Universidade Estadual do Ceará
Disciplina: Padrões de software
Prof.: Paulo Henrique Maia

Exercícios Padrões de Projeto

1. **Singleton:** Uma boa prática no desenvolvimento de aplicações é o registro de exceções, de informações de controle ou de segurança nas aplicações. Chamamos isso de Log da aplicação. Uma aplicação não precisa ter mais do que uma classe gerando e registrando as informações do log. Nesse contexto, o padrão de projeto Singleton pode ser implementado. Portanto, aplique esse padrão nas classes apresentadas a seguir. A classe Logger usa a variável ativo para indicar se as informações podem ser exibidas, enquanto que a classe Aplicacao é a classe que utiliza dois objetos de tipo Logger.

```
public class Logger {
	private boolean ativo = false;
	public Logger() {}

	public boolean isAtivo() {
		return this.ativo;
	}

	public void setAtivo(boolean b) {
		this.ativo = b;
	}

	public void log(String s) {
		if(this.ativo)
			System.out.println("LOG :: " + s);
	}
}


public class Aplicacao {
	public static void main(String[] args) {
		Logger log1 = new Logger();
		log1.setAtivo(true);
		log1.log("PRIMEIRA MENSAGEM DE LOG");
		Logger log2 = new Logger();
		log2.log("SEGUNDA MENSAGEM DE LOG");
	}
}
```

2. **Strategy:** Considere que sua aplicação agora vai precisar especializar o comportamento de Logger. Teremos agora as classes: LoggerEventViewer – que registra os eventos no EventViewer; LoggerFile – que registra o evento em arquivo de sistema ou LoggerDatabase – que registra o evento em um banco de dados. Considere que a aplicação pode escolher qualquer uma dessas formas de registro de log. Mostre como esse problema pode ser resolvido utilizando o padrão de projeto Strategy. Utilize as classes: Logger e Aplicacao da questão anterior.

3. **Factory Method:** Você foi contratado por uma empresa que desenvolve aplicações para edição e manipulação de imagens. Seu chefe apresentou para você as seguintes classes:

```
public class Visualizador{
	public void Visualizar(){
		Imagem img = new Imagem();
		 img.carregar();
		 img.exibir();
		 img.fechar();
	}
}

public class Imagem{
	public void carregar() {
		 System.out.println("Imagem BMP:");
		 System.out.println("Carregando imagem BMP...");
	}

	public void exibir() {
		 System.out.println("Exibindo imagem por 20 segundos");
	}

	public void fechar() {
		System.out.println("Fechando imagem");
	}
}
```

Essas classes são responsáveis por carregar imagens do tipo BMP. Seu chefe mandou que você alterasse o código de forma que a criação de novos visualizadores de imagem ficasse mais flexível. Crie as seguintes classes: VisualizadorJPG – que visualiza imagens do tipo JPG e ImagemJPG – que trata de imagens JPG. O código deve ser desenvolvido de forma que a criação de qualquer outro visualizador seja rápida e flexível. Para resolver esse problema altere o código acima usando o padrão Factory Method.

4. **Decorator:** Você irá desenvolver um sistema para uma Frozen Yogurt que vende iogurtes montados pelos próprios clientes. Um yogurt contém uma base de yogurt de fruta, topos variados (frutas cristalizadas, castanha,...) e coberturas (calda de chocolate,...). O cliente pode escolher quantos topos e quantas coberturas quiser, sendo que o preço do iogurte final aumenta para cada topo e cobertura adicionados. Utilize o padrão Decorator para modelar e implementar o problema de cálculo do valor de um pedido minimizando o número de classes.

5. **Chain of responsability**: 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.:R $ 1,00 o refrigerante, R$ 2,50 o chips, etc.), a máquina deve entregar o produto e informar o troco.

6. **Command:** Essa questão trata-se de uma abstração de um joystick. Use o padrão Command pra interagir com diferentes jogos. O joystick possui 2 botões: A e B (os direcionais são omitidos pois para o exemplo os botões de ação são suficientes). Os receptores são os jogos, que executarão diferentes ações pra mesma tecla pressionada. Exemplo: o botão X faz

correr no jogo de futebol, acelera no de corrida e chuta alto no de luta. Implemente o código para 3 jogos (futebol, corrida e luta).

7. **Observer**: 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.

8. **Template Method**: 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.

9. **Composite**: Um congresso inscreve participantes, que podem ser um indivíduo ou uma instituição, e cada indivíduo tem um assento reservado no congresso que lhe é dado no ato da inscrição**.** Faça um programa que crie um congresso, adicione participantes (indivíduos e instituições) e mostre uma listagem de todos os inscritos. Se o congresso estiver cheio, não é permitido inscrever mais participantes. Use o padrão Composite.

10. **Bridge**: utilizar o padrão Bridge para separar duas hierarquias que irão tratar aspectos diferentes de um objeto. Queremos 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. **Abstract Factory**: 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.

12. **Adapter**: 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.

```java
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;
		}
	}
```

13.**Memento**: Faça um programa que simule uma calculadora simples com as operações aritméticas básicas (soma, subtração, multiplicação e divisão). Permita que o usuário faça uma operação com vários operadores e operandos. Ex: 4 + 85 – 30. Use o padrão Memento para armazenar os estados do seu sistema, onde cada operação e operando é um memento, e permita o usuário desfazer a última inserção. Por exemplo, se o usuário quiser desfazer a operação anterior, ele verá 8 + 45 -. Ao final, mostre o valor do cálculo solicitado ou uma mensagem de erro caso não seja possível realizá-lo.

14. **Mediator**: Crie um sistema que permita a comunicação entre professor e alunos de uma sala de aula de forma a suportar que novos alunos entrem a qualquer momento. Seu sistema deve prover uma forma de professor enviar um trabalho para todos os alunos, um aluno enviar seu trabalho direto para o professor, e alunos trocarem mensagens entre si. Use o padrão Mediator para implementar o sistema.

15. **Builder**: 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. 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.

```
import java.util.*;

public class Pedido {
	private ArrayList<String> dentroDaCaixa = new ArrayList<String>();
	private ArrayList<String> foraDaCaixa = new ArrayList<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();
	}
}
```

16. **Iterator**: Em um jogo de cartas online, dois participantes possuem, inicialmente, uma coleção de 20 cartas consigo (podem ser repetidas). A cada rodada, os jogadores mostram suas cartas atuais. Quem tiver a carta menor, ganha a carta do seu adversário e a adiciona, juntamente com sua carta, ao final da sua lista de cartas. Caso as cartas sejam iguais, essas cartas vão para o fim da lista de cartas dos seus respectivos jogadores. Ganha o jogo quem finalizar primeiro sua lista. Para jogar este jogo, cada jogador deve implementar um Iterator de cartas, contendo os métodos próximo (pega a primeira da lista), remover (remove a carta em questão), e colocar no final (coloca uma ou mais cartas no fim da fila), e estaVazia (verifica se a lsita está vazia). Os Jogadores A e B se inscreveram no jogo, mas A armazenou suas cartas em um ArrayList, enquanto B as armazenou em uma pilha. Implemente esse jogo e os jogadores usando o padrão Iterator.