Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco		**Data:**	_____
Aluno(a):	_____	**Nota:**	
Professor:	Paulo Maurício Gonçalves Júnior		
Curso:	Tecnologia em Análise e Desenvolvimento de Sistemas		
Disciplina:	Padrões de Projeto de Software Orientado a Objetos		

Exercícios de Fixação

1 Strategy

1. Escreva um programa que exiba uma mensagem diferente para cada dia da semana usando o padrão Strategy.

2. Em arquivo anexo (pacote `strategy.sort`) estão implementadas quatro formas bastante conhecidas de ordenação: *bubble sort*, *insertion sort*, *merge sort* e *quick sort*. Coloque-as no padrão Strategy e escreva um cliente que alterna de estratégia de ordenação livremente. Se estiver curioso, cronometre a execução de cada método para verificar qual é o mais eficiente (deve ser usada uma quantidade grande de números no vetor para perceber a diferença).

2 Observer

3. Em arquivo anexo (pacote `observer.janelas`) está implementado um programa gráfico que contém quatro componentes que manipulam uma mesma classe de modelo. Implemente o padrão Observer utilizando as classes da API Java para este padrão. Você precisará dos seguintes métodos:

   - Alterar o valor de uma caixa de seleção `(JComboBox): getModel().setSelectedItem(valor);`
   - Alterar o valor de uma lista `(JList): setSelectedIndex(valor);`
   - Alterar o valor de um campo texto `(JTextField): setText("" + valor);`
   - Alterar o valor de um slider `(JSlider): setValue(valor)`.

4. Monte uma estrutura multi-níveis de observadores e observáveis. Crie uma classe que representa um sistema de alarme que monitora diversos sensores. O sistema de alarme, por sua vez, é observado por uma classe que representa a delegacia de polícia e outra que representa a companhia de seguros. Quando um sensor detecta o movimento deve alertar o sistema que, em cadeia, alerta a delegacia e a cia. seguros.

3 Decorator

5. Implemente 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.

6. 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.

4 Factory Method

7. Construa um programa que receba como parâmetro um ou mais nomes, cada um podendo estar em um dos seguintes formatos:

   - "nome sobrenome";
   - "sobrenome, nome".

   Escreva duas aplicações de construção de nomes, uma para cada formato. Cada uma deve ser responsável por armazenar os nomes criados e imprimi-los quando requisitado. Implemente o padrão Factory Method de forma que somente a criação do nome seja delegada às aplicações concretas (subclasses). Seu programa deve criar as duas aplicações e construir objetos da classe Nome, que deve ter propriedades nome e sobrenome para armazenar as informações em separado. Os nomes não precisam ser impressos em ordem.

   Ex.:

   ```
   $ java Nomes "McNealy, Scott" "James Gosling" "Naughton, Patrick"
   James Gosling
   Scott McNealy
   Patrick Naughton
   ```

8. Crie dois arquivos texto em um diretório qualquer:

   `publico.txt`
   ```
   Estas são informações públicas sobre
   qualquer coisa.  Todo mundo pode ver
   este arquivo.
   ```

   `confidencial.txt`
   ```
   Estas são informações confidenciais, o
   que significa que você provavelmente
   sabe a palavra secreta!
   ```

   Usando o padrão Factory Method, crie duas provedoras de informação: uma que retorna informações públicas e outra que retorna informações confidenciais. Utilize o provedor confidencial se o usuário informar a senha `''designpatterns''` como parâmetro para o programa, que deve recuperar a informação e exibi-la na tela.

9. Escreva um programa que conte até 10 e envie os números para uma ferramenta de log. Esta ferramenta de log deve ser construída por uma fábrica. Utilize Factory Method para permitir a escolha entre dois tipos de log: em arquivo (`log.txt`) ou diretamente no console. A escolha deve ser por um parâmetro passado ao programa ("arquivo" ou "console").

5 Abstract Factory

10. Crie um "Hello, World" que utilize o padrão Abstract Factory para escolher dentre duas formas de impressão: (a) na tela ou (b) num arquivo chamado `output.txt`. Seu programa deve escolher dentre as duas fábricas aleatoriamente.

11. Considere os seguintes conceitos do mundo real: pizzaria, pizzaiolo, pizza, consumidor. Considere ainda que em uma determinada pizzaria, dois pizzaiolos se alternam. Um deles trabalha segundas, quartas e sextas e só sabe fazer pizza de calabresa (queijo + calabresa + tomate), o outro trabalha terças, quintas e sábados e só sabe fazer pizza de presunto (queijo + presunto + tomate). A pizzaria fecha aos domingos. Tente mapear os conceitos acima para o padrão Abstract Factory (hierarquia de fábricas, hierarquia de produtos, cliente) e implemente um programa que receba uma data como parâmetro (formato dd/mm/yyyy) e imprima os ingredientes da pizza que é feita no dia ou, se a pizzaria estiver fechada, informe isso na tela. Agora imagine que a pizzaria agora faz também calzones (novamente, de calabresa ou presunto). Complemente a solução com mais este componente

6 Singleton

12. Escreva, compile e execute o programa abaixo. Em seguida, troque sua implementação para que a classe Incremental seja Singleton. Execute novamente e veja os resultados.

```
class Incremental {
  private static int count = 0;
  private int numero;
  public Incremental() {
    numero = ++count;
  }
  public String toString() {
    return "Incremental " + numero;
  }
}

public class TesteIncremental {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      Incremental inc = new Incremental();
      System.out.println(inc);
    }
  }
}
```

7 Adapter

13. 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.

14. 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;
  }
```

```
}
```

8 Template Method

15. Exercite o padrão Template Method criando uma classe abstrata que lê uma String do console, transforma-a e imprime-a transformada. A transformação é delegada às subclasses. Implemente quatro subclasses, uma que transforme a string toda para maiúsculo, outra que transforme em tudo minúsculo, uma que duplique a string e a última que inverta a string.

16. Os Comparators de Java podem ser considerados uma variação do Template Method, apesar de não serem feitos via herança. Monte um vetor de doubles e escreva um comparator que compare os números de ponto-flutuante pelo valor decimal (desconsidere o valor antes da vírgula). Em seguida, use `Arrays.sort()` para ordenar o vetor e `Arrays.toString()` para imprimi-la.

9 Composite

17. 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.

18. 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);
  }
}

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);
  }
}
```

10 Bridge

19. 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.

20. Similar ao exercício anterior, 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.

11 Builder

21. Na cadeia de restaurantes fast-food PatternBurgers há um padrão para montagem de lanches de crianças. O sanduíche (hambúrguer ou cheeseburger), a batata (pequena, média ou grande) e o brinquedo (carrinho ou bonequinha) são colocados dentro de uma caixa e o refrigerante (coca ou guaraná) é entregue fora da caixa. A classe abaixo é dada para representar o pedido de um consumidor:

```
import java.util.*;
public class Pedido {
  private Set<String> dentroDaCaixa = new HashSet<String>();
  private Set<String> foraDaCaixa = new HashSet<String>();
  public void adicionarDentroDaCaixa(String item) {
    dentroDaCaixa.add(item);
  }
  public void adicionarForaDaCaixa(String item) {
    foraDaCaixa.add(item);
  }
  public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("Seu pedido:\n");
    buffer.append("Dentro da caixa:\n");
    for (String item : dentroDaCaixa) buffer.append("\t" + item + "\n");
    buffer.append("Fora da caixa:\n");
    for (String item : foraDaCaixa) buffer.append("\t" + item + "\n");
    buffer.append("\nTenha um bom dia!\n\n");
    return buffer.toString();
  }
}
```

Neste caso, o padrão Builder pode ser usado para separar as tarefas do atendente e do funcionário que monta o pedido. Somente este último sabe como montar os pedidos segundo os padrões da empresa, mas é o atendente quem lhe informa quais itens o consumidor pediu. Implemente a simulação do restaurante fast-food descrita acima utilizando o padrão Builder e escreva uma classe cliente que pede um lanche ao atendente, recebe-o do outro funcionário e imprime o pedido.

22. Escreva classes para satisfazer os seguintes papéis do padrão Builder:

    - Client: recebe como parâmetros o nome, endereço, telefone e e-mail de uma pessoa, solicita ao director que construa informações de contato, recupera a informação do builder e imprime;
    - Director: recebe como parâmetro o builder a ser utilizado e os dados de contato. Manda o builder construir o contato;

- Builder: constrói o contato. Existem três tipos de contato e um builder para cada tipo:
  - ContatoInternet: armazena nome e e-mail;
  - ContatoTelefone: armazena nome e telefone;
  - ContatoCompleto: armazena nome, endereço, telefone e e-mail.

A classe que representa o papel client deve ter o método `main()` que irá criar um director e um builder de cada tipo. Em seguida, deve pedir ao director que crie um contato de cada tipo e imprimi-los (use o `toString()` da classe que representa a informação de contato).

12 Chain of Responsibility

23. Crie um programa que simule uma máquina de vendas (de refrigerante, salgadinhos, etc.). A máquina possui diversos "slots", cada um capaz de receber um tipo de moeda diferente: 1, 5, 10 centavos, etc. A máquina deve receber moedas e delegar aos slots que capturem-nas. Quando chegar ao valor do produto (ex.: $ 1,00 o refrigerante, $ 2,50 o chips, etc.), a máquina deve entregar o produto e informar o troco.

24. Crie uma Chain of Responsibility formada por 15 objetos Handlers, cada um com um número de 1 a 15, cujo método `handleRequest()` deve receber um número de requisição e imprimir uma mensagem caso o handler decida tratar aquela requisição (imprima o número do handler). A requisição deve ser tratada caso o handler não tenha tratado uma requisição nos últimos 200 milissegundos. Para verificar isso, use o método `System.currentTimeMillis()`, que retorna o número de milissegundos passados desde o "momento zero" (01/01/1970 00:00:00:000). Caso o handler esteja ainda ocupado com a requisição anterior, passar a requisição para frente na cadeia.