


# PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

---

Daniel Cordeiro

 25 de maio de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

## PADRÕES ESTRUTURAIS

---

São padrões relacionados à composição de classes (com herança) e objetos.

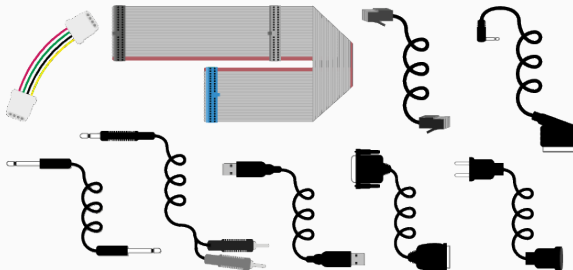
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Private Class Data
- Proxy

### Problema

Um componente “pronto para usar” oferece alguma funcionalidade interessante que você gostaria de usar, mas a sua “visão do mundo” não é compatível com a filosofia e arquitetura do sistema que está sendo desenvolvido.

## ADAPTER: OBJETIVO

- Converter a interface de uma classe na interface esperada pelo código cliente
- Embalar uma classe existente com uma nova interface
- “Casar a impedância” de um componente antigo em um sistema novo



- Reutilização de código antigo é sempre um problema no desenvolvimento de um sistema novo
- Passamos por esse tipo de problema recentemente:

## ADAPTER: DISCUSSÃO

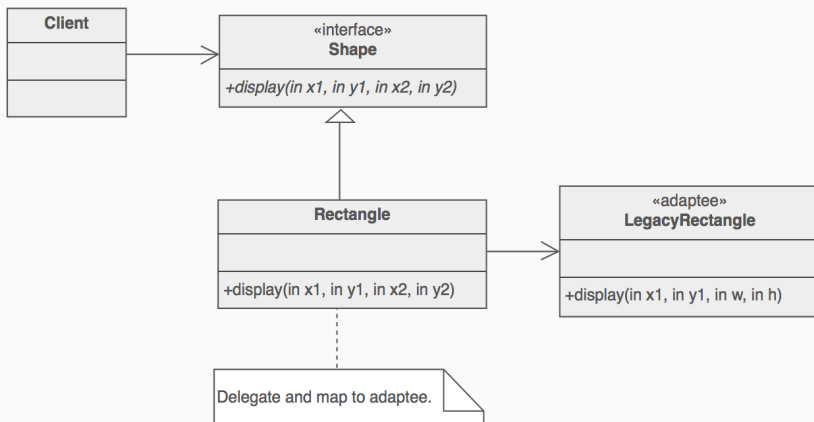
- Reutilização de código antigo é sempre um problema no desenvolvimento de um sistema novo
- Passamos por esse tipo de problema recentemente:



- O padrão Adapter trata do problema de criar uma abstração intermediária que traduza (ou mapeie) um componente antigo para um sistema novo
- O cliente chama os métodos do objeto adaptador, que os redirecionam para o componente legado

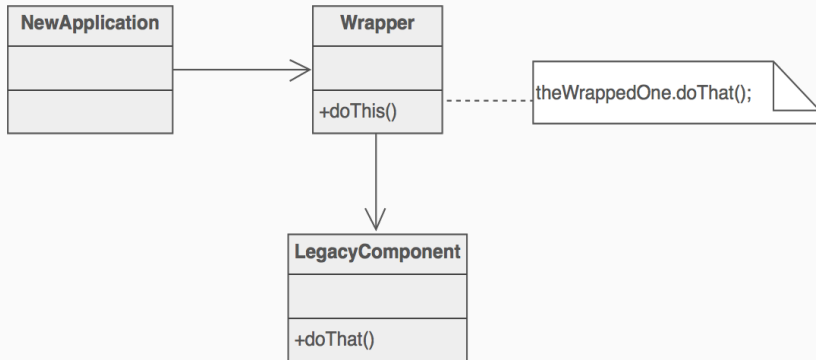
## Exemplo:

Um componente legado **LegacyRectangle** que espera receber um ponto, largura e altura de um retângulo, mas o cliente quer passar as coordenadas do ponto inferior esquerdo e superior direito.

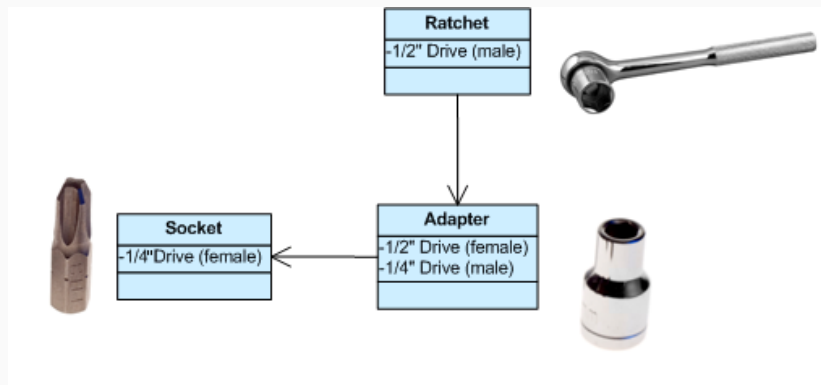




## ADAPTER: ESTRUTURA



## ADAPTER: EXEMPLO



- Identifique os envolvidos: o(s) componente(s) que devem ser acomodados (o cliente) e os componentes que precisam ser adaptador (o adaptado)
- Identifique a interface que o cliente necessita
- Projete uma classe *wrapper* que case a impedância entre o adaptado e o cliente
- A classe do adaptador/*wrapper* “tem uma” instância do adaptado
- O adaptador/*wrapper* “mapeia” a interface do cliente para a interface do adaptado
- O cliente usa (é acoplado à) nova interface

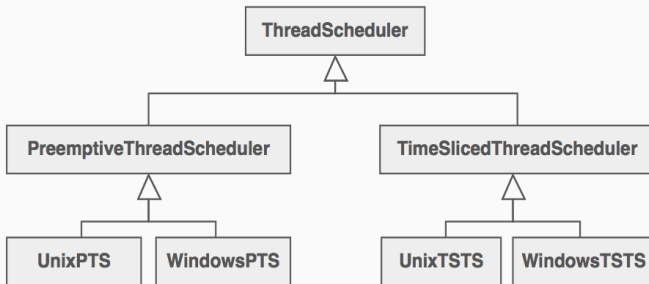
## BRIDGE

---

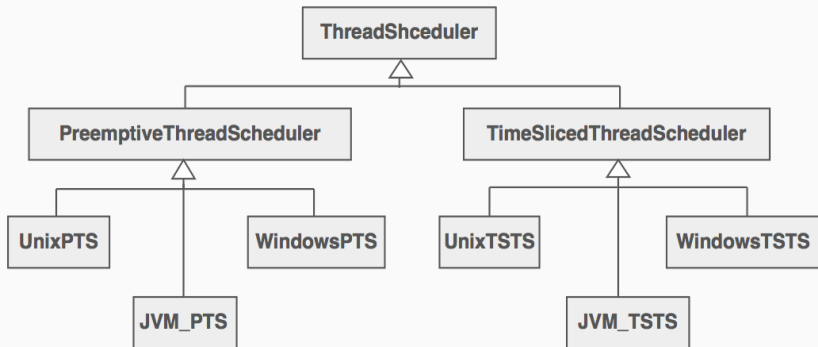
- Desacoplar uma abstração de sua abstração de modo que os dois possam variar independentemente
- Definir a interface em uma hierarquia de classes, e colocar as implementações em outra hierarquia
- Ir além de encapsulamento, provê isolamento

## Problema

Criar subclasses de uma classe abstrata base pode “engessar o código”, fazendo com que seja difícil prover implementações alternativas. Isso faz com que seja criado um laço forte entre interfaces e implementações em tempo de compilação. A abstração e as implementações não podem ser estendidas ou combinadas de forma independente.



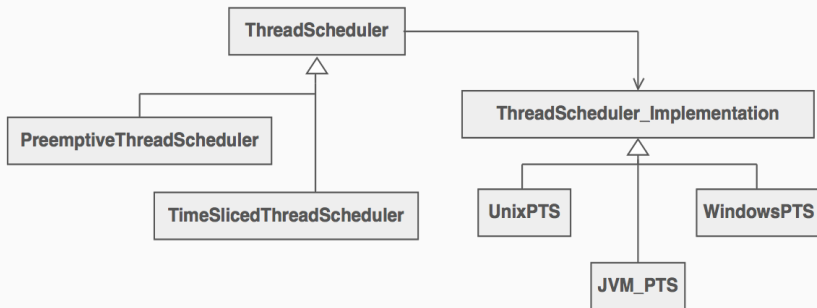
- Há dois tipos de escalonadores e dois tipos de sistemas operacionais
- Seguindo essa abordagem, temos que definir uma classe para cada uma das permutações dessas duas dimensões



E se tivéssemos três tipos de escalonadores e quatro plataformas?

## Ideia

O padrão Bridge propõe refatorar essa hierarquia de heranças exponencial em duas hierarquias ortogonais: uma para as abstrações independentes de plataforma, outra para as dependentes de plataforma.





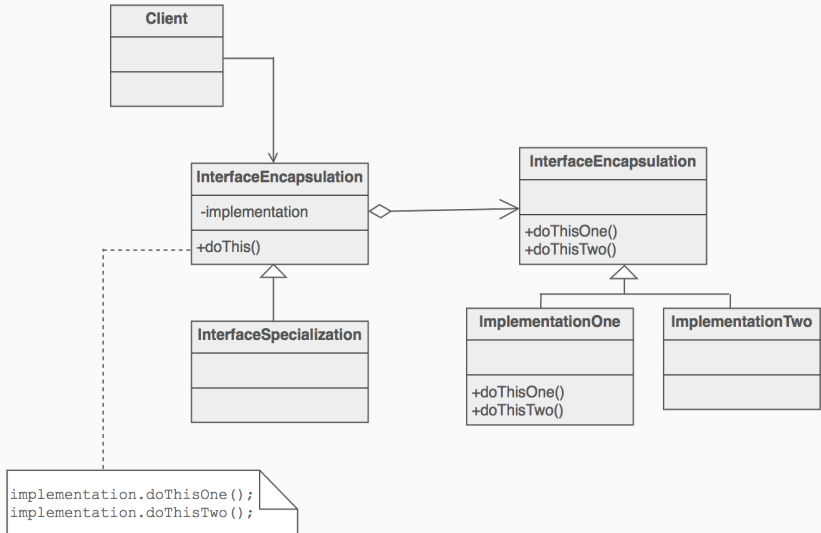
- Decomponha a interface do componente e a implementação em duas hierarquias de classes ortogonais.
- A classe da interface possui um ponteiro para a classe abstrata de implementações, inicializada com uma instância concreta com a implementação requerida

## Use o padrão Bridge quando:

- você quiser escolher a implementação em tempo de execução
- você tiver uma proliferação de classes resultada do acoplamento da interface com várias implementações
- você quiser compartilhar uma implementação entre vários objetos

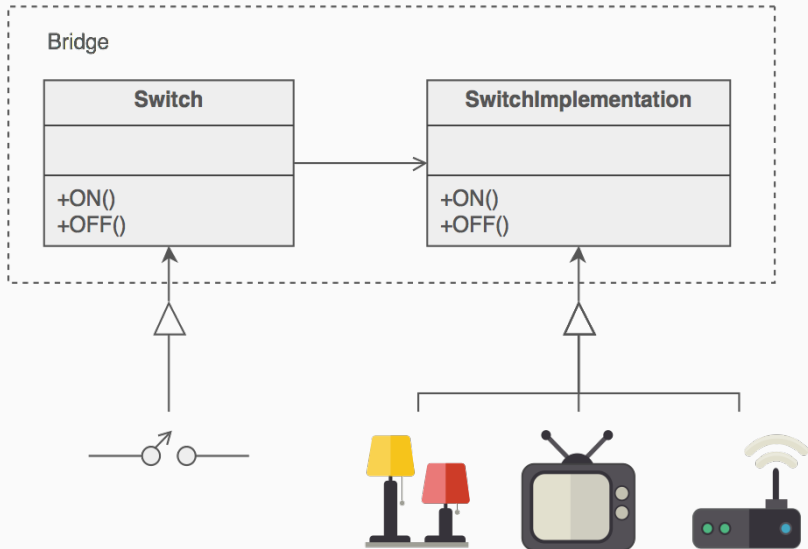
## Consequências

- desacoplamento da interface do objeto
- extensibilidade — você pode estender (ou seja, criar subclasses) as abstrações e implementações de forma independente
- esconde os detalhes dos clientes



(à direita é InterfaceImplementation e não InterfaceEncapsulation)

# EXEMPLO



# IMPLEMENTAÇÃO I

```
class Node {
    public int value;
    public Node prev, next;
    public Node( int i ) { value = i; }
}

class Stack {
    private StackImpl impl;
    public Stack( String s ) {
        if (s.equals("array")) impl = new StackArray();
        else if (s.equals("list")) impl = new StackList();
        else System.out.println( "Stack: unknown parameter" );
    }
    public Stack() { this( "array" ); }
    public void push( int in ) { impl.push( in ); }
    public int pop() { return impl.pop(); }
    public int top() { return impl.top(); }
    public boolean isEmpty() { return impl.isEmpty(); }
    public boolean isFull() { return impl.isFull(); }
}
```

## IMPLEMENTAÇÃO II

```
class StackHanoi extends Stack {  
    private int totalRejected = 0;  
    public StackHanoi() { super( "array" ); }  
    public StackHanoi( String s ) { super( s ); }  
    public int reportRejected() { return totalRejected; }  
    public void push( int in ) {  
        if ( ! isEmpty() && in > top() )  
            totalRejected++;  
        else super.push( in );  
    }  
}
```

```
class StackFIFO extends Stack {  
    private StackImpl temp = new StackList();  
    public StackFIFO() { super( "array" ); }  
    public StackFIFO( String s ) { super( s ); }  
    public int pop() {  
        while ( ! isEmpty() )  
            temp.push( super.pop() );  
        int ret = temp.pop();  
        while ( ! temp.isEmpty() )  
            push( temp.pop() );  
        return ret; }  
}
```

## IMPLEMENTAÇÃO III

```
interface StackImpl {
    void    push( int i );
    int     pop();
    int     top();
    boolean isEmpty();
    boolean isFull();
}

class StackArray implements StackImpl {
    private int[] items = new int[12];
    private int  total = -1;
    public void push( int i ) { if ( ! isFull()) items[++total] = i; }
    public boolean isEmpty() { return total == -1; }
    public boolean isFull()  { return total == 11; }
    public int top() {
        if (isEmpty()) return -1;
        return items[total];
    }
    public int pop() {
        if (isEmpty()) return -1;
        return items[total--]; } } }
```

## IMPLEMENTAÇÃO IV

```
class StackList implements StackImpl {
    private Node last;
    public void push( int i ) {
        if (last == null)
            last = new Node( i );
        else {
            last.next = new Node( i );
            last.next.prev = last;
            last = last.next;    } }
    public boolean isEmpty() { return last == null; }
    public boolean isFull() { return false; }
    public int top() {
        if (isEmpty()) return -1;
        return last.value;
    }
    public int pop() {
        if (isEmpty()) return -1;
        int ret = last.value;
        last = last.prev;
        return ret;
    } }
```



## COMPOSITE

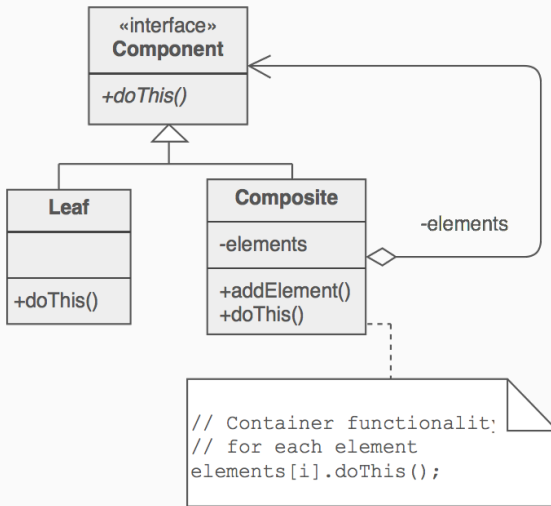
---

- Compor objetos em estruturas de árvore que representem hierarquias do tipo todo–parte
- Permite que os usuários tratem *objetos individuais* e *composições de objetos* da mesma forma
- Composição recursiva
- “Diretórios possuem entradas e cada uma pode ser um diretório”

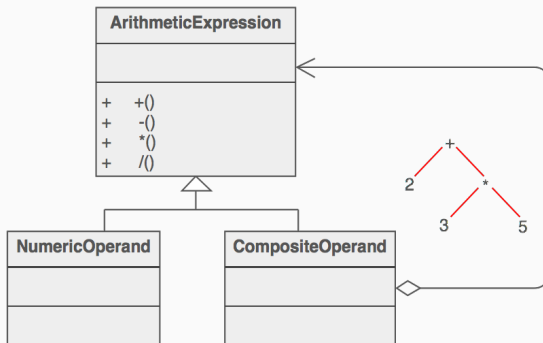
### Problema

Uma aplicação precisa manipular uma coleção hierárquica de objetos “primitivos” e “compostos”. O processamento de um objeto primitivo é feito de um jeito, enquanto que o processamento do objeto composto é feito de outro. Precisar verificar qual o “tipo” do objeto antes de processá-lo não é desejável.

- Defina uma classe abstrata (**Componente**) que especifica o comportamento que precisa ser uniforme entre os objetos primitivos e compostos
- Faça as classes **Primitiva** e **Composta** serem subclasses de **Componente**
- Cada objeto **Componente** deve estar acoplado apenas a objetos do tipo abstrato
- Use esse padrão quando tiver “objetos compostos que podem conter componentes, que por sua vez podem ser compostos”



## EXEMPLO



- Uma expressão aritmética é um exemplo de objeto composto
- Uma expressão binária tem um operando, um operador (+ - \* /) e um outro operando
- Cada operando pode ser um número ou uma nova expressão

# IMPLEMENTAÇÃO (SEM COMPOSITE) I

```
class File {  
    public File(String name) {  
        m_name = name;  
    }  
    public void ls() {  
        System.out.println(Composite.g_indent + m_name);  
    }  
    private String m_name;  
}
```

## IMPLEMENTAÇÃO (SEM COMPOSITE) II

```
class Directory {
    public Directory(String name) {
        m_name = name;
    }
    public void add(Object obj) {
        m_files.add(obj);
    }
    public void ls() {
        System.out.println(Composite.g_indent + m_name);
        Composite.g_indent.append(" ");
        for (int i = 0; i < m_files.size(); ++i) {
            Object obj = m_files.get(i);
            // Recover the type of this object
            if (obj.getClass().getName().equals("Directory"))
                ((Directory)obj).ls();
            else
                ((File)obj).ls();
        }
        Composite.g_indent.setLength(CompositeDemo.g_indent.length() - 3);
    }
    private String m_name;
    private ArrayList m_files = new ArrayList();}
```

# IMPLEMENTAÇÃO (SEM COMPOSITE) III

```
public class CompositeDemo {  
    public static StringBuffer g_indent = new StringBuffer();  
  
    public static void main(String[] args) {  
        Directory one = new Directory("dir111"), two = new Directory("dir222"),  
            thr = new Directory("dir333");  
        File a = new File("a"), b = new File("b"), c = new File("c"), d = new  
            File("d"), e = new File("e");  
        one.add(a);  
        one.add(two);  
        one.add(b);  
        two.add(c);  
        two.add(d);  
        two.add(thr);  
        thr.add(e);  
        one.ls();  
    }  
}
```



# IMPLEMENTAÇÃO (COM COMPOSITE) I

```
// Define a "lowest common denominator"
interface AbstractFile {
    public void ls();
}

// File implements the "lowest common denominator"
class File implements AbstractFile {
    public File(String name)
    {
        m_name = name;
    }
    public void ls()
    {
        System.out.println(CompositeDemo.g_indent + m_name);
    }
    private String m_name;
}
```

## IMPLEMENTAÇÃO (COM COMPOSITE) II

```
// Directory implements the "lowest common denominator"
class Directory implements AbstractFile {
    public Directory(String name) {
        m_name = name;
    }
    public void add(Object obj) {
        m_files.add(obj);
    }
    public void ls() {
        System.out.println(CompositeDemo.g_indent + m_name);
        CompositeDemo.g_indent.append("  ");
        for (int i = 0; i < m_files.size(); ++i) {
            // Leverage the "lowest common denominator"
            AbstractFile obj = (AbstractFile)m_files.get(i);
            obj.ls();
        }
        CompositeDemo.g_indent.setLength(CompositeDemo.g_indent.length() - 3);
    }
    private String m_name;
    private ArrayList m_files = new ArrayList();
}
```

# IMPLEMENTAÇÃO (COM COMPOSITE) III

```
public class CompositeDemo {  
    public static StringBuffer g_indent = new StringBuffer();  
  
    public static void main(String[] args) {  
        Directory one = new Directory("dir111"), two = new Directory("dir222"),  
            thr = new Directory("dir333");  
        File a = new File("a"), b = new File("b"), c = new File("c"), d = new  
            File("d"), e = new File("e");  
        one.add(a);  
        one.add(two);  
        one.add(b);  
        two.add(c);  
        two.add(d);  
        two.add(thr);  
        thr.add(e);  
        one.ls();  
    }  
}
```

## LISTA DE VERIFICAÇÃO

1. Verifique se o problema é representar uma relação hierárquica “todo–parte”
2. Veja se você consegue expressar o problema como “um contêiner cujo conteúdo pode ser um contêiner”.
3. Crie uma interface que seja o “menor denominador comum” que permita você usar tanto o contêiner quanto o seu conteúdo. Ela deve especificar o comportamento que deve ser igual para ambos
4. As classes contêiner e conteúdo definem uma relação do tipo “é um”
5. Toda classe contêiner define uma relação de um–para–muitos do tipo “tem um” em sua interface
6. Classes contêiner usam polimorfismo para delegar operações aos seus objetos conteúdo

## DECORATOR

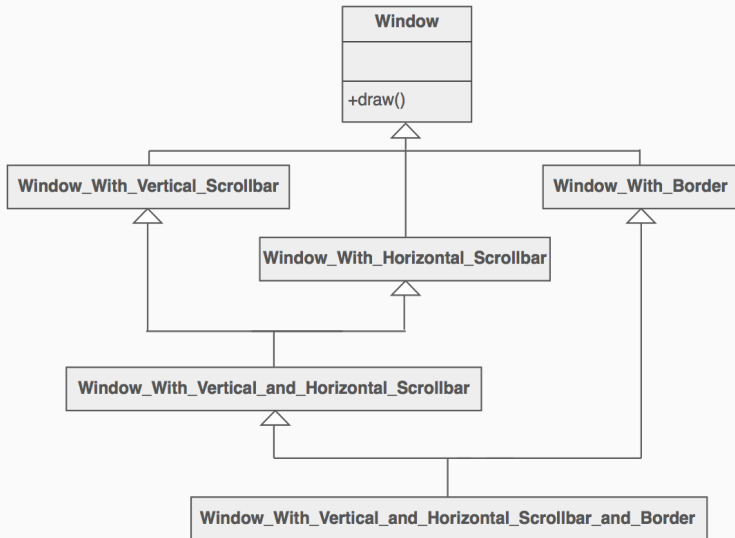
---

- Adicionar novas responsabilidades a um objeto dinamicamente (alternativa mais flexível à herança)
- “Embelezar” um objeto ao embrulhá-lo em outro objeto (recursivamente)
- “Embrulhe o presente, coloque-o em uma caixa e embrulhe a caixa”

### Problema

Você quer adicionar um novo comportamento ou estado a um objeto individual em tempo de execução. Herança não é uma solução possível não só porque é estática, mas também porque se aplica a uma classe inteira.

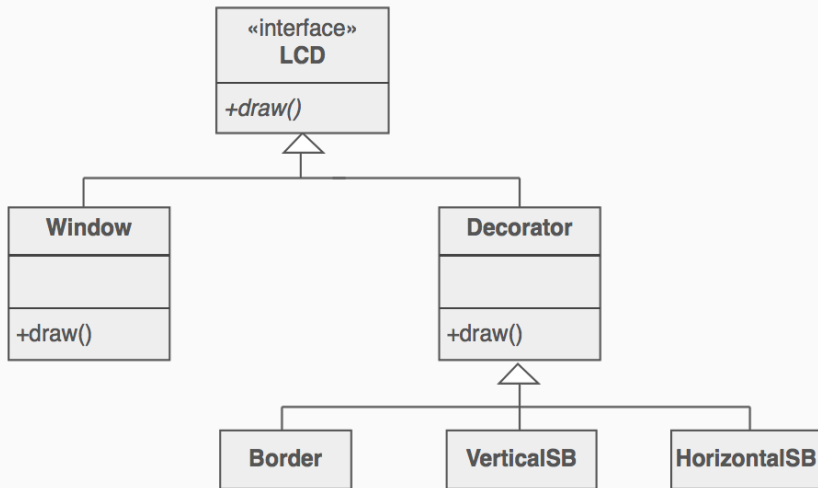
Qual o problema dessa hierarquia de classes?

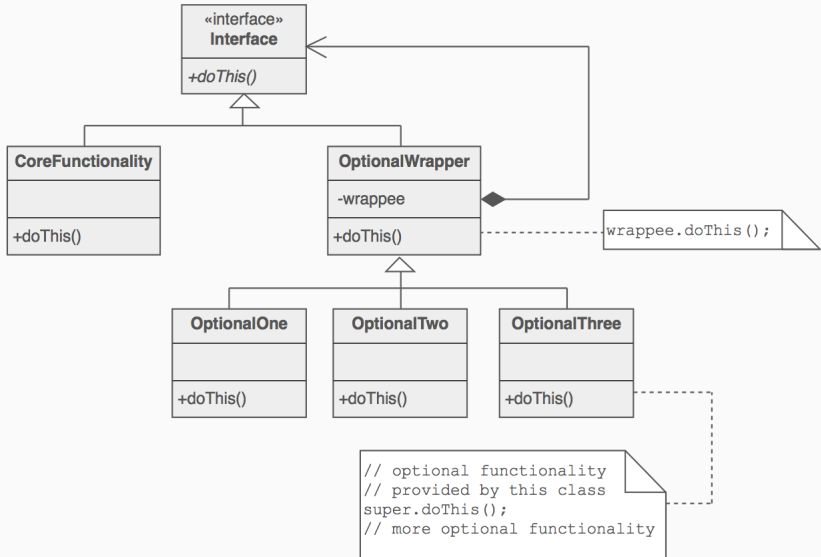


O padrão Decorator sugere que o cliente pode combinar novas funcionalidades quando quiser:

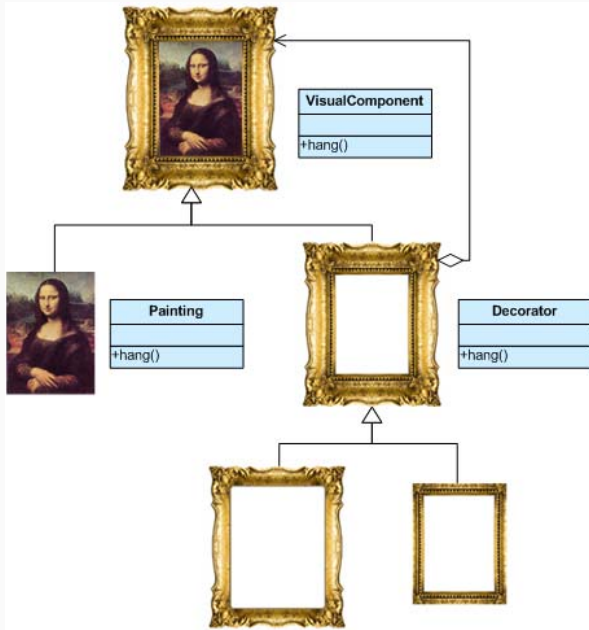
```
Widget* aWidget = new BorderDecorator(  
    new HorizontalScrollBarDecorator(  
        new VerticalScrollBarDecorator(  
            new Window( 80, 24 ))));  
aWidget->draw();
```







# EXEMPLO



# IMPLEMENTAÇÃO I

```
// 1. "lowest common denominator"
interface Widget {
    void draw();
}

// 2. "Core" class with "is a" relationship
class TextField implements Widget {
    private int width, height;
    public TextField( int w, int h ) {
        width = w;
        height = h;
    }
    public void draw() {
        System.out.println( "TextField: " + width + ", " + height );
    }
}
```

## IMPLEMENTAÇÃO II

```
// 3. Second level base class with "is a" relationship
abstract class Decorator implements Widget {
    private Widget wid; // 4. "has a" relationship

    public Decorator( Widget w ) {
        wid = w;
    }

    // 4. Delegation
    public void draw() {
        wid.draw();
    }
}

// 5. Optional embellishment
class BorderDecorator extends Decorator {
    public BorderDecorator( Widget w ) {
        super( w );
    }
    public void draw() {
        super.draw(); // 7. Delegate to base class and add extra stuff
        System.out.println("  BorderDecorator");  }}
}
```

## IMPLEMENTAÇÃO III

```
// 6. Optional embellishment
class ScrollDecorator extends Decorator {
    public ScrollDecorator( Widget w ) {
        super( w );
    }
    public void draw() {
        super.draw(); // 7. Delegate to base class and add extra stuff
        System.out.println( " ScrollDecorator" );
    }
}

public class DecoratorDemo {
    public static void main( String[] args ) {
        // 8. Client has the responsibility to compose desired configurations
        Widget aWidget = new BorderDecorator(
            new BorderDecorator(
                new ScrollDecorator(
                    new TextField( 80, 24 )
                )
            )
        );
        aWidget.draw();
    }
}
```

## LISTA DE VERIFICAÇÃO

1. Verifique se o problema tem: um elemento principal, vários tipos de características opcionais que melhoram esse elemento e uma interface que é comum a todos
2. Crie uma interface que seja o “menor denominador comum” (MDC) e que permita usar qualquer tipo de classe
3. Crie uma classe base (**Decorator**) como base para todos os *wrappers*
4. As classes **Core** e **Decorator** herdam a interface do **MDC**
5. A classe **Decorator** declara uma relação de composição com a interface **MDC**, e seu valor é inicializado no construtor
6. A classe **Decorator** delega as ações ao objeto **MDC**
7. Defina uma classe derivada de **Decorator** para todos os “embelezamentos” opcionais
8. As classes derivadas implementam sua funcionalidade adicional e delega o resto à classe base de **Decorator**
9. O cliente define o tipo e a ordem dos objetos principais e **Decorator**

- The Gang of Four Book, ou GoF: E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns — Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- Alexander Shvets. Design patterns explained simply.  
[https://sourcemaking.com/design\\_patterns/](https://sourcemaking.com/design_patterns/)