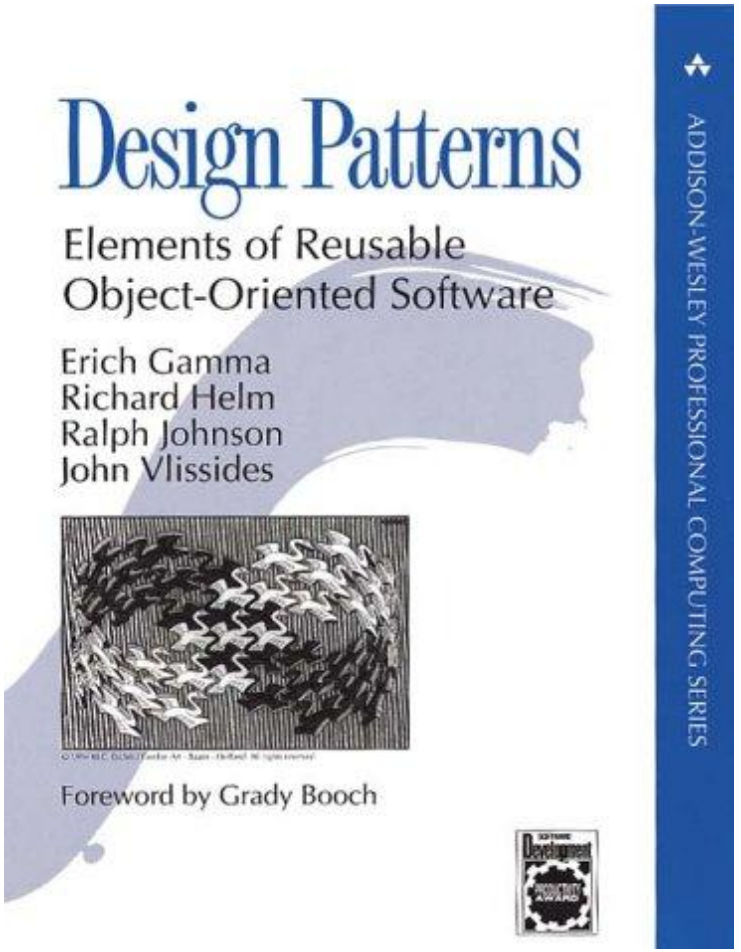
Padrões de Projeto

Padrões Gang of Four

Fernando Pedrosa – fpedrosa@gmail.com

Bibliografia

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – **Padrões de Projeto**. Editora: Bookman Companhia



Motivação

- A orientação a objetos, por si só, não garante sistemas reusáveis e extensíveis
- Profissionais experientes conseguem projetar bons sistemas, novatos não
- Primeiro aprende-se as regras
  - Algoritmos, estruturas, linguagens
- Depois os princípios
  - Projeto estruturado, Projeto OO

Motivação

- Mas, sistemas complexos necessitam de projetos robustos, que foram postos à prova
  - Estes **Padrões de Projeto** têm que ser compreendidos, lembrados e usados
- Padrões de Projeto representam soluções comprovadas para problemas recorrentes em desenvolvimento de software

Evolução

- A ideia original surgiu em 1979, na Arquitetura e Engenharia Civil
- Christopher Alexander, arquiteto, queria melhorar o processo de projeto de edifícios e áreas urbanas
- Hoje, projetos de engenharia civil seguem padrões estabelecidos
  - Arcos, colunas, portas, janelas, etc. – a solução para tudo isso já é bem conhecida

Evolução

"Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para aquele problema, de tal maneira que pode-se usar essa solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes"

Christopher Alexander, sobre padrões na arquitetura e engenharia civil

Evolução

- Na Engenharia de Software, quatro autores (Gang of Four) se basearam em Christopher Alexander para criar Padrões de Projeto de software
- Em 1994 descreveram 23 padrões em seu livro
  - Hoje ele já está na sua trigésima sexta edição
  - Mais de 500 mil cópias vendidas, traduzido para 13 línguas

Padrões de Projeto

"Descrição de uma solução para resolver um problema genérico de projeto em um contexto específico. [...] Um padrão de projeto dá nome, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la reutilizável"

Erich Gamma, et. al, sobre padrões de projeto de software

Benefícios

- Padrões capturam a estrutura estática e a colaboração dinâmica entre objetos parcitipantes no projeto de sistemas
- São especialmente bons para descrever como e por que resolver problemas não funcionais
- Facilitam o reuso de soluções arquiteturais que deram certo antes
- Aumentam a coesão, diminuem o acoplamento

Elementos

- Padrões de projeto são compostos por quatro elementos essenciais
  - Nome do padrão
  - Problema a ser resolvido
  - Solução dada pelo padrão
  - Consequências

Nome

- Um identificador utilizado para resumir
  - O problema em questão
  - Suas soluções
  - Suas consequências
- Aumenta o vocabulário e melhora a comunicação

  “A parte mais difícil de programação é dar bons nomes às variáveis”

Problema

- Descreve quando aplicar o padrão
- Explica o problema e seu contexto
- Pode conter uma lista de pré condições que precisam estar presentes antes de levar em consideração a aplicação do padrão

Solução

- Descrição abstrata de como o padrão resolve o problema em questão
- Descreve os elementos que compõem
  - Relacionamentos
  - Responsabilidades
  - Colaborações
- Inclui algum exemplo concreto de implementação
  - Porém o padrão deve ser adaptado ao seu contexto específico

Consequências

- Vantagens e desvantagens de aplicar o padrão
- Esta seção serve para
  - Avaliar várias alternativas de padrões
  - Entender os custos e desafios
  - Entender os benéficios de aplicar o padrão
- Inclui análise de impacto envolvendo
  - Flexibilidade
  - Extensibilidade
  - Portabilidade

Padrões de Projeto NÃO são

- Soluções prontas, que podem ser codificadas diretamente nas classes e reutilizadas sem adaptação (como API's, coleções de código, etc.)
- Projetos para contextos abrangentes e complexos (uma aplicação ou subsistema inteiro)
  - São aplicáveis em situações específicas

Classificação

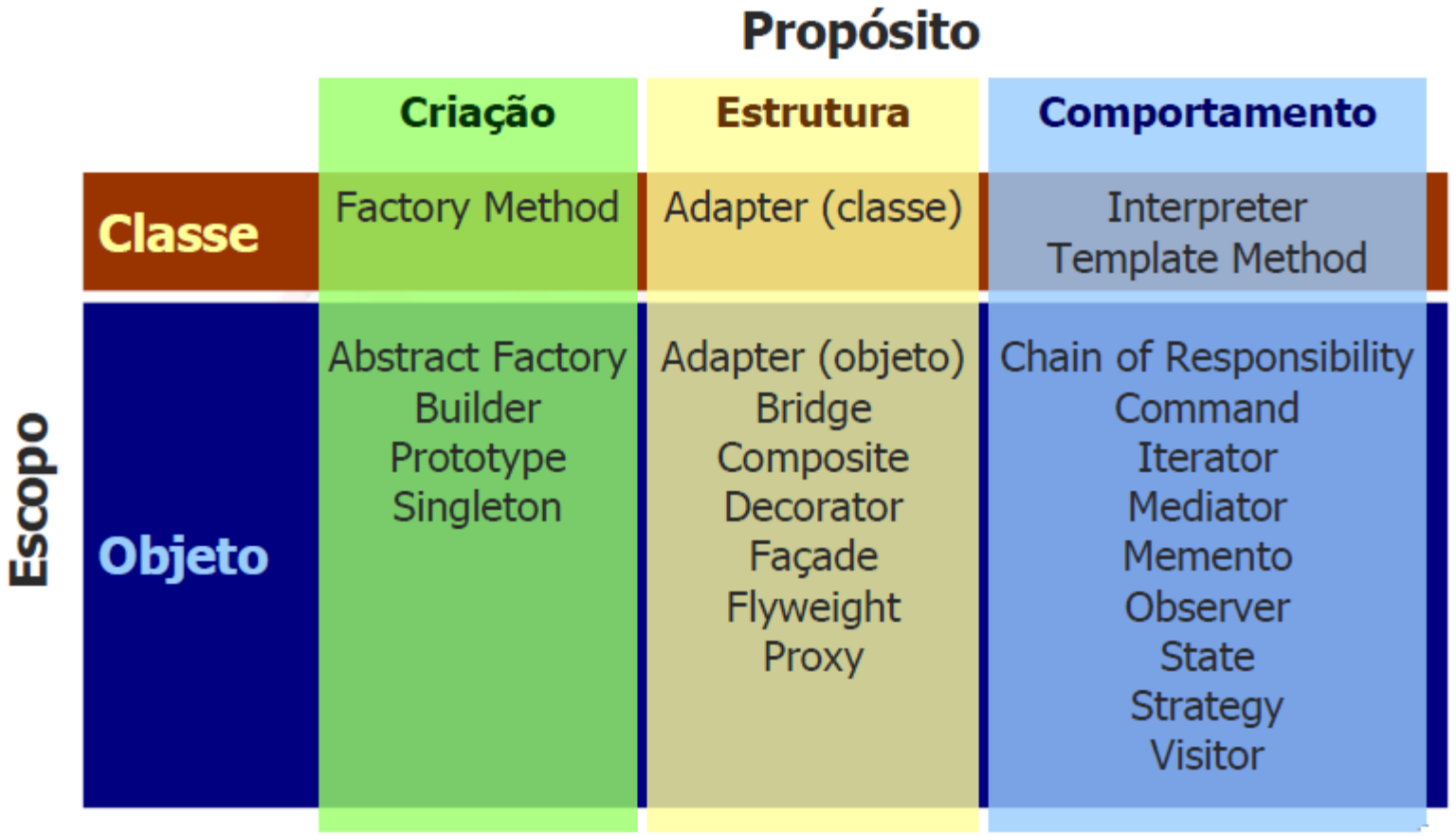
Podem ser classificados por **propósito**

- Padrões de Criação
  - Abstraem o processo de criação de objetos a partir da instanciação de classes
- Padrões Estruturais
  - Tratam da forma como classes e objetos estão organizados para formar estruturas maiores
- Padrões Comportamentais
  - Preocupam-se com algoritmos e responsabilidades dos objetos

Classificação

- Podem ser subclassificados por **escopo**
- Padrões de Classes
  - Tratam de relações entre classes e subclasses (herança)
  - São estáticos, definidos em tempo de compilação
- Padrões de Objetos
  - Tratam das relações entre objetos, que podem mudar em tempo de execução

Classificação



Exercícios [1]

(INFRAERO – FCC 2009)

[52] Os padrões de projeto (design patterns)

I. foram testados: refletem a experiência e conhecimento dos desenvolvedores que utilizaram estes padrões com sucesso em seu trabalho;

II. são reutilizáveis: fornecem uma solução pronta que só não pode ser adaptada para diferentes problemas;

III. são expressivos: formam um vocabulário comum para expressar grandes soluções sucintamente;

IV. facilitam o aprendizado: reduzem o tempo de aprendizado de uma determinada biblioteca de classes;

V. diminuem retrabalho: quanto mais cedo são usados, menor será o retrabalho em etapas mais avançadas do projeto.

Está INCORRETO o que consta APENAS em

(A) I. (B) II. (C) III. (D) IV. (E) V

Exercícios [1]

(DECEA – CESGRANRIO 2009)

[33] A equipe de desenvolvimento de sistemas de uma empresa utiliza padrões de projetos (design patterns) em seus projetos orientados a objetos. Nesse contexto, NÃO é uma característica o(a)

(A) uso de soluções específicas e distintas para projetos similares.

(B) identificação de problemas comuns de projeto de software.

(C) utilização de soluções testadas e bem documentadas.

(D) utilização eficiente de herança, polimorfismo e composição.

(E) facilidade na conversão de um modelo de análise em um modelo de implementação.

Exercícios [1]

(SERPRO – CESPE 2010)

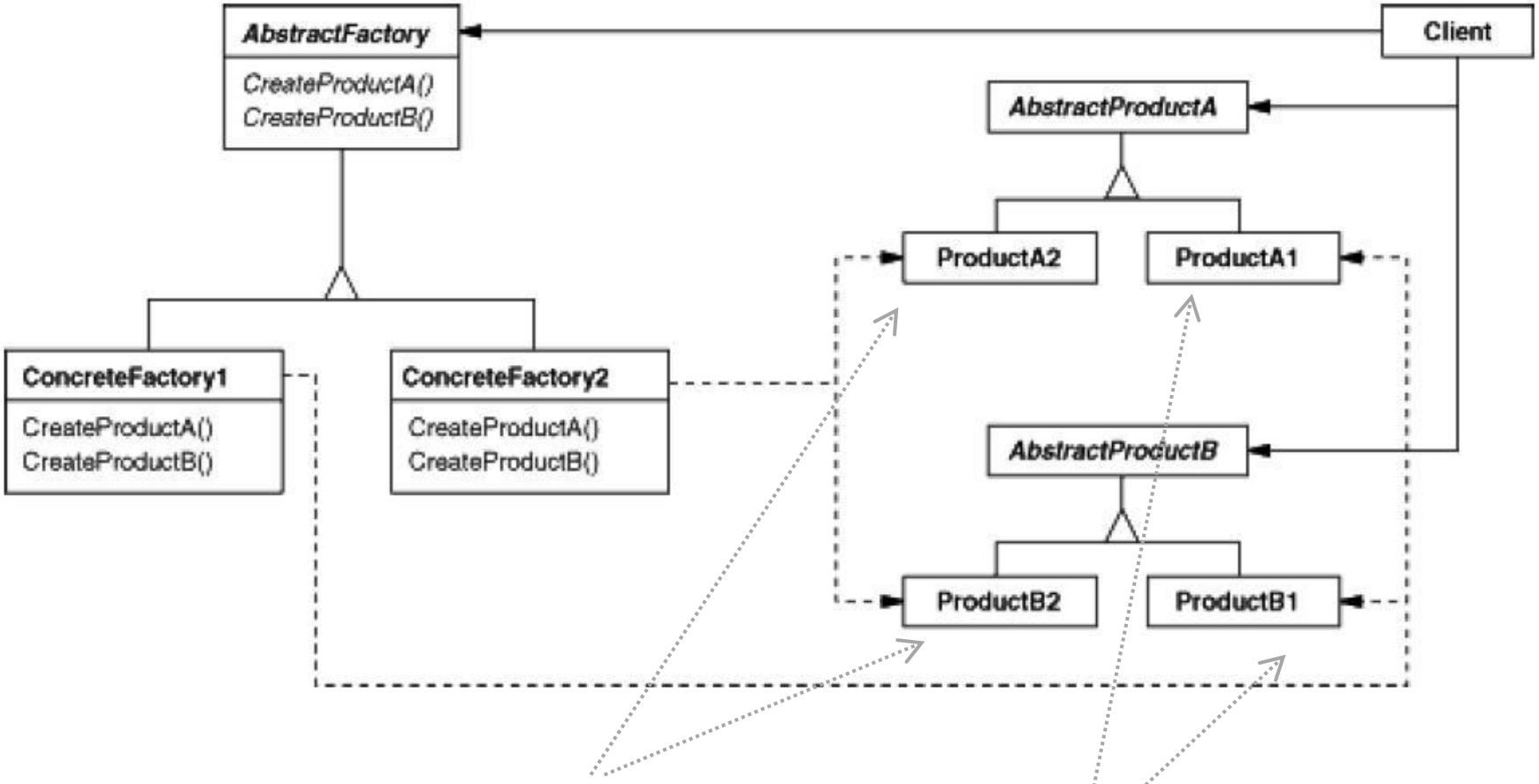
[82] O emprego de padrões de projeto reusáveis, como façade, builder e singleton, é uma prática com nível inferior de abstração, quando comparado ao emprego de estilos arquiteturais de software, como camadas, cliente-servidor e peer-to-peer.

Padrões Criacionais

Abstract Factory

- Proporciona uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas
- Use Abstract Factory quando:
  - O sistema deve ser configurado com uma de múltiplas famílias de produtos
  - Estes produtos relacionados são projetados para serem utilizados juntos, e você quer garantir essa restrição

Abstract Factory



Família de produtos #2

Família de produtos #1

Abstract Factory

```
interface GUIFactory {
    public Button createButton();
}

class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}


class OSXFactory implements GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }
}
```

Fábrica abstrata

Fábricas concretas

Produto abstrato

```
interface Button {
    public void paint();
}

class WinButton implements Button {
    public void paint() {
        System.out.println("I'm a WinButton");
    }
}

class OSXButton implements Button {
    public void paint() {
        System.out.println("I'm an OSXButton");
    }
}
```

Produtos concretos

Abstract Factory (executando)

```
class Application {
    public Application(GUIFactory factory){
        Button button = factory.createButton();
        button.paint();
    }
}

public class ApplicationRunner {
    public static void main(String[] args) {
        new Application(createOsSpecificFactory());
    }

    public static GUIFactory createOsSpecificFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
}
```

A fábrica é escolhida em tempo de execução

Exercícios [2]

(CENSIPAM – CESPE 2006)

[57-I] Um software está sendo desenvolvido e algumas decisões foram tomadas quando do seu projeto. A seguir, tem-se as decisões I, II e III que deverão ser atendidas usando-se padrões de projeto (design patterns) adequados.

I Os formatos dos dados de entrada serão validados por métodos nas classes que os modelam. Por exemplo, para validar uma senha, a classe Senha terá um método apropriado. Como o software será fornecido para clientes cujos dados terão diferentes formatos, essas classes devem ser substituídas em conjunto e essas substituições não devem resultar em alterações nos códigos que instanciam essas classes

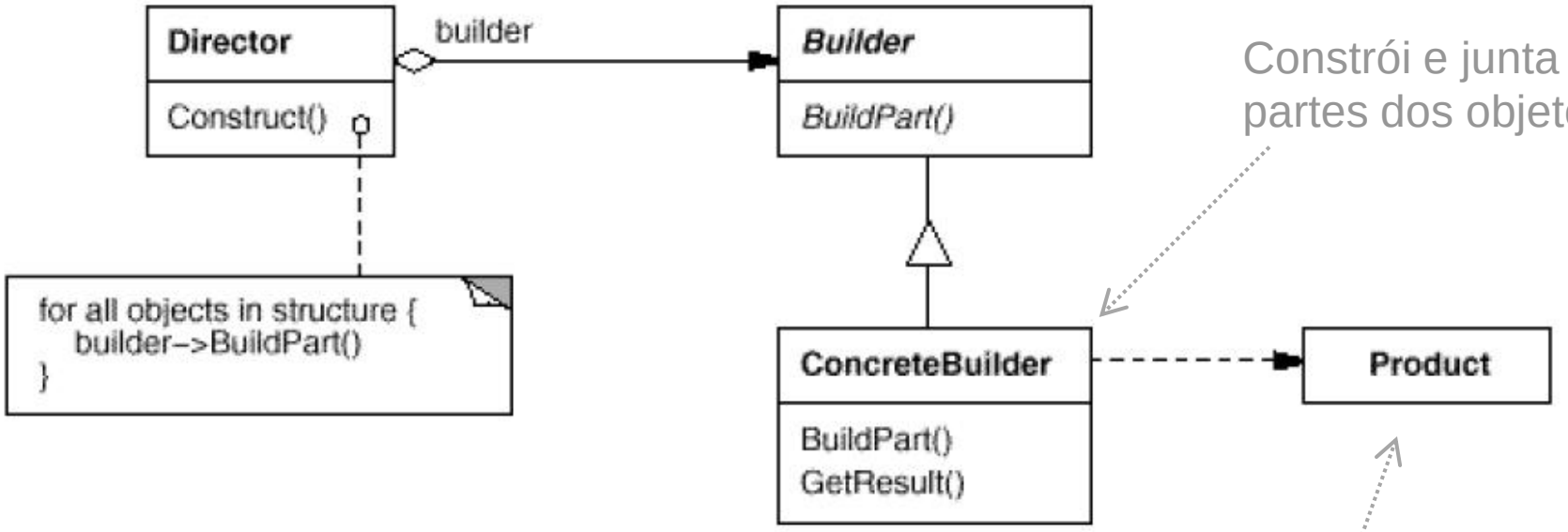
A decisão I pode ser atendida usando-se o padrão de projeto Abstract Factory;

Builder

- Separa a construção de um objeto complexo da sua representação, de forma que o mesmo processo possa criar diferentes tipos de representações
- Use Builder quando:
  - O algoritmo para criar um objeto deve ser independente de suas partes e de como elas são montadas
- Dica: enquanto Abstract Factory enfatiza famílias de objetos, Builder constrói partes de objetos passo a passo

Builder

Coordena a sequência de construção dos objetos



Constrói e junta as partes dos objetos

Partes a serem construídas

Builder

Produto a ser construído

Partes do produto

```
class Pizza {
        private String tempero = "";
        private String  cobertura = "";

        public void setTempero(String tempero) {
                this.tempero = tempero;
        }
        public void setCobertura(String cobertura) {
                this.cobertura = cobertura;
        }
}
```

Classe construtora abstrata (genérica)

Métodos para construir cada parte

```
abstract class PizzaBuilder {
        protected Pizza pizza;

        public Pizza getPizza() {
                return pizza;
        }

        public void criarNovoProdutoPizza() {
                pizza = new Pizza();
        }

        public abstract void buildTempero();
        public abstract void buildCobertura();
}
```

Builder

Classe construtora concreta (específica).
Constrói as partes do produto

```
class BuilderPizzaMarguerita extends PizzaBuilder {
        public void buildTempero() {
                pizza.setTempero("quente");
        }

        public void buildCobertura() {
                pizza.setCobertura("tomate");
        }
}
```

```
class BuilderPizzaCalabresa extends PizzaBuilder {
        public void buildTempero() {
                pizza.setTempero("medio");
        }

        public void buildCobertura() {
                pizza.setCobertura("calabresa");
```

```
class Cozinhar {
        private PizzaBuilder pizzaBuilder;

        public void setPizzaBuilder(PizzaBuilder pb) {
                pizzaBuilder = pb;
        }

        public Pizza getPizza() {
                return pizzaBuilder.getPizza();
        }

        public void constructPizza() {
                pizzaBuilder.criarNovoProdutoPizza();
                pizzaBuilder.buildTempero();
                pizzaBuilder.buildCobertura();
        }
}
```

Classe diretora.
Coordena a ordem de construção das partes

Builder (executando)

Seta o builder na classe diretora

Constrói as partes

Retorna o objeto construído

```
public class ExemploBuilder {
        public static void main(String[] args) {
                Cozinhar c = new Cozinhar();

                PizzaBuilder builderPizzarMarguerita = new BuilderPizzaMarguerita();
                PizzaBuilder builderPizzarCalabresa = new BuilderPizzaCalabresa();

                c.setPizzaBuilder(builderPizzaMarguerita);

                c.construirPizza();

                Pizza marguerita = c.getPizza();

                c.setPizzaBuilder(builderPizzaCalabresa);

                c.construirPizza();

                Pizza calabresa = c.getPizza();
        }
}
```

Exercícios [3]

(IRB – ESAF 2006)

[58-I] A intenção do Padrão de Projeto Builder, também conhecido como Command, é adaptar a interface de uma ou mais classes para permitir que classes com interfaces incompatíveis possam interagir.

(BACEN – CESGRANRIO 2010)

[33-B] Builder garante que uma classe seja instanciada somente uma vez, fornecendo também um ponto de acesso global.

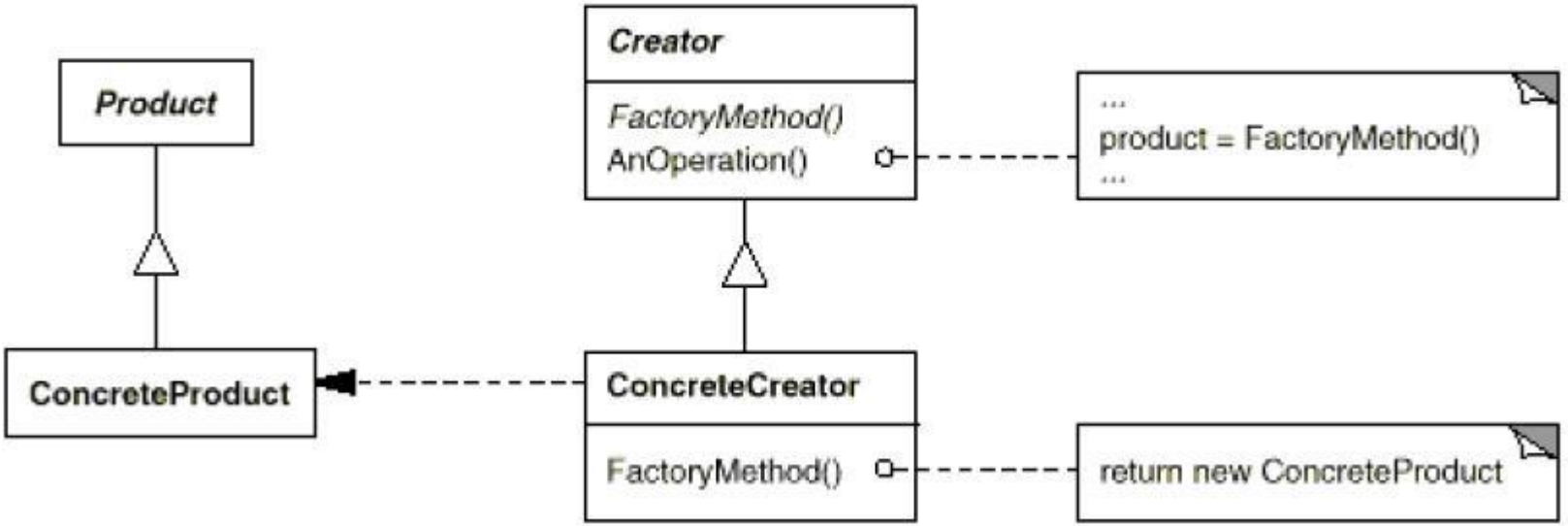
(SERPRO – CESPE 2010)

[88] No padrão builder, a responsabilidade pela criação de instâncias é compartilhada por um diretor e um construtor, sendo o vínculo entre eles estabelecido pelo cliente do padrão.

Factory Method

- Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar
- Use Factory Method quando
  - Uma classe não pode antecipar a classe de objetos que ela deve criar
  - Uma classe quer que suas subclasses especifiquem os objetos que ela cria

Factory Method



Factory Method

Produto abstrato

```
public abstract class Pessoa {
    public String nome;
    private String sexo;

    public String getSexo() {
        return this.sexo;
    }
}
```

Produto concreto

```
public class Homem extends Pessoa {
    public Homem(String nomeCompleto){
        System.out.println("Ola, Senhor " + nomeCompleto);
    }
}
```

Produto concreto

```
public class Mulher extends Pessoa {
    public Mulher(String nomeCompleto){
        System.out.println("Ola, Senhora " + nomeCompleto);
    }
}
```

Factory

```
public class FactoryPessoa {
    public Pessoa getPessoa(String nome, String sexo) {
        if (sexo.equals("M"))
            return new Homem (nome);
        if (sexo.equals("F"))
            return new Mulher(nome);
        else
            return null;
    }
}
```

Factory Method (executando)

```
public class Aplicacao

    public static void main(String args[]) {

        FactoryPessoa factory = new FactoryPessoa();

        String nome = args[0];
        String sexo = args[1];
        factory.getPessoa(nome, sexo);
}
```

Que saudação devemos usar, “Senhor” ou “Senhora”?

Em vez de usar vários “if’s”, deixamos para a **Fábrica** decidir!

Exercícios [4]

(INFRAERO – FCC 2009)

[53] NÃO é um elemento contido no padrão de projeto Factory Method

(A) Product.

(B) ConcreteProduct.

(C) Director.

(D) Creator.

(E) ConcreteCreator.

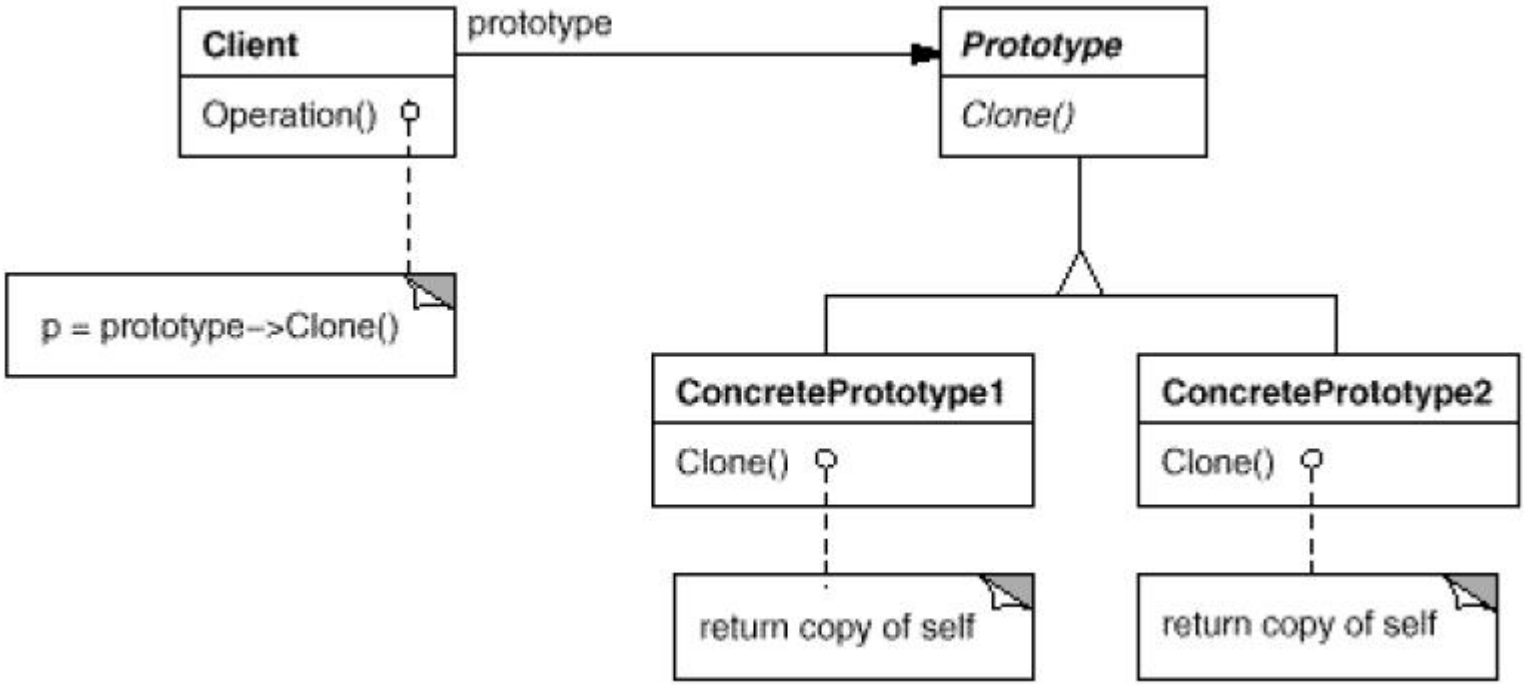
(TRE/MS – FCC 2007)

[50-C] Método Fábrica provê a habilidade de retornar o estado do objeto a seu estado anterior.

Prototype

- Especifica os tipos de objetos para criar usando uma instância prototípica, e cria novos objetos copiando este protótipo (clonando o objeto original)
- Use Prototype quando:
  - O sistema possui componentes cujo estado inicial tem poucas variações, e é conveniente disponibilizar um conjunto pré estabelecido de protótipos que dão origem aos objetos que compõem o sistema

Prototype


Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

Prototype

```
abstract class Documento implements Cloneable {
        protected Documento clone() {
                Object clone = null;
                try {
                        clone = super.clone();
                } catch (CloneNotSupportedException ex) {
                        ex.printStackTrace();
                }
                return (Documento) clone;
        }
}

class ASCII extends Documento { }

class PDF extends Documento { }
```

Protótipo abstrato

Protótipos concretos

```
class Cliente {

        static final int DOCUMENTO_TIPO_ASCII = 0;
        static final int DOCUMENTO_TIPO_PDF = 1;

        private Documento ascii = new ASCII();
        private Documento pdf = new PDF();

        public Documento criarDocumento(int tipo) {
                if (tipo==Cliente.DOCUMENTO_TIPO_ASCII) {
                        return ascii.clone();
                } else {
                        return pdf.clone();
                }
        }
}
```

Classe cliente

Baseado no tipo passado como parâmetro, são retornados clones dos objetos originais

Exercícios [5]

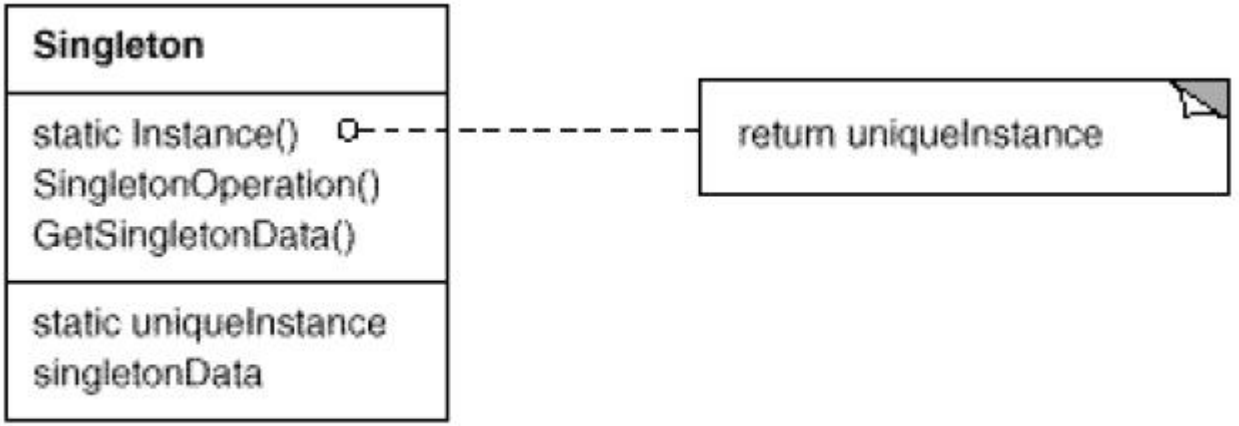
(IRB - ESAF 2006)

[58-III] A intenção do Padrão de Projeto Prototype é permitir a criação de famílias de objetos relacionados ou dependentes através de uma única interface e sem que a classe concreta seja especificada. Por exemplo, cria-se uma classe abstrata que declara uma interface genérica para criação dos controles visuais e uma classe abstrata para criação de cada tipo de controle. Em cada um dos padrões tecnológicos contemplados existirá uma classe concreta que deverá conter a implementação relativa a cada controle.

Singleton

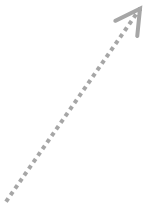
- Garante que uma classe tem apenas uma instância e provê um ponto de acesso global a ela
- Use Singleton quando:
  - Deve haver exatamente uma instância de uma classe, e ela deve ser acessível aos clientes a partir de um ponto de acesso conhecido

Singleton


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

Singleton

```
public class Singleton {
  private static final Singleton INSTANCE = new Singleton();

  // O construtor privado impede que outras classes o acessem diretamente
  private Singleton() {}

  public static Singleton getInstance() {
     return INSTANCE;
  }
}
```



Este é o único modo de acessar a instância singular da classe

Exercícios [6]

(IRB - ESAF 2006)

[58-II] A intenção do Padrão de Projeto Singleton é garantir que exista apenas uma instância de sua classe.

(BNDES - CESGRANRIO 2009)

[53] Por motivo de segurança, deseja-se adicionar registro (log) das operações efetuadas no sistema de contabilidade de uma empresa. O arquiteto do sistema decide que deve existir somente uma instância de uma classe de registro (log) e que esta será o ponto de acesso global para os demais componentes do sistema. Que padrão de projeto pode ser utilizado nesse caso?

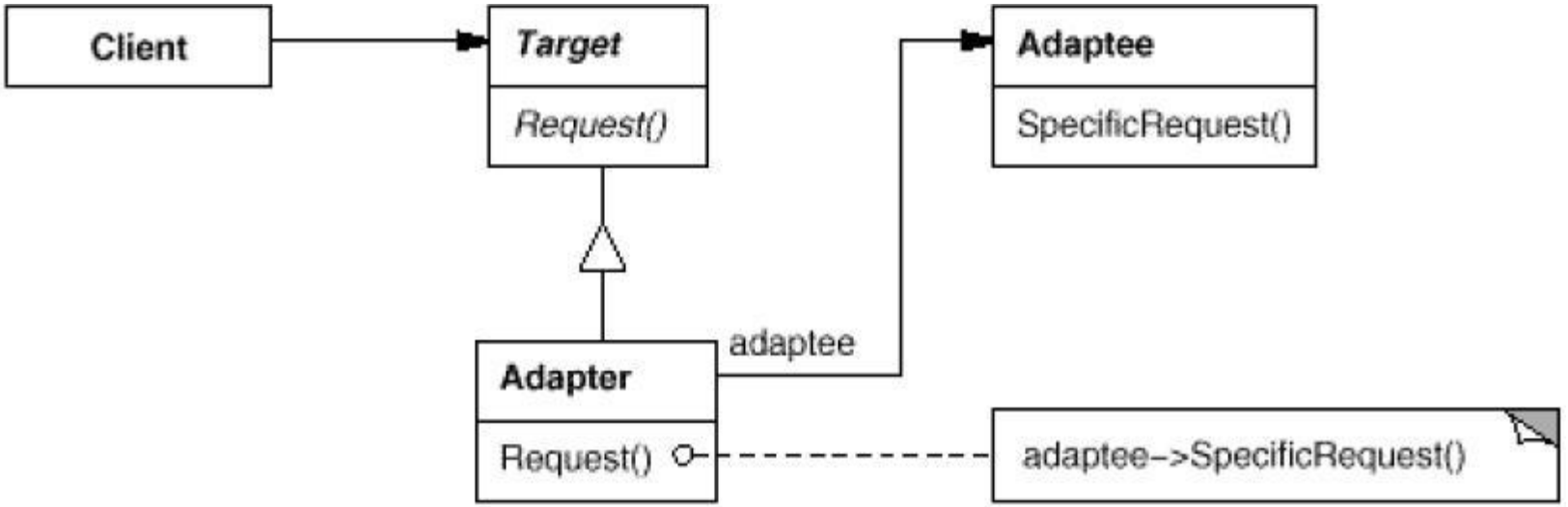
(A) Iterator (B) Visitor (C) Bridge (D) Memento (E) Singleton

Padrões Estruturais

Adapter

- Converte a interface de uma classe em outra interface que normalmente não poderiam trabalhar juntas
- Use o Adapter quando:
  - Você quer usar uma classe existente, e sua interface não é adequada àquela que você precisa

Adapter


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()

Adapter

```
public class PlugDoisPinos {
        public void ligarDoisPinos(Tomada t) {
        System.out.println("Dois pinos");
    }
}

public class PlugTresPinos {
        public void ligarTresPinos(Tomada t) {
        System.out.println("Tres pinos");
    }
}

public class AdapterTomada extends PlugDoisPinos {
    private PlugTresPinos plugTresPinos;

    public AdapterTomada(PlugTresPinos p) {
        this.plugTresPinos = p;
    }

    public void ligarDoisPinos(Tomada t) {
        plugTresPinos.ligarTresPinos(t);
    }
}
```

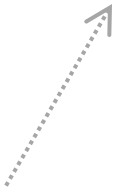
Classe alvo (target): é o que o cliente possui

Classe adaptada (adaptee): é o que o cliente necessita

Adaptador

Adapter (executando)

```
public class Aplicacao {

    public static void main(String args[]) {
        PlugTresPinos p3 = new PlugTresPinos();

        AdapterTomada a = new AdapterTomada(p3);
        a.ligarDoisPinos(new Tomada());
    }
}
```



O cliente faz a chamada usando o plug de dois pinos, que é o que ele enxerga, mas na verdade esta chamada está sendo “adaptada” para um plug de três pinos

Exercícios [7]

(DATAPREV – CESPE 2006)

[68] As seguintes situações justificam o uso do padrão Adapter: é necessário um objeto local que se faça passar por um objeto localizado em outro espaço de endereçamento; é necessário controlar o acesso a um objeto; um objeto persistente deve ser carregado em memória somente quando for referenciado.

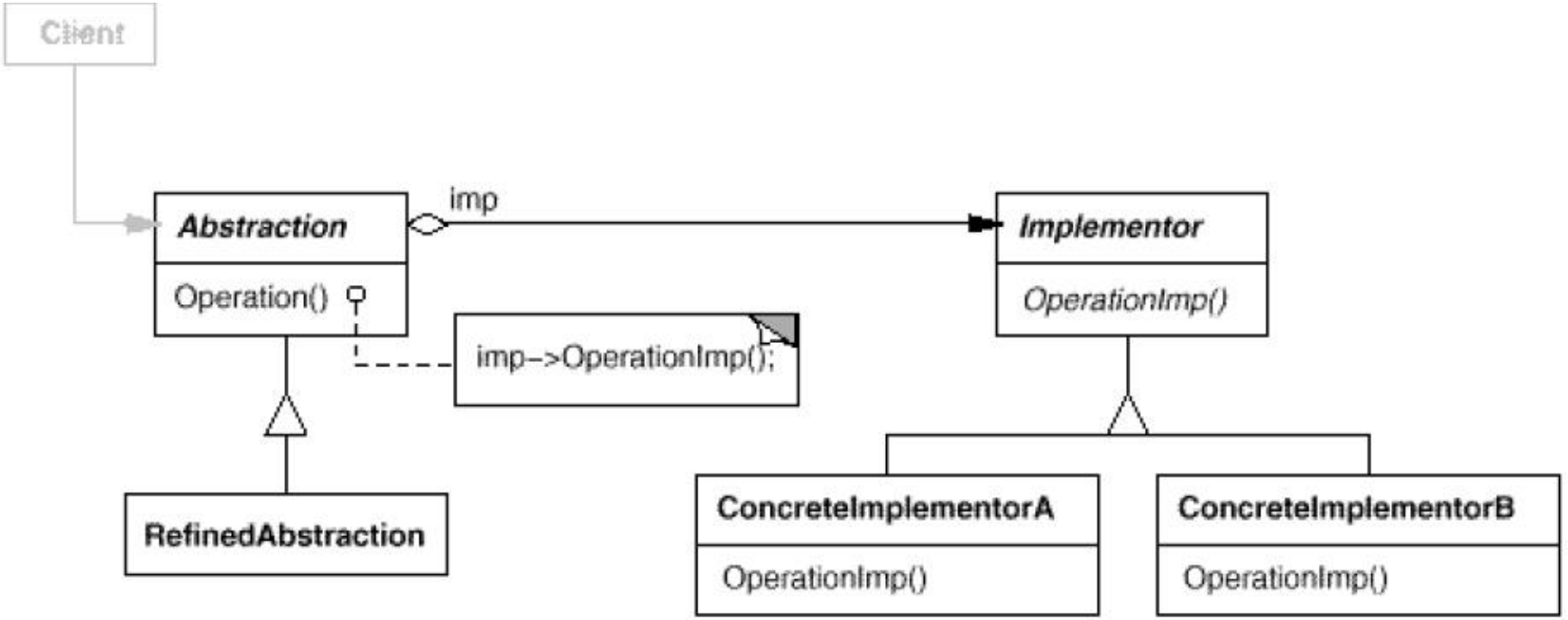
(SERPRO – CESPE 2008)

[115] Adapter é um padrão estrutural utilizado para compatibilizar interfaces de modo que elas possam interagir.

Bridge

- Desacopla uma interface de sua implementação, de forma que elas possa variar independentemente
- Use o Bridge quando:
  - Você quer evitar um vínculo entre a abstração e a implementação
  - Mudanças na implementação de uma abstração não deveriam ter impacto nos clientes, isto é, seu código não deveria ser recompilado

Bridge


Client
Abstraction
Operation()
imp
imp->OperationImp();
Implementor
OperationImp()
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

Bridge

```
/** "Implementor" */
interface APIDeDesenho {
    public void desenharLinha(int x, int y);
}

/** "ConcreteImplementor" 1/2 */
class APIDeDesenho1 implements APIDeDesenho {
   public void desenharLinha(int x, int y) {
        System.out.println("Linha desenhada, do ponto x ao ponto y");
   }
}

/** "ConcreteImplementor" 2/2 */
class APIDeDesenho2 implements APIDeDesenho {
   public void desenharLinha(int x, int y) {
       System.out.println("Linha desenhada, do ponto x ao ponto y," +
                          " mas um pouco diferente");
   }
}
```

Classes que implementam a API de desenho.
A implementação pode variar livremente

Bridge

```
/** "Abstraction" */
interface Forma {
   public void desenharLinha();
}

/** "Refined Abstraction" */
class Linha implements Forma {
   private APIDeDesenho api;
   public Linha(int x, int y, APIDeDesenho api) {
        this.x = x;  this.y = y;
        this.api = api;
   }
   public void desenharLinha() {
        api.desenharLinha(x, y);
   }
}
```

Note como a abstração de Implementação é passada para cá

Essas são as classes que o cliente enxerga. Ele quer usar suas funcionalidades, mas a implementação pode variar, como vimos no slide passado.

Bridge (executando)

```
/** "Client" */
class Aplicacao {
    public static void main(String[] args) {
        Forma[] formas = new Shape[2];
        formas[0] = new Linha(1, 2, new APIDeDesenho1());
        formas[1] = new Linha(5, 7, new APIDeDesenho2());

        for (Forma forma : formas) {
            forma.desenharLinha();
        }
    }
}
```

É possível variar a implementação da abstração, sem impacto no cliente

Exercícios [8]

(IRB - ESAF 2006)

[58-IV] A intenção do Padrão de Projeto Bridge é garantir, quando desejável, que uma interface possa variar independentemente das suas implementações, como por exemplo, na implementação de um sistema gráfico de janelas.

(BACEN - CESGRANRIO 2010)

[33-A] Bridge separa a construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

Exercícios [9]

(BNDES – CESGRANRIO 2009)

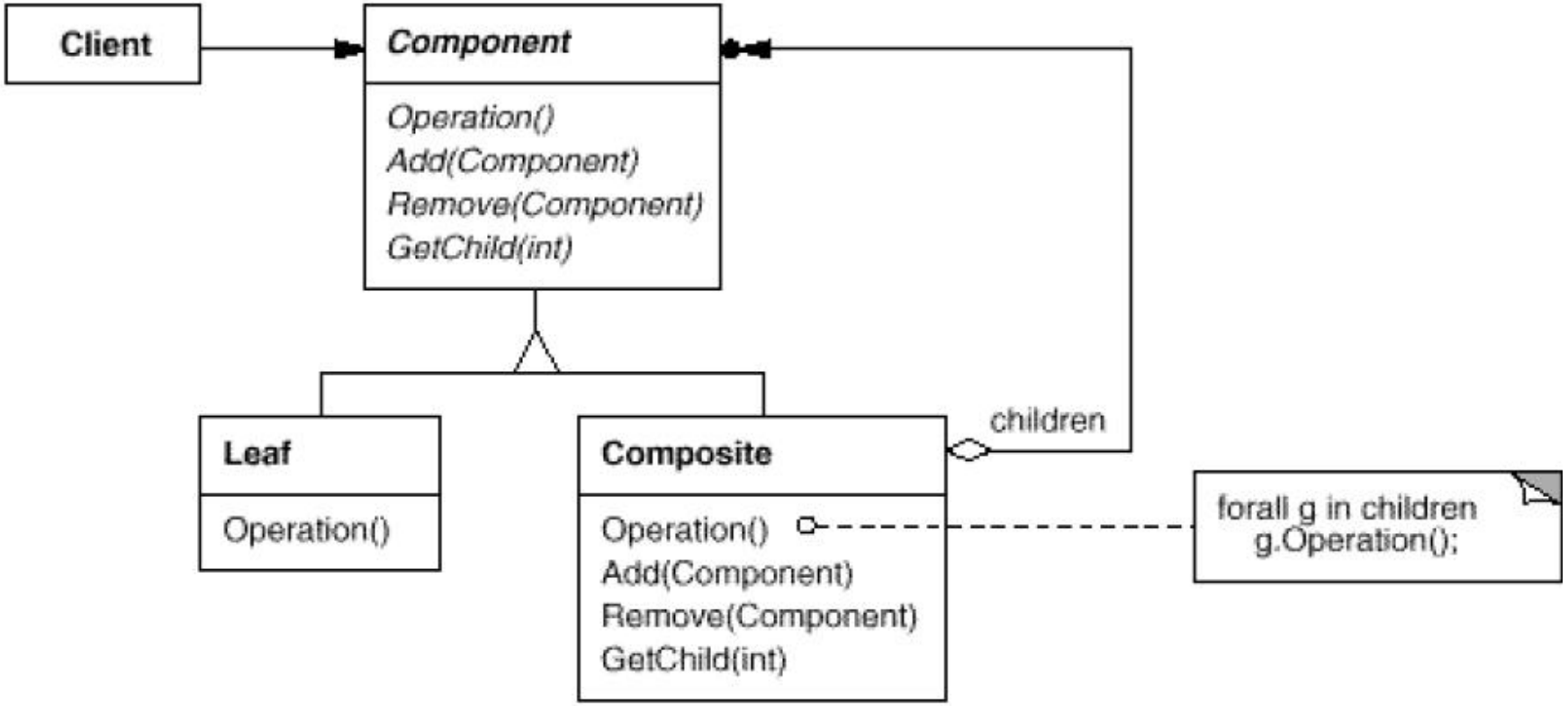
[60] Ao consultar informações a respeito dos padrões de projeto Adapter e Bridge, um Analista de Sistemas identificou uma afirmativa INCORRETA. Assinale-a.

(A) Ambos promovem a flexibilidade ao fornecer um nível de endereçamento indireto para outro objeto.
(B) Ambos são padrões estruturais que possuem alguns atributos em comum.
(C) O foco do Adapter é a solução de incompatibilidades entre duas interfaces existentes.
(D) O Adapter é inferior ao Bridge porque não evita a replicação de código.
(E) O Bridge estabelece uma ponte entre uma abstração e suas possíveis implementações.

Composite

- Compõe zero ou mais objetos similares de forma que eles possam ser manipulados como um só
- Use Composite quando:
  - Você quer representar hierarquias parte-todo de objetos
  - Você quer que o cliente ignore a diferença entre objetos compostos e objetos individuais

Composite


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

Composite

```
//Componente
class Component {
    public void print();
}

//Composite
class Composite extends Component {

    private List<Component> childComponents
                = new ArrayList<Component>();

    public void print() {
        for(Component c : childComponents) {
            c.print();
        }
    }
    public void add(Component c) {
        childComponents.add(c);
    }
    public void remove(Component c) {
        childComponents.remove(c);
    }
}
```

Classe composta

Classe folha

```
//Leaf
class Leaf extends Component {
    public void print() {
        system.out.println("Folha")
    }
}
```

Composite (executando)

```
public class CompositeDemo {

    public static void main(String args[]) {

        Leaf folha1 = new Leaf();
        Leaf folha2 - new Leaf();

        Composite c = new Composite)();
        Composite c2 = new Composite)();
        Composite c3 = new Composite)();

        c.add(folha1);
        c.add(folha2);

        c2.add(folha2);

        c.add(c2);
        c.add(c3);

        c.print();
    }
}
```

Note que, para o cliente, tanto faz manipular uma folha ou uma composição de objetos

Exercícios [10]

(TRE/MS – FCC 2007)

[50-D] Composite realiza a adaptação da interface de uma determinada classe para a interface que um cliente espera.

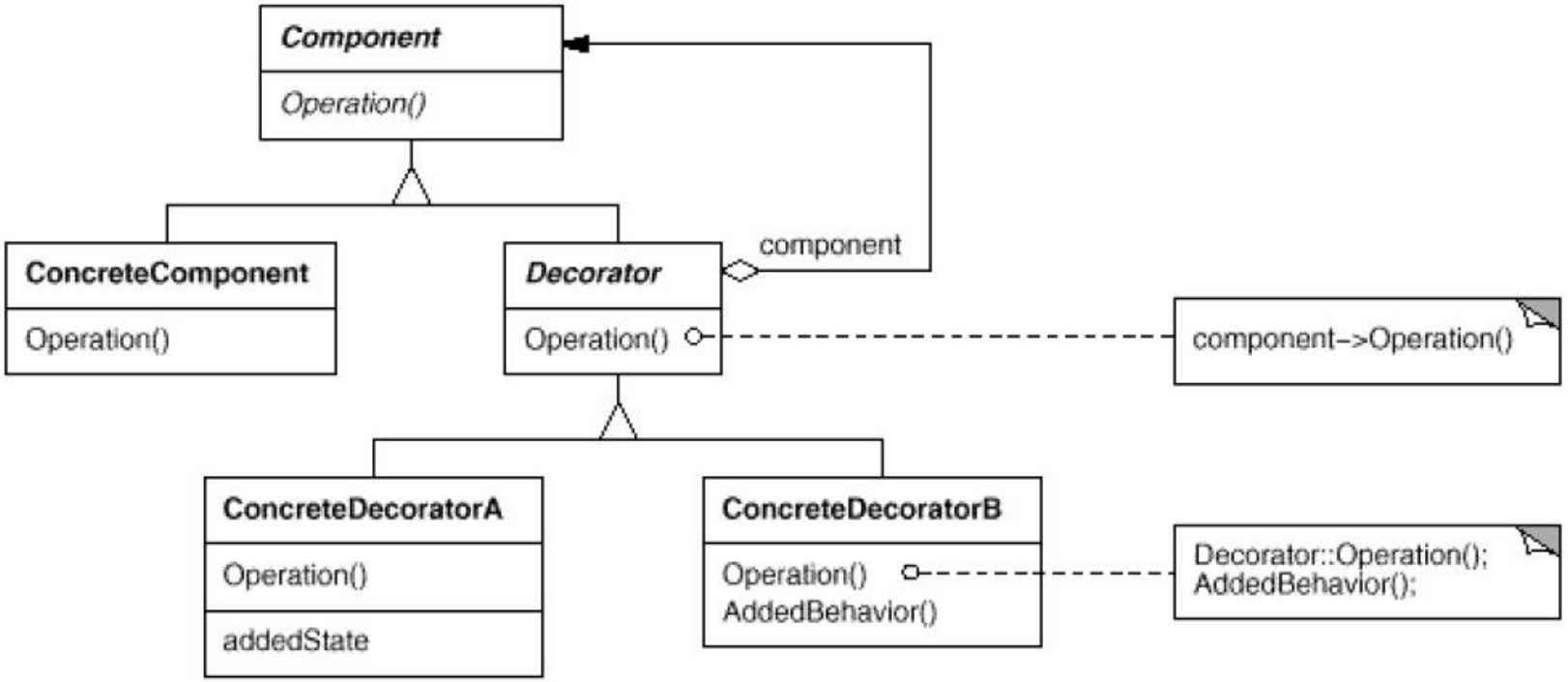
(TRE/AP – CESPE 2007)

[11-II] A implementação de montadores de árvores sintáticas apóia-se mais no uso do padrão Singleton que no uso do padrão Composite.

Decorator

- Anexa responsabilidades adicionais a um objeto dinamicamente
- Decoradores fornecem uma alternativa flexível em relação a herança para estender funcionalidades
- Use o Decorator quando:
  - Quiser adicionar responsabilidades a objetos dinamicamente
  - Quando a extensão por subclasses é impraticável

Decorator


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

Decorator

```
abstract class Janela {
    public abstract void draw();
}

class JanelaSimples extends Janela {
    public void draw() {
        ....
    }
}

abstract class JanelaDecorator extends Janela {
    protected Janela janelaDecorada;

    public JanelaDecorator (Janela janelaDecorada) {
        this.janelaDecorada = janelaDecorada;
    }
}
```

```
class DecoradorBarraVertical extends JanelaDecorator {
    public DecoradorBarraVertical (Janela janelaDecorada) {
        super(janelaDecorada);
    }
    public void draw() {
        drawBarraVertical();
        janelaDecorada.draw();
    }
    private void drawBarraVertical() { ... }
}
```

O decorator adiciona novos comportamentos

Decorator (executando)

```
public class DecoratorDemo {
    public static void main(String args[]) {
        Janela janelaDecorada = new DecoradorBarraVertical(new JanelaSimples());
        janelaDecorada.pintar();
    }
}
```

Este método pintar() combina o comportamento base mais o comportamento “decorado”

Exercícios [11]

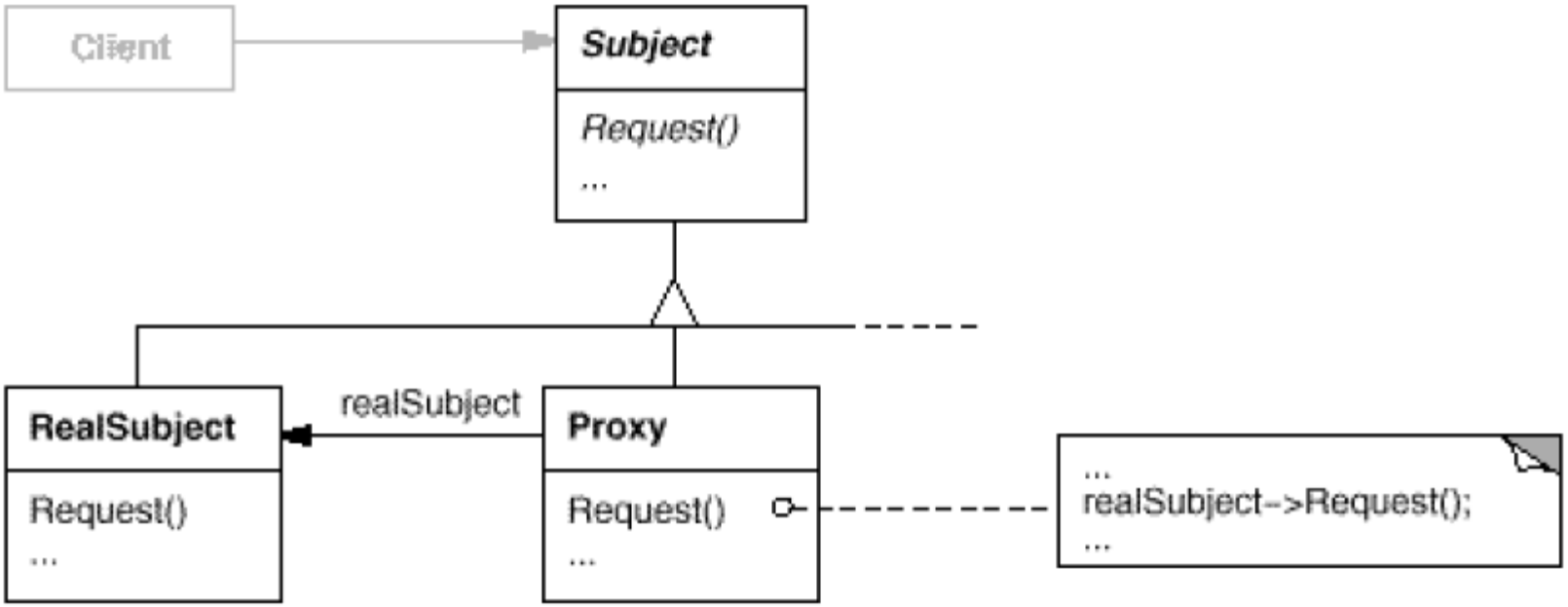
(INMETRO – CESPE 2009)

[88-C] Caso se adote o padrão Decorator para adicionar responsabilidades a um conjunto de instâncias que possuem uma superclasse comum denominada X, então, quando um objeto da classe X for decorado por uma instância de uma classe qualquer Y, os métodos presentes na classe X não estarão presentes na interface de Y.

Proxy

- Provê um substituto ou ponto através do qual um objeto possa controlar o acesso a outro
- Use Proxy quando:
  - Toda vez que há uma necessidade de uma referência mais versátil ou sofisticada do que um simples ponteiro para um objeto

Proxy


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

Proxy

Objeto real

```
interface Pessoa {
	public String getNome();
}

//Objeto real
class PessoaImpl implements Pessoa {
	private String nome;

	public PessoaImpl(String nome) {
		this.nome = nome;
	}
	public String getNome() {
		return nome;
	}
}
```

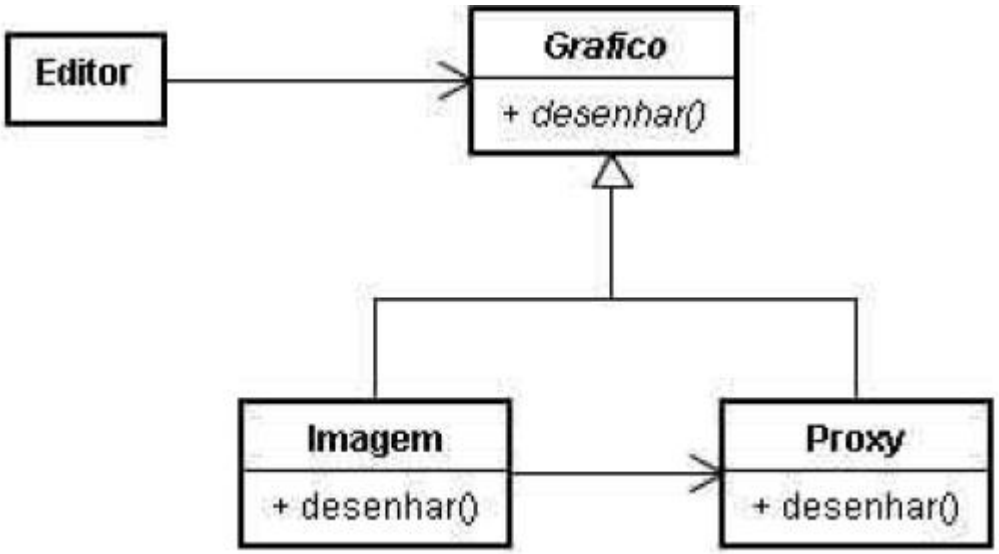
Proxy

```
class ProxyPessoa implements Pessoa {

	private String nome
	private Pessoa pessoa;//mesma interface

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

	public String getNome() {
		if (pessoa == null) {
			//Apenas cria o objeto real quando chamar este método
			pessoa = PessoaDAO.getPessoaByNome(this.nome);
		}
		/** Delega para o objeto real **/
		return pessoa.getNome();
	}
}
```

Exercícios [12]

(Min. Comunicações - CESPE 2008)

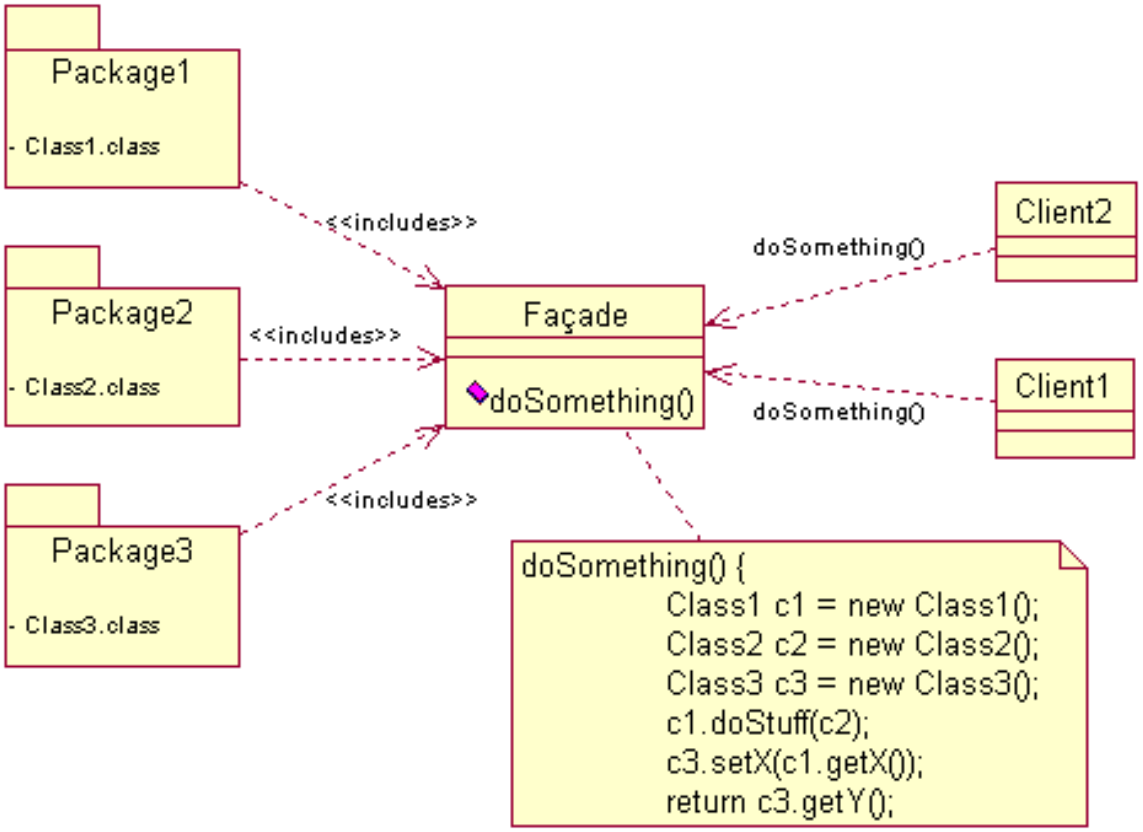
[99] O padrão proxy está corretamente documentado no seguinte diagrama UML.



Façade

- Provê uma interface unificada para um conjunto de interfaces de um subsistema
- Define uma interface de mais alto nível que torna o subsistema mais fácil de manipular
- Use o Façade quando
  - Você quiser prover uma interface simples para um subsistema complexo

Façade


Package1
- Class1.class
<<includes>>
Package2
- Class2.class
<<includes>>
Package3
- Class3.class
<<includes>>
Façade
doSomething()
Client2
doSomething()
Client1
doSomething()
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2);
c3.setX(c1.getX());
return c3.getY();

Façade

Partes complexas, com várias interfaces

```
/* Complex parts */
class CPU {
        public void freeze() { ... }
        public void jump(long position) { ... }
        public void execute() { ... }
}

class Memory {
        public void load(long position, byte[] data) {
                ...
        }
}
class HardDrive {
        public byte[] read(long lba, int size) {
                ...
        }
}
```

```
/* Facade */

class Computer {
        private CPU cpu=null;
        private Memory memory=null;
        private HardDrive hardDrive=null;

        public Computer() {
                this.cpu=new CPU();
                this.memory=new Memory();
                this.hardDrive=new HardDrive();
        }

        public void startComputer() {
                cpu.freeze();
                memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
                cpu.jump(BOOT_ADDRESS);
                cpu.execute();
        }
}
```

Interface unificada na Façade

Exercícios [13]

(BNDES - CESGRANRIO 2009)

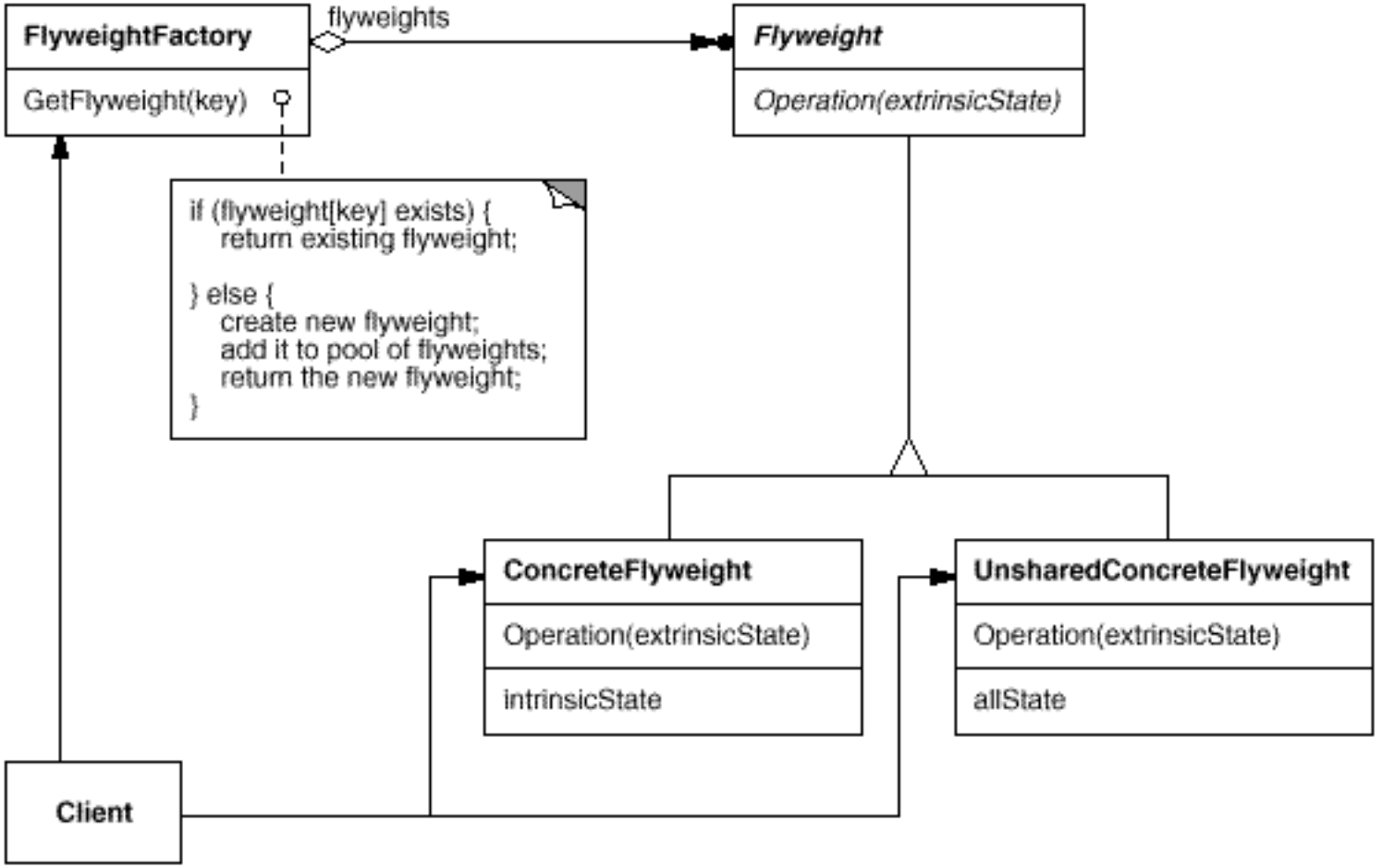
[59] O presidente de uma empresa determinou que fosse disponibilizado um sistema de vendas na Internet. No entanto, o software de controle de estoque que deve ser acessado pela aplicação de vendas é muito antigo e provê uma API (Application Programming Interface) de uso muito complicado. Para que os desenvolvedores possam acessar uma interface mais simples, o arquiteto do sistema pode determinar o uso do padrão de projeto

(A) Prototype.
(B) Decorator.
(C) Observer.
(D) Façade.
(E) Flyweight.

Flyweight

- Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente
- Use o Flyweight quando
  - Uma aplicação utiliza um grande número de objetos e o custo para armazená-los é muito alto
  - A maioria dos estados dos objetos pode ser tornada extrínseca

Flyweight


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Flyweight

```
class Caractere {
   //podem existir MILHARES deste objeto Flyweight
   private char caractere;
   public Caractere (char c) {
       this.caractere = c;
   }
   public void desenharNaTela(Contexto c) {
       /* Lógica para desenhar o caractere na tela,
       DE ACORDO COM O CONTEXTO FORNECIDO
       PELO CLIENTE...
       */
   }
}

class Contexto {
    //o CLIENTE fornece o contexto
    private int linha;
    private int coluna;
    Contexto (int linha, int coluna) {
        this.linha = linha;
        this.coluna = coluna;
    }
}
```

Objeto Flyweight

Factory de Flyweights

```
class Factory {
    //gerencia o pool, a "cache" de Flyweights
    private Caractere[] pool;
    public Factory (int max) {
        pool = new Caractere[max];
    }

    public getCaractere(char c) {
        if (pool[c] == null) {
            pool[c] = new Caractere(c);
        }
        return pool[c];
    }
}
```

Flyweight (executando)

```
public class FlyweightDemo {
    public static void main( String[] args ) {
      Factory f = new Factory (Character.MAX_VALUE);

      //cria o primeiro e o segundo Flyweights
      f.getCaractere('p').desenharNaTela(new Contexto(1,1));
      f.getCaractere('a').desenharNaTela(new Contexto(1,2));

      //cria o terceiro Flyweight
      f.getCaractere('s').desenharNaTela(new Contexto(1,3));

      //não cria nada... utiliza um Flyweight já existente
      f.getCaractere('s').desenharNaTela(new Contexto(1,4));


      //cria os últimos Flyweights
      f.getCaractere('e').desenharNaTela(new Contexto(1,5));
      f.getCaractere('i').desenharNaTela(new Contexto(1,6));
    }
}
```

A fábrica retorna os Flyweights para o cliente, que os utiliza passando uma configuração de contexto.

Exercícios [14]

(MPE/BA – FESMIP/BA 2011)

[51] O Design Pattern que tem a finalidade de usar compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente, é denominado

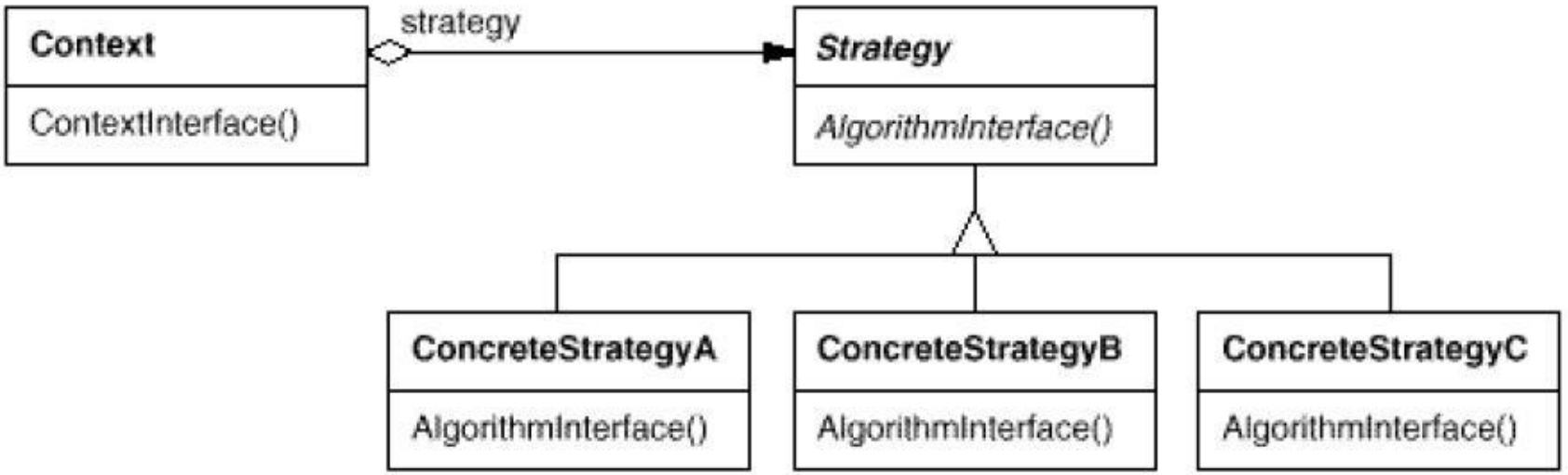
a) Strategy
b) Composite
c) Flyweight
d) State
e) Builder

Padrões Comportamentais

Strategy

- Define uma família de algoritmos, encapsula cada um, e faz deles intercambiáveis
- Use Strategy quando:
  - Várias classes relacionadas diferem apenas em seus comportamentos
  - Você precisa de diferentes variantes de um algoritmo
  - Uma classe define muitos comportamentos e eles aparecem como declarações condicionais nas suas operações

Strategy


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Strategy

```
interface Strategy {
    int execute(int a, int b);
}


class ConcreteStrategyAdd implements Strategy {
    public int execute(int a, int b) {
        return a + b;
    }
}

class ConcreteStrategySubtract implements Strategy {
    public int execute(int a, int b) {
        return a - b;
    }
}
```

Estratégias diferentes

```
class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

Classe de contexto

Strategy (executando)

```
class StrategyExample {
    public static void main(String[] args) {
        Context context;

        context = new Context(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);
    }
}
```

Diferentes variações de algoritmos, apenas configurando a estratégia, sem a necessidade de estruturas de seleção

Exercícios [15]

(TRE/MS – FCC 2007)

[50-B] Strategy permite a criação de uma família de algoritmos encapsulados na forma de objetos que podem ser selecionados e substituídos dinamicamente pela aplicação.

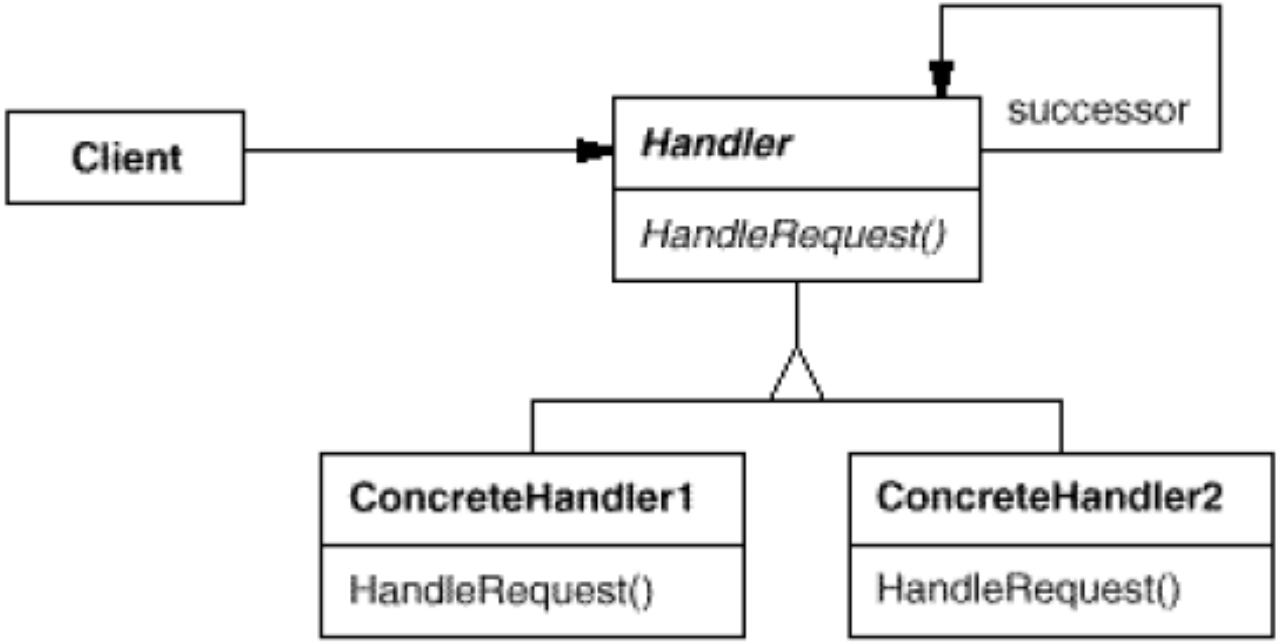
(INMETRO – CESPE 2009)

[88-D] Caso se deseje incorporar a um software um conjunto de algoritmos de uma mesma família, os quais são aplicáveis de forma intercambiável a um agregado de objetos similares, no qual o conjunto é passível de expansão em tempo de manutenção do software, então é mais recomendada a adoção do padrão Composite.

Chain of Responsibility

- Evita o acoplamento do remetente de uma solicitação ao seu receptor
- Encadeia os objetos receptores, passando a solicitação ao longo da cadeira até que um objeto a trate
- Use o Chain of Responsibility quando:
  - Você quer emitir uma solicitação para um dentre vários objetos, sem especificar explicitamente o receptor

Chain of Responsibility



Chain of Responsibility

Handler genérico

```
public interface HelpHandler {
    //Handler genérico
    public void handleHelp();
}
```

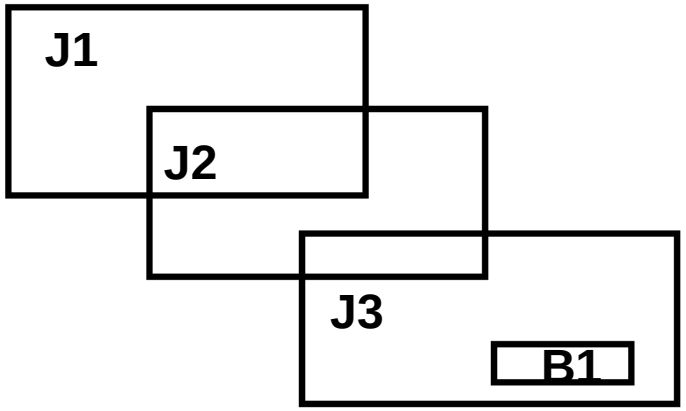
```
public class Janela implements HelpHandler {
    //Handler concreto
    private HelpHandler helpSuccessor;
    public Janela (HelpHandler helpSuccessor) {
       this.helpSuccessor = helpSuccessor;
     }

     public void handleHelp() {
        if (temHelp) {
            //codigo de tratamento do help...
        } else {
            helpSuccessor.handleHelp();
        }
    }
}
```

```
public class Botao {
    //Handler concreto
    private HelpHandler helpSuccessor;
    public Botao (Helphandle helpSuccessor) {
       this.helpSuccessor = helpSuccessor
    }

    public void handleHelp() {
        if (temHelp) {
            //codigo de tratamento do help...
        } else {
            helpSuccessor.handleHelp();
        }
    }
}
```

Handlers concretos

Chain of Responsibility (executando)

```
public ChainExemplo {

    public static void main(String args[]) {
        Janela j1, j2, j3;
        Botao b1;

        b1 = new Botao (j3);
        j3 = new Janela(j2);
        j2 = new Janela (j1);

        //o usuário clica na ajuda do botão
        b1.handleHelp();

        //se b1 tiver aquele Help, ele mesmo trata.

        //se não, passa para j3 tratar, que pode passar para j2, que pode passar para j1,
        //e assim por diante até um Handler "default" tratar a requisição
    }

}
```



Exercícios [16]

(BACEN – CESGRANRIO 2010)

[33] Um arquiteto de software estuda que padrões de projeto são apropriados para o novo sistema de vendas de uma empresa. Ele deve considerar que o padrão

a) Bridge separa a construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

b) Builder garante que uma classe seja instanciada somente uma vez, fornecendo também um ponto de acesso global.

c) Singleton separa uma abstração de sua implementação, de modo que os dois conceitos possam variar de modo independente.

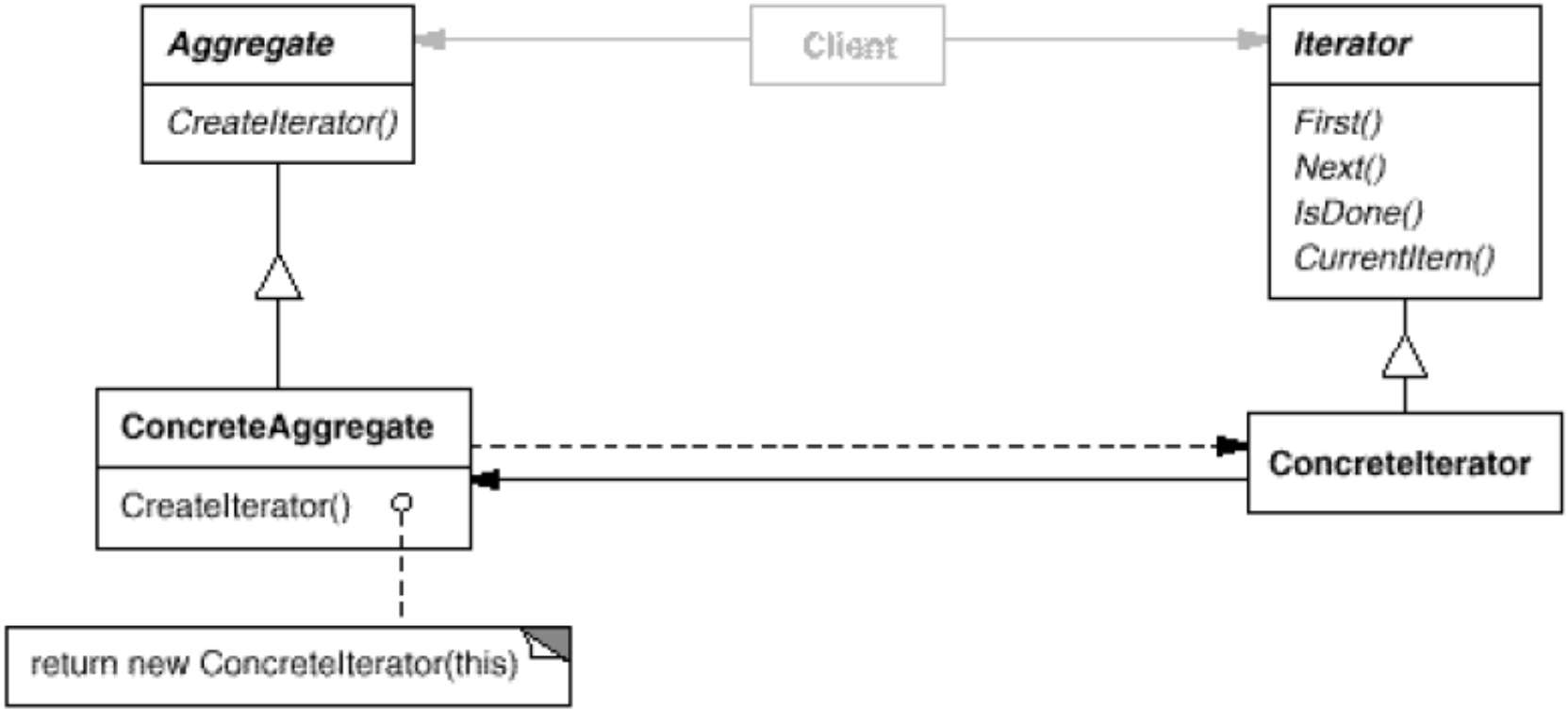
d) Chain of Responsibility evita o acoplamento entre o remetente de uma solicitação e seu destinatário, dando oportunidade para mais de um objeto tratar a solicitação.

e) Template Method utiliza compartilhamento para suportar, eficientemente, grandes quantidades de objetos de granularidade fina.

Iterator

- Fornece um meio de acessar sequencialmente os elementos de um objeto agregado sem expor a sua representação subjacente
- Use Iterator quando:
  - Você quer acessar o conteúdo de uma coleção sem expor a sua representação interna

Iterator


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Iterator (executando)

```
class IteratorDemo {
        public static void main(String args[]) {
                // cria um ArrayList
                ArrayList al = new ArrayList();
                // adiciona elementos à coleção
                al.add("C");
                al.add("A");
                al.add("E");
                al.add("B");
                al.add("D");
                al.add("F");
                // utiliza o Iterator para visualizar o conteúdo da coleção
                Iterator itr = al.iterator();
                while(itr.hasNext()) {
                        Object element = itr.next();
                        System.out.print(element + " ");
                }
}
```

hasNext() => IsDone()
next () => Next() seguido por CurrentItem()
Note que não há First(). First() é feito automaticamente quando o iterador é criado.

Exercícios [17]

(TJ/PI – FCC 2009)

[59]

I. É o responsável pela especificação dos tipos de objetos a serem criados usando uma "instância" prototípica e pela criação de novos objetos copiando este protótipo.

II. Define uma interface de nível mais alto que torna o subsistema mais fácil de usar e fornece uma interface única para um subsistema com diversas interfaces; compõe o grupo de padrões estruturais.

III. Integrante do grupo de padrões comportamentais, ele provê uma forma de acessar sequencialmente os elementos de um agregado de objetos, sem expor a representação interna desse agregado.

IV. As consequências do uso deste padrão é que o encapsulamento é mantido, já que objetos usam sua própria informação para cumprir responsabilidades; leva ao fraco acoplamento entre objetos e à alta coesão, uma vez que objetos fazem tudo que é relacionado à sua própria informação.

Exercícios [17]

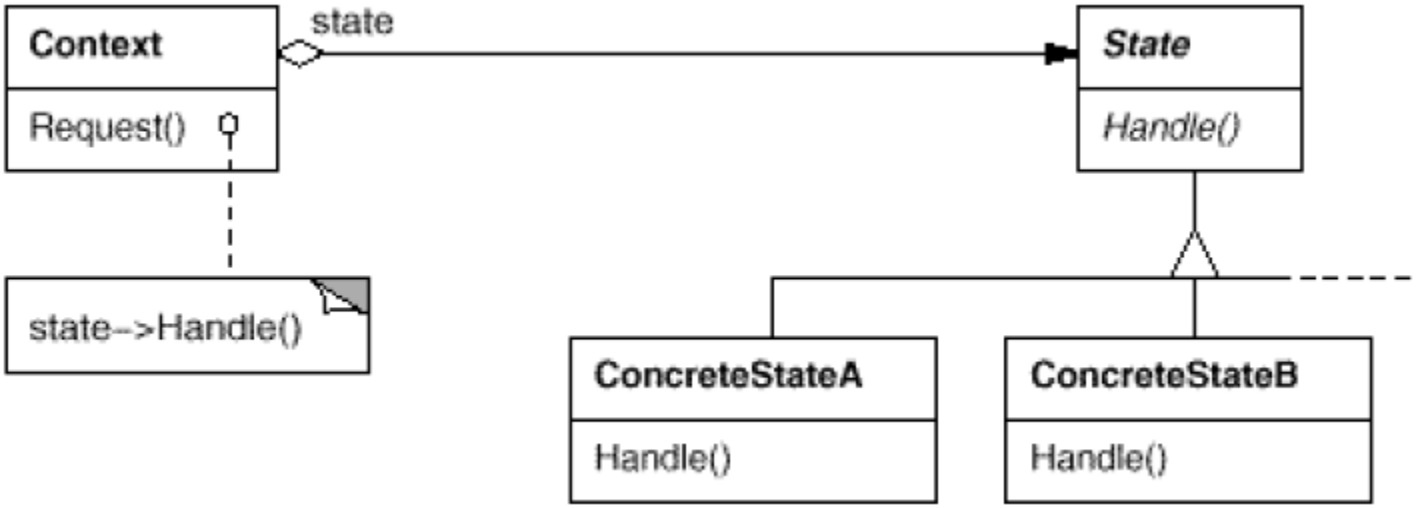
As afirmações correspondem, respectivamente, aos padrões

a) Command, Iterator, Singleton e Expert.
b) Controller, Expert, Singleton e Prototype.
c) Command, Singleton, Controller e Façade.
d) Prototype, Façade, Iterator e Expert.
e) Adapter, Façade, Command e Iterator.

State

- Permite que um objeto mude o seu comportamento quando o seu estado interno mudar
- O objeto parecerá ter mudado de classe
- Use State quando:
  - O comportamento de um objeto depende do seu estado, e ele deve mudar este comportamento em tempo de execução de acordo com este estado

State


Context
state
Request()
state->Handle()
State
Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

State

```
public class TCPConnection
//classe de contexto
/*vai mudar o comportamento de acordo
com o seu estado*/
{
    private TCPState state;

    public void setState( TCPState state ) {
            this.state = state;
    }
    public TCPState getState() {
            return this.state;
    }
    public void open() {
            state.open(this);
    }
    public void close() {
            state.close(this);
    }
}
```

```
public interface TCPState {
    //estado genérico
    void open(TCPConnection conn);
    void close(TCPConnection conn);
}

public class TCPEstablished implements TCPState {
    //estado concreto 01
    public void open(TCPConnection conn) {
        /*lógica para tratar uma requisição
        de OPEN quando uma conexão já
        foi estabelecida*/
    }
    public void close(TCPConnection conn) {
        /*lógica para tratar uma requisição
        de CLOSE quando uma conexão já foi
       estabelecida*/
    }

}

public class TCPClosed implements TCPState {
    //estado concreto 02
    public void open(TCPConnection conn) {
        /*lógica para tratar uma requisição
        de OPEN quando uma conexão está
        fechada*/
    }
    public void close(TCPConnection conn) {
        /*lógica para tratar uma requisição
        de CLOSE quando uma conexão está
        fechada*/
    }
}
```

State

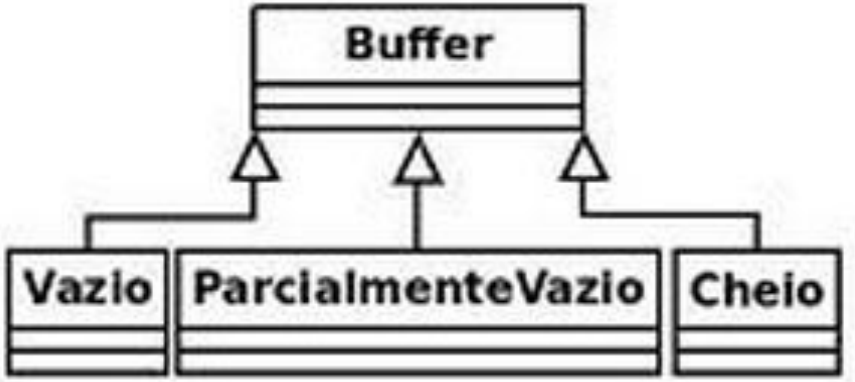
```
public class StateDemo {
   public static void main( String arg[] )  {

         //criando o objeto de contexto
         TCPConnection conn = new TCPConnection();

         //configurando o objeto para o estado CLOSED
         TCPState stateClosed = new TCPClosed();
         conn.setState(stateClosed);

         //requisitando um comportamento do objeto
         conn.open();

         //mudando o estado do objeto
         TCPState stateEstablished = new TCPEstablished();
         conn.setState(stateEstablished);

         //chamando o MESMO comportamento, mas o tratamento será DIFERENTE!!!
         conn.open();
   }
}
```

Exercícios [18]

(COPEVE-UFAL – UFAL 2011)

[51] O diagrama de classes apresentado na figura a seguir não Representa fielmente um buffer que passa por estados sucessivos de transformação. Em outras palavras, um buffer, que está inicialmente vazio, depois pode ficar parcialmente cheio e, possivelmente, pode ficar cheio. Dentre as opções apresentadas a seguir, qual o padrão de projetos que melhor se adequaria para modelar essa característica dinâmica do buffer?

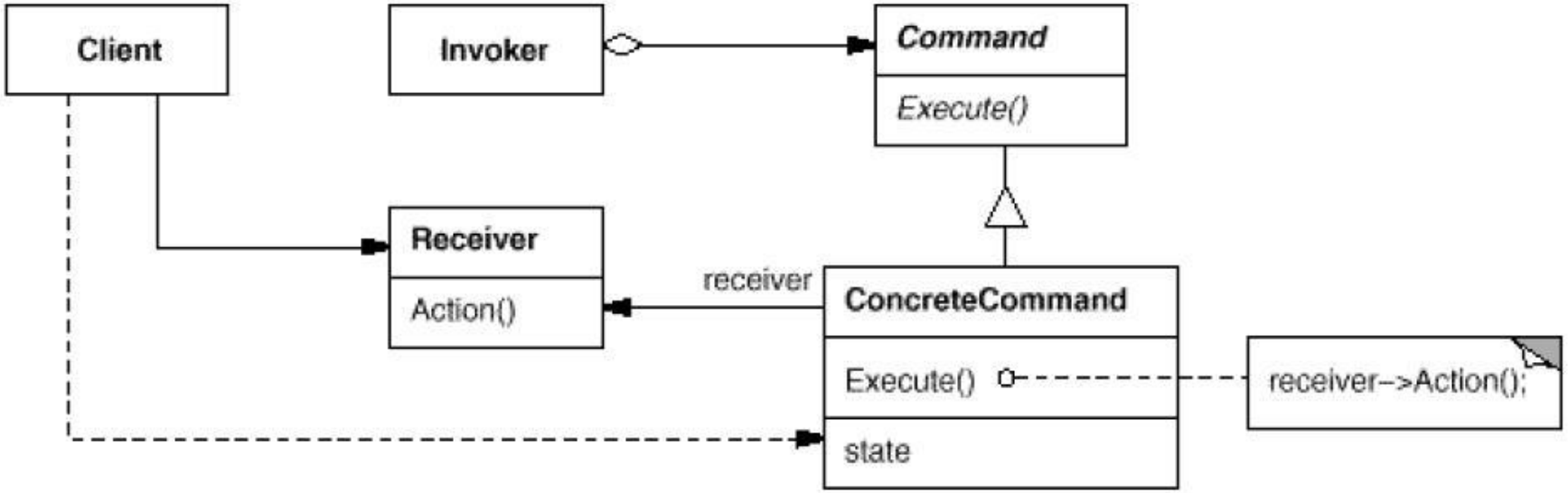
a) Singleton.
b) Dynamic behavior.
c) Mediator.
d) Composite.
e) State.



Command

- Encapsula uma requisição como um objeto, deixando-o, dessa forma, parametrizar os clientes com diferentes requisições
- Use o Command para:
  - Parametrizar objetos para realizar alguma ação
  - Suportar *undo*
  - Suportar transações

Command


Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver->Action();

Command

O invoker "controla" os comandos. Aqui eles são genéricos, para ser possível variá-los mais tarde

```
/* Invoker */
public class Interruptor {

    private Command ligar;
    private Command desligar;

    public Interruptor(Command ligar, Command desligar) {
        this.ligar = ligar;
        this.desligar = desligar;
    }
    public void ligar() { ligar.execute(); }
    public void desligar() { desligar.execute();}
}

/* Receiver */
public class Luz {
     public Luz() {  }
     public void acender() { System.out.println("Acendeu"); }
     public void apagar() { System.out.println("Apagou"); }
}
```

```
/*Command*/
public interface Command {
    void execute();
}
```

Comando genérico (abstrato)

Command

```
/*Command concreto*/
public class CommandAcender implements Command {
    private Luz aLuz;
    public CommandAcender(Luz luz) {
         this.aLuz=luz;                    }

    public void execute(){
        aLuz.acender();
    }
}

/* Command concreto */
public class CommandApagar implements Command {
    private Luz aLuz;
    public CommandApagar(Luz luz) {
         this.aLuz = luz;
    }
    public void execute() {
        aLuz.apagar();
    }
}
```

Command (executando)

```
public class Aplicacao {

    public static void main(String[] args) {
        Luz luz = new Luz();
        Command acender = new CommandAcender(luz);
        Command apagar = new CommandApagar(luz);

        Interruptor i = new Interruptor(acender,apagar);

        if(args[0].equalsIgnoreCase("ON"))
            i.ligar();
        else if(args[0].equalsIgnoreCase("OFF"))
            i.desligar();
    }
}
```

Exercícios [19]

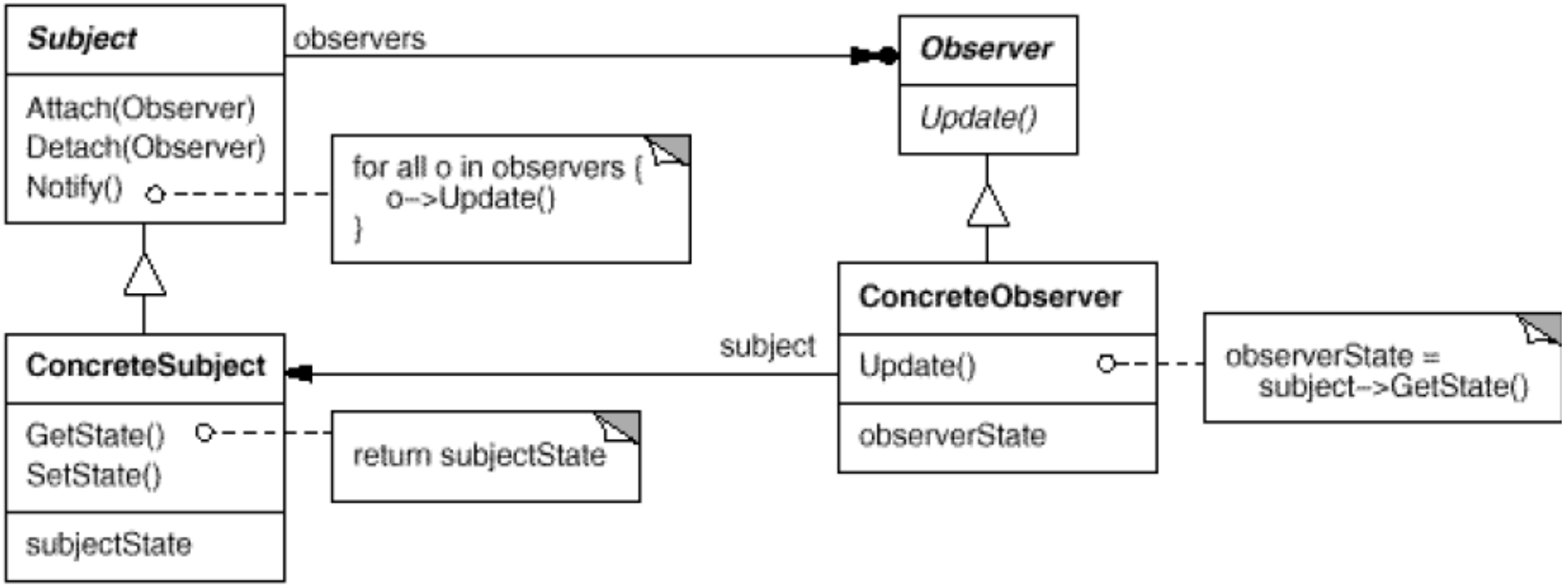
(INMETRO – CESPE 2009)

[101] O uso do padrão Command apresenta consequências como um objetoCommand é usualmente refratário ao enfileiramento; um objeto Command é usualmente transiente, isto é, não é passível de serialização e o uso disseminado de Commands dificulta a estruturação de um sistema em operações de alto nível.

Observer

- Define uma dependência entre objetos de forma que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente
- Use o Observer quando
  - Quando uma mudança em um objeto requer mudanças em outros objetos e você não sabe quantos objetos serão mudados

Observer


Subject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState

Observer

```
public interface Sirene {
        public void adicionarObservador( Operario o );
        public void removerObservador( Operario o );
}

public interface Operario {

        public void atualizar(Sirene s);

}
```

Notificador abstrato

Observador abstrato

Observer

```
public class SireneConcreta implements Sirene {

        private Boolean alertaSonoro = false;
        private ArrayList observadores = new ArrayList();

        public void alterarAlerta(){
                if(alertaSonoro)
                        alertaSonoro = false;
                else
                        alertaSonoro = true;
                notificarObservadores();
        }

        public Boolean getAlerta(){ return alertaSonoro; }

        public void adicionarObservador(Operario o) { observadores.add(o); }

        public void removerObservador(Operario o) { observadores.remove(o); }

        private void notificarObservadores(){
                Iterator i = observadores.iterator();
                while(i.hasNext()){
                        Operario o = (Operario) i.next();
                        o.atualizar(this);
                }
        }
}
```

Notificador concreto

Mudança de estado

Notificação

Observer

Observador concreto

```
public class OperarioConcreto implements Operario {

        private SireneConcreta objetoObservado;

        public OperarioConcreto(SireneConcreta o){
                this.objetoObservado = o;
                objetoObservado.adicionarObservador(this);
        }

        public void atualizar(Sirene s) {
                if(s == objetoObservado){
                        System.out.println("[INFO] A Sirene mudou seu estado para: " +
                                                "  objetoObservado.getAlerta());
                }
        }
}
```

Observer (executando)

```
public class Aplicacao  {

        public static void main(String[] args) {

                SireneConcreta sirene = new SireneConcreta();
                // Sirete ja começa com valor default false

                OperarioConcreto obs1 = new OperarioConcreto(sirene);
                OperarioConcreto obs2 = new OperarioConcreto(sirene);
                // Já passando a sirene como parametro

                sirene.alterarAlerta();
                // Nesse momento é chamado o método atualizar
                // das instâncias obs1 e obs2, saída:
                // [INFO] A Sirene mudou seu estado para: true
                // [INFO] A Sirene mudou seu estado para: true

                sirene.alterarAlerta();
                //[INFO] A Sirene mudou seu estado para: false
                //[INFO] A Sirene mudou seu estado para: false

                // Obs: 2 saídas porque temos 2 observadores
        }
}
```

Exercícios [20]

(TRE/MS – FCC 2007)

[50-A] Um exemplo de padrão de projetos apresentado pelo Gang of Four (GOF) é o Observer, que é utilizado quando se faz necessária a instanciação de um e apenas um objeto de uma determinada classe.

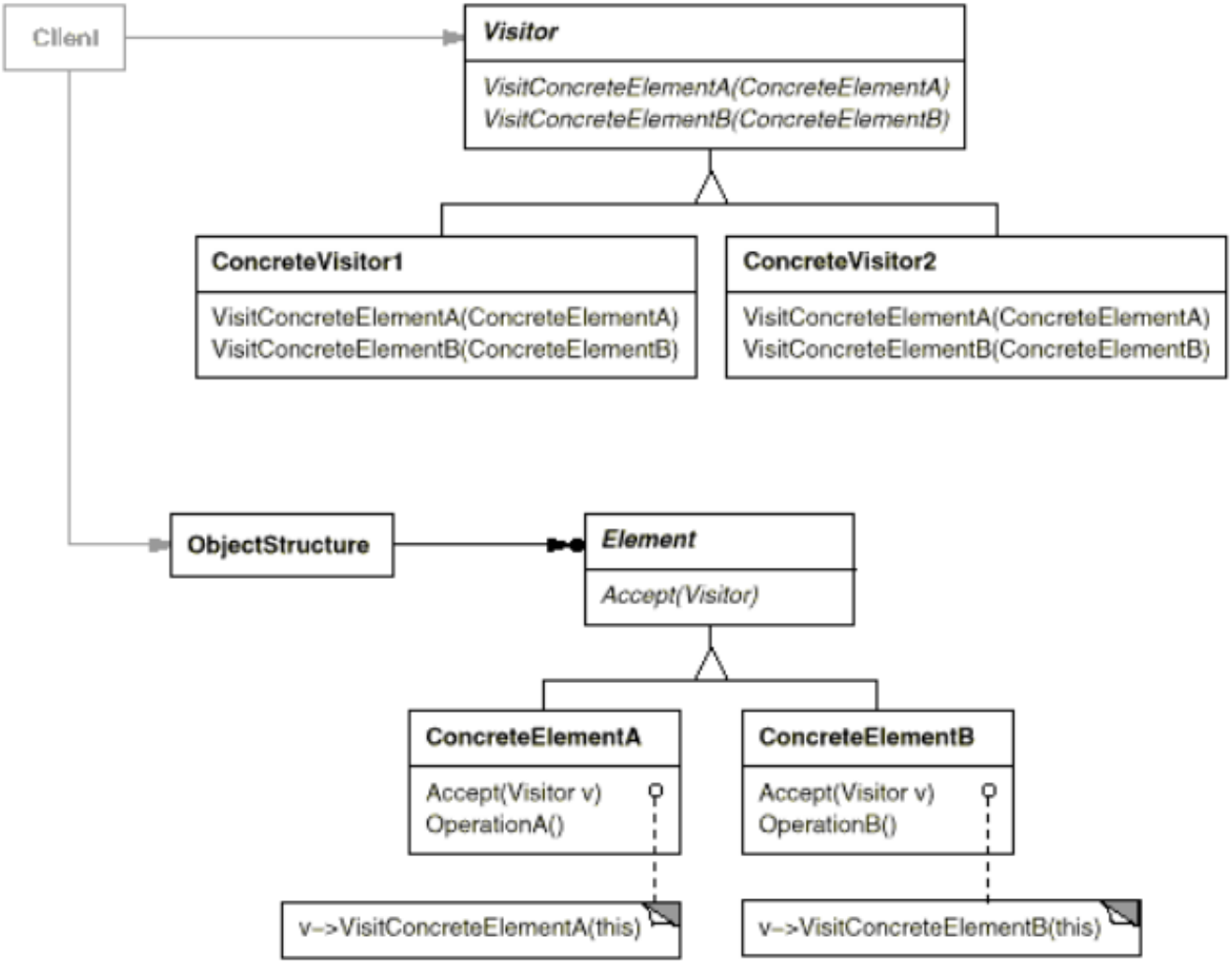
(Casa da Moeda – CESGRANRIO 2009)

[26] Em determinado sistema de análise estatística, é necessário definir uma dependência "um para muitos" entre objetos, de forma que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados. Que padrão de projeto pode ser utilizado nessa situação?
(A) AJAX (B) Memento (C) Singleton (D) Observer (E) JSON

Visitor

- Representa uma operação a ser executada sobre os elementos da estrutura de um objeto
- Permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera
- Use Visitor quando:
  - Muitas operações distintas e não relacionadas precisarem ser executadas sobre uma estrutura de objetos

Visitor


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Visitor

```
interface CarElementVisitor {
    //visitor abstrato para Elementos de um carro
    void visit(Wheel wheel);
    void visit(Engine engine);
}

interface CarElement {
    //objeto abstrato a ser visitado
    void accept(CarElementVisitor visitor);
}
```

Visitor (abstrato)
Elemento (abstrato)

Elementos concretos.
Serão visitados pelo Visitor.

```
class Wheel implements CarElement {
    //primeiro Objeto concreto
    private String name;
    public Wheel(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

Visitor

```
class Car implements CarElement {
    CarElement[] elements;

    public Car() {
        //cria o array de Elementos
        this.elements = new CarElement[] { new Wheel("dianteira esquerda"),
            new Wheel("dianteira direita"), new Wheel("traseira esquerda") ,
            new Wheel("traseira direita"), new Engine() };
    }

    public void accept(CarElementVisitor visitor) {
        for(CarElement elem : elements) {
            elem.accept(visitor);
        }
    }
}
```

Estrutura de objetos

Visitor concreto (poderia haver outros – um para cada operação)

```
class CarElementPrintVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visitando a roda " + wheel.getName());
    }
    public void visit(Engine engine) {
        System.out.println("Visiting motor");
    }
}
```

Visitor (executando)

```
public class VisitorDemo {
    static public void main(String[] args) {
        //Cria a estrutura de objetos
        Car car = new Car();

        //Visita cada um dos objetos chamando a operacao "visit"
        car.accept(new CarElementPrintVisitor());
    }
}
```

Ao percorrer os elementos que o aceitam, o Visitor pode executar operações diferentes sobre eles

Exercícios [21]

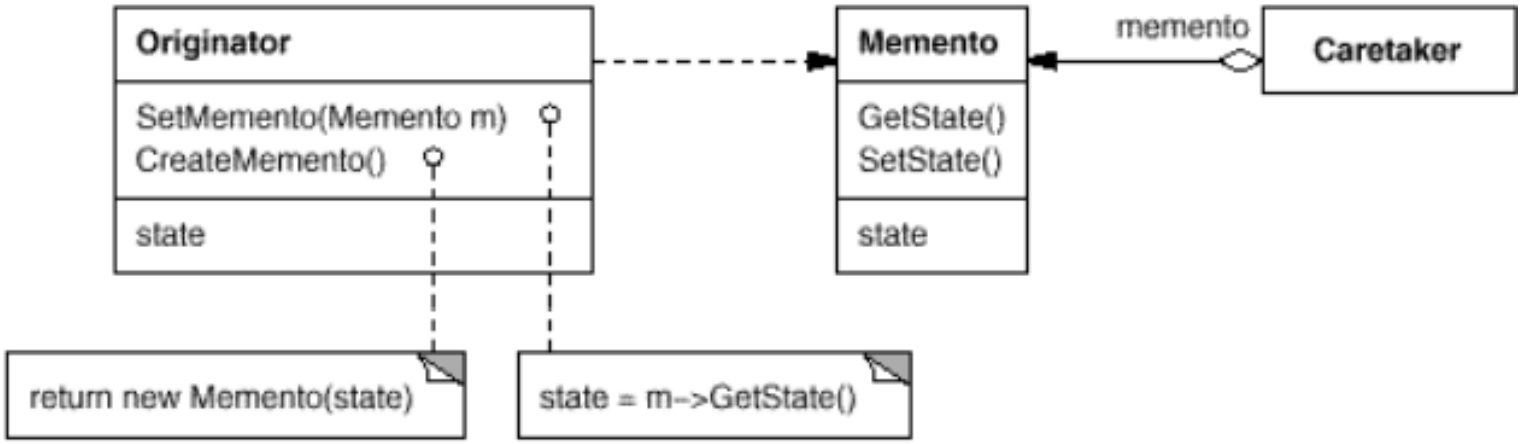
(TRE/AP – CESPE 2007)

[34-III] A implementação de tratadores de eventos de interface gráfica apóia-se mais no uso do padrão Observer que no uso do padrão Visitor.

Memento

- Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, de forma que ele possa ser recuperado depois
- Use Memento quando:
  - Uma "fotografia" (parte) do objeto precisa ser salva, de forma que ela possa ser recuperada depois

Memento


Originator
SetMemento(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
memento
Caretaker
return new Memento(state)
state = m->GetState()

Memento

```
class Originator {
    //o estado que vai ser salvo
    private String state;

    public void set(String state) {
        this.state = state;
    }

    public Memento saveToMemento() {
        return new Memento(state);
    }

    public void restoreFromMemento(Memento memento) {
        state = memento.getSavedState();
    }
}
```

Esta é a classe que vai ter a sua “fotografia” salva

Esta é a classe que vai guardar uma “fotografia”

```
public static class Memento {
    private final String state;

    public Memento(String stateToSave) {
        state = stateToSave;
    }
    public String getSavedState() {
        return state;
    }
}
```

Memento (executando)

```
class MementoDemo {
    public static void main(String[] args) {
        //Caretaker

        List<Memento> savedStates = new ArrayList<Memento>();

        Originator originator = new Originator();
        originator.set("State1");

        originator.set("State2");
        //salvando State 2 como um Memento
        savedStates.add(originator.saveToMemento());

        originator.set("State3");
        //salvando State 3 como um Memento
        savedStates.add(originator.saveToMemento());

        originator.set("State4");

        //dando rollback para o State 3
        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

Exercícios [22]

(PETROBRAS – CESGRANRIO 2006)

[37] Quanto à indicação para o uso dos padrões de projeto é FALSO afirmar que o padrão:

a) Abstract Factory é indicado quando: um sistema deve ser independente de como seus produtos são criados, compostos ou representados; um sistema deve ser configurado como um produto de uma família de múltiplos produtos; uma família de objetos-produto for projetada para ser usada em conjunto, e você necessita garantir esta restrição; você quer fornecer uma biblioteca de classes de produtos e quer revelar somente suas interfaces, não suas implementações.

b) Builder é indicado quando: uma classe não pode antecipar a classe de objetos que deve criar; uma classe quer que suas subclasses especifiquem os objetos que criam; classes delegam responsabilidade para uma dentre várias subclasses auxiliares, e você quer localizar o conhecimento de qual subclasse auxiliar que é a delegada.

Exercícios [22]

c) Mediator é indicado quando: um conjunto de objetos se comunica de maneiras bem definidas, porém complexas; a reutilização de um objeto é difícil porque ele referencia e se comunica com muitos outros objetos; um comportamento que está distribuído entre várias classes deveria ser customizável, ou adaptável, sem excessiva especialização em subclasses.

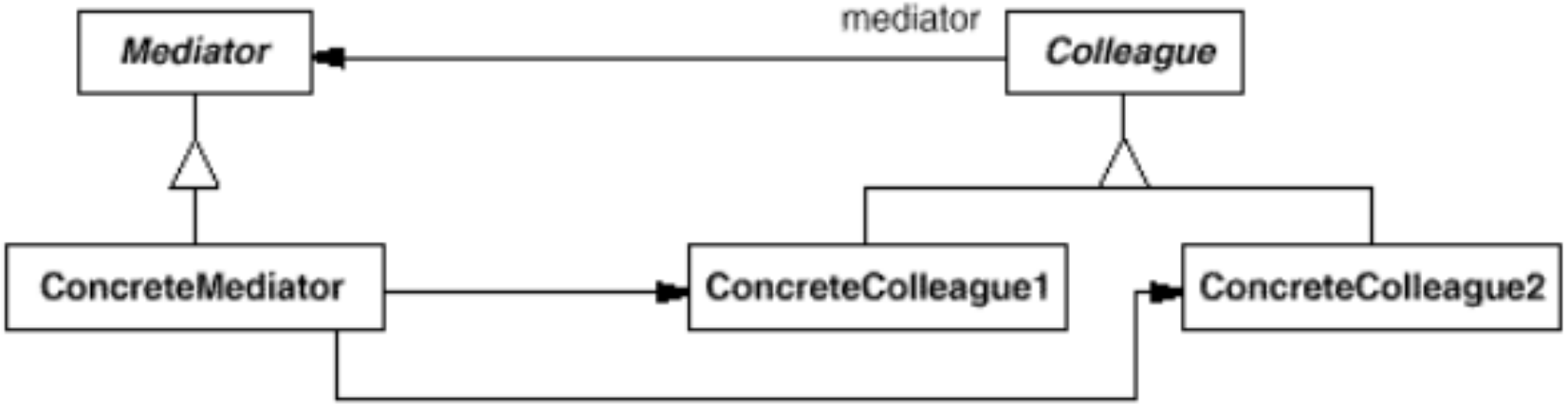
d) Memento é indicado quanto: um instantâneo de estado de um objeto deve ser salvo de maneira que possa ser restaurado para esse estado mais tarde; uma interface direta para obtenção do estado exporia detalhes de implementação e romperia o encapsulamento do objeto.

e) Composite é indicado quando: quiser representar hierarquias partes-todo de objetos; quiser que os clientes sejam capazes de ignorar a diferença entre composições de objetos e objetos individuais, neste caso, os clientes tratarão todos os objetos na estrutura composta de maneira uniforme

Mediator

- Define um objeto que encapsula a forma como um conjunto de objetos interage
- Promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros
- Use Mediator quando:
  - Um conjunto de objetos se comunica de maneira bem-definida,porém complexas
  - O reúso de um objeto é difícil, porque ele referencia e se comunica com muitos outros objetos

Mediator


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Mediator

Define a interface de comunicação entre objetos da classe Colleague

```
//Mediator interface
public interface Mediator {
    public void send(String message, Colleague colleague);
}
```

```
//Colleague interface
public abstract class Colleague
{
    private Mediator mediator;

    public Colleague(Mediator m) {
      mediator = m;
    }
    //envia uma msg através do Mediator
    public void send(String message) {
      mediator.send(message, this);
    }
    public abstract void receive(String message);
}
```

Super Classe de “colegas”

Mediator

```
public class ChatMediator implements Mediator {
    private ArrayList<Colleague> colleagues;

    public ChatMediator() {
        colleagues = new ArrayList<Colleague>();
    }

    public void addColleague(Colleague colleague){
        colleagues.add(colleague);
    }

    public void send(String message, Colleague originator) {
    //permita que todas as outras telas saibam que a nossa tela mudou
        for(Colleague colleague: colleagues) {
        //mas não precisa avisar a mim mesmo
            if(colleague != originator) {
                colleague.receive(message);
            }
        }
    }
}
```

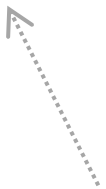
Mediator Concreto para o Chat

Vários tipos de “colegas”

```
public class DesktopColleague extends Colleague {
    public void receive(String message) {
        System.out.println("Desktop Received: " + message);
    }
}

public class MobileColleague extends Colleague {
    public void receive(String message) {
        System.out.println("Mobile Received: " + message);
    }
}
```

Mediator (executando)

```
public class MediatorDemo {
    public static void main(String[] args) {
        ChatMediator mediator = new ChatMediator();

        Colleague desktop = new DesktopColleague(mediator);
        Colleague mobile = new MobileColleague(mediator);

        mediator.addColleague(desktop);
        mediator.addColleague(mobile);

        desktop.send("Hello World");
        mobile.send("Hello");
    }
}
```



Na verdade, na implementação do método “send”, o Mediator está sendo chamado, e é ele quem se responsabiliza por coordenar a comunicação entre os vários “colegas”

Exercícios [23]

(UNEAL – COPEVE 2010)

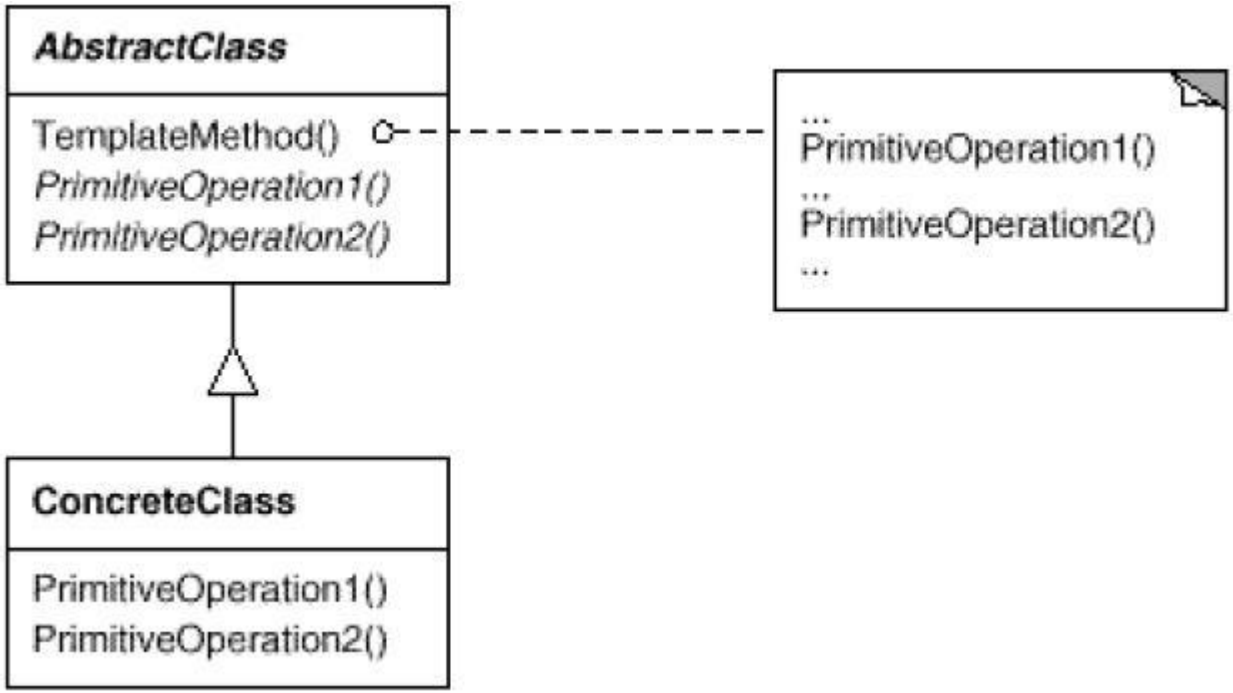
29. Definir um objeto que encapsula a forma como um conjunto de objetos interage. Promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente. Qual opção abaixo corresponde à descrição anterior?

A) Intenção do padrão de projeto proxy
B) Intenção do padrão de projeto composite
C) Intenção do padrão de projeto strategy
D) Intenção do padrão de projeto command
E) Intenção do padrão de projeto mediator

Template Method

- Define o esqueleto de um algoritmo em uma operação, deferindo alguns passos para as subclasses
- Template Method permite que subclasses redefinam certos passos de algum algoritmo sem mudar a estrutura do algoritmo
- Use Template Method para:
  - Implementar a parte invariante de um algoritmo uma vez e deixar para as subclasses a implementação do comportamento que pode variar

Template Method



Template Method

```
public abstract class AbstractClass {

        public final void templateMethod() {
                //aqui vai alguma implementação FIXA
                primitiveOperation1();
                primitiveOperation2();

        }
        public abstract void primitiveOperation1();
        public abstract void primitiveOperation2();
}
```

Classe abstrata, com o método template definido

```
public class Concrete1 extends AbstractClass {

        public void primitiveOperation1() {
                //implementação específica
        }

        public void primitiveOperation2() {
                //implementação específica
        }
}
```

```
public class Concrete2 extends AbstractClass {

        public void primitiveOperation1() {
                //implementação específica
        }

        public void primitiveOperation2() {
                //implementação específica
        }
}
```

Classes concretas, com implementações específicas

Template Method (executando)

```
public class TestTemplateMethod {
        public static void main(String[] args) {
                AbstractClass class1 = new Concrete1();
                AbstractClass class2 = new Concrete2();

                class1.templateMethod();
                class2.templateMethod();
        }
}
```

Executa-se o mesmo template method, que irá executar um bloco fixo de código mais os trechos variáveis, definidos em cada classe concreta

Exercícios [24]

(PETROBRAS - CESGRANRIO 2010)

[33] Um dos participantes da equipe de desenvolvimento de um framework deve implementar uma operação em uma das classes desse framework. Seja X o nome dessa classe. Essa operação implementa um algoritmo em particular. Entretanto, há passos desse algoritmo que devem ser implementados pelos usuários do framework através da definição de uma subclasse de X. Sendo assim, qual o padrão de projeto do catálogo GoF (Gang of Four) a ser usado pelo desenvolvedor do framework na implementação da referida operação, dentre os listados a seguir?

a) Singleton.

b) Decorator.

c) Interpreter.

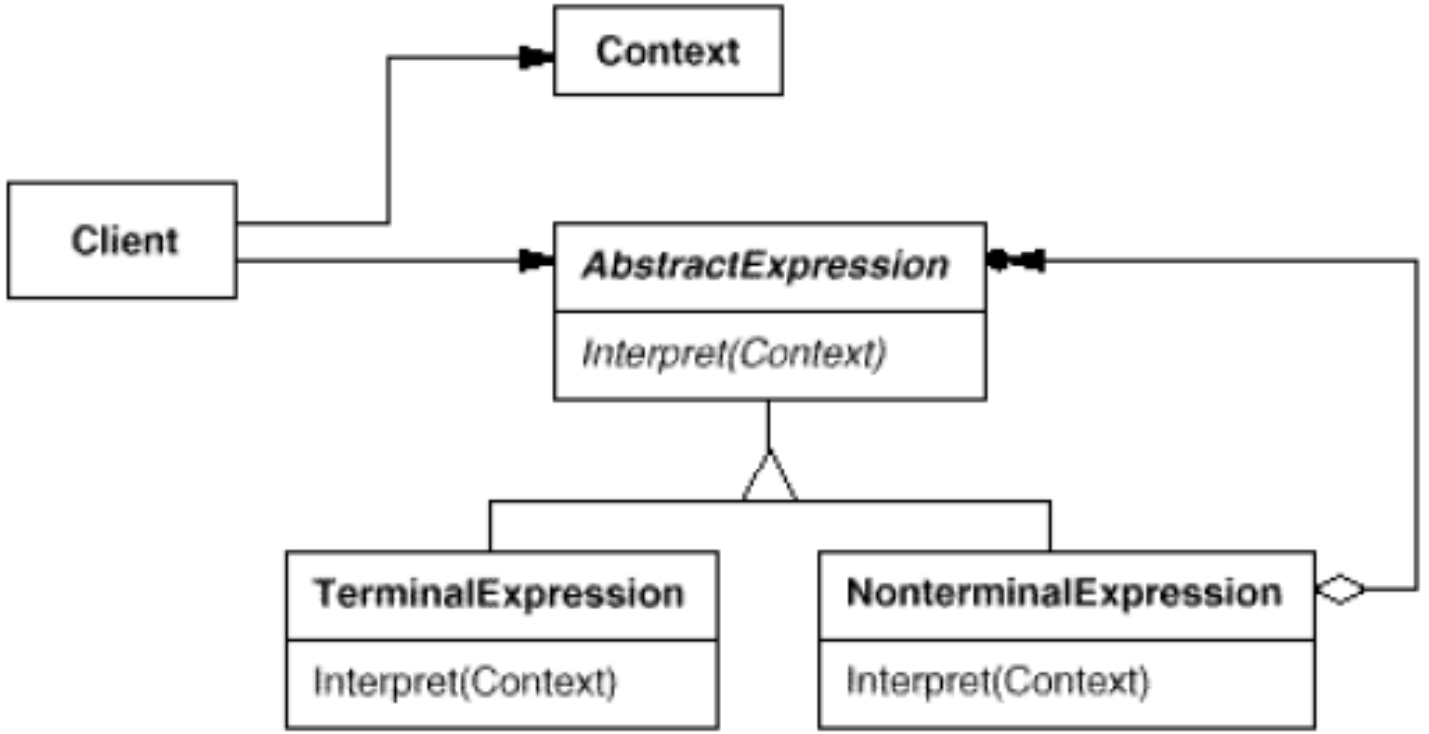
d) Template Method.

e) Observer.

Interpreter

- Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador para as sentenças dessa linguagem
- Use Interpreter quando:
  - Houver uma linguagem para interpretar e você puder representar as sentenças da linguagem como árvores sintáticas abstratas

Interpreter



Interpreter

```
public class ReprodutorDeNotas {
    /*Método que produz a onda sonora
    na frequencia que ele recebeu*/
    public void tocarSom(int freq) {
        /*toda a lógica de reproduzir
         o som em uma determinada
         frequencia*/
    }
}
```

```
public class ConstantesFrequenciasNotas {

    //valores em Hertz
    public static final int Do = 256;
    public static final int Re = 288;
    public static final int Sol = 320;
    ...
}
```

Sabemos como reproduzir uma nota em determinada frequencia, e sabemos as frequencias das notas. Mas como mapeamos uma Nota para uma determinada Frequência? Precisamos de um Interpreter.

Interpreter

```
public class InterpreterNotas {
    private Nota nota;

    //Método que recebe uma nota do teclado
    public void getNotaDoTeclado(Nota nota) {
        int freq = getFrequencia(nota);
        enviarNota(freq);
    }

    //Método que retorna a frequência a partir de uma nota
    private int getFrequencia(Nota nota) {
        int freq = /*faz o mapeamento entre a nota enviada
        e a sua frequencia, de acordo com as
        constantes especificadas...*/
        return freq;
    }


    /*Método que envia a frequência para algum instrumento
    eletrônico que irá produzir os sons*/
    private void enviarNota(int freq) {
        ReprodutorDeNotas rep = new ReprodutorDeNotas();
        rep.tocarSom(freq);
    }
}
```

Exercícios [25]

(MEC – FGV 2009)

[92] Os padrões de projeto orientados a objeto podem ter finalidade de criação, estrutural ou comportamental. Os padrões de criação se preocupam com o processo de criação de objetos. Os padrões estruturais lidam com a composição de classes ou de objetos. Os padrões comportamentais caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades. Assinale a alternativa que apresenta apenas padrões de projeto comportamentais.

a) Prototype, Abstract Factory e Builder.

b) Singleton, Composite e Interpreter.

c) Mediator, Interpreter e Command.

d) Composite, Decorato e Proxy.

e) Proxy, Builder e Mediator.

Gabaritos dos Exercícios

[1] – [52] B, [33] A, [82] C
[2] – [57-I] C
[3] – [58-I] E, [33-B] E, [88] C
[4] – [53] C, [50-C] E
[5] – [58-III] E
[6] – [58-II] C, [53] E
[7] – [68] E, [115] C
[8] – [58-IV] C, [33-A] E
[9] – [60] D
[10] – [50-D] E, [11-II] E
[11] – [88-C] E
[12] – [99] E
[13] – [59] D
[14] – [51] C
[15] – [50-B] C, [88-D] E
[16] – [33] D
[17] – [59] D
[18] – [51] E
[19] – [101] E
[20] – [50-A] E, [26] D
[21] – [34-III] V
[22] – [37] B
[23] – [29] E
[24] – [33] D
[25] – [92] C

FIM