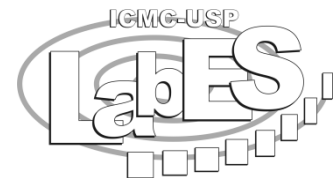




Design Patterns

Lucas Bueno Ruas de Oliveira
Profa. Elisa Yumi Nakagawa

SSC 122 – Engenharia de Software II
1. Semestre 2010



Breve Revisão

Métodos

Atributos

<i>Classe</i>
+ variavelPublica : int - variavelPrivada : float # variavelProtegida : Date <u>- variavelEstatica : int</u>
+ metodoPublico(primitivoA : int, primitivoB : float) : void - metodoPrivado(objeto : Date) : float # metodoProtegido() : void + <i>metodoAbstrato() : void</i>

+ metodoAbstrato() : void # metodoProtegido() : void - metodoPrivado(objeto : Date) : float

Tipos de acesso

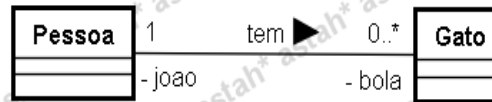
Breve Revisão



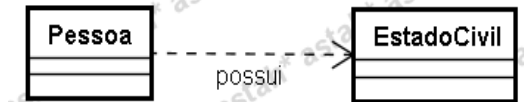
Herança



Interface



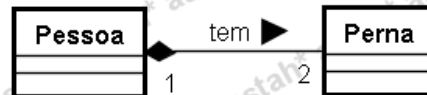
Associação



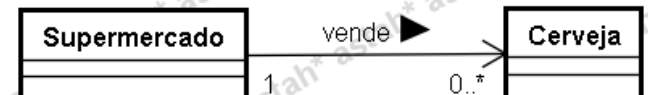
Dependência



Agregação



Composição



Associações Unidimensionais

Breve Revisão

camelCase

Pojo
- variavelBooleana : boolean - outraVariavel : int
+ Pojo() : void + isVariavelBooleana() : boolean + setVariavelBooleana(var : boolean) : void + getOutraVariavel() : int + setOutraVariavel(var : int) : void

POJO

Bar
- listaDeCoisas : List<Coisa>
+ sizeOfListaDeCoisas() : int + iteratorListaDeCoisas() : Iterator<Coisa> + addListaDeCoisas(var : boolean) : void + removeListaDeCoisas(var : Coisa) : Coisa

Boas Práticas

Motivação

- Projetar software é tarefa difícil; projetar software com componentes reutilizáveis é mais difícil ainda.
- Um projeto deve ser específico o suficiente para resolver um problema, mas também genérico o bastante para poder ser reutilizado em outros projetos.

Definição

- Segundo Christopher Alexander (1977), cada padrão descreve um problema que acontece repetidamente.
- Apresentação da ideia de uma solução que pode ser utilizada diversas vezes sem ser empregada duas vezes da mesma forma. (Solução Genérica)

Estrutura

- Em resumo, um *Pattern* é composto por quatro partes:
 - Um Nome que representa a essência do Problema.
 - O Problema que descreve quando o *pattern* é aplicado e em que contexto está inserido.
 - A Solução que é composta de um modelo, suas entidades e o relacionamento entre elas.
 - As Consequências de seu uso, importante característica para determinar o custo-benefício

Classificação

- Os padrões já eram utilizados há muito tempo; porém, não havia uma documentação formal sobre eles.
- Gamma et al. (1995) definiram 23 padrões que possuem aplicação em diversas áreas.
- Os padrões são classificados em três grupos
 - Padrões de Estrutura
 - Padrões de Criação
 - Padrões de Comportamento

Padrões de Estrutura

- Auxiliam o programador na composição dos objetos.
- Proporcionam economia de código.
- Torna o entendimento do código mais fácil.

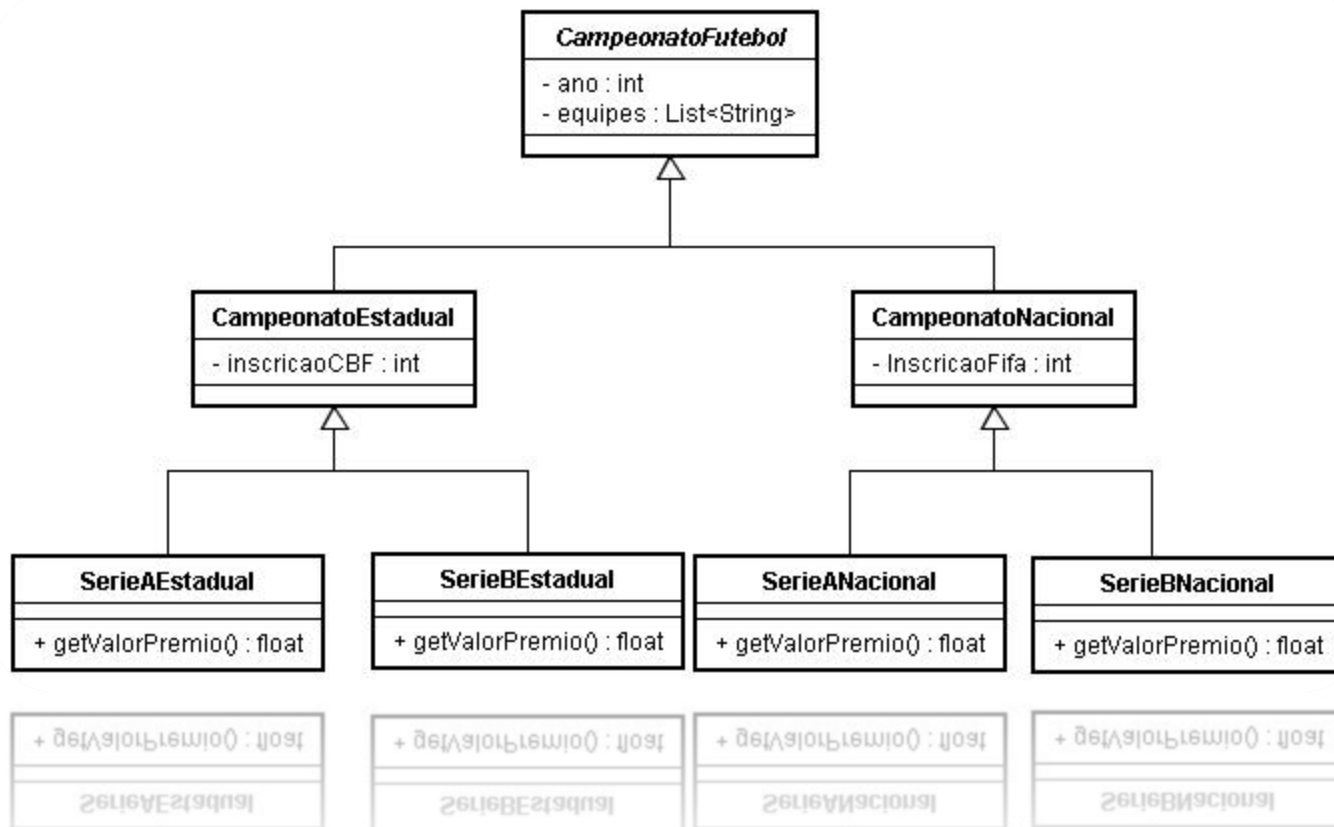
Padrões de Estrutura

- *Adapter* (Adaptador)
- *Bridge* (Ponte)
- *Composite* (Compositor)
- *Decorator* (Decorador)
- *Facade* (Fachada)
- *Flyweight*
- *Proxy* (Procurador)

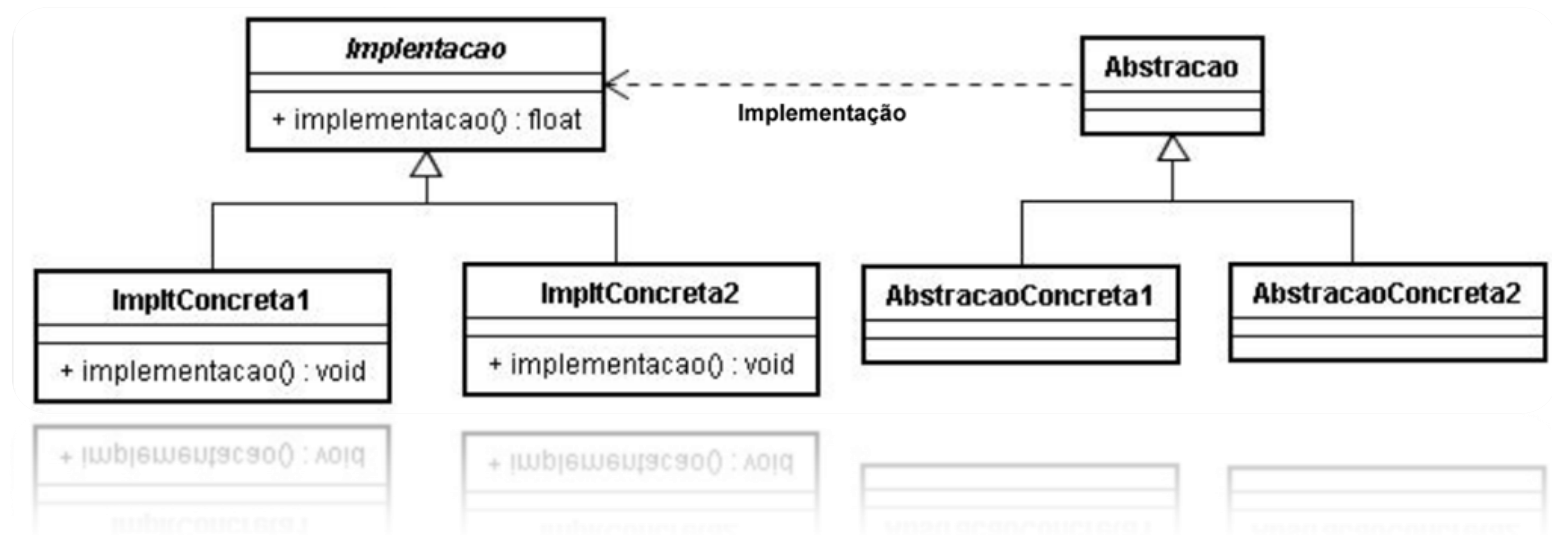
Bridge (Ponte) - Definição

- Esse padrão permite que a abstração de sua implementação possa variar de maneira independente

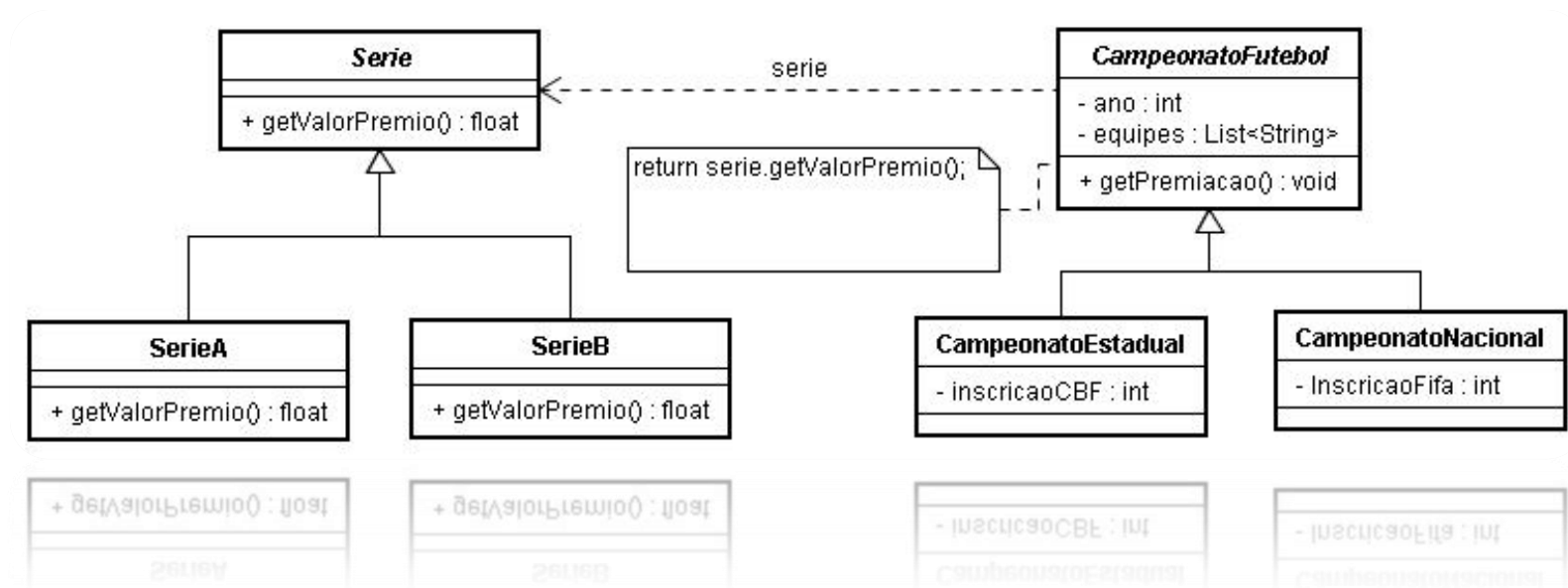
Bridge (Ponte) - Problema



Bridge (Ponte) - Solução



Bridge (Ponte) - Solução



Bridge (Ponte) - Solução

```
3 public abstract class Serie {  
4  
5     public abstract float getValorPremio(float patrocinio);  
6 }  
7
```

```
3 public class SerieA extends Serie{  
4  
5     //construtor SerialA  
6     public SerieA(){  
7  
8     }  
9     //retorna uma característica particular da série A  
10    public float getValorPremio(float patrocinio) {  
11        return patrocinio* 0.5f;  
12    }  
13  
14 }  
15
```

Bridge (Ponte) - Solução

```
3 public class SerieB extends Serie{
4
5     //construtor SerialB
6     public SerieB(){
7
8     }
9     //retorna uma característica particular da série B
10    public float getValorPremio(float patrocínio) {
11        return patrocínio* 0.1f;
12    }
13 }
```


Bridge (Ponte) - Solução

```
6 public abstract class CampeonatoFutebol{
7
8     private Serie serie;
9     private float patrocínio;
10
11 public CampeonatoFutebol(Serie serie) {
12     this.serie = serie;
13 }
14
15 public float getPremiacao(){return serie.getValorPremio(patrocínio);}
16
17 public float getPatrocínio() {return patrocínio;}
18
19 public void setPatrocínio(float patrocínio){this.patrocínio = patrocínio;}
20
21 }
22
```

Bridge (Ponte) - Solução

```
3 public class CampeonatoEstadual extends CampeonatoFutebol{
4
5     private int inscricaoCbf;
6
7     public CampeonatoEstadual(Serie serie){
8         super(serie);
9     }
10
11     public float getPatrocinio() {return super.getPatrocinio();}
12
13     public void setPatrocinio(float patrocinio)
14     {super.setPatrocinio(patrocinio);}
15
16     public int getInscricaoCbf(){return inscricaoCbf;}
17
18     public void setInscricaoCbf(int inscrCbf)
19     {inscricaoCbf = inscrCbf;}
20
21 }
22
```

Bridge (Ponte) - Solução

```
3 public class CampeonatoNacional extends CampeonatoFutebol{
4
5     private int inscricaoFifa;
6
7     public CampeonatoNacional(Serie serie){
8         super(serie);
9     }
10
11     public float getPatrocinio() {return super.getPatrocinio();}
12
13     public void setPatrocinio(float patrocinio)
14     {super.setPatrocinio(patrocinio);}
15
16     public int getInscricaoFifa() {return inscricaoFifa;}
17
18     public void setInscricaoFifa(int inscrFifa)
19     {this.inscricaoFifa = inscrFifa;}
20 }
```

```
3 public class App {
4
5     public static void main(String args[]){
6
7         SerieA serieA = new SerieA();
8         SerieB serieB = new SerieB();
9         //cria o campeonato nacional serie A
10        CampeonatoNacional campNasA = new CampeonatoNacional(serieA);
11        campNasA.setInscricaoFifa(1001);
12        campNasA.setPatrocinio(3000000);
13        //cria o campeonato nacional serie B
14        CampeonatoNacional campNasB = new CampeonatoNacional(serieB);
15        campNasB.setInscricaoFifa(1101);
16        campNasB.setPatrocinio(5000);
17        //cria o campeonato estadual serie A
18        CampeonatoEstadual campEstA = new CampeonatoEstadual(serieA);
19        campEstA.setInscricaoCbf(1011);
20        campEstA.setPatrocinio(10000);
21        //cria o campeonato estadual serie B
22        CampeonatoEstadual campEstB = new CampeonatoEstadual(serieA);
23        campEstB.setInscricaoCbf(1111);
24        campEstB.setPatrocinio(500);
25        System.out.println("Campeonato Nacional SerieA possui R$: "
26            +campNasA.getPremiacao()+ " em premios");
27        System.out.println("Campeonato Nacional SerieB possui R$: "
28            +campNasB.getPremiacao()+ " em premios");
29        System.out.println("Campeonato Estadual SerieA possui R$: "
30            +campEstA.getPremiacao()+ " em premios");
31        System.out.println("Campeonato Estadual SerieB possui R$: "
32            +campEstB.getPremiacao()+ " em premios");
33    }
34 }
```

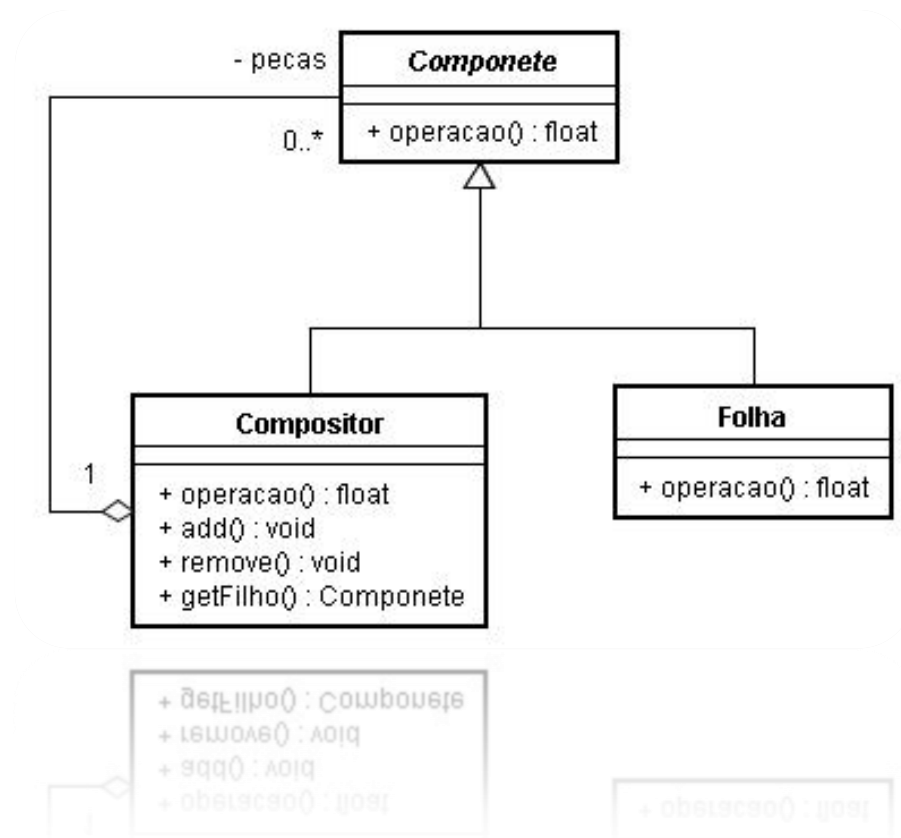
Bridge (Ponte) - Consequências

- É possível ocultar do cliente as regras de negócio
- Diminuição do número de linhas de código
- Redução de herança na modelagem

Composite (Compositor)- Problema

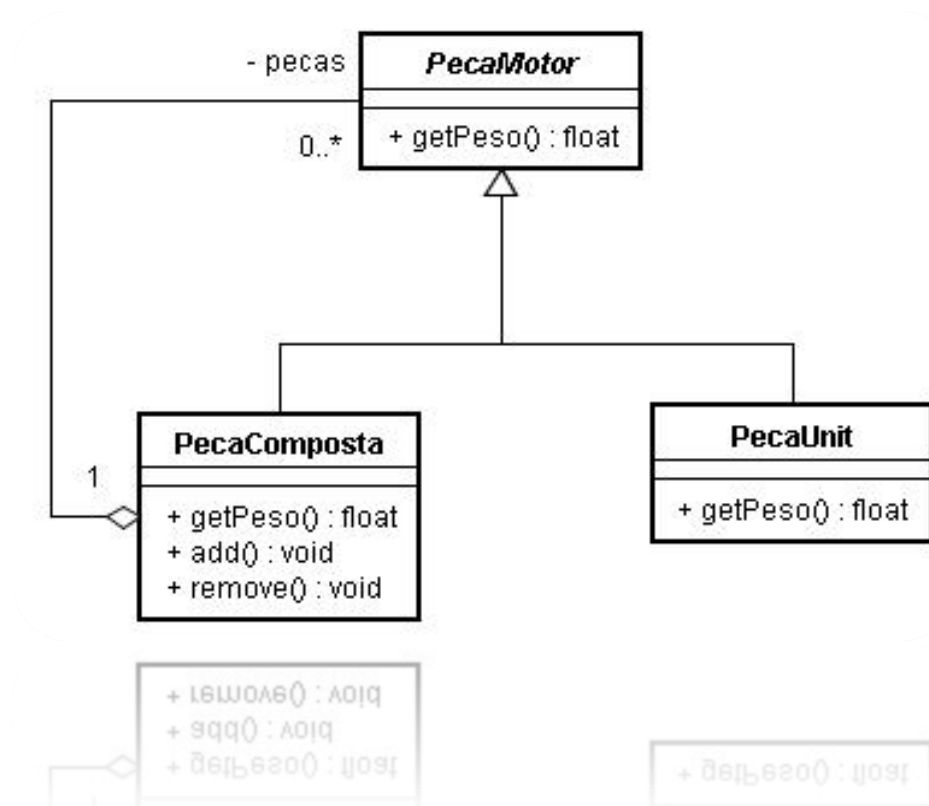
- Utilizado quando se deseja obter um tratamento uniforme de objetos que possuem uma estrutura do tipo “todo-parte”.
- Um exemplo de estrutura todo parte:
 - Um motor que possui um conjunto de peças, sendo essas compostas de outras peças menores

Composite (Compositor)- Solução



Composite (Compositor)- Exemplo

Quanto pesa este motor?



Composite (Compositor)- Exemplo

```
3 public class PecaMotor{
4
5     protected float peso;
6
7     public PecaMotor(){}
8
9     public PecaMotor(float peso){this.peso = peso;}
10
11     public float getPeso() {return peso;}
12
13     public void setPeso(float peso){this.peso = peso;}
14 }
```

Composite (Compositor)- Exemplo

```
7 public class PecaComposta extends PecaMotor{
8
9     private List<PecaMotor> pecas = new ArrayList<PecaMotor>();
10
11     public PecaComposta(){}
12
13     public boolean add(PecaMotor peca){return pecas.add(peca);}
14
15     public boolean remove(PecaMotor peca){return pecas.remove(peca);}
16
17     public float getPeso(){
18         float peso = 0;
19         for(PecaMotor pecaMotor : pecas){
20             peso += pecaMotor.getPeso();
21         }
22         return peso;
23     }
24 }
```

Composite (Compositor)- Exemplo

```
3 public class PecaUnit extends PecaMotor{
4
5     public PecaUnit() {}
6
7     public PecaUnit(float peso) {super(peso); }
8
9     public float getPeso() {return super.getPeso(); }
10
11     public void setPeso(float peso) {super.setPeso(peso); }
12 }
```

```
3 public class App {
4
5     public static void main(String args[]){
6
7         PecaUnit parafuso;
8         PecaUnit porca;
9         PecaComposta bloco = new PecaComposta();
10        //adiciona as sub-peças ao bloco do motor
11        for(int i = 0; i <= 70; ++i){
12            parafuso = new PecaUnit(0.08f);
13            porca = new PecaUnit(0.1f);
14            bloco.add(parafuso);
15            bloco.add(porca);
16        }
17        PecaComposta cabecote = new PecaComposta();
18        //adiciona as sub-peças ao cabeçote do motor
19        for(int i = 0; i <= 30; ++i){
20            parafuso = new PecaUnit(0.05f);
21            porca = new PecaUnit(0.15f);
22            cabecote.add(parafuso);
23            cabecote.add(porca);
24        }
25        PecaComposta motor = new PecaComposta();
26        motor.add(cabecote);
27        motor.add(bloco);
28        System.out.println("O cabeçote do motor pesa " + cabecote.getPeso() + "Kg");
29        System.out.println("O bloco do motor pesa " + bloco.getPeso() + "Kg");
30        System.out.println("O motor pesa " + motor.getPeso() + "Kg");
31
32    }
33 }
```

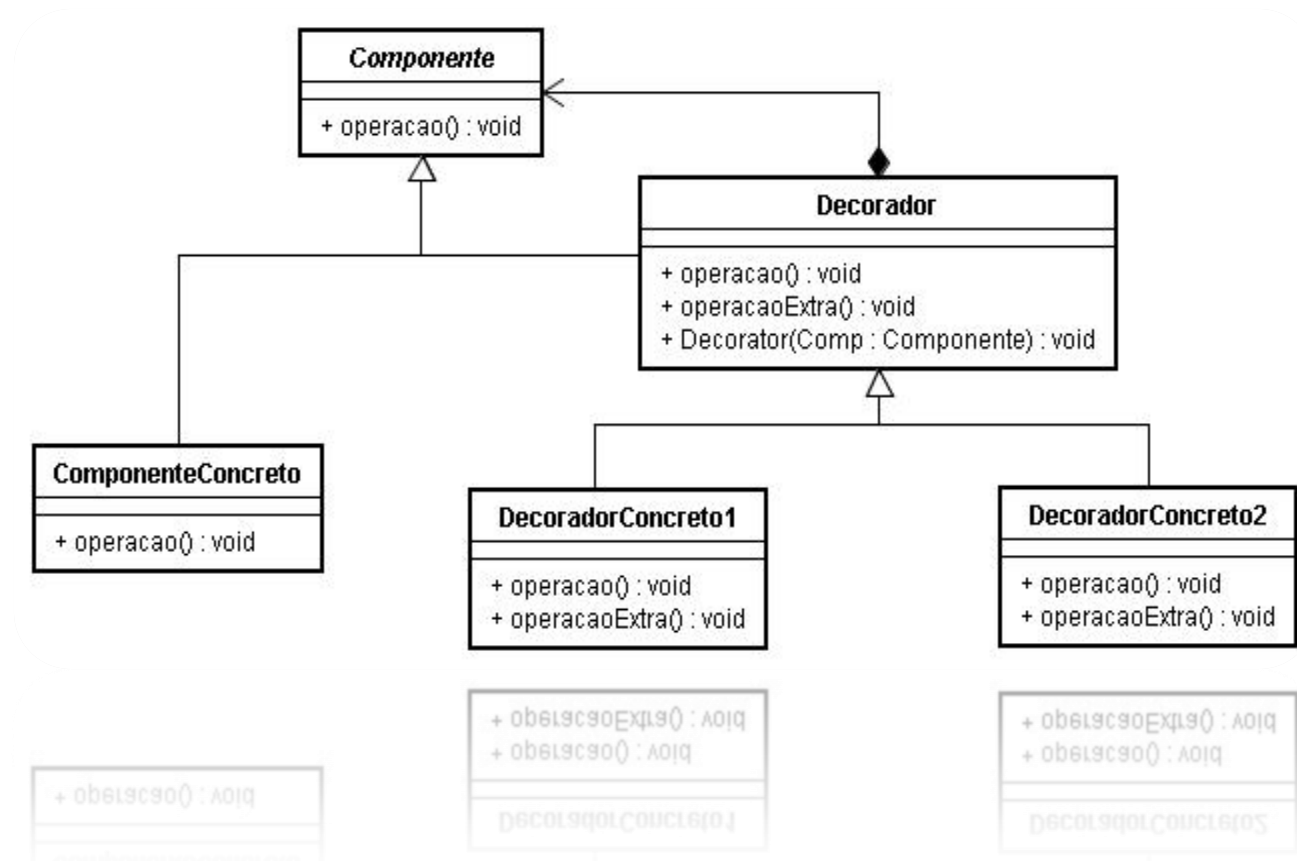
Composite - Consequências

- É possível tratar objetos em uma estrutura de árvore de maneira uniforme.
- Não há diferença na chamada de métodos entre um objeto qualquer ou uma composição deles.

Decorator (Decorador)- Motivação

- Adicionar novas funcionalidades a uma classe em tempo de execução
- Exemplo:
 - `FileInputStream(File)` => Lê Bytes
 - `InputStreamReader(FileInputStream)` => Lê Char
 - `BufferedReader(InputStreamReader)` => Lê Linha

Decorator (Decorador)- Solução



Decorator (Decorador)- Exemplo

```
10 public class DecoratorFileExemple {
11
12     public static void main(String[] args) {
13
14         File arquivo = new File("teste.txt");
15
16         FileInputStream in;
17         try {
18             in = new FileInputStream(arquivo);
19             //APENAS TRABALHA COM BYTES
20             //byte byteLido = (byte)in.read();
21
22             //CRIA UM DECORATOR PARA "in" PARA PERMITIR LEITURA DE UM CARACTER
23             InputStreamReader chf = new InputStreamReader(in);
24             //char letraLida = (char)chf.read();
25
26             //PERMITE LER LINHAS DO ARQUIVO ADICIONANDO UM DECORADOR A "chf"
27             BufferedReader br = new BufferedReader (chf);
28             String linha = br.readLine();
29             while(linha != null){
30                 System.out.println(linha);
31                 linha = br.readLine();
32             }
33         } catch (FileNotFoundException e) {
34             e.printStackTrace();
35         } catch (IOException e) {
36             e.printStackTrace();
37         }
38     }
```


Decorator - Consequências

- É possível adicionar novas funcionalidades a uma determinada classe acoplando-se a ela novos elementos.
- Extensibilidade de funções em tempo de execução

Pesquisar

- *Adapter* (Adaptador)
- *Facade* (Fachada)
- *Flyweight*
- *Proxy* (Procurador)

Exercício

- Algumas Empresas com Natura e HerbaLife possuem um sistema de vendas que funciona por meio de consultores.
- Um *Revendedor* ao atingir um grande número em vendas pode empregar outros revendedores, para que esses o auxiliem, tornando-se um *Consultor*.
- Revendedores ganham 15% do valor bruto de suas vendas.
- Consultores recebem, além das porcentagens de suas vendas, 10% do valor das vendas sob sua responsabilidade.

Exercício

- Um *Consultor* dispende muito tempo calculando quanto ganhou em comissões.
- 1 - Qual *Design Pattern* você usaria para auxiliar na resolução desse problema?
- 2- Modele, utilizando UML, a resolução desse problema

Bibliografia

- ALEXANDER, Christopher. *A Pattern Language: Towns, Buildings, Construction*. Ed. USA: Oxford University Press, 1977.
- GAMMA, Erich et al, *A Design Patterns: Elements of Reusable Object-Oriented Software*, Construction. 36. Ed. Portland: Addison-Wesley, 1995.
- SERAPHIM, Enzo. *Tópicos Especiais de Programação*. Notas de aula, Universidade Federal de Itajubá-UNIFEI, Itajubá-MG, Brasil, 2008.