UNIVERSIDADE NOVE DE JULHO - UNINOVE
ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

DESIGN PATTERN

YANN SANTANA – 923106990

LUCAS LUIS - 923108658

São Paulo
2024

RESUMO

Nesta atividade, será implementado um sistema utilizando Design Patterns de três categorias: Criacional, Estrutural e Comportamental. O objetivo é aplicar conceitos de Design Patterns para resolver problemas comuns de desenvolvimento de software de forma eficiente, mantendo o código flexível, modular e de fácil manutenção. Serão escolhidos padrões de design, como Builder, Composite e Chain of Responsibility

BUILDER

Padrão Criacional: Builder

O padrão **Builder** é um padrão criacional que permite a construção de um objeto complexo, passo a passo, fornecendo uma interface para criar diferentes representações de um objeto.

Exemplo: Construção de um Computador Personalizado

Imagine que estamos criando um sistema que permite a construção de um computador com diferentes componentes (CPU, RAM, armazenamento etc.). Vamos usar o padrão **Builder** para permitir a montagem flexível de computadores.

Classe Computador:

```
public class Computador {
    private String processador;
    private String memoria;
    private String armazenamento;

    public void setProcessador(String processador) {
        this.processador = processador;
    }

    public void setMemoria(String memoria) {
        this.memoria = memoria;
    }

    public void setArmazenamento(String armazenamento) {
        this.armazenamento = armazenamento;
    }

    @Override
    public String toString() {
        return "Computador [Processador=" + processador + ", Memória=" + memoria + ",
Armazenamento=" + armazenamento + "]";
    }
}
```

Classe ComputadorBuilder:

```
public class ComputadorBuilder {
    private Computador computador;
```

```
    public ComputadorBuilder() {
        computador = new Computador();
    }

    public ComputadorBuilder comProcessador(String processador) {
        computador.setProcessador(processador);
        return this;
    }

    public ComputadorBuilder comMemoria(String memoria) {
        computador.setMemoria(memoria);
        return this;
    }

    public ComputadorBuilder comArmazenamento(String armazenamento) {
        computador.setArmazenamento(armazenamento);
        return this;
    }

    public Computador construir() {
        return computador;
    }
}
```

Classe Main

```
public class Main {
    public static void main(String[] args) {
        Computador computador = new ComputadorBuilder()
                .comProcessador("Intel i9")
                .comMemoria("32GB RAM")
                .comArmazenamento("1TB SSD")
                .construir();

        System.out.println(computador);
    }
}
```

COMPOSITE

O padrão **Composite** é um padrão estrutural que permite tratar objetos individuais e composições de objetos de maneira uniforme. Ele é usado quando você tem uma estrutura de objetos hierárquica e precisa tratar todos os objetos da mesma maneira, independentemente de serem objetos simples ou compostos.

Interface Componente

```
public interface Componente {
    void exibir();
}
```

Classe Arquivo

```
public class Arquivo implements Componente {
    private String nome;

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

    @Override
    public void exibir() {
        System.out.println("Arquivo: " + nome);
    }
}
```

Classe Diretorio

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

public class Diretorio implements Componente {
    private String nome;
    private List<Componente> componentes = new ArrayList<>();

    public Diretorio(String nome) {
        this.nome = nome;
    }
```

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

    @Override
    public void exibir() {
        System.out.println("Diretório: " + nome);
        for (Componente componente : componentes) {
            componente.exibir();
        }
    }
}
```

Classe Main

```
public class Main {
    public static void main(String[] args) {
        Arquivo arquivo1 = new Arquivo("arquivo1.txt");
        Arquivo arquivo2 = new Arquivo("arquivo2.txt");

        Diretorio diretorio = new Diretorio("diretorio1");
        diretorio.adicionarComponente(arquivo1);
        diretorio.adicionarComponente(arquivo2);

        Diretorio subDiretorio = new Diretorio("subdiretorio1");
        subDiretorio.adicionarComponente(new Arquivo("subarquivo1.txt"));

        diretorio.adicionarComponente(subDiretorio);

        diretorio.exibir();
    }
}
```

CHAIN OF RESPONSIBILITY

O padrão **Chain of Responsibility** é um padrão comportamental que permite passar uma solicitação por uma cadeia de manipuladores até que um deles a trate. Ele é útil quando você tem uma série de manipuladores e quer que cada um tenha a oportunidade de processar ou passar adiante a solicitação.

Classe Atendente:

```
public abstract class Atendente {
    protected Atendente proximoAtendente;

    public void setProximoAtendente(Atendente proximoAtendente) {
        this.proximoAtendente = proximoAtendente;
    }

    public abstract void atenderPedido(String tipoDeProblema);
}
```

Classe Main:

```
public class Main {
    public static void main(String[] args) {
        Atendente atendente1 = new AtendenteNivel1();
        Atendente atendente2 = new AtendenteNivel2();
        Atendente atendente3 = new AtendenteNivel3();

        atendente1.setProximoAtendente(atendente2);
        atendente2.setProximoAtendente(atendente3);

        // Testando os atendentes com diferentes tipos de problema
        atendente1.atenderPedido("problema simples");
        atendente1.atenderPedido("problema médio");
        atendente1.atenderPedido("problema complexo");
    }
}
```

CONCLUSÃO

Neste trabalho, exploramos a aplicação de três **Design Patterns** fundamentais nas áreas **Criacional**, **Estrutural** e **Comportamental**, com o objetivo de resolver problemas típicos do desenvolvimento de software de maneira eficiente, flexível e escalável. Cada padrão foi aplicado em um exemplo prático, com a finalidade de demonstrar sua utilidade na construção de sistemas modulares e de fácil manutenção.

1. Padrão Criacional - Builder

O **Builder** foi utilizado para a criação de um **computador personalizado**, permitindo a montagem de um objeto complexo (o computador) de forma clara e incremental, sem sobrecarregar o construtor com múltiplos parâmetros. Este padrão mostrou como facilitar a criação de objetos com configurações variáveis e complexas, promovendo uma interface fluida e simplificada.

2. Padrão Estrutural - Composite

O **Composite** foi implementado em um **sistema de arquivos**, onde arquivos e diretórios são tratados de forma uniforme, permitindo que diretórios contenham tanto arquivos quanto outros diretórios. Esse padrão facilita a manipulação de estruturas hierárquicas complexas, tornando o código mais flexível e escalável. O exemplo evidenciou como o Composite pode ser útil ao trabalhar com dados estruturados em árvore, como sistemas de arquivos ou elementos de interface gráfica.

3. Padrão Comportamental - Chain of Responsibility

O **Chain of Responsibility** foi aplicado a um **sistema de suporte ao cliente**, onde diferentes níveis de atendentes resolvem problemas de complexidade crescente. Este padrão distribui a responsabilidade de processar solicitações entre uma cadeia de objetos, permitindo que cada atendente trate um tipo específico de problema e passe o caso adiante se não puder resolvê-lo. O exemplo ilustrou como esse padrão facilita a organização de fluxos de trabalho e a delegação de tarefas, promovendo maior flexibilidade na resolução de problemas.

A aplicação desses três **Design Patterns** no trabalho permitiu uma compreensão mais profunda de como os padrões de projeto ajudam a estruturar e organizar o código de maneira eficaz. O padrão **Builder** contribuiu para a criação de objetos complexos de forma modular e sem complexidade excessiva. O **Composite** facilitou o trabalho com hierarquias de objetos, como no exemplo do sistema de arquivos, enquanto o **Chain of Responsibility** mostrou como distribuir responsabilidades de forma flexível e escalável em sistemas que requerem tratamento de diferentes tipos de solicitações.

Esses padrões de design não só melhoram a **manutenção**, **escalabilidade** e **flexibilidade** do software, como também ajudam a resolver problemas de maneira mais elegante e eficaz. O entendimento desses padrões proporciona uma base sólida para o desenvolvimento de sistemas mais robustos, adaptáveis e de fácil evolução, sendo essencial para o desenvolvimento de software de alta qualidade.

ESTUDO

https://www.opus-software.com.br/insights/design-patterns/

https://www.javatpoint.com/builder-design-pattern

https://www.devmedia.com.br/design-patterns-em-net-composite-e-chain-of-responsibility/17691