Departamento de Computação – DComp - IFMA
Padrões de Software – 7º Período
Entrega: 04-05-2023

Atividade Etapa 02

Padrões de Projeto

Padrões Builder e Step Builder

1. Escreva classes para satisfazer os seguintes papéis do padrão Builder: (1 pt)

   - **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)

2. Considere a classe Livro apresentada a seguir. (1,0 pt)

```
public class Livro {
  private String nomeNacional;
  private int ano;
  private List<String> autores;
  private int edicao;
  private String cidade;
  private String editora;
  private String nomeOriginal;
  private List<String> tradutores;
  private int paginas;
  private long isbn;

  public Livro(String nomeNacional, int ano, List<String> autores,
               int edicao, String cidade, String editora, String
```

```
nomeOriginal,
                List<String> tradutores, int paginas, long isbn) {
    this.nomeNacional = nomeNacional;
    this.ano = ano;
    this.autores = autores;
    this.edicao = edicao;
    this.cidade = cidade;
    this.editora = editora;
    this.nomeOriginal = nomeOriginal;
    this.tradutores = tradutores;
    this.paginas = paginas;
    this.isbn = isbn;
  }

  // getters e setters omitidos
}
```

Observe o construtor da classe Livro. Veja como fica complicado escrever tantos parâmetros. Uma solução seria a utilização do padrão de projeto Step Builder, que seria responsável pela construção do objeto em passos pré-definidos, conforme exemplo:

```
Livro livro = new Livro.LivroBuilder("Java, como programar")
        .publicadoEm(2003)
        .dosAutores("H. M. Deitel", "P. J. Deitel")
        .edicao(4)
        .cidade("Porto Alegre")
        .editora("Bookman")
        .nomeOriginal("Java How to Program")
        .tradutores("Carlos Arthur Lang Lisbôa")
        .paginas(1386)
        .isbn(9788536301235L)
        .build();
```

Desenvolva a solução requisitada, utilizando o padrão de projeto Step *Builder* e *fluent interface*.

Padrão State

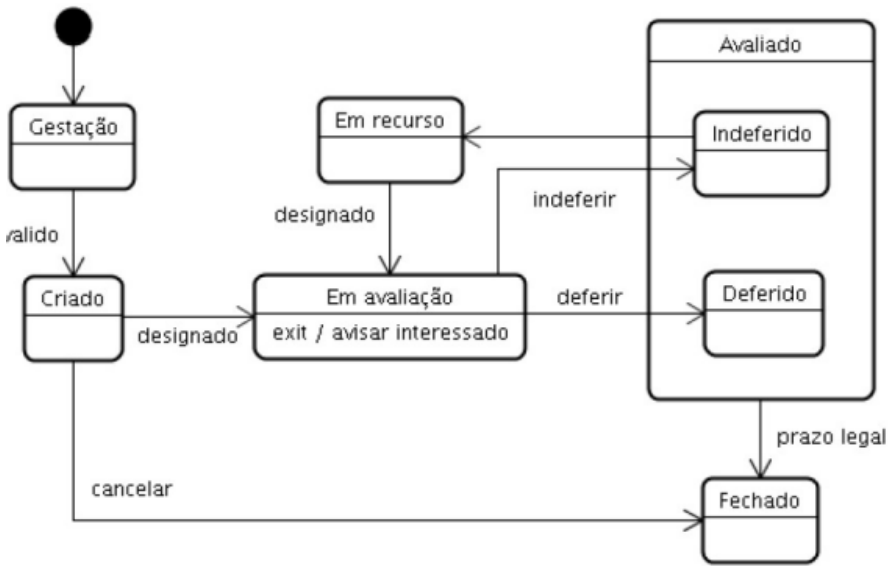
Preâmbulo

O padrão State é empregado para implementar diagramas de transição de estados. Em tais diagramas existem estados e transições entre eles. Eventos provocam transições que, por sua vez, disparam a execução de ações. Tanto as transições quanto as ações dependem do estado ativo no momento em que o evento em questão é gerado.

A implementação sugerida deste padrão faz uso de uma interface que deverá definir todos as reações possíveis para todos os estados a serem contemplados. Seja Estado

esta interface. Cada reação pode ser definida na interface através de operações que representam os eventos possíveis. Para todo e qualquer possível evento haverá uma operação correspondente nesta interface. Será a execução destas operações que irão provocar mudanças do estado corrente. Cada estado possível do diagrama será representado por uma classe que implementa a interface Estado e, naturalmente, possui uma implementação para cada operação (evento). É cada implementação específica que identificará como é que o estado em questão reage ao evento gerado.

3. Faça uso do padrão State para implementar o comportamento registrado no diagrama abaixo. Este comportamento correspondente ao comportamento esperado para toda instância da classe Processo. Crie a classe Processo e implemente o comportamento de tal forma que, dada a ocorrência de um evento, possivelmente ocorre a transição para um dado estado, juntamente com a execução das ações julgadas oportunas. Neste caso, apenas uma ação deve ser executada como resultado da saída do estado "Em avaliação". Implemente a ação "avisar interessado" por meio de uma simples mensagem produzida na saída padrão. Um cenário real poderia exigir o envio de correspondência eletrônica para o email do interessado, o que não é exigido neste exercício. Por fim, observe que o estado "Avaliado" é uma composição dos estados "Indeferido" e "Deferido". São estes dois últimos que deverão ser tratados. Ou seja, "Avaliado" é apenas uma abstração que não precisa ser tratada da perspectiva de implementação. Observe que, transcorrido o prazo legal após avaliação de um processo, este é conduzido ao estado "Fechado", independente se o estado é "Indeferido" ou "Deferido". (1,5 pt)



Padrão Factory Method

4. Queremos montar uma lanchonete, onde vários sanduíches diferentes podem ser feitos. Um sanduíche básico contém: duas fatias de pão, uma fatia de queijo, uma fatia de presunto e salada (estrutura base de qualquer sanduiche). No entanto, existem variações do sanduiche básico de acordo com os tipos diferentes de ingredientes (veja quadro). Tendo o código para o sanduíche base, aplique o padrão de projeto *Factory Method* para que os sanduíches abaixo possam ser feitos na nossa lanchonete. (0.5pt)

		Sanduiches		
Ingredientes		**Lanchonete CG**	**Lanchonete JP**	**Lanchonete RT**
Pão	**Integral**	X		
	Francês		X	
	Bola			X
Queijo	**Prato**	X		
	Mussarela		X	
	Cheddar			X
Presunto	**De Frango**	X	X	
	De Peru			X
Salada	**Com verdura**		X	
	Sem verdura	X		X

Padrão Abstract Factory

Preâmbulo

Se em algum momento tivermos objetos fortemente relacionados em nosso sistema, podemos separar a criação deles em uma única classe, para garantir que eles sejam criados sempre em conjunto com o seu "par" correto. Essa família de classes estendem o que é conhecida como Abstract Factory, por ser uma fábrica abstrata, que pode criar famílias de objetos, e não apenas um objeto específico.

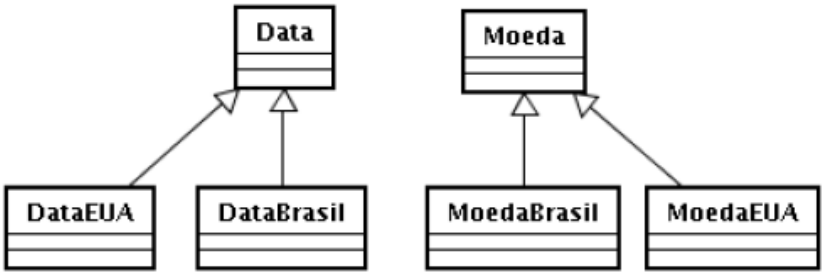
Veja exemplos de implementação nesse link:

https://refactoring.guru/design-patterns/abstract-factory

5. A implementação deste padrão envolve o emprego de várias classes, em um cenário relativamente complexo de interação entre as classes. Suponha a existência de uma classe Data e a classe Moeda. A primeira representa uma data qualquer, por exemplo, 7 de setembro de 2004. A outra representa uma

quantidade em dinheiro, por exemplo, R$25,00. Em ambos os casos você deve ter observado que a forma empregada para exibir a data e a quantia em dinheiro são típicas do Brasil. Para um país de língua inglesa outra forma deveria ser utilizada. Ou seja, September 07, 2004 para a data e $25.00 para a moeda (estou assumindo a paridade de um dólar para um real).

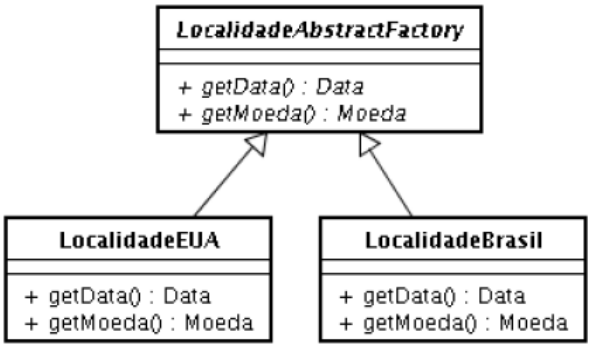
O que é preciso neste caso, é a implementação das classes Data e Moeda conforme a localidade. Para o Brasil, por exemplo, temse implementações distintas daquelas para países de língua inglesa, conforme a figura abaixo.



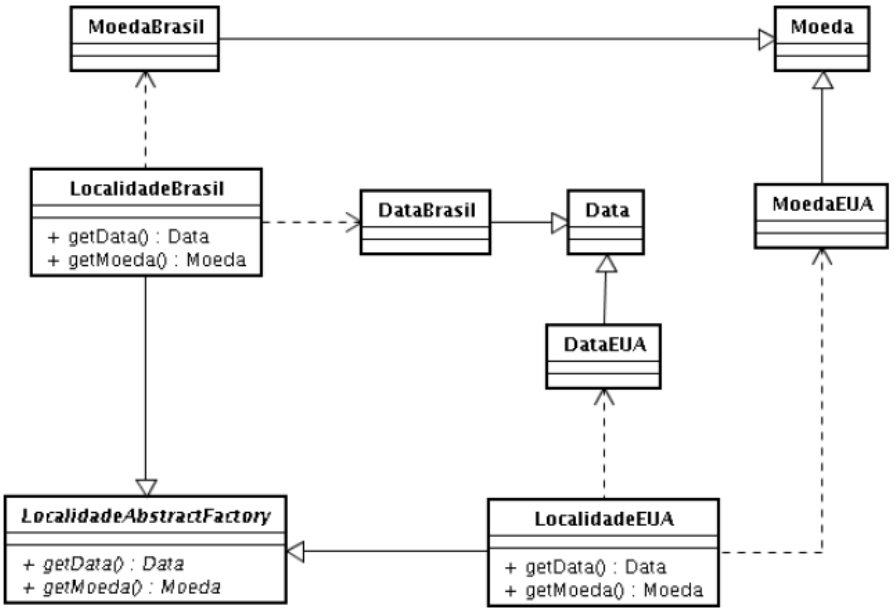
Para uma aplicação cliente, o padrão Abstract Factory pode ser empregado para evitar que a aplicação use, por exemplo, a implementação para o Brasil da classe Data e a implementação para um país de língua inglesa para a classe Moeda, misturando data de um com moeda de outro. O que é preciso, neste caso, é recuperar instâncias de classes correlatas.

Uma classe que implemente os métodos getMoeda e getData podem identificar a localidade, através de um teste, por exemplo, e retornar a instância da classe identificada. Contudo, esta abordagem é inapropriada, principalmente à medida que o número de opções crescer. Dois países não oferecem tantos desafios, mas não podemos dizer o mesmo para centenas deles. A manutenção seria dificultada, mesmo todo este código estando devidamente confinado em uma única classe.

Uma abordagem mais atrativa envolve o emprego de uma classe abstrata, LocalidadeAbstractFactory, nossa fábrica abstrata de objetos. Esta classe deve definir os métodos getMoeda e getData, mas não implementálos, tarefa que seria delegada para as subclasses concretas, tantas quantas forem as opções existentes. Para o nosso exemplo e para a localidade do Brasil podese criar a classe LocalidadeBrasil, para a localidade EUA podese criar a classe LocalidadeEUA, conforme ilustra a figura abaixo



A implementação dos métodos getMoeda e getData é trivial, simplesmente retornando uma instância da classe MoedaBrasil e DataBrasil, respectivamente. Naturalmente, MoedaBrasil herda da classe Moeda e DataBrasil herda da classe Data. Estas classes de sufixo Brasil, convém relembrar, devem implementar o formato empregado no Brasil para a exibição de valores monetários e de datas. Algo similar seria obtido com outra localidade qualquer, conforme ilustra a figura.



A solução apresentada no diagrama acima faz uso do padrão Abstract Factory. Para a aplicação cliente, convém ressaltar, há dependências para as classes LocalidadeAbstractFactory, Moeda e Data. O cliente depende de LocalidadeAbstractFactory porque é a partir desta que as instâncias adequadas de Moeda e Data são obtidas. Naturalmente, código cliente depende das classes Moeda e Data, pois são estas que são efetivamente fornecem a funcionalidade

desejada. Obserque que o código cliente, ao fazer uso de uma instância de Moeda não sabe se se trata de uma instância da classe MoedaBrasil ou MoedaEUA, que foi retornada por uma classe derivada de LocalidadeAbstractFactory, também desconhecida da aplicação.

Implemente o modelo comentado acima, onde as seguintes restrições deverão ser satisfeitas. (1pt)

i. Uma classe Cliente deverá representar código cliente.

ii. Crie a classe Factory a partir da qual o método newLocalidade() deverá retornar uma instância de LocalidadeAbstractFactory. A decisão deverá vir da configuração de um arquivo de propriedades, conhecido pela classe Factory. A propriedade localidade deverá indicar Brasil para a localidade do Brasil ou EUA para a localidade dos EUA.

iii. A classe Cliente, ao fazer uso de Data e Moeda, não deverá conhecer a implementação da classe LocalidadeAbstractFactory assim como também não conhecerá quem implementa Data e Moeda.

iv. A funcionalidade depositada em Data é apresentar, conforme a localidade, a dia da semana corrente. Por exemplo, para o Brasil pode ser “Seg”, enquanto para a localidade EUA seria “Mon”.

v. A funcionalidade de Moeda é apresentar um valor qualquer na moeda em questão. Por exemplo, para o Brasil, R$10,0 é uma saída correta, enquanto para os EUA um valor correto seria $5.00. Esta funcionalidade deverá ser obtida, tanto da classe Moeda quanto da classe Data através do método toString(), herdado de Object.

Preâmbulo

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

6. Exercício para consolidação do conhecimento. Uma abordagem simples para implementação do padrão Observer (0,5 pt)

   a) A classe Ação

```
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!

7. 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. (2pt)

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].

Padrão Decorator

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

8. **Exercício para consolidação do conhecimento. Uma abordagem simples para implementação do padrão decorator. (0,5 pt)**

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.

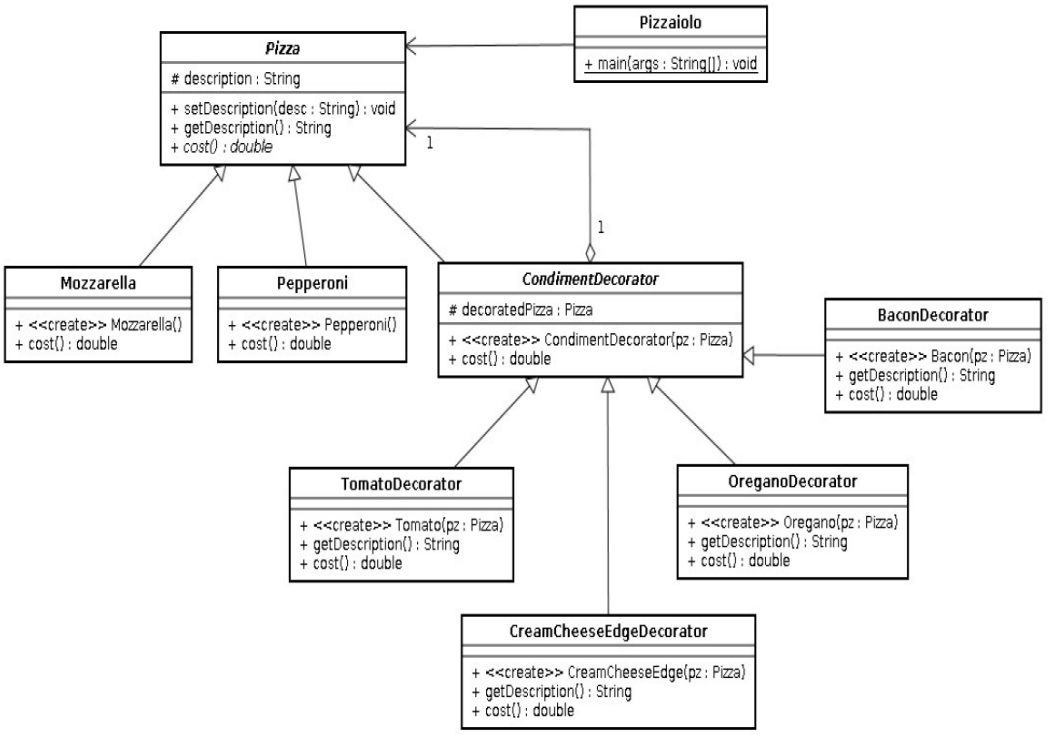
```
public class TesteEmissorDecorator {

  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!

9. **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. (1pt)

10. **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. (1pt)



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