {
  public static void main(String[] args) {
      AcessorDeDado categorias = new Categoria();
      categorias.executar(3);

      AcessorDeDado produtos = new Produto();
      produtos.executar(4);
  }
}
```

Observe o diagrama de sequência abaixo demonstrando o fluxo de execução do sistema a partir da classe Principal.



Este código demonstra como o padrão de projeto Template Method permite definir um esqueleto de algoritmo em uma classe base e deixar a implementação de alguns passos para subclasses específicas.

Solução do Exercício 49

Para solucionar esta questão, primeiro definiremos as interfaces e classes necessárias para implementar o padrão Builder. Começaremos com a interface IConstrutorVeiculo, que declara os métodos para construir diferentes partes de um veículo.

```
public interface IConstrutorVeiculo {
  void construirChassi();
  void construirMotor();
  void construirRodas();
  Veiculo obterVeiculo();
}
```

Em seguida, criamos a classe concreta ConstrutorCarroImpl, que implementa a interface IConstrutorVeiculo e define a construção de um carro.

```
public class ConstrutorCarroImpl implements IConstrutorVeiculo {
  private Veiculo carro = new Veiculo();

  @Override
  public void construirChassi() {
    carro.adicionarParte("Chassi de carro");
  }

  @Override
  public void construirMotor() {
    carro.adicionarParte("Motor de carro");
  }

  @Override
  public void construirRodas() {
    carro.adicionarParte("Rodas de carro");
  }

  @Override
  public Veiculo obterVeiculo() {
    return carro;
  }
}
```

A seguir, criamos a classe Veiculo, que representa o produto final a ser construído pelo Builder.

```
public class Veiculo {
  private List<String> partes = new ArrayList<>();

  public void adicionarParte(String parte) {
    partes.add(parte);
  }
```

```
  public void mostrarPartes() {
    for (String parte : partes) {
      System.out.println(parte);
    }
  }
}
```

Agora, implementamos o padrão Decorator. Começamos com a classe VeiculoDecorator, que implementa a interface Veiculo e é usada como classe base para todos os decoradores concretos.

```
public abstract class VeiculoDecorator extends Veiculo {
  protected Veiculo veiculoDecorado;

  public VeiculoDecorator(Veiculo veiculoDecorado) {
    this.veiculoDecorado = veiculoDecorado;
  }

  @Override
  public void mostrarPartes() {
    veiculoDecorado.mostrarPartes();
  }
}
```

Depois, criamos decoradores concretos para adicionar funcionalidades aos veículos. Aqui está um exemplo de decorador para adicionar um sistema de som.

```
public class SistemaDeSomDecorator extends VeiculoDecorator {

  public SistemaDeSomDecorator(Veiculo veiculoDecorado) {
    super(veiculoDecorado);
  }

  @Override
```

```
  public void mostrarPartes() {
    super.mostrarPartes();
    System.out.println("Sistema de som adicionado");
  }
}
```

Outro exemplo de decorador é o RodasEsportivasDecorator, que adiciona rodas esportivas ao veículo.

```
public class RodaEsportivaDecorator extends VeiculoDecorator {

  public RodaEsportivaDecorator(Veiculo veiculoDecorado) {
    super(veiculoDecorado);
  }

  @Override
  public void mostrarPartes() {
    super.mostrarPartes();
    System.out.println("Rodas esportivas adicionadas");
  }
}
```

Finalmente, implementamos a classe principal para demonstrar o uso dos padrões Builder e Decorator.

```
public class Principal {
  public static void main(String[] args) {
    IConstrutorVeiculo construtor = new ConstrutorCarroImpl();
    construtor.construirChassi();
    construtor.construirMotor();
    construtor.construirRodas();
    Veiculo carro = construtor.obterVeiculo();

    Veiculo carroComSom = new SistemaDeSomDecorator(carro);
```

```
        Veiculo carroCompleto = new RodaEsportivaDecorator(carroComSom);

        carroCompleto.mostrarPartes();
    }
}
```

A saída no console será semelhante a esta:

```
Chassi de carro
Motor de carro
Rodas de carro
Sistema de som adicionado
Rodas esportivas adicionadas
```

Este código demonstra como os padrões de projeto Builder e Decorator podem ser usados em conjunto para construir e decorar objetos de forma flexível e extensível.

Solução do Exercício 50

Para solucionar a questão, primeiro define-se as classes necessárias para implementar o padrão Memento. Começamos com a classe Contato, que representa o estado de um contato.

```
public class Contato {
    private String nome;
    private String telefone;

    public Contato(String nome, String telefone) {
        this.nome = nome;
        this.telefone = telefone;
    }

    public String getNome() {
        return nome;
    }
```

```
    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    @Override
    public String toString() {
        return "Contato{" + "nome='" + nome + '\'' + ", telefone='" + telefone + '\'' + '}';
    }
}
```

A seguir, implementamos a classe Memento, que armazena o estado de um objeto Contato.

```
public class Memento {
    private final Contato contato;

    public Memento(Contato contato) {
        this.contato = new Contato(contato.getNome(), contato.getTelefone());
    }

    public Contato getContato() {
        return contato;
    }
}
```

A classe HistoricoContato atua como o "Caretaker" no contexto do padrão Memento, gerenciando a pilha de mementos para permitir operações de desfazer.

```
public class HistoricoContato {
  private Stack<Memento> historico = new Stack<>();

  public void adicionarMemento(Memento memento) {
     historico.push(memento);
  }

  public Memento obterUltimoMemento() {
     if (!historico.isEmpty()) {
        return historico.pop();
     }
     return null;
  }
}
```

Agora, implementamos o padrão Command. Começamos com a interface Comando, que declara o método executar.

```
public interface IComando {
  public void executar();
  public void desfazer();
}
```

Implementamos a classe SalvarContatoComando, que encapsula a operação de salvar um contato.

```
public class SalvarContatoComandoImpl implements IComando {
  private Contato contato;
  private List<Contato> listaContatos;
  private HistoricoContato historico;

  public SalvarContatoComandoImpl(Contato contato, List<Contato> listaContatos,
HistoricoContato historico) {
     this.contato = contato;
```

```
        this.listaContatos = listaContatos;
        this.historico = historico;
    }

    @Override
    public void executar() {
        historico.adicionarMemento(new Memento(contato));
        listaContatos.add(contato);
        System.out.println("Contato salvo: " + contato);
    }

    @Override
    public void desfazer() {
        Memento memento = historico.obterUltimoMemento();
        if (memento != null) {
            listaContatos.remove(contato);
            System.out.println("Salvamento desfeito: " + contato);
        }
    }
}
```

A classe ExcluirContatoComando encapsula a operação de excluir um contato.

```
public class ExcluirContatoComandoImpl implements IComando {
    private Contato contato;
    private List<Contato> listaContatos;
    private HistoricoContato historico;

    public ExcluirContatoComandoImpl(Contato contato, List<Contato> listaContatos,
HistoricoContato historico) {
        this.contato = contato;
        this.listaContatos = listaContatos;
        this.historico = historico;
```

```
    }

    @Override
    public void executar() {
        if (listaContatos.contains(contato)) {
            historico.adicionarMemento(new Memento(contato));
            listaContatos.remove(contato);
            System.out.println("Contato excluído: " + contato);
        }
    }

    @Override
    public void desfazer() {
        Memento memento = historico.obterUltimoMemento();
        if (memento != null) {
            listaContatos.add(memento.getContato());
            System.out.println("Exclusão desfeita: " + memento.getContato());
        }
    }
}
```

Por fim, implementamos a classe principal para demonstrar o uso dos padrões Memento e Command.

```
public class Principal {
    public static void main(String[] args) {
        List<Contato> listaContatos = new ArrayList<>();
        HistoricoContato historico = new HistoricoContato();

        Contato contato1 = new Contato("Alice", "123456789");
        Contato contato2 = new Contato("Bob", "987654321");

        SalvarContatoComandoImpl salvarContato1 = new
```

```
        SalvarContatoComandoImpl(contato1, listaContatos, historico);
    SalvarContatoComandoImpl salvarContato2 = new
        SalvarContatoComandoImpl(contato2, listaContatos, historico);

    salvarContato1.executar();
    salvarContato2.executar();

    System.out.println("Lista de contatos: " + listaContatos);

    ExcluirContatoComandoImpl excluirContato1 = new
      ExcluirContatoComandoImpl(contato1, listaContatos, historico);
    excluirContato1.executar();

    System.out.println("Lista de contatos após exclusão: " + listaContatos);

    excluirContato1.desfazer();

    System.out.println("Lista de contatos após desfazer exclusão: " + listaContatos);
  }
}
```

A saída no console será semelhante a esta:

```
Contato salvo: Contato{nome='Alice', telefone='123456789'}
Contato salvo: Contato{nome='Bob', telefone='987654321'}
Lista de contatos: [Contato{nome='Alice', telefone='123456789'}, Contato{nome='Bob', telefone='987654321'}]
Contato excluído: Contato{nome='Alice', telefone='123456789'}
Lista de contatos após exclusão: [Contato{nome='Bob', telefone='987654321'}]
Exclusão desfeita: Contato{nome='Alice', telefone='123456789'}
Lista de contatos após desfazer exclusão: [Contato{nome='Bob', telefone='987654321'}, Contato{nome='Alice', telefone='123456789'}]
```

Este código demonstra como os padrões de projeto Memento e Command podem ser usados em conjunto para implementar a funcionalidade de desfazer (UNDO) em operações de salvar e excluir contatos.

Solução do Exercício 51

Para solucionar esta questão, primeiro organiza-se o código aplicando os princípios SOLID e o padrão Strategy para a geração de números. Refatora a geração e validação de CPFs, CNPJs e RGs em classes separadas e aplicaremos o padrão Strategy para a geração de números.

Inicialmente é criada uma interface IGeradorDocumento e posteriormente implementamos diferentes estratégias para CPF, CNPJ e RG.

```
public interface IGeradorDocumento {
  public String gerar(boolean comPontos);
  public boolean validar(String documento);
}
```

Cada gerador específico de CPF, CNPJ e RG, será responsável por implementar a lógica de geração e validação de seu respectivo documento, encapsulando essa lógica e permitindo que diferentes tipos de documentos sejam tratados de forma polimórfica.

Para a implementação do CPF, criamos a classe GeradorCPFImpl que implementa a interface IGeradorDocumento. Esta classe é responsável por gerar e validar CPFs, seguindo as regras específicas para esse tipo de documento.

```
public class GeradorCPFImpl implements IGeradorDocumento {
  @Override
  public String gerar(boolean comPontos) {
     System.out.println("Gerando CPF...");
     return "CPF fictício gerado";
  }

  @Override
  public boolean validar(String CPF) {
     return true;
```

```
  }
}
```

Para a implementação do CNPJ, criamos a classe GeradorCNPJImpl que também implementa a interface IGeradorDocumento. Esta classe é responsável por gerar e validar CNPJs.

```
public class GeradorCNPJImpl implements IGeradorDocumento {
  @Override
  public String gerar(boolean comPontos) {
    System.out.println("Gerando CNPJ...");
    return "CNPJ fictício gerado";
  }

  @Override
  public boolean validar(String CNPJ) {
    return true;
  }
}
```

Para a implementação do RG, criamos a classe GeradorRGImpl, que também implementa a interface IGeradorDocumento. Esta classe é responsável por gerar e validar RGs.

```
public class GeradorRGImpl implements IGeradorDocumento {
  @Override
  public String gerar(boolean comPontos) {
    System.out.println("Gerando RG...");
    return "RG fictício gerado";
  }

  @Override
  public boolean validar(String RG) {
    return true;
  }
}
```

Criamos uma fábrica para instanciar os diferentes geradores de documentos utilizando o padrão Factory Method:

```
public class GeradorDocumentoFactory {
  public IGeradorDocumento criarGerador(String tipo) {
    switch (tipo) {
      case "CPF":
        return new GeradorCPFImpl();
      case "CNPJ":
        return new GeradorCNPJImpl();
      case "RG":
        return new GeradorRGImpl();
      default:
        throw new IllegalArgumentException("Tipo de documento desconhecido");
    }
  }
}
```

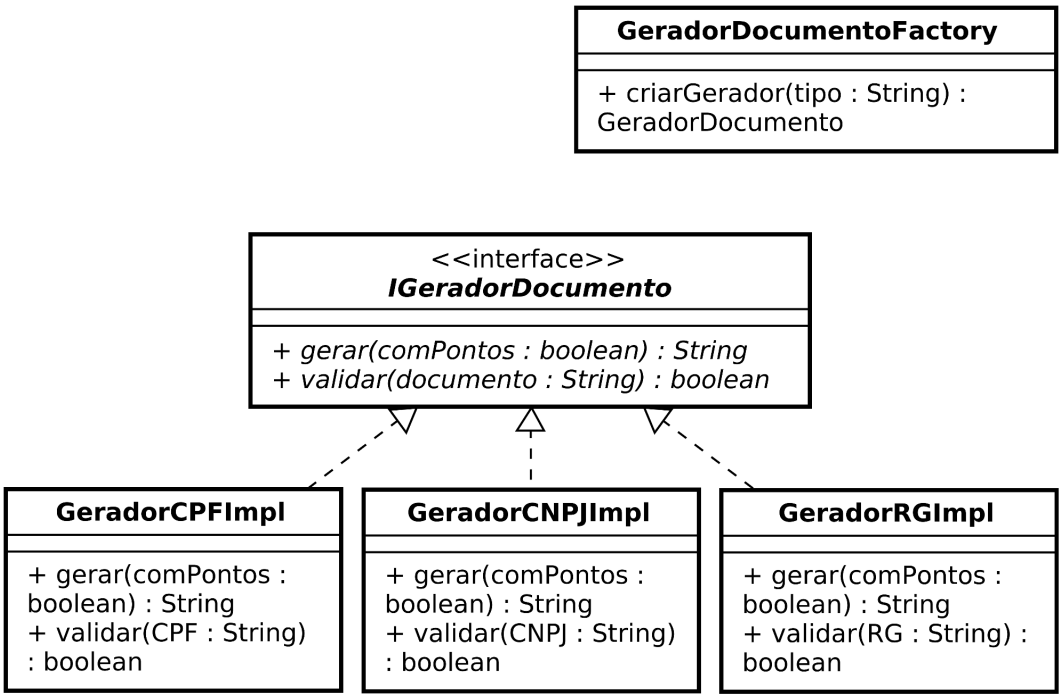
Classe principal para testar a implementação:

```
public class Principal {
  public static void main(String[] args) {
    GeradorDocumentoFactory factory = new GeradorDocumentoFactory();

    IGeradorDocumento geradorCPF = factory.criarGerador("CPF");
    String cpf = geradorCPF.gerar(true);
    System.out.printf("CPF: %s, Valido: %s\n", cpf, geradorCPF.validar(cpf));

    IGeradorDocumento geradorCNPJ = factory.criarGerador("CNPJ");
    String cnpj = geradorCNPJ.gerar(false);
    System.out.printf("CNPJ: %s, Valido: %s\n", cnpj, geradorCNPJ.validar(cnpj));

    IGeradorDocumento geradorRG = factory.criarGerador("RG");
    String rg = geradorRG.gerar(true);
```

```
        System.out.printf("RG: %s\n", rg);
    }
}
```

Após implementar e testar o código, podemos exportá-lo como um JAR para reutilização em outros projetos. Compilamos o código-fonte com javac -d out src/*.java e criamos o arquivo JAR com jar cfv gerador-documentos.jar -C out/ ..

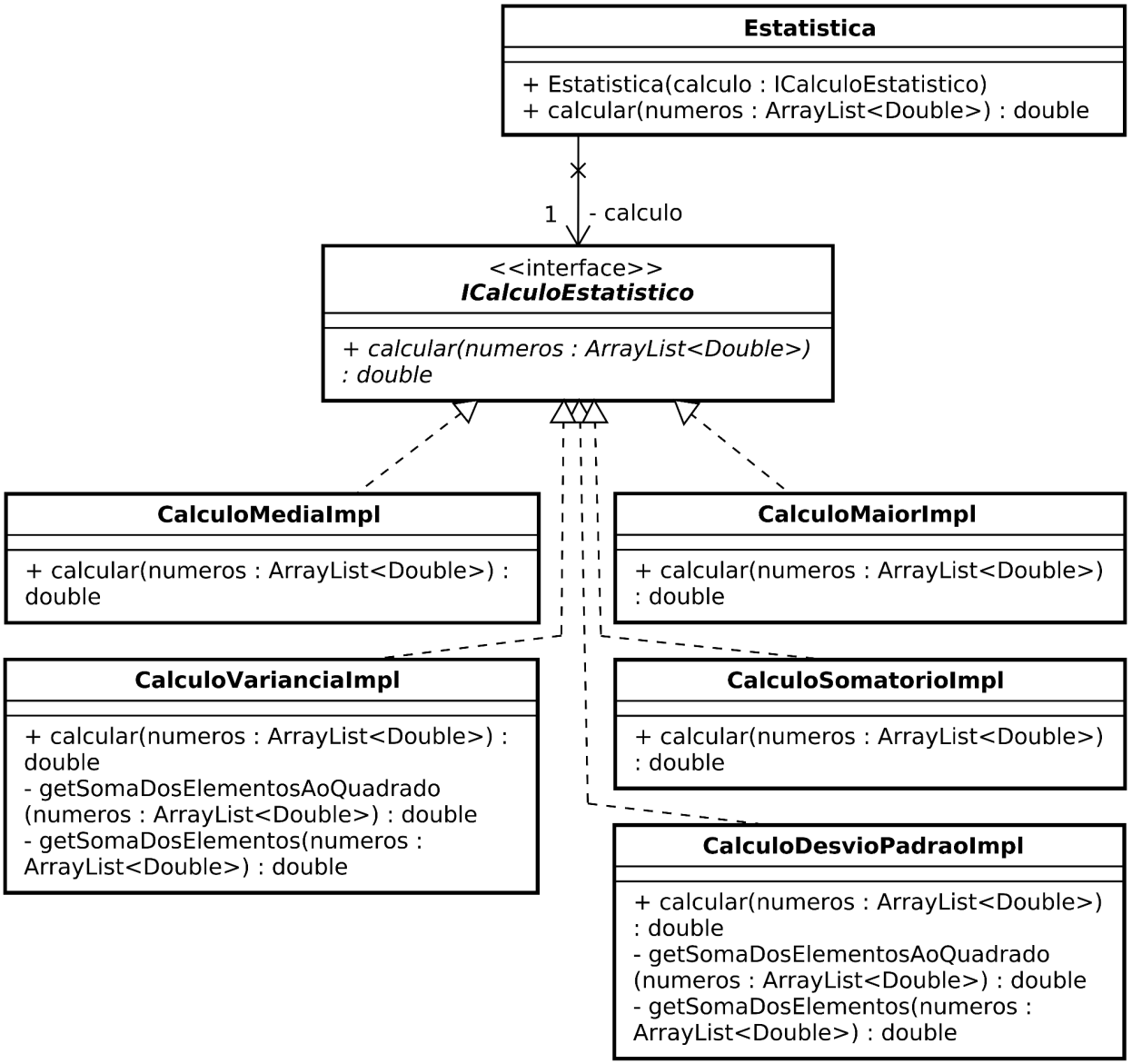
Veja a seguir o diagrama de classes da solução proposta.



Com esta refatoração, foi aplicado os princípios SOLID, DRY e LoD, além de vários padrões de projeto para criar uma solução mais modular e reutilizável. Exportando a solução como um JAR, facilitamos sua integração em outros projetos, promovendo a reutilização de código e a manutenção eficiente.

Solução do Exercício 52

Observe a seguir o diagrama de classes da solução e posteriormente a explicação textual.



Para aplicar o Princípio do Aberto Fechado ao código fornecido, utilizamos o padrão Strategy. Esse padrão nos permite definir uma família de algoritmos, encapsulá-los e torná-los intercambiáveis. Assim, cada cálculo estatístico é implementado como uma classe separada que segue uma interface comum, possibilitando a adição de novos cálculos sem a necessidade de modificar a classe principal.

Primeiro, criamos a interface ICalculoEstatistico, que define o método para realizar o cálculo estatístico. Em seguida, implementamos diferentes classes para cada tipo de cálculo: Média, Somatório, Maior, Menor, Variância e Desvio Padrão. Cada uma dessas classes implementa a interface ICalculoEstatistico e fornece sua própria lógica para o método calcular.

A classe Estatistica possui um atributo do tipo ICalculoEstatistico, inicializado no seu construtor. O método calcular da classe Estatistica delega a execução do cálculo à implementação específica de ICalculoEstatistico que foi passada na criação da instância de Estatistica.

Com essa abordagem, o código da classe Estatistica está aberto para a extensão (novos cálculos podem ser adicionados) e fechado para a modificação (não é necessário alterar a classe Estatistica para adicionar novos cálculos). Isso garante uma maior flexibilidade e manutenção mais fácil do código. Novos cálculos podem ser adicionados simplesmente criando novas implementações da interface ICalculoEstatistico.

Solução do Exercício 53

A implementação original da classe SistemaDeAlarme não atende totalmente aos princípios SOLID, LoD (Lei de Demeter), e DRY (Don't Repeat Yourself), e também não está utilizando adequadamente os padrões de projeto recomendados.

Vamos refatorar o código para alinhar com os requisitos propostos e aplicar os princípios e padrões de projeto mencionados. Vamos adotar o padrão Strategy para gerenciar diferentes tipos de sensores, permitindo a adição de novos tipos de sensores sem modificar a classe SistemaDeAlarme. Vamos usar o padrão Factory Method para criar sensores de forma flexível. Também vamos aplicar o padrão Singleton para garantir que o sistema de alarme tenha uma única instância. Além disso, utilizaremos o padrão Command para gerenciar a ativação e desativação dos sensores de forma desacoplada.

Inicialmente, criamos a interface ISensor que define um contrato para os sensores. Essa interface possui dois métodos: ativar e desativar, que todos os sensores devem implementar.

```
public interface ISensor {
  public void ativar();
  public void desativar();
}
```

A seguir, criamos a classe SensorDeMovimento que implementa a interface ISensor. Esta classe representa um sensor de movimento e fornece as implementações específicas para os métodos ativar e desativar.

```
public class SensorDeMovimentoImpl implements ISensor {
  @Override
  public void ativar() {
    System.out.println("Sensor de movimento ativado.");
  }

  @Override
  public void desativar() {
    System.out.println("Sensor de movimento desativado.");
  }
}
```

De forma semelhante, criamos a classe SensorDeAbertura que também implementa a interface ISensor. Esta classe representa um sensor de abertura de portas e janelas.

```
public class SensorDeAberturaImpl implements ISensor {
  @Override
  public void ativar() {
    System.out.println("Sensor de abertura ativado.");
  }

  @Override
  public void desativar() {
    System.out.println("Sensor de abertura desativado.");
  }
}
```

Para facilitar a criação dos diferentes tipos de sensores, utilizamos o padrão Factory Method. Criamos a interface ISensorFactory:

```
public interface ISensorFactory {
  public ISensor criarSensor();
}
```

Em seguida, implementamos fábricas concretas para cada tipo de sensor. Abaixo é codficado para o sensor de movimento e sensor de abertura:

```
public class SensorDeMovimentoFactory implements ISensorFactory {
  @Override
  public ISensor criarSensor() {
    return new SensorDeMovimentoImpl();
  }
}

public class SensorDeAberturaFactory implements ISensorFactory {
  @Override
  public ISensor criarSensor() {
    return new SensorDeAberturaImpl();
  }
}
```

Implementamos o padrão Command para encapsular as operações de ativação e desativação dos sensores. A classe AtivarSensorCommand representa o comando para ativar um sensor.

```
public class AtivarSensorCommand {
  private ISensor sensor;

  public AtivarSensorCommand(ISensor sensor) {
    this.sensor = sensor;
  }

  public void executar() {
    sensor.ativar();
  }
}
```

De forma análoga, a classe DesativarSensorCommand representa o comando para desativar um sensor.

```
public class DesativarSensorCommand {
  private ISensor sensor;

  public DesativarSensorCommand(ISensor sensor) {
    this.sensor = sensor;
  }

  public void executar() {
    sensor.desativar();
  }
}
```

Para garantir que o sistema de alarme tenha uma única instância, utilizamos o padrão Singleton na classe SistemaDeAlarme. Esta classe gerencia uma lista de sensores e permite adicionar novos sensores, ativar e desativar todos os sensores utilizando a fábrica para criar sensores.

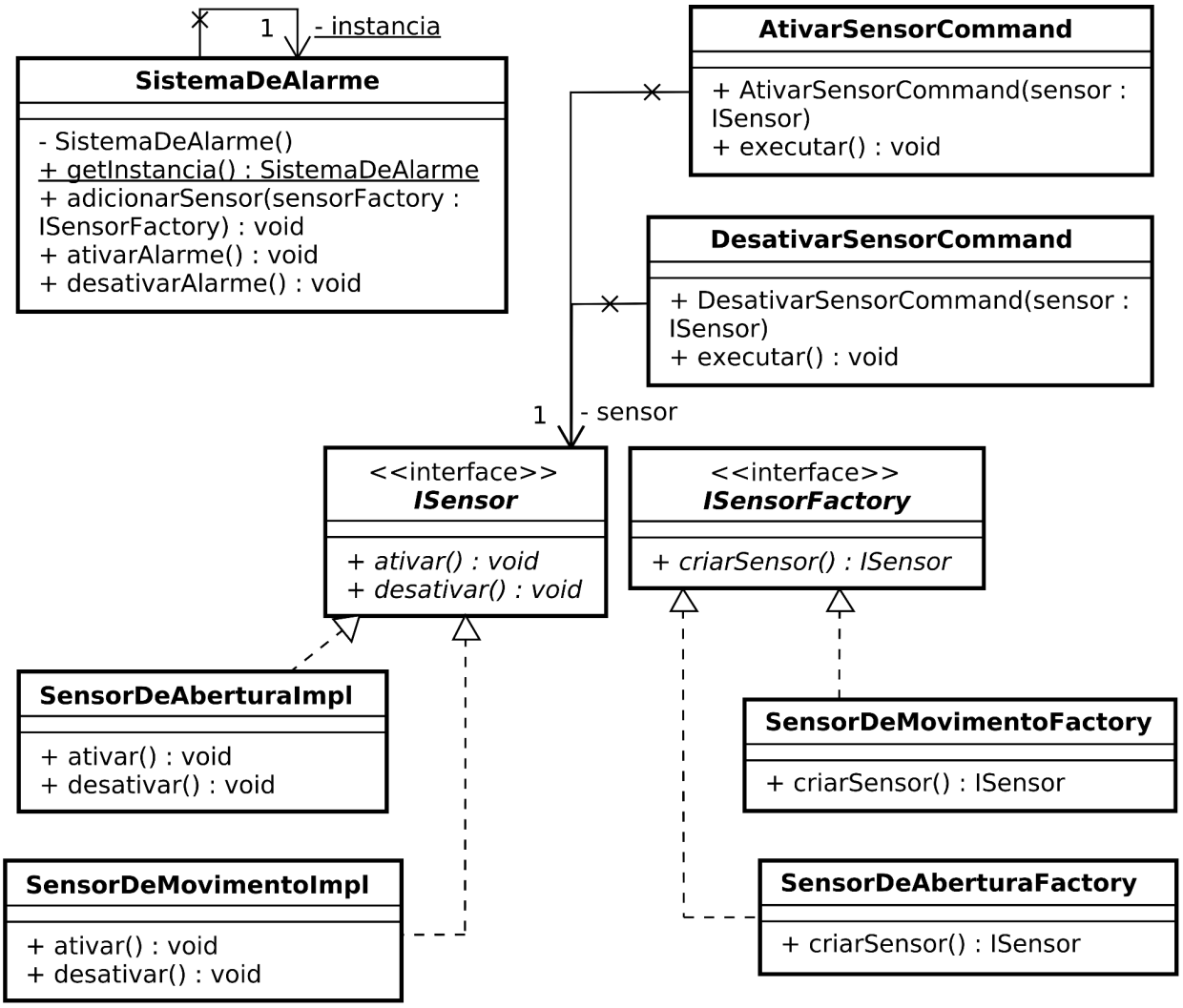
```
public class SistemaDeAlarme {
  private static SistemaDeAlarme instancia;
  private List<ISensor> sensores;

  private SistemaDeAlarme() {
    sensores = new ArrayList<>();
  }

  public static SistemaDeAlarme getInstancia() {
    if (instancia == null) {
      instancia = new SistemaDeAlarme();
    }
    return instancia;
  }

  public void adicionarSensor(ISensorFactory sensorFactory) {
    sensores.add(sensorFactory.criarSensor());
```

```
  }

  public void ativarAlarme() {
    for (ISensor sensor : sensores) {
      new AtivarSensorCommand(sensor).executar();
    }
  }

  public void desativarAlarme() {
    for (ISensor sensor : sensores) {
      new DesativarSensorCommand(sensor).executar();
    }
  }
}
```

Por fim, na classe Principal, demonstramos o uso do sistema de alarme. Criamos sensores utilizando a fábrica e os adicionamos ao sistema de alarme. Em seguida, ativamos e desativamos o alarme.

```
public class Principal {
  public static void main(String[] args) {
    SistemaDeAlarme sistemaDeAlarme = SistemaDeAlarme.getInstancia();

    sistemaDeAlarme.adicionarSensor(new SensorDeMovimentoFactory());
    sistemaDeAlarme.adicionarSensor(new SensorDeAberturaFactory());

    System.out.println("Ativando alarme:");
    sistemaDeAlarme.ativarAlarmes();

    System.out.println("\nDesativando alarme:");
    sistemaDeAlarme.desativarAlarmes();
  }
}
```

O diagrama de classes da solução proposta está a seguir.



Solução do Exercício 54

Vamos refatorar o código para alinhar com os requisitos propostos e aplicar os princípios e padrões de projeto mencionados. Vamos adotar o padrão Chain of Responsibility para gerenciar as regras de autorização de transações bancárias, permitindo a adição de novas regras sem modificar a classe principal.

Inicialmente, criamos a interface IRegraAutorizacao que define um contrato para as regras de autorização. Esta interface possui um método autorizar, que todas as regras de autorização devem implementar.

```
public interface IRegraAutorizacao {
```

```
  boolean autorizar(Transacao transacao);
  void setProxima(IRegraAutorizacao proxima);
}
```

A seguir, criamos a classe abstrata RegraAutorizacaoBaseImpl que implementa a interface IRegraAutorizacao e fornece a lógica básica para encadeamento das regras.

```
public abstract class RegraAutorizacaoBaseImpl implements IRegraAutorizacao {
  protected IRegraAutorizacao proxima;

  @Override
  public void setProxima(IRegraAutorizacao proxima) {
    this.proxima = proxima;
  }

  protected boolean autorizarProxima(Transacao transacao) {
    if (proxima == null) {
      return true;
    }
    return proxima.autorizar(transacao);
  }
}
```

Agora, criamos a classe RegraSaldo que implementa a lógica específica para a verificação de saldo suficiente.

```
public class RegraSaldo extends RegraAutorizacaoBaseImpl {
  @Override
  public boolean autorizar(Transacao transacao) {
    if (transacao.getSaldoDisponivel() >= transacao.getValor()) {
      return autorizarProxima(transacao);
    }
    transacao.adicionarAutorizacao("Regra de Saldo", false);
    return false;
```

```
  }
}
```

Em seguida, criamos a classe RegraValorMaximo que implementa a lógica específica para a verificação de valor máximo de transação, dependendo do tipo de conta.

```
public class RegraValorMaximo extends RegraAutorizacaoBaseImpl {
  @Override
  public boolean autorizar(Transacao transacao) {
    String tipoConta = transacao.getConta().getTipoConta();
    double valorMaximo = tipoConta.equals("Premium") ? 10000.0 : 1000.0;
    if (transacao.getValor() <= valorMaximo) {
      return autorizarProxima(transacao);
    }
    transacao.adicionarAutorizacao("Regra de Valor Máximo", false);
    return false;
  }
}
```

Posteriormente, criamos a classe RegraBloqueioJudicial que implementa a lógica específica para a verificação de bloqueios judiciais na conta.

```
public class RegraBloqueioJudicial extends RegraAutorizacaoBaseImpl {
  @Override
  public boolean autorizar(Transacao transacao) {
    if (!transacao.isBloqueioJudicial()) {
      return autorizarProxima(transacao);
    }
    transacao.adicionarAutorizacao("Regra de Bloqueio Judicial", false);
    return false;
  }
}
```

Criamos a classe SistemaAutorizacaoTransacao que será responsável por gerenciar as regras de autorização e processar as transações.

```
public class SistemaAutorizacaoTransacao {
  private IRegraAutorizacao primeiraRegra;

  public SistemaAutorizacaoTransacoes() {
     IRegraAutorizacao regraSaldo = new RegraSaldo();
     IRegraAutorizacao regraValorMaximo = new RegraValorMaximo();
     IRegraAutorizacao regraBloqueioJudicial = new RegraBloqueioJudicial();

     regraSaldo.setProxima(regraValorMaximo);
     regraValorMaximo.setProxima(regraBloqueioJudicial);

     this.primeiraRegra = regraSaldo;
  }

  public boolean autorizarTransacao(Transacao transacao) {
     return primeiraRegra.autorizar(transacao);
  }
}
```

Abaixo, é mostrado o código Java das classes do tipo model que foram apresentadas no enunciado.

```
public class Conta {
  private String tipoConta;
  private double saldoDisponivel;
  private boolean bloqueioJudicial;

  public Conta(String tipoConta, double saldoDisponivel, boolean bloqueioJudicial) {
     this.tipoConta = tipoConta;
     this.saldoDisponivel = saldoDisponivel;
     this.bloqueioJudicial = bloqueioJudicial;
  }
```

```
    public String getTipoConta() { return tipoConta;}

    public double getSaldoDisponivel() {return saldoDisponivel;}

    public boolean isBloqueioJudicial() {return bloqueioJudicial;}

}

public class Autorizacao {

    private String nomeRegra;
    private LocalDateTime dataHora;
    private boolean aprovada;

    public Autorizacao(String nomeRegra, boolean aprovada) {
        this.nomeRegra = nomeRegra;
        this.dataHora = LocalDateTime.now();
        this.aprovada = aprovada;
    }

    public String getNomeRegra() {return nomeRegra;}
    public LocalDateTime getDataHora() { return dataHora; }
    public boolean isAprovada() {return aprovada;}

}

public class Transacao {

    private Conta conta;
    private double valor;
    private LocalDateTime horario;
    private List<Autorizacao> autorizacoes = new ArrayList<>();
```

```
  public Transacao(Conta conta, double valor) {
     this.conta = conta;
     this.valor = valor;
     this.horario = LocalDateTime.now();
  }

  public void adicionarAutorizacao(String nomeRegra, boolean aprovada) {
     autorizacoes.add(new Autorizacao(nomeRegra, aprovada));
  }

  // gets
  public boolean isBloqueioJudicial() { return conta.isBloqueioJudicial();}

  public boolean isAprovada() {
     for (Autorizacao autorizacao : autorizacoes) {
       if (autorizacao.getDataHora().toLocalDate().equals(this.horario.toLocalDate()) &&
!autorizacao.isAprovada()) {
            return false;
        }
     }
     return true;
  }

  public double getSaldoDisponivel() {return conta.getSaldoDisponivel();}

}
```

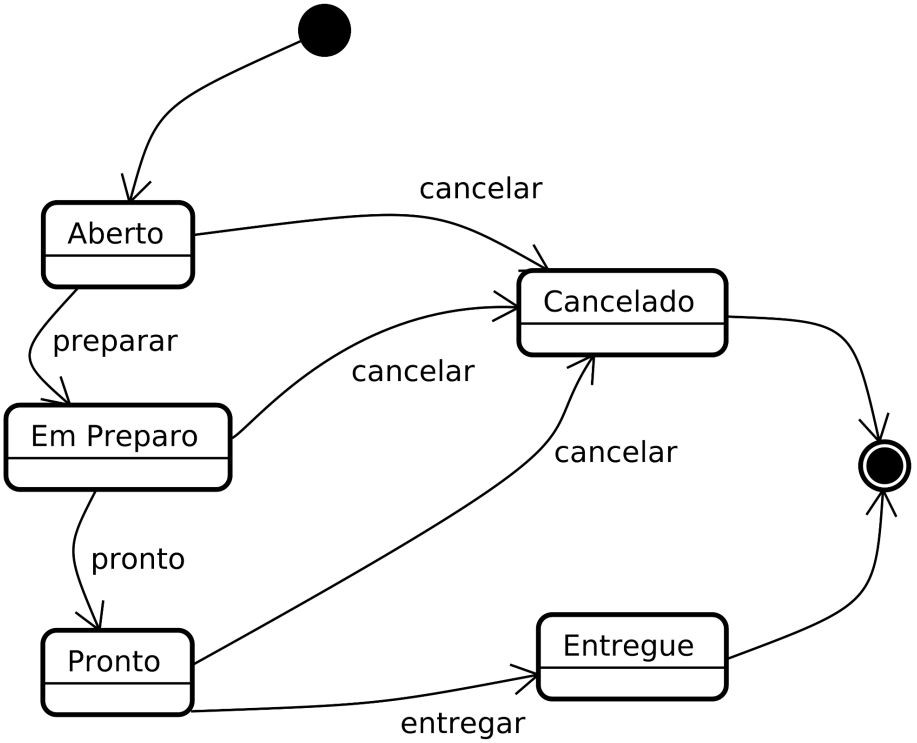
Finalmente, na classe Principal, demonstramos o uso do sistema de autorização de transações. Criamos uma transação e a processamos através do sistema de autorização.

```
public class Principal {
  public static void main(String[] args) {
```

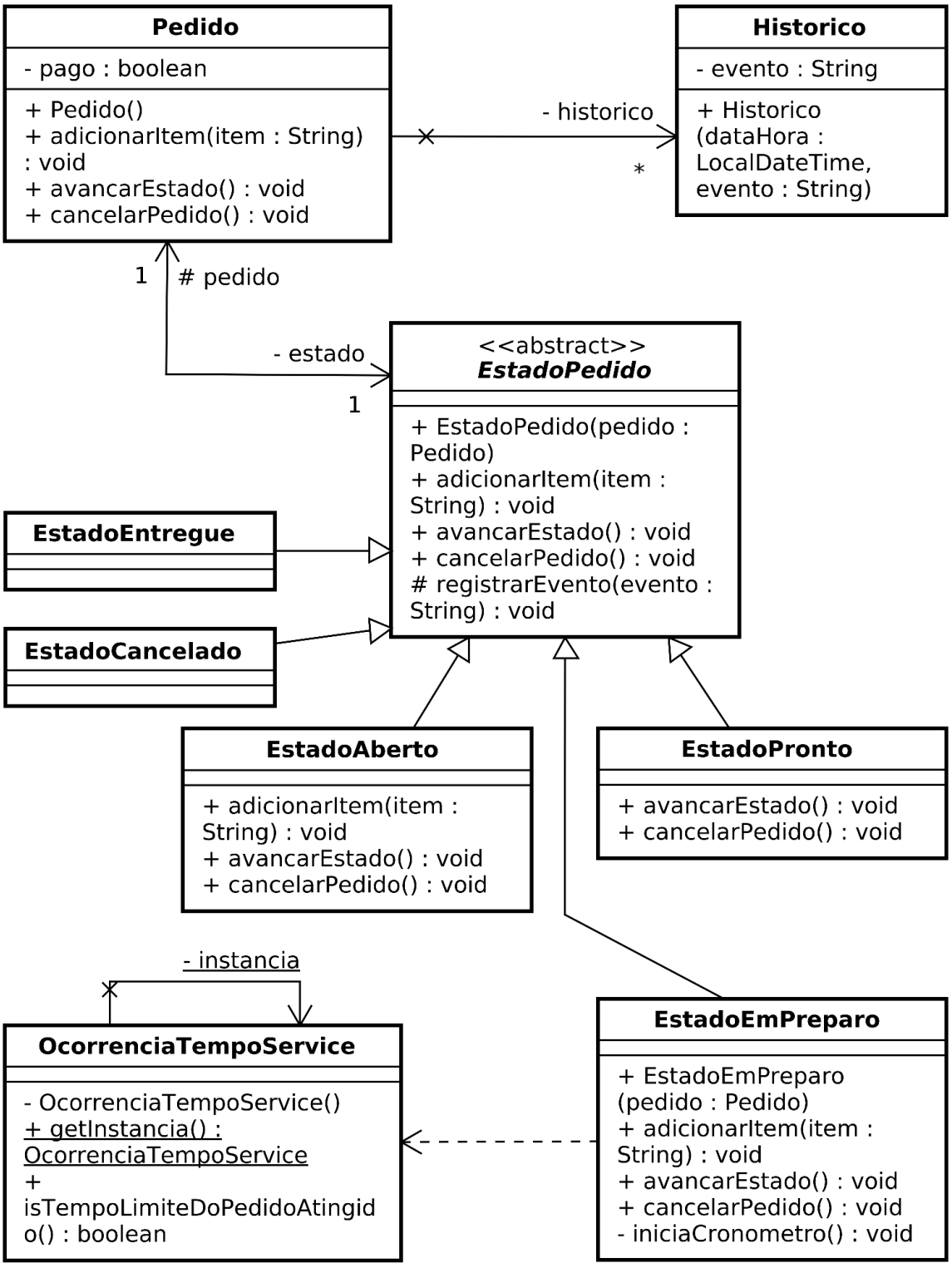
```
        Conta conta = new Conta("Comum", 1500.0, false);
        Transacao transacao = new Transacao(conta, 500.0);

        SistemaAutorizacaoTransacao sistema = new SistemaAutorizacaoTransacao();
        boolean autorizada = sistema.autorizarTransacao(transacao);

        System.out.println("Transação autorizada: " + autorizada);
    }
}
```

Solução do Exercício 55

A)



B)



Abaixo é mostrado em detalhes a solução com aplicação em código Java.

A implementação original não está utilizando adequadamente os padrões de projeto recomendados. Vamos refatorar o código para alinhar com os requisitos propostos. Iremos aplicar o padrão State para gerenciar os diferentes estados de um pedido, permitindo a adição de novos estados sem modificar a classe principal. Também utilizaremos o padrão Singleton para gerenciar o serviço de tempo.

Inicialmente, criamos a classe abstrata EstadoPedido que define um contrato para os estados de um pedido. Esta classe possui métodos para as operações permitidas em um pedido, como adicionar item, avançar estado e cancelar pedido. Além disso, ela fornece a lógica básica para registrar eventos no histórico.

```
public abstract class EstadoPedido {
  protected Pedido pedido;

  public EstadoPedido(Pedido pedido) {
    this.pedido = pedido;
  }

  public void adicionarItem(String item) {
    throw new UnsupportedOperationException("Não é possível adicionar itens");
  }

  public void avancarEstado() {
    throw new UnsupportedOperationException("Não é possível avançar o estado");
  }

  public void cancelarPedido() {
    throw new UnsupportedOperationException("Não é possível cancelar o pedido");
  }

  protected void registrarEvento(String evento) {
    pedido.getHistorico().add(new Historico(LocalDateTime.now(), evento));
  }
```

```
}
```

A seguir, criamos a classe EstadoAberto que implementa a lógica específica para o estado "aberto".

```
public class EstadoAberto extends EstadoPedido {
  // Construtor
  @Override
  public void adicionarItem(String item) {
     pedido.getItens().add(item);
     registrarEvento("Item adicionado: " + item);
  }

  @Override
  public void avancarEstado() {
     pedido.setEstado(new EstadoEmPreparo(pedido));
     registrarEvento("Pedido em preparo");
  }

  @Override
  public void cancelarPedido() {
     pedido.setEstado(new EstadoCancelado(pedido));
     registrarEvento("Pedido cancelado");
  }
}
```

De forma semelhante, criamos a classe EstadoEmPreparo que implementa a lógica específica para o estado "em preparo". Implementamos também o método privado iniciaCronometro, caso o tempo de preparo seja igual ou maior que 10 minutos, o pedido é cancelado e é exibido uma mensagem no console.

```
public class EstadoEmPreparo extends EstadoPedido {
  public EstadoEmPreparo(Pedido pedido) {
     super(pedido);
```

```
        iniciaCronometro();
    }

    @Override
    public void adicionarItem(String item) {
        pedido.getItens().add(item);
        registrarEvento("Item adicionado: " + item);
    }

    @Override
    public void avancarEstado() {
        pedido.setEstado(new EstadoPronto(pedido));
        registrarEvento("Pedido pronto");
    }

    @Override
    public void cancelarPedido() {
        pedido.setEstado(new EstadoCancelado(pedido));
        registrarEvento("Pedido cancelado");
    }

    private void iniciaCronometro() {
        new Thread(() -> {
            if (OcorrenciaTempoService.getInstancia().isTempoLimiteDoPedidoAtingido()) {
                pedido.setEstado(new EstadoCancelado(pedido));
                System.out.println("Pedido cancelado por tempo de preparo excedido.");
            }
        }).start();
    }
}
```

A seguir, criamos a classe EstadoPronto que implementa a lógica específica para o estado "pronto".

```
public class EstadoPronto extends EstadoPedido {
  //Construtor
  @Override
  public void avancarEstado() {
     pedido.setEstado(new EstadoEntregue(pedido));
     registrarEvento("Pedido entregue");
  }

  @Override
  public void cancelarPedido() {
     pedido.setEstado(new EstadoCancelado(pedido));
     registrarEvento("Pedido cancelado");
  }
}
```

A seguir, criamos as classes EstadoEntregue e EstadoCancelado que implementam a lógica específica para os estados "entregue" e "cancelado", respectivamente.

```
public class EstadoEntregue extends EstadoPedido {
  //Construtor
}

public class EstadoCancelado extends EstadoPedido {
  //Construtor
}
```

A seguir, criamos a classe Pedido que gerencia o estado atual e as operações de mudança de estado, adição de itens e cancelamento.

```
public class Pedido {
  private List<String> itens;
  private List<Historico> historico;
  private EstadoPedido estado;
  private boolean pago;
```

```
  public Pedido() {
     historico = new ArrayList<>();
     itens = new ArrayList<>();
     estado = new EstadoAberto(this);
  }

  public void adicionarItem(String item) {
     estado.adicionarItem(item);
  }

  public void avancarEstado() {
     estado.avancarEstado();
  }

  public void cancelarPedido() {
     estado.cancelarPedido();
  }
  //getters e setters
}
```

O código da classe Historico manteve-se inalterado.

```
public class Historico {
  private LocalDateTime dataHora;
  private String evento;
   public Historico(LocalDateTime dataHora,
     String evento) {
     this.dataHora = dataHora;
     this.evento = evento;
  }
  // ...
}
```

Para garantir que o sistema possa verificar automaticamente o tempo de preparo e cancelar pedidos que ultrapassem 10 minutos, utilizamos a classe OcorrenciaTempoService como um Singleton.

```
public class OcorrenciaTempoService {
  private static final OcorrenciaTempoService instancia = new OcorrenciaTempoService();

  private OcorrenciaTempoService() {}

  public static OcorrenciaTempoService getInstancia() {
    return instancia;
  }

  public boolean isTempoLimiteDoPedidoAtingido() {
    try {
      Thread.sleep(600000); // 10 minutos
    } catch (InterruptedException e) {
      throw new RuntimeException("Falha na simulação de tempo: " + e.getMessage());
    }
    return new Random().nextBoolean();
  }
}
```

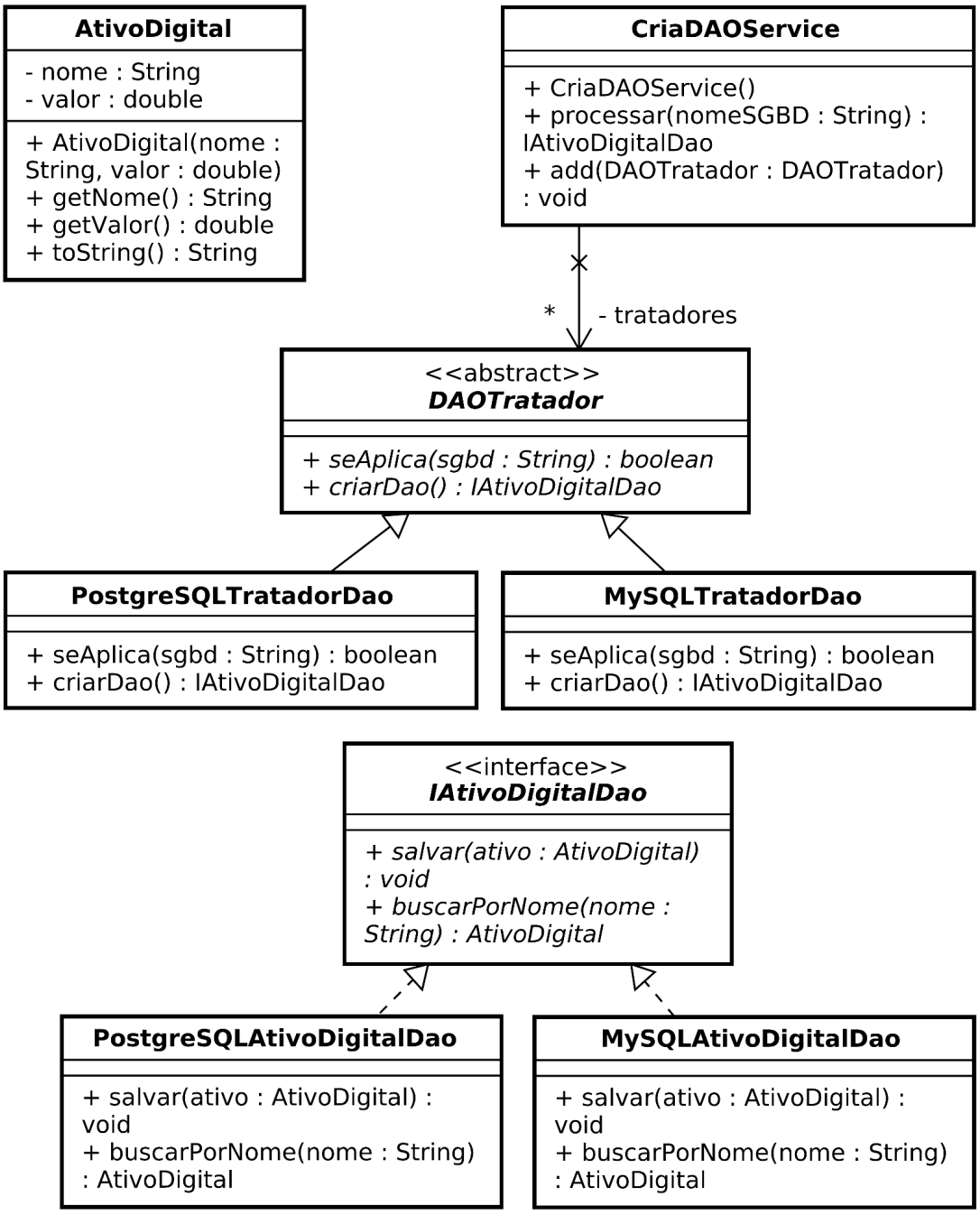
Finalmente, na classe Principal, demonstramos o uso do sistema de pedidos. Dentro do bloco try-catch, criamos um pedido, adicionamos itens e avançamos estados.

```
public class Principal {
  public static void main(String[] args) {
    try {
      Pedido pedido = new Pedido();

      pedido.adicionarItem("Pizza");
      pedido.adicionarItem("Refrigerante");
```

```
            pedido.avancarEstado(); // Avança para 'em preparo'
            pedido.avancarEstado(); // Avança para 'pronto'
            pedido.avancarEstado(); // Avança para 'entregue'

        } catch (Exception e) {
            System.out.println("Erro: " + e.getMessage());
        }

    }
}
```

Solução do Exercício 56

a)



b)

Para resolver o problema proposto, utilizamos o padrão de projeto Chain of Responsibility para criar uma cadeia de manipuladores responsáveis pela criação das DAOs específicas para cada SGBD. Cada manipulador na cadeia verificará se pode criar a DAO para o SGBD

fornecido, e, se não puder, passará a responsabilidade para o próximo manipulador na cadeia. Se nenhum manipulador for capaz de lidar com o SGBD fornecido, uma exceção será lançada.

Inicialmente, definimos a classe CriaDAOService que gerencia a cadeia de responsabilidade e fornece um método para processar a criação da DAO com base no nome do SGBD fornecido.

```
public class CriaDAOService {
  private List<DAOTratador> tratadores;

  public CriaDAOService() {
    tratadores = new ArrayList<>();
    tratadores.add(new MySQLTratadorDao());
    tratadores.add(new PostgreSQLTratadorDao());
  }

  public IAtivoDigitalDao processar(String nomeSGBD) {
    for (DAOTratador tratador : tratadores) {
      if (tratador.seAplica(nomeSGBD)) {
        return tratador.criarDao();
      }
    }
    throw new IllegalArgumentException("SGBD não suportado: " + nomeSGBD);
  }

  public void add(DAOTratador DAOTratador) {
    if (!tratadores.contains(DAOTratador)) {
      this.tratadores.add(DAOTratador);
    }
  }
}
```

Definimos a classe abstrata DAOTratador que serve como base para os manipuladores na cadeia. Ela define dois métodos abstratos: seAplica para verificar se o manipulador pode lidar com o SGBD fornecido e criarDao para criar a DAO correspondente.

```
public abstract class DAOTratador {
  public abstract boolean seAplica(String sgbd);
  public abstract IAtivoDigitalDao criarDao();
}
```

Criamos as classes MySQLTratadorDao e PostgreSQLTratadorDao que implementam a lógica específica para verificar e criar DAOs para MySQL e PostgreSQL, respectivamente.

```
public class MySQLTratadorDao extends DAOTratador {
  @Override
  public boolean seAplica(String sgbd) {
     return "MySQL".equalsIgnoreCase(sgbd);
  }

  @Override
  public IAtivoDigitalDao criarDao() {
     return new MySQLAtivoDigitalDao();
  }
}

public class PostgreSQLTratadorDao extends DAOTratador {
  @Override
  public boolean seAplica(String sgbd) {
     return "PostgreSQL".equalsIgnoreCase(sgbd);
  }

  @Override
  public IAtivoDigitalDao criarDao() {
     return new PostgreSQLAtivoDigitalDao();
  }
```

```
}
```

Definimos a interface IAtivoDigitalDao que contém os métodos para salvar e buscar ativos digitais.

```
public interface IAtivoDigitalDao {
  public void salvar(AtivoDigital ativo);
  public AtivoDigital buscarPorNome(String nome);
}
```

Implementamos as classes MySQLAtivoDigitalDao e PostgreSQLAtivoDigitalDao que fornecem a implementação concreta dos métodos definidos na interface IAtivoDigitalDao para MySQL e PostgreSQL, respectivamente.

```
public class MySQLAtivoDigitalDao implements IAtivoDigitalDao {
  @Override
  public void salvar(AtivoDigital ativo) {
     System.out.println("Salvando ativo digital: " + ativo.getNome() + " no MySQL");
  }

  @Override
  public AtivoDigital buscarPorNome(String nome) {
     System.out.println("Buscando ativo digital: " + nome + " no MySQL");
     return new AtivoDigital(nome, Math.round((5 + Math.random() * 9995) * 100.0) / 100.0);
  }
}

public class PostgreSQLAtivoDigitalDao implements IAtivoDigitalDao {
  @Override
  public void salvar(AtivoDigital ativo) {
     System.out.println("Salvando ativo digital: " + ativo.getNome() + " no PostgreSQL");
  }

  @Override
```

```
    public AtivoDigital buscarPorNome(String nome) {
        System.out.println("Buscando ativo digital: " + nome + " no PostgreSQL");
        return new AtivoDigital(nome, Math.round((5 + Math.random() * 9995) * 100.0) / 100.0);
    }
}
```

Definimos a classe AtivoDigital que representa um ativo digital com um nome e um valor.

```
public class AtivoDigital {
    private String nome;
    private double valor;

    public AtivoDigital(String nome, double valor) {
        this.nome = nome;
        this.valor = valor;
    }

    public String getNome() {
        return nome;
    }

    public double getValor() {
        return valor;
    }

    @Override
    public String toString() {
        return "AtivoDigital{" + "nome='" + nome + '\'' + ", valor= R$ " + valor + '}';
    }
}
```

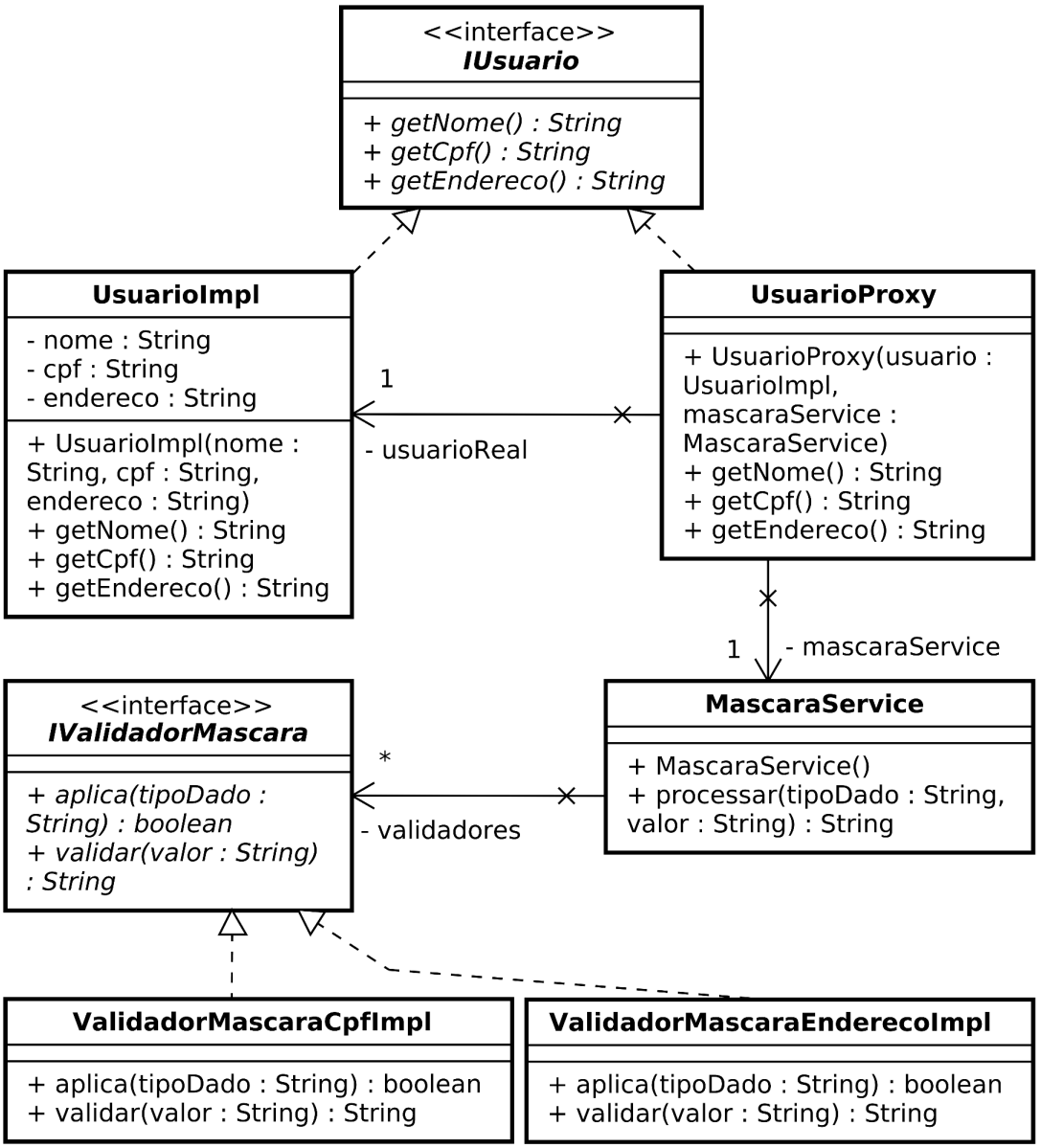
c)

Finalmente, na classe Principal, demonstramos o uso do sistema de gerenciamento de investimentos para criar e salvar ativos digitais em diferentes SGBDs.

```
public class Principal {
  public static void main(String[] args) {
    try {
      AtivoDigital ativo = new AtivoDigital("Bitcoin", 40000.0);
      CriaDAOService servicoDao = new CriaDAOService();

      IAtivoDigitalDao daoMySQL = servicoDao.processar("MySQL");
      System.out.println("Processando com MySQL");
      daoMySQL.salvar(ativo);
      AtivoDigital ativoEncontrado = daoMySQL.buscarPorNome(ativo.getNome());
      System.out.println(ativoEncontrado);


      IAtivoDigitalDao daoPostgreSQL = servicoDao.processar("PostgreSQL");
      System.out.println("Processando com PostgreSQL");
      daoPostgreSQL.salvar(ativo);
      ativoEncontrado = daoPostgreSQL.buscarPorNome(ativo.getNome());
      System.out.println(ativoEncontrado);

      IAtivoDigitalDao daoOracle = servicoDao.processar("Oracle");
      System.out.println("Processando com Oracle");
      daoOracle.salvar(ativo);
      daoOracle.buscarPorNome(ativo.getNome());

    } catch (IllegalArgumentException e) {
      System.out.println("Falha: " + e.getMessage());
    }
  }
}
```

Solução do Exercício 57

a)



b)

Para desenvolver a solução que atende ao enunciado, implementaremos dois padrões de projeto: Proxy e Chain of Responsibility. O Proxy será usado para filtrar e adequar a exibição dos dados do usuário, enquanto a Chain of Responsibility será utilizada para verificar os critérios de exibição dos atributos da classe Usuario. A seguir, apresentamos o código em Java da solução:

Inicialmente, definimos a interface IUsuario que será implementada pelas classes UsuarioImpl e UsuarioProxy.

```
public interface IUsuario {
  public String getNome();
  public String getCpf();
  public String getEndereco();
}
```

Em seguida, implementamos a classe UsuarioImpl, que representa o usuário real com atributos nome, cpf e endereco.

```
public class UsuarioImpl implements IUsuario {
  private String nome;
  private String cpf;
  private String endereco;

  public UsuarioImpl(String nome, String cpf, String endereco) {
    this.nome = nome;
    this.cpf = cpf;
    this.endereco = endereco;
  }

  @Override
  public String getNome() {
    return nome;
  }

  @Override
  public String getCpf() {
    return cpf;
  }

  @Override
```

```
  public String getEndereco() {
    return endereco;
  }
}
```

Implementamos a interface IValidadorMascara, que define os métodos para aplicar as máscaras de proteção aos dados sensíveis.

```
public interface IValidadorMascara {
  public boolean aplica(String tipoDado);
  public String validar(String valor);
}
```

A seguir, implementamos a classe ValidadorMascaraCpfImpl, que aplica a máscara ao CPF do usuário, ocultando os três primeiros e os dois últimos dígitos.

```
public class ValidadorMascaraCpfImpl implements IValidadorMascara {
  @Override
  public boolean aplica(String tipoDado) {
    return "cpf".equalsIgnoreCase(tipoDado);
  }

  @Override
  public String validar(String valor) {
    return "***.***." + valor.substring(7, 10) + "-" + valor.substring(12);
  }
}
```

Implementamos também a classe ValidadorMascaraEnderecoImpl, que assegura que o endereço do usuário não seja exibido.

```
public class ValidadorMascaraEnderecoImpl implements IValidadorMascara {
  @Override
  public boolean aplica(String tipoDado) {
    return "endereco".equalsIgnoreCase(tipoDado);
```

```
    }

    @Override
    public String validar(String valor) {
        return "Acesso ao endereço não permitido.";
    }
}
```

A classe MascaraService gerencia a aplicação das máscaras, integrando os validadores em uma cadeia de responsabilidade.

```
public class MascaraService {
    private List<IValidadorMascara> validadores;

    public MascaraService() {
        validadores = new ArrayList<>();
        validadores.add(new ValidadorMascaraCpfImpl());
        validadores.add(new ValidadorMascaraEnderecoImpl());
    }

    public String processar(String tipoDado, String valor) {
        for (IValidadorMascara validador : validadores) {
            if (validador.aplica(tipoDado)) {
                return validador.validar(valor);
            }
        }
        return valor;
    }
}
```

A seguir, implementamos a classe UsuarioProxy, que usa o MascaraService para aplicar as máscaras de proteção aos dados sensíveis do usuário.

```
public class UsuarioProxy implements IUsuario {
    private UsuarioImpl usuarioReal;
```

```
    private MascaraService mascaraService;

    public UsuarioProxy(UsuarioImpl usuario, MascaraService mascaraService) {
        this.usuarioReal = usuario;
        this.mascaraService = mascaraService;
    }

    @Override
    public String getNome() {
        return usuarioReal.getNome();
    }

    @Override
    public String getCpf() {
        return mascaraService.processar("cpf", usuarioReal.getCpf());
    }

    @Override
    public String getEndereco() {
        return mascaraService.processar("endereco", usuarioReal.getEndereco());
    }
}
```

c)

Agora, criamos a classe Principal que demonstra o uso do sistema de controle de acesso aos dados do usuário, conforme as diretrizes de proteção de dados.

```
public class Principal {
    public static void main(String[] args) {
        UsuarioImpl usuario = new UsuarioImpl("João Silva", "123.456.789-10", "Rua Exemplo, 123");

        MascaraService mascaraService = new MascaraService();
```

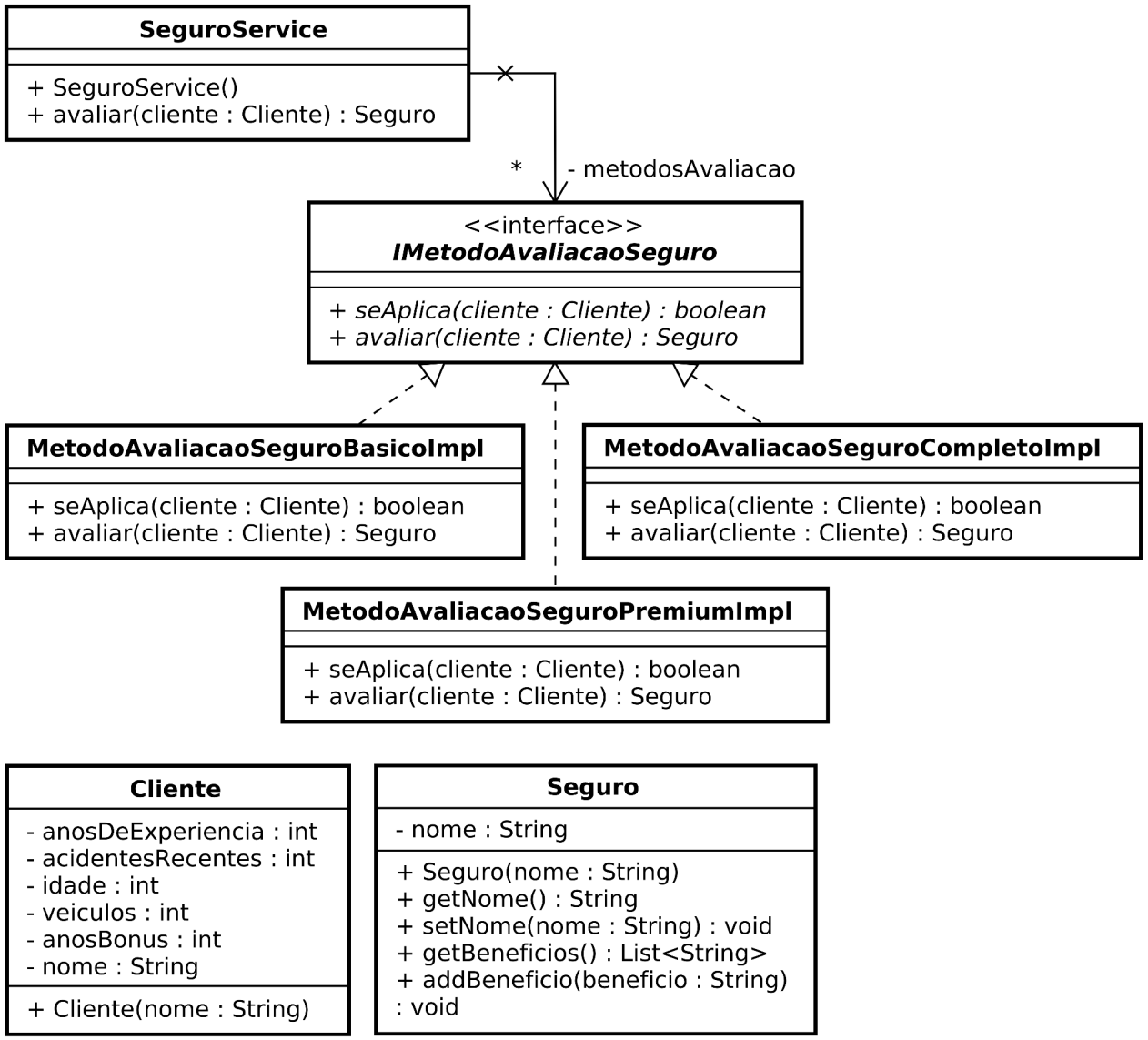
```
        UsuarioProxy usuarioProxy = new UsuarioProxy(usuario, mascaraService);

        System.out.println("Nome: " + usuarioProxy.getNome());
        System.out.println("CPF: " + usuarioProxy.getCpf());
        System.out.println("Endereço: " + usuarioProxy.getEndereco());
    }
}
```

Ao executar a classe Principal, espera-se a seguinte saída:

```
Nome: João Silva
CPF: ***.***.789-10
Endereço: Acesso ao endereço não permitido.
```

Solução do Exercício 58

a)



b)

Para desenvolver a solução que atende ao enunciado, implementaremos o padrão de projeto Chain of Responsibility para avaliar o tipo de seguro mais adequado para cada cliente com base em seus atributos. A seguir, apresentamos o código em Java da solução.

Inicialmente, definimos a interface IMetodoAvaliacaoSeguro que será implementada pelas classes de métodos de avaliação específicos.

```
public interface IMetodoAvaliacaoSeguro {
```

```
  public boolean seAplica(Cliente cliente);
  public Seguro avaliar(Cliente cliente);
}
```

Em seguida, implementamos a classe MetodoAvaliacaoSeguroBasicoImpl, que verifica se um cliente se qualifica para o Seguro Básico.

```
public class MetodoAvaliacaoSeguroBasicoImpl implements IMetodoAvaliacaoSeguro {
  @Override
  public boolean seAplica(Cliente cliente) {
    return cliente.getAnosDeExperiencia() > 10
        && cliente.getAcidentesRecentes() == 0
        && cliente.getIdade() > 35;
  }

  @Override
  public Seguro avaliar(Cliente cliente) {
    Seguro seguro = new Seguro("Seguro Básico");
    seguro.addBeneficio("Cobertura de danos a terceiros");
    return seguro;
  }
}
```

Implementamos a classe MetodoAvaliacaoSeguroCompletoImpl, que verifica se um cliente se qualifica para o Seguro Completo.

```
public class MetodoAvaliacaoSeguroCompletoImpl implements IMetodoAvaliacaoSeguro {
  @Override
  public boolean seAplica(Cliente cliente) {
    return cliente.getAcidentesRecentes() <= 1
        && cliente.getIdade() >= 25 && cliente.getIdade() <= 35
        && cliente.getAnosBonus() >= 5;
  }
```

```
    @Override
    public Seguro avaliar(Cliente cliente) {
        Seguro seguro = new Seguro("Seguro Completo");
        seguro.addBeneficio("Cobertura de danos a terceiros");
        seguro.addBeneficio("Cobertura de danos ao próprio veículo");
        seguro.addBeneficio("Cobertura contra furto");
        return seguro;
    }
}
```

Implementamos a classe MetodoAvaliacaoSeguroPremiumImpl, que verifica se um cliente se qualifica para o Seguro Premium.

```
public class MetodoAvaliacaoSeguroPremiumImpl implements IMetodoAvaliacaoSeguro {
    @Override
    public boolean seAplica(Cliente cliente) {
        return cliente.getVeiculos() > 2
                && cliente.getAnosBonus() < 5;
    }

    @Override
    public Seguro avaliar(Cliente cliente) {
        Seguro seguro = new Seguro("Seguro Premium");
        seguro.addBeneficio("Cobertura de danos a terceiros");
        seguro.addBeneficio("Cobertura de danos ao próprio veículo");
        seguro.addBeneficio("Cobertura contra furto");
        seguro.addBeneficio("Assistência 24 horas");
        return seguro;
    }
}
```

A classe SeguroService gerencia os diferentes métodos de avaliação e determina o tipo de seguro mais adequado para o cliente, com base nos critérios definidos.

```
public class SeguroService {
  private List<IMetodoAvaliacaoSeguro> metodosAvaliacao;

  public SeguroService() {
    metodosAvaliacao = new ArrayList<>();
    metodosAvaliacao.add(new MetodoAvaliacaoSeguroBasicoImpl());
    metodosAvaliacao.add(new MetodoAvaliacaoSeguroCompletoImpl());
    metodosAvaliacao.add(new MetodoAvaliacaoSeguroPremiumImpl());
  }

  public Seguro avaliar(Cliente cliente) {
    for (IMetodoAvaliacaoSeguro metodoAvaliacao : metodosAvaliacao) {
      if (metodoAvaliacao.seAplica(cliente)) {
        return metodoAvaliacao.avaliar(cliente);
      }
    }
    throw new RuntimeException("Nenhum tipo de seguro se aplica ao cliente " +
cliente.getNome());
  }
}
```

A classe Cliente representa os dados do cliente, incluindo atributos como anos de experiência, acidentes recentes, idade, número de veículos, anos de bônus e nome.

```
public class Cliente {
  private int anosDeExperiencia;
  private int acidentesRecentes;
  private int idade;
  private int veiculos;
  private int anosBonus;
  private String nome;

  public Cliente(String nome) {
```

```
    this.nome = nome;
}

// Getters e setters para os atributos
public int getAnosDeExperiencia() {
    return anosDeExperiencia;
}

public void setAnosDeExperiencia(int anosDeExperiencia) {
    this.anosDeExperiencia = anosDeExperiencia;
}

public int getAcidentesRecentes() {
    return acidentesRecentes;
}

public void setAcidentesRecentes(int acidentesRecentes) {
    this.acidentesRecentes = acidentesRecentes;
}

public int getIdade() {
    return idade;
}

public void setIdade(int idade) {
    this.idade = idade;
}

public int getVeiculos() {
    return veiculos;
}
```

```
    public void setVeiculos(int veiculos) {
        this.veiculos = veiculos;
    }

    public int getAnosBonus() {
        return anosBonus;
    }

    public void setAnosBonus(int anosBonus) {
        this.anosBonus = anosBonus;
    }

    public String getNome() {
        return nome;
    }
}
```

A classe Seguro representa os diferentes tipos de seguro e seus benefícios.

```
public class Seguro {
    private String nome;
    private List<String> beneficios;

    public Seguro(String nome) {
        this.nome = nome;
        this.beneficios = new ArrayList<>();
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
```

```
        this.nome = nome;
    }

    public List<String> getBeneficios() {
        return beneficios;
    }

    public void addBeneficio(String beneficio) {
        this.beneficios.add(beneficio);
    }
}
```

c)

Agora, criamos a classe Principal que demonstra o uso do sistema de avaliação de seguros, criando diferentes clientes e avaliando-os para determinar o tipo de seguro adequado.

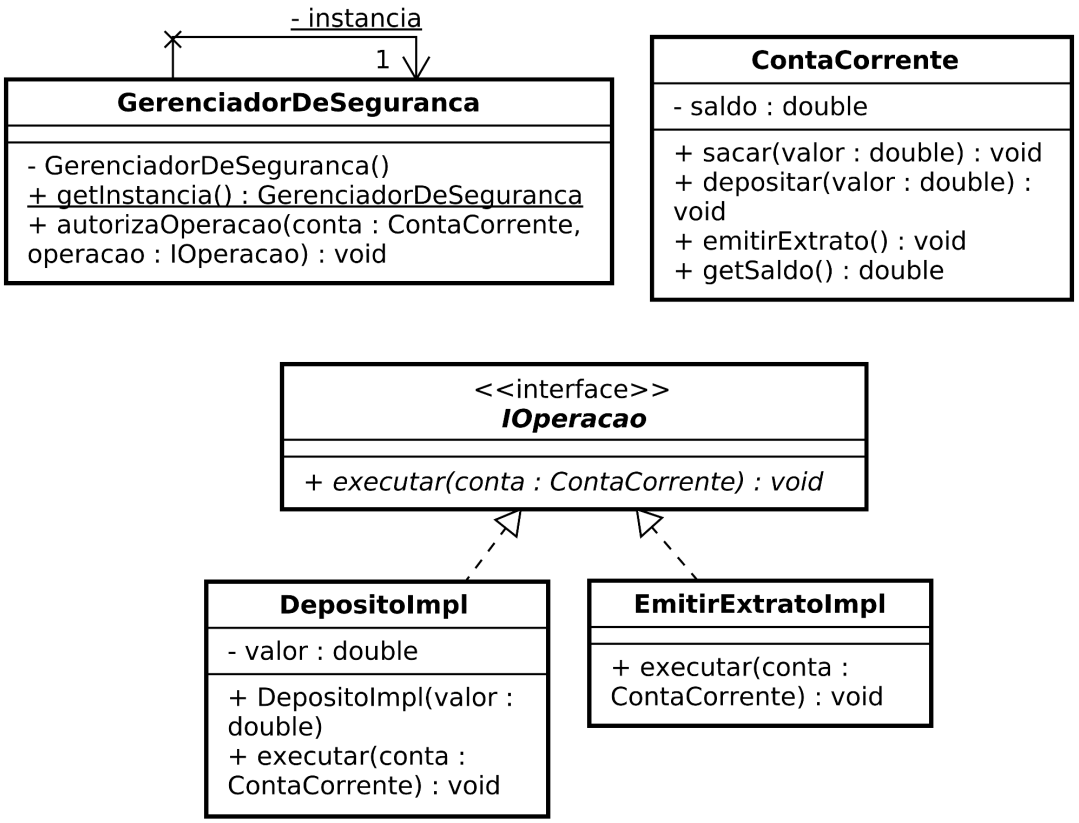
```
public class Principal {
    public static void main(String[] args) {
        try {
            SeguroService seguroService = new SeguroService();

            // "Seguro Básico"
            Cliente cliente1 = new Cliente("Cliente 1");
            cliente1.setAnosDeExperiencia(11);
            cliente1.setAcidentesRecentes(0);
            cliente1.setIdade(36);
            cliente1.setVeiculos(1);
            cliente1.setAnosBonus(2);
            Seguro seguro1 = seguroService.avaliar(cliente1);
            System.out.println(seguro1.getNome());

            // "Seguro Completo"
            Cliente cliente2 = new Cliente("Cliente 2");
```

```
        cliente2.setAnosDeExperiencia(10);
        cliente2.setAcidentesRecentes(1);
        cliente2.setIdade(32);
        cliente2.setVeiculos(2);
        cliente2.setAnosBonus(6);
        Seguro seguro2 = seguroService.avaliar(cliente2);
        System.out.println(seguro2.getNome());

        // "Seguro Premium"
        Cliente cliente3 = new Cliente("Cliente 3");
        cliente3.setAnosDeExperiencia(5);
        cliente3.setAcidentesRecentes(0);
        cliente3.setIdade(40);
        cliente3.setVeiculos(3);
        cliente3.setAnosBonus(2);
        Seguro seguro3 = seguroService.avaliar(cliente3);
        System.out.println(seguro3.getNome());

        Cliente cliente4 = new Cliente("Cliente 4");
        cliente4.setAnosDeExperiencia(0);
        cliente4.setAcidentesRecentes(1);
        cliente4.setIdade(18);
        cliente4.setVeiculos(1);
        cliente4.setAnosBonus(0);

        Seguro seguro4 = seguroService.avaliar(cliente4);
        System.out.println(seguro4.getNome());

    } catch (RuntimeException e) {
        // "Nenhum tipo de seguro se aplica ao cliente Cliente 4"
        System.out.println(e.getMessage());
    }
```

```
  }
}
```

Solução do Exercício 59

Para desenvolver a solução que atende ao enunciado, aplicaremos diversos princípios de projeto SOLID e os padrões de design Strategy, Singleton, Open/Closed e Command. Veja o diagrama de classes da solução proposta.



A seguir, apresentamos o código em Java da solução.

Primeiramente, definimos a interface IOperacao que será implementada pelas classes de operações específicas, utilizando o padrão Strategy e Open/Closed.

```
public interface IOperacao {
  public void executar(ContaCorrente conta);
}
```

Implementamos a classe DepositoImpl, que representa a operação de depósito.

```
public class DepositoImpl implements IOperacao {
  private double valor;

  public DepositoImpl(double valor) {
    this.valor = valor;
  }

  @Override
  public void executar(ContaCorrente conta) {
    conta.depositar(valor);
  }
}
```

Implementamos a classe EmitirExtratoImpl, que representa a operação de emissão de extrato.

```
public class EmitirExtratoImpl implements IOperacao {
  @Override
  public void executar(ContaCorrente conta) {
    conta.emitirExtrato();
  }
}
```

A classe GerenciadorDeSeguranca será responsável por autorizar operações, seguindo o padrão Singleton.

```
public class GerenciadorDeSeguranca {
  private static GerenciadorDeSeguranca instancia;

  private GerenciadorDeSeguranca() {}

  public static GerenciadorDeSeguranca getInstancia() {
    if (instancia == null) {
      instancia = new GerenciadorDeSeguranca();
```

```
        }
        return instancia;
    }

    public void autorizaOperacao(ContaCorrente conta, IOperacao operacao) {
        // Implementação da lógica de autorização
        boolean autorizacao = true;
        if (autorizacao) {
            operacao.executar(conta);
        }
    }
}
```

Criamos a classe do tipo model, ContaCorrente que representa a conta bancária e suas operações.

```
public class ContaCorrente {
    private double saldo;

    public void sacar(double valor) {
        saldo -= valor;
    }

    public void depositar(double valor) {
        saldo += valor;
    }

    public void emitirExtrato() {
        System.out.println("Saldo: " + saldo);
    }

    public double getSaldo() {
        return saldo;
```

```
  }
}
```

A classe Principal demonstra o uso do sistema de gerenciamento de operações, encapsulando o tratamento de segurança.

```
public class Principal {
  public static void main(String[] args) {
    GerenciadorDeSeguranca gerenciador = GerenciadorDeSeguranca.getInstancia();
    ContaCorrente conta = new ContaCorrente();

    IOperacao deposito = new DepositoImpl(100);
    gerenciador.autorizaOperacao(conta, deposito);

    IOperacao emitirExtrato = new EmitirExtratoImpl();
    gerenciador.autorizaOperacao(conta, emitirExtrato);
  }
}
```

Solução do Exercício 60

Para resolver a questão, iremos implementar o padrão Facade para simplificar a interação com as partes do carro, fornecendo uma interface clara para as operações "DIRIGIR" e "ESTACIONAR". A entrada do motorista e dos passageiros é gerenciada pela CarroFacade, que encapsula a complexidade dos detalhes internos das partes do carro.

A seguir, é mostrado a codificação das classes das partes do carro (CintoDeSeguranca, Radio, Portas, Motor e Farol), que se mantiveram inalteradas.

```
public class CintoDeSeguranca {
  public void travar() {
    System.out.println("Cinto de segurança travado!");
  }

  public void destravar() {
```

```
        System.out.println("Cinto de segurança destravado!");
    }
}

public class Radio {
    public void ligar() {
        System.out.println("Ligando o rádio");
    }

    public void sintonizar(double frequencia) {
        System.out.println("Sintonizando o rádio na frequência " + frequencia);
    }

    public void desligar() {
        System.out.println("Desligando o rádio");
    }
}

public class Portas {
    public void travar() {
        System.out.println("Portas travadas!");
    }

    public void abrir() {
        System.out.println("Portas abertas!");
    }
}

public class Motor {
    public void ligar() {
        System.out.println("Motor ligado!");
    }
```

```
  public void desligar() {
    System.out.println("Motor desligado!");
  }
}

public class Farol {
  public void acender() {
    System.out.println("Farol aceso!");
  }

  public void apagar() {
    System.out.println("Farol apagado!");
  }
}
```

Agora, codificamos a classe CarroFacade que irá encapsular os métodos conforme o enunciado, mantendo expostos somente os métodos dirigir e estacionar.

```
public class CarroFacade {
  private CintoDeSeguranca cintoDeSeguranca;
  private Radio radio;
  private Portas portas;
  private Motor motor;
  private Farol farol;

  public CarroFacade() {
    this.cintoDeSeguranca = new CintoDeSeguranca();
    this.radio = new Radio();
    this.portas = new Portas();
    this.motor = new Motor();
    this.farol = new Farol();
  }
```

```
  public void dirigir(String motorista, String[] passageiros) {
    System.out.println("Motorista " + motorista + " entrando no carro");
    for (String passageiro : passageiros) {
      System.out.println("Passageiro " + passageiro + " entrando no carro");
    }
    portas.travar();
    cintoDeSeguranca.travar();
    motor.ligar();
    radio.ligar();
    farol.acender();
  }

  public void estacionar() {
    farol.apagar();
    radio.desligar();
    motor.desligar();
    cintoDeSeguranca.destravar();
    portas.abrir();
  }
}
```

Por fim, criamos a classe Principal que demonstra o uso da solução. Inicialmente, criamos uma ojeto da classe CarroFacade que representa um carro, em seguida criamos as strings com o nome do motorista e, posteriormente, dos passageiros. Chamamos o método dirigir e em seguida o método estacionar.

```
public class Principal {
  public static void main(String[] args) {
    CarroFacade carro = new CarroFacade();
    String motorista = "João";
    String[] passageiros = {"Maria", "Pedro", "Ana"};
```

```
        carro.dirigir(motorista, passageiros);
        System.out.println("Carro está em movimento...");
        carro.estacionar();
    }
}
```

Solução do Exercício 61

Para desenvolver a solução que atende ao enunciado, implementaremos dois padrões de projeto: Builder e Decorator. O padrão Builder será usado para criar instâncias de diferentes tipos de apartamentos, enquanto o padrão Decorator será utilizado para adicionar personalizações aos apartamentos. Além disso, utilizaremos o padrão Singleton para configurar o preço unitário do metro quadrado de forma flexível e acessível por todas as classes. A seguir, apresentamos o código em Java da solução:

Inicialmente criamos a classe Configuracao que utiliza o padrão Singleton para garantir que haja apenas uma instancia, e fornecerá um método para obter o preço unitário do metro quadrado.

```
public class Configuracao {
    private static Configuracao instancia;
    private double precoPorMetroQuadrado;

    private Configuracao() {
        this.precoPorMetroQuadrado = 3188.98;
    }

    public static Configuracao getInstancia() {
        if (instancia == null)
            instancia = new Configuracao();

        return instancia;
    }
```

```
  public double getPrecoPorMetroQuadrado() {
    return precoPorMetroQuadrado;
  }

  public void setPrecoPorMetroQuadrado(double precoPorMetroQuadrado) {
    this.precoPorMetroQuadrado = precoPorMetroQuadrado;
  }
}
```

Primeiro, definimos a interface IApartamento que será implementada pelas classes concretas representando os diferentes tipos de apartamentos.

```
public interface IApartamento {
  public double getPreco();
  public String getDescricao();
}
```

Em seguida criamos as classes concretas específicas que implementam IApartamento, utilizando a instância da classe Configuracao para obter o preço por metro quadrado.

```
public class ApartamentoPadraoImpl implements IApartamento {
  private double area;

  public ApartamentoPadraoImpl(double area) {
    this.area = area;
  }

  @Override
  public double getPreco() {
    double precoPorMetroQuadrado =
Configuracao.getInstancia().getPrecoPorMetroQuadrado();
    return area * precoPorMetroQuadrado;
  }
```

```
    @Override
    public String getDescricao() {
        return "Apartamento Padrão";
    }
}

public class KitnetImpl implements IApartamento {
    private double area;

    public KitnetImpl(double area) {
        this.area = area;
    }

    @Override
    public double getPreco() {
        double precoPorMetroQuadrado =
Configuracao.getInstancia().getPrecoPorMetroQuadrado();
        return area * precoPorMetroQuadrado;
    }

    @Override
    public String getDescricao() {
        return "Kitnet";
    }
}

public class FlatImpl implements IApartamento {
    private double area;

    public FlatImpl(double area) {
        this.area = area;
    }
```

```
  @Override
  public double getPreco() {
      double precoPorMetroQuadrado =
Configuracao.getInstancia().getPrecoPorMetroQuadrado();
      return area * precoPorMetroQuadrado;
  }

  @Override
  public String getDescricao() {
      return "Flat";
  }
}
```

Para criar instâncias dos apartamentos, implementamos o padrão Builder. Criamos interfaces e classes concretas para construir apartamentos específicos.

```
public interface IApartamentoBuilder {
  public void setArea(double area);
  public IApartamento build();
}

public class ApartamentoPadraoBuilder implements IApartamentoBuilder {
  private double area;

  @Override
  public void setArea(double area) {
      this.area = area;
  }

  @Override
  public IApartamento build() {
      return new ApartamentoPadraoImpl(area);
```

```
  }
}

public class KitnetBuilder implements IApartamentoBuilder {
  private double area;

  @Override
  public void setArea(double area) {
    this.area = area;
  }

  @Override
  public IApartamento build() {
    return new KitnetImpl(area);
  }
}

public class FlatBuilder implements IApartamentoBuilder {
  private double area;

  @Override
  public void setArea(double area) {
    this.area = area;
  }

  @Override
  public IApartamento build() {
    return new FlatImpl(area);
  }
}
```

Para adicionar personalizações aos apartamentos, utilizamos o padrão Decorator. Definimos uma classe abstrata ApartamentoDecorator e classes concretas para diferentes tipos de personalizações.

```
public abstract class ApartamentoDecorator implements IApartamento {
  protected IApartamento apartamento;

  public ApartamentoDecorator(IApartamento apartamento) {
    this.apartamento = apartamento;
  }

  @Override
  public double getPreco() {
    return apartamento.getPreco();
  }

  @Override
  public String getDescricao() {
    return apartamento.getDescricao();
  }
}

public class ChurrasqueiraDecorator extends ApartamentoDecorator {
  public ChurrasqueiraDecorator(IApartamento apartamento) {
    super(apartamento);
  }

  @Override
  public double getPreco() {
    return super.getPreco() + 5000;//valor fictício
  }

  @Override
```

```
    public String getDescricao() {
        return super.getDescricao() + ", com Churrasqueira";
    }
}

public class PiscinaDecorator extends ApartamentoDecorator {
    public PiscinaDecorator(IApartamento apartamento) {
        super(apartamento);
    }

    @Override
    public double getPreco() {
        return super.getPreco() + 15000;//valor fictício
    }

    @Override
    public String getDescricao() {
        return super.getDescricao() + ", com Piscina";
    }
}
```

Por fim, criamos a classe Principal com o método main que demonstra o da solução. Inicialmente configuramos o preço do metro quadrado através da instância única de Configuracao, garantindo consistência e facilidade de alteração do valor. Utilizamos o padrão Builder para criar um apartamento do tipo padrão, definindo a área e construindo a instância do apartamento. Exibimos a descrição e o preço inicial do apartamento padrão. Em seguida, aplicamos personalizações usando o padrão Decorator. Adicionamos uma churrasqueira ao apartamento e exibimos a descrição e o preço atualizados.

```
public class Principal {
    public static void main(String[] args) {
        Configuracao configuracao = Configuracao.getInstancia();
        configuracao.setPrecoPorMetroQuadrado(3500.00);
```

```
        IApartamentoBuilder builder = new ApartamentoPadraoBuilder();
        builder.setArea(100);
        IApartamento apartamento = builder.build();

        System.out.println("Descrição: " + apartamento.getDescricao());
        System.out.println("Preço: R$ " + apartamento.getPreco());

        apartamento = new churrasqueiraDecorator(apartamento);

        System.out.println("Descrição após decoração: " + apartamento.getDescricao());
        System.out.println("Preço após decoração: R$ " + apartamento.getPreco());
    }
}
```

Esse fluxo demonstra a criação e personalização de apartamentos de forma flexível e extensível.

Solução do Exercício 62

Para resolver o problema de pesquisa de títulos de filmes e séries na plataforma de streaming de vídeo, aplicamos o padrão de projeto Facade. A implementação da classe PesquisadorDeConteudo unifica o acesso às diferentes fontes de dados, facilitando a busca de títulos sem que o usuário precise se preocupar com as diferentes interfaces e estruturas de dados.

Primeiramente, criamos a classe Conteudo, que representa os dados retornados pelas pesquisas. Esta classe possui os atributos titulo e tipo, além de seus respectivos métodos getters e setters.

```
public class Conteudo {
    private String titulo;
    private String tipo;
    public Conteudo(String titulo, String tipo) {
        this.titulo = titulo;
```

```
        this.tipo = tipo;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getTipo() {
        return tipo;
    }
    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
}
```

Em seguida, implementamos as classes que se mantiveram inalteradas e são responsáveis por acessar as diferentes fontes de dados: BancoDeDadoInterno, ApiDeFilme e ApiDeSerie. Cada uma dessas classes possui o método pesquisarPorTitulo, que retorna uma lista de objetos Conteudo de acordo com o título pesquisado.

```
public class BancoDeDadoInterno {
    public List<Conteudo> pesquisarPortitulo(String titulo) {
        List<Conteudo> resultados = new ArrayList<>();

        if (titulo.equalsIgnoreCase("vingadores")) {
            resultados.add(new Conteudo("Vingadores", "filme"));
            resultados.add(new Conteudo("Vingadores 2: A Era de Ultron", "filme"));
            resultados.add(new Conteudo("Vingadores: Guerra Infinita", "filme"));
        } else if (titulo.equalsIgnoreCase("homem-aranha")) {
            resultados.add(new Conteudo("Homem-Aranha: No Aranhaverso", "filme"));
            resultados.add(new Conteudo("Homem-Aranha: De Volta ao Lar", "filme"));
            resultados.add(new Conteudo("Homem-Aranha: Longe de Casa", "filme"));
```

```
        }

        return resultados;
    }
}

public class ApiDeFilme {
    public List<Conteudo> pesquisarPortitulo(String titulo) {
        List<Conteudo> resultados = new ArrayList<>();

        if (titulo.equalsIgnoreCase("vingadores")) {
            resultados.add(new Conteudo("Vingadores: Endgame", "filme"));
        } else if (titulo.equalsIgnoreCase("homem-aranha")) {
            resultados.add(new Conteudo("Homem-Aranha: No Aranhaverso", "filme"));
            resultados.add(new Conteudo("Homem-Aranha: De Volta ao Lar", "filme"));
            resultados.add(new Conteudo("Homem-Aranha: Longe de Casa", "filme"));
        }

        return resultados;
    }
}

public class ApiDeSerie {
    public List<Conteudo> pesquisarPortitulo(String titulo) {
        List<Conteudo> resultados = new ArrayList<>();

        if (titulo.equalsIgnoreCase("stranger things")) {
            resultados.add(new Conteudo("Stranger Things", "série"));
            resultados.add(new Conteudo("Stranger Things 2", "série"));
            resultados.add(new Conteudo("Stranger Things 3", "série"));
        } else if (titulo.equalsIgnoreCase("game of thrones")) {
            resultados.add(new Conteudo("Game of Thrones", "série"));
```

```
        }

        return resultados;
    }
}
```

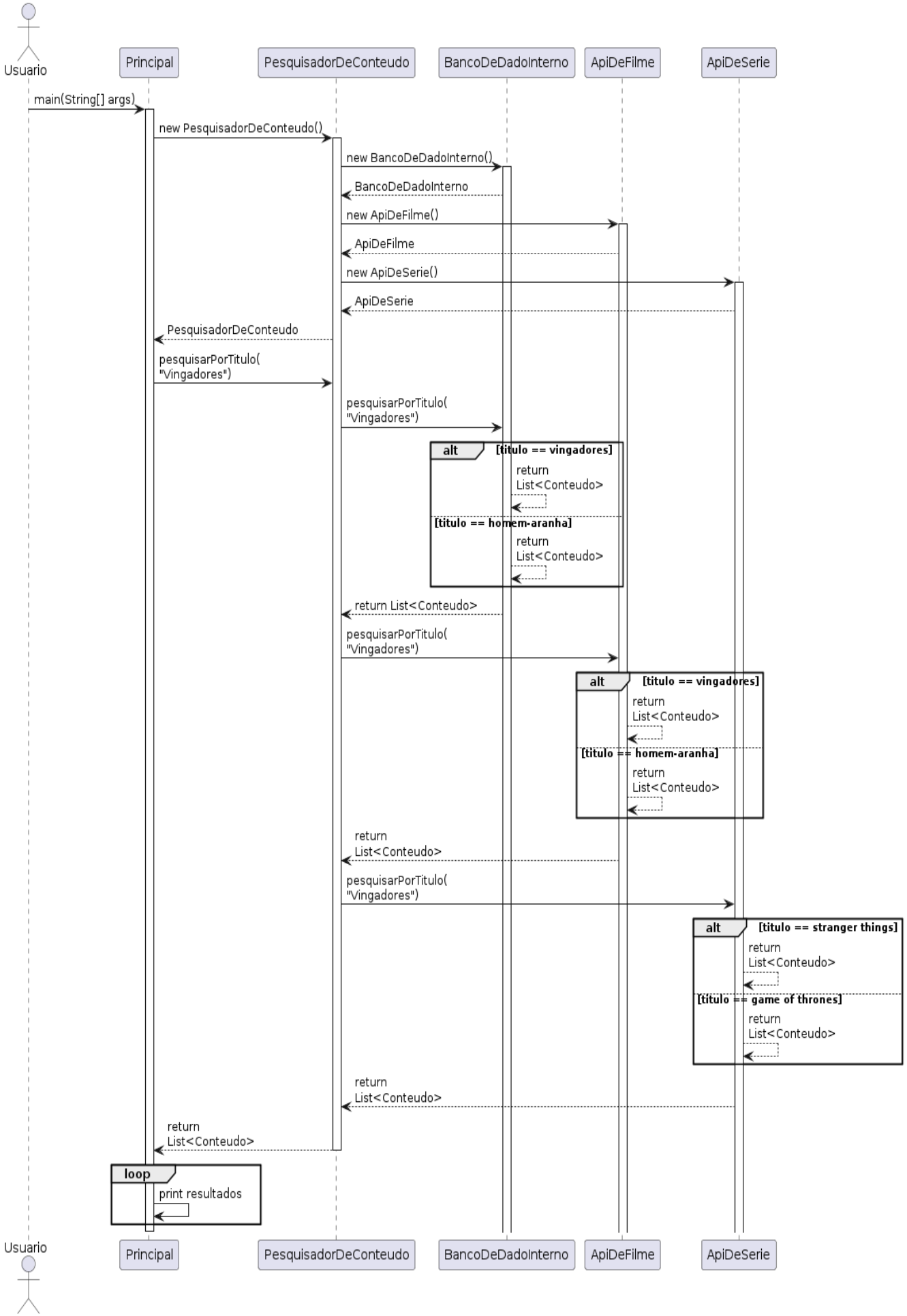
Agora, codidicamos a classe PesquisadorDeConteudo que atua como uma fachada, encapsulando as interações com as três fontes de dados. No construtor da classe, inicializamos as instâncias das classes de acesso a dados. O método pesquisarPorTitulo agrega os resultados das três fontes e retorna uma lista consolidada de objetos Conteudo.

```
public class PesquisadorDeConteudo {
    private BancoDeDadoInterno bancoDeDadosInterno;
    private ApiDeFilme apiDeFilmes;
    private ApiDeSerie apiDeSeries;

    public PesquisadorDeConteudo() {
        bancoDeDadosInterno = new BancoDeDadoInterno();
        apiDeFilmes = new ApiDeFilme();
        apiDeSeries = new ApiDeSerie();
    }

    public List<Conteudo> pesquisarPorTitulo(String titulo) {
        List<Conteudo> resultados = new ArrayList<>();

        resultados.addAll(bancoDeDadosInterno.pesquisarPorTitulo(titulo));
        resultados.addAll(apiDeFilmes.pesquisarPorTitulo(titulo));
        resultados.addAll(apiDeSeries.pesquisarPorTitulo(titulo));

        return resultados;
    }
}
```

Para demonstrar o uso da solução, criamos a classe principal Principal que contem o método main. Nesta classe, instanciamos um objeto PesquisadorDeConteudo e utilizamos o método pesquisarPorTitulo para buscar o título "Vingadores". Em seguida, iteramos sobre os resultados e exibimos o título e o tipo de cada conteúdo encontrado.

```
public class Principal {
  public static void main(String[] args) {
    PesquisadorDeConteudo pesquisador = new PesquisadorDeConteudo();
    List<Conteudo> resultados = pesquisador.pesquisarPorTitulo("Vingadores");

    for (Conteudo conteudo : resultados) {
      System.out.println(conteudo.getTitulo() + " (" + conteudo.getTipo() + ")");
    }
  }
}
```

O diagrama de sequencia da classe Principal é exibido a seguir.


Usuario
Principal
PesquisadorDeConteudo
BancoDeDadoInterno
ApiDeFilme
ApiDeSerie
main(String[] args)
new PesquisadorDeConteudo()
new BancoDeDadoInterno()
BancoDeDadoInterno
new ApiDeFilme()
ApiDeFilme
new ApiDeSerie()
ApiDeSerie
PesquisadorDeConteudo
pesquisarPorTitulo(
"Vingadores")
pesquisarPorTitulo(
"Vingadores")
alt
[titulo == vingadores]
return
List<Conteudo>
[titulo == homem-aranha]
return
List<Conteudo>
return List<Conteudo>
pesquisarPorTitulo(
"Vingadores")
alt
[titulo == vingadores]
return
List<Conteudo>
[titulo == homem-aranha]
return
List<Conteudo>
return
List<Conteudo>
pesquisarPorTitulo(
"Vingadores")
alt
[titulo == stranger things]
return
List<Conteudo>
[titulo == game of thrones]
return
List<Conteudo>
return
List<Conteudo>
return
List<Conteudo>
loop
print resultados
Usuario
Principal
PesquisadorDeConteudo
BancoDeDadoInterno
ApiDeFilme
ApiDeSerie

Solução do Exercício 63

Para implementar o exemplo prático utilizando o padrão de projeto Abstract Factory em Java, no qual os produtos sejam classes DAO (Data Access Object), começamos definindo a fábrica abstrata. Esta fábrica define os métodos para criar classes DAO de clientes e produtos. A interface IFabricaDao estabelece a estrutura para as fábricas concretas que criarão os DAOs específicos para cada tipo de conexão.

```
public interface IFabricaDao {
  public IDaoCliente criarDaoCliente();
  public IDaoProduto criarDaoProduto();
}
```

A seguir, implementamos a fábrica concreta para criar classes DAO com conexão JDBC. A classe FabricaDaoJdbcImpl implementa a interface IFabricaDao e cria instâncias de DaoClienteJdbcImpl e DaoProdutoJdbcImpl.

```
public class FabricaDaoJdbcImpl implements IFabricaDao {
  @Override
  public IDaoCliente criarDaoCliente() {
     return new DaoClienteJdbcImpl();
  }

  @Override
  public IDaoProduto criarDaoProduto() {
     return new DaoProdutoJdbcImpl();
  }
}
```

Em seguida, temos a fábrica concreta para criar classes DAO com conexão Hibernate. A classe FabricaDaoHibernateImpl também implementa a interface IFabricaDao e cria instâncias de DaoClienteHibernateImpl e DaoProdutoHibernateImpl.

```
public class FabricaDaoHibernateImpl implements IFabricaDao {
  @Override
```

```
    public IDaoCliente criarDaoCliente() {
        return new DaoClienteHibernateImpl();
    }

    @Override
    public IDaoProduto criarDaoProduto() {
        return new DaoProdutoHibernateImpl();
    }
}
```

Para os DAOs, definimos interfaces para DAO de Cliente IDaoCliente e DAO de Produto IDaoProduto. Essas interfaces declaram métodos para salvar e buscar clientes e produtos.

```
public interface IDaoCliente {
    public void salvarCliente(Cliente cliente);
    public Cliente buscarCliente(int id);
}

public interface IDaoProduto {
    public void salvarProduto(Produto produto);
    public Produto buscarProduto(int id);
}
```

Implementamos as classes DAO com conexão JDBC. As classes DaoClienteJdbcImpl e DaoProdutoJdbcImpl implementam as interfaces IaoCliente e IDaoProduto, respectivamente, utilizando JDBC para acesso a dados.

```
public class DaoClienteJdbcImpl implements IDaoCliente {
    @Override
    public void salvarCliente(Cliente cliente) {
        // Implementação para salvar cliente usando JDBC
    }

    @Override
```

```
    public Cliente buscarCliente(int id) {
        // Implementação para buscar cliente usando JDBC
        return null;
    }
}

public class DaoProdutoJdbcImpl implements IDaoProduto {
    @Override
    public void salvarProduto(Produto produto) {
        // Implementação para salvar produto usando JDBC
    }

    @Override
    public Produto buscarProduto(int id) {
        // Implementação para buscar produto usando JDBC
        return null;
    }
}
```

Em seguida, implementamos as classes DAO com conexão Hibernate. As classes DaoClienteHibernateImpl e DaoProdutoHibernateImpl implementam as interfaces IDaoCliente e IDaoProduto, respectivamente, utilizando Hibernate para acesso a dados.

```
public class DaoClienteHibernateImpl implements IDaoCliente {
    @Override
    public void salvarCliente(Cliente cliente) {
        // Implementação para salvar cliente usando Hibernate
    }

    @Override
    public Cliente buscarCliente(int id) {
        // Implementação para buscar cliente usando Hibernate
        return null;
```

```
  }
}

public class DaoProdutoHibernateImpl implements IDaoProduto {
  @Override
  public void salvarProduto(Produto produto) {
    // Implementação para salvar produto usando Hibernate
  }

  @Override
  public Produto buscarProduto(int id) {
    // Implementação para buscar produto usando Hibernate
    return null;
  }
}
```

Codificamos as classes do tipo model Cliente e Produto que representam as entidades no sistema. Essas classes possuem atributos básicos e seus respectivos métodos getters e setters.

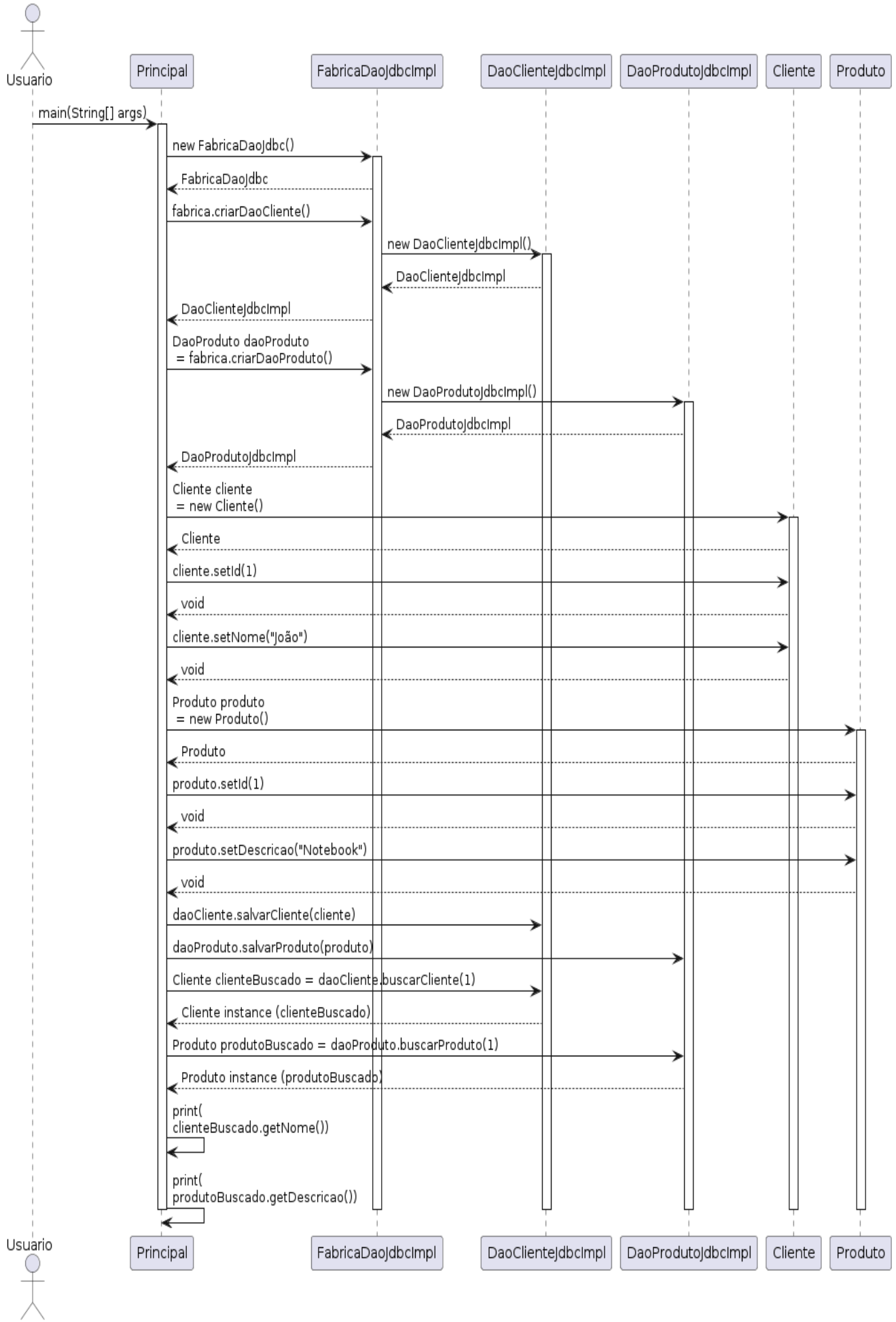
```
public class Cliente {
  private int id;
  private String nome;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getNome() {
```

```
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}

public class Produto {
    private int id;
    private String descricao;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
}
```

Finalmente, criamos a classe Principal para demonstrar o uso da solução. No método main, instanciamos a fábrica concreta FabricaDaoJdbcImpl e criamos DAOs para clientes e produtos. Em seguida, utilizamos esses DAOs para salvar e buscar instâncias de Cliente e Produto.

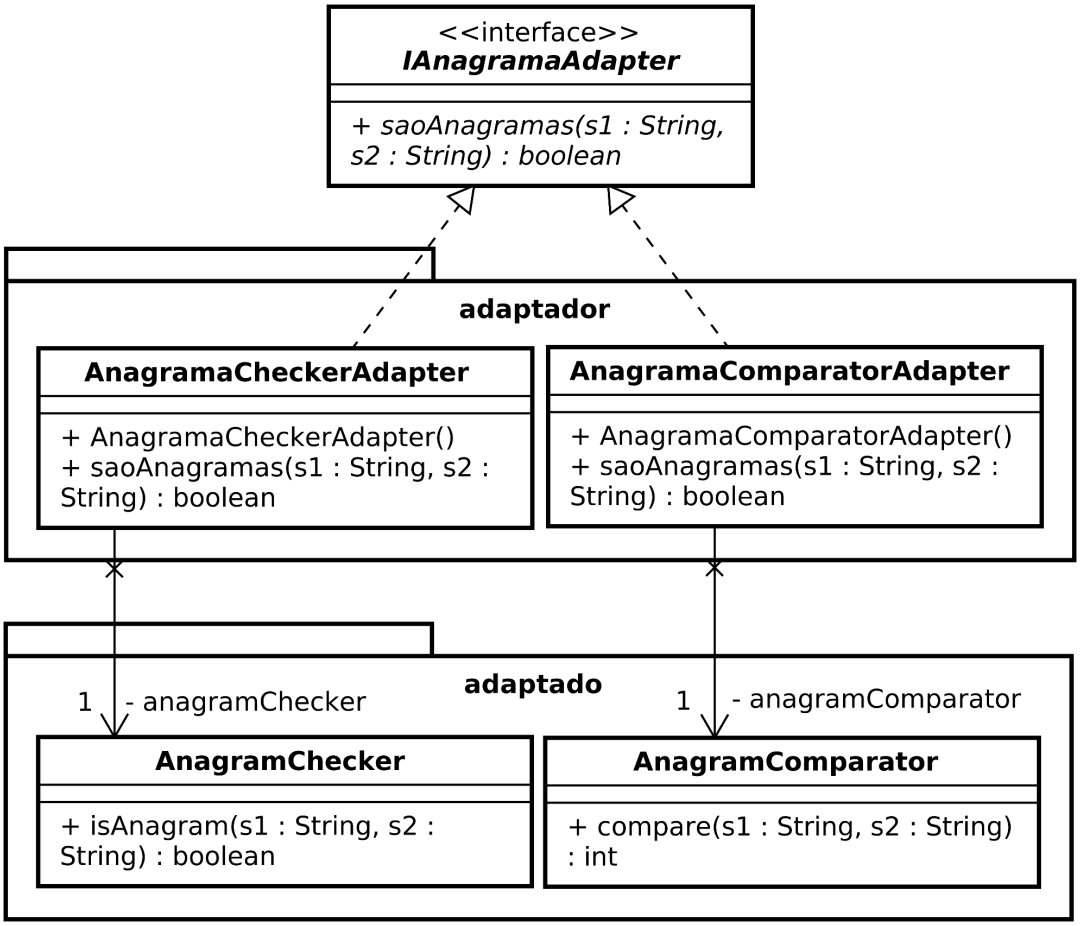
```
public class Principal {
  public static void main(String[] args) {
    IFabricaDao fabrica = new FabricaDaoJdbcImpl();

    IDaoCliente daoCliente = fabrica.criarDaoCliente();
    IDaoProduto daoProduto = fabrica.criarDaoProduto();

    Cliente cliente = new Cliente();
    cliente.setId(1);
    cliente.setNome("João");

    Produto produto = new Produto();
    produto.setId(1);
    produto.setDescricao("Notebook");

    daoCliente.salvarCliente(cliente);
    daoProduto.salvarProduto(produto);

    Cliente clienteBuscado = daoCliente.buscarCliente(1);
    Produto produtoBuscado = daoProduto.buscarProduto(1);

    System.out.println("Cliente buscado: " + clienteBuscado.getNome());
    System.out.println("Produto buscado: " + produtoBuscado.getDescricao());
  }
}
```

A seguir é mostrado o diagrama de sequência.


Usuario
Principal
FabricaDaoJdbcImpl
DaoClienteJdbcImpl
DaoProdutoJdbcImpl
Cliente
Produto
main(String[] args)
new FabricaDaoJdbc()
FabricaDaoJdbc
fabrica.criarDaoCliente()
new DaoClienteJdbcImpl()
DaoClienteJdbcImpl
DaoClienteJdbcImpl
DaoProduto daoProduto
= fabrica.criarDaoProduto()
new DaoProdutoJdbcImpl()
DaoProdutoJdbcImpl
DaoProdutoJdbcImpl
Cliente cliente
= new Cliente()
Cliente
cliente.setId(1)
void
cliente.setNome("João")
void
Produto produto
= new Produto()
Produto
produto.setId(1)
void
produto.setDescricao("Notebook")
void
daoCliente.salvarCliente(cliente)
daoProduto.salvarProduto(produto)
Cliente clienteBuscado = daoCliente.buscarCliente(1)
Cliente instance (clienteBuscado)
Produto produtoBuscado = daoProduto.buscarProduto(1)
Produto instance (produtoBuscado)
print(
clienteBuscado.getNome())
print(
produtoBuscado.getDescricao())
Usuario
Principal
FabricaDaoJdbcImpl
DaoClienteJdbcImpl
DaoProdutoJdbcImpl
Cliente
Produto

Solução do Exercício 64

Para resolver o problema proposto, acompanhe a ilustração no diagrama de classes abaixo onde utilizamos o padrão de projeto Adapter, que permite que classes com interfaces incompatíveis trabalhem juntas. Nosso objetivo é criar uma aplicação que possa comparar duas strings e determinar se elas são anagramas, utilizando duas bibliotecas diferentes: AnagramChecker e AnagramComparator. Para isso, criamos duas classes adaptadoras, AnagramaCheckerAdapter e AnagramaComparatorAdapter, que implementam a interface IAnagramaAdapter. Essa interface define o método saoAnagramas, que será implementado por ambas as classes adaptadoras.



Primeiro, criamos a interface IAnagramaAdapter, que define o método saoAnagramas. Esse método recebe duas strings como parâmetros e retorna um booleano indicando se as strings são anagramas.

```
public interface IAnagramaAdapter {
```

```
  public boolean saoAnagramas(String s1, String s2);
}
```

Em seguida, implementamos a classe AnagramaCheckerAdapter, que utiliza a biblioteca AnagramChecker para verificar se as strings são anagramas. No construtor da classe, instanciamos a biblioteca AnagramChecker. O método saoAnagramas utiliza o método isAnagram da biblioteca para realizar a verificação.

```
public class AnagramaCheckerAdapter implements IAnagramaAdapter {
  private AnagramChecker anagramChecker;

  public AnagramaCheckerAdapter() {
    this.anagramChecker = new AnagramChecker();
  }

  @Override
  public boolean saoAnagramas(String s1, String s2) {
    return this.anagramChecker.isAnagram(s1, s2);
  }
}
```

A próxima classe é a AnagramaComparatorAdapter, que utiliza a biblioteca AnagramComparator para realizar a comparação das strings. No construtor da classe, instanciamos a biblioteca AnagramComparator. O método saoAnagramas utiliza o método compare da biblioteca, que retorna 0 se as strings forem anagramas. Comparamos o resultado e retornamos true se forem anagramas.

```
public class AnagramaComparatorAdapter implements IAnagramaAdapter {
  private AnagramComparator anagramComparator;

  public AnagramaComparatorAdapter() {
    this.anagramComparator = new AnagramComparator();
  }
```

```
  @Override
  public boolean saoAnagramas(String s1, String s2) {
    int resultado = this.anagramComparator.compare(s1, s2);
    return resultado == 0;
  }
}
```

As dependências Maven necessárias para incluir as bibliotecas AnagramChecker e AnagramComparator são as seguintes.

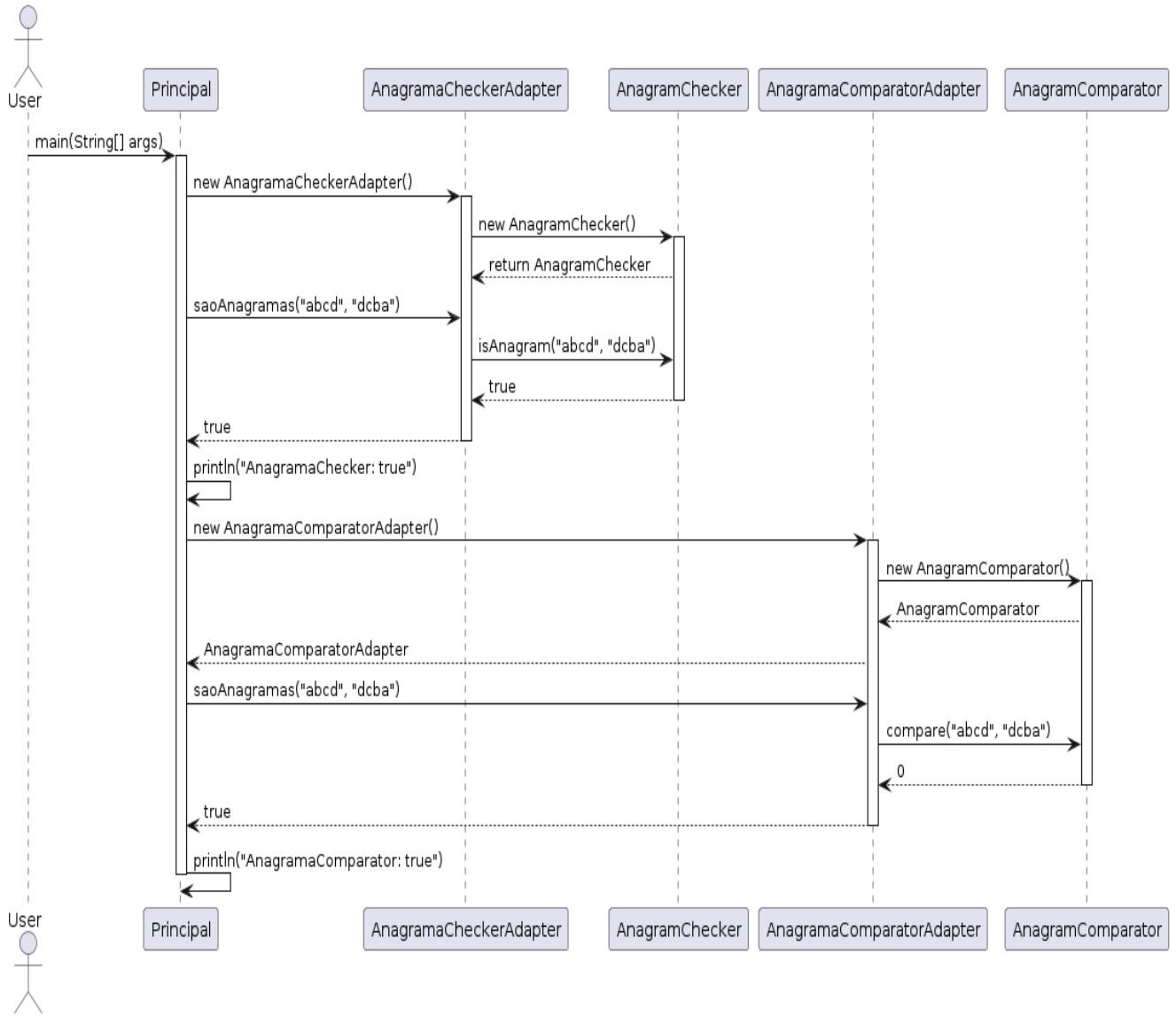
```
<dependency>
  <groupId>com.github.anagramchecker</groupId>
  <artifactId>anagram-checker</artifactId>
  <version>1.0.0</version>
</dependency>
<dependency>
  <groupId>com.github.anagramcomparator</groupId>
  <artifactId>anagram-comparator</artifactId>
  <version>1.0.0</version>
</dependency>
```

Finalmente, criamos a classe principal Principal para demonstrar o uso das classes adaptadoras. No método main, instanciamos os adaptadores AnagramaCheckerAdapter e AnagramaComparatorAdapter, e utilizamos o método saoAnagramas para verificar se duas strings são anagramas, imprimindo os resultados.

```
public class Principal {
  public static void main(String[] args) {
    IAnagramaAdapter anagramaCheckerAdapter = new AnagramaCheckerAdapter();
    boolean resultado1 = anagramaCheckerAdapter.saoAnagramas("abcd", "dcba");
    System.out.println("AnagramaChecker: " + resultado1);

    IAnagramaAdapter anagramaComparatorAdapter = new AnagramaComparatorAdapter();
    boolean resultado2 = anagramaComparatorAdapter.saoAnagramas("abcd", "dcba");
```

```
        System.out.println("AnagramaComparator: " + resultado2);
    }
}
```

Entenda o fluxo de execução do método main no diagrama de sequência abaixo:



Solução do Exercício 65

A proposta deste exercício é implementar o padrão de projeto Adapter para adaptar as bibliotecas Apache Commons CSV e OpenCSV para leitura de arquivos CSV. A interface de leitura deve possuir métodos para ler todos os registros de um arquivo e para ler o próximo registro.

A interface CSVReader define os métodos que devem ser implementados pelos adaptadores. Esta interface possui dois métodos: readAll, que lê todos os registros de um arquivo CSV, e readNext, que lê o próximo registro do arquivo CSV.

```
public interface CSVReader {
  List<String[]> readAll(String filePath);
  String[] readNext(String filePath);
}
```

A classe ApacheCommonsCSVAdapter implementa a interface CSVReader e define os métodos readAll e readNext utilizando as classes da biblioteca Apache Commons CSV. O método readAll abre o arquivo CSV e utiliza um CSVParser para iterar sobre os registros, adicionando cada registro a uma lista de arrays de strings. O método readNext abre o arquivo CSV e utiliza um CSVParser para ler o próximo registro.

```
import org.apache.commons.csv.CSVFormat;
import org.apache.commons.csv.CSVParser;
import org.apache.commons.csv.CSVRecord;
import java.io.IOException;
import java.io.Reader;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ApacheCommonsCSVAdapter implements CSVReader {
  @Override
  public List<String[]> readAll(String filePath) {
    List<String[]> rows = new ArrayList<>();
    try (Reader reader = Files.newBufferedReader(Paths.get(filePath));
        CSVParser parser = new CSVParser(reader, CSVFormat.DEFAULT)) {
```

```
            for (CSVRecord record : parser) {
                String[] row = new String[record.size()];
                for (int i = 0; i < record.size(); i++) {
                    row[i] = record.get(i);
                }
                rows.add(row);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return rows;
    }

    @Override
    public String[] readNext(String filePath) {
        try (Reader reader = Files.newBufferedReader(Paths.get(filePath));
             CSVParser parser = new CSVParser(reader, CSVFormat.DEFAULT)) {
            Iterator<CSVRecord> iterator = parser.iterator();
            if (iterator.hasNext()) {
                CSVRecord record = iterator.next();
                String[] row = new String[record.size()];
                for (int i = 0; i < record.size(); i++) {
                    row[i] = record.get(i);
                }
                return row;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return new String[0];
    }
}
```

A classe OpenCSVAdapter implementa a interface CSVReader e define os métodos readAll e readNext utilizando a classe CSVReader da biblioteca OpenCSV. O método readAll abre o arquivo CSV e utiliza um CSVReader para iterar sobre os registros, adicionando cada registro a uma lista de arrays de strings. O método readNext abre o arquivo CSV e utiliza um CSVReader para ler o próximo registro.

```
import com.opencsv.CSVReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class OpenCSVAdapter implements CSVReader {
  @Override
  public List<String[]> readAll(String filePath) {
    List<String[]> rows = new ArrayList<>();
    try (CSVReader reader = new CSVReader(new FileReader(filePath))) {
      String[] row;
      while ((row = reader.readNext()) != null) {
        rows.add(row);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    return rows;
  }

  @Override
  public String[] readNext(String filePath) {
    try (CSVReader reader = new CSVReader(new FileReader(filePath))) {
      String[] row = reader.readNext();
      if (row != null) {
        return row;
```

```
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return new String[0];
  }
}
```

A classe Principal demonstra o uso das classes adaptadoras ApacheCommonsCSVAdapter e OpenCSVAdapter. No método main, primeiramente, é criado um objeto ApacheCommonsCSVAdapter para ler um arquivo CSV utilizando a biblioteca Apache Commons CSV. O método readAll é chamado para ler todos os registros do arquivo e cada linha lida é impressa no console. Em seguida, é criado um objeto OpenCSVAdapter para ler o mesmo arquivo CSV utilizando a biblioteca OpenCSV. O método readNext é chamado para ler o próximo registro do arquivo e a linha lida é impressa no console.

```
public class Principal {
  public static void main(String[] args) {
    CSVReader apacheReader = new ApacheCommonsCSVAdapter();
    List<String[]> apacheRows = apacheReader.readAll("file.csv");
    for (String[] row : apacheRows) {
      System.out.println(String.join(", ", row));
    }

    CSVReader openCSVReader = new OpenCSVAdapter();
    String[] openCSVRow = openCSVReader.readNext("file.csv");
    System.out.println(String.join(", ", openCSVRow));
  }
}
```

As dependências Maven para as bibliotecas Apache Commons CSV e OpenCSV são as seguintes:

```
<dependencies>
```

```
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-csv</artifactId>
        <version>1.8</version>
    </dependency>

    <dependency>
        <groupId>com.opencsv</groupId>
        <artifactId>opencsv</artifactId>
        <version>7.1</version>
    </dependency>
</dependencies>
```

Solução do Exercício 66

Primeiramente criamos a interface IJsonSerializer, que define um método chamado serialize que recebe um objeto e retorna sua representação em JSON como string.

```
public interface IJsonSerializer {
    public String serializar(Object obj);
}
```

A classe GsonAdapter implementa a interface IJsonSerializer e utiliza a biblioteca Gson para realizar a serialização de objetos para o formato JSON.

```
import com.google.gson.Gson;

public class GsonAdapter implements IJsonSerializer {
    private Gson gson = new Gson();

    @Override
    public String serializar(Object obj) {
        return gson.toJson(obj);
    }
```

```
}
```

A classe JacksonAdapter implementa a interface IJsonSerializer eutiliza a biblioteca Jackson para realizar a serialização de objetos para o formato JSON. Em caso de erro na serialização, uma exceção JsonProcessingException é capturada e uma string vazia é retornada.

```
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonAdapter implements IJsonSerializer {
  private ObjectMapper mapper = new ObjectMapper();

  @Override
  public String serializar(Object obj) {
    try {
      return mapper.writeValueAsString(obj);
    } catch (JsonProcessingException e) {
      e.printStackTrace();
      return "";
    }
  }
}
```

Adicione as seguintes dependências ao seu arquivo pom.xml para incluir as bibliotecas Gson e Jackson no projeto.

```
<dependencies>
  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.6</version>
  </dependency>

  <dependency>
```

```
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.11.3</version>
    </dependency>
</dependencies>
```

Para testar os adaptadores, criamos a classe Pessoa que possui os atributos nome, idade e cidade, e métodos getters e setters.

```
public class Pessoa {
  private String nome;
  private int idade;
  private String cidade;

  public Pessoa(String nome, int idade, String cidade) {
     this.nome = nome;
     this.idade = idade;
     this.cidade = cidade;
  }

  public String getNome() {
     return nome;
  }

  public void setNome(String nome) {
     this.nome = nome;
  }

  public int getIdade() {
     return idade;
  }

  public void setIdade(int idade) {
```

```
        this.idade = idade;
    }

    public String getCidade() {
        return cidade;
    }

    public void setCidade(String cidade) {
        this.cidade = cidade;
    }
}
```

A classe Principal demonstra como utilizar as classes adaptadoras GsonAdapter e JacksonAdapter. No método main, são criadas instâncias dos adaptadores e uma instância da classe Pessoa. A serialização do objeto pessoa é realizada usando ambos os adaptadores, e os resultados são impressos no console.

```
public class Principal {
    public static void main(String[] args) {
        IJsonSerializer gsonSerializador = new GsonAdapter();
        IJsonSerializer jacksonSerializador = new JacksonAdapter();

        Pessoa pessoa = new Pessoa("João", 30, "Nova Iorque");

        String gsonJson = gsonSerializador.serializar(pessoa);
        System.out.println("Gson JSON: " + gsonJson);

        String jacksonJson = jacksonSerializador.serializar(pessoa);
        System.out.println("Jackson JSON: " + jacksonJson);
    }
}
```

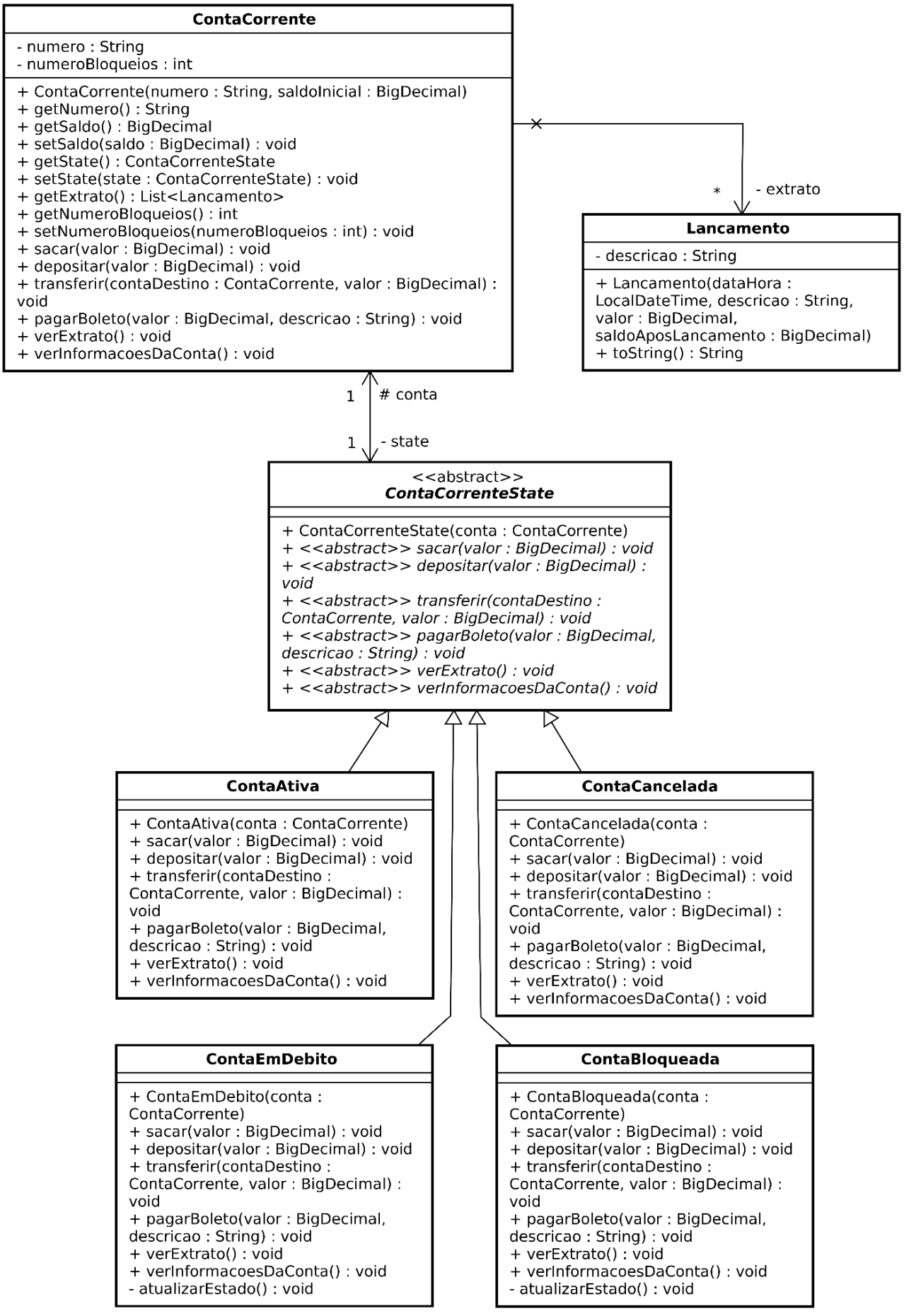
Espera-se exibir no console o seguinte conteúdo após a execução do método main.

Gson JSON: {"nome":"João","idade":30,"cidade":"Nova Iorque"}

Jackson JSON: {"nome":"João","idade":30,"cidade":"Nova Iorque"}

Solução do Exercício 67

Para resolver essa questão, acompanhe no diagrama de classes abaixo que utilizaremos o padrão de projeto State para representar os diferentes estados de uma conta corrente (Ativa, Em Débito, Bloqueada, Cancelada). A classe ContaCorrenteState será uma classe abstrata que define o comportamento básico de uma conta corrente e fornecerá uma interface comum para os diferentes estados. Cada estado específico (ContaAtiva, ContaEmDebito, ContaBloqueada, ContaCancelada) implementará essa interface e definirá o comportamento adequado para as operações permitidas no respectivo estado. A classe ContaCorrente manterá uma referência ao estado atual e delegará as operações para o estado atual, garantindo que o comportamento da conta seja consistente com seu estado atual.


ContaCorrente
- numero : String
- numeroBloqueios : int
+ ContaCorrente(numero : String, saldoInicial : BigDecimal)
+ getNumero() : String
+ getSaldo() : BigDecimal
+ setSaldo(saldo : BigDecimal) : void
+ getState() : ContaCorrenteState
+ setState(state : ContaCorrenteState) : void
+ getExtrato() : List<Lancamento>
+ getNumeroBloqueios() : int
+ setNumeroBloqueios(numeroBloqueios : int) : void
+ sacar(valor : BigDecimal) : void
+ depositar(valor : BigDecimal) : void
+ transferir(contaDestino : ContaCorrente, valor : BigDecimal) : void
+ pagarBoleto(valor : BigDecimal, descricao : String) : void
+ verExtrato() : void
+ verInformacoesDaConta() : void
*
- extrato
Lancamento
- descricao : String
+ Lancamento(dataHora : LocalDateTime, descricao : String, valor : BigDecimal, saldoAposLancamento : BigDecimal)
+ toString() : String
1
conta
1
- state
<<abstract>>
ContaCorrenteState
+ ContaCorrenteState(conta : ContaCorrente)
+ <<abstract>> sacar(valor : BigDecimal) : void
+ <<abstract>> depositar(valor : BigDecimal) : void
+ <<abstract>> transferir(contaDestino : ContaCorrente, valor : BigDecimal) : void
+ <<abstract>> pagarBoleto(valor : BigDecimal, descricao : String) : void
+ <<abstract>> verExtrato() : void
+ <<abstract>> verInformacoesDaConta() : void
ContaAtiva
+ ContaAtiva(conta : ContaCorrente)
+ sacar(valor : BigDecimal) : void
+ depositar(valor : BigDecimal) : void
+ transferir(contaDestino : ContaCorrente, valor : BigDecimal) : void
+ pagarBoleto(valor : BigDecimal, descricao : String) : void
+ verExtrato() : void
+ verInformacoesDaConta() : void
ContaCancelada
+ ContaCancelada(conta : ContaCorrente)
+ sacar(valor : BigDecimal) : void
+ depositar(valor : BigDecimal) : void
+ transferir(contaDestino : ContaCorrente, valor : BigDecimal) : void
+ pagarBoleto(valor : BigDecimal, descricao : String) : void
+ verExtrato() : void
+ verInformacoesDaConta() : void
ContaEmDebito
+ ContaEmDebito(conta : ContaCorrente)
+ sacar(valor : BigDecimal) : void
+ depositar(valor : BigDecimal) : void
+ transferir(contaDestino : ContaCorrente, valor : BigDecimal) : void
+ pagarBoleto(valor : BigDecimal, descricao : String) : void
+ verExtrato() : void
+ verInformacoesDaConta() : void
- atualizarEstado() : void
ContaBloqueada
+ ContaBloqueada(conta : ContaCorrente)
+ sacar(valor : BigDecimal) : void
+ depositar(valor : BigDecimal) : void
+ transferir(contaDestino : ContaCorrente, valor : BigDecimal) : void
+ pagarBoleto(valor : BigDecimal, descricao : String) : void
+ verExtrato() : void
+ verInformacoesDaConta() : void
- atualizarEstado() : void

A classe ContaCorrenteState é uma classe abstrata que define o comportamento básico de uma conta corrente. Ela fornece uma interface comum para os diferentes estados de uma conta corrente (Ativa, Em Débito, Bloqueada, Cancelada).

```
import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

public abstract class ContaCorrenteState {
  protected ContaCorrente conta;

  public ContaCorrenteState(ContaCorrente conta) {
    this.conta = conta;
  }

  public abstract void sacar(BigDecimal valor);
  public abstract void depositar(BigDecimal valor);
  public abstract void transferir(ContaCorrente contaDestino, BigDecimal valor);
  public abstract void pagarBoleto(BigDecimal valor, String descricao);
  public abstract void verExtrato();
  public abstract void verInformacoesDaConta();
}
```

A classe ContaAtiva herda de ContaCorrenteState e implementa os métodos para uma conta ativa. Esta classe permite saques, depósitos, transferências e pagamento de boletos, desde que o saldo seja suficiente.

```
public class ContaAtiva extends ContaCorrenteState {
  public ContaAtiva(ContaCorrente conta) {
    super(conta);
  }

  @Override
```

```
  public void sacar(BigDecimal valor) {
    if (valor.compareTo(conta.getSaldo()) > 0) {
      System.out.println("Saldo insuficiente para realizar o saque");
      return;
    }
    conta.setSaldo(conta.getSaldo().subtract(valor));
    conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Saque", valor,
conta.getSaldo()));
  }

  @Override
  public void depositar(BigDecimal valor) {
    conta.setSaldo(conta.getSaldo().add(valor));
    conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Depósito", valor,
conta.getSaldo()));
  }

  @Override
  public void transferir(ContaCorrente contaDestino, BigDecimal valor) {
    if (valor.compareTo(conta.getSaldo()) > 0) {
      System.out.println("Saldo insuficiente para realizar a transferência");
      return;
    }
    conta.setSaldo(conta.getSaldo().subtract(valor));
    contaDestino.setSaldo(contaDestino.getSaldo().add(valor));
    conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Transferência para
conta " + contaDestino.getNumero(), valor, conta.getSaldo()));
    contaDestino.getExtrato().add(new Lancamento(LocalDateTime.now(), "Transferência da
conta " + conta.getNumero(), valor, contaDestino.getSaldo()));
  }

  @Override
```

```
  public void pagarBoleto(BigDecimal valor, String descricao) {
    if (valor.compareTo(conta.getSaldo()) > 0) {
      System.out.println("Saldo insuficiente para pagar o boleto");
      return;
    }
    conta.setSaldo(conta.getSaldo().subtract(valor));
    conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Pagamento de boleto " +
descricao, valor, conta.getSaldo()));
  }

  @Override
  public void verExtrato() {
    System.out.println("Extrato da conta " + conta.getNumero() + ":");
    for (Lancamento l : conta.getExtrato()) {
      System.out.println(l);
    }
  }

  @Override
  public void verInformacoesDaConta() {
    System.out.println("Informações da conta " + conta.getNumero() + ":");
    System.out.println("Saldo atual: " + conta.getSaldo());
    System.out.println("Situação: ativa");
    System.out.println("Número de bloqueios: " + conta.getNumeroBloqueios());
  }
}
```

A classe ContaEmDebito também herda de ContaCorrenteState e representa o estado em que a conta tem saldo negativo. Ela permite saques, depósitos, transferências e pagamento de boletos até o limite de -R$ 1500,00. Além disso, aplica uma taxa de débito de 5% para operações com saldo negativo.

```
public class ContaEmDebito extends ContaCorrenteState {
```

```
  public ContaEmDebito(ContaCorrente conta) {
    super(conta);
  }

  @Override
  public void sacar(BigDecimal valor) {
    BigDecimal saldoDisponivel = conta.getSaldo().add(new BigDecimal("1500.00"));
    if (valor.compareTo(saldoDisponivel) > 0) {
      System.out.println("Saldo insuficiente para realizar o saque");
      return;
    }
    conta.setSaldo(conta.getSaldo().subtract(valor));
    conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Saque", valor,
conta.getSaldo()));
    if (conta.getSaldo().compareTo(new BigDecimal("-1500.00")) < 0) {
      BigDecimal taxaDebito = valor.multiply(new BigDecimal("0.05"));
      conta.setSaldo(conta.getSaldo().subtract(taxaDebito));
      conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Taxa de débito",
taxaDebito, conta.getSaldo()));
    }
    atualizarEstado();
  }

  @Override
  public void depositar(BigDecimal valor) {
    conta.setSaldo(conta.getSaldo().add(valor));
    conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Depósito", valor,
conta.getSaldo()));
    atualizarEstado();
  }

  @Override
```

```
  public void transferir(ContaCorrente contaDestino, BigDecimal valor) {
      BigDecimal saldoDisponivel = conta.getSaldo().add(new BigDecimal("1500.00"));
      if (valor.compareTo(saldoDisponivel) > 0) {
          System.out.println("Saldo insuficiente para realizar a transferência");
          return;
      }
      conta.setSaldo(conta.getSaldo().subtract(valor));
      contaDestino.setSaldo(contaDestino.getSaldo().add(valor));
      conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Transferência para
conta " + contaDestino.getNumero(), valor, conta.getSaldo()));
      contaDestino.getExtrato().add(new Lancamento(LocalDateTime.now(), "Transferência da
conta " + conta.getNumero(), valor, contaDestino.getSaldo()));
      if (conta.getSaldo().compareTo(new BigDecimal("-1500.00")) < 0) {
          BigDecimal taxaDebito = valor.multiply(new BigDecimal("0.05"));
          conta.setSaldo(conta.getSaldo().subtract(taxaDebito));
          conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Taxa de débito",
taxaDebito, conta.getSaldo()));
      }
      atualizarEstado();
  }

  @Override
  public void pagarBoleto(BigDecimal valor, String descricao) {
      BigDecimal saldoDisponivel = conta.getSaldo().add(new BigDecimal("1500.00"));
      if (valor.compareTo(saldoDisponivel) > 0) {
          System.out.println("Saldo insuficiente para pagar o boleto");
          return;
      }
      conta.setSaldo(conta.getSaldo().subtract(valor));
      conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Pagamento de boleto " +
descricao, valor, conta.getSaldo()));
      if (conta.getSaldo().compareTo(new BigDecimal("-1500.00")) < 0) {
```

```
            BigDecimal taxaDebito = valor.multiply(new BigDecimal("0.05"));
            conta.setSaldo(conta.getSaldo().subtract(taxaDebito));
            conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Taxa de débito",
taxaDebito, conta.getSaldo()));
        }
        atualizarEstado();
    }

    @Override
    public void verExtrato() {
        System.out.println("Extrato da conta " + conta.getNumero() + ":");
        for (Lancamento l : conta.getExtrato()) {
            System.out.println(l);
        }
    }

    @Override
    public void verInformacoesDaConta() {
        System.out.println("Informações da conta " + conta.getNumero() + ":");
        System.out.println("Saldo atual: " + conta.getSaldo());
        System.out.println("Situação: em débito");
        System.out.println("Número de bloqueios: " + conta.getNumeroBloqueios());
    }

    private void atualizarEstado() {
        if (conta.getSaldo().compareTo(new BigDecimal("-1500.00")) >= 0) {
            conta.setState(new ContaAtiva(conta));
        } else {
            if (conta.getNumeroBloqueios() >= 3) {
                conta.setState(new ContaCancelada(conta));
            } else {
                conta.setNumeroBloqueios(conta.getNumeroBloqueios() + 1);
```

```
            System.out.println("Atenção, a conta " + conta.getNumero() + " foi bloqueada " +
conta.getNumeroBloqueios() + " vezes, o limite são 3 bloqueios.");
            conta.setState(new ContaBloqueada(conta));
        }
    }
  }
}
```

A classe ContaBloqueada representa uma conta que está temporariamente bloqueada. Nesse estado, não são permitidas operações de saque, transferência ou pagamento de boletos, mas depósitos são permitidos para tentar regularizar a situação da conta.

```
public class ContaBloqueada extends ContaCorrenteState {
  public ContaBloqueada(ContaCorrente conta) {
    super(conta);
  }

  @Override
  public void sacar(BigDecimal valor) {
    System.out.println("Não é possível realizar saques em uma conta bloqueada");
  }

  @Override
  public void depositar(BigDecimal valor) {
    conta.setSaldo(conta.getSaldo().add(valor));
    conta.getExtrato().add(new Lancamento(LocalDateTime.now(), "Depósito", valor,
conta.getSaldo()));
    atualizarEstado();
  }

  @Override
  public void transferir(ContaCorrente contaDestino, BigDecimal valor) {
    System.out.println("Não é possível realizar transferências em uma conta bloqueada");
```

```
}

@Override
public void pagarBoleto(BigDecimal valor, String descricao) {
    System.out.println("Não é possível pagar boletos em uma conta bloqueada");
}

@Override
public void verExtrato() {
    System.out.println("Extrato da conta " + conta.getNumero() + ":");
    for (Lancamento l : conta.getExtrato()) {
        System.out.println(l);
    }
}

@Override
public void verInformacoesDaConta() {
    System.out.println("Informações da conta " + conta.getNumero() + ":");
    System.out.println("Saldo atual: " + conta.getSaldo());
    System.out.println("Situação: bloqueada");
    System.out.println("Número de bloqueios: " + conta.getNumeroBloqueios());
}

private void atualizarEstado() {
    if (conta.getSaldo().compareTo(new BigDecimal("0.00")) >= 0) {
        conta.setState(new ContaAtiva(conta));
    } else {
        if (conta.getSaldo().compareTo(new BigDecimal("-1500.00")) >= 0) {
            conta.setState(new ContaEmDebito(conta));
        } else {
            if (conta.getNumeroBloqueios() >= 3) {
                conta.setState(new ContaCancelada(conta));
```

```
        }
      }
    }
  }
}
```

A classe ContaCancelada representa uma conta que foi definitivamente cancelada após 3 bloqueios. Nesse estado, nenhuma operação é permitida.

```
public class ContaCancelada extends ContaCorrenteState {
  public ContaCancelada(ContaCorrente conta) {
    super(conta);
  }

  @Override
  public void sacar(BigDecimal valor) {
    System.out.println("Não é possível realizar saques em uma conta cancelada");
  }

  @Override
  public void depositar(BigDecimal valor) {
    System.out.println("Não é possível realizar depósitos em uma conta cancelada");
  }

  @Override
  public void transferir(ContaCorrente contaDestino, BigDecimal valor) {
    System.out.println("Não é possível realizar transferências em uma conta cancelada");
  }

  @Override
  public void pagarBoleto(BigDecimal valor, String descricao) {
    System.out.println("Não é possível pagar boletos em uma conta cancelada");
  }
```

```
  @Override
  public void verExtrato() {
    System.out.println("Extrato da conta " + conta.getNumero() + ":");
    for (Lancamento l : conta.getExtrato()) {
      System.out.println(l);
    }
  }

  @Override
  public void verInformacoesDaConta() {
    System.out.println("Informações da conta " + conta.getNumero() + ":");
    System.out.println("Saldo atual: " + conta.getSaldo());
    System.out.println("Situação: cancelada");
    System.out.println("Número de bloqueios: " + conta.getNumeroBloqueios());
  }
}
```

A classe ContaCorrente mantém o estado atual da conta e delega as operações para o estado atual.

```
public class ContaCorrente {
  private String numero;
  private BigDecimal saldo;
  private ContaCorrenteState state;
  private List<Lancamento> extrato;
  private int numeroBloqueios;

  public ContaCorrente(String numero, BigDecimal saldoInicial) {
    this.numero = numero;
    this.saldo = saldoInicial;
    this.state = new ContaAtiva(this);
    this.extrato = new ArrayList<>();
```

```
    this.numeroBloqueios = 0;
}

public String getNumero() {
    return numero;
}

public BigDecimal getSaldo() {
    return saldo;
}

public void setSaldo(BigDecimal saldo) {
    this.saldo = saldo;
}

public ContaCorrenteState getState() {
    return state;
}

public void setState(ContaCorrenteState state) {
    this.state = state;
}

public List<Lancamento> getExtrato() {
    return extrato;
}

public int getNumeroBloqueios() {
    return numeroBloqueios;
}

public void setNumeroBloqueios(int numeroBloqueios) {
```

```
        this.numeroBloqueios = numeroBloqueios;
    }

    public void sacar(BigDecimal valor) {
        state.sacar(valor);
    }

    public void depositar(BigDecimal valor) {
        state.depositar(valor);
    }

    public void transferir(ContaCorrente contaDestino, BigDecimal valor) {
        state.transferir(contaDestino, valor);
    }

    public void pagarBoleto(BigDecimal valor, String descricao) {
        state.pagarBoleto(valor, descricao);
    }

    public void verExtrato() {
        state.verExtrato();
    }

    public void verInformacoesDaConta() {
        state.verInformacoesDaConta();
    }
}
```

A classe Lancamento representa um registro de uma operação realizada na conta.

```
public class Lancamento {
    private LocalDateTime dataHora;
    private String descricao;
```

```
    private BigDecimal valor;
    private BigDecimal saldoAposLancamento;

        public Lancamento(LocalDateTime dataHora, String descricao, BigDecimal valor,
BigDecimal saldoAposLancamento) {
        this.dataHora = dataHora;
        this.descricao = descricao;
        this.valor = valor;
        this.saldoAposLancamento = saldoAposLancamento;
    }

    @Override
    public String toString() {
        return dataHora + " - " + descricao + ": " + valor + " (Saldo: " + saldoAposLancamento + ")";
    }
}
```

Agora, criamos classe Principal que demonstra o uso da solução.

```
import java.math.BigDecimal;

public class Principal {
    public static void main(String[] args) {
        ContaCorrente conta = new ContaCorrente("12345", new BigDecimal("1000.00"));

        conta.verInformacoesDaConta();
        conta.sacar(new BigDecimal("200.00"));
        conta.verInformacoesDaConta();
        conta.depositar(new BigDecimal("300.00"));
        conta.verInformacoesDaConta();
        conta.transferir(new ContaCorrente("67890", new BigDecimal("500.00")), new
BigDecimal("400.00"));
        conta.verInformacoesDaConta();
```

```
        conta.pagarBoleto(new BigDecimal("500.00"), "Internet");
        conta.verInformacoesDaConta();

        // Simulando entrar em débito
        conta.sacar(new BigDecimal("1600.00"));
        conta.verInformacoesDaConta();
        conta.depositar(new BigDecimal("1000.00"));
        conta.verInformacoesDaConta();
    }
}
```

Solução do Exercício 68

Para solucionar a implementação dos conversores de moeda, utilizaremos o padrão de projeto Chain of Responsibility. Cada handler concreto terá o nome da moeda de origem, o nome da moeda de destino e a taxa de conversão correspondente. Cada handler implementará um método aceitar que verifica se deve realizar a conversão ou passar para o próximo handler na cadeia. Se o método aceitar retornar verdadeiro, o método converter do handler deve receber a cotação, realizar a conversão, preencher o valor convertido e retorná-la. A chamada dos handlers será encapsulada em uma classe chamada ConversorService.

A classe Cotacao é responsável por representar a cotação de uma moeda, armazenando os valores e as informações necessárias para realizar a conversão entre duas moedas.

```
import java.time.OffsetDateTime;

public class Cotacao {
    private double valor;
    private String nomeMoedaAtual;
    private OffsetDateTime dataHoraLocal;
    private String nomeMoedaDestino;
    private double valorConvertido;
```

```
    public Cotacao(double valor, String nomeMoedaAtual, OffsetDateTime dataHoraLocal,
String nomeMoedaDestino) {
        this.valor = valor;
        this.nomeMoedaAtual = nomeMoedaAtual;
        this.dataHoraLocal = dataHoraLocal;
        this.nomeMoedaDestino = nomeMoedaDestino;
    }

    public double getValor() {
        return valor;
    }

    public String getNomeMoedaAtual() {
        return nomeMoedaAtual;
    }

    public OffsetDateTime getDataHoraLocal() {
        return dataHoraLocal;
    }

    public String getNomeMoedaDestino() {
        return nomeMoedaDestino;
    }

    public double getValorConvertido() {
        return valorConvertido;
    }

    public void setValorConvertido(double valorConvertido) {
        this.valorConvertido = valorConvertido;
    }
}
```

A classe abstrata ConversorHandler define a estrutura para os handlers de conversão de moeda. Cada handler será responsável por uma conversão específica, verificando se pode realizar a conversão e, se sim, processando-a. Se não puder realizar a conversão, passará a responsabilidade para o próximo handler.

```
public abstract class ConversorHandler {
  protected ConversorHandler proximo;

  public void setProximo(ConversorHandler proximo) {
    this.proximo = proximo;
  }

  public abstract boolean aceitar(Cotacao cotacao);
  public abstract Cotacao converter(Cotacao cotacao);
}
```

A classe ConversorRealParaDolarHandler é responsável por converter valores de Real para Dólar, utilizando uma taxa de conversão fixa. Ela verifica se a conversão pode ser realizada e, em caso afirmativo, realiza a conversão. Caso contrário, passa a responsabilidade para o próximo handler.

```
public class ConversorRealParaDolarHandler extends ConversorHandler {
  private static final double TAXA_CONVERSAO = 0.25;

  @Override
  public boolean aceitar(Cotacao cotacao) {
    return cotacao.getNomeMoedaAtual().equals("Real") &&
cotacao.getNomeMoedaDestino().equals("Dolar");
  }

  @Override
  public Cotacao converter(Cotacao cotacao) {
    if (aceitar(cotacao)) {
      double valorConvertido = cotacao.getValor() * TAXA_CONVERSAO;
```

```
            cotacao.setValorConvertido(valorConvertido);
            return cotacao;
        }
        return proximo.converter(cotacao);
    }
}
```

A classe ConversorRealParaEuroHandler é responsável por converter valores de Real para Euro, utilizando uma taxa de conversão fixa. Ela verifica se a conversão pode ser realizada e, em caso afirmativo, realiza a conversão. Caso contrário, passa a responsabilidade para o próximo handler.

```
public class ConversorRealParaEuroHandler extends ConversorHandler {
    private static final double TAXA_CONVERSAO = 0.20;

    @Override
    public boolean aceitar(Cotacao cotacao) {
        return cotacao.getNomeMoedaAtual().equals("Real") &&
cotacao.getNomeMoedaDestino().equals("Euro");
    }

    @Override
    public Cotacao converter(Cotacao cotacao) {
        if (aceitar(cotacao)) {
            double valorConvertido = cotacao.getValor() * TAXA_CONVERSAO;
            cotacao.setValorConvertido(valorConvertido);
            return cotacao;
        }
        return proximo.converter(cotacao);
    }
}
```

A classe ConversorDolarParaRealHandler é responsável por converter valores de Dólar para Real, utilizando uma taxa de conversão fixa. Ela verifica se a conversão pode ser realizada e,

em caso afirmativo, realiza a conversão. Caso contrário, passa a responsabilidade para o próximo handler.

```
public class ConversorDolarParaRealHandler extends ConversorHandler {
  private static final double TAXA_CONVERSAO = 4.00;

  @Override
  public boolean aceitar(Cotacao cotacao) {
    return cotacao.getNomeMoedaAtual().equals("Dolar") &&
cotacao.getNomeMoedaDestino().equals("Real");
  }

  @Override
  public Cotacao converter(Cotacao cotacao) {
    if (aceitar(cotacao)) {
      double valorConvertido = cotacao.getValor() * TAXA_CONVERSAO;
      cotacao.setValorConvertido(valorConvertido);
      return cotacao;
    }
    return proximo.converter(cotacao);
  }
}
```

A classe ConversorDolarParaEuroHandler é responsável por converter valores de Dólar para Euro, utilizando uma taxa de conversão fixa. Ela verifica se a conversão pode ser realizada e, em caso afirmativo, realiza a conversão. Caso contrário, passa a responsabilidade para o próximo handler.

```
public class ConversorDolarParaEuroHandler extends ConversorHandler {
  private static final double TAXA_CONVERSAO = 0.80;

  @Override
  public boolean aceitar(Cotacao cotacao) {
```

```
        return cotacao.getNomeMoedaAtual().equals("Dolar") &&
cotacao.getNomeMoedaDestino().equals("Euro");
    }

    @Override
    public Cotacao converter(Cotacao cotacao) {
        if (aceitar(cotacao)) {
            double valorConvertido = cotacao.getValor() * TAXA_CONVERSAO;
            cotacao.setValorConvertido(valorConvertido);
            return cotacao;
        }
        return proximo.converter(cotacao);
    }
}
```

A classe ConversorEuroParaRealHandler é responsável por converter valores de Euro para Real, utilizando uma taxa de conversão fixa. Ela verifica se a conversão pode ser realizada e, em caso afirmativo, realiza a conversão. Caso contrário, passa a responsabilidade para o próximo handler.

```
public class ConversorEuroParaRealHandler extends ConversorHandler {
    private static final double TAXA_CONVERSAO = 5.00;

    @Override
    public boolean aceitar(Cotacao cotacao) {
        return cotacao.getNomeMoedaAtual().equals("Euro") &&
cotacao.getNomeMoedaDestino().equals("Real");
    }

    @Override
    public Cotacao converter(Cotacao cotacao) {
        if (aceitar(cotacao)) {
            double valorConvertido = cotacao.getValor() * TAXA_CONVERSAO;
```

```
            cotacao.setValorConvertido(valorConvertido);
            return cotacao;
        }
        return proximo.converter(cotacao);
    }
}
```

A classe ConversorEuroParaDolarHandler é responsável por converter valores de Euro para Dólar, utilizando uma taxa de conversão fixa. Ela verifica se a conversão pode ser realizada e, em caso afirmativo, realiza a conversão. Caso contrário, passa a responsabilidade para o próximo handler.

```
public class ConversorEuroParaDolarHandler extends ConversorHandler {
    private static final double TAXA_CONVERSAO = 1.25;

    @Override
    public boolean aceitar(Cotacao cotacao) {
        return cotacao.getNomeMoedaAtual().equals("Euro") &&
cotacao.getNomeMoedaDestino().equals("Dolar");
    }

    @Override
    public Cotacao converter(Cotacao cotacao) {
        if (aceitar(cotacao)) {
            double valorConvertido = cotacao.getValor() * TAXA_CONVERSAO;
            cotacao.setValorConvertido(valorConvertido);
            return cotacao;
        }
        return proximo.converter(cotacao);
    }
}
```

A classe ConversorService gerencia a cadeia de handlers e coordena o processo de conversão. Ela inicializa os handlers e define a ordem em que serão chamados para tentar realizar a conversão de moeda.

```
public class ConversorService {
  private ConversorHandler conversorRealParaDolar;
  private ConversorHandler conversorRealParaEuro;
  private ConversorHandler conversorDolarParaReal;
  private ConversorHandler conversorDolarParaEuro;
  private ConversorHandler conversorEuroParaReal;
  private ConversorHandler conversorEuroParaDolar;

  public ConversorService() {
    conversorRealParaDolar = new ConversorRealParaDolarHandler();
    conversorRealParaEuro = new ConversorRealParaEuroHandler();
    conversorDolarParaReal = new ConversorDolarParaRealHandler();
    conversorDolarParaEuro = new ConversorDolarParaEuroHandler();
    conversorEuroParaReal = new ConversorEuroParaRealHandler();
    conversorEuroParaDolar = new ConversorEuroParaDolarHandler();

    conversorRealParaDolar.setProximo(conversorRealParaEuro);
    conversorRealParaEuro.setProximo(conversorDolarParaReal);
    conversorDolarParaReal.setProximo(conversorDolarParaEuro);
    conversorDolarParaEuro.setProximo(conversorEuroParaReal);
    conversorEuroParaReal.setProximo(conversorEuroParaDolar);
  }

  public Cotacao converter(Cotacao cotacao) {
    return conversorRealParaDolar.converter(cotacao);
  }
}
```

Agora, é criado a classe Main para demonstrar o uso. Inicialmente, instanciamos a classe ConversorService e criamos cotações para converter valores entre diferentes moedas. A conversão é realizada utilizando o método converter da classe ConversorService.

```
import java.time.OffsetDateTime;

public class Main {
  public static void main(String[] args) {
    ConversorService conversorService = new ConversorService();

    Cotacao cotacao1 = new Cotacao(100, "Real", OffsetDateTime.now(), "Dolar");
    cotacao1 = conversorService.converter(cotacao1);
    System.out.println("Valor convertido de Real para Dolar: " +
cotacao1.getValorConvertido());

    Cotacao cotacao2 = new Cotacao(100, "Euro", OffsetDateTime.now(), "Real");
    cotacao2 = conversorService.converter(cotacao2);
    System.out.println("Valor convertido de Euro para Real: " +
cotacao2.getValorConvertido());
  }
}
```

Solução do Exercício 69

Para criar uma classe de validação de dados de pessoas para uma aplicação, utilizaremos uma interface chamada IDadosPessoais, que representa os parâmetros do método de validação, incluindo nome, endereço, CPF e data de nascimento. Cada uma dessas verificações será realizada por um tratador específico, que será executado em sequência. Se qualquer um dos tratadores retornar uma mensagem de erro, a validação deve parar e retornar essa mensagem. Caso contrário, a validação deve retornar uma mensagem vazia, indicando que todos os dados são válidos.

A interface IDadoPessoal define os métodos que devem ser implementados para acessar os dados de uma pessoa, como nome, endereço, CPF e data de nascimento.

```
import java.time.LocalDate;

public interface IDadoPessoal {
  public String getNome();
  public String getEndereco();
  public String getCPF();
  public LocalDate getDataNascimento();
}
```

A classe abstrata TratadorDadoPessoal fornece a estrutura para os tratadores de validação de dados. Ela mantém uma lista de tratadores e processa os dados de uma pessoa, verificando cada tratador em sequência. Se qualquer tratador encontrar um erro, a mensagem de erro é retornada.

```
import java.util.ArrayList;

public abstract class TratadorDadoPessoal {
  private ArrayList<TratadorDadoPessoal> tratadores = new ArrayList<>();

  public void adicionarTratador(TratadorDadoPessoal tratador) {
    tratadores.add(tratador);
  }

  public String processar(IDadoPessoal dadosPessoais) {
    for (TratadorDadoPessoal tratador : tratadores) {
      String resultado = tratador.validar(dadosPessoais);
      if (resultado != null) {
        return resultado;
      }
    }
    return "";
```

```
  }

  public abstract String validar(IDadoPessoal dadosPessoais);
}
```

A classe TratadorNome é responsável por validar o nome de uma pessoa. Ela verifica se o nome não é nulo e não está vazio. Se a validação falhar, uma mensagem de erro é retornada.

```
public class TratadorNome extends TratadorDadoPessoal {
  @Override
  public String validar(IDadoPessoal dadosPessoais) {
    if (dadosPessoais.getNome() == null || dadosPessoais.getNome().trim().isEmpty()) {
      return "O nome é inválido";
    }
    return null;
  }
}
```

A classe TratadorEndereco é responsável por validar o endereço de uma pessoa. Ela verifica se o endereço não é nulo e não está vazio. Se a validação falhar, uma mensagem de erro é retornada.

```
public class TratadorEndereco extends TratadorDadoPessoal {
  @Override
  public String validar(IDadoPessoal dadosPessoais) {
    if (dadosPessoais.getEndereco() == null ||
dadosPessoais.getEndereco().trim().isEmpty()) {
      return "O endereço é inválido";
    }
    return null;
  }
}
```

A classe TratadorCPF é responsável por validar o CPF de uma pessoa. Ela verifica se o CPF não é nulo e não está vazio. Se a validação falhar, uma mensagem de erro é retornada.

```
public class TratadorCPF extends TratadorDadoPessoal {
  @Override
  public String validar(IDadoPessoal dadosPessoais) {
    if (dadosPessoais.getCPF() == null || dadosPessoais.getCPF().trim().isEmpty()) {
      return "O CPF é inválido";
    }
    return null;
  }
}
```

A classe TratadorDataNascimento é responsável por validar a data de nascimento de uma pessoa. Ela verifica se a data de nascimento não é nula. Se a validação falhar, uma mensagem de erro é retornada.

```
public class TratadorDataNascimento extends TratadorDadoPessoal {
  @Override
  public String validar(IDadoPessoal dadosPessoais) {
    if (dadosPessoais.getDataNascimento() == null) {
      return "A data de nascimento é inválida";
    }
    return null;
  }
}
```

A classe ValidacaoDadoPessoalService gerencia a cadeia de tratadores e coordena o processo de validação. Ela permite adicionar tratadores à cadeia e processar os dados pessoais, verificando cada tratador em sequência.

```
import java.util.ArrayList;

public class ValidacaoDadoPessoalService {
  private ArrayList<TratadorDadoPessoal> tratadores = new ArrayList<>();

  public void adicionarTratador(TratadorDadoPessoal tratador) {
```

```
        tratadores.add(tratador);
    }

    public String processar(IDadoPessoal dadosPessoais) {
        for (TratadorDadoPessoal tratador : tratadores) {
            String resultado = tratador.validar(dadosPessoais);
            if (resultado != null) {
                return resultado;
            }
        }
        return "";
    }
}
```

A classe DadoPessoal implementa a interface IDadoPessoal e fornece os dados de uma pessoa, incluindo nome, endereço, CPF e data de nascimento.

```
import java.time.LocalDate;

public class DadoPessoal implements IDadoPessoal {
    private String nome;
    private String endereco;
    private String cpf;
    private LocalDate dataNascimento;

    public DadoPessoal(String nome, String endereco, String cpf, LocalDate dataNascimento) {
        this.nome = nome;
        this.endereco = endereco;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }

    @Override
```

```
  public String getNome() {
    return nome;
  }

  @Override
  public String getEndereco() {
    return endereco;
  }

  @Override
  public String getCPF() {
    return cpf;
  }

  @Override
  public LocalDate getDataNascimento() {
    return dataNascimento;
  }
}
```

Agora, é criado a classe Main para demonstrar o uso. Inicialmente, instanciamos a classe ValidacaoDadoPessoalService e criamos instâncias de DadoPessoal com dados válidos e inválidos. A validação é realizada utilizando o método processar da classe ValidacaoDadoPessoalService.

```
import java.time.LocalDate;

public class Main {
  public static void main(String[] args) {
    ValidacaoDadoPessoalService servico = new ValidacaoDadoPessoalService();
    servico.adicionarTratador(new TratadorNome());
    servico.adicionarTratador(new TratadorEndereco());
    servico.adicionarTratador(new TratadorCPF());
```

```
        servico.adicionarTratador(new TratadorDataNascimento());

        DadoPessoal dadosValidos = new DadoPessoal("João da Silva", "Rua das Flores, 123",
"123.456.789-10", LocalDate.of(1980, 1, 1));
        String resultado = servico.processar(dadosValidos);
        System.out.println(resultado);

        DadoPessoal dadosInvalidos = new DadoPessoal("", "Rua das Flores, 123",
"123.456.789-10", null);
        resultado = servico.processar(dadosInvalidos);
        System.out.println(resultado);
    }
}
```

Solução do Exercício 70

Quando se trabalha com diferentes tipos de usuários em uma aplicação, é essencial respeitar o Princípio da Segregação de Interfaces (ISP) para evitar problemas potenciais na arquitetura do sistema. O não cumprimento do ISP pode resultar em interfaces abrangentes e genéricas que forçam os usuários a implementar métodos ou lidar com funcionalidades que não são relevantes para suas funções específicas.

Na situação inicial, há uma única interface, IRegistroDeTransporte, que contém métodos para registrar origem, destino, peso, data e horário de partida e chegada, entre outros. Todos os tipos de usuários que implementam IRegistroDeTransporte são obrigados a codificar todos os métodos, mesmo que algumas funcionalidades não sejam necessárias para suas funções específicas. Isso gera complexidade desnecessária e dificulta a manutenção e uso do código.

Para corrigir essa situação, a interface RegistroDeTransporte será segregada em interfaces menores, cada uma com métodos que possuam relaçao e não violem o Princípio de Responsabilidade Única.

A interface IInformacaoDeTransporte contém métodos para registrar informações sobre origem, destino e peso.

```
public interface IInformacaoDeTransporte {
  void registrarOrigem();
  void registrarDestino();
  void registrarPeso();
}
```

A interface IEdicaoDeTransporte contém métodos para editar informações de data de partida e chegada.

```
public interface IEdicaoDeTransporte {
  void registrarDataPartida(Date data);
  void registrarDataChegada(Date data);
}
```

As classes UsuarioAImpl e UsuarioBImpl agora implementam apenas as interfaces relevantes para suas funções.

```
public class UsuarioAImpl implements IInformacaoDeTransporte {
  @Override
  public void registrarOrigem() {
     System.out.println("Origem registrada.");
  }

  @Override
  public void registrarDestino() {
     System.out.println("Destino registrado.");
  }

  @Override
  public void registrarPeso() {
     System.out.println("Peso registrado.");
  }
}
```

```
public class UsuarioBImpl implements  IEdicaoDeTransporte {
  @Override
  public void registrarDataPartida(Date data) {
     System.out.println("Data partida registrada.");
  }

  @Override
  public void registrarDataChegada(Date data) {
     System.out.println("Data cheagada registrada.");
  }
}
```

Segregando as interfaces dessa maneira, o sistema se torna mais modular, flexível e fácil de manter, com usuários lidando apenas com as funcionalidades necessárias para suas tarefas específicas. Isso também facilita a evolução do aplicativo, pois novas funcionalidades ou tipos de usuários podem ser adicionados sem impactar os existentes.

Solução do Exercício 71

Para resolver este exercício utilizaremos o padrão de projeto Chain of Responsibility mas com uma abordagem diferente da codificação, atribuindo a responsabilidade de iterar sobre as classes específícas de tratadores para a classe ProcessadorService. Portanto nesta resolução não usaremos o método setProximo para criar o encadeamento.

Inicialmente, criamos a classe entidade Conteudo que representa um conteúdo de vídeo na plataforma de streaming.

```
public class Conteudo {
  private String tipo;
  private String titulo;
  private int classificacaoEtaria;

  public Conteudo(String tipo, String titulo, int classificacaoEtaria) {
     this.tipo = tipo;
     this.titulo = titulo;
```

```
        this.classificacaoEtaria = classificacaoEtaria;
    }

    public String getTipo() {
        return tipo;
    }

    public String getTitulo() {
        return titulo;
    }

    public int getClassificacaoEtaria() {
        return classificacaoEtaria;
    }
}
```

Agora, definimos um contrato com a interface ITratador para todos os tratadores de conteúdo. Cada tratador específico deve implementar esta interface e fornecer a lógica para processar o conteúdo de acordo com as regras específicas.

```
public interface ITratador {
    public String processar(Conteudo conteudo);
}
```

A classe TratadorFilmeImpl é uma implementação concreta da interface ITratador que valida se um filme pode ser exibido com base na sua classificação etária. Se a classificação etária do filme for até 12 anos, ele pode ser exibido.

```
public class TratadorFilmeImpl implements ITratador {
    @Override
    public String processar(Conteudo conteudo) {
        if (conteudo.getTipo().equals("Filme") && conteudo.getClassificacaoEtaria() <= 12) {
            return "Conteúdo válido para exibição";
        }
```

```
        return null;
    }
}
```

A classe TratadorSerieImpl é uma implementação concreta da interface ITratador que valida se uma série pode ser exibida com base na sua classificação etária. Se a classificação etária da série for até 16 anos, ela pode ser exibida.

```
public class TratadorSerieImpl implements ITratador {
    @Override
    public String processar(Conteudo conteudo) {
        if (conteudo.getTipo().equals("Série") && conteudo.getClassificacaoEtaria() <= 16) {
            return "Conteúdo válido para exibição";
        }
        return null;
    }
}
```

A classe TratadorDocumentarioImpl é uma implementação concreta da interface ITratador que valida se um documentário pode ser exibido. Documentários podem ser exibidos independentemente da sua classificação etária.

```
public class TratadorDocumentarioImpl implements ITratador {
    @Override
    public String processar(Conteudo conteudo) {
        if (conteudo.getTipo().equals("Documentário")) {
            return "Conteúdo válido para exibição";
        }
        return null;
    }
}
```

A classe ProcessadorService é responsável por gerenciar a lista de tratadores e processar um conteúdo de vídeo, chamando o método processar de cada tratador na ordem até que

um tratador determine que o conteúdo pode ser exibido ou todos os tratadores sejam verificados.

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

public class ProcessadorService {
  private List<ITratador> tratadores;

  public ProcessadorService() {
    tratadores = new ArrayList<>();
    tratadores.add(new TratadorFilmeImpl());
    tratadores.add(new TratadorSerieImpl());
    tratadores.add(new TratadorDocumentarioImpl());
  }

  public String processar(Conteudo conteudo) {
    for (ITratador tratador : tratadores) {
      String resultado = tratador.processar(conteudo);
      if (resultado != null) {
        return resultado;
      }
    }
    return "Conteúdo inválido para exibição";
  }
}
```

A classe Main demonstra o uso da classe ProcessadorService. Ela cria diferentes tipos de conteúdo e usa o processador de serviço para verificar se o conteúdo pode ser exibido.

```
public class Main {
  public static void main(String[] args) {
    ProcessadorService processador = new ProcessadorService();
```

```
        Conteudo conteudo1 = new Conteudo("Série", "Breaking Bad", 18);
        String resultado1 = processador.processar(conteudo1);
        System.out.println(resultado1);

        Conteudo conteudo2 = new Conteudo("Documentário", "Natureza Selvagem", 12);
        String resultado2 = processador.processar(conteudo2);
        System.out.println(resultado2);
    }
}
```

A execução deste código resultará na seguinte saída no console:

Conteúdo inválido para exibição

Conteúdo válido para exibição

Solução do Exercício 72

Para resolver a violação mencionada, é necessário garantir que a subclasse não altere a semântica fundamental da operação de remoção definida pela classe base. Em vez de redefinir completamente a estratégia de remoção, a GerenciadorListaPrioritaria pode incorporar um sistema de priorização que coexista com o mecanismo FIFO, mantendo a ordem de entrada.

Código Corrigido

Classe Abstrata GerenciadorListaEspera

```
import java.util.LinkedList;
import java.util.Queue;

public abstract class GerenciadorListaEspera {
    protected Queue<IUsuario> filaEspera = new LinkedList<>();

    public void adicionarUsuario(IUsuario usuario) {
        filaEspera.add(usuario);
    }
```

```
    public IUsuario removerUsuario() {
        return filaEspera.poll();
    }

    public abstract void processar();
}
```

Subclasse GerenciadorListaPrioritaria

```
import java.util.Comparator;
import java.util.PriorityQueue;

public class GerenciadorListaPrioritaria extends GerenciadorListaEspera {
    private PriorityQueue<IUsuario> filaPrioritaria;
    private boolean usarPrioridade;

    public GerenciadorListaPrioritaria(boolean usarPrioridade) {
        this.usarPrioridade = usarPrioridade;
        this.filaPrioritaria = new
PriorityQueue<>(Comparator.comparing(IUsuario::isVip).reversed());
    }

    @Override
    public void adicionarUsuario(IUsuario usuario) {
        if (usarPrioridade) {
            filaPrioritaria.add(usuario);
        } else {
            super.adicionarUsuario(usuario);
        }
    }

    @Override
    public IUsuario removerUsuario() {
```

```
        if (usarPrioridade) {
            return filaPrioritaria.poll();
        } else {
            return super.removerUsuario();
        }
    }

    @Override
    public void processar() {
        while (usarPrioridade ? !filaPrioritaria.isEmpty() : !filaEspera.isEmpty()) {
            IUsuario usuario = removerUsuario();
            notificarUsuarioProcessado(usuario);
        }
    }

    private void notificarUsuarioProcessado(IUsuario usuario) {
        System.out.println("Usuário " + usuario.getNome() + " foi processado.");
    }
}
```

Este código adere ao Princípio de Substituição de Liskov, mantendo o comportamento essencial da classe base, mas permitindo flexibilidade para implementar funcionalidades adicionais, como a priorização de usuários, de forma controlada. A configuração usarPrioridade permite alternar entre comportamento puramente FIFO e priorização de usuários sem comprometer a integridade do sistema.

Classe Principal Main

```
public class Main {
    public static void main(String[] args) {
        GerenciadorListaEspera gerenciadorPrioritario = new GerenciadorListaPrioritaria(true);

        // Adicionando usuários à fila prioritária
        gerenciadorPrioritario.adicionarUsuario(new Usuario("Alice", false));
```

```
        gerenciadorPrioritario.adicionarUsuario(new Usuario("Bob", true));  // Usuário VIP
        gerenciadorPrioritario.adicionarUsuario(new Usuario("Charlie", false));

        // Criando o serviço de gerenciamento de filas de espera com a instância prioritária
        GerenciadorFilaEsperaService gerenciadorService =
                new GerenciadorFilaEsperaService(gerenciadorPrioritario);

        // Processando usuários na fila prioritária
        gerenciadorService.processarUsuarios();

        // Demonstrando com outro tipo de gerenciador se necessário
        GerenciadorListaEspera outroGerenciador = new GerenciadorListaPadrao();  // Outra
subclasse de GerenciadorListaEspera
        gerenciadorService.setGerenciador(outroGerenciador);

        // Adicionando usuários ao novo gerenciador e processando
        outroGerenciador.adicionarUsuario(new Usuario("David", false));
        gerenciadorService.processarUsuarios();
    }
}
```

Classe GerenciadorFilaEsperaService

```
public class GerenciadorFilaEsperaService {
    private GerenciadorListaEspera gerenciador;

    public GerenciadorFilaEsperaService(GerenciadorListaEspera gerenciador) {
        this.gerenciador = gerenciador;
    }

    public void setGerenciador(GerenciadorListaEspera gerenciador) {
        this.gerenciador = gerenciador;
    }
```

```
    public void processarUsuarios() {
        gerenciador.processar();
    }
}
```

Neste exemplo, a classe Main inicializa diferentes tipos de gerenciadores de lista de espera e os utiliza através do serviço GerenciadorFilaEsperaService. Isso ilustra a flexibilidade do design e como diferentes comportamentos de gerenciamento podem ser integrados e manipulados usando uma única interface de serviço.

Solução do Exercício 73

Para solucionar esta questão, utilizaremos o padrão de projeto Decorator para permitir a adição dinâmica dessas funcionalidades aos arquivos. Cada arquivo pode ser decorado com funcionalidades adicionais sem modificar a estrutura original da classe.

Inicialmente, criamos a classe Arquivo representa um arquivo de sistema com os atributos nome e tamanho. Esta classe serve como base para os decoradores que serão implementados para adicionar funcionalidades extras aos arquivos.

```
public class Arquivo {
  private String nome;
  private int tamanho;

  public Arquivo(String nome, int tamanho) {
    this.nome = nome;
    this.tamanho = tamanho;
  }

  public String getNome() {
    return nome;
  }

  public int getTamanho() {
```

```
    return tamanho;
  }
}
```

Para adicionar funcionalidades como criptografia e compressão, utilizamos decoradores que estendem a classe ArquivoDecorator. Esta é uma classe abstrata que estende Arquivo e armazena uma referência ao objeto Arquivo decorado.

```
public abstract class ArquivoDecorator extends Arquivo {
  protected Arquivo arquivoDecorado;

  public ArquivoDecorator(Arquivo arquivoDecorado, String nome, int tamanho) {
    super(nome, tamanho);
    this.arquivoDecorado = arquivoDecorado;
  }
}
```

O decorador ArquivoComCriptografia adiciona criptografia ao arquivo, com os atributos adicionais senha e algoritmo, que representam a senha utilizada para criptografar o arquivo e o algoritmo de criptografia, respectivamente.

```
public class ArquivoComCriptografia extends ArquivoDecorator {
  private String senha;
  private String algoritmo;

  public ArquivoComCriptografia(Arquivo arquivoDecorado, String senha, String algoritmo) {
    super(arquivoDecorado, arquivoDecorado.getNome(), arquivoDecorado.getTamanho());
    this.senha = senha;
    this.algoritmo = algoritmo;
  }

  public String getSenha() {
    return senha;
  }
```

```
  public String getAlgoritmo() {
    return algoritmo;
  }
}
```

O decorador ArquivoComCompressao adiciona compressão ao arquivo, com o atributo adicional taxaDeCompressao, que representa a taxa de compressão aplicada ao arquivo.

```
public class ArquivoComCompressao extends ArquivoDecorator {
  private int taxaDeCompressao;

  public ArquivoComCompressao(Arquivo arquivoDecorado, int taxaDeCompressao) {
    super(arquivoDecorado, arquivoDecorado.getNome(), arquivoDecorado.getTamanho());
    this.taxaDeCompressao = taxaDeCompressao;
  }

  public int getTaxaDeCompressao() {
    return taxaDeCompressao;
  }
}
```

A classe GerenciadorDeArquivo é responsável por gerenciar uma lista de arquivos. Ela possui um método adicionarArquivo que recebe um objeto Arquivo e o adiciona à lista de arquivos. Se o objeto passado for nulo, o método lança uma exceção ArquivoInvalidoException.

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

public class GerenciadorDeArquivo {
  private List<Arquivo> arquivos;

  public GerenciadorDeArquivo() {
    arquivos = new ArrayList<>();
```

```
    }

    public void adicionarArquivo(Arquivo arquivo) throws ArquivoInvalidoException {
        if (arquivo == null) {
            throw new ArquivoInvalidoException("Arquivo não pode ser nulo.");
        }
        arquivos.add(arquivo);
    }
}
```

A exceção ArquivoInvalidoException é definida para tratar casos em que um arquivo inválido (nulo) seja adicionado ao gerenciador.

```
public class ArquivoInvalidoException extends Exception {
    public ArquivoInvalidoException(String mensagem) {
        super(mensagem);
    }
}
```

Por fim, a classe Main demonstra o uso das classes e decoradores implementados. No método main, são criados diferentes tipos de arquivos, aplicadas as decorações de criptografia e compressão, e os arquivos são adicionados ao gerenciador de arquivos. A exceção é tratada para garantir que arquivos inválidos não sejam adicionados.

```
public class Main {
    public static void main(String[] args) {
        GerenciadorDeArquivo gerenciador = new GerenciadorDeArquivo();

        try {
            Arquivo foto = new Arquivo("foto.jpg", 1000);
            gerenciador.adicionarArquivo(foto);

            Arquivo documento = new ArquivoComCriptografia(
                    new Arquivo("documento.txt", 2000), "123456", "AES");
```

```
        gerenciador.adicionarArquivo(documento);

        Arquivo musicas = new ArquivoComCompressao(
                new ArquivoComCriptografia(
                        new Arquivo("musicas.mp3", 5000), "abcdef", "DES"), 50);
        gerenciador.adicionarArquivo(musicas);

    } catch (ArquivoInvalidoException e) {
        System.out.println(e.getMessage());
    }

  }
}
```

Solução do Exercício 74

Para resolver este exercício, utilizaremos o padrão de projeto Decorator do GoF, que permite a adição dinâmica de comportamentos adicionais a um objeto, mantendo a mesma interface.

Inicialmente, criamos a interface de autenticação que define o método autenticar, que deve ser implementado pelas classes concretas de autenticação.

```
public interface IAutenticacao {
  boolean autenticar(String login, String senha) throws AutenticacaoException;
}
```

A seguir, implementamos a classe de exceção que será utilizada para tratar erros de autenticação.

```
public class AutenticacaoException extends Exception {
  public AutenticacaoException(String message) {
    super(message);
  }
}
```

A classe de autenticação simples, que verifica apenas o login e a senha, é uma implementação concreta da interface de autenticação.

```
public class AutenticacaoSimples implements IAutenticacao {
  @Override
  public boolean autenticar(String login, String senha) throws AutenticacaoException {
    // Lógica para autenticação com login e senha
    return true;
  }
}
```

Para adicionar funcionalidades adicionais, criamos uma classe abstrata que servirá como base para os decoradores de autenticação.

```
public abstract class AutenticacaoDecorator implements IAutenticacao {
  protected IAutenticacao autenticacao;

  public AutenticacaoDecorator(IAutenticacao autenticacao) {
    this.autenticacao = autenticacao;
  }
}
```

Um dos decoradores possíveis é o de autenticação por código enviado por SMS. Este decorador verifica se o código de autenticação recebido é válido.

```
public class AutenticacaoSMS extends AutenticacaoDecorator {
  public AutenticacaoSMS(IAutenticacao autenticacao) {
    super(autenticacao);
  }

  @Override
  public boolean autenticar(String login, String senha) throws AutenticacaoException {
    String codigo = obterCodigoSMS();
    System.out.println("Verificando código de autenticação SMS: " + codigo);
    if (codigo == null) {
```

```
            throw new AutenticacaoException("Código de autenticação SMS inválido ou expirado");
        }
        return autenticacao.autenticar(login, senha);
    }

    private String obterCodigoSMS() {
        // Lógica para obter o código enviado por SMS
        return "123456";
    }
}
```

Outro decorador é o de autenticação por token gerado por aplicativo. Este decorador verifica se o token gerado pelo aplicativo é válido.

```
public class AutenticacaoToken extends AutenticacaoDecorator {
    public AutenticacaoToken(IAutenticacao autenticacao) {
        super(autenticacao);
    }

    @Override
    public boolean autenticar(String login, String senha) throws AutenticacaoException {
        String token = obterToken();
        System.out.println("Verificando token de autenticação: " + token);
        if (token == null) {
            throw new AutenticacaoException("Token de autenticação inválido ou expirado");
        }
        return autenticacao.autenticar(login, senha);
    }

    private String obterToken() {
        // Lógica para obter o token gerado pelo aplicativo
        return "abcdef";
    }
```

```
}
```

O decorador de autenticação por reconhecimento de voz verifica se a frase falada pelo usuário é válida.

```
public class AutenticacaoVoz extends AutenticacaoDecorator {
  public AutenticacaoVoz(IAutenticacao autenticacao) {
    super(autenticacao);
  }

  @Override
  public boolean autenticar(String login, String senha) throws AutenticacaoException {
    String frase = obterFrase();
    System.out.println("Verificando frase de autenticação por voz: " + frase);
    if (frase == null) {
      throw new AutenticacaoException("Frase de autenticação por voz inválida ou não
reconhecida");
    }
    return autenticacao.autenticar(login, senha);
  }

  private String obterFrase() {
    // Lógica para obter a frase falada pelo usuário
    return "minha frase de autenticação";
  }
}
```

O decorador de autenticação por reconhecimento facial verifica se a imagem do rosto do usuário é válida.

```
public class AutenticacaoFacial extends AutenticacaoDecorator {
  public AutenticacaoFacial(IAutenticacao autenticacao) {
    super(autenticacao);
  }
```

```
  @Override
  public boolean autenticar(String login, String senha) throws AutenticacaoException {
    byte[] imagem = obterImagem();
    System.out.println("Verificando imagem de autenticação por reconhecimento facial");
    if (imagem == null) {
      throw new AutenticacaoException("Imagem de autenticação por reconhecimento
facial inválida ou não reconhecida");
    }
    return autenticacao.autenticar(login, senha);
  }

  private byte[] obterImagem() {
    // Lógica para obter a imagem do rosto do usuário
    return new byte[0];
  }
}
```

Por fim, é criado classe Main que demonstra o uso dos decoradores de autenticação, criando uma cadeia de autenticação que inclui autenticação por SMS e token gerado por aplicativo.

```
public class Main {
  public static void main(String[] args) {
    // Cria a cadeia de decoradores de autenticação
    IAutenticacao autenticacao = new AutenticacaoSMS(
            new AutenticacaoToken(new AutenticacaoSimples()));

    try {
      String login = "fulano";
      String senha = "123456";
      if (autenticacao.autenticar(login, senha)) {
        System.out.println("Autenticação realizada com sucesso");
      } else {
```

```
            System.out.println("Autenticação falhou");
        }
    } catch (AutenticacaoException e) {
        System.out.println("Erro na autenticação: " + e.getMessage());
    }
  }
}
```

Solução do Exercício 75

Para resolver este exercício, utilizaremos o padrão de projeto Decorator para adicionar funcionalidades adicionais aos pagamentos, permitindo a aplicação de descontos e acréscimos.

Inicialmente, criamos a classe abstrata PagamentoDecorator, que estende a classe Pagamento e inclui um campo para armazenar a referência ao pagamento original. A classe PagamentoDecorator possui um construtor que inicializa este campo e sobrescreve o método getValor para ser implementado pelas subclasses concretas.

```
public abstract class PagamentoDecorator extends Pagamento {
  protected Pagamento pagamento;

  public PagamentoDecorator(Pagamento pagamento) {
    super(pagamento.getValor());
    this.pagamento = pagamento;
  }

  @Override
  public abstract BigDecimal getValor();
}
```

A classe Desconto é uma subclasse concreta de PagamentoDecorator que aplica um desconto ao valor do pagamento original. O construtor desta classe inicializa o valor do desconto e o método getValor subtrai o valor do desconto do pagamento original.

```
public class Desconto extends PagamentoDecorator {
  private BigDecimal desconto;

  public Desconto(Pagamento pagamento, BigDecimal desconto) {
    super(pagamento);
    this.desconto = desconto;
  }

  @Override
  public BigDecimal getValor() {
    return pagamento.getValor().subtract(desconto);
  }
}
```

A classe Acrescimo é outra subclasse concreta de PagamentoDecorator, que adiciona um acréscimo ao valor do pagamento original. O construtor inicializa o valor do acréscimo e o método getValor adiciona este valor ao pagamento original.

```
public class Acrescimo extends PagamentoDecorator {
  private BigDecimal acrescimo;

  public Acrescimo(Pagamento pagamento, BigDecimal acrescimo) {
    super(pagamento);
    this.acrescimo = acrescimo;
  }

  @Override
  public BigDecimal getValor() {
    return pagamento.getValor().add(acrescimo);
  }
}
```

Por fim, a classe Main demonstra o uso dos decoradores aplicando um desconto e um acréscimo ao valor de um pagamento com cartão de crédito, imprimindo o valor final do pagamento.

```
public class Main {
  public static void main(String[] args) {
    Pagamento pagamento = new CartaoCredito(new BigDecimal(100));
    pagamento = new Desconto(pagamento, new BigDecimal(10));
    pagamento = new Acrescimo(pagamento, new BigDecimal(5));
    System.out.println(pagamento.getValor());
  }
}
```

Solução do Exercício 76

Para resolver este exercício, utilizaremos o padrão de projeto Decorator para gerenciar as permissões de um usuário. Inicialmente, criamos a interface IPermissao que define os métodos podeLer, podeEscrever e podeExcluir.

```
public interface IPermissao {
  public boolean podeLer();
  public boolean podeEscrever();
  public boolean podeExcluir();
}
```

A classe Leitor é uma implementação concreta da interface IPermissao. Ela representa um usuário com permissões de leitura, mas sem permissões de escrita ou exclusão.

```
public class Leitor implements IPermissao {
  @Override
  public boolean podeLer() {
    return true;
  }

  @Override
```

```
  public boolean podeEscrever() {
    return false;
  }

  @Override
  public boolean podeExcluir() {
    return false;
  }
}
```

A classe abstrata PermissaoDecorator também implementa a interface Permissao. Esta classe inclui um campo para armazenar a referência a uma permissão existente, delegando as operações de leitura, escrita e exclusão a essa permissão.

```
public abstract class PermissaoDecorator implements IPermissao {
  protected IPermissao permissao;

  public PermissaoDecorator(IPermissao permissao) {
    if (permissao == null) {
      throw new IllegalArgumentException("Permissão não pode ser nula");
    }
    this.permissao = permissao;
  }

  @Override
  public boolean podeLer() {
    return permissao.podeLer();
  }

  @Override
  public boolean podeEscrever() {
    return permissao.podeEscrever();
  }
```

```
  @Override
  public boolean podeExcluir() {
    return permissao.podeExcluir();
  }
}
```

A classe Editor é uma subclasse concreta de PermissaoDecorator. Ela adiciona a permissão de escrita ao usuário.

```
public class Editor extends PermissaoDecorator {
  public Editor(IPermissao permissao) {
    super(permissao);
  }

  @Override
  public boolean podeEscrever() {
    return true;
  }
}
```

A classe Administrador é outra subclasse concreta de PermissaoDecorator. Além da permissão de escrita, ela também adiciona a permissão de exclusão ao usuário.

```
public class Administrador extends PermissaoDecorator {
  public Administrador(IPermissao permissao) {
    super(permissao);
  }

  @Override
  public boolean podeEscrever() {
    return true;
  }
```

```
  @Override
  public boolean podeExcluir() {
    return true;
  }
}
```

Por fim, a classe Principal demonstra o uso dos decoradores aplicando as permissões de leitor, editor e administrador a um usuário e imprimindo as permissões finais.

```
public class Principal {
  public static void main(String[] args) {
    IPermissao usuario = new Leitor();
    usuario = new Editor(usuario);
    usuario = new Administrador(usuario);

    System.out.println("Pode ler: " + usuario.podeLer()); //true
    System.out.println("Pode escrever: " + usuario.podeEscrever()); //true
    System.out.println("Pode deletar: " + usuario.podeExcluir()); //true

  }
}
```

Solução do Exercício 77

Para resolver este exercício, utilizaremos o padrão de projeto Facade para gerenciar as operações bancárias.

Partindo da premissa que as classes Banco, ContaBancaria e Cliente já foram codificadas, criamos a classe FacadeBancario que encapsula as funcionalidades das outras classes e oferece métodos mais simples para realizar operações bancárias. Esta classe proporciona uma interface unificada para as operações de criar conta, realizar depósito, realizar saque e consultar saldo.

```
public class FacadeBancario {
  private Banco banco;
```

```
  public FacadeBancario() {
    banco = new Banco();
  }

  public void criarConta(int numeroConta, String nomeTitular, String cpfTitular) {
    banco.criarConta(numeroConta, nomeTitular, cpfTitular);
  }

  public void realizarDeposito(int numeroConta, double valor) {
    banco.realizarDeposito(numeroConta, valor);
  }

  public void realizarSaque(int numeroConta, double valor) {
    banco.realizarSaque(numeroConta, valor);
  }

  public double consultarSaldo(int numeroConta) {
    return banco.consultarSaldo(numeroConta);
  }
}
```

Em seguida criamos a classe Principal que demonstra o uso da classe FacadeBancario. Ela cria uma conta bancária, realiza um depósito, um saque e consulta o saldo da conta.

```
public class Principal {
  public static void main(String[] args) {
    FacadeBancario facade = new FacadeBancario();

    facade.criarConta(12345, "João da Silva", "123.456.789-10");
    facade.realizarDeposito(12345, 100);
    facade.realizarSaque(12345, 50);

    double saldo = facade.consultarSaldo(12345);
```

```
        System.out.println("Saldo da conta 12345: " + saldo);
    }
}
```

Solução do Exercício 78

Para refatorar o código de maneira a seguir o princípio da responsabilidade única, você pode criar uma nova classe chamada EnviadorDeEmail, que será responsável por enviar e-mails. A classe ContaBancaria ficará responsável apenas por gerenciar a conta bancária, sem a responsabilidade de enviar e-mails.

```
public class ContaBancaria {
    private double saldo;

    public void depositar(double valor) {
        // código para depositar valor na conta bancária
        saldo += valor;
    }

    public void sacar(double valor) {
        // código para sacar valor da conta bancária
        if (valor <= saldo) {
            saldo -= valor;
        } else {
            throw new IllegalArgumentException("Saldo insuficiente");
        }
    }

    public double verSaldo() {
        // código para retornar o saldo atual da conta bancária
        return saldo;
    }
}
```

```
public class EnviadorDeEmail {
  public void enviarEmail(String mensagem) {
    // código para enviar e-mail com a mensagem especificada
    System.out.println("Enviando e-mail: " + mensagem);
  }
}
```

Solução do Exercício 79

A solução para este exercício utiliza o padrão de projeto Command para encapsular cada comando em uma classe separada. Cada comando implementa a interface IComando, que possui um método executar() para executar o comando correspondente.

Inicialmente criamos a classe SistemaAutomatico que representa o sistema de automação residencial, contendo métodos para controlar as luzes e o ar-condicionado. Ela possui métodos para ligar, desligar e alternar o estado das luzes e do ar-condicionado.

```
public class SistemaAutomatico {
  private boolean luzesLigadas;
  private boolean arCondicionadoLigado;

  public void ligarLuzes() {
    luzesLigadas = true;
    System.out.println("Luzes ligadas");
  }

  public void desligarLuzes() {
    luzesLigadas = false;
    System.out.println("Luzes desligadas");
  }

  public void alternarLuzes() {
    luzesLigadas = !luzesLigadas;
    System.out.println("Estado das luzes alternado");
```

```
  }

  public void ligarArCondicionado() {
    arCondicionadoLigado = true;
    System.out.println("Ar-condicionado ligado");
  }

  public void desligarArCondicionado() {
    arCondicionadoLigado = false;
    System.out.println("Ar-condicionado desligado");
  }

  public void alternarArCondicionado() {
    arCondicionadoLigado = !arCondicionadoLigado;
    System.out.println("Estado do ar-condicionado alternado");
  }
}
```

Em seguida, é criado a interface IComando define o método executar(), que é implementado por todos os comandos concretos. Esta interface permite que todos os comandos tenham um método comum para execução.

```
public interface IComando {
  public void executar();
}
```

A classe ComandoLigarLuz implementa a interface IComando e encapsula o comando para ligar as luzes.

```
public class ComandoLigarLuz implements IComando {
  private SistemaAutomatico sistema;

  public ComandoLigarLuz(SistemaAutomatico sistema) {
    this.sistema = sistema;
```

```
  }

  @Override
  public void executar() {
      sistema.ligarLuzes();
  }
}
```

A classe ComandoDesligarLuzes implementa a interface IComando e encapsula o comando para desligar as luzes.

```
public class ComandoDesligarLuz implements IComando {
  private SistemaAutomatico sistema;

  public ComandoDesligarLuz(SistemaAutomatico sistema) {
      this.sistema = sistema;
  }

  @Override
  public void executar() {
      sistema.desligarLuzes();
  }
}
```

A classe ComandoAlternarLuzes implementa a interface IComando e encapsula o comando para alternar o estado das luzes.

```
public class ComandoAlternarLuz implements IComando {
  private SistemaAutomatico sistema;

  public ComandoAlternarLuz(SistemaAutomatico sistema) {
      this.sistema = sistema;
  }
```

```
  @Override
  public void executar() {
      sistema.alternarLuzes();
  }
}
```

A classe ComandoLigarArCondicionado implementa a interface IComando e encapsula o comando para ligar o ar-condicionado.

```
public class ComandoLigarArCondicionado implements IComando {
  private SistemaAutomatico sistema;

  public ComandoLigarArCondicionado(SistemaAutomatico sistema) {
      this.sistema = sistema;
  }

  @Override
  public void executar() {
      sistema.ligarArCondicionado();
  }
}
```

A classe ComandoDesligarArCondicionado implementa a interface IComando e encapsula o comando para desligar o ar-condicionado.

```
public class ComandoDesligarArCondicionado implements IComando {
  private SistemaAutomatico sistema;

  public ComandoDesligarArCondicionado(SistemaAutomatico sistema) {
      this.sistema = sistema;
  }

  @Override
  public void executar() {
```

```
        sistema.desligarArCondicionado();
    }
}
```

A classe ComandoAlternarArCondicionado implementa a interface IComando e encapsula o comando para alternar o estado do ar-condicionado.

```
public class ComandoAlternarArCondicionado implements IComando {
    private SistemaAutomatico sistema;

    public ComandoAlternarArCondicionado(SistemaAutomatico sistema) {
        this.sistema = sistema;
    }

    @Override
    public void executar() {
        sistema.alternarArCondicionado();
    }
}
```

A classe Principal, demonstra o uso do sistema de automação residencial. Ela cria instâncias dos comandos e os executa.

```
public class Principal {
    public static void main(String[] args) {
        SistemaAutomatico sistema = new SistemaAutomatico();
        IComando ligarLuzes = new ComandoLigarLuz(sistema);
        IComando desligarLuzes = new ComandoDesligarLuz(sistema);
        IComando alternarLuzes = new ComandoAlternarLuz(sistema);
        IComando ligarArCondicionado = new ComandoLigarArCondicionado(sistema);
        IComando desligarArCondicionado = new ComandoDesligarArCondicionado(sistema);
        IComando alternarArCondicionado = new ComandoAlternarArCondicionado(sistema);

        // Executar os comandos
```

```
        ligarLuzes.executar();
        desligarLuzes.executar();
        alternarLuzes.executar();
        ligarArCondicionado.executar();
        desligarArCondicionado.executar();
        alternarArCondicionado.executar();
    }
}
```

Solução do Exercício 80

Para resolver este exercício utiliza-se o padrão de projeto Template Method para definir a estrutura básica do algoritmo para adicionar, remover e listar tarefas. Os detalhes específicos de exibição das tarefas são fornecidos pelas subclasses.

Primeiramente, é criada a classe Tarefa que representa uma tarefa individual. Ela contém informações como o nome da tarefa, a data de criação e a prioridade.

```
import java.util.Date;

public class Tarefa {
    private String nome;
    private Date dataCriacao;
    private int prioridade;

    public Tarefa(String nome, Date dataCriacao, int prioridade) {
        this.nome = nome;
        this.dataCriacao = dataCriacao;
        this.prioridade = prioridade;
    }

    public String getNome() {
        return nome;
    }
```

```
  public Date getDataCriacao() {
    return dataCriacao;
  }

  public int getPrioridade() {
    return prioridade;
  }
}
```

A classe abstrata GerenciadorTarefas define a estrutura básica para gerenciar as tarefas. Ela inclui métodos para adicionar, remover e listar tarefas. O método listarTarefas itera sobre todas as tarefas e chama o método abstrato exibirTarefa para cada uma delas.

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

public abstract class GerenciadorTarefa {
  protected List<Tarefa> tarefas = new ArrayList<>();

  public void adicionarTarefa(Tarefa tarefa) {
    tarefas.add(tarefa);
  }

  public void removerTarefa(Tarefa tarefa) {
    tarefas.remove(tarefa);
  }

  public void listarTarefas() {
    for (Tarefa tarefa : tarefas) {
      exibirTarefa(tarefa);
    }
  }
```

```
  protected abstract void exibirTarefa(Tarefa tarefa);
}
```

A classe GerenciadorTarefasPorDataCriacao estende GerenciadorTarefas e implementa o método exibirTarefa para exibir as tarefas por data de criação.

```
public class GerenciadorTarefaPorDataCriacao extends GerenciadorTarefa {
  @Override
  protected void exibirTarefa(Tarefa tarefa) {
    System.out.println("Tarefa: " + tarefa.getNome());
    System.out.println("Data de criação: " + tarefa.getDataCriacao());
    System.out.println("Prioridade: " + tarefa.getPrioridade());
    System.out.println();
  }
}
```

A classe GerenciadorTarefasPorPrioridade estende GerenciadorTarefas e implementa o método exibirTarefa para exibir as tarefas por prioridade.

```
public class GerenciadorTarefaPorPrioridade extends GerenciadorTarefa {
  @Override
  protected void exibirTarefa(Tarefa tarefa) {
    System.out.println("Tarefa: " + tarefa.getNome());
    System.out.println("Prioridade: " + tarefa.getPrioridade());
    System.out.println("Data de criação: " + tarefa.getDataCriacao());
    System.out.println();
  }
}
```

A classe principal Principal demonstra o uso do sistema de gerenciamento de tarefas. Ela cria algumas tarefas, adiciona-as aos gerenciadores e exibe as tarefas de acordo com as diferentes visualizações.

```
import java.util.Date;
```

```
public class Principal {
  public static void main(String[] args) {
    GerenciadorTarefa gerenciadorPorData = new GerenciadorTarefaPorDataCriacao();
    GerenciadorTarefa gerenciadorPorPrioridade = new GerenciadorTarefaPorPrioridade();

    Tarefa tarefa1 = new Tarefa("Estudar para a prova", new Date(), 1);
    Tarefa tarefa2 = new Tarefa("Fazer compras", new Date(), 2);
    Tarefa tarefa3 = new Tarefa("Lavar o carro", new Date(), 3);

    gerenciadorPorData.adicionarTarefa(tarefa1);
    gerenciadorPorData.adicionarTarefa(tarefa2);
    gerenciadorPorData.adicionarTarefa(tarefa3);

    gerenciadorPorPrioridade.adicionarTarefa(tarefa1);
    gerenciadorPorPrioridade.adicionarTarefa(tarefa2);
    gerenciadorPorPrioridade.adicionarTarefa(tarefa3);

    System.out.println("Listando tarefas por data de criação:");
    gerenciadorPorData.listarTarefas();

    gerenciadorPorData.removerTarefa(tarefa2);

    System.out.println("Listando tarefas por data de criação (após remoção):");
    gerenciadorPorData.listarTarefas();

    System.out.println("Listando tarefas por prioridade:");
    gerenciadorPorPrioridade.listarTarefas();
  }
}
```

Ao executar a o método main, espera-se uma semelhante a seguinte

Listando tarefas por data de criação:
Tarefa: Estudar para a prova
Data de criação: [data atual]
Prioridade: 1

Tarefa: Fazer compras
Data de criação: [data atual]
Prioridade: 2

Tarefa: Lavar o carro
Data de criação: [data atual]
Prioridade: 3

Listando tarefas por data de criação (após remoção):
Tarefa: Estudar para a prova
Data de criação: [data atual]
Prioridade: 1

Tarefa: Lavar o carro
Data de criação: [data atual]
Prioridade: 3

Listando tarefas por prioridade:
Tarefa: Estudar para a prova
Prioridade: 1
Data de criação: [data atual]

Tarefa: Fazer compras
Prioridade: 2
Data de criação: [data atual]

Tarefa: Lavar o carro

Prioridade: 3

Data de criação: [data atual]

Solução do Exercício 81

A solução para este exercício utiliza o padrão de projeto Command para permitir que os comandos sejam executados de maneira desacoplada da fila de atendimento. A seguir, apresentamos a implementação das classes necessárias para resolver o exercício, em Java:

A classe FilaAtendimento representa a fila de atendimento de uma loja. Esta classe possui métodos para adicionar um cliente à fila, remover o próximo cliente da fila e consultar a quantidade de clientes na fila. A lista de clientes é representada por um ArrayList.

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

public class FilaAtendimento {
  private List<String> clientes;

  public FilaAtendimento() {
    this.clientes = new ArrayList<>();
  }

  public void adicionarCliente(String nome) {
    clientes.add(nome);
    System.out.println("Cliente " + nome + " adicionado à fila");
  }

  public void removerProximoCliente() {
    if (clientes.isEmpty()) {
      System.out.println("A fila está vazia");
    } else {
      String clienteRemovido = clientes.remove(0);
      System.out.println("Cliente " + clienteRemovido + " removido da fila");
```

```
    }
  }

  public int consultarQuantidadeClientes() {
    return clientes.size();
  }
}
```

A interface IComando define um contrato para a execução de comandos. Cada comando deve implementar o método executar.

```
public interface IComando {
  public void executar();
}
```

A classe ComandoAdicionarCliente implementa a interface IComando e representa o comando para adicionar um cliente à fila. Ela armazena a fila de atendimento e o nome do cliente a ser adicionado.

```
public class ComandoAdicionarCliente implements IComando {
  private FilaAtendimento fila;
  private String nomeCliente;

  public ComandoAdicionarCliente(FilaAtendimento fila, String nomeCliente) {
    this.fila = fila;
    this.nomeCliente = nomeCliente;
  }

  @Override
  public void executar() {
    fila.adicionarCliente(nomeCliente);
  }
}
```

A classe ComandoRemoverProximoCliente implementa a interface IComando e representa o comando para remover o próximo cliente da fila. Ela armazena a fila de atendimento.

```
public class ComandoRemoverProximoCliente implements IComando {
  private FilaAtendimento fila;

  public ComandoRemoverProximoCliente(FilaAtendimento fila) {
    this.fila = fila;
  }

  @Override
  public void executar() {
    fila.removerProximoCliente();
  }
}
```

A classe ComandoConsultarQuantidadeCliente implementa a interface IComando e representa o comando para consultar a quantidade de clientes na fila. Ela armazena a fila de atendimento.

```
public class ComandoConsultarQuantidadeCliente implements IComando {
  private FilaAtendimento fila;

  public ComandoConsultarQuantidadeCliente(FilaAtendimento fila) {
    this.fila = fila;
  }

  @Override
  public void executar() {
    int quantidade = fila.consultarQuantidadeClientes();
    System.out.println("Quantidade de clientes na fila: " + quantidade);
  }
}
```

A classe Main demonstra o uso do sistema de fila de atendimento. Ela cria a fila de atendimento e os comandos, e executa os comandos para adicionar clientes, remover o próximo cliente e consultar a quantidade de clientes na fila.

```
public class Main {
  public static void main(String[] args) {
    FilaAtendimento fila = new FilaAtendimento();

    IComando adicionarCliente1 = new ComandoAdicionarCliente(fila, "João");
    IComando adicionarCliente2 = new ComandoAdicionarCliente(fila, "Maria");
    IComando removerProximoCliente = new ComandoRemoverProximoCliente(fila);
    IComando consultarQuantidadeClientes =
            new ComandoConsultarQuantidadeCliente(fila);

    adicionarCliente1.executar();
    adicionarCliente2.executar();
    consultarQuantidadeClientes.executar();
    removerProximoCliente.executar();
    consultarQuantidadeClientes.executar();
  }
}
```

Saída esperada ao executar o método main:

```
Cliente João adicionado à fila
Cliente Maria adicionado à fila
Quantidade de clientes na fila: 2
Cliente João removido da fila
Quantidade de clientes na fila: 1
```

Solução do Exercício 82

A solução deve utilizar o padrão de projeto Memento com Zelador para permitir que essas operações possam ser desfeitas. A seguir, apresentamos a implementação das classes necessárias para resolver o exercício em Java.

A classe Tarefa representa uma tarefa com os atributos titulo e descricao. Esta classe possui métodos para obter e definir os valores desses atributos.

```
public class Tarefa {
  private String titulo;
  private String descricao;

  public Tarefa(String titulo, String descricao) {
    this.titulo = titulo;
    this.descricao = descricao;
  }

  public String getTitulo() {
    return titulo;
  }

  public void setTitulo(String titulo) {
    this.titulo = titulo;
  }

  public String getDescricao() {
    return descricao;
  }

  public void setDescricao(String descricao) {
    this.descricao = descricao;
  }
}
```

A interface Estado representa o estado de uma tarefa. Ela possui um método para obter o estado da tarefa.

```
public interface IEstado {
  public Tarefa getEstado();
```

```
}
```

A classe Memento implementa a interface Estado e representa um memento de uma tarefa. Ela armazena uma cópia dos atributos de uma tarefa para que o estado possa ser restaurado posteriormente.

```
public class Memento implements IEstado {
  private Tarefa tarefa;

  public Memento(Tarefa tarefa) {
    this.tarefa = new Tarefa(tarefa.getTitulo(), tarefa.getDescricao());
  }

  @Override
  public Tarefa getEstado() {
    return new Tarefa(tarefa.getTitulo(), tarefa.getDescricao());
  }
}
```

A classe Zelador gerencia os mementos. Ela mantém uma lista de mementos e um índice do memento atual. Métodos são fornecidos para adicionar mementos e para obter os mementos anterior e posterior.

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

public class Zelador {
  private List<IEstado> estados;
  private int indiceAtual;

  public Zelador() {
    estados = new ArrayList<>();
    indiceAtual = -1;
  }
```

```
  public void adicionarMemento(IEstado memento) {
    estados.add(memento);
    indiceAtual++;
  }

  public IEstado getMementoAnterior() {
    if (indiceAtual > 0) {
      indiceAtual--;
      return estados.get(indiceAtual);
    }
    return null;
  }

  public IEstado getMementoPosterior() {
    if (indiceAtual < estados.size() - 1) {
      indiceAtual++;
      return estados.get(indiceAtual);
    }
    return null;
  }
}
```

A classe Main demonstra o uso do sistema de desfazer. Ela cria um zelador, adiciona tarefas e desfaz a adição da última tarefa. Em seguida, imprime a tarefa desfeita.

```
public class Main {
  public static void main(String[] args) {
    Zelador zelador = new Zelador();

    // Adiciona uma tarefa
    Tarefa tarefa1 = new Tarefa("Tarefa 1", "Descrição da tarefa 1");
    zelador.adicionarMemento(new Memento(tarefa1));
```

```
        // Adiciona outra tarefa
        Tarefa tarefa2 = new Tarefa("Tarefa 2", "Descrição da tarefa 2");
        zelador.adicionarMemento(new Memento(tarefa2));

        // Desfaz a adição da última tarefa
        IEstado estadoAnterior = zelador.getMementoAnterior();
        Tarefa tarefaDesfeita = estadoAnterior.getEstado();

        // Imprime a tarefa desfeita
        System.out.println("Título: " + tarefaDesfeita.getTitulo());
        System.out.println("Descrição: " + tarefaDesfeita.getDescricao());
    }
}
```

Saída esperada ao executar o método main.

Título: Tarefa 1

Descrição: Descrição da tarefa 1

Solução do Exercício 83

Para solucionar este exercício, deve ser utilizado o padrão de projeto Memento com Zelador, implementado com o padrão de projeto Singleton. A seguir, apresentamos a implementação das classes necessárias para resolver o exercício em Java.

A classe ContaBancaria representa uma conta bancária com os atributos numero, titular e saldo. Esta classe possui métodos para obter e definir os valores desses atributos.

```
public class ContaBancaria {
    private String numero;
    private String titular;
    private double saldo;

    public ContaBancaria(String numero, String titular, double saldo) {
        this.numero = numero;
```

```
        this.titular = titular;
        this.saldo = saldo;
    }

    public String getNumero() {
        return numero;
    }

    public void setNumero(String numero) {
        this.numero = numero;
    }

    public String getTitular() {
        return titular;
    }

    public void setTitular(String titular) {
        this.titular = titular;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
}
```

A interface Estado será implementada pelos objetos Memento e possui um método getEstado que retorna uma cópia do estado atual da conta bancária.

```
public interface IEstado {
```

```
  ContaBancaria getEstado();
}
```

A classe Memento implementa a interface Estado e armazena o estado da conta bancária. A classe possui um construtor que recebe a conta bancária como parâmetro e inicializa o estado com uma cópia da conta bancária.

```
public class Memento implements IEstado {
  private ContaBancaria contaBancaria;

  public Memento(ContaBancaria contaBancaria) {
                    this.contaBancaria = new ContaBancaria(contaBancaria.getNumero(),
contaBancaria.getTitular(), contaBancaria.getSaldo());
  }

  @Override
  public ContaBancaria getEstado() {
          return new ContaBancaria(contaBancaria.getNumero(), contaBancaria.getTitular(),
contaBancaria.getSaldo());
  }
}
```

A classe Zelador gerencia os estados das contas bancárias. Ela mantém uma lista de estados e um índice do estado atual. Métodos são fornecidos para adicionar mementos, obter o memento anterior e obter o memento posterior. Esta classe é implementada como um Singleton para garantir que apenas uma instância dela exista.

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

public class Zelador {
  private static Zelador instancia;
  private List<IEstado> estados;
  private int indiceAtual;
```

```
private Zelador() {
    estados = new ArrayList<>();
    indiceAtual = -1;
}

public static Zelador getInstancia() {
    if (instancia == null) {
        instancia = new Zelador();
    }
    return instancia;
}

public void adicionarMemento(IEstado memento) {
    estados.add(memento);
    indiceAtual++;
}

public IEstado getMementoAnterior() {
    if (indiceAtual > 0) {
        indiceAtual--;
        return estados.get(indiceAtual);
    }
    return null;
}

public IEstado getMementoPosterior() {
    if (indiceAtual < estados.size() - 1) {
        indiceAtual++;
        return estados.get(indiceAtual);
    }
    return null;
```

```
  }
}
```

A classe Main demonstra o uso do sistema de desfazer. Ela cria um zelador, adiciona contas bancárias e desfaz a adição da última conta. Em seguida, imprime a conta desfeita.

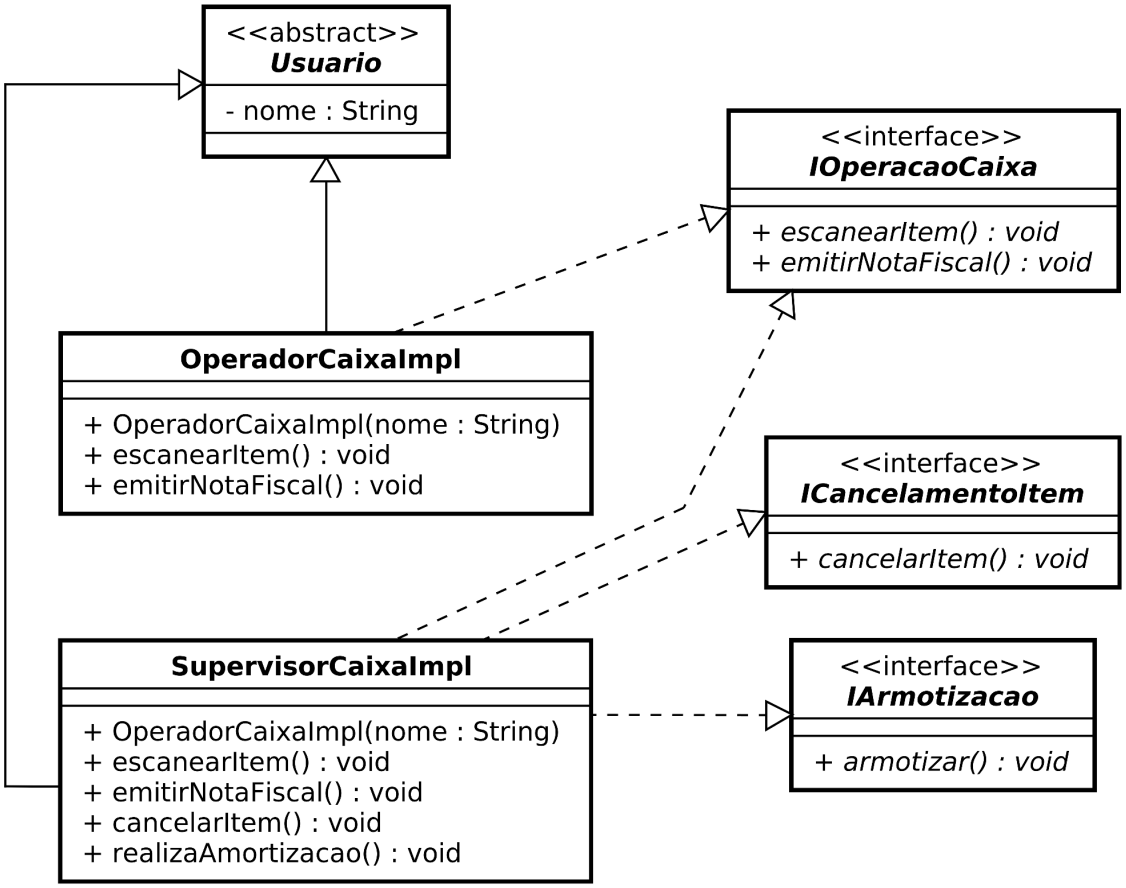
```
public class Main {
  public static void main(String[] args) {
    Zelador zelador = Zelador.getInstancia();

    // Adiciona uma conta bancária
    ContaBancaria conta1 = new ContaBancaria("12345", "João da Silva", 1000.0);
    zelador.adicionarMemento(new Memento(conta1));

    // Adiciona outra conta bancária
    ContaBancaria conta2 = new ContaBancaria("23456", "Maria da Silva", 2000.0);
    zelador.adicionarMemento(new Memento(conta2));

    // Desfaz a adição da última conta bancária
    IEstado estadoAnterior = zelador.getMementoAnterior();
    ContaBancaria contaDesfeita = estadoAnterior.getEstado();

    // Imprime a conta bancária desfeita
    System.out.println("Número: " + contaDesfeita.getNumero());
    System.out.println("Titular: " + contaDesfeita.getTitular());
    System.out.println("Saldo: " + contaDesfeita.getSaldo());
  }
}
```

Saída esperada ao executar o método main:

```
Número: 12345
Titular: João da Silva
Saldo: 1000.0
```

Solução do Exercício 84

Para solucionar esta violação, apresenta-se no diagrama de classes na Figura X, a solução proposta que podemos dividir a interface `ICaixa` em interfaces menores e mais específicas, segregando-as nas interfaces: `IOperacaoCaixa`, `ICancelamentoItem`, `IArmotizacao`.




```
public abstract class  Usuario {
  private String nome;
}
public interface IOperacaoCaixa {
  public void escanearItem();
  public void emitirNotaFiscal();
}

public interface ICancelamentoItem {
```

```
    public void cancelarItem();
}

public interface IArmotizacao {
    public void armotizar();
}

public class OperadorCaixaImpl extends Usuario implements IOperacaoCaixa {
    public OperadorCaixaImpl(String nome) {
        super(nome);
    }
    @Override
    public void escanearItem() {
        System.out.println("Realizando escaneamento do item...");
    }
    @Override
    public void emitirNotaFiscal() {
        System.out.println("Emitindo nota fiscal...");
    }
}

public class SupervisorCaixaImpl extends Usuario implements IOperacaoCaixa,
ICancelamentoItem, IArmotizacao {
    public OperadorCaixaImpl(String nome) {
        super(nome);
    }
    @Override
    public void escanearItem() {
```

```
        System.out.println("Realizando escaneamento do item...");
    }
    @Override
    public void emitirNotaFiscal() {
        System.out.println("Emitindo nota fiscal...");
    }

    @Override
    public void cancelarItem() {
        System.out.println("Cancelando item...");
    }

    @Override
    public void realizaAmortizacao() {
        System.out.println("Realizando armotização...");
    }
}
```

Essa abordagem mantém as interfaces coesas e ajuda a evitar o acoplamento excessivo entre as classes, garantindo que cada classe implemente apenas os métodos necessários para suas operações específicas e tornando o sistema de caixa do supermercado mais extensível e fácil de realizar manutenção.

Solução do Exercício 85

A classe abstrata SujeitoBase define o método template executar e métodos hook verificarPermissao, verificarPapel e verificarAutenticacao. O método executar é responsável por coordenar o processo de autorização do usuário. Ele chama os métodos hook em uma ordem específica: primeiro verificarAutenticacao, depois verificarPapel e, finalmente, verificarPermissao. Se todas as verificações forem bem-sucedidas, o método realizarAcao é

chamado para executar a ação. O método executar é final para garantir que a ordem das operações de autorização não seja alterada.

```
public abstract class SujeitoBase {
  protected String usuario;

  public SujeitoBase(String usuario) {
    this.usuario = usuario;
  }

  public final void executar(String acao) {
    if (verificarAutenticacao()) {
      if (verificarPapel() && verificarPermissao(acao)) {
        realizarAcao(acao);
      } else {
        System.out.println("Acesso negado: Permissões insuficientes.");
      }
    } else {
      System.out.println("Acesso negado: Usuário não autenticado.");
    }
  }

  protected abstract boolean verificarPermissao(String acao);
  protected abstract boolean verificarPapel();
  protected abstract boolean verificarAutenticacao();

  private void realizarAcao(String acao) {
    System.out.println("Executando ação: " + acao);
  }
}
```

Na classe SujeitoAdmin, que estende SujeitoBase, os métodos hook são implementados para verificar as permissões específicas para administradores. Nesta implementação,

administradores têm permissão para todas as ações, têm o papel de admin e estão autenticados.

```
public class SujeitoAdmin extends SujeitoBase {
  private RepositorioUsuario repositorio;

  public SujeitoAdmin(String usuario, RepositorioUsuario repositorio) {
    super(usuario);
    this.repositorio = repositorio;
  }

  @Override
  protected boolean verificarPermissao(String acao) {
    return repositorio.verificarPermissao(usuario, acao);
  }

  @Override
  protected boolean verificarPapel() {
    return repositorio.verificarPapel(usuario, "admin");
  }

  @Override
  protected boolean verificarAutenticacao() {
    return repositorio.estaAutenticado(usuario);
  }
}
```

Na classe SujeitoUsuario, que também estende SujeitoBase, os métodos hook são implementados para verificar as permissões específicas para usuários. Nesta implementação, usuários têm permissão apenas para ações de leitura, têm o papel de usuário e estão autenticados.

```
public class SujeitoUsuario extends SujeitoBase {
  private RepositorioUsuario repositorio;
```

```
    public SujeitoUsuario(String usuario, RepositorioUsuario repositorio) {
        super(usuario);
        this.repositorio = repositorio;
    }

    @Override
    protected boolean verificarPermissao(String acao) {
        return repositorio.verificarPermissao(usuario, acao);
    }

    @Override
    protected boolean verificarPapel() {
        return repositorio.verificarPapel(usuario, "usuario");
    }

    @Override
    protected boolean verificarAutenticacao() {
        return repositorio.estaAutenticado(usuario);
    }
}
```

A classe RepositorioUsuario simula um banco de dados de usuários, papéis e permissões. Ela contém métodos para verificar permissões, papéis e autenticação, além de métodos para autenticar usuários.

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class RepositorioUsuario {
    private Map<String, Usuario> usuarios;
```

```
public RepositorioUsuario() {
    usuarios = new HashMap<>();
    carregarDados();
}

private void carregarDados() {
    Usuario admin = new Usuario("admin", "admin123");
    admin.adicionarPapel("admin");
    admin.adicionarPermissao("leitura");
    admin.adicionarPermissao("escrita");

    Usuario user = new Usuario("user", "user123");
    user.adicionarPapel("usuario");
    user.adicionarPermissao("leitura");

    usuarios.put(admin.getNome(), admin);
    usuarios.put(user.getNome(), user);
}

public boolean verificarPermissao(String nomeUsuario, String permissao) {
    Usuario usuario = usuarios.get(nomeUsuario);
    return usuario != null && usuario.getPermissoes().contains(permissao);
}

public boolean verificarPapel(String nomeUsuario, String papel) {
    Usuario usuario = usuarios.get(nomeUsuario);
    return usuario != null && usuario.getPapeis().contains(papel);
}

public boolean estaAutenticado(String nomeUsuario) {
    Usuario usuario = usuarios.get(nomeUsuario);
```

```
        return usuario != null && usuario.isAutenticado();
    }

    public boolean autenticarUsuario(String nomeUsuario, String senha) {
        Usuario usuario = usuarios.get(nomeUsuario);
        if (usuario != null && usuario.getSenha().equals(senha)) {
            usuario.setAutenticado(true);
            return true;
        }
        return false;
    }

    public Map<String, Usuario> getUsuarios(){
        return this.usuarios;
    }
}
```

A classe Usuario representa a entidade do usuário, com atributos para nome, senha, papéis, permissões e estado de autenticação.

```
import java.util.HashSet;
import java.util.Set;

public class Usuario {
    private String nome;
    private String senha;
    private Set<String> papeis;
    private Set<String> permissoes;
    private boolean autenticado;

    public Usuario(String nome, String senha) {
        this.nome = nome;
        this.senha = senha;
```

```
    this.papeis = new HashSet<>();
    this.permissoes = new HashSet<>();
    this.autenticado = false;
}

public String getNome() {
    return nome;
}

public String getSenha() {
    return senha;
}

public Set<String> getPapeis() {
    return papeis;
}

public void adicionarPapel(String papel) {
    this.papeis.add(papel);
}

public Set<String> getPermissoes() {
    return permissoes;
}

public void adicionarPermissao(String permissao) {
    this.permissoes.add(permissao);
}

public boolean isAutenticado() {
    return autenticado;
}
```

```
  public void setAutenticado(boolean autenticado) {
    this.autenticado = autenticado;
  }
}
```

Na classe Principal, demonstramos o uso dessas classes. Criamos instâncias de RepositorioUsuario, autenticamos os usuários admin e user, e utilizamos instâncias de SujeitoAdmin e SujeitoUsuario para executar ações como leitura e escrita. Os resultados das verificações são impressos no console, mostrando se a ação foi permitida ou negada com base nas permissões e papéis dos usuários.

```
public class Principal {
  public static void main(String[] args) {
    RepositorioUsuario repositorio = new RepositorioUsuario();

    repositorio.autenticarUsuario("admin", "admin123");
    repositorio.autenticarUsuario("user", "user123");

    SujeitoBase admin = new SujeitoAdmin("admin", repositorio);
    admin.executar("leitura");
    admin.executar("escrita");

    SujeitoBase user = new SujeitoUsuario("user", repositorio);
    user.executar("leitura");
    user.executar("escrita");

    System.out.println("\nTentativa de acesso com credenciais inválidas:");
    if (!repositorio.autenticarUsuario("admin", "senhaIncorreta")) {
      System.out.println("Falha na autenticação para o usuário admin com senha incorreta.");
    }
    SujeitoBase userWithInvalidAuth = new SujeitoUsuario("user", repositorio);
    Map<String, Usuario> usuarios = repositorio.getUsuarios();
```

```
        usuarios.get("user").setAutenticado(false);
        userWithInvalidAuth.executar("leitura");
    }
}
```

A saída esperada após a execução do método main é a seguinte:

Executando ação: leitura
Executando ação: escrita
Executando ação: leitura
Acesso negado: Permissões insuficientes.

Tentativa de acesso com credenciais inválidas:
Falha na autenticação para o usuário admin com senha incorreta.
Acesso negado: Usuário não autenticado.

Solução do Exercício 86

Para resolver a violação do Princípio da Substituição de Liskov apresentada no cenário do processador de documentos, é necessário garantir que as subclasses mantenham a coerência nas pós-condições estabelecidas pela superclasse ProcessadorDocumento. Assim, todas as subclasses devem aderir rigorosamente às pós-condições que qualquer documento considerado como validado deve estar livre de inconsistências identificadas pelo processo de validação.

Código Corrigido
Superclasse ProcessadorDocumento

```
public class ProcessadorDocumento {
    public boolean validarDocumento(IDocumento documento) {
        // Verificação básica
        if (documento.getConteudo().contains("ERRO")) {
            documento.setValidado(false);
        } else {
            documento.setValidado(true);
```

```
        }
        return documento.isValidado();
    }
}
```

Subclasse ProcessadorDocumentoFinanceiro

```
public class ProcessadorDocumentoFinanceiro extends ProcessadorDocumento {
    @Override
    public boolean validarDocumento(IDocumento documento) {
        // Primeiro, realiza a validação básica usando a implementação da superclasse
        boolean isValidoBase = super.validarDocumento(documento);
        // Se passar pela validação básica, realiza verificações financeiras adicionais
        if (isValidoBase) {
            return validarConformidadeFinanceira(documento);
        }
        return isValidoBase;
    }

    private boolean validarConformidadeFinanceira(IDocumento documento) {
        // Simula uma verificação financeira mais específica
        if (documento.getConteudo().contains("FALHA_FINANCEIRA")) {
            documento.setValidado(false);
            return false;
        }
        return true;
    }
}
```

Neste código, a classe ProcessadorDocumentoFinanceiro mantém a consistência com a pós-condição estabelecida pela superclasse ProcessadorDocumento, garantindo que, se o documento passar pela validação básica, sua validade seja avaliada sem ser imediatamente invalidada por verificações adicionais, a menos que especificamente falhe nas verificações financeiras. Essa abordagem assegura que as modificações introduzidas na

subclassificação não violam o contrato estabelecido pela superclasse, mantendo a integridade e confiabilidade do sistema de validação de documentos.

Classe Principal Main

```
public class Main {
    public static void main(String[] args) {
        DocumentoImpl documento = new DocumentoImpl("CONTRATO; FALHA_FINANCEIRA");

        ProcessadorDocumento processador = new ProcessadorDocumentoFinanceiro();

        boolean resultadoValidacao = processador.validarDocumento(documento);

        System.out.println("Documento validado: " + resultadoValidacao);
        System.out.println("Estado de validação no documento: " + documento.isValidado());

        // Verifica se a validação do documento foi conforme esperado
        if (resultadoValidacao && documento.isValidado()) {
            System.out.println("O documento passou por todas as verificações e está
devidamente validado.");
        } else {
            System.out.println("O documento falhou em alguma verificação e não está validado.");
        }
    }
}
```

A saída final comunica se o documento foi validado corretamente ou se houve falhas nas verificações adicionais, mantendo a consistência com a pós-condição original definida pela superclasse ProcessadorDocumento.

Solução do Exercício 87

Para resolver os problemas apresentados, utilizamos os padrões de projeto Strategy, Chain of Responsibility e Memento. O padrão Strategy permitiu configurar o tipo de moeda utilizada pelo caixa eletrônico, por sua vez o padrão Chain of Responsibility possibilitou que diferentes

tratadores processem os valores de cédulas e moedas de forma modular. E o padrão Memento para salvar e restaurar o estado do sistema, garantindo consistência em caso de falha.

Começamos criando a classe abstrata MoedaStrategy para definir a configuração de moeda.

```
public abstract class MoedaStrategy {
  private final String simbolo;
  private final int valorMinimoCedula;

  public MoedaStrategy(String simbolo, int valorMinimoCedula) {
    this.simbolo = simbolo;
    this.valorMinimoCedula = valorMinimoCedula;
  }

  public String getSimbolo() {
    return simbolo;
  }

  public int getValorMinimoCedula() {
    return valorMinimoCedula;
  }
}
```

Em seguida, é criado a classe MoedaRealStrategy que estende a MoedaStrategy, sendo uma classe específica para a moeda Real.

```
public class MoedaRealStrategy extends MoedaStrategy {

  public MoedaRealStrategy(int valorMinimoCedula) {
    super("R$", valorMinimoCedula);
  }
}
```

Em seguida, criamos a classe Tratador, que será responsável por processar um valor específico de cédula ou moeda.

```
import java.math.BigDecimal;

public class Tratador {
  protected double valorCedulaMoeda;
  protected int quantidadeDisponivel;
  protected String mensagem = "";
  private MoedaStrategy estrategiaMoeda;

  public Tratador(double valorCedulaMoeda, int quant) {
     this.quantidadeDisponivel = quant;
     this.valorCedulaMoeda = valorCedulaMoeda;
  }

  public final BigDecimal handleRequest(BigDecimal valor) throws Exception {
     BigDecimal resto = valor;
     valor = valor.setScale(2, BigDecimal.ROUND_HALF_UP);
     if (accept(valor)) {
        resto = determinaQuantidade(valor);
     }
     return resto;
  }

  public final boolean accept(BigDecimal valor) {
     return valor.doubleValue() >= this.valorCedulaMoeda && quantidadeDisponivel > 0;
  }

  public final BigDecimal determinaQuantidade(BigDecimal valor) {
     String tipo = " moeda(s)";
     if (valor.doubleValue() >= estrategiaMoeda.getValorMinimoCedula()) {
        tipo = " nota(s)";
```

```
        }
        BigDecimal resto;
        int quantidadeNecessaria = (int) (valor.doubleValue() / this.valorCedulaMoeda);
        mensagem = tipo + " de " + estrategiaMoeda.getSimbolo() + " " + this.valorCedulaMoeda
+ "\n";
        if (quantidadeDisponivel >= quantidadeNecessaria) {
            mensagem = quantidadeNecessaria + mensagem;
            resto = new BigDecimal(valor.doubleValue() - (quantidadeNecessaria *
this.valorCedulaMoeda));
            quantidadeDisponivel -= quantidadeNecessaria;
        } else {
            mensagem = quantidadeDisponivel + mensagem;
            resto = new BigDecimal(valor.doubleValue() - (quantidadeDisponivel *
this.valorCedulaMoeda));
        }
        return resto;
    }

    public final String getMensagem() {
        return mensagem;
    }

    public final void setMensagem(String mensagem) {
        this.mensagem = mensagem;
    }

    public final int getQuantidade() {
        return quantidadeDisponivel;
    }

    public final double getValorCedulaMoeda() {
        return valorCedulaMoeda;
```

```
  }

  public final void setTipoMoeda(MoedaStrategy estrategiaMoeda) {
    this.estrategiaMoeda = estrategiaMoeda;
  }
}
```

Com os tratadores definidos, criamos a classe CaixaEletronicoService, que será responsável por gerenciar as operações de saque e restaurar o estado do sistema em caso de falha.

```
import java.math.BigDecimal;
import java.util.ArrayList;

public class CaixaEletronicoService {
  private final static String MENSAGEM = "É necessário indicar a moeda usada pelo Caixa Eletrônico";
  private ArrayList<Tratador> tratadores = new ArrayList<>();
  private MoedaStrategy estrategiaMoeda;

  public void addTratador(Tratador tratador) throws Exception {
    if (estrategiaMoeda != null) {
      tratador.setTipoMoeda(estrategiaMoeda);
      tratadores.add(tratador);
    } else {
      throw new Exception(MENSAGEM);
    }
  }

  public void efetuarSaque(double valorDouble) throws Exception {
    Zelador zelador = Zelador.getInstance();
    zelador.setMemento(this.criaMemento());
    if (estrategiaMoeda == null) {
      throw new Exception(MENSAGEM);
```

```
        }
        BigDecimal valor = new BigDecimal(valorDouble).setScale(2,
BigDecimal.ROUND_HALF_EVEN);
        StringBuilder mensagem = new StringBuilder();
        mensagem.append(valor.toPlainString());
        mensagem.append("\n");
        for (Tratador cedulaHandler : tratadores) {
            valor = cedulaHandler.handleRequest(valor);
            mensagem.append(cedulaHandler.getMensagem());
            cedulaHandler.setMensagem("");
        }
        if (valor.doubleValue() == 0) {
            System.out.print("\nVALOR DO SAQUE: " + estrategiaMoeda.getSimbolo());
            System.out.println(mensagem);
        } else {
            System.out.println("\nNão foi possível realizar o saque. Dinheiro insuficiente");
            restaurar(Zelador.getInstance().getCopia());
        }
    }

    void restaurar(MementoCaixaEletronico copia) {
        tratadores.clear();
        tratadores = copia.getTratadores();
    }

    public void setTipoMoeda(MoedaStrategy estrategiaMoeda) {
        this.estrategiaMoeda = estrategiaMoeda;
    }

    protected MementoCaixaEletronico criaMemento() {
        return new MementoCaixaEletronico(new ArrayList<>(tratadores));
    }
```

```
}
```

Para implementar o padrão Memento, criamos a classe MementoCaixaEletronico, que armazenará o estado dos tratadores, e a classe Zelador, que gerenciará esses mementos.

```
public class MementoCaixaEletronico {
  private ArrayList<Tratador> tratadores;

  MementoCaixaEletronico(ArrayList<Tratador> tratadores) {
    this.tratadores = new ArrayList<>(tratadores);
  }

  ArrayList<Tratador> getTratadores() {
    return tratadores;
  }
}



public class Zelador {
  protected MementoCaixaEletronico copia;
  private static Zelador instancia;

  private Zelador() {}

  public static Zelador getInstance() {
    if (instancia == null) {
      instancia = new Zelador();
    }
    return instancia;
  }

  public void setMemento(MementoCaixaEletronico copia) {
    this.copia = copia;
```

```
  }

  public MementoCaixaEletronico getCopia() {
    MementoCaixaEletronico memento = copia;
    return memento;
  }
}
```

Por fim, a classe Principal verifica o funcionamento do caixa eletrônico refatorado. Nesta classe, configuramos o caixa eletrônico com diferentes tratadores e tentamos realizar um saque, ilustrando o processo completo.

```
public class Principal {
  public static void main(String[] args) {
    try {
      CaixaEletronicoService caixaEletronico = new CaixaEletronicoService();
      caixaEletronico.setTipoMoeda(new MoedaRealStrategy(2));
      caixaEletronico.addTratador(new Tratador(100, 4));
      caixaEletronico.addTratador(new Tratador(50, 2));
      caixaEletronico.addTratador(new Tratador(10, 5));
      caixaEletronico.addTratador(new Tratador(1, 5));
      caixaEletronico.addTratador(new Tratador(0.01, 5));
      caixaEletronico.efetuarSaque(500.03);
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

A saída esperada após a execução do método main é a seguinte:

```
VALOR DO SAQUE: R$500.03
4 nota(s) de R$ 100.0
2 nota(s) de R$ 50.0
3 moeda(s) de R$ 0.01
```

Solução do Exercício 88

Para resolver o problema de desenvolver um sistema de avaliação de desempenho dos funcionários de uma empresa utilizando o padrão de projeto Template Method, começamos criando uma classe abstrata que define o esqueleto do algoritmo de avaliação. A classe AvaliacaoDesempenho contém o método avaliarDesempenho, que descreve as etapas gerais do processo de avaliação: coleta de dados, cálculo da nota, emissão do relatório e finalização da avaliação.

```
public abstract class AvaliacaoDesempenho {
  public void avaliarDesempenho() {
    coletarDados();
    calcularNota();
    emitirRelatorio();
    System.out.println("Avaliação de desempenho concluída");
  }

  void coletarDados() {
    // Lógica de coleta dos dados necessários para a avaliação
    System.out.println("Coletando dados para a avaliação...");
  }

  abstract void calcularNota();
  abstract void emitirRelatorio();
}
```

Em seguida, implementamos a avaliação de produtividade. A classe AvaliacaoProdutividade herda de AvaliacaoDesempenho e define o comportamento específico para calcular a nota de produtividade e emitir o relatório correspondente.

```
public class AvaliacaoProdutividade extends AvaliacaoDesempenho {
  @Override
  void calcularNota() {
    System.out.println("Calculando nota de produtividade...");
```

```
    }

    @Override
    void emitirRelatorio() {
        System.out.println("Emitindo relatório de produtividade...");
    }
}
```

Posteriormente, implementamos a avaliação da qualidade do trabalho. A classe AvaliacaoQualidadeTrabalho segue o mesmo padrão, definindo o comportamento para calcular a nota de qualidade do trabalho e emitir o relatório correspondente.

```
public class AvaliacaoQualidadeTrabalho extends AvaliacaoDesempenho {
    @Override
    void calcularNota() {
        System.out.println("Calculando nota de qualidade do trabalho...");
    }

    @Override
    void emitirRelatorio() {
        System.out.println("Emitindo relatório de qualidade do trabalho...");
    }
}
```

Adicionalmente, implementamos a avaliação do atendimento ao cliente. A classe AvaliacaoAtendimentoCliente herda de AvaliacaoDesempenho e define o comportamento específico para calcular a nota de atendimento ao cliente e emitir o relatório correspondente.

```
public class AvaliacaoAtendimentoCliente extends AvaliacaoDesempenho {
    @Override
    void calcularNota() {
        System.out.println("Calculando nota de atendimento ao cliente...");
    }
```

```
  @Override
  void emitirRelatorio() {
    System.out.println("Emitindo relatório de atendimento ao cliente...");
  }
}
```

Finalmente, implementamos a classe principal para demonstrar o uso do sistema de avaliação de desempenho. Na classe Main, criamos instâncias de cada tipo de avaliação, executamos o processo de avaliação completo para cada uma e exibimos os resultados finais.

```
public class Principal {
  public static void main(String[] args) {
    AvaliacaoDesempenho avaliacaoProdutividade =
            new AvaliacaoProdutividade();
    AvaliacaoDesempenho avaliacaoQualidadeTrabalho =
            new AvaliacaoQualidadeTrabalho();
    AvaliacaoDesempenho avaliacaoAtendimentoCliente =
            new AvaliacaoAtendimentoCliente();

    avaliacaoProdutividade.avaliarDesempenho();
    System.out.println();
    avaliacaoQualidadeTrabalho.avaliarDesempenho();
    System.out.println();
    avaliacaoAtendimentoCliente.avaliarDesempenho();
  }
}
```

A saída no console esperada após a execução do código é a seguinte.

```
Coletando dados para a avaliação...
Calculando nota de produtividade...
Emitindo relatório de produtividade...
Avaliação de desempenho concluída
```

Coletando dados para a avaliação...
Calculando nota de qualidade do trabalho...
Emitindo relatório de qualidade do trabalho...
Avaliação de desempenho concluída

Coletando dados para a avaliação...
Calculando nota de atendimento ao cliente...
Emitindo relatório de atendimento ao cliente...
Avaliação de desempenho concluída

Desafio: Para melhorar esta implementação, desenvolva uma solução que realize a geração de relatórios de cada métodos de avaliação em diferentes formatos aplicando o padrão de projetos Strategy.

Solução do Exercício 89

Para resolver o problema de implementar um sistema de venda de ingressos para eventos esportivos utilizando o padrão de projeto Observer, começaremos definindo uma classe abstrata Ingresso. Esta classe será estendida por subclasses específicas para cada tipo de ingresso: IngressoCadeiraPremium, IngressoArquibancada e IngressoCamarote. A classe Ingresso gerencia o preço e os observadores, além de notificar sobre mudanças de preço e descontos.

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

public abstract class Ingresso {
  private double preco;
  private double desconto;
  private List<ObservadorIngresso> observadores;

  public Ingresso(double preco, double desconto) {
    this.preco = preco;
    this.desconto = desconto;
```

```
    observadores = new ArrayList<>();
}

public void adicionarObservador(ObservadorIngresso observador) {
    observadores.add(observador);
}

public void removerObservador(ObservadorIngresso observador) {
    observadores.remove(observador);
}

public void setPreco(double preco) {
    this.preco = preco;
    notificarPreco();
}

public void setDesconto(double desconto) {
    this.desconto = desconto;
    notificarDesconto();
}

public double getPreco() {
    return preco;
}

public double getDesconto() {
    return desconto;
}

private void notificarPreco() {
    for (ObservadorIngresso observador : observadores) {
        observador.atualizarPreco();
```

```
    }
  }

  private void notificarDesconto() {
    for (ObservadorIngresso observador : observadores) {
      observador.atualizarDesconto();
    }
  }
}
```

Em seguida, criamos as subclasses para os tipos específicos de ingressos.

```
public class IngressoCadeiraPremium extends Ingresso {
  public IngressoCadeiraPremium(double preco, double desconto) {
    super(preco, desconto);
  }
}

public class IngressoArquibancada extends Ingresso {
  public IngressoArquibancada(double preco, double desconto) {
    super(preco, desconto);
  }
}

public class IngressoCamarote extends Ingresso {
  public IngressoCamarote(double preco, double desconto) {
    super(preco, desconto);
  }
}
```

Definimos uma classe abstrata ObservadorIngresso que será implementada por todas as classes que desejam ser notificadas sobre mudanças nos ingressos.

```
public abstract class ObservadorIngresso {
```

```
    protected Ingresso ingresso;

    public ObservadorIngresso(Ingresso ingresso) {
        this.ingresso = ingresso;
        this.ingresso.adicionarObservador(this);
    }

    public abstract void atualizarPreco();

    public abstract void atualizarDesconto();
}
```

Implementamos classes concretas de observadores para preço, desconto e promoções de ingressos.

```
public class ObservadorPrecoIngresso extends ObservadorIngresso {
    public ObservadorPrecoIngresso(Ingresso ingresso) {
        super(ingresso);
    }

    @Override
    public void atualizarPreco() {
        System.out.println("O preço do ingresso mudou para: " + ingresso.getPreco());
    }

    @Override
    public void atualizarDesconto() {
        // Lógica para notificar os observadores
    }
}

public class ObservadorDescontoIngresso extends ObservadorIngresso {
    public ObservadorDescontoIngresso(Ingresso ingresso) {
```

```
        super(ingresso);
    }

    @Override
    public void atualizarPreco() {
        // Lógica para notificar os observadores
    }

    @Override
    public void atualizarDesconto() {
        System.out.println("O desconto do ingresso mudou para: " + ingresso.getDesconto());
    }
}

public class ObservadorPromocaoIngresso extends ObservadorIngresso {
    public ObservadorPromocaoIngresso(Ingresso ingresso) {
        super(ingresso);
    }

    @Override
    public void atualizarPreco() {
        if (ingresso.getPreco() < 50) {
            System.out.println("Promoção! O preço do ingresso está abaixo de 50: " +
ingresso.getPreco());
        }
    }

    @Override
    public void atualizarDesconto() {
        if (ingresso.getDesconto() > 20) {
            System.out.println("Grande desconto! O desconto do ingresso está acima de 20%: " +
ingresso.getDesconto());
```

```
    }
  }
}
```

Por fim, criamos a classe principal Main para demonstrar o uso do sistema, alterando o preço e aplicando descontos aos ingressos, e mostrando como os observadores são notificados.

```
public class Main {
  public static void main(String[] args) {
    Ingresso cadeiraPremium = new IngressoCadeiraPremium(100.0, 0.0);
    Ingresso arquibancada = new IngressoArquibancada(50.0, 0.0);
    Ingresso camarote = new IngressoCamarote(200.0, 0.0);

    ObservadorPrecoIngresso observadorPreco1 =
            new ObservadorPrecoIngresso(cadeiraPremium);
    ObservadorDescontoIngresso observadorDesconto1 =
            new ObservadorDescontoIngresso(cadeiraPremium);
    ObservadorPromocaoIngresso observadorPromocao1 =
            new ObservadorPromocaoIngresso(cadeiraPremium);

    ObservadorPrecoIngresso observadorPreco2 =
            new ObservadorPrecoIngresso(arquibancada);
    ObservadorDescontoIngresso observadorDesconto2 =
            new ObservadorDescontoIngresso(arquibancada);
    ObservadorPromocaoIngresso observadorPromocao2 =
            new ObservadorPromocaoIngresso(arquibancada);

    ObservadorPrecoIngresso observadorPreco3 =
            new ObservadorPrecoIngresso(camarote);
    ObservadorDescontoIngresso observadorDesconto3 =
            new ObservadorDescontoIngresso(camarote);
    ObservadorPromocaoIngresso observadorPromocao3 =
```

```
                new ObservadorPromocaoIngresso(camarote);

        System.out.println("Alterando o preço do ingresso Cadeira Premium para 120.0");
        cadeiraPremium.setPreco(120.0);

        System.out.println("\nAlterando o desconto do ingresso Cadeira Premium para 15.0");
        cadeiraPremium.setDesconto(15.0);

        System.out.println("\nAlterando o preço do ingresso Arquibancada para 45.0");
        arquibancada.setPreco(45.0);

        System.out.println("\nAlterando o desconto do ingresso Arquibancada para 25.0");
        arquibancada.setDesconto(25.0);

        System.out.println("\nAlterando o preço do ingresso Camarote para 150.0");
        camarote.setPreco(150.0);

        System.out.println("\nAlterando o desconto do ingresso Camarote para 30.0");
        camarote.setDesconto(30.0);
    }
}
```

A saída no console esperada após a execução do código é a seguinte.

```
Alterando o preço do ingresso Cadeira Premium para 120.0
O preço do ingresso mudou para: 120.0

Alterando o desconto do ingresso Cadeira Premium para 15.0
O desconto do ingresso mudou para: 15.0

Alterando o preço do ingresso Arquibancada para 45.0
O preço do ingresso mudou para: 45.0
Promoção! O preço do ingresso está abaixo de 50: 45.0
```

Alterando o desconto do ingresso Arquibancada para 25.0
O desconto do ingresso mudou para: 25.0
Grande desconto! O desconto do ingresso está acima de 20%: 25.0

Alterando o preço do ingresso Camarote para 150.0
O preço do ingresso mudou para: 150.0

Alterando o desconto do ingresso Camarote para 30.0
O desconto do ingresso mudou para: 30.0
Grande desconto! O desconto do ingresso está acima de 20%: 30.0

Solução do Exercício 90

Para resolver o problema de gerar relatórios automáticos de vendas diários, semanais e mensais utilizando o padrão de projeto Observer, iniciamos pela implementação da classe SistemaVenda. Esta classe gerencia as vendas e notifica os observadores sempre que uma venda é registrada. Ela mantém uma lista de observadores que são notificados sempre que o valor de uma venda é registrado, permitindo que eles possam gerar os relatórios apropriados.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class SistemaVenda {
  private List<IObservador> observadores;
  private float valorVenda;

  public SistemaVenda() {
    this.observadores = new ArrayList<>();
  }

  public void adicionarObservador(IObservador observador) {
    observadores.add(observador);
```

```
  }

  public void removerObservador(IObservador observador) {
    observadores.remove(observador);
  }

  public void registrarVenda(float valorVenda) {
    this.valorVenda = valorVenda;
    notificarObservadores();
  }

  private void notificarObservadores() {
    for (IObservador observador : observadores) {
      observador.atualizar(valorVenda, LocalDate.now());
    }
  }
}
```

Em seguida, definimos a interface IObservador, que exige que os observadores implementem o método atualizar. Este método recebe o valor da venda e a data da venda, permitindo que os observadores reajam apropriadamente às mudanças no sistema de vendas.

```
import java.time.LocalDate;

public interface IObservador {
  void atualizar(float valorVenda, LocalDate dataVenda);
}
```

Para a geração de PDF, utilizaremos a biblioteca iText. A seguir, a configuração no pom.xml para incluir a dependência necessária.

```
  <dependencies>
    <dependency>
      <groupId>com.itextpdf</groupId>
```

```
        <artifactId>itextpdf</artifactId>
        <version>5.5.13.2</version>
    </dependency>
</dependencies>
```

Para implementar a geração de relatórios diários, criamos a classe RelatorioDiario. Esta classe coleta os dados da venda diária e gera um arquivo PDF correspondente, que é armazenado em um diretório específico.

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.time.LocalDate;
import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.pdf.PdfWriter;

public class RelatorioDiario implements IObservador {
  private static final String DIRETORIO_RELATORIOS = "relatorios";

  @Override
  public void atualizar(float valorVenda, LocalDate dataVenda) {
    StringBuilder conteudoRelatorio = new StringBuilder();
    conteudoRelatorio.append("Relatório de vendas - " + dataVenda.toString() + "\n");
    conteudoRelatorio.append("Total de vendas: " + valorVenda + "\n");

    gerarPDF(conteudoRelatorio.toString(), "relatorio-diario-" + dataVenda.toString() + ".pdf");
  }

  private void gerarPDF(String conteudo, String nomeArquivo) {
    File diretorio = new File(DIRETORIO_RELATORIOS);
    if (!diretorio.exists()) {
      diretorio.mkdir();
```

```
        }

        Document document = new Document();
        try {
            PdfWriter.getInstance(document, new FileOutputStream(DIRETORIO_RELATORIOS + "/"
+ nomeArquivo));
            document.open();
            document.add(new Paragraph(conteudo));
        } catch (DocumentException | FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            document.close();
        }
    }
}
```

De forma semelhante, implementamos a classe RelatorioSemanal que gera relatórios semanais. Esta classe determina o início e o fim da semana correspondente à data da venda e gera o relatório em formato PDF, armazenando-o no diretório especificado.

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.time.DayOfWeek;
import java.time.LocalDate;
import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.pdf.PdfWriter;

public class RelatorioSemanal implements IObservador {
    private static final String DIRETORIO_RELATORIOS = "relatorios";

    @Override
```

```
    public void atualizar(float valorVenda, LocalDate dataVenda) {
        StringBuilder conteudoRelatorio = new StringBuilder();
        LocalDate inicioSemana = dataVenda.with(DayOfWeek.MONDAY);
        LocalDate fimSemana = dataVenda.with(DayOfWeek.SUNDAY);
        conteudoRelatorio.append("Relatório de vendas - Semana de " + inicioSemana.toString()
+ " a " + fimSemana.toString() + "\n");
        conteudoRelatorio.append("Total de vendas: " + valorVenda + "\n");

        gerarPDF(conteudoRelatorio.toString(), "relatorio-semanal-" + inicioSemana.toString() +
"-" + fimSemana.toString() + ".pdf");
    }

    private void gerarPDF(String conteudo, String nomeArquivo) {
        File diretorio = new File(DIRETORIO_RELATORIOS);
        if (!diretorio.exists()) {
            diretorio.mkdir();
        }

        Document document = new Document();
        try {
            PdfWriter.getInstance(document, new FileOutputStream(DIRETORIO_RELATORIOS + "/"
+ nomeArquivo));
            document.open();
            document.add(new Paragraph(conteudo));
        } catch (DocumentException | FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            document.close();
        }
    }
}
```

A classe RelatorioMensal implementa a lógica para a geração de relatórios mensais. Esta classe coleta dados sobre todas as vendas do mês e gera o relatório correspondente em formato PDF.

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.time.LocalDate;
import java.time.Month;
import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.pdf.PdfWriter;

public class RelatorioMensal implements IObservador {
  private static final String DIRETORIO_RELATORIOS = "relatorios";

  @Override
  public void atualizar(float valorVenda, LocalDate dataVenda) {
    StringBuilder conteudoRelatorio = new StringBuilder();
    Month mes = dataVenda.getMonth();
    int ano = dataVenda.getYear();
    conteudoRelatorio.append("Relatório de vendas - " + mes.toString() + "/" + ano + "\n");
    conteudoRelatorio.append("Total de vendas: " + valorVenda + "\n");

    gerarPDF(conteudoRelatorio.toString(), "relatorio-mensal-" + mes.toString() + "-" + ano +
".pdf");
  }

  private void gerarPDF(String conteudo, String nomeArquivo) {
    File diretorio = new File(DIRETORIO_RELATORIOS);
    if (!diretorio.exists()) {
      diretorio.mkdir();
    }
```

```
        Document document = new Document();
        try {
            PdfWriter.getInstance(document, new FileOutputStream(DIRETORIO_RELATORIOS + "/"
+ nomeArquivo));
            document.open();
            document.add(new Paragraph(conteudo));
        } catch (DocumentException | FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            document.close();
        }
    }
}
```

Finalmente, na classe principal Main, instanciamos o sistema de vendas, criamos os observadores para cada tipo de relatório e registramos algumas vendas para demonstrar a funcionalidade completa do sistema.

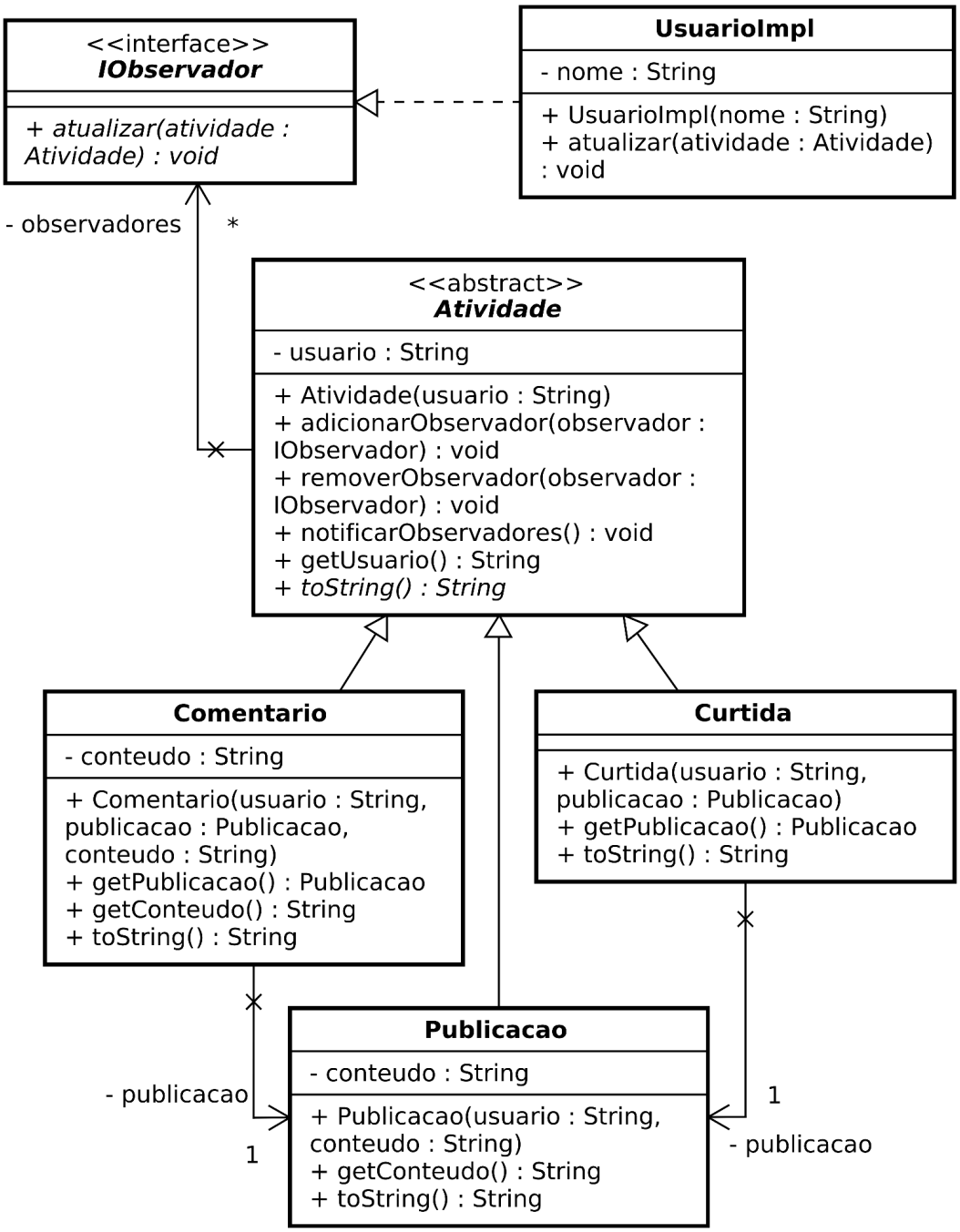
```
public class Main {
    public static void main(String[] args) {
        SistemaVenda sistemaVenda = new SistemaVenda();

        IObservador relatorioDiario = new RelatorioDiario();
        IObservador relatorioSemanal = new RelatorioSemanal();
        IObservador relatorioMensal = new RelatorioMensal();

        sistemaVenda.adicionarObservador(relatorioDiario);
        sistemaVenda.adicionarObservador(relatorioSemanal);
        sistemaVenda.adicionarObservador(relatorioMensal);

        sistemaVenda.registrarVenda(150.75f);
        sistemaVenda.registrarVenda(200.50f);
```

```
        sistemaVenda.registrarVenda(75.25f);
    }
}
```

Espera-se como output após a execução do método main, a geração de 3 arquivos do tipo PDF no diretório “relatorios”, dentro da pasta do projeto. Caso o diretório “relatorios” não exista, esta implementação garante a criação do mesmo.

Solução do Exercício 91

Para resolver o problema de implementar um sistema de notificações para uma rede social, acompanhe no diagrama de classes abaixo o esquema que utilizaremos o padrão de projeto Observer. Este padrão permitirá que os usuários se inscrevam para receber notificações sobre atividades específicas, como publicações, curtidas e comentários. Quando uma atividade é realizada, todos os inscritos receberão uma notificação.



Vamos começar definindo a classe base Atividade, que representará as atividades na rede social. Esta classe conterá uma lista de observadores e métodos para adicionar, remover e notificar esses observadores.

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

public abstract class Atividade {
```

```
    private String usuario;
    private List<IObservador> observadores = new ArrayList<>();

    public Atividade(String usuario) {
        this.usuario = usuario;
    }

    public void adicionarObservador(IObservador observador) {
        observadores.add(observador);
    }

    public void removerObservador(IObservador observador) {
        observadores.remove(observador);
    }

    public void notificarObservadores() {
        for (IObservador observador : observadores) {
            observador.atualizar(this);
        }
    }

    public String getUsuario() {
        return usuario;
    }

    @Override
    public abstract String toString();
}
```

Em seguida, criaremos subclasses concretas de Atividade para representar diferentes tipos de atividades na rede social. A primeira subclasse será Publicacao, que representará uma nova publicação feita por um usuário. Esta classe conterá o conteúdo da publicação e um método para retornar uma descrição da atividade.

```
public class Publicacao extends Atividade {
  private String conteudo;

  public Publicacao(String usuario, String conteudo) {
    super(usuario);
    this.conteudo = conteudo;
  }

  public String getConteudo() {
    return conteudo;
  }

  @Override
  public String toString() {
    return "Nova publicação de " + getUsuario() + ": " + getConteudo();
  }
}
```

Além das publicações, também precisamos representar curtidas. A classe Curtida indicará que um usuário curtiu uma publicação específica. Ela conterá uma referência à publicação que foi curtida e um método para retornar uma descrição da atividade.

```
public class Curtida extends Atividade {
  private Publicacao publicacao;

  public Curtida(String usuario, Publicacao publicacao) {
    super(usuario);
    this.publicacao = publicacao;
  }

  public Publicacao getPublicacao() {
    return publicacao;
  }
```

```
  @Override
  public String toString() {
                return "Nova curtida de " + getUsuario() + " em publicação de " +
getPublicacao().getUsuario();
  }
}
```

Por fim, criamos a classe Comentario para representar um comentário feito em uma publicação. Esta classe conterá o conteúdo do comentário e uma referência à publicação comentada, além de um método para retornar uma descrição da atividade.

```
public class Comentario extends Atividade {
  private Publicacao publicacao;
  private String conteudo;

  public Comentario(String usuario, Publicacao publicacao, String conteudo) {
    super(usuario);
    this.publicacao = publicacao;
    this.conteudo = conteudo;
  }

  public Publicacao getPublicacao() {
    return publicacao;
  }

  public String getConteudo() {
    return conteudo;
  }

  @Override
  public String toString() {
```

```
        return "Novo comentário de " + getUsuario() + " em publicação de " +
getPublicacao().getUsuario() + ": " + getConteudo();
    }
}
```

Agora, vamos definir a interface Observador, que será implementada pelos observadores que desejam ser notificados sobre as atividades. Esta interface conterá um método atualizar que será chamado quando uma atividade ocorrer.

```
public interface IObservador {
    void atualizar(Atividade atividade);
}
```

Em seguida, criaremos a classe Usuario, que representará um usuário da rede social. Esta classe implementará a interface Observador e terá um método para exibir a notificação da atividade.

```
public class UsuarioImpl implements IObservador {
    private String nome;

    public UsuarioImpl(String nome) {
        this.nome = nome;
    }

    @Override
    public void atualizar(Atividade atividade) {
        System.out.println(nome + ": " + atividade.toString());
    }
}
```

Finalmente, para testar o sistema de notificações, criaremos uma classe Principal que instanciará usuários, atividades e inscreverá os usuários nas atividades. Esta classe também simulará a realização das atividades para demonstrar as notificações.

```
public class Principal {
```

```
public static void main(String[] args) {
    // Cria usuários
    IObservador joao = new UsuarioImpl("João");
    IObservador maria = new UsuarioImpl("Maria");
    IObservador pedro = new UsuarioImpl("Pedro");

    // Cria publicações
    Publicacao publicacao1 = new Publicacao("João", "Olá mundo!");
    Publicacao publicacao2 = new Publicacao("Maria", "Bom dia!");

    // Inscreve usuários nas publicações
    publicacao1.adicionarObservador(joao);
    publicacao1.adicionarObservador(maria);
    publicacao1.adicionarObservador(pedro);
    publicacao2.adicionarObservador(joao);
    publicacao2.adicionarObservador(maria);

    // Realiza atividades
    publicacao1.notificarObservadores();
    publicacao2.notificarObservadores();

    Curtida curtida1 = new Curtida("João", publicacao1);
    Curtida curtida2 = new Curtida("Maria", publicacao2);

    curtida1.adicionarObservador(joao);
    curtida1.adicionarObservador(maria);
    curtida1.adicionarObservador(pedro);
    curtida2.adicionarObservador(joao);
    curtida2.adicionarObservador(maria);

    curtida1.notificarObservadores();
    curtida2.notificarObservadores();
```

```
        Comentario comentario1 = new Comentario("Pedro", publicacao1, "Muito bom!");
        Comentario comentario2 = new Comentario("Maria", publicacao2, "Adorei!");

        comentario1.adicionarObservador(joao);
        comentario1.adicionarObservador(maria);
        comentario1.adicionarObservador(pedro);
        comentario2.adicionarObservador(joao);
        comentario2.adicionarObservador(maria);

        comentario1.notificarObservadores();
        comentario2.notificarObservadores();
    }
}
```

A saída no console esperada após a execução do código é a seguinte.

```
João: Nova publicação de João: Olá mundo!
Maria: Nova publicação de João: Olá mundo!
Pedro: Nova publicação de João: Olá mundo!
João: Nova publicação de Maria: Bom dia!
Maria: Nova publicação de Maria: Bom dia!
João: Nova curtida de João em publicação de João
Maria: Nova curtida de João em publicação de João
Pedro: Nova curtida de João em publicação de João
João: Nova curtida de Maria em publicação de Maria
Maria: Nova curtida de Maria em publicação de Maria
João: Novo comentário de Pedro em publicação de João: Muito bom!
Maria: Novo comentário de Pedro em publicação de João: Muito bom!
Pedro: Novo comentário de Pedro em publicação de João: Muito bom!
João: Novo comentário de Maria em publicação de Maria: Adorei!
Maria: Novo comentário de Maria em publicação de Maria: Adorei!
```

Solução do Exercício 92

Para resolver o problema apresentado, utilizamos o padrão de projeto Abstract Factory para fabricar os comandos específicos para cada dispositivo. Isso facilita a adição e remoção de comandos ao controle, melhorando a manutenção e a extensibilidade do código.

Primeiro, criamos a interface IComando, que define um método executar.

```
public interface IComando {
  public void executar();
}
```

Em seguida, definimos uma interface comum IDispositivo para todos os dispositivos, que implementa métodos essenciais como ligar e desligar.

```
public interface IDispositivo {
  public String getLocalizacao();
  public boolean isLigado();
  public void ligar();
  public void desligar();
}
```

Criamos uma classe abstrata base DispositivoAbstrato que implementa a interface Dispositivo e fornece a funcionalidade básica para os dispositivos.

```
public abstract class DispositivoAbstrato implements IDispositivo {
  protected String localizacao;
  protected boolean ligado;

  public DispositivoAbstrato(String localizacao) {
    this.localizacao = localizacao;
    this.ligado = false;
  }

  @Override
  public String getLocalizacao() {
```

```
    return localizacao;
  }

  @Override
  public boolean isLigado() {
    return ligado;
  }

  @Override
  public void ligar() {
    ligado = true;
  }

  @Override
  public void desligar() {
    ligado = false;
  }
}
```

Para representar um dispositivo específico, como uma luz, criamos a classe Luz, que herda de DispositivoAbstrato e implementa métodos específicos para ligar e desligar a luz.

```
public class Luz extends DispositivoAbstrato {
  public Luz(String localizacao) {
    super(localizacao);
  }

  @Override
  public void ligar() {
    super.ligar();
    System.out.println("A luz na " + localizacao + " está ligada.");
  }
```

```
  @Override
  public void desligar() {
    super.desligar();
    System.out.println("A luz na " + localizacao + " está desligada.");
  }
}
```

Criamos também a classe Cortina, que possui métodos específicos para abrir e fechar a cortina.

```
public class Cortina extends DispositivoAbstrato {
  public Cortina(String localizacao) {
    super(localizacao);
  }

  public void abrir() {
    System.out.println("A cortina na " + localizacao + " está aberto.");
  }

  public void fechar() {
    System.out.println("A cortina na " + localizacao + " está fechada.");
  }
}
```

Em seguida, implementamos comandos concretos para ligar e desligar dispositivos, como a classe ComandoLigar que liga um dispositivo.

```
public class ComandoLigar implements IComando {
  private IDispositivo dispositivo;

  public ComandoLigar(IDispositivo dispositivo) {
    this.dispositivo = dispositivo;
  }
```

```
  @Override
  public void executar() {
    dispositivo.ligar();
  }
}
```

E a classe ComandoDesligar que desliga um dispositivo.

```
public class ComandoDesligar implements IComando {
  private IDispositivo dispositivo;

  public ComandoDesligar(IDispositivo dispositivo) {
    this.dispositivo = dispositivo;
  }

  @Override
  public void executar() {
    dispositivo.desligar();
  }
}
```

Para a cortina, criamos comandos concretos ComandoAbrirCortina e ComandoFecharCortina.

```
public class ComandoAbrirCortina implements IComando {
  private Cortina cortina;

  public ComandoAbrirCortina(Cortina cortina) {
    this.cortina = cortina;
  }

  @Override
  public void executar() {
    cortina.abrir();
```

```
  }
}


public class ComandoFecharCortina implements IComando {
  private Cortina cortina;

  public ComandoFecharCortina(Cortina cortina) {
    this.cortina = cortina;
  }

  @Override
  public void executar() {
    cortina.fechar();
  }
}
```

A interface IFabricaComando define métodos para criar os comandos, e cada fábrica concreta implementa essa interface para criar comandos específicos para os dispositivos.

```
public interface IFabricaComando {
  public IComando criarComandoLigar();
  public IComando criarComandoDesligar();
  public IComando criarComandoAbrir();
  public IComando criarComandoFechar();
}
```

Implementamos a fábrica de comandos para a luz com a classe FabricaComandoLuz.

```
public class FabricaComandoLuz implements IFabricaComando {
  private Luz luz;

  public FabricaComandoLuz(Luz luz) {
    this.luz = luz;
```

```
    }

    @Override
    public IComando criarComandoLigar() {
        return new ComandoLigar(luz);
    }

    @Override
    public IComando criarComandoDesligar() {
        return new ComandoDesligar(luz);
    }

    @Override
    public IComando criarComandoAbrir() {
        throw new UnsupportedOperationException("Comando abrir não suportado para Luz");
    }

    @Override
    public IComando criarComandoFechar() {
        throw new UnsupportedOperationException("Comando fechar não suportado para Luz");
    }
}
```

Para a cortina, implementamos a fábrica de comandos FabricaComandoCortina.

```
public class FabricaComandoCortina implements IFabricaComando {
    private Cortina cortina;

    public FabricaComandoCortina(Cortina cortina) {
        this.cortina = cortina;
    }

    @Override
```

```
  public IComando criarComandoLigar() {
    throw new UnsupportedOperationException("Comando ligar não suportado para
Cortina");
  }

  @Override
  public IComando criarComandoDesligar() {
    throw new UnsupportedOperationException("Comando desligar não suportado para
Cortina");
  }

  @Override
  public IComando criarComandoAbrir() {
    return new ComandoAbrirCortina(cortina);
  }

  @Override
  public IComando criarComandoFechar() {
    return new ComandoFecharCortina(cortina);
  }
}
```

A classe Controle gerencia a adição e execução dos comandos, facilitando a adição e remoção de comandos criados por múltiplas fábricas. Isso melhora a manutenção e a extensibilidade do aplicativo de automação residencial, permitindo fácil adição de novos dispositivos e ações.

```
public class Controle {
  private List<IComando> comandos = new ArrayList<>();

  public void adicionarComando(IComando comando) {
    comandos.add(comando);
  }
```

```
  public void executarComandos() {
    for (IComando comando : comandos) {
      comando.executar();
    }
    comandos.clear();
  }
}
```

Por fim, criamos a classe Principal para demonstrar o uso do sistema de automação residencial refatorado e no cenário de uso indicado. Para isso utilizaremos as fábricas de comandos e os dispositivos já implementados. Primeiramente, instanciamos os dispositivos Luz e Cortina para o ambiente da sala. Em seguida, criamos as fábricas de comando específicas para cada dispositivo FabricaComandoLuz e FabricaComandoCortina. Posteriormente, instanciamos a classe Controle, que gerencia a adição e execução dos comandos. Então, adicionamos os comandos necessários para assistir ao filme: desligar as luzes e fechar as cortinas. Finalmente, executamos os comandos adicionados ao controle, resultando na execução das ações de desligar a luz e fechar a cortina, simulando o cenário de assistir a um filme.

```
public class Principal {
  public static void main(String[] args) {
    Luz luzSala = new Luz("Sala");
    Cortina cortinaQuarto = new Cortina("Quarto");

    IFabricaComando fabricaLuz = new FabricaComandoLuz(luzSala);
    IFabricaComando fabricaCortina = new FabricaComandoCortina(cortinaQuarto);

    Controle controle = new Controle();

    controle.adicionarComando(fabricaLuz.criarComandoLigar());
    controle.adicionarComando(fabricaCortina.criarComandoAbrir());
```

```
        controle.executarComandos();
        controle.adicionarComando(fabricaCortina.criarComandoFechar());
        controle.adicionarComando(fabricaLuz.criarComandoDesligar());

        controle.executarComandos();
    }
}
```

A saída esperada após a execução do método main é a seguinte:

A luz na Sala está ligada.
A cortina na Quarto está aberto.
A cortina na Quarto está fechada.
A luz na Sala está desligada.

Solução do Exercício 93

Para resolver o problema seguindo o Princípio da Inversão de Dependência, começamos definindo uma abstração para os descontos e desacoplamos a lógica de desconto da classe Pedido.

Primeiro, definimos uma interface IDesconto com o método aplicarDesconto, que será implementado por várias classes de desconto concretas.

```
public interface IDesconto {
    void aplicarDesconto(Pedido pedido);
}
```

Em seguida, criamos as classes concretas de desconto: DescontoPorQuantidade, DescontoPorValorTotal e DescontoPorClienteFidelidade, todas implementando a interface IDesconto.

```
public class DescontoPorQuantidade implements IDesconto {
    @Override
    public void aplicarDesconto(Pedido pedido) {
        if (pedido.getQuantidade() > 5) {
```

```
            double desconto = pedido.getValorTotal() * 0.10;
            pedido.setValorTotal(pedido.getValorTotal() - desconto);
        }
    }
}

public class DescontoPorValorTotal implements IDesconto {
    @Override
    public void aplicarDesconto(Pedido pedido) {
        if (pedido.getValorTotal() > 500) {
            double desconto = pedido.getValorTotal() * 0.05;
            pedido.setValorTotal(pedido.getValorTotal() - desconto);
        }
    }
}

public class DescontoPorClienteFidelidade implements IDesconto {
    @Override
    public void aplicarDesconto(Pedido pedido) {
        if (pedido.getCliente().isFidelidade()) {
            double desconto = pedido.getValorTotal() * 0.025;
            pedido.setValorTotal(pedido.getValorTotal() - desconto);
        }
    }
}
```

Criamos uma classe DescontoService para gerenciar a aplicação de descontos, mantendo uma lista de estratégias de desconto e iterando sobre elas para aplicar os descontos apropriados.

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

```
public class DescontoService {
    private List<IDesconto> descontos;

    public DescontoService() {
        this.descontos = new ArrayList<>();
    }

    public void adicionarDesconto(IDesconto desconto) {
        descontos.add(desconto);
    }

    public void aplicarDescontos(Pedido pedido) {
        for (IDesconto desconto : descontos) {
            desconto.aplicarDesconto(pedido);
        }
    }
}
```

Atualizamos a classe Pedido para permitir a atualização do valor total após a aplicação do desconto e a inclusão de um cliente.

```
public class Pedido {
    private int quantidade;
    private double valorTotal;
    private Cliente cliente;

    public Pedido(int quantidade, double valorTotal, Cliente cliente) {
        this.quantidade = quantidade;
        this.valorTotal = valorTotal;
        this.cliente = cliente;
    }

    public int getQuantidade() {
```

```
        return quantidade;
    }

    public double getValorTotal() {
        return valorTotal;
    }

    public void setValorTotal(double valorTotal) {
        this.valorTotal = valorTotal;
    }

    public Cliente getCliente() {
        return cliente;
    }
}
```

A classe Cliente mantém informações sobre o nome do cliente e se ele é um cliente de fidelidade.

```
public class Cliente {
    private String nome;
    private boolean fidelidade;

    public Cliente(String nome, boolean fidelidade) {
        this.nome = nome;
        this.fidelidade = fidelidade;
    }

    public String getNome() {
        return nome;
    }

    public boolean isFidelidade() {
```

```
        return fidelidade;
    }
}
```

Finalmente, a classe principal Main demonstra o uso das diferentes estratégias de desconto, aplicando-as aos pedidos e exibindo os resultados no console.

```
public class Main {
    public static void main(String[] args) {
        Cliente joao = new Cliente("João", true);
        Pedido pedido1 = new Pedido(10, 1000.0, joao);
        Pedido pedido2 = new Pedido(2, 100.0, joao);

        DescontoService descontoService = new DescontoService();

        descontoService.adicionarDesconto(new DescontoPorQuantidade());
        descontoService.adicionarDesconto(new DescontoPorValorTotal());
        descontoService.adicionarDesconto(new DescontoPorClienteFidelidade());

        descontoService.aplicarDescontos(pedido1);
        System.out.println("Valor total do pedido1 após descontos: " + pedido1.getValorTotal());

        descontoService.aplicarDescontos(pedido2);
        System.out.println("Valor total do pedido2 após descontos: " + pedido2.getValorTotal());
    }
}
```

Ao executar a aplicação, veremos os descontos aplicados corretamente no console:

```
Valor total do pedido1 após descontos: 833.625
Valor total do pedido2 após descontos: 97.5
```

Solução do Exercício 94

Para resolver o problema de implementar um protótipo que realiza o gerenciamento de imagens utilizando o padrão de projeto Proxy para adicionar funcionalidade de cache ao

carregamento de imagens, começamos criando a classe Imagem. Esta classe representa a imagem original que será carregada do disco.

```
public class Imagem {
  private String nomeArquivo;

  public Imagem(String nomeArquivo) {
    this.nomeArquivo = nomeArquivo;
    carregarImagemDoDisco();
  }

  public void exibirImagem() {
    System.out.println("Exibindo " + nomeArquivo);
  }

  private void carregarImagemDoDisco() {
    System.out.println("Carregando " + nomeArquivo);
  }
}
```

Para utilizar a bilbioteca Caffeine, inclua o seguinte trecho no arquivo pom.xml do seu projeto Maven.

```
<dependency>
  <groupId>com.github.ben-manes.caffeine</groupId>
  <artifactId>caffeine</artifactId>
  <version>2.9.2</version>
</dependency>
```

Em seguida, criamos a classe ImagemProxy, que atua como um intermediário para controlar o acesso ao objeto Imagem. Utilizamos a biblioteca Caffeine para implementar o cache. O proxy verifica se a imagem já está no cache e, se não estiver, carrega-a do disco e a armazena no cache.

```
import com.github.benmanes.caffeine.cache.Cache;
```

```
import com.github.benmanes.caffeine.cache.Caffeine;

public class ImagemProxy {
  private Cache<String, Imagem> cache;

  public ImagemProxy() {
    cache = Caffeine.newBuilder()
      .maximumSize(10)
      .build();
  }

  public void exibirImagem(String nomeArquivo) {
    Imagem imagem = cache.get(nomeArquivo, k -> new Imagem(k));
    imagem.exibirImagem();
  }
}
```

Na classe Principal, criamos uma instância de ImagemProxy e a utilizamos para exibir imagens. Quando uma imagem é solicitada pela primeira vez, ela é carregada do disco e armazenada em cache. Nas solicitações subsequentes para a mesma imagem, o proxy a recupera do cache, evitando a necessidade de carregá-la do disco novamente. Isso melhora a eficiência do sistema e demonstra a aplicação do padrão Proxy para adicionar funcionalidade de cache.

```
public class Principal {
  public static void main(String[] args) {
    ImagemProxy imagemProxy = new ImagemProxy();

    imagemProxy.exibirImagem("Imagem 1"); // Carrega a imagem 1 do disco e exibe
    imagemProxy.exibirImagem("Imagem 1"); // Não precisa carregar a imagem 1 do disco novamente, apenas exibe

    imagemProxy.exibirImagem("Imagem 2"); // Carrega a imagem 2 do disco e exibe
```

```
        imagemProxy.exibirImagem("Imagem 2"); // Não precisa carregar a imagem 2 do disco novamente, apenas exibe
    }
}
```

Espera-se como saída no console o seguinte:

```
Carregando Imagem 1
Exibindo Imagem 1
Exibindo Imagem 1
Carregando Imagem 2
Exibindo Imagem 2
Exibindo Imagem 2
```

Solução do Exercício 95

Para resolver o problema de implementar um módulo de gerenciamento de usuários utilizando o padrão de projeto State, começamos pela criação da classe Usuario. Esta classe representa um usuário do sistema, contendo informações como nome, tipo, senha, estado da conta, número de advertências e histórico de ações realizadas. A classe também define métodos para ativar, desativar, advertir e desbanir um usuário, delegando a lógica de estado para a classe apropriada que implementa o comportamento específico do estado atual do usuário.

```
import java.util.List;
import java.util.ArrayList;
import java.time.LocalDateTime;

public class Usuario {
    private LocalDateTime dataEHoraLocalDeCriacao;
    private String nomeDoUsuario;
    private String tipo;
    private String senha;
    private EstadoDaConta estadoDaConta;
    private int numeroDeAdvertencias;
```

```
private List<HistoricoDeAcao> historicoDeAcao;

public Usuario(String nomeDoUsuario, String tipo, String senha) {
    this.dataEHoraLocalDeCriacao = LocalDateTime.now();
    this.nomeDoUsuario = nomeDoUsuario;
    this.tipo = tipo;
    this.senha = senha;
    this.estadoDaConta = new Novo();
    this.numeroDeAdvertencias = 0;
    this.historicoDeAcao = new ArrayList<>();
}

public String getTipo() {
    return tipo;
}

public void setEstadoDaConta(EstadoDaConta estadoDaConta) {
    this.estadoDaConta = estadoDaConta;
}

public EstadoDaConta getEstadoDaConta() {
    return estadoDaConta;
}

public LocalDateTime getDataEHoraLocalDeCriacao() {
    return dataEHoraLocalDeCriacao;
}

public void setDataEHoraLocalDeCriacao(LocalDateTime dataEHoraLocalDeCriacao) {
    this.dataEHoraLocalDeCriacao = dataEHoraLocalDeCriacao;
}
```

```
  public List<HistoricoDeAcao> getHistoricoDeAcao() {
     return historicoDeAcao;
  }

  public void addHistoricoAcao(Usuario usuario, EstadoDaConta estadoDaConta, Usuario
administrador, String acao){
     this.historicoDeAcao.add(new HistoricoDeAcao(usuario, estadoDaConta, administrador,
acao));
  }

  public int getNumeroDeAdvertencias() {
     return numeroDeAdvertencias;
  }

  public String getNomeDoUsuario() {
     return nomeDoUsuario;
  }

  public void setNumeroDeAdvertencias(int numeroDeAdvertencias) {
     this.numeroDeAdvertencias = numeroDeAdvertencias;
  }

  public void ativar(Usuario administrador) {
     this.estadoDaConta.ativar(this, administrador);
  }

  public void desativar(Usuario administrador) {
     this.estadoDaConta.desativar(this, administrador);
  }

  public void advertir(Usuario administrador) {
     if ("ADMINISTRADOR".equals(this.tipo)) {
```

```
            throw new IllegalStateException("Não é possível advertir um usuário administrador");
        }
        this.estadoDaConta.advertir(this, administrador);
    }

    public void desbanir() {
        this.estadoDaConta.desbanir(this);
    }
}
```

Em seguida, criamos a classe abstrata EstadoDaConta, que define os métodos abstratos que cada estado específico deve implementar. Cada estado específico (Novo, Ativo, Desativado, BanidoPor7Segundos, BanidoPor30Segundos, BanidoDefinitivamente) estende EstadoDaConta e implementa o comportamento apropriado para cada operação permitida.

```
public abstract class EstadoDaConta {
    public abstract void ativar(Usuario usuario, Usuario administrador);
    public abstract void desativar(Usuario usuario, Usuario administrador);
    public abstract void advertir(Usuario usuario, Usuario administrador);
    public abstract void desbanir(Usuario usuario);
}
```

A classe Novo representa o estado inicial de um usuário. Neste estado, apenas a ativação é permitida.

```
public class Novo extends EstadoDaConta {
    @Override
    public void ativar(Usuario usuario, Usuario administrador) {
        usuario.setEstadoDaConta(new Ativo());
        usuario.addHistoricoAcao(usuario, usuario.getEstadoDaConta(), administrador, "ativar");
    }

    @Override
    public void desativar(Usuario usuario, Usuario administrador) {
```

```
        throw new IllegalStateException("Não é possível desativar um usuário novo");
    }

    @Override
    public void advertir(Usuario usuario, Usuario administrador) {
        throw new IllegalStateException("Não é possível advertir um usuário novo");
    }

    @Override
    public void desbanir(Usuario usuario) {
        throw new IllegalStateException("Não é possível desbanir um usuário novo");
    }
}
```

A classe Ativo permite desativar e advertir o usuário, incrementando o número de advertências e mudando o estado conforme necessário.

```
public class Ativo extends EstadoDaConta {
    @Override
    public void ativar(Usuario usuario, Usuario administrador) {
        throw new IllegalStateException("O usuário já está ativo");
    }

    @Override
    public void desativar(Usuario usuario, Usuario administrador) {
        usuario.setEstadoDaConta(new Desativado());
        usuario.addHistoricoAcao(usuario, usuario.getEstadoDaConta(), administrador,
"desativar");
    }

    @Override
    public void advertir(Usuario usuario, Usuario administrador) {
        usuario.setNumeroDeAdvertencias(usuario.getNumeroDeAdvertencias() + 1);
```

```
        if (usuario.getNumeroDeAdvertencias() == 1) {
            usuario.setEstadoDaConta(new BanidoPor7Segundos());
        } else if (usuario.getNumeroDeAdvertencias() == 2) {
            usuario.setEstadoDaConta(new BanidoPor30Segundos());
        } else {
            usuario.setEstadoDaConta(new BanidoDefinitivamente());
        }
        usuario.addHistoricoAcao(usuario, usuario.getEstadoDaConta(), administrador,
"advertir");
    }

    @Override
    public void desbanir(Usuario usuario) {
        throw new IllegalStateException("Não é possível desbanir um usuário ativo");
    }
}
```

A classe Desativado permite ativar o usuário novamente, enquanto as classes BanidoPor7Segundos, BanidoPor30Segundos e BanidoDefinitivamente gerenciam os estados de banimento, com durações específicas e restrições adequadas.

```
public class Desativado extends EstadoDaConta {
    @Override
    public void ativar(Usuario usuario, Usuario administrador) {
        usuario.setEstadoDaConta(new Ativo());
        usuario.addHistoricoAcao(usuario, usuario.getEstadoDaConta(), administrador, "ativar");
    }

    @Override
    public void desativar(Usuario usuario, Usuario administrador) {
        throw new IllegalStateException("O usuário já está desativado");
    }
```

```
  @Override
  public void advertir(Usuario usuario, Usuario administrador) {
    throw new IllegalStateException("Não é possível advertir um usuário desativado");
  }

  @Override
  public void desbanir(Usuario usuario) {
    throw new IllegalStateException("Não é possível desbanir um usuário desativado");
  }
}
```

A classe abstrata Banido define métodos que podem ser executado para um Usuario.

```
import java.time.Duration;

public abstract class Banido extends EstadoDaConta {
  protected final Duration duracaoDoBanimento;

  public Banido(Duration duracaoDoBanimento) {
    this.duracaoDoBanimento = duracaoDoBanimento;
  }

  @Override
  public void ativar(Usuario usuario, Usuario administrador) {
    throw new IllegalStateException("Não é possível ativar um usuário banido");
  }

  @Override
  public void desativar(Usuario usuario, Usuario administrador) {
    throw new IllegalStateException("Não é possível desativar um usuário banido");
  }
```

```
  @Override
  public void advertir(Usuario usuario, Usuario administrador) {
    throw new IllegalStateException("Não é possível advertir um usuário banido");
  }

  @Override
  public void desbanir(Usuario usuario) {
    if (Duration.between(usuario.getDataEHoraLocalDeCriacao(),
LocalDateTime.now()).compareTo(duracaoDoBanimento) >= 0) {
      usuario.setEstadoDaConta(new Ativo());
      usuario.addHistoricoAcao(usuario, usuario.getEstadoDaConta(), null, "desbanir");
    } else {
      throw new IllegalStateException("O usuário ainda está banido pelo tempo
especificado");
    }
  }
}
```

A classe concreta BanidoPor7Segundos estende a classe Banido e sobrescreve o método desbanir.

```
import java.time.Duration;

public class BanidoPor7Segundos extends Banido {
  public BanidoPor7Segundos() {
    super(Duration.ofSeconds(7));
  }

  @Override
  public void desbanir(Usuario usuario) {
    if (Duration.between(usuario.getDataEHoraLocalDeCriacao(),
LocalDateTime.now()).compareTo(super.duracaoDoBanimento) >= 0) {
      usuario.setEstadoDaConta(new Ativo());
```

```
            usuario.addHistoricoAcao(usuario, usuario.getEstadoDaConta(), null, "desbanir");
        } else {
            throw new IllegalStateException("O usuário ainda está banido por 7 segundos");
        }
    }
}
```

A classe concreta BanidoPor30Segundos estende a classe Banido e sobrescreve o método desbanir.

```
import java.time.Duration;
import java.time.LocalDateTime;

public class BanidoPor30Segundos extends Banido {
    public BanidoPor30Segundos() {
        super(Duration.ofSeconds(30));
    }

    @Override
    public void desbanir(Usuario usuario) {
        if (Duration.between(usuario.getDataEHoraLocalDeCriacao(),
LocalDateTime.now()).compareTo(super.duracaoDoBanimento) >= 0) {
            usuario.setEstadoDaConta(new Ativo());
            usuario.addHistoricoAcao(usuario, usuario.getEstadoDaConta(), null, "desbanir");
        } else {
            throw new IllegalStateException("O usuário ainda está banido por 30 segundos");
        }
    }
}
```

A classe concreta BanidoDefinitivamente estende a classe Banido e sobrescreve o método desbanir.

```
import java.time.Duration;
```

```
public class BanidoDefinitivamente extends Banido {
  public BanidoDefinitivamente() {
    super(Duration.ofDays(Long.MAX_VALUE));
  }

  @Override
  public void desbanir(Usuario usuario) {
    throw new IllegalStateException("O usuário está banido definitivamente e não pode ser desbanido");
  }
}
```

Também criamos a classe HistoricoDeAcao para registrar as ações realizadas pelos administradores sobre os usuários, armazenando informações relevantes como o usuário, o estado da conta, o administrador que realizou a ação e a data e hora da ação.

```
import java.time.LocalDateTime;

public class HistoricoDeAcao {
  private Usuario usuario;
  private EstadoDaConta estadoDaConta;
  private Usuario administrador;
  private String acao;
  private LocalDateTime dataEHoraDaAcao;

  public HistoricoDeAcao(Usuario usuario, EstadoDaConta estadoDaConta, Usuario administrador, String acao) {
    this.usuario = usuario;
    this.estadoDaConta = estadoDaConta;
    this.administrador = administrador;
    this.acao = acao;
    this.dataEHoraDaAcao = LocalDateTime.now();
```

```
    }

    public Usuario getUsuario() {
        return usuario;
    }

    public EstadoDaConta getEstadoDaConta() {
        return estadoDaConta;
    }

    public String getAcao() {
        return acao;
    }
}
```

Agora, adicione a seguinte dependência da biblioteca Jackson no seu arquivo de configuração de projeto Maven, pom.xml.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.11.3</version>
</dependency>
```

A classe HistoricoDeAcaoService é responsável por gerenciar o histórico de ações realizadas no sistema. Ela mantém um registro em memória das ações mais recentes e também armazena essas ações em um arquivo JSON quando o limite de histórico em memória é excedido. Para isso, ela utiliza a biblioteca Jackson para serialização e desserialização de objetos JSON.

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;
import java.io.File;
```

```
import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class HistoricoDeAcaoService {
  private static final int MAX_HISTORICO_EM_MEMORIA = 10;
  private static final String NOME_ARQUIVO_HISTORICO =
"src/main/resources/historico.json";

  private final LinkedList<HistoricoDeAcao> historicoEmMemoria = new LinkedList<>();
  private final ObjectMapper mapper = new ObjectMapper();

  public void adicionarAcao(HistoricoDeAcao acao) {
    historicoEmMemoria.addFirst(acao);
    if (historicoEmMemoria.size() > MAX_HISTORICO_EM_MEMORIA) {
      HistoricoDeAcao acaoASerRemovida = historicoEmMemoria.removeLast();
      try {
        salvarAcaoEmArquivo(acaoASerRemovida);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

  public List<HistoricoDeAcao> obterHistorico() {
    List<HistoricoDeAcao> acoes = new ArrayList<>(historicoEmMemoria);
    try {
      acoes.addAll(carregarAcoesDoArquivo());
    } catch (IOException e) {
      e.printStackTrace();
    }
    return acoes;
  }
```

```
  private void salvarAcaoEmArquivo(HistoricoDeAcao acao) throws IOException {
     File arquivoHistorico = new File(NOME_ARQUIVO_HISTORICO);
     List<HistoricoDeAcao> acoes = carregarAcoesDoArquivo();
     acoes.add(acao);
     mapper.writeValue(arquivoHistorico, acoes);
  }

  private List<HistoricoDeAcao> carregarAcoesDoArquivo() throws IOException {
     File arquivoHistorico = new File(NOME_ARQUIVO_HISTORICO);
     if (!arquivoHistorico.exists()) {
        return new ArrayList<>();
     }
     return mapper.readValue(arquivoHistorico,
mapper.getTypeFactory().constructCollectionType(List.class, HistoricoDeAcao.class));
  }
}
```

Na classe Principal é demonstrado o uso da solução protótipo. Ela cria instâncias de usuários e realiza operações de ativação, advertência e desbanimento, demonstrando a funcionalidade do sistema. Também utiliza a HistoricoDeAcaoService para adicionar e obter o histórico de ações.

```
public class Principal {
  public static void main(String[] args) {
     Usuario admin = new Usuario("Admin", "ADMINISTRADOR", "admin123");
     Usuario user1 = new Usuario("User1", "COMUM", "user123");
     Usuario user2 = new Usuario("User2", "COMUM", "user123");

     // Exemplo de operações
     admin.ativar(admin); // Administrador se ativa
     user1.ativar(admin); // Administrador ativa user1
     user2.ativar(admin); // Administrador ativa user2
```

```
        // Administrador adverte user1
        user1.advertir(admin);
        System.out.println("User1 estado: " +
user1.getEstadoDaConta().getClass().getSimpleName());

        // Simula o tempo para o banimento de 7 segundos expirar
        user1.setDataEHoraLocalDeCriacao(LocalDateTime.now().minusSeconds(8));
        user1.desbanir();
        System.out.println("User1 estado após desbanimento: " +
user1.getEstadoDaConta().getClass().getSimpleName());

        // Adiciona histórico de ações
        HistoricoDeAcaoService historicoService = new HistoricoDeAcaoService();
        historicoService.adicionarAcao(new HistoricoDeAcao(user1, user1.getEstadoDaConta(),
admin, "ativar"));
        historicoService.adicionarAcao(new HistoricoDeAcao(user1, user1.getEstadoDaConta(),
admin, "advertir"));

        // Obtém e exibe o histórico
        for (HistoricoDeAcao acao : historicoService.obterHistorico()) {
            System.out.println("Ação: " + acao.getAcao() + ", Usuário: " +
acao.getUsuario().getNomeDoUsuario() + ", Estado: " +
acao.getEstadoDaConta().getClass().getSimpleName());
        }
    }
}
```

Espera-se como saída no console o seguinte:

```
User1 estado: BanidoPor7Segundos
User1 estado após desbanimento: Ativo
Ação: advertir, Usuário: User1, Estado: Ativo
Ação: ativar, Usuário: User1, Estado: Ativo
```

Solução do Exercício 96

Para resolver o problema de gerenciamento de pedidos de cestas básicas utilizando o padrão de projeto Prototype, foi desenvolvida uma solução que permite a clonagem e modificação dos pedidos mensais. A implementação envolve várias classes para representar os pedidos, itens e a funcionalidade de clonagem.

Primeiramente, a classe abstrata PedidoPrototype define a estrutura para os pedidos e inclui métodos para clonagem e manipulação dos itens do pedido. Esta classe contém informações sobre os itens, tipo de cliente, data e hora do pedido, peso e distância para entrega. O construtor padrão inicializa esses campos e o construtor de cópia cria um novo pedido copiando os detalhes do pedido original.

```
import java.util.ArrayList;
import java.util.List;
import java.time.LocalDateTime;

public abstract class PedidoPrototype {
  protected List<Item> itens;
  protected String tipoCliente;
  protected LocalDateTime dataHoraPedido;
  protected double taxaEntrega;
  protected double peso;
  protected double distancia;

  public PedidoPrototype(String tipoCliente, LocalDateTime dataHoraPedido, double peso,
double distancia) {
    this.itens = new ArrayList<>();
    this.tipoCliente = tipoCliente;
    this.dataHoraPedido = dataHoraPedido;
    this.peso = peso;
    this.distancia = distancia;
  }
```

```
public PedidoPrototype(PedidoPrototype pedidoPrototype) {
    this.itens = new ArrayList<>();
    for (Item item : pedidoPrototype.itens) {
        this.itens.add(new Item(item));
    }
    this.tipoCliente = pedidoPrototype.tipoCliente;
    this.dataHoraPedido = pedidoPrototype.dataHoraPedido;
    this.taxaEntrega = pedidoPrototype.taxaEntrega;
    this.peso = pedidoPrototype.peso;
    this.distancia = pedidoPrototype.distancia;
}

public abstract PedidoPrototype clonar();

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

public double calcularTotalPedido() {
    double totalItens = 0;
    for (Item item : itens) {
        totalItens += item.getTotal();
    }
    return totalItens;
}

public List<Item> getItens() {
    return itens;
}

public void setDataHoraPedido(LocalDateTime dataHoraPedido) {
    this.dataHoraPedido = dataHoraPedido;
```

```
    }

    public void exibirDetalhesPedido() {
        for (Item item : itens) {
            System.out.println(item.getNome() + ": " + item.getQuantidade() + " unidades a R$ " +
item.getPreco() + " cada, Total: R$ " + item.getTotal());
        }
        System.out.println("Total do Pedido: R$ " + calcularTotalPedido());
    }
    // Getters e setters adicionais
}
```

Em seguida, a implementação concreta do protótipo é realizada pela classe PedidoConcreto, que permite a clonagem específica dos pedidos. Esta classe herda de PedidoPrototype e implementa o método clonar para criar uma cópia do pedido atual. Além disso, adicionamos o método substituirItens, que pertence logicamente à classe PedidoConcreto, para realizar a substituição dos itens no pedido clonado.

```
import java.time.LocalDateTime;
import java.util.Random;

public class PedidoConcreto extends PedidoPrototype {
    public PedidoConcreto(String tipoCliente, LocalDateTime dataHoraPedido, double peso,
double distancia) {
        super(tipoCliente, dataHoraPedido, peso, distancia);
    }

    public PedidoConcreto(PedidoPrototype pedidoPrototype) {
        super(pedidoPrototype);
    }

    @Override
    public PedidoPrototype clonar() {
```

```
        return new PedidoConcreto(this);
    }

    public void substituirItens() {
        String[] produtosOrigemAnimal = {"Carne", "Leite"};
        String[] graos = {"Arroz", "Feijão", "Farinha de Trigo"};
        String[] produtosIndustrializados = {"Café", "Óleo", "Manteiga", "Açúcar", "Pão"};
        String[] legumesEFrutas = {"Batata", "Tomate", "Banana"};

        Random random = new Random();
        for (int i = 0; i < 6; i++) {
            int itemIndex = random.nextInt(getItens().size());
            Item item = getItens().get(itemIndex);
            String novoNome = getNovoNome(item.getNome(), produtosOrigemAnimal, graos,
produtosIndustrializados, legumesEFrutas);
            if (novoNome != null) {
                item.setNome(novoNome);
            }
        }
    }

    private String getNovoNome(String nomeAtual, String[]... grupos) {
        Random random = new Random();
        for (String[] grupo : grupos) {
            for (String item : grupo) {
                if (item.equals(nomeAtual)) {
                    int index;
                    do {
                        index = random.nextInt(grupo.length);
                    } while (grupo[index].equals(nomeAtual));
                    return grupo[index];
                }
```

```
        }
      }
      return null;
  }
  // Getters e setters adicionais, se necessário
}
```

A classe Item é responsável por manter as informações sobre cada produto na cesta. Cada item tem um nome, quantidade e preço, além de um construtor de cópia para facilitar a clonagem dos itens.

```
public class Item {
  private String nome;
  private double quantidade;
  private double preco;

  public Item(String nome, double quantidade, double preco) {
    this.nome = nome;
    this.quantidade = quantidade;
    this.preco = preco;
  }

  public Item(Item item) {
    this.nome = item.nome;
    this.quantidade = item.quantidade;
    this.preco = item.preco;
  }

  public double getTotal() {
    return quantidade * preco;
  }

  public String getNome() {
```

```
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public double getQuantidade() {
        return quantidade;
    }

    public double getPreco() {
        return preco;
    }
    // Getters e setters adicionais
}
```

Na classe Principal, inicializa o pedido original, clona o pedido para o próximo mês, realiza substituições de itens e exibe os detalhes dos pedidos original e clonado.

```
import java.time.LocalDateTime;

public class Principal {
    public static void main(String[] args) {
        // Criação do pedido original
        PedidoPrototype pedidoOriginal = new PedidoConcreto("Cliente A", LocalDateTime.now(),
10.0, 5.0);
        pedidoOriginal.adicionarItem(new Item("Carne", 6, 40.0));
        pedidoOriginal.adicionarItem(new Item("Leite", 7, 3.0));
        pedidoOriginal.adicionarItem(new Item("Arroz", 3, 5.0));
        pedidoOriginal.adicionarItem(new Item("Feijão", 4.5, 6.0));
        pedidoOriginal.adicionarItem(new Item("Farinha de Trigo", 1.5, 2.0));
        pedidoOriginal.adicionarItem(new Item("Café", 0.6, 20.0));
```

```
        pedidoOriginal.adicionarItem(new Item("Óleo", 0.9, 5.0));
        pedidoOriginal.adicionarItem(new Item("Manteiga", 0.75, 10.0));
        pedidoOriginal.adicionarItem(new Item("Açúcar", 3, 4.0));
        pedidoOriginal.adicionarItem(new Item("Pão", 6, 1.0));
        pedidoOriginal.adicionarItem(new Item("Batata", 6, 2.0));
        pedidoOriginal.adicionarItem(new Item("Tomate", 9, 3.0));
        pedidoOriginal.adicionarItem(new Item("Banana", 7.5, 1.5));

        // Clonagem do pedido original para criar um novo pedido para o próximo mês
        PedidoPrototype pedidoClone = pedidoOriginal.clonar();
        pedidoClone.setDataHoraPedido(LocalDateTime.now().plusMonths(1));

        // Substituição de até 6 itens da cesta com itens do mesmo grupo
        ((PedidoConcreto) pedidoClone).substituirItens();

        // Exibir detalhes dos pedidos
        System.out.println("Pedido Original:");
        pedidoOriginal.exibirDetalhesPedido();

        System.out.println("Pedido Clonado:");
        pedidoClone.exibirDetalhesPedido();
    }
}
```

A saída esperada após a execução do método main é a seguinte:

```
Pedido Original:
Carne: 6.0 unidades a R$ 40.0 cada, Total: R$ 240.0
Leite: 7.0 unidades a R$ 3.0 cada, Total: R$ 21.0
Arroz: 3.0 unidades a R$ 5.0 cada, Total: R$ 15.0
Feijão: 4.5 unidades a R$ 6.0 cada, Total: R$ 27.0
Farinha de Trigo: 1.5 unidades a R$ 2.0 cada, Total: R$ 3.0
Café: 0.6 unidades a R$ 20.0 cada, Total: R$ 12.0
```

Óleo: 0.9 unidades a R$ 5.0 cada, Total: R$ 4.5
Manteiga: 0.75 unidades a R$ 10.0 cada, Total: R$ 7.5
Açúcar: 3.0 unidades a R$ 4.0 cada, Total: R$ 12.0
Pão: 6.0 unidades a R$ 1.0 cada, Total: R$ 6.0
Batata: 6.0 unidades a R$ 2.0 cada, Total: R$ 12.0
Tomate: 9.0 unidades a R$ 3.0 cada, Total: R$ 27.0
Banana: 7.5 unidades a R$ 1.5 cada, Total: R$ 11.25
Total do Pedido: R$ 398.25
Pedido Clonado:
Leite: 6.0 unidades a R$ 40.0 cada, Total: R$ 240.0
Leite: 7.0 unidades a R$ 3.0 cada, Total: R$ 21.0
Arroz: 3.0 unidades a R$ 5.0 cada, Total: R$ 15.0
Arroz: 4.5 unidades a R$ 6.0 cada, Total: R$ 27.0
Farinha de Trigo: 1.5 unidades a R$ 2.0 cada, Total: R$ 3.0
Café: 0.6 unidades a R$ 20.0 cada, Total: R$ 12.0
Manteiga: 0.9 unidades a R$ 5.0 cada, Total: R$ 4.5
Açúcar: 0.75 unidades a R$ 10.0 cada, Total: R$ 7.5
Açúcar: 3.0 unidades a R$ 4.0 cada, Total: R$ 12.0
Pão: 6.0 unidades a R$ 1.0 cada, Total: R$ 6.0
Batata: 6.0 unidades a R$ 2.0 cada, Total: R$ 12.0
Tomate: 9.0 unidades a R$ 3.0 cada, Total: R$ 27.0
Banana: 7.5 unidades a R$ 1.5 cada, Total: R$ 11.25
Total do Pedido: R$ 398.25

Solução do Exercício 97

Para resolver o problema de gerenciamento de estratégias de cobrança para serviços de consultoria utilizando o padrão de projeto Strategy, começamos definindo uma interface IEstrategiaCobranca que será implementada por diferentes classes de cobrança. Essa interface declara o método calcularCobranca que será implementado de maneira específica por cada estratégia de cobrança.

```
public interface IEstrategiaCobranca {
  double calcularCobranca(double tempoOuProjeto, double taxa);
```

```
}
```

Em seguida, criamos a classe CobrancaPorMinutagemImpl, que implementa a interface IEstrategiaCobranca. Esta classe define a lógica para calcular a cobrança baseada na minutagem, ou seja, o valor é calculado multiplicando os minutos trabalhados pela taxa por minuto.

```
public class CobrancaPorMinutagemImpl implements IEstrategiaCobranca {
  @Override
  public double calcularCobranca(double minutos, double taxaPorMinuto) {
     return minutos * taxaPorMinuto;
  }
}
```

Em seguida, temos a classe CobrancaPorHoraImpl, que também implementa a interface IEstrategiaCobranca. A lógica aqui é calcular a cobrança baseada em horas, multiplicando o número de horas trabalhadas pela taxa horária.

```
public class CobrancaPorHoraImpl implements IEstrategiaCobranca {
  @Override
  public double calcularCobranca(double horas, double taxaPorHora) {
     return horas * taxaPorHora;
  }
}
```

Para finalizar as classes de estratégias, criamos CobrancaPorProjetoImpl, que implementa a interface IEstrategiaCobranca. Esta classe calcula a cobrança com base em um valor fixo por projeto, independentemente do tempo gasto.

```
public class CobrancaPorProjetoImpl implements IEstrategiaCobranca {
  @Override
  public double calcularCobranca(double valorProjeto, double taxa) {
     return valorProjeto;
  }
}
```

Agora, para gerenciar essas estratégias de cobrança, criamos a classe de serviço CobrancaService. Esta classe possui um campo para a estratégia de cobrança atual e métodos para definir a estratégia e calcular a cobrança. O método setEstrategiaCobranca permite a alteração da estratégia em tempo de execução, enquanto o método cobrar aplica a estratégia definida.

```
public class CobrancaService {
  private IEstrategiaCobranca estrategiaCobranca;

  public void setEstrategiaCobranca(IEstrategiaCobranca estrategiaCobranca) {
    this.estrategiaCobranca = estrategiaCobranca;
  }

  public double cobrar(double tempoOuProjeto, double taxa) {
    if (estrategiaCobranca == null) {
      throw new IllegalStateException("Estrategia de cobrança não definida");
    }
    return estrategiaCobranca.calcularCobranca(tempoOuProjeto, taxa);
  }
}
```

Na classe principal, demonstramos o uso da solução alternando entre todas as estratégias de cobrança. Criamos uma instância de CobrancaService, definimos diferentes estratégias e calculamos a cobrança para cada uma delas, exibindo os resultados.

```
public class Principal {
  public static void main(String[] args) {
    CobrancaService cobrancaService = new CobrancaService();

    // Usando a estratégia de cobrança por minutagem
    cobrancaService.setEstrategiaCobranca(new CobrancaPorMinutagemImpl());
    double valorCobrancaMinutagem = cobrancaService.cobrar(120, 2.0); // 120 minutos,
R$2.0 por minuto
    System.out.println("Cobrança por Minutagem: R$ " + valorCobrancaMinutagem);
```

```
        // Usando a estratégia de cobrança por hora
        cobrancaService.setEstrategiaCobranca(new CobrancaPorHoraImpl());
        double valorCobrancaHora = cobrancaService.cobrar(2, 50.0); // 2 horas, R$50.0 por hora
        System.out.println("Cobrança por Hora: R$ " + valorCobrancaHora);

        // Usando a estratégia de cobrança por projeto
        cobrancaService.setEstrategiaCobranca(new CobrancaPorProjetoImpl());
        double valorCobrancaProjeto = cobrancaService.cobrar(1000, 0); // Projeto de valor fixo
R$1000
        System.out.println("Cobrança por Projeto: R$ " + valorCobrancaProjeto);
    }
}
```

A saída esperada após a execução do método main é a seguinte:

```
Cobrança por Minutagem: R$ 240.0
Cobrança por Hora: R$ 100.0
Cobrança por Projeto: R$ 1000.0
```

Solução do Exercício 98

Para começar, vamos definir a interface ValidadorPerfil. Esta interface contém um método chamado validar que aceita um Map<String, String> representando o perfil do usuário e retorna uma lista de mensagens de erro indicando quais validações falharam.

```
import java.util.List;
import java.util.Map;

public interface IValidadorPerfil {
    List<String> validar(Map<String, String> perfil);
}
```

Em seguida, implementamos a classe ValidadorPerfilBasico que implementa a interface ValidadorPerfil. Esta classe valida perfis básicos, verificando se os campos "nome" e "email"

estão presentes. Além disso, valida se o email está em um formato correto e se o nome contém pelo menos 3 caracteres e apenas caracteres alfabéticos.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class ValidadorPerfilBasico implements IValidadorPerfil {
  @Override
  public List<String> validar(Map<String, String> perfil) {
    List<String> mensagensErro = new ArrayList<>();
    validarCamposObrigatorios(perfil, mensagensErro);
    validarEmail(perfil, mensagensErro);
    validarNome(perfil, mensagensErro);
    return mensagensErro;
  }

  private void validarCamposObrigatorios(Map<String, String> perfil, List<String>
mensagensErro) {
    if (!perfil.containsKey("nome") || !perfil.containsKey("email")) {
      mensagensErro.add("Campos obrigatórios ausentes: nome e/ou email.");
    }
  }

  private void validarEmail(Map<String, String> perfil, List<String> mensagensErro) {
    String email = perfil.get("email");
    if (email == null || !email.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
      mensagensErro.add("Email inválido.");
    }
  }

  private void validarNome(Map<String, String> perfil, List<String> mensagensErro) {
    String nome = perfil.get("nome");
```

```
    if (nome == null || nome.length() < 3 || !nome.matches("^[\\p{L} ]+$")) {
      mensagensErro.add("Nome inválido. Deve conter pelo menos 3 caracteres
alfabéticos.");
    }
  }
}
```

Depois, criamos a classe ValidadorPerfilPremium que também implementa a interface ValidadorPerfil. Esta classe valida perfis premium, verificando se os campos "nome", "email", "telefone" e "endereco" estão presentes. Adicionalmente, valida se o email está em um formato correto, se o telefone contém exatamente 9 dígitos numéricos e se o endereço contém pelo menos 10 caracteres.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class ValidadorPerfilPremium implements IValidadorPerfil {
  @Override
  public List<String> validar(Map<String, String> perfil) {
    List<String> mensagensErro = new ArrayList<>();
    validarCamposObrigatorios(perfil, mensagensErro);
    validarEmail(perfil, mensagensErro);
    validarTelefone(perfil, mensagensErro);
    validarEndereco(perfil, mensagensErro);
    return mensagensErro;
  }

  private void validarCamposObrigatorios(Map<String, String> perfil, List<String>
mensagensErro) {
    if (!perfil.containsKey("nome") || !perfil.containsKey("email") ||
!perfil.containsKey("telefone") || !perfil.containsKey("endereco")) {
```

```
            mensagensErro.add("Campos obrigatórios ausentes: nome, email, telefone e/ou
endereco.");
        }
    }

    private void validarEmail(Map<String, String> perfil, List<String> mensagensErro) {
        String email = perfil.get("email");
        if (email == null || !email.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
            mensagensErro.add("Email inválido.");
        }
    }

    private void validarTelefone(Map<String, String> perfil, List<String> mensagensErro) {
        String telefone = perfil.get("telefone");
        if (telefone == null || !telefone.matches("\\d{9}")) {
            mensagensErro.add("Telefone inválido. Deve conter exatamente 9 dígitos.");
        }
    }

    private void validarEndereco(Map<String, String> perfil, List<String> mensagensErro) {
        String endereco = perfil.get("endereco");
        if (endereco == null || endereco.length() < 10) {
            mensagensErro.add("Endereço inválido. Deve conter pelo menos 10 caracteres.");
        }
    }
}
```

A seguir, temos a classe ValidacaoPerfilService. Esta classe utiliza uma instância de ValidadorPerfil para realizar a validação dos perfis de usuários. A classe contém métodos para definir o validador a ser usado, validar um perfil e processar um perfil, exibindo uma lista de mensagens de erro específicas se o perfil for inválido.

```
import java.util.List;
```

```
import java.util.Map;

public class ValidacaoPerfilService {
  private IValidadorPerfil validador;

  public ValidacaoPerfilService(IValidadorPerfil validador) {
    this.validador = validador;
  }

  public void setValidador(IValidadorPerfil validador) {
    this.validador = validador;
  }

  public List<String> validarPerfil(Map<String, String> perfil) {
    return validador.validar(perfil);
  }

  public List<String> processarPerfil(Map<String, String> perfil) {
    List<String> mensagensErro = validarPerfil(perfil);

    return mensagensErro;
  }
}
```

Por fim, a classe Principal demonstra o uso dessas classes. Inicialmente, ela cria uma instância de ValidacaoPerfilService utilizando ValidadorPerfilBasico para validar perfis básicos. Depois, altera a estratégia de validação para ValidadorPerfilPremium para validar perfis premium.

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
public class Principal {
  public static void main(String[] args) {
    ValidacaoPerfilService servico = new
            ValidacaoPerfilService(new ValidadorPerfilBasico());

    Map<String, String> perfilBasico = new HashMap<>();
    perfilBasico.put("nome", "João Silva");
    perfilBasico.put("email", "joao.silva@example.com");

    Map<String, String> perfilInvalido = new HashMap<>();
    perfilInvalido.put("nome", "Maria Souza");

    Map<String, String> perfilPremium = new HashMap<>();
    perfilPremium.put("nome", "Ana Paula");
    perfilPremium.put("email", "ana.paula@example.com");
    perfilPremium.put("telefone", "123456789");
    perfilPremium.put("endereco", "Rua das Flores, 123");

    System.out.println("Validação de Perfil Básico:");
    List<String> errosBasico = servico.processarPerfil(perfilBasico);
    if (errosBasico.isEmpty()) {
      System.out.println("Perfil válido: " + perfilBasico);
    } else {
      System.out.println("Perfil inválido: " + perfilBasico);
      for (String erro : errosBasico) {
        System.out.println(erro);
      }
    }

    List<String> errosInvalido = servico.processarPerfil(perfilInvalido);
    if (errosInvalido.isEmpty()) {
      System.out.println("Perfil válido: " + perfilInvalido);
```

```
        } else {
            System.out.println("Perfil inválido: " + perfilInvalido);
            for (String erro : errosInvalido) {
                System.out.println(erro);
            }
        }

        servico.setValidador(new publicPremium());

        System.out.println("\nValidação de Perfil Premium:");
        List<String> errosPremium = servico.processarPerfil(perfilPremium);
        if (errosPremium.isEmpty()) {
            System.out.println("Perfil válido: " + perfilPremium);
        } else {
            System.out.println("Perfil inválido: " + perfilPremium);
            for (String erro : errosPremium) {
                System.out.println(erro);
            }
        }

        List<String> errosBasicoPremium = servico.processarPerfil(perfilBasico);
        if (errosBasicoPremium.isEmpty()) {
            System.out.println("Perfil válido: " + perfilBasico);
        } else {
            System.out.println("Perfil inválido: " + perfilBasico);
            for (String erro : errosBasicoPremium) {
                System.out.println(erro);
            }
        }
    }
}
```

A saída esperada após a execução do método main é a seguinte:

Validação de Perfil Básico:
Perfil válido: {nome=João Silva, email=joao.silva@example.com}
Perfil inválido: {nome=Maria Souza}
Campos obrigatórios ausentes: nome e/ou email.
Email inválido.

Validação de Perfil Premium:
Perfil válido: {telefone=123456789, endereco=Rua das Flores, 123, nome=Ana Paula, email=ana.paula@example.com}
Perfil inválido: {nome=João Silva, email=joao.silva@example.com}
Campos obrigatórios ausentes: nome, email, telefone e/ou endereco.
Telefone inválido. Deve conter exatamente 9 dígitos.
Endereço inválido. Deve conter pelo menos 10 caracteres.

Solução do Exercício 99

Para resolver este problema, começamos definindo uma interface IObservador que possui um método atualizar, responsável por receber as notificações de eventos.

```
public interface IObservador {
    void atualizar(String evento, String mensagem);
}
```

Em seguida, criamos uma classe abstrata Observavel que mantém uma lista de observadores e fornece métodos para adicionar e remover observadores, bem como para notificar todos os observadores registrados quando um evento ocorre.

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

public abstract class Observavel {
    private List<IObservador> observadores = new ArrayList<>();

    public void adicionarObservador(IObservador observador) {
```

```
        observadores.add(observador);
    }

    public void removerObservador(IObservador observador) {
        observadores.remove(observador);
    }

    protected void notificarObservadores(String evento, String mensagem) {
        for (IObservador observador : observadores) {
            observador.atualizar(evento, mensagem);
        }
    }
}
```

Com essas bases estabelecidas, implementamos a classe Video que estende Observavel e implementa Observador. A classe Video possui um estado que pode ser modificado através de um método setEstado. Quando o estado é alterado, o vídeo notifica todos os seus observadores. Além disso, o vídeo também reage às notificações que recebe, atualizando seu próprio estado conforme necessário.

```
public class Video extends Observavel implements IObservador {
    private String titulo;
    private String estado;

    public Video(String titulo) {
        this.titulo = titulo;
    }

    public String getTitulo() {
        return titulo;
    }

    public String getEstado() {
```

```
        return estado;
    }

    public void setEstado(String estado) {
        this.estado = estado;
        notificarObservadores("estadoAlterado", "O estado do vídeo \"" + titulo + "\" foi alterado
para: " + estado);
    }

    @Override
    public void atualizar(String evento, String mensagem) {
        System.out.println("Vídeo \"" + titulo + "\" recebeu atualização: " + mensagem);
        if (evento.equals("favorito") || evento.equals("assistido")) {
            this.estado = mensagem.split(": ")[1];
            System.out.println("Vídeo \"" + titulo + "\" reagiu à atualização. Novo estado: " +
this.estado);
        }
    }
}
```

Da mesma forma, implementamos a classe Usuario que também estende Observavel e implementa Observador. A classe Usuario possui métodos para assistir e adicionar vídeos à lista de favoritos. Quando uma ação é realizada, o usuário notifica todos os seus observadores. Além disso, o usuário reage às notificações recebidas dos vídeos, adaptando suas ações de acordo com as mudanças de estado dos vídeos.

```
public class Usuario extends Observavel implements IObservador {
    private String nome;

    public Usuario(String nome) {
        this.nome = nome;
    }
```

```
    public String getNome() {
        return nome;
    }

    public void assistirVideo(Video video) {
        System.out.println("Usuario " + nome + ": Assistindo ao vídeo \"" + video.getTitulo() +
"\".");
        video.setEstado("Assistido por " + nome);
    }

    public void adicionarFavorito(Video video) {
        System.out.println("Usuario " + nome + ": Adicionando o vídeo \"" + video.getTitulo() + "\"
à lista de favoritos.");
        video.setEstado("Favorito por " + nome);
    }

    @Override
    public void atualizar(String evento, String mensagem) {
        System.out.println("Usuario " + nome + " recebeu atualização: " + mensagem);
        if (evento.equals("estadoAlterado")) {
            System.out.println("Usuario " + nome + " está reagindo à atualização do vídeo.");
        }
    }
}
```

Finalmente, na classe principal, criamos instâncias de vídeos e usuários, registramos os observadores de maneira que eles possam reagir mutuamente às notificações e modificamos os estados e ações para demonstrar o ciclo de observação.

```
public class Main {
    public static void main(String[] args) {
        Video videoGameOfThrones = new Video("Game of Thrones");
        Video videoBreakingBad = new Video("Breaking Bad");
```

```
        Usuario usuarioAlice = new Usuario("Alice");
        Usuario usuarioBob = new Usuario("Bob");

        // Registrando observadores
        videoGameOfThrones.adicionarObservador(usuarioAlice);
        videoBreakingBad.adicionarObservador(usuarioBob);
        usuarioAlice.adicionarObservador(videoGameOfThrones);
        usuarioBob.adicionarObservador(videoBreakingBad);

        // Modificando estado dos vídeos para disparar eventos
        videoGameOfThrones.setEstado("Novo episódio disponível");
        videoBreakingBad.setEstado("disponível");

        // Usuários realizando ações com os vídeos
        usuarioAlice.assistirVideo(videoGameOfThrones);
        usuarioBob.adicionarFavorito(videoBreakingBad);
    }
}
```

A saída esperada após a execução do método main é a seguinte:

```
Usuario Alice recebeu atualização: O estado do vídeo "Game of Thrones" foi alterado para: Novo episódio disponível
Usuario Alice está reagindo à atualização do vídeo.
Usuario Bob recebeu atualização: O estado do vídeo "Breaking Bad" foi alterado para: disponível
Usuario Bob está reagindo à atualização do vídeo.
Usuario Alice: Assistindo ao vídeo "Game of Thrones".
Usuario Alice recebeu atualização: O estado do vídeo "Game of Thrones" foi alterado para: Assistido por Alice
Usuario Alice está reagindo à atualização do vídeo.
Usuario Bob: Adicionando o vídeo "Breaking Bad" à lista de favoritos.
Usuario Bob recebeu atualização: O estado do vídeo "Breaking Bad" foi alterado para: Favorito por Bob
```

Usuario Bob está reagindo à atualização do vídeo.

Solução do Exercício 100

A)

Inicialmente iremos implementar o projeto Java Maven da Biblioteca de Ordenação. Para isso, criamos a interface IAlgoritmoOrdenacao que define o contrato que os algoritmos de ordenação devem seguir. Ela possui um método ordenar que recebe um array de strings e um booleano indicando se a ordem deve ser crescente ou decrescente.

```
package com.mycompany.ordenacao;

public interface IAlgoritmoOrdenacao {
  void ordenar(String[] array, boolean ordem);
}
```

Em seguida, criamos a classe BubbleSort que implementa o algoritmo de ordenação BubbleSort. Ele percorre repetidamente a lista de elementos, comparando e trocando elementos adjacentes se estiverem na ordem errada, até que a lista esteja ordenada.

```
package com.mycompany.ordenacao;

public class BubbleSort implements IAlgoritmoOrdenacao {

  @Override
  public void ordenar(String[] array, boolean ordem) {
    int n = array.length;
    boolean swapped;
    for (int i = 0; i < n - 1; i++) {
      swapped = false;
      for (int j = 0; j < n - i - 1; j++) {
        if (ordem ? array[j].compareTo(array[j + 1]) > 0 : array[j].compareTo(array[j + 1]) < 0) {
          String temp = array[j];
          array[j] = array[j + 1];
```

```
                array[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
  }
}
```

A classe SelectionSort implementa o algoritmo de ordenação SelectionSort. Este algoritmo divide a lista em duas partes: a parte ordenada e a parte não ordenada. Ele repete o processo de encontrar o menor (ou maior, dependendo da ordem) elemento da parte não ordenada e trocá-lo com o primeiro elemento da parte não ordenada até que a lista inteira esteja ordenada.

```
package com.mycompany.ordenacao;

public class SelectionSort implements IAlgoritmoOrdenacao {

  @Override
  public void ordenar(String[] array, boolean ordem) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
      int index = i;
      for (int j = i + 1; j < n; j++) {
        if (ordem ? array[j].compareTo(array[index]) < 0 : array[j].compareTo(array[index]) > 0)
{
          index = j;
        }
      }
      String temp = array[index];
      array[index] = array[i];
      array[i] = temp;
```

```
    }
  }
}
```

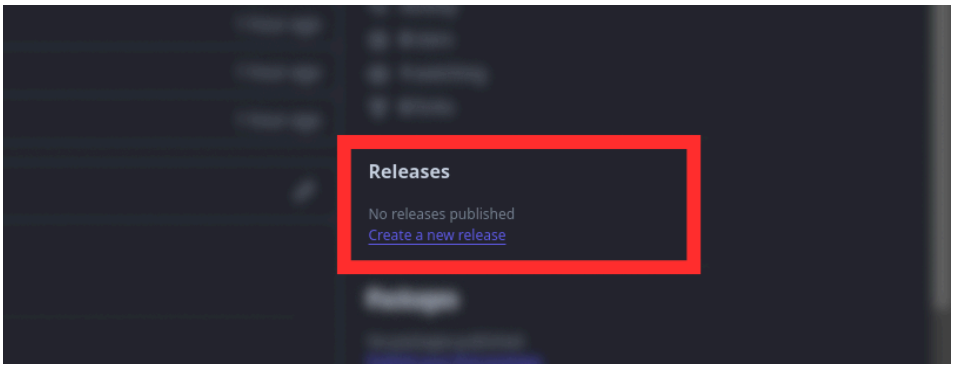
A classe de serviço OrdenadorService funcionará como interface da biblioteca, gerenciando os algoritmos de ordenação disponíveis e fornecendo o método para ordenar arrays usando esses algoritmos. Também permite a adição de novos algoritmos de ordenação de acordo com o Princípio do Aberto/Fechado.

```
package com.mycompany.ordenacao;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class OrdenadorService {
  private Map<String, IAlgoritmoOrdenacao> algoritmosDisponivel;

  public OrdenadorService(){
    this.algoritmosDisponivel = new HashMap<>();
    this.algoritmosDisponivel.put("BobbleSort", new BubbleSort());
    this.algoritmosDisponivel.put("SelectionSort", new SelectionSort());
  }

  public void ordenar(String metodo, String[] array, boolean ordem) {
    IAlgoritmoOrdenacao algoritmo = this.algoritmosDisponivel.get(metodo);
    algoritmo.ordenar(array, ordem);
  }

  public List<String> getNomesAlgoritmos(){
    List<String> nomes = new ArrayList<>();
```

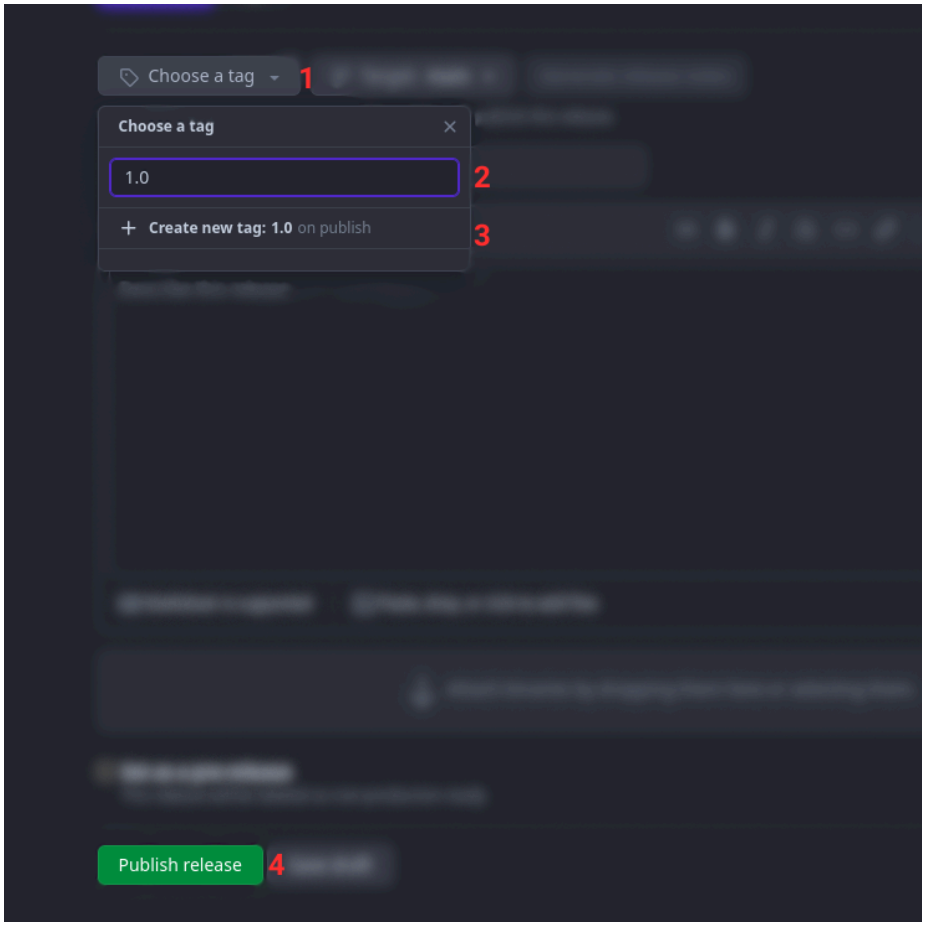
```
        for(Map.Entry<String, IAlgoritmoOrdenacao> elem : algoritmosDisponivel.entrySet()){
            nomes.add(elem.getKey());
        }
        return nomes;
    }
}
```

Partindo do pressuposto que o código acima se encontra em um repositório público do Github, para a publicação do arquivo Jar com o serviço Jitpack, importante ressaltar que este guia é válido na data de publicação deste material. Portanto, certifique-se da disponibilidade dos serviços utilizados e realize os seguintes passos:

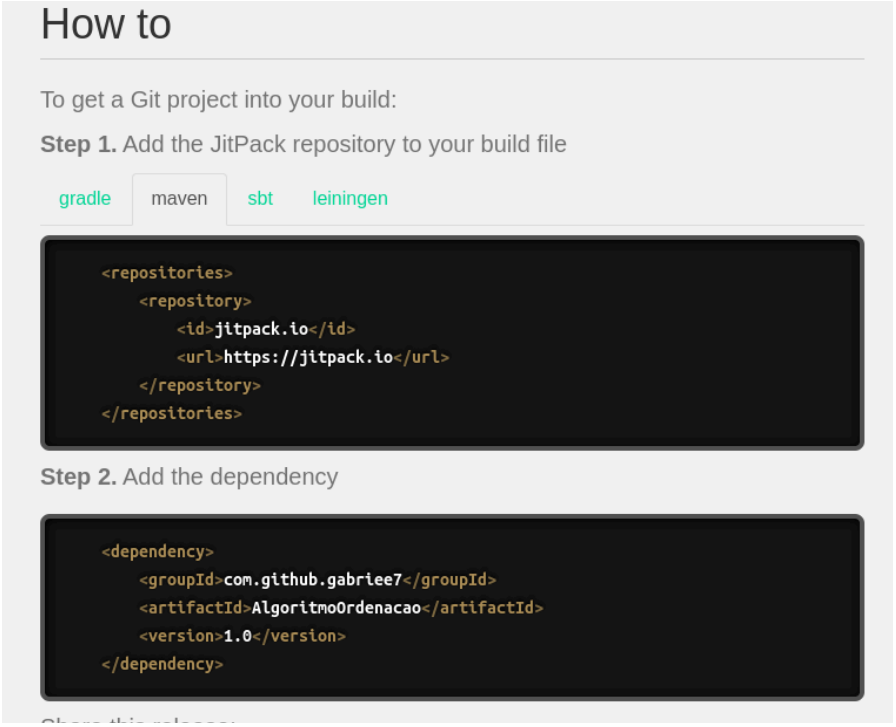
1. Na página inicial do repositório do Github clique em “Create a new release”, veja na imagem abaixo.



2. Em seguida realize os passos descritos na imagem abaixo para finalizar a criação de uma nova tag e publicar um release.



3. Acesse o site do Jitpack (https://jitpack.io/) para o builder e publicação do arquivo Jar do seu projeto. Na página inicial, no campo “Git repo url” cole o link do repositório no Github, por exemplo “https://github.com/seu-usuario/seu-repositorio”, em seguida clique em “Look up”.
4. Em seguida, o Jitpack identificará as versões lançadas do repositório, clique em “Get it” na linha da versão desejada.
5. Logo irá mostrar um passo a passo como usar no seu projeto Maven, adicionando trechos de configuração no arquivo pom.xml, veja abaixo.



B)

Agora, é implementado o segundo projeto que envolve o desenvolvimento da interface gráfica com Java Swing.

Para utilizar o arquivo Jar publicado pelo Jitpack neste projeto, insira os seguintes trechos de configuração no arquivo pom.xml, lembre-se de substituir as variáveis dentro das tags.

```
<dependencies>
  <dependency>
  <groupId>com.github.seu-usuario</groupId>
  <artifactId>seu-repositorio</artifactId>
  <version>sua-versao</version>
</dependency>
</dependencies>

<repositories>
    <repository>
      <id>jitpack.io</id>
      <url>https://jitpack.io</url>
```

```
        </repository>
    </repositories>
```

Com a adotação a arquitetura MVP , a aplicação é dividida em Model contendo dados e lógica de negócios, View representando a interface gráfica passiva e a Presenter responsável pelo controle da lógica da interface. Isso separa responsabilidades, facilitando manutenção e testes.

A classe OrdenacaoPresenter controla a lógica da aplicação de ordenação de elementos, atuando como intermediária entre a interface gráfica passiva OrdenacaoView e os serviços de ordenação externo e carregamento de arquivos. Ela configura os componentes da interface, gerencia eventos disparados pelas ações do usuário, realiza a ordenação conforme o método escolhido e exibe mensagens de erro ou sucesso para o usuário.

```
package com.mycompany.interfacegrafica;

import com.mycompany.ordenacao.OrdenadorService; //importando do arquivo Jar
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.List;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;

public class OrdenacaoPresenter {
    private OrdenacaoView view;
    private OrdenadorService ordenadorService;
    private CarregadorDeArquivo carregadorArquivo;
    private String[] elementos;

    public OrdenacaoPresenter() {
        this.view = new OrdenacaoView();
        this.ordenadorService = new OrdenadorService();
        this.carregadorArquivo = new CarregadorDeArquivo();
```

```
        configuraTela(ordenadorService.getNomesAlgoritmos());
        view.setVisible(true);
    }

    private void configuraTela(List<String> algoritmosDisponíveis){
        for(String elem : algoritmosDisponíveis){
            view.getCmbMetodo().addItem(elem);
        }

        view.getBtnCarregarArquivo().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                carregarArquivo();
            }
        });

        view.getBtnOrdenar().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                ordenar();
            }
        });
    }

    private void carregarArquivo() {
        JFileChooser fileChooser = new JFileChooser();
        int returnValue = fileChooser.showOpenDialog(null);
        if (returnValue == JFileChooser.APPROVE_OPTION) {
            try {
                elementos =
carregadorArquivo.carregarArrayDoArquivo(fileChooser.getSelectedFile().getPath());
                view.getLstSemOrdem().setListData(elementos);
```

```
        } catch (IOException e) {
            exibirMensagem("Erro ao carregar o arquivo: " + e.getMessage());
        }
    }
}

private void ordenar() {
    String metodo = view.getCmbMetodo().getSelectedItem().toString();
    boolean ordem = view.getRbtnCrescente().isSelected();

    if (metodo != null && elementos != null) {
        long startTime = System.nanoTime();
        ordenadorService.ordenar(metodo, elementos, ordem);
        long endTime = System.nanoTime();
        long duration = (endTime - startTime) / 1000000;

        view.getLstOrdenados().setListData(elementos);
        view.getLblTempo().setText(duration + " ms");
    }else{
        exibirMensagem("Método de ordenação inválida");
    }
}

private void exibirMensagem(String mensagem){
    JOptionPane.showMessageDialog(view, mensagem);
}
}
```

A classe OrdenacaoView define a interface gráfica da aplicação utilizando o framework Java Swing. Esta classe contém todos os componentes visuais necessários, como botões, listas, labels e caixas de seleção. Ela oferece métodos para acessar esses componentes, permitindo que outras partes da aplicação interajam com a interface gráfica.

```
package com.mycompany.interfacegrafica;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JScrollPane;
import javax.swing.JFrame;

public class OrdenacaoView extends JFrame {
  private JButton btnCarregarArquivo;
  private JButton btnOrdenar;
  private ButtonGroup buttonGroup1;
  private JComboBox<String> cmbMetodo;
  private JLabel jLabel5;
  private JPanel jPanel1;
  private JScrollPane jScrollPane1;
  private JScrollPane jScrollPane2;
  private JLabel lblMetodoOrdenacao;
  private JLabel lblOrdem;
  private JLabel lblOrdenados;
  private JLabel lblSemOrdem;
  private JLabel lblTempo;
  private JList<String> lstOrdenados;
  private JList<String> lstSemOrdem;
  private JRadioButton rbtnCrescente;
  private JRadioButton rbtnDecrescente;

  public OrdenacaoView() {
```

```
    initComponents();
}

private void initComponents() {
    buttonGroup1 = new javax.swing.ButtonGroup();
    jPanel1 = new javax.swing.JPanel();
    lblSemOrdem = new javax.swing.JLabel();
    jScrollPane1 = new javax.swing.JScrollPane();
    lstSemOrdem = new javax.swing.JList<>();
    btnCarregarArquivo = new javax.swing.JButton();
    cmbMetodo = new javax.swing.JComboBox<>();
    lblMetodoOrdenacao = new javax.swing.JLabel();
    lblOrdem = new javax.swing.JLabel();
    rbtnCrescente = new javax.swing.JRadioButton();
    rbtnDecrescente = new javax.swing.JRadioButton();
    jScrollPane2 = new javax.swing.JScrollPane();
    lstOrdenados = new javax.swing.JList<>();
    lblOrdenados = new javax.swing.JLabel();
    btnOrdenar = new javax.swing.JButton();
    jLabel5 = new javax.swing.JLabel();
    lblTempo = new javax.swing.JLabel();

    javax.swing.GroupLayout jPanel1Layout = new javax.swing.GroupLayout(jPanel1);
    jPanel1.setLayout(jPanel1Layout);
    jPanel1Layout.setHorizontalGroup(
        jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGap(0, 100, Short.MAX_VALUE)
    );
    jPanel1Layout.setVerticalGroup(
        jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGap(0, 100, Short.MAX_VALUE)
    );
```

```
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setTitle("Ordenação");
        setMinimumSize(new java.awt.Dimension(600, 300));
        lblSemOrdem.setText("Elementos a serem ordenados");
        jScrollPane1.setViewportView(lstSemOrdem);
        btnCarregarArquivo.setText("Carregar do arquivo");
        lblMetodoOrdenacao.setText("Método de ordenação");
        lblOrdem.setText("Ordem");
        buttonGroup1.add(rbtnCrescente);
        rbtnCrescente.setText("Crescente");
        buttonGroup1.add(rbtnDecrescente);
        rbtnDecrescente.setText("Decrescente");
        jScrollPane2.setViewportView(lstOrdenados);
        lblOrdenados.setText("Elementos ordenados");
        btnOrdenar.setText("Ordenar");
        jLabel5.setText("Tempo:");
        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addContainerGap()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                    .addComponent(lblOrdenados, javax.swing.GroupLayout.PREFERRED_SIZE, 185,
javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(lblSemOrdem, javax.swing.GroupLayout.PREFERRED_SIZE,
259, javax.swing.GroupLayout.PREFERRED_SIZE)
```

```
                    .addComponent(btnOrdenar, javax.swing.GroupLayout.Alignment.TRAILING)
                    .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                        .addComponent(btnCarregarArquivo)
                        .addComponent(jScrollPane1,
javax.swing.GroupLayout.PREFERRED_SIZE, 185,
javax.swing.GroupLayout.PREFERRED_SIZE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                        .addComponent(lblMetodoOrdenacao)
                        .addComponent(cmbMetodo,
javax.swing.GroupLayout.PREFERRED_SIZE, 123,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(lblOrdem)
                        .addComponent(rbtnCrescente)
                        .addComponent(rbtnDecrescente))))

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addGroup(layout.createSequentialGroup()
                      .addGap(88, 88, 88)
                      .addComponent(jLabel5)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                      .addComponent(lblTempo))
                    .addGroup(layout.createSequentialGroup()

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                        .addComponent(jScrollPane2, javax.swing.GroupLayout.PREFERRED_SIZE,
185, javax.swing.GroupLayout.PREFERRED_SIZE)))))
```

```
            .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addGap(51, 51, 51)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                    .addComponent(lblSemOrdem)
                    .addComponent(lblOrdenados))
                .addGap(18, 18, 18)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED_SIZE,
203, javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addGroup(layout.createSequentialGroup()
                                .addComponent(lblMetodoOrdenacao)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                                .addComponent(cmbMetodo, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addGap(18, 18, 18)
                                .addComponent(lblOrdem)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                                .addComponent(rbtnCrescente)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                                .addComponent(rbtnDecrescente)
```

```
.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                        .addComponent(btnOrdenar)))
                .addGap(18, 18, 18)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                        .addComponent(jLabel5)
                        .addComponent(lblTempo))
                    .addComponent(btnCarregarArquivo)))
            .addComponent(jScrollPane2, javax.swing.GroupLayout.PREFERRED_SIZE, 203,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addContainerGap(23, Short.MAX_VALUE))
      );
      pack();
  }

  public JButton getBtnCarregarArquivo() {
      return btnCarregarArquivo;
  }

  public JButton getBtnOrdenar() {
      return btnOrdenar;
  }

  public ButtonGroup getButtonGroup1() {
      return buttonGroup1;
  }

  public JComboBox<String> getCmbMetodo() {
      return cmbMetodo;
```

```
}

public JLabel getLblMetodoOrdenacao() {
    return lblMetodoOrdenacao;
}

public JLabel getLblOrdem() {
    return lblOrdem;
}

public JLabel getLblOrdenados() {
    return lblOrdenados;
}

public JLabel getLblSemOrdem() {
    return lblSemOrdem;
}

public JLabel getLblTempo() {
    return lblTempo;
}

public JList<String> getLstOrdenados() {
    return lstOrdenados;
}

public JList<String> getLstSemOrdem() {
    return lstSemOrdem;
}

public JRadioButton getRbtnCrescente() {
    return rbtnCrescente;
```

```
  }

  public JRadioButton getRbtnDecrescente() {
    return rbtnDecrescente;
  }
}
```

A classe CarregadorDeArquivo é responsável por ler os dados de um arquivo e carregá-los em um array de strings. Este componente permite que a aplicação importe dados externos, que serão posteriormente ordenados. Utilizando BufferedReader e FileReader, o CarregadorDeArquivo percorre o arquivo linha por linha, armazenando cada linha em uma lista que, ao final, é convertida em um array de strings.

```
package com.mycompany.interfacegrafica;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class CarregadorDeArquivo {
  public String[] carregarArrayDoArquivo(String caminhoArquivo) throws IOException {
    List<String> lista = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(caminhoArquivo))) {
      String linha;
      while ((linha = br.readLine()) != null) {
        lista.add(linha);
      }
    }
    return lista.toArray(new String[0]);
  }
}
```

C)

A classe Principal contém o método main, que é o ponto de entrada da aplicação. Quando a aplicação é iniciada, uma instância de OrdenacaoPresenter é criada, iniciando o processo de configuração e exibição da interface gráfica. Esta classe encapsula a inicialização da aplicação, delegando a responsabilidade de controle e interação à OrdenacaoPresenter.

```
package com.mycompany.interfacegrafica;

public class Principal {
  public static void main(String[] args) {
     OrdenacaoPresenter presenter = new OrdenacaoPresenter();
  }
}
```

GUIA DE RESPOSTAS

S.O.L.I.D. E DESIGN PATTERNS

CADERNO DE EXERCÍCIOS

PRATIQUE E EVOLUA

Neste guia, você encontrará respostas detalhadas para cada exercício do "Caderno de Exercícios: Pratique e Evolua", elucidando as práticas recomendadas no desenvolvimento de software. As explicações fornecidas evidenciam a aplicação dos princípios S.O.L.I.D. e dos diversos padrões de design abordados, fortalecendo seu aprendizado ao demonstrar como cada conceito teórico é implementado na prática.

Recomenda-se o uso deste Guia de Respostas em conjunto com a prática ativa e o diálogo com colegas, profissionais e professores. Utilize este material como uma referência contínua durante seus estudos.

Desejamos a todos um aprofundamento proveitoso nos estudos e na prática dos princípios S.O.L.I.D. e dos padrões de projeto.

Clayton Vieira Fraga Filho

Universidade Federal
do Espírito Santo

ISBN 978-650107208-1