

Atividade 4 - ATIVIDADE 4 – Visao_Analise_Projetos_Design_e_Padrões_2023_2			
Curso: Engenharia Software/ADS	UC: Modelos, Métodos e Técnicas Engenharia de Software		
Período: Turma: N	Semestre: 2	Ano letivo: 2023-2	Turno: Noite
Professor(a): Rubem Koide			
Aluno(a): AUGUSTO DOMICIANO JEANDER KAUAN PRZYBEUKA ALLAN KLYNSMAMM GUILHERME SAIKI			

Link GITHUB: <https://github.com/augustoDomiciano/controlededefrotas>

4.1 – Testar códigos para dois padrões do Criação (um para classe e um para objeto) e explicar o funcionamento dos padrões;

FACTORY

Código:

/**

* Engenharia de Software Moderna – Padrões de Projeto (Cap. 6)

* Prof. Marco Tulio Valente

```
*  
  
* Exemplo do padrão de projeto Fábrica  
  
*  
  
*/  
  
  
/**  
  
* Interface e classe dos objetos que serão fabricados.  
  
*/  
  
  
interface Channel {}  
  
  
class TCPChannel implements Channel {}  
  
  
class UDPChannel implements Channel {}  
  
  
  
  
  
/**  
  
* A classe ChannelFactory implementa um método fábrica  
estático.  
  
* Isto é, um método que centraliza a criação de objetos que  
* implementam a interface Channel  
  
*  
  
* Se amanhã quisermos que o sistema use UDPChannel, basta  
* mudar a implementação de create()  
  
*/
```

```
class ChannelFactory {

    public static Channel create() { // método fábrica estático

        System.out.println("Neste momento, estamos trabalhando com
TCPChannel");

        return new TCPChannel();

    }

}

public class Main {

    void f() {

        Channel c = ChannelFactory.create();

    }

    void g() {

        Channel c = ChannelFactory.create();

    }

    void h() {

        Channel c = ChannelFactory.create();

    }

    public static void main(String [] args) {

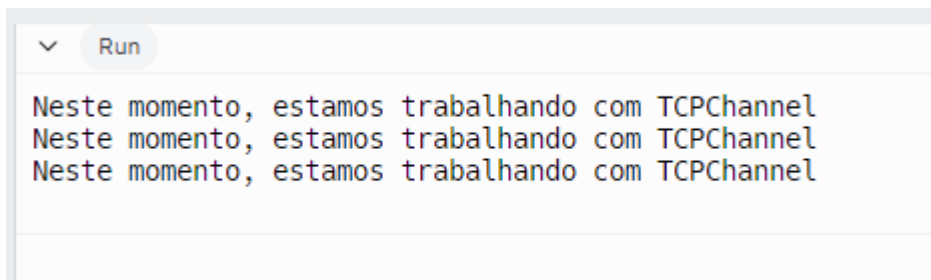
        Main m = new Main();

    }

}
```

```
m.f();  
  
m.g();  
  
m.h();  
  
}  
  
}
```

Resultado:



Explicação:

Esse padrão de projeto é utilizado quando não se sabe ao certo quais as dependências que serão utilizadas no futuro, ele é bom, pois possibilita bastante flexibilidade.

Por exemplo, no código acima, primeiro é criado uma interface (channel) no qual os objetos serão criados. Depois são criados as subclasses TCP e UDP channel, que implementam a interface Channel. A superclasse é a `ChannelFactory`, que executa os métodos e retorna a subclasse. Nesse caso foi criado um retorno para TCP, para retornar UDP, simplesmente seria necessário, alterar o retorno da chamada da `ChannelFactory`, de `Return TCPChannel` para `UDPChannel`.

Isso facilita na adaptabilidade do código e futuras alterações.

Builder

Código:

```
/**
 * Engenharia de Software Moderna - Padrões de Projeto (Cap. 6)
 * Prof. Marco Tulio Valente
 *
 * Exemplo do padrão de projeto Builder
 *
 */

class Livro {

    private String nome;

    private String autores;

    private String editora;

    private String ano;

    private Livro (String nome, String autores, String editora,
String ano) {

        this.nome = nome;

        this.autores = autores;

        this.editora = editora;

        this.ano = ano;

    }
```

```

public String toString() {

    return nome + ". " + autores + ", " + editora + ", " + ano;

}

/**

 * Livro.Builder é uma classe interna, pública e estática de
Livro.

 * Por isso, é que podemos chamar "new Livro.Builder()"
diretamente,

 * sem precisar de instanciar antes um objeto do tipo Livro.

 *

 */

public static class Builder {

    private String nome;

    private String autores;

    private String editora;

    private String ano;


    public Builder setName(String nome) {

        this.nome = nome;

        return this;

    }


    public Builder setAutores(String autores) {

        this.autores = autores;

```

```

        return this;
    }

    public Builder setEditora(String editora) {
        this.editora = editora;
        return this;
    }

    public Builder setAno(String ano) {
        this.ano = ano;
        return this;
    }

    public Livro build() {
        return new Livro(nome, autores, editora, ano);
    }
}

}

public class Main {

    public static void main(String [] args) {

        Livro esm = new Livro.Builder()

            .setNome("Engenharia Soft Moderna")

```

```
        .setEditora("UFMG")

        .setAno("2020")

        .build();

System.out.println("Livro 1: " + esm.toString());

    Livro gof = new Livro.Builder()

        .setNome("Design Patterns")

        .setAutores("GoF")

        .setAno("1995")

        .build();

System.out.println("Livro 2: " + gof.toString());

}

}
```

Resultado:

Livro 1: Engenharia Soft Moderna. null,UFMG,2020

Livro 2: Design Patterns. GoF,null,1995

Explicação:

Permite fazer a criação de objetos complexos, esses podendo ser longo e extenso. O correto seria fazer todas as especificações que este deveria ter, e assim fazendo a criação e complemento.

Fazendo vários tipos de representação, mas usando o mesmo código.

Começa com o objeto principal ex: Livro, os próximos passos são as suas subclasses, estes podendo ser Nome, ano, editora, autor e etc.

Mas esse código inicial poderia ser usado para outros livros, assim fazendo a criação de vários outros e enriquecendo o objeto.

4.2 - Testar códigos para dois padrões de Estrutura (um para classe e um para objeto) e explicar o funcionamento dos padrões;

ADAPTER

Código:

```
/**
 * Engenharia de Software Moderna - Padrões de Projeto (Cap. 6)
 * Prof. Marco Tulio Valente
 *
 * Exemplo do padrão de projeto Adaptador
 *
 */

/**
 * Classe concreta, representando um projetor da Samsung
 */
class ProjetorSamsung {
```

```

    public void turnOn() {

        System.out.println("Ligando projetor da Samsung");

    }

}

/**
 * Classe concreta, representando um projetor da LG
 */
class ProjetorLG {

    public void enable(int timer) {

        System.out.println("Ligando projetor da LG em " + timer + "
minutos");

    }

}

/**
 * Interface para "abstrair" o tipo de projetor (Samsung ou LG)
 */
interface Projetor {

    void liga();

}

```

```

/**
 * Adaptador de ProjetorSamsung para Projetor
 *
 * Um objeto da classe a seguir é um Projetor (pois implementa essa
 * interface),
 *
 * mas internamente repassa toda chamada de método para o objeto
 * adaptado
 *
 * (no caso, um ProjetorSamssung)
 */
class AdaptadorProjetorSamsung implements Projetor {

    private ProjetorSamsung projetor;

    AdaptadorProjetorSamsung (ProjetorSamsung projetor) {
        this.projetor = projetor;
    }

    public void liga() {
        projetor.turnOn(); // chama método do objeto adaptado
        (ProjetorSamsung)
    }

}

/**
 * Idem classe anterior, mas agora adaptando ProjetoLG para
 * Projetor
 */

```

```
class AdaptadorProjektorLG implements Projektor {

    private ProjektorLG projetor;

    AdaptadorProjektorLG (ProjektorLG projetor) {

        this.projetor = projetor;

    }

    public void liga() {

        projetor.enable(0); // chama método de objeto adaptado
        (ProjektorLG)

    }

}

class SistemaControleProjetores { // não tem conhecimento de
    "projetores concretos"

    void init(Projektor projetor) {

        projetor.liga(); // liga qualquer projetor, sem precisar
        saber se é Samsung ou LG

    }

}

class Main {
```

```

    public static void main(String[] args) {

        AdaptadorProjetoSamsung samsung = new
        AdaptadorProjetoSamsung(new ProjetoSamsung());

        AdaptadorProjetoLG lg = new AdaptadorProjetoLG(new
        ProjetoLG());

        SistemaControleProjetores scp = new
        SistemaControleProjetores();

        scp.init(samsung); // recebem como parâmetros objetos
        adaptadores,

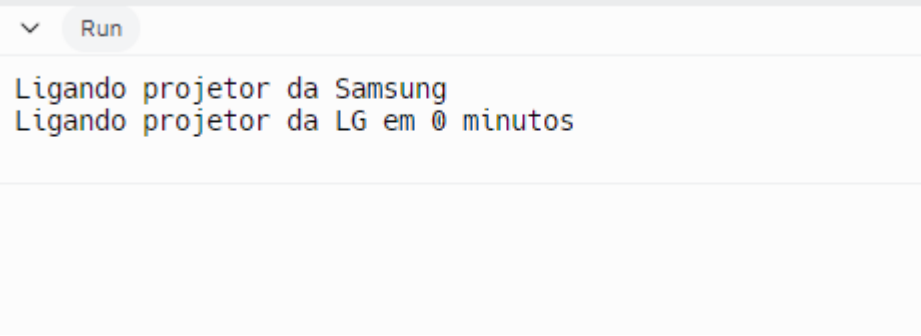
        scp.init(lg);      // que possuem internamente objetos (isto
        é, projetores) concretos

    }

}

```

Resultado:



```

  Run
  Ligando projetor da Samsung
  Ligando projetor da LG em 0 minutos

```

Explicação:

Esse padrão de projeto auxilia na adaptação de de um objeto para outro, por exemplo, você tem um dado que recebe em xml, mas outra aplicação envia os dados em formato JSON, se você tiver uma aplicação que tem pronto para receber

dados em xml, você só precisaria criar esse adaptador, não precisando recriar todo o código.

No caso do código acima, são criadas duas classes distintas, uma para cada marca de projetor. Depois é criada uma interface, na qual, as classes adaptadoras irão se basear. As classes adaptadoras utilizam a interface. A classe main cria objetos dos adaptadores da Samsung e da LG, que liga o objeto sem precisar saber de fato qual objeto é.

Object Adapter

Código:

```
/**  
 * Engenharia de Software Moderna - Padrões de Projeto (Cap. 6)  
 * Prof. Marco Tulio Valente  
 *  
 * Exemplo do padrão de projeto Adaptador  
 */
```

```
/**  
 * Classe concreta, representando um projetor da Samsung  
 */  
  
class ProjetorSamsung {  
  
    public void turnOn() {
```

```

        System.out.println("Ligando projetor da Samsung");
    }

}

/**
 * Classe concreta, representando um projetor da LG
 */
class ProjetorLG {

    public void enable(int timer) {

        System.out.println("Ligando projetor da LG em " + timer +
" minutos");

    }

}

/**
 * Interface para "abstrair" o tipo de projetor (Samsung ou LG)
 */
interface Projetor {

    void liga();

}

/**

```

```

* Adaptador de ProjetorSamsung para Projetor

* Um objeto da classe a seguir é um Projetor (pois implementa
essa interface),

* mas internamente repassa toda chamada de método para o
objeto adaptado

* (no caso, um ProjetorSamssung)

*/

class AdaptadorProjetorSamsung implements Projetor {

    private ProjetorSamsung projetor;

    AdaptadorProjetorSamsung (ProjetorSamsung projetor) {
        this.projetor = projetor;
    }

    public void liga() {
        projetor.turnOn(); // chama método do objeto adaptado
        (ProjetorSamsung)
    }

}

/**

* Idem classe anterior, mas agora adaptando ProjetoLG para
Projetor

*/

```



```
class AdaptadorProjektorLG implements Projektor {

    private ProjektorLG projetor;

    AdaptadorProjektorLG (ProjektorLG projetor) {

        this.projetor = projetor;

    }

    public void liga() {

        projetor.enable(0); // chama método de objeto adaptado
        (ProjektorLG)

    }

}

class SistemaControleProjetores { // não tem conhecimento de
    "projetores concretos"

    void init(Projektor projetor) {

        projetor.liga(); // liga qualquer projetor, sem precisar
        saber se é Samsung ou LG

    }

}

class Main {
```

```

    public static void main(String[] args) {

        AdaptadorProjetoSamsung samsung = new
        AdaptadorProjetoSamsung(new ProjetoSamsung());

        AdaptadorProjetoLG lg = new AdaptadorProjetoLG(new
        ProjetoLG());

        SistemaControleProjetores scp = new
        SistemaControleProjetores();

        scp.init(samsung); // recebem como parâmetros objetos
        adaptadores,

        scp.init(lg);      // que possuem internamente objetos
        (isto é, projetores) concretos

    }

}

```

Resultado:

Ligando projetor da Samsung

Ligando projetor da LG em 0 minutos

Explicação:

Usado para adaptar duas classes incompatíveis, por exemplo um fornecedor de dados XML porém o receptor só aceita JSON, então será feito um código de adaptação, onde fará a transformação do XML para JSON. O bom seria que não precisasse fazer modificação no código, visando que poderia prejudicar o código inicial, ou talvez não ter acesso ao código de uma categoria específica. Utilizando esse padrão pode ser feita a adaptação sem danificar o código.

4.3 - Testar códigos para dois padrões Comportamental (um para classe e um para objeto) e explicar o funcionamento dos padrões.

Template Method

Código:

```
/**
 * Engenharia de Software Moderna - Padrões de Projeto (Cap. 6)
 * Prof. Marco Tulio Valente
 *
 * Exemplo do padrão de projeto Template Method
 *
 */

/**
 * Classe que implementa um Template Method
 * (calcSalarioLiquido)
 * Veja que essa classe é abstrata
 */
abstract class Funcionario {

    protected double salario;
```

```

public Funcionario(double salario) {

    this.salario = salario;

}

abstract double calcDescontosPrevidencia();

abstract double calcDescontosPlanoSaude();

abstract double calcOutrosDescontos();

/**

* Template Method: define o esqueleto de um algoritmo

* Ele ainda é um "template" porque os métodos chamados são
abstratos

*/

public double calcSalarioLiquido() {

    double prev = calcDescontosPrevidencia();

    double saude = calcDescontosPlanoSaude();

    double outros = calcOutrosDescontos();

    return salario - prev - saude - outros;

}

}

/**

* Subclasse que implementa os métodos abstratos chamados pelo
Template Method

* Ela vai herdar o template method (calcSalarioLiquido)

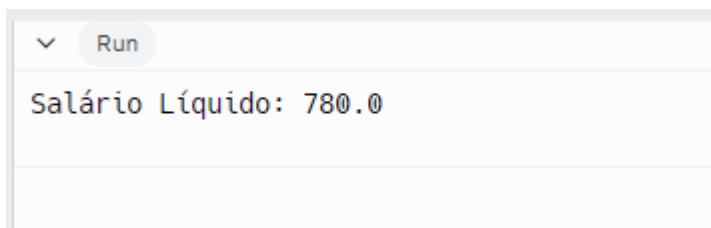
* Mas vai ter que implementar os métodos abstratos chamados
por ele

```

```
*/  
  
class FuncionarioCLT extends Funcionario {  
  
    public FuncionarioCLT(double salario) {  
        super(salario);  
    }  
  
    // implementa método abstrato  
    double calcDescontosPrevidencia() {  
        return salario * 0.1;    // somente um exemplo  
    }  
  
    // implementa método abstrato  
    double calcDescontosPlanoSaude() {  
        return 100.0;  
    }  
  
    // implementa método abstrato  
    double calcOutrosDescontos() {  
        return 20.0;  
    }  
  
}  
  
class Main {
```

```
public static void main(String[] args) {  
  
    FuncionarioCLT func = new FuncionarioCLT(1000);  
  
    double salario = func.calcSalarioLiquido();  
  
    System.out.println("Salário Líquido: " + salario);  
  
}  
  
}
```

Resultado:



The screenshot shows a Java IDE's output window. At the top, there is a dropdown menu with a downward arrow and a button labeled 'Run'. Below this, the output text 'Salário Líquido: 780.0' is displayed in a monospaced font.

Explicação:

Esse tipo de padrão define uma classe base fixa, mas permite alterações nas subclasses, sem alterar as estruturas globais. A classe abstrata `Funcionário`, contém alguns templates methods. Os métodos abstratos são implementados nas subclasses concretas. A subclasse `FuncionárioCLT` implementa o template method de `Funcionário` e tem implementações específicas para as operações abstratas. A classe `Main` instância um `FuncionárioCLT` com um determinado salário, e é calculado o salário Líquido, baseado no template method que tem as operações primitivas, como desconto da previdência e plano de saúde.

Visitor

Código:

```
/**
 * Engenharia de Software Moderna - Padrões de Projeto (Cap. 6)
 * Prof. Marco Tulio Valente
 *
 * Exemplo do padrão de projeto Visitor
 *
 */

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

/**
 * Veiculo é a raiz de uma hierarquia de classes
 *
 * Todas as classes dessa hierarquia aceitam (método accept)
 * visitas de objetos "Visitor"
 *
 * Ou seja, Veiculos e suas subclasses estão abertas para tais
 * visitas
 *
 * Mas elas não sabem exatamente o que o Visitor vai fazer com
 * o seus dados
 *
 */

abstract class Veiculo {
    private String placa;
```

```
public Veiculo(String placa) {  
    this.placa = placa;  
}  
  
public String getPlaca() {  
    return placa;  
}  
  
abstract public void accept(Visitor v);  
  
}  
  
class Carro extends Veiculo {  
  
    public Carro (String placa) {  
        super(placa);  
    }  
  
    public void accept(Visitor v) {  
        v.visit(this); // compilador já conhece o tipo de this  
        (= Carro)  
    } // porém, a chamada é dinâmica, pois diversas classes  
    podem implementar a interface Visitor
```



```
}
```

```
class Onibus extends Veiculo {
```

```
    public Onibus (String placa) {
```

```
        super(placa);
```

```
    }
```

```
    public void accept(Visitor v) {
```

```
        v.visit(this);
```

```
    }
```

```
}
```

```
/**
```

```
 * A interface Visitor deve ser implementada por classes  
visitantes
```

```
 *
```

```
 */
```

```
interface Visitor {
```

```
    void visit(Carro c);
```

```
    void visit(Onibus o);
```

```
}
```

```
/**
```

```
 * PrintVisitor é uma classe visitante
```

```
* Ela imprime a placa de Veiculos concretos (isto é, Carro e
Onibus) na tela
```

```
*
```

```
*/
```

```
class PrintVisitor implements Visitor {
```

```
    public void visit(Carro c) {
```

```
        System.out.println("Visitando um Carro com placa: " +
c.getPlaca());
```

```
    }
```

```
    public void visit(Onibus o) {
```

```
        System.out.println("Visitando um Onibus com placa: " +
o.getPlaca());
```

```
    }
```

```
}
```

```
public class Main {
```

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

```
        List<Veiculo> list = new ArrayList<Veiculo>();
```

```
        list.add(new Carro("GHJ-1020"));
```

```
        list.add(new Onibus("BNM-3456"));
```

```
        list.add(new Carro("IOP-1234"));
```

```
list.add(new Onibus("BVC-7923"));

// Vamos "visitar", com um PrintVisitor, cada Veiculo da
lista

PrintVisitor visitor = new PrintVisitor();

for (Veiculo veiculo: list) {

    veiculo.accept(visitor);

}

// Benefício do padrão Visitor:

// Podemos implementar uma outra classe Visitor sem ter
que mexer na implementação

// da classe Veiculo e de suas subclasses. Em seguida,
podemos usar esse Visitor

// para visitar todos os veículos da nossa lista.

}

}
```

Resultado:

Visitando um Carro com placa: GHJ-1020

Visitando um Ônibus com placa: BNM-3456

Visitando um Carro com placa: IOP-1234

Visitando um Ônibus com placa: BVC-7923

Explicação:

Padrão utilizado para visualizar os objetos e seus complementos, ele ajuda a ver as especificações e informações importantes do objeto, assim auxiliando e ajudando o cliente para o entendimento.

Para ser usado, deve ser pontuado todas as especificações dos objetos, para assim fazer a apresentação.

O propósito desta solução, seria apresentar o mesmo objeto ex: carro, porém ainda assim mostrar suas particularidades por ex: placa, assim mesmo o objeto sendo o mesmo, na parte de visitante ainda conseguiria ver a diferença de cada, mesmo vindo do mesmo nome.