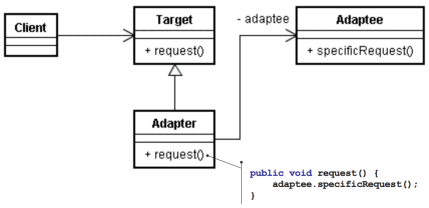
Padrões Estruturais

Exemplos Práticos

2015/2016

João Alegria | 68861



Adapter

CelciusReporter.java

```
public class CelciusReporter {
   double temperaturaC;
   public CelciusReporter() {}
   public double getTemperatura(){
     return temperaturaC;
   }
   public void setTemperatura(double temperaturaC){
     this.temperaturaC = temperaturaC;
   }
}
```

A classe **CelciusReporter** armazena um valor da temperatura em Celcius.

TemperatureInfo.java

```
public interface TemperatureInfo {
   public double getTemperaturaF();
   public void setTemperaturaF(double temperaturaF);
   public double getTemperaturaC();
   public void setTemperaturaC(double temperaturaC);
}
```

TemperatureInfo é a interface que será implementada pelo Adapter. Ela define as ações que o Adapter irá realizar.

TemperatureClassReporter.java

```
public class TemperatureClassReporter extends CelcuisReporter implements TemperatureInfo{
   @Override public double getTemperaturaC() {return temperaturaC;}
   @Override public double getTemperaturaF() {return cToF(temperaturaC);}
   @Override public void setTemperaturaC(double temperaturaC){
     this.temperaturaC = temperaturaC;
   }
   @Override public void setTemperaturaF(double temperaturaF){
     this.temperaturaF = temperaturaF;
   }

   private double fToC(double f){
     return ((f-32)*5/9);
   }
   private double cToF(double c){
     return ((c*9/5)+32);
   }
}
```

TemperatureClassReporter é um Adapter. Estende o CelciusReporter (o adaptado) e implementa TemperatureInfo (interface de destino).
Se a temperatura está em Celsius, TemperatureClassReporter utiliza temperaturaC do CelciusReporter. Os pedidos em Fahrenheit são manipulados internamente em Celcius.

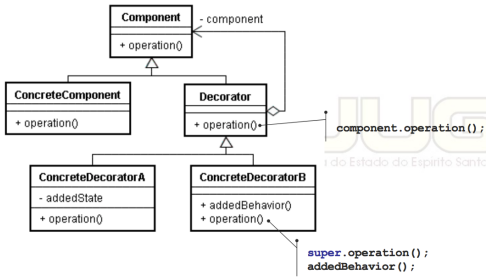
AdapterDemo.java

```
public class AdapterDemo {
             //Adapter Class
   public static void main(String[] args){
   System.out.println("class adapter test");
   TemperatureInfo tempInfo = new TemperatureClassReporter();
   testTempInfo(tempInfo);

             //Adapter Object
   System.out.println("objeto adapter test");
   tempInfo = new TemperatureObjectReporter();
   testTempInfo(tempInfo);
   }

   public static void testTempInfo(TemperatureInfo tempInfo){
      tempInfo.setTemperaturaC(0);
      System.out.println("temp in C:" + tempInfo.getTemperatureC();
      System.out.println("temp in F:" + tempInfo.getTemperatureF();

      tempInfo.setTemperaturaF(85);
      System.out.println("temp in C:" + tempInfo.getTemperatureC();
      System.out.println("temp in F:" + tempInfo.getTemperatureF();
   }
}
```

```
ClassAdapter Test    ObjectAdapter Test
. temp in C: 0.0     . temp in C: 0.0
. temp in F: 32.0    . temp in F: 32.0
. temp in C: 29.4    . temp in C: 29.4
. temp in F: 85.0    . temp in F: 85.0
```

A classe **AdapterDemo** é a classe *Client*. Primeiro, é criado um objeto TemperatureClassReporter e faz referência a ele através de uma temperaturaInfo.
Depois, é criado um objeto TemperatureObjetReporter e faz referência a ele através da mesma referência de TemperatureInfo. Ele então demonstra as chamadas para o objeto Adapter.

Decorator



Animal.java
```
public interface Animal{
   public void describe();
}
```

A interface **Animal** é a interface *Component*.

LivingAnimal.java
```
public class LivingAnimal implements Animal{
   @Override
   public void describe(){
     System.out.println("Sou um animal");
   }
}
```

LivingAnimal implementa Animal e é *ConcreteComponent*. O método describe( ) exibe uma mensagem indicando que é um animal.

AnimalDecorator.java
```
public abstract AnimalDecorator implements Animal{
   Animal animal;
   public AnimalDecorator (Animal animal){
     this.animal = animal;
   }
}
```

AnimalDecorator é o *Decorator*. Implementa Animal, mas como é classe abstrata, não precisa implementar describe( ).

LegDecorator.java
```
public class LegDecorator extends AnimalDecorator{
   public LegDecorator (Animal animal){
     super(animal);
   }
   @Override
   public void describe(){
     animal.describe();
     System.out.println("Tenho pernas");
     dance();
   }
   public void dance(){
     System.out.print("Eu danço");
   }
}
```

LegDecorator é um Concrete*Decorator*.
O construtor AnimalDecorator recebe como parâmetro um Animal. O método describe( ) de AnimalDecorator invoca a referência Animal describe( ) e em seguida, envia uma mensagem adicional. Depois, é chamado o método dance( ), mostrando que funcionalidades adicionais podem ser adicionadas pelo ConcreteDecorator.

WingDecorator.java
```
public class WingDecorator extends AnimalDecorator{
   public WingDecorator (Animal animal){
     super(animal);
   }
   @Override
   public void describe(){
     animal.describe();
     System.out.println("Tenho asas");
     fly();
   }
   public void fly(){
     System.out.print("Eu voo");
   }
}
```

WingDecorator é um Concrete*Decorator*, muito semelhante ao LegDecorator.

GrowlDecorator.java

```
public class GrowlDecorator extends AnimalDecorator{
   public GrowlDecorator (Animal animal){
     super(animal);
   }
   @Override
   public void describe(){
     animal.describe();
     System.out.println("Tenho asas");
     growl();
   }
   public void growl(){
     System.out.print("Grrr");
   }
}
```

GrowlDecorator é outro Concrete*Decorator*.

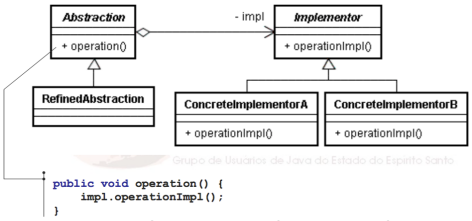
DecoratorDemo.java

```
public class DecoratorDemo {
   public static void main(String[] args){
   Animal animal = new LivingAnimal();
   animal.describe();

   animal = new LegDecorator(animal);
   animal.describe();

   animal = new WingDecorator(animal);
   animal.describe();

   animal = newGrowlDecorator(animal);
   animal = newGrowlDecorator(animal);
   animal.describe();
   }
}
```

```
Eu sou um animal

Eu sou um animal
Tenho pernas
Eu danço

Eu sou um animal
Tenho pernas
Eu danço
Tenho asas

Eu sou um animal
Tenho pernas
Eu danço
Tenho asas
Eu voo

Eu sou um animal
Tenho pernas
Eu danço
Tenho asas
Eu voo
Grrr
Grrr
```

A classe **DecoratorDemo** demonstra o padrão Decorator.
Primeiro, um objeto LivingAnimal é criado e referenciado através da referência Animal( ). O método describe( ) é chamado. Após isto, envolvemos o LivingAnimal num objeto LegDecorator e chama-se novamente describe( ).
Note-se que, as pernas foram adicionadas. Após isto, envolvemos o LegDecorator num WingDecorator que acrescenta asas ao nosso animal.
Mais tarde, envolvemos o animal em duas GrowlDecorator. Na saída de describe( ), podemos ver que duas mensagens de "rosnar" são exibidas.

Neste exemplo, usamos a referência Animal para se referir a objetos. Usando esta abordagem, apenas acedemos às operações partilhadas pela interface Animal, isto é, o método describe( ).
Invés de referenciar a interface, nós podemos referenciar a classe ConcreteDecorator. Isto expõe a funcionalidade excluías do ConcretDecorator.



Bridge

Supondo que temos uma classe Veículo. Podemos extrair a implementação do motor de uma classe Motor. Podemos referenciar este mecanismo no nosso veículo através do campo motor.

Veiculo.java

```
public abstract class Veiculo{
   Motor motor;
   int pesoKilos;
   public abstract void drive();
   public void setMotor (Motor motor) {this.motor = motor; }

   public void relatarVelocidade (int ncavalos){
     int racio = pesoKilos / ncavalos;
     if(racio<3) {System.out.println("Veículo vai a exceder a velocidade");}
     else if((racio>=3)&&(racio<8)) {System.out.println("Veículo está numa velocidade média");}
     else {System.out.println("Veículo vai a uma velocidade baixa");}
   }
}
```

A classe **Veiculo** vai ser abstrata. Subclasses de Veiculo precisam implementar o método drive( ). Observe que a referência Motor pode ser alterada a partir do método setMotor( ).

BigBus.java

```
public class BigBus extends Veiculo{
   public BigBus(Motor motor){
     this.pesoKilos = 3000;
     this.motor = motor;
   }
   @Override public void drive(){
   System.out.print("BigBus está a andar");
   int ncavalos = motor.go();
   relatarVelocidade(ncavalos);
   }
}
```

BigBus é uma subclasse de Veiculo. Tem um peso de 3000Kg e o seu método drive( ) exibe uma mensagem, invoca o método go( ) do Motor e em seguida, invoca o relatarVelocidade com a potência do motor para saber o quão rápido o veículo se está a mover.

SmallCar.java

```
public class SmallCar extends Veiculo{
   public SmallCar(Motor motor){
     this.pesoKilos = 600;
     this.motor = motor;
   }
   @Override public void drive(){
   System.out.print("SmallCar está a andar");
   int ncavalos = motor.go();
   relatarVelocidade(ncavalos);
   }
}
```

SmallCar é semelhante a BigBus só que mais leve.

Motor.java

```
public interface Motor{
   public int go();
}
```

A nossa interface implementadora é **Motor** que declara o método go( ).

BigMotor.java

```
public class BigMotor implements Motor{
   int ncavalos;
   public BigMotor(){ncavalos=350;}
   @Override public int go(){
   System.out.println("BigMotor está a andar");
   return ncavalos;
   }
```

Um **BigMotor** implementa Motor. BigMotor tem 350 cavalos e o método go( ) relata se está ou não em funcionamento e retorna a potência.

SmallMotor.java

```
public class SmallMotor implements Motor{
   int ncavalos;
   public BigMotor(){ncavalos=100;}
   @Override public int go(){
   System.out.println("SmallMotor está a andar");
   return ncavalos;
   }
```

SmallMotor é semelhante ao BigMotor. Tem apenas 100 cavalos.

BridgeDemo.java

```
public class BridgeDemo{
   public static void main(String[] args){
     Veiculo v = new BigBus(new SmallMotor());
     v.drive();
     v.setMotor(new BigMotor());
     v.drive();

     v = new SmallCar(new SmallMotor());
     v.drive();
     v.setMotor(new BigMotor());
     v.drive();
   }
}
```

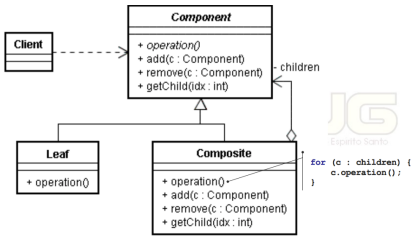
A classe **BridgeDemo** demonstra o padrão Bridge. É criado um veículo BigBus com um implementador SmallMotor. É invocado o método drive( ) e, em seguida, é alterado o implementador para BigMotor e é chamado novamente drive( ). Depois disto, é criado um SmallCar com um implementador SmallMotor. É invocado drive( ) e, em seguida, altera-se o motor para um BigMotor e mais uma vez drive( ).

```
BigBus está a andar
SmallMotor está a andar
Veículo vai a uma velocidade baixa.

SmallCar está a andar
SmallMotor está a andar
Veículo está numa velocidade média.
```

```
BigBus está a andar
BigMotor está a andar
Veículo vai a uma velocidade baixa.

SmallCar está a andar
BigMotor está a andar
Veículo vai a exceder a velocidade.
```

Observe que é possível mudar o implementador (Motor) dinamicamente para cada veículo. Estas mudanças não afetam o código do cliente em BridgeDemo. Além disso, apesar de BigBus e SmallCar serem ambos subclasses abstratas de Veículo, foi possível apontar a referência veículo para um objeto BigBus e um objeto SmallCar e chamar o método drive( ) à mesma para ambos os tipos de veículos.

Composite



Component.java

```
public interface Component{
   public void sayHello();
   public void sayGoodBye();
}
```

Declaração da interface **Component** que declara quais as operações que são comuns para a classe Composite e classe Folha. Isto permite realizar operações em "compósitos" e folhas, utilizando uma interface padrão.

Leaf.java

```
public class Leaf implements Component{
   String nome;
   public Leaf(String nome){this.nome = nome;}
   @Override public void sayHello(){
     System.out.println("folha" + nome + "diz olá");
   }
   @Override public void sayGoodBye(){
     System.out.println("folha" + nome + "diz xau");
   }
}
```

A classe **Leaf** tem um campo nome e implementa os métodos sayHello( ) e sayGoodBye( ) da interface Component, indicando as mensagens de saída do padrão.

Composite.java

```
import java.util.ArrayList;
import java.util.List;
public class Composite implements Component{
   List<Component> componentes = new ArrayList<Component>();
   @Override public void sayHello(){
     for(Component componente:componentes){componente.sayHello();}
   }
   @override public void sayGoodBye(){
     for(Component componente:componentes){componente.sayGoodBye();}
   }
   public void add(Component componente){componentes.add(componente);}
   public void remove(Component componente){componentes.remove(componente);}
   public List<Component> getComponents(){return componentes;}
   public Component getComponent(int i){return componentes.get(i);}
}
```

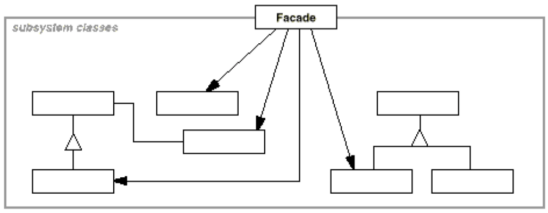
A classe **Composite** implementa a interface de Component. Implementa os métodos sayHello( ) e sayGoodBye( ), chamando esses métodos em todos os seus filhos, que são os Components (uma vez que eles podem ser objetos Leaf e objetos Composite, implementando ambos a interface Componente).

CompositeDemo.java

```
public class CompositeDemo{
   public static void main(String[] args){
   Leaf folha1 = new Leaf("Bob");
   Leaf folha2 = new Leaf("Fred");
   Leaf folha3 = new Leaf("Sue");
   Leaf folha4 = new Leaf("Ellen");
   Leaf folha5 = new Leaf("Joe");

   Composite composite1 = new Composite();
   composite1.add(folha1);
   composite1.add(folha2);

   Composite composite2 = new Composite();
   composite2.add(folha3);
   composite2.add(folha4);
   Composite composite3 = new Composite();
   composite3.add(composite1);
   composite3.add(composite2);
   composite3.add(folha5);

   System.out.println("-Calling 'sayHello' na folha1");
   folha1.sayHello();

   System.out.println("-Calling 'sayHello' no composite1");
   composite1.sayHello();

   System.out.println("-Calling 'sayHello' no composite2");
   composite2.sayHello();

   System.out.println("-Calling 'sayGoodBye' no composite3");
   composite3.sayGoodBye();
   }
}
```

```
-Calling 'sayHello' na folha1
A folha Bob diz Olá
-Calling 'sayHello' no composite1
A folha Bob diz olá
A folha Fred diz olá
-Calling 'sayHello' no composite2
A folha Sue diz olá
A folha Ellen diz olá
-Calling 'sayGoodBye' no composite3
A folha Bob diz xau
A folha Fred diz xau
A folha Sue diz xau
A folha Ellen diz xau
A folha Joe diz xau
```

Como visto na saída, o padrão Composite permite-nos realizar operações sobre os compósitos e as folhas que compõem uma estrutura de árvore através de uma interface comum.



Façade

Supondo que temos três classes horrivelmente escritas… Baseando no nome das classes e dos métodos (falta de documentação), será muito difícil para um cliente interagir com as classes.

Class1.java

```
public class Class1{
   public int doSomethingComplicated(int x){
      return x*x*x;
   }
}
```

O método doSomethingComplicated( ) da **Class1** recebe um inteiro e retorna o seu cubo.

Class2.java

```
public class Class2{
   public int doAnotherThing(Class1 c1, int x){
      return 2*c1.doSomethingComplicated(x);
   }
}
```

O método doAnotherThing( ) da **Class2** duplica o cubo de um número inteiro e retorna-o.

Class3.java

```
public class Class3{
   public int doMore(Class1 c1, Class2 c2, int x){
     return c1.doSomethingComplicated(x) *
            c2.doAnotherThing(c1,x);
   }
}
```

O método doMore( ) da **Class3** leva como argumentos um objeto da class1, um objeto da class2 e um inteiro e retorna duas o sêxtuplo do inteiro.

Para um cliente que não esteja familiarizado com a Class1, Class2 e Class3, será muito difícil descobrir como interagir com essas classes. As classes interagem e realizam as tarefas de forma pouco clara. Assim, temos de simplificar a interação com estes sistemas de classes para que os clientes possam interagir com essas classes de uma maneira simples e padronizada.Podemos fazer isso com uma classe Façade.

Façade.java

```
public class Façade{
   public int cubeX(int x){
     Class1 c1 = new Class1();
     return c1.doSomethingComplicated(x);
   }
   public int cubeXTimes2(int x){
     Class1 c1 = new Class1();
     Class2 c2 = new Class2();
     return c2.doAnotherThing(c1, x);
   }
   public int xToSixthPowerTimes2(int x){
     Class1 c1 = new Class1();
     Class2 c2 = new Class2();
     Class3 c3 = new Class3();
     return c3.doMore(c1, c2, x);
   }
}
```

A classe **Façade** tem três métodos: cubeX( ), cubeXTimes2( ), xToSixthPowerTimes2( ). Os nomes desses métodos indicam claramente o que fazem e escondem a interação com as Class1,2 e 3 do código cliente.

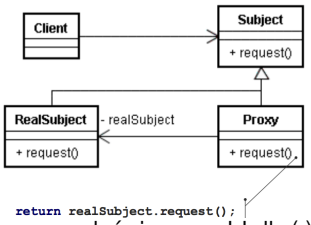
A classe **FaçadeDemo** contém o código cliente. Ele cria um objeto Façade e chama os seus três métodos com um valor de parâmetro 3. Ele exibe os resultados retornados.

FaçadeDemo.java

```
public class FaçadeDemo{
   public static void main(String[] args){
   Façade facade = new Façade();
   private final static int x = 3;

   System.out.println("O cubo de " + x +":" + facade.cubeX(x));
   System.out.println("O cubo de " + x +"*2:" + facade.cubeXTimes2(x));
   System.out.println(x + "ao sextuplo*2:" + facade.xToSixthPowerTimes2(x));
   }
}
```

```
Cubo de 3: 27
Cubo de 3*2: 54
3 ao sêxtuplo*2: 1458
```

Este padrão demonstra como o padrão Façade pode ser usado para simplificar interações um sistema de classes, fornecendo um único ponto de interação com o subsistema, escondendo os detalhes complexos de interação do subsistema do código cliente.



Proxy

Vamos criar uma classe abstrata chamada **Thing**, com uma mensagem básica sayHello( ) que inclui a data/hora que a mensagem é exibida.

Thing.java

```
import java.util.Date;
public abstract class Thing(){
   public void sayHello(){
     System.out.println("this.getClass().getSimpleName() + "diz olá em" + new Date());
   }
}
```

FastThing.java

```
public class FastThing extends Thing{
   public FastThing(){}
}
```

FastThing é subclasse de Thing.

SlowThing.java

```
public class SlowThing extends Thing{
   public FastThing(){
    try{
     Thread.sleep(500);
    }catch(InterruptedException e)
       {e.printStackTrace();}
   }
}
```

SlowThing também é subclasse de Thing. No entanto, o seu construtor leva 5 segundos a executar.

Proxy.java

```
import java.util.Date;
public class Proxy{
   SlowThing slowthing;
   public Proxy(){
     System.out.println("Criando um proxy em: new Date());
   }
   public void sayHello(){
     if(slowThing==null){
       slowthing = new SlowThing();
     }
     slowthing.sayHello();
   }
}
```

A classe **Proxy** é um proxy para o objeto SlowThing. Uma vez que o objeto SlowThing leva 5 segundos para ser criado, é usado um proxy para SlowThing para que este possa ser criado por "demanda". Isto ocorre quando o método sayHello( ) do proxy é executado. Instancia um objeto SlowThing, caso não exista, então chama sayHello( ) no objeto SlowThing

ProxyDemo.java

```
public class ProxyDemo{
   public static void main(String[] args){
   Proxy proxy = new Proxy();
   FastThing fasting = new FastThing();
   fastthing.sayHello();
   proxy.sayHello();
   }
}
```

```
Criando um proxy em Sat Mon 03 16:41:06 2015
FastThing diz olá em Sat Mon 03 16:41:06 2015
SlowThing diz olá em Sat Mon 03 16:41:11 2015
```

A classe **ProxyDemo** demonstra o uso do proxy. Cria um objeto Proxy e depois cria um objeto FastThing e invoca sayHello( ) no objeto fastthing. Em seguida, ele invoca sayHello( ) no objeto proxy.

Na saída, repara que é criado um objeto Proxy e chamado sayHello( ) no objeto FastThing às 16:41:06 e chama sayHello( ) no objeto Proxy e não é invocado sayHello( ) no objeto SlowThing até às 16:41:11. Podemos ver que a criação de SlowThing foi um processo demorado. No entanto, isto não atrasa a execução da aplicação até que o objeto SlowThing seja realmente necessário.
Podemos ver que o padrão evita a criação de objetos demoratos até que sejam realmente necessários.