Departamento de Computação – DComp - IFMA
Padrões de Software – 7º Período
Entrega: 24-09-2021

Atividade Etapa 02 – Parte 04

Padrões de Projetos Decorator e Observer

Preâmbulo

O Decorator precisar possuir a mesma interface do objeto que ele está decorando. Para entender melhor a teoria do padrão e estes detalhes, você pode conferir este link:
https://refactoring.guru/design-patterns/decorator

O padrão *Observer* é comumente utilizado por diversas bibliotecas que trabalham com eventos. Muitas tecnologias como o Spring, sistema de eventos no JavaScript, também para desacoplamento entre a camada Model e View no padrão arquitetural MVC.

Para entender mais sobre esse padrão, veja o link: https://refactoring.guru/design-patterns/observer

1. Exercício para consolidação do conhecimento... Uma abordagem simples para implementação do padrão decorator.

   a) Defina o Emissor

```
public interface Emissor {
  void envia(String mensagem);
}
```

   b) Crie a classe EmissorBasico

```
public class EmissorBasico implements Emissor {
  public void envia(String mensagem) {
    System.out.println("Enviando uma mensagem: ");
    System.out.println(mensagem);
  }
}
```

c) Crie uma classe EmissorDecorator para modelar um decorador de emissores.

```
public abstract class EmissorDecorator implements Emissor {
  private Emissor emissor;

  public EmissorDecorator(Emissor emissor) {
    this.emissor = emissor;
  }

  public abstract void envia(String mensagem);

  public Emissor getEmissor() {
    return this.emissor;
  }
}
```

d) Crie um decorador que envia mensagens criptografadas e outro que envia mensagens comprimidas.

```
public class EmissorDecoratorComCriptografia extends EmissorDecorator {

  public EmissorDecoratorComCriptografia(Emissor emissor) {
    super(emissor);
  }

  void envia(String mensagem) {
    System.out.println("Enviando mensagem criptografada: ");
    this.getEmissor().envia(criptografa(mensagem));
  }

  private String criptografa(String mensagem) {
    String mensagemCriptografada = new StringBuilder(mensagem).reverse().toString();
    return mensagemCriptografada;
  }
}
```

```
public class EmissorDecoratorComCompressao extends EmissorDecorator {

  public EmissorDecoratorComCompressao(Emissor emissor) {
    super(emissor);
  }

  void envia(String mensagem) {
    System.out.println("Enviando mensagem comprimida: ");
    String mensagemComprimida;
    try {
      mensagemComprimida = comprime(mensagem);
    } catch (IOException e) {
      mensagemComprimida = mensagem;
    }
    this.getEmissor().envia(mensagemComprimida);
  }

  private String comprime(String mensagem) throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DeflaterOutputStream dout = new DeflaterOutputStream(out, new Deflater());
    dout.write(mensagem.getBytes());
    dout.close();
    return new String(out.toByteArray());
  }
}
```

e) Teste os decoradores.

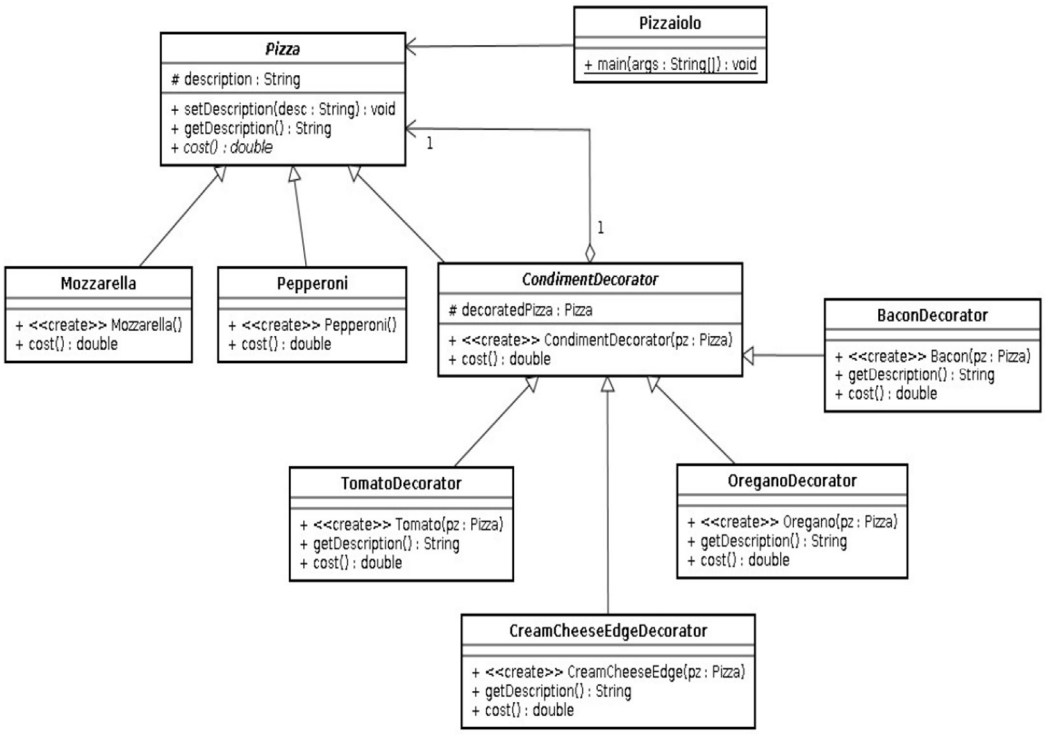
```
public class TesteEmissorDecorator {

  public static void main(String[] args) {
    String mensagem = "";

    Emissor emissorCript = new EmissorComCriptografia(new EmissorBasico());
    emissorCript.envia(mensagem);

    Emissor emissorCompr = new EmissorComCompressao(new EmissorBasico());
    emissorCompr.envia(mensagem);

    Emissor emissorCriptCompr = new EmissorComCriptografia(new EmissorComCompressao(↩
        new EmissorBasico()));
    emissorCriptCompr.envia(mensagem);
  }
}
```

Você conseguiu observar alguma vantagem nessa abordagem. Comente!

2. Problema – “{[(1)]}”

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.

3. Utilizando o padrão decorator, crie uma forma de acrescentar novos condimentos à objeto Pizza, sem alterar as classes já implementadas: Pizza, Mozzarella, Pepperoni e Pizzaiolo. Com base no diagrama de classes a seguir, implemente a classe abstrata CondimentDecorator e todas as sub-classes decoradoras concretas.



Considere a seguinte tabela de preços:

Pizzas

Pizza Mozzarella --- 11.90

Pizza Pepperoni --- 14.90

Condimentos

Bacon --- 0.80

Oregano --- 0.50

Tomato --- 0.10

CreamCheeseEdge --- 1.20

A saída do seu sistema deve ser:

Pizza --- Valor

Mozzarella Pizza --- 11.9

Mozzarella Pizza, Tomato --- 12.0

Mozzarella Pizza, Tomato , CreamCheeseEdge --- 13.2

Pepperoni Pizza --- 14.9

Pepperoni Pizza, Oregano, Bacon --- 16.2

Pepperoni Pizza, Oregano, Bacon , Tomato --- 16.3

4. Exercício para consolidação do conhecimento... Uma abordagem simples para implementação do padrão Observer

a) Defina a classe Acao

```
public class Acao {
    private final String codigo;
    private double valor;

    public Acao(String codigo, double valor) {
        this.codigo = codigo;
        this.valor = valor;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public String getCodigo() {
        return codigo;
    }
}
```

b) Para notificar os interessados sobre as alterações nos valores da ação, devemos registrar os interessados e notificá-los. Para padronizar a notificação dos interessados, criemos a interface AcaoObserver.

```
public interface AcaoObserver {
   void notificaAlteracao (Acao acao);
}
```

c) Altere a classe Acao para registrar os interessados e notificá-los sobre a alteração no valor da ação

```
public class Acao {
    private String codigo;
    private double valor;
    private Set<AcaoObserver> interessados = new HashSet<>();

    public Acao(String codigo, double valor) {
        this.codigo = codigo;
        this.valor = valor;
    }

    public void registraInteressado(AcaoObserver interessado) {
        this.interessados.add(interessado);
```

```
    }

    public void cancelaInteresse(AcaoObserver interessado) {
        this.interessados.remove(interessado);
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
        for (AcaoObserver interessado : this.interessados) {
            interessado.notificaAlteracao(this);
        }
    }
}
```

d) Defina a classe Corretora e implemente a interface AcaoObserver para que as corretoras sejam notificadas sobre as alterações nos valores das ações

```
public class Corretora implements AcaoObserver {
    private String nome;

    public Corretora(String nome) {
        this.nome = nome;
    }

    public void notificaAlteracao(Acao acao) {
        System.out.println(" Corretora " + this.nome + " sendo
                                                notificada :");
        System.out.println("A ação " + acao.getCodigo()
                + " teve o seu valor alterado para " +
                acao.getValor());
    }
}
```

e) Faça uma classe para testar as classes Corretora e Acao.

```
public class TestaObserver {
    public static void main(String[] args) {
        Acao acao = new Acao("VALE3", 45.27);
        Corretora corretora1 = new Corretora(" Corretora1 ");
        Corretora corretora2 = new Corretora(" Corretora2 ");
        acao.registraInteressado(corretora1);
        acao.registraInteressado(corretora2);

        acao.setValor(50);
    }
}
```

Você conseguiu observar alguma vantagem nessa abordagem. Comente!

5. Jogo de Bingo com verificação automática

No Bingo, seu sistema (BingoSystem) tem o controle das cartelas distribuídas aos participantes. Portanto, ao mesmo tempo em que seu sistema sorteia um número novo, ele detecta se algum dos participantes completou sua cartela. Isto é interessante para evitar fraudes e agilizar o processo de premiação - não será mais necessário verificar manualmente número por número do participante que alega ter ganhado o prêmio.

Você deve gerar as cartelas (BingoCard) aleatoriamente e cadastrá-las no BingoSystem.

Dica para geração de números aleatórios:

```
public static void main(String[] args) {
  Random random = new Random();
  // gera um numero aleatório dentro com conjunto (0, 1, 2, 3)
  int numero = random.nextInt(4);
  System.out.println(numero);
}
```

Você deverá utilizar o gerador de números aleatórios, tanto para cartela quanto para o sorteio dos números do bingo.

Dica: o padrão Observer pode ser incorporado à solução.

BingoSystem seria o Subject e BingoCard seria o Observer.

Uma dica de roteiro para facilitar a codificação...

Fique à vontade para variar a sua implementação:

1. Implemente a estrutura básica do padrão Observer: Interfaces Subject e Observer.

2. Implemente BingoCard.

    a. BingoCard é um Observer;

    b. Atributos básicos:

        i. int cardId;

        ii. int [] numbers;

        iii. BingoSystem subject;

    c. O construtor de BingoCard inicializa as variáveis e constrói a cartela randomicamente; Parâmetros do construtor:

i. BingoSystem subject;

ii. int cardId;

iii. int numberOfSlots; -> número de casas de cada cartela

iv. int maxNumber; -> maior número possível da cartela

d. O método update, herdado de Observer, deve receber como argumento um número sorteado, e atualizar uma *dada casa* com -1 caso o valor sorteado esteja na cartela. Assim podemos controlar se o jogador bateu ou não o bingo.

e. Implemente um método boolean didIWin(), que verifica se a cartela ganhou o bingo;

f. Finalmente, implemente os métodos getCardId() e toString();

3. Implemente BingoSystem.

a. BingoSystem é um Subject;

b. Atributos básicos:

i. BingoSystem uniqueInstance; singleton

ii. List<Observer> bingoCards; -> referente ao padrão observer;

iii. int numberDrawn; -> num sorteado

iv. boolean gameEnded; -> inidica se alguém bateu

c. O construtor de BingoSystem é privado (singleton) e apenas inicializa a lista de bingoCards;

d. Métodos de BingoSystem:

i. getInstance(); -> retorna uma instância de BingoSystem

ii. subscribe(Observer), notifyObservers() -> métodos do padrão observer;

iii. startBingo(maxNumber) -> sorteia os números e avisa aos jogadores, ou seja, às cartelas (observadores);

iv. bingo(String message) -> método que o jogador/cartela deve chamar quando bater o bingo. A mensagem do jogador deve conter o cardId, identificador de sua cartela.

4. Main: crie um jogo de Bingo com 5 cartelas, onde cada cartela tem 6 números, e cujo número máximo do sorteio é 50, ou seja, os números sorteados variam dentro do intervalo [0,50].