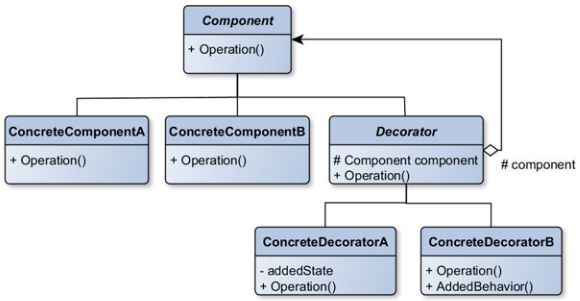
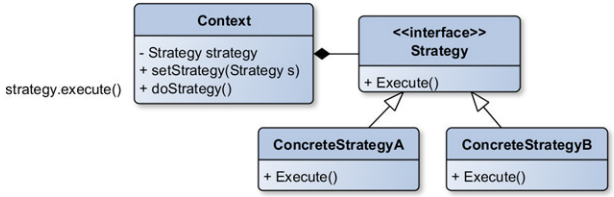


Padrão de Projeto: (DECORATOR)

Utilizado quando precisa-se anexar responsabilidades dinamicamente sem precisar de uma grande hierarquia de subclasses. ... O Decorator é mais utilizado quando quisermos adicionar responsabilidades a objetos dinamicamente, e quando a extensão por subclasses é impraticável, pois teríamos muitas alterações e dessa forma diversas subclasses. ..... **Dica:** Possíveis palavras chaves para você identificar o padrão decorator: "Incorporar", "Compor", "Acoplamento", "Juntar", "Mesclar", "Incluir", "Adicionar". ..... **Consequências? {A}** Mais flexibilidade do que herança (Adição ou remoção de responsabilidades em tempo de execução + Adição da mesma propriedade mais de uma vez) **{B}** Evita o excesso de funcionalidades nas classes **{C}** Decorator e seu componente não são idênticos **{D}** Comparações tornam-se mais complexas **{E}** Resulta em um design que tem vários pequenos objetos, todos parecidos

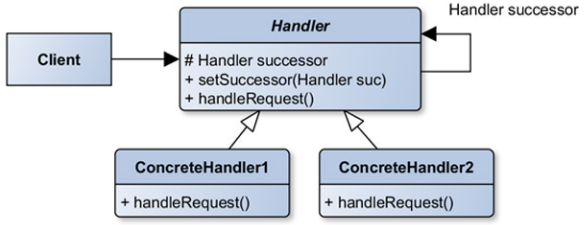
```java
public abstract class Pacote {
  private String desc;
  private Double preco;
  public Pacote(String desc, Double preco) {
    this.desc = desc;  this.preco = preco;
  }
  public String getDesc() { return desc; }
  public Double getPreco() { return preco; }
}
public class PacotePraia extends Pacote {
  public PacotePraia(String desc, Double preco) {
    super("Pacote para praia: " + desc, preco);
  }
}
public abstract class PacoteDecorator extends Pacote {
  protected Pacote pacote;
  public PacoteDecorator(Pacote pacote, String descServ, Double precoServ) {
    super(pacote.getDesc() + " + " + descServ, pacote.getPreco() + precoServ);
    this.pacote = pacote;
  }
}
public class ServicoBebidas extends PacoteDecorator {
  public ServicoBebidas(Pacote pacote) {
    super(pacote, "Bebidas", 10.0);
  }
}
public class Main { public static void main(String[] args) {
    System.out.println("== PACOTE PRAIA (SANTOS-$500): INCLUI BEBIDA ($10) + MASSAGEM ($50)");
    Pacote pacotePraia =
        new ServicoMassagem( new ServicoBebidas( new PacotePraia("Santos", 500.0) ) );
    System.out.println("DESCRIÇÃO " + pacotePraia.getDesc());
    System.out.println("TOTAL R$ " + pacotePraia.getPreco());
    /* console:
     *  == PACOTE PRAIA (SANTOS-$500): INCLUI BEBIDA ($10) + MASSAGEM ($50)
     *  DESCRIÇÃO Pacote para praia: Santos + Bebidas + Serviço de Massagem
     *  TOTAL R$ 560.0
     */ }
```



Padrão de Projeto (STRATEGY)

Define uma família de algoritmos, encapsula e os torna intercambiaveis (variáveis). O Strategy é utilizado quando você tem um determinado algoritmo, rotina ou algo deste tipo, e que pode mudar em determinadas ocasiões. Suponhamos que você por exemplo tem uma classe de cálculo de juros e que em uma determinada data do ano, a taxa de juros diminui por conta de uma promoção. Então em cenários como este você, utilizaria o Strategy para auxiliar na solução desta demanda sem causar grande impacto para efetuar a mudança. ****Principio***: Encapsule o que varia.* **Consequências? {A}** Não fere a regra de aberto e fechado. **{B}** Facilidade ao debugar. **{C}** Não cresce esponencialmente

```java
public abstract class Produto {
  private String nome;
  private double preco;
  private Promocao promocao;
  public void setPromocao(Promocao promocao) { this.promocao = promocao; }
  public Produto(String nome, double preco) {
    this.nome = nome; this.preco = preco;
  }
  public double calcularPreco(){ return promocao.descontar() * preco; }
  public String toString(){ return "Produto: " + nome; }
}
public final class Brinquedo extends Produto{
  public Brinquedo(String nome, double preco) { super(nome, preco); }
}
public interface Promocao {
  double descontar();
}
public class PromocaoRegular implements Promocao{
  private double extra;
  public Regular(double extra){
    // desconta cada produto em 10% mais um desconto extra varia de 5% a 10%
    if(extra <= 0.90 && extra >= 0.95) { this.extra = extra; }
    else { this.extra = 1.0; }
  }
  // volta $100*0.9 = $90 (desconta 10%+extra)
  public double descontar() {  return 0.9*extra; }
}
```

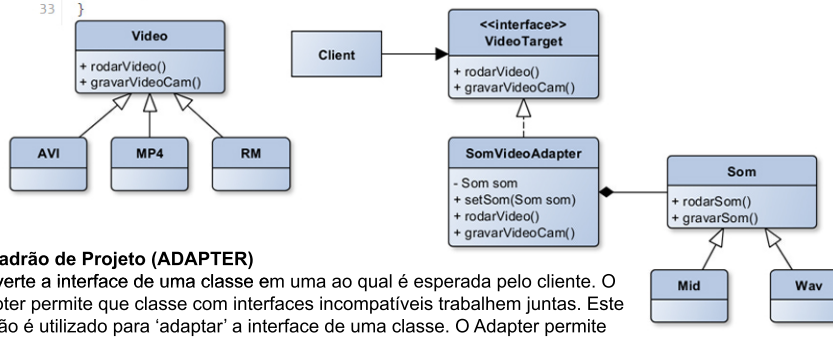


Padrão de Projeto (CHAIN of Responsibility)

Evita **Acoplamento** (com if) entre o "Sender" de uma requisição Z, o recebedor dando a chance de mais de um objeto efetuar o tratamento. A cadeia de objetos trata a requisição conforme alguma requisição, caso não consiga o próximo elemento fica com a responsabilidade do tratamento.
> *Decorator, Chain e Strategy tem polimofismo*
Estrutura: Existem três partes do padrão Chain of Responsibility: sender, receiver e request. O sender faz o request. O receiver é uma cadeia de um ou mais objetos que escolhe se quer lidar com o request ou transmiti-lo. O request em si pode ser um objeto que encapsula todos os dados apropriados.
Consequências? {A} Fornece um acoplamento mais fraco por evitar a associação explícita do remetente de uma solicitação ao seu receptor e dar a mais de um objeto a oportunidade de tratar a solicitação
Dica: Possíveis palavras chaves para você identificar o padrão CHAIN: "Passar ou Transferir Responsabilidade para o proximo", "Deixar o outro ou algo tentar, manusear, arcar, manobrar, controlar".

```java
// DesignPatterns-ChainOfResponsibility/src/VerbaAprovacaoBanco/Cargo.java
// HANDLER
abstract public class Cargo {
        protected Cargo suc;
        protected String nomeFunc;
        public Cargo(String nomeFunc){
                this.nomeFunc = nomeFunc;
        }
        public void setSuc(Cargo suc){
                this.suc = suc;
        }
        abstract public void aprovar(Verba v); // Polimorfismo
        // o this que esta dentro do syso mostra o esta aqui
        public String toString(){
                return this.getClass().getSimpleName();
        }
}
//DesignPatterns-ChainOfResponsibility/src/VerbaAprovacaoBanco/Gerente.java
public class Gerente extends Cargo {
        public Gerente(String nomeFunc){
                super(nomeFunc);
        }
        @Override
        public void aprovar(Verba v) {
                if(v.getValor() <= 80000 && v instanceof Normal){
                        // this foi usado por causa do toString do Cargo
                        System.out.println("Verba de " + v.getValor() +
                        " aprovada por " + nomeFunc + " cargo: " + this );
                }else {
                        suc.aprovar(v);
                }
        }
}
```

```java
Gerente gerent = new Gerente("Ernesto"); // cargos
Superintendente superi = new Superintendente("Cadu");
Vp vp = new Vp("Afonso");
Ceo ceo = new Ceo("Luiz");
gerent.setSuc(superi); // sets
superi.setSuc(vp);
vp.setSuc(ceo);
Importante verbImpor = new Importante(50000); // verba
gerent.aprovar(verbImpor); // aprovar
```



Padrão de Projeto (ADAPTER)

Converte a interface de uma classe em uma ao qual é esperada pelo cliente. O Adapter permite que classe com interfaces incompatíveis trabalhem juntas. Este padrão é utilizado para 'adaptar' a interface de uma classe. O Adapter permite que classes com interfaces incompatíveis possam interagir. Adapter permite que um objeto cliente utilize serviços de outros objetos com interfaces diferentes por meio de uma interface única. Ou seja, dado um conjunto de classes com mesma responsabilidade, mas interfaces diferentes, utilizamos o Adapter para unificar o acesso a qualquer uma delas.". > Imita o DuckType

```java
// DesignPatterns-Adapter/src/PlayerVideo/VideoTarget.java
public interface VideoTarget {
        void rodarVideo();
        void gravarVideoCam();
}
// DesignPatterns-Adapter/src/PlayerVideo/SomVideoAdapter.java
public class SomVideoAdapter implements VideoTarget {
        private Som som;
        public SomVideoAdapter(Som som){ this.som = som; } // setSom()
        @Override public void rodarVideo(){ som.rodarSom(); }
        @Override public void gravarVideoCam(){ som.gravarSom(); }
        ## System.out.println("Rodando som... " + this.getClass().getSimpleName());
}
// DesignPatterns-Adapter/src/PlayerVideo/Main.java
public class Main {
        public static void main(String[] args) {
                Avi videoAvi = new Avi();
                videoAvi.rodarVideo();
                SomVideoAdapter somMidAdaptadoVideo = new SomVideoAdapter( new Mid() );
                somMidAdaptadoVideo.rodarVideo();
        }
}
```