Aplicando Princípios SOLID em Projetos Educacionais em Java: Um Estudo de Caso na Refatoração de uma Aplicação Cliente-Servidor com Sockets

Introdução

Os princípios SOLID — **Single Responsibility Principle (SRP)**, **Open/Closed Principle (OCP)**, **Liskov Substitution Principle (LSP)**, **Interface Segregation Principle (ISP)** e **Dependency Inversion Principle (DIP)** — são diretrizes fundamentais para o desenvolvimento de software manutenível, escalável e robusto. Embora frequentemente associados a sistemas de grande porte, esses princípios são igualmente valiosos em projetos educacionais pequenos. Aplicá-los em tais contextos promove um raciocínio disciplinado, melhora a lógica de design e prepara desenvolvedores para desafios do mundo real.

Este artigo explora a refatoração de uma aplicação cliente-servidor em Java que utiliza sockets para comunicação. O projeto original, criado para fins educacionais, demonstra comunicação multithreaded básica com sockets, mas carece de modularidade e extensibilidade. Ao refatorá-lo com os princípios SOLID, transformamos o sistema em uma solução mais coesa, manutenível e reutilizável. Vamos:

1. Revisar brevemente os princípios SOLID.
2. Apresentar o projeto refatorado.
3. Detalhar o processo de refatoração, justificando as mudanças com base nos princípios SOLID.
4. Discutir os benefícios dessas alterações em termos de manutenibilidade, reutilização, legibilidade, coesão e acoplamento.

Princípios SOLID: Uma Revisão Rápida

- **Princípio da Responsabilidade Única (SRP)**: Uma classe deve ter apenas uma razão para mudar, ou seja, uma única responsabilidade bem definida.
- **Princípio Aberto/Fechado (OCP)**: Entidades de software devem estar abertas para extensão, mas fechadas para modificação.
- **Princípio da Substituição de Liskov (LSP)**: Subtipos devem ser substituíveis por seus tipos base sem alterar a corretude do programa.
- **Princípio da Segregação de Interfaces (ISP)**: Clientes não devem ser forçados a depender de interfaces que não utilizam.
- **Princípio da Inversão de Dependência (DIP)**: Módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações. Abstrações não devem depender de detalhes.

Esses princípios guiam nossa refatoração para garantir um código modular, extensível e fácil de manter.

O Projeto Original e o Refatorado

Visão Geral do Projeto Original

O projeto original é composto por três classes:

- **Servidor**: Um servidor que escuta na porta 52000, aceita conexões de clientes e processa cada conexão em uma thread separada. Ele lê uma string do cliente, converte-a para maiúsculas e a envia de volta.
- **Cliente**: Um cliente que se conecta ao servidor, envia uma string fornecida pelo usuário e exibe a resposta do servidor.
- **Leitura**: Uma classe utilitária para leitura de entrada do console.

Embora funcional, o código original apresenta limitações:

- **Acoplamento Forte**: A lógica de gerenciamento de conexões e processamento de texto está misturada, dificultando modificações ou extensões.
- **Violação do SRP**: A classe `Servidor` lida com gerenciamento de sockets, threading e processamento de texto.
- **Extensibilidade Limitada**: Adicionar novas lógicas de processamento de texto ou suportar outros tipos de dados (como transferência de arquivos) exige mudanças significativas no código.
- **Comportamento Fixo**: O servidor apenas converte texto para maiúsculas, e o cliente está restrito à troca de texto.

Visão Geral do Projeto Refatorado

O projeto refatorado introduz uma arquitetura modular com vários pacotes (`application`, `client`, `conection.manager`, `input`, `server`, `text.processor`). Os principais componentes incluem:

- **ServerMain**: Inicializa o servidor com um processador de texto configurável e um gerenciador de conexões.
- **ConnectionManager**: Gerencia conexões de socket e delega o processamento a um `ConnectionProcessor`.
- **ConnectionProcessor**: Uma interface com implementações como `TextConnectionProcessor` (para texto) e `FileConnectionProcessor`(para arquivos).
- **TextProcessor**: Uma interface com implementações como `UpperCaseProcessor`, `HaschCodeProcessor`, `LowerCaseProcessor` e `ReverseTextProcessor`.
- **ClientMain**: Gerencia conexões do cliente e delega a comunicação a um `ConnectionHandler`.
- **ConnectionHandler**: Uma interface com implementações como `TextHandler` (para texto) e `FileHandler` (para arquivos).
- **InputReader**: Uma interface com uma implementação `ConsoleInputReader` para leitura de entrada do usuário.

Essa estrutura permite que o sistema suporte múltiplos tipos de processamento de dados e protocolos de comunicação, aderindo aos princípios SOLID.

Processo de Refatoração: Aplicando os Princípios SOLID

Vamos percorrer o processo de refatoração, destacando como os princípios SOLID moldaram as decisões de design.

Passo 1: Resolvendo o SRP no Servidor

Problema: Na classe original `Servidor`, uma única classe gerencia sockets, threading e processamento de texto (conversão para maiúsculas). Isso viola o SRP, pois a classe tem múltiplas responsabilidades, dificultando modificações e testes.

Solução: Dividimos as responsabilidades em classes separadas:

- **ConnectionManager**: Responsável por aceitar conexões de socket e gerenciar threads.
- **ConnectionProcessor**: Uma interface para processar conexões individuais.
- **TextProcessor**: Uma interface para lógica de transformação de texto.

Justificativa SOLID:

- **SRP**: Cada classe tem uma única responsabilidade:
  - `ConnectionManager` gerencia o socket do servidor e threading.
  - `TextConnectionProcessor` lida com entrada/saída de sockets para comunicação baseada em texto.
  - Implementações de `TextProcessor` (como `UpperCaseProcessor`) focam apenas na transformação de texto.
- **DIP**: `ConnectionManager` depende da interface `ConnectionProcessor`, não de implementações concretas, permitindo flexibilidade na lógica de processamento.

Exemplo de Código:

```
// ConnectionManager.java
public class ConnectionManager {
    private final ServerSocket serverSocket;
    private final ConnectionProcessor processor;

    public ConnectionManager(int port, ConnectionProcessor processor) throws IOException {
        this.serverSocket = new ServerSocket(port);
        this.processor = processor;
    }

    public void start() {
        Logger.getLogger(ConnectionManager.class.getName()).info("Servidor iniciado na porta " + serverSocket.getLocalPort
());
        while (true) {
            try {
                Socket connection = serverSocket.accept();
                Thread thread = new Thread(() -> processor.process(connection));
                thread.start();
            } catch (IOException e) {
                Logger.getLogger(ConnectionManager.class.getName()).log(Level.SEVERE, "Erro ao aceitar conexão", e);
            }
        }
    }
}
```

Esse design modular permite trocar o `ConnectionProcessor` (por exemplo, para transferência de arquivos) sem modificar o `ConnectionManager`.

Passo 2: Habilitando Extensibilidade com o OCP

Problema: O servidor original é fixo para converter texto em maiúsculas. Adicionar nova lógica de processamento de texto (como inverter texto) exige modificar o método `run`, violando o OCP.

Solução: Introduzimos a interface `TextProcessor` com várias implementações (`UpperCaseProcessor`, `HaschCodeProcessor`, `LowerCaseProcessor`, `ReverseTextProcessor`). O `TextConnectionProcessor` usa um `TextProcessor` para lidar com transformações de texto, facilitando a adição de novos processadores.

Justificativa SOLID:

- **OCP**: O sistema está aberto para extensão (novas implementações de `TextProcessor` podem ser adicionadas) mas fechado para modificação (não é necessário alterar `TextConnectionProcessor`).
- **DIP**: `TextConnectionProcessor` depende da interface `TextProcessor`, não de implementações específicas.

Exemplo de Código:

```
// TextProcessor.java
public interface TextProcessor {
    String process(String input);
}

// UpperCaseProcessor.java
public class UpperCaseProcessor implements TextProcessor {
    @Override
    public String process(String input) {
        return "operação: [UPPERCASE] : " + input.toUpperCase();
    }
}

// TextConnectionProcessor.java
public class TextConnectionProcessor implements ConnectionProcessor {
    private final TextProcessor textProcessor;

    public TextConnectionProcessor(TextProcessor textProcessor) {
        this.textProcessor = textProcessor;
    }

    @Override
    public void process(Socket connection) {
        try (DataInputStream input = new DataInputStream(connection.getInputStream());
             DataOutputStream output = new DataOutputStream(connection.getOutputStream())) {
            String inputText = input.readUTF();
            String processedText = textProcessor.process(inputText);
            output.writeUTF(processedText);
        } catch (IOException e) {
            Logger.getLogger(TextConnectionProcessor.class.getName()).log(Level.SEVERE, "Erro ao processar conexão", e);
        }
    }
}
```

Para adicionar um novo processador de texto, basta implementar `TextProcessor` sem alterar o código existente.

Passo 3: Suportando Múltiplos Tipos de Comunicação com LSP e ISP

Problema: O projeto original suporta apenas comunicação baseada em texto. Adicionar transferência de arquivos exigiria mudanças significativas no servidor e no cliente, e o design monolítico não suporta polimorfismo.

Solução: Introduzimos as interfaces `ConnectionProcessor` e `ConnectionHandler`. O servidor suporta `TextConnectionProcessor` e `FileTransferConnectionProcessor`, enquanto o cliente suporta `TextHandler` e `FileHandler`. Essas implementações podem ser trocadas sem afetar a lógica principal.

Justificativa SOLID:

- **LSP**: Subtipos como `TextConnectionProcessor` e `FileConnectionProcessor`podem substituir `ConnectionProcessor` sem quebrar o sistema. Da mesma forma, `TextHandler` e `FileHandler` são intercambiáveis com `ConnectionHandler`.
- **ISP**: As interfaces são focadas e específicas. `ConnectionProcessor` define um único método `process`, e `ConnectionHandler` define um único método `handle`, garantindo que os clientes dependam apenas de métodos relevantes.
- **DIP**: Tanto o servidor quanto o cliente dependem de abstrações (`ConnectionProcessor` e `ConnectionHandler`), não de classes concretas.

Exemplo de Código:

```
// FileTransferConnectionProcessor.java
public class FileConnectionProcessorimplements ConnectionProcessor {
    private final Path baseDirectory;

    public FileTransferConnectionProcessor(Path baseDirectory) {
        this.baseDirectory = baseDirectory;
    }

    @Override
    public void process(Socket connection) {
        try (DataInputStream input = new DataInputStream(connection.getInputStream());
             DataOutputStream output = new DataOutputStream(connection.getOutputStream())) {
            String fileName = input.readUTF();
            Path filePath = baseDirectory.resolve(fileName);
            if (Files.exists(filePath)) {
                byte[] fileBytes = Files.readAllBytes(filePath);
                output.writeInt(fileBytes.length);
                output.write(fileBytes);
            } else {
                output.writeInt(0);
            }
        } catch (IOException e) {
            Logger.getLogger(FileTransferConnectionProcessor.class.getName()).log(Level.SEVERE, "Erro ao transferir arquiv
o", e);
        }
    }
}
```

Isso permite que o servidor lide com transferências de arquivos sem modificar a lógica de gerenciamento de conexões.

Passo 4: Abstraindo a Entrada de Dados com DIP

Problema: A classe original `Leitura` está fortemente acoplada à entrada do console, limitando a flexibilidade (por exemplo, para testes ou fontes de entrada alternativas).

Solução: Introduzimos a interface `InputReader` com uma implementação `ConsoleInputReader`. Isso permite que o cliente use diferentes fontes de entrada (como arquivos ou entradas simuladas para testes) ao trocar implementações.

Justificativa SOLID:

- **DIP**: `ClientMain` e implementações de `ConnectionHandler` dependem da interface `InputReader`, não de `ConsoleInputReader`.
- **SRP**: `ConsoleInputReader` é responsável apenas por ler entrada do console.

Exemplo de Código:

```
// InputReader.java
public interface InputReader {
    String readInput(String prompt);
}

// ConsoleInputReader.java
public class ConsoleInputReader implements InputReader {
    private final BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

    @Override
    public String readInput(String prompt) {
        System.out.println(prompt);
        try {
            return reader.readLine();
        } catch (IOException e) {
            Logger.getLogger(ConsoleInputReader.class.getName()).log(Level.SEVERE, "Erro ao ler entrada", e);
            return "";
        }
    }
}
```

Essa abstração torna o cliente mais flexível e testável.

Passo 5: Estruturando o Cliente com SRP e DIP

Problema: A classe original `Cliente` lida com gerenciamento de sockets, entrada/saída e interação com o usuário, violando o SRP e dificultando extensões.

Solução: Refatoramos o cliente em:

- **ClientMain**: Gerencia o ciclo de vida do socket e delega a comunicação a um `ConnectionHandler`.
- **ConnectionHandler**: Uma interface com implementações para tipos específicos de comunicação (`TextHandler`, `FileHandler`).
- **InputReader**: Gerencia a entrada do usuário.

Justificativa SOLID:

- **SRP**: `ClientMain` gerencia conexões, `ConnectionHandler` lida com a lógica de comunicação, e `InputReader` gerencia a entrada.
- **DIP**: `ClientMain` depende da interface `ConnectionHandler`, permitindo diferentes protocolos de comunicação.
- **OCP**: Novos tipos de comunicação podem ser adicionados implementando `ConnectionHandler`.

Exemplo de Código:

```
// ClientMain.java
public class ClientMain {
    private final ConnectionHandler handler;

    public ClientMain(ConnectionHandler handler) {
        this.handler = handler;
    }

    public void connect(String host, int port) {
        try (Socket socket = new Socket(host, port)) {
            handler.handle(socket);
        } catch (ConnectException e) {
            System.err.println("Não foi possível conectar ao servidor: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Erro na comunicação: " + e.getMessage());
        }
    }
}
```

Essa estrutura espelha a modularidade do servidor, garantindo consistência e extensibilidade.

Passo 6: Instanciando as Implementações das Interfaces

Problema: No código original, a instânciação era rígida, com a lógica de processamento (como a conversão para maiúsculas) diretamente embutida no servidor. Isso limitava a flexibilidade e dificultava a troca de comportamentos sem alterar o código principal.

Solução: No design refatorado, utilizamos injeção de dependência para instanciar implementações das interfaces `TextProcessor` e `ConnectionProcessor`, permitindo escolher dinamicamente o comportamento do servidor. Abaixo está um exemplo de como instanciar essas implementações no método `main` da classe `ServerMain`:

```
public static void main(String[] args) {
    try {
        //TextProcessor processor = new UpperCaseProcessor();
        TextProcessor processor = new HashCodeProcessor();
        ConnectionProcessor connectionProcessor = new TextConnectionProcessor(processor);
        ConnectionManager manager = new ConnectionManager(52000, connectionProcessor);
        manager.start();
    } catch (IOException e) {
        Logger.getLogger(ServerMain.class.getName()).log(Level.SEVERE, "Erro ao iniciar servidor", e);
    }
}
```

Justificativa SOLID:

- **DIP**: A classe `ServerMain` depende da interface `ConnectionProcessor`, não de uma implementação concreta como `TextConnectionProcessor`. Isso permite substituir o processador (ex.: por um `FileConnectionProcessor`) sem alterar o código de inicialização.
- **OCP**: Ao comentar `UpperCaseProcessor` e instanciar `HashCodeProcessor`, demonstramos que novas implementações de `TextProcessor` podem ser injetadas facilmente, extendendo o comportamento sem modificar o núcleo do sistema.
- **SRP**: A responsabilidade de criar instâncias é delegada ao ponto de entrada (`ServerMain`), enquanto a lógica de processamento fica nas classes específicas.

Explicação:

- O código permite alternar entre diferentes processadores de texto (ex.: `UpperCaseProcessor` ou `HashCodeProcessor`) apenas ajustando a instânciação, sem impactar o `ConnectionManager` ou o `TextConnectionProcessor`.

- A injeção de `TextProcessor` no construtor de `TextConnectionProcessor` reflete um design desacoplado, onde o comportamento pode ser configurado em tempo de execução.

Essa abordagem prática ilustra como o design refatorado suporta flexibilidade e manutenção, sendo um exemplo valioso para projetos educacionais que buscam ensinar a aplicação de SOLID.

Benefícios do Design Refatorado

A aplicação dos princípios SOLID trouxe melhorias significativas:

- **Manutenibilidade**: Cada classe tem uma única responsabilidade, facilitando a compreensão, modificação e depuração. Por exemplo, alterar a lógica de processamento de texto exige apenas atualizar ou adicionar uma implementação de `TextProcessor`.
- **Reutilização**: Interfaces como `TextProcessor`, `ConnectionProcessor` e `ConnectionHandler` permitem reutilização do código em diferentes contextos. O mesmo `ConnectionManager` pode ser usado para texto ou transferência de arquivos.
- **Legibilidade**: A separação clara de responsabilidades e nomes significativos de classes/interfaces (como `TextConnectionProcessor`) tornam o código autodocumentado. Comentários e Javadoc aumentam ainda mais a clareza.
- **Coesão**: As classes são altamente coesas, focando em tarefas específicas (por exemplo, `UpperCaseProcessor` lida apenas com conversão para maiúsculas). Isso reduz a complexidade e aumenta a confiabilidade.
- **Baixo Acoplamento**: Dependências em abstrações (como `TextProcessor`, `InputReader`) reduzem o acoplamento, permitindo que componentes sejam trocados ou modificados independentemente. Por exemplo, mudar de `ConsoleInputReader` para um leitor baseado em arquivos não exige alterações na lógica principal do cliente.

Do ponto de vista educacional, esse processo de refatoração treina desenvolvedores a:

- Pensar de forma modular e antecipar extensões futuras.
- Identificar e corrigir problemas de design (como acoplamento forte e responsabilidades misturadas).
- Aplicar padrões de projeto implicitamente (como o padrão Strategy em `TextProcessor`).
- Escrever código testável ao isolar dependências.

Conclusão

Mesmo projetos educacionais pequenos se beneficiam enormemente dos princípios SOLID. Ao refatorar uma aplicação cliente-servidor simples baseada em sockets, transformamos um código monolítico e rígido em um sistema modular, extensível e manutenível. O processo não apenas melhorou a qualidade do código, mas também serviu como um exercício prático na aplicação dos princípios SOLID, promovendo raciocínio lógico e pensamento disciplinado em design. Para estudantes e educadores, tais exercícios conectam conceitos teóricos à engenharia de software do mundo real, preparando desenvolvedores para desafios mais complexos enquanto mantêm o foco em um código limpo e robusto.

O projeto deste artigo está no github: https://github.com/rgiovann/exemplo-solid