# Design Patterns / Padrões de Projeto
# Exercícios – Padrões de Estrutura

## 1. Adapter

### 1.1. Exercício:

A classe java.util.Map da API de coleções de Java permite que sejam armazenados pares de objetos (chave e valor) em uma de suas implementações (as mais conhecidas são HashMap e TreeMap). No entanto, estas classes não possuem um construtor que receba como parâmetro uma matriz de duas linhas e que monte o mapa usando a primeira linha como chave e a segunda como coluna. Crie um adaptador (dica: use Adapter de classe) que tenha este construtor.

### 1.2. Exercício:

Abaixo estão os códigos fonte de um cliente, uma interface para um somador que ele espera utilizar e uma classe concreta que implementa uma soma, mas não da maneira esperada pelo cliente. Como você pode ver abaixo, o cliente espera usar uma classe que soma inteiros em um vetor, mas a classe pronta soma inteiros em uma lista. Crie um adaptador (dica: use Adapter de objeto) para resolver esta situação.

```
public class Cliente {
  private SomadorEsperado somador;

  private Cliente(SomadorEsperado somador) {
    this.somador = somador;
  }

  public void executar() {
    int[] vetor = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int soma = somador.somaVetor(vetor);
    System.out.println("Resultado: " + soma);
  }
}


public interface SomadorEsperado {
  int somaVetor(int[] vetor);
}


import java.util.List;

public class SomadorExistente {
  public int somaLista(List<Integer> lista) {
    int resultado = 0;
    for (int i : lista) resultado += i;
    return resultado;
  }
}
```

## 2. Bridge

### 2.1. Exercício:

Para ilustrar o padrão Bridge, vamos simulá-lo com um exemplo do mundo real. Em lanchonetes você pode comprar refrigerantes de vários tipos (coca-cola, guaraná, etc.) e tamanhos (pequeno, médio, etc.). Se fôssemos criar classes representando estes elementos, teríamos uma proliferação de classes: CocaColaPequena, CocaColaMedia, GuaranaPequeno, etc. Utilizando o padrão Bridge, as classes abaixo foram criadas:

```
public abstract class AbstracaoTamanho {
  protected ImplementacaoRefrigerante refrigerante;

  public AbstracaoTamanho(ImplementacaoRefrigerante refrigerante) {
    this.refrigerante = refrigerante;
  }

  public abstract void beber();
}
```

```
public class TamanhoPequeno extends AbstracaoTamanho {
  public TamanhoPequeno(ImplementacaoRefrigerante refrigerante) {
    super(refrigerante);
  }

  public void beber() {
    System.out.println("Toma um gole de " + refrigerante);
    System.out.println("Acabou o(a) " + refrigerante);
    System.out.println();
  }
}
```

```
public class TamanhoMedio extends AbstracaoTamanho {
  public TamanhoMedio(ImplementacaoRefrigerante refrigerante) {
    super(refrigerante);
  }

  public void beber() {
    System.out.println("Toma um gole de " + refrigerante);
    System.out.println("Toma um gole de " + refrigerante);
    System.out.println("Acabou o(a) " + refrigerante);
    System.out.println();
  }
}
```

A hierarquia acima abstrai o quesito "tamanho" de um refrigerante. Ela se preocupa em prover classes diferentes para tamanhos diferentes. Temos ainda que tratar o quesito "tipo" do refrigerante. Para não proliferar classes, como já enunciado, criamos uma outra hierarquia para tratar o tipo dos refrigerantes.

```
public interface ImplementacaoRefrigerante { }
```

```
public class CocaCola implements ImplementacaoRefrigerante {
  public String toString() {
    return "coca-cola";
  }
}
```

```
public class Guarana implements ImplementacaoRefrigerante {
  public String toString() {
    return "guaraná";
  }
}
```

Experimente as classes acima criando um programa que criará diferentes tipos de refrigerantes de diferentes tamanhos. Chame o método "beber()" e note que o tipo e o tamanho estão certos (o tipo é impresso e o tamanho é notado pela quantidade de goles que se toma antes de acabar o refrigerante). Implemente mais tipos de refrigerante (Fanta, Sprite, etc.) e mais tamanhos (Grande, Tamanho Família, etc.) para experimentar como o padrão Bridge funciona.

### 2.2. Exercício:

Similar ao exercício anterio, utilizar o padrão Bridge para separar duas hierarquias que irão tratar aspectos diferentes de um objeto. Queremos, agora, implementar listas ordenadas e não ordenadas e que podem ser impressas como itens numerados, letras ou marcadores (“*”, “-”, etc.).

Sugestão: defina a abstração (hierarquia da esquerda) como sendo uma interface de uma lista que declara métodos adicionar(String s) e imprimir() e suas implementações (abstrações refinadas) seriam a lista ordenada e não ordenada. Como implementador (hierarquia da direita), defina uma interface que imprime itens de lista, e suas implementações seriam responsáveis por imprimir com números, letras, marcadores, etc.

## 3. Composite

### 3.1. Exercício:

O padrão Composite serve para implementar uma árvore de itens e tratar todos os nós, folhas ou não, de maneira uniforme. Implemente classes que representem um sistema de arquivos, com pastas e arquivos. Pastas possuem nome e diversos arquivos e subpastas. Arquivos possuem nome e tamanho em KB. Seu programa deve navegar pela árvore e imprimir seus itens e tamanhos.

### 3.2. Exercício:

As classes abaixo implementam uma tabela que contém linhas que por sua vez contêm células com conteúdo texto de até 15 caracteres. Altere-as tal que elas fiquem no padrão Composite para que você possa escrever na classe Main um método imprimir() recursivo que recebe um componente genérico e imprime-o e também seus filhos. O método imprimir() deve, no final, imprimir toda a tabela. Como queremos exercitar o padrão Composite, o método imprimir() não pode conhecer as classes específicas Tabela, Linha e Coluna (o método main() pode).

```
public class Tabela {
  private List<Linha> linhas = new ArrayList<Linha>();

  public void adicionar(Linha l) {
    linhas.add(l);
  }
}
```

---

```
public class Linha {
  private List<Celula> celulas = new ArrayList<Celula>();

  public void imprimir() {
    // Imprime a borda lateral.
    System.out.println(" |");

    // Imprime a linha.
    int tamanho = (celulas.size() * 17) + 5;
    char[] linha = new char[tamanho];
    for (int i = 0; i < tamanho; i++) linha[i] = '-';
    System.out.println(" " + new String(linha));
  }

  public void adicionar(Celula c) {
    celulas.add(c);
  }
}
```

---

// Continua na próxima página.

```
public class Celula {
  private String conteudo;

  public Celula(String conteudo) {
    this.conteudo = conteudo;
  }

  public void imprimir() {
    // Limita o conteúdo a exatamente 15 caracteres.
    conteudo = conteudo + "               ";
    conteudo = conteudo.substring(0, 15);

    // Imprime na mesma linha e com borda lateral.
    System.out.print(" | " + conteudo);
  }
}
```

# 4. Decorator

### 4.1. Exercício:

Implemente o interceptador cronômetro apresentado no slide 53. Implemente também outros dois interceptadores: um que imprima uma mensagem de log antes de executar a tarefa ("<data/hora>: mensagem") e outro que verifique se o minuto atual é um número par e, se for, interrompe a execução com uma mensagem de justificativa ("Execução interrompida em minuto par: <hora atual>"). Coloque os interceptadores na ordem log -> verificador-de-minuto -> cronômetro -> componente-concreto.

### 4.2. Exercício:

Crie uma classe NumeroUm que tem um método imprimir() que imprime o número "1" na tela. Implemente decoradores para colocar parênteses, colchetes e chaves ao redor do número (ex.: "{1}"). Combine-os de diversas formas.