INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA MARANHÃO

Departamento de Computação
Sistemas de Informação – 7º Período
Prof. João Carlos Pinheiro

Atividade – Padrões de Software – Etapa 03

Data de entrega: 15/10/2021

1. Responda:
   a) Descreva o padrão de projeto *Flyweight*
   b) Discorra em que situação deve-se utilizá-lo e cite um exemplo.

2. Responda:
   a) Descreva o padrão de projeto Facade
   b) Discorra em que situação deve-se utilizá-lo e cite um exemplo.

3. 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 {
        /* por default não imprime o log */
        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);
        }
}
```

Agora a classe Aplicacao:

```
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");
        }
```

}

Resultado da aplicação antes de aplicar o padrão:

PRIMEIRA MENSAGEM DE LOG

Ao aplicar o padrão, a classe aplicação deverá utilizar o mesmo objeto do tipo Logger nas duas chamadas ao método log, portanto o resultado da aplicação será:

PRIMEIRA MENSAGEM DE LOG
SEGUNDA MENSAGEM DE LOG

Passo 1: Torne o construtor de Logger privado;
Passo 2: Crie uma variável estática logger para conter uma referência única ao objeto de Logger; instancie a variável;
Passo 3: Crie um método estático getInstance.
Passo 4: Na classe Aplicacao, substitua todos os "new Logger()" pelo uso do método estático getInstance criado no passo 3;
Passo 5: Teste o código.

4. 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;
  }
}
```

5. Padrão Command

Um laboratório de pesquisa está automatizando seu processo de testes biológicos. Cada experimento envolve uma sequência de ações automáticas a ser aplicadas em uma cultura de microrganismos, a fim de verificar a reação dos mesmos. Exemplos de ações são: elevação da temperatura ambiente, aplicação

de um produto químico etc. Cada ação é aplicada automaticamente por um robô controlado por um objeto de software que implementa a interface:

```
public interface RoboAction {
    public void execute();
}
```

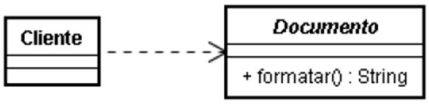
Cada robô executa uma única ação e não é necessário informar parâmetros da ação. Escreva uma classe denominada Experimento cujos objetos são capazes de aplicar um teste envolvendo uma sequência de ações, que podem ser realizadas por diferentes robôs.

Cada objeto Experimento deve guardar uma lista com a sequência de ações a ser executada (cada ação é executada por um único objeto robô o qual ele guarda a referência) e deve implementar operações que permitam adicionar ações (sempre no final da sequência) e executar o teste (na sequência em que as ações foram cadastradas).

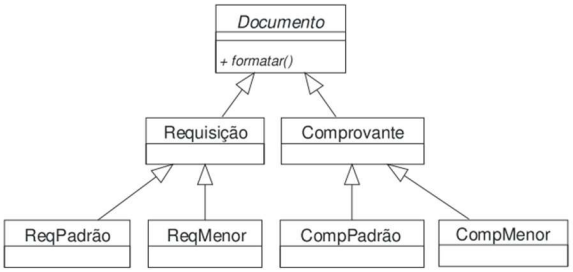
Dica: http://en.wikipedia.org/wiki/Command_pattern

6. Padrão Bridge

Seja Documento uma classe que abstratamente representa documentos do mundo real manipulados por um software. A classe Requisição herda da classe Documento assim como Comprovante, entre muitas outras. Um método abstrato (serviço) fornecido na classe Documento e de interesse dos clientes é formatar(). Esta mensagem, quando enviada à uma instância de Documento deverá retornar uma String cujo conteúdo está pronto para ser exibido. Para os clientes, conforme ilustra a figura abaixo, a dependência está restrita à classe abstrata Documento.

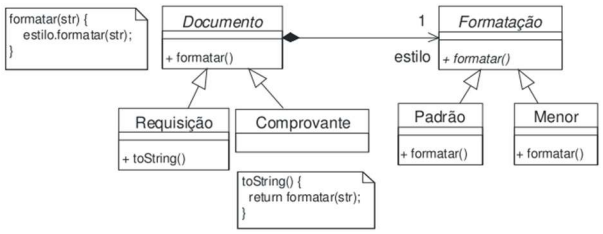


Se desejarmos uma implementação do serviço adequada para a exibição em monitores pequenos como aqueles de dispositivos móveis, provavelmente terminaremos com duas subclasses para Requisição e outras duas para Comprovante. Cada uma delas tratando especificamente cada um dos cenários apresentados, conforme o diagrama abaixo ilustra. Foram empregados os sufixos Padrão e Menor, respectivamente, para representar o monitor convencional e aquele de dimensões reduzidas.



Embora o modelo acima implemente os requisitos apresentados, observa-se facilmente que para cada tipo de documento teremos duas subclasses. O cenário é ainda menos favorável se imaginarmos o surgimento de um novo tipo de monitor, pois teríamos um crescimento significativo do número de classes e, com certeza, uma dificuldade na manutenção delas. Felizmente, os inconvenientes deste cenário podem ser eliminados com o emprego do padrão Bridge.

O que chama a atenção nesta hierarquia é que residem nela tanto a abstração de documentos quanto as implementações correspondentes. O padrão Bridge sugere que a abstração, ou seja, as classes Documento, Requisição e Comprovante sejam mantidas em uma hierarquia distinta daquela da implementação, onde residem as classes Formatação, Padrão e Menor, conforme ilustra o diagrama abaixo.



Implemente o esse diagrama e faça uma pequena classe de teste para simular a criação de documentos com formatações específicas.

Observações Importantes::

Observe que a hierarquia anterior foi fragmentada em duas: uma contendo as abstrações de documentos, relevante para a classe Cliente, e outra contendo a implementação destas abstrações. Neste exemplo, a implementação resume-se ao serviço de formatar documentos para exibição em dois tipos de monitores. No futuro, caso novas formas de formatação sejam criadas, a inserção não irá interferir nos clientes da classe Documento, assim como também não provocarão um crescimento significativo do número de classes e a decorrente dificuldade de manutenção.

Convém ressaltar que a classe Documento e suas subclasses não terão que se preocupar com a formatação para dispositivos específicos, pois este conhecimento foi isolado na hierarquia da direita. O método toString(), conforme a nota, deverá chamar o método formatar() sem conhecer quem estará oferecendo este serviço. No processo de construção de uma instância de uma subclasse de Documento, o construtor pode receber como argumento a formataçao a ser empregada, isto é, uma instância de Formatação, seja esta da classe Padrão ou Menor.