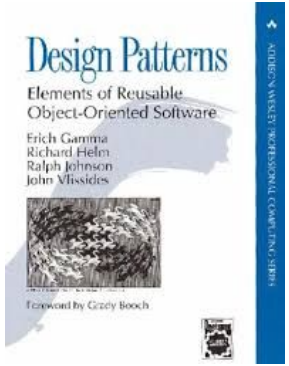
Padrões de Projeto

Prof. Marco Tulio Valente

mtov@dcc.ufmg.br

Padrões de Projeto



- Ideia proposta pelo arquiteto Cristopher Alexander:
  - A Pattern Language: Towns, Buildings, Construction, 1977
  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".
- Na área de projeto de software, popularizados pelo livro:
  - Design Patterns: Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995
  - Conhecido como “GoF (Gang of Four) book"

Definição e Objetivos

- "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."
- Objetivo:
  - “Catálogo de soluções” para problemas recorrentes no projeto de sistemas
  - “Vocabulário comum” para comunicar experiências, problemas e soluções recorrentes no projeto de sistemas
  - Ajudar na documentação e facilitar o entendimento de projetos
- Padrões de projeto favorecem “**design for change**”

Quando eu vou usar padrões de projeto?

- Em duas situações principais:
  - Quando estiver implementando o seu próprio sistema (principalmente se ele for um framework; mais detalhes nos slides seguintes). Neste caso, conhecer padrões de projeto pode te ajudar a adotar uma solução de projeto interessante, no seu próprio sistema
  - Quando estiver usando um sistema de terceiros (de novo, principalmente, se ele for um um framework). Neste caso, conhecer os principais padrões de projeto pode ajudar bastante a entender a lógica e os princípios de projeto deste sistema que precisa usar.

Classificação

- Segundo o **propósito**:
  - Criacionais: tratam do processo de criação de objetos
  - Estruturais: tratam da composição de classes e objetos
  - Comportamentais: tratam das interações e distribuição de responsabilidades entre classes e objetos
- Segundo o **escopo**: padrões podem se aplicar a:
  - Classes: propõem relacionamentos entre classes e subclasses
  - Objetos: propõem relacionamentos entre objetos

Padrões de Projeto (vamos estudar os em negrito)

- **Criacionais**
  - **Abstract factory**
  - Builder
  - **Factory method**
  - Prototype
  - **Singleton**
- **Estruturais**
  - Adapter
  - Bridge
  - Composite
  - **Decorator**
  - **Facade**
  - Flyweight
  - **Proxy**
- **Comportamentais**
  - Chain of responsibility
  - Command
  - Interpreter
  - **Iterator**
  - Mediator
  - Memento
  - **Observer**
  - State
  - **Strategy**
  - **Template method**
  - **Visitor**

Padrões de Projeto (GoF)

- **Nome** e classificação (criacional, estrutural ou comportamental)
- **Objetivo**: o que este padrão faz?
- **Outros nomes**
- **Motivação**: cenários de uso do padrão
- **Aplicações**: em quais outros cenários o padrão pode ser usado
- **Estrutura**: diagrama de classes do padrão (notação OMT)
- **Participantes**: papel desempenhado pelas classes do padrão
- **Colaboração**: “mecânica” de funcionamento do padrão
- **Consequências**: “trade-offs” resultantes do uso do padrão
- **Implementação**: como implementar o padrão
- **Código de exemplo**: implementações de exemplo (em C++)
- **Usos conhecidos**: sistemas que usam o padrão
- **Padrões relacionados**

(1) Factory Method

- Define uma interface para criação de objetos, deixando para subclasses a definição da classe concreta a ser instanciada
- Usado quando uma classe não pode antecipar o tipo dos objetos que a mesma precisa criar
- Suponha uma "tabela" implementada pela seguinte classe:

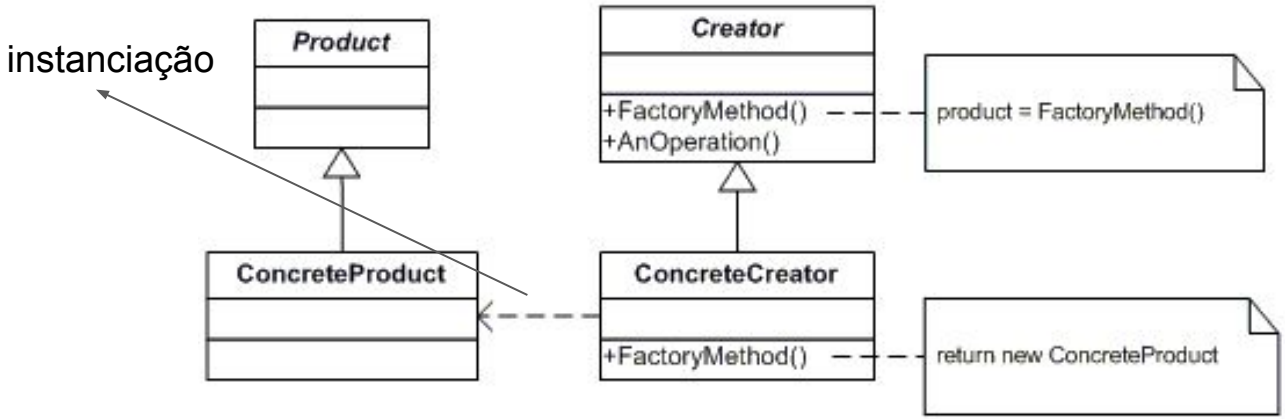
```
class MyTable {
   HashTable create() { return new HashTable; }
   add(...) { ... }
   search(...) { ... }
}
```

- Problema: trecho de código acima é pouco robusto a mudanças; amanhã, podemos querer usar uma tabela persistente, um arquivo XML, SGBD etc

Factory Method: Exemplo 1

```
interface Table {
   ....
}
class PersistentTable implements Table {
   ....
}
abstract class MyTable { // reutilizada em diversos registros
    abstract Table createTable();
    insert(...) { ... }
    search(..) { ... }
}
class MyPersistentTable extends MyTable {
    Table createTable() { return new PersistentTable(); }
}
```

Factory Method: Diagrama de Classes



Deve ser usado quando uma classe não consegue antecipar o tipo de objetos que ela precisa manipular; assim, ela delega essa responsabilidade para suas subclasses

- Creator: MyTable
- FactoryMethod: createTable()
- ConcreteCreator: MyPersistentTable
- Product: Table
- ConcreteProduct: HashTable, PersistentTable, XMLTable etc

Fonte: http://www.dofactory.com/Patterns/Patterns.aspx
(incluindo figuras dos próximos slides)

(2) Abstract Factory

- Define uma interface para criação de famílias de objetos, sem requerer a especificação de suas classes concretas; isto é, trata-se de uma interface inteira, contendo métodos-fábrica
- Suponha o código de um sistema que usa TCP/IP para comunicação:

```
void foo() {
    ....
    TCPSocket s = new TCPSocket();
    ....
    TCPPort p = new TCPPort();
    ....
}
```

- Problema: esse trecho de código é pouco reutilizável; veja que amanhã podemos querer trabalhar com outros tipos de protocolos (UDP, HTTP)

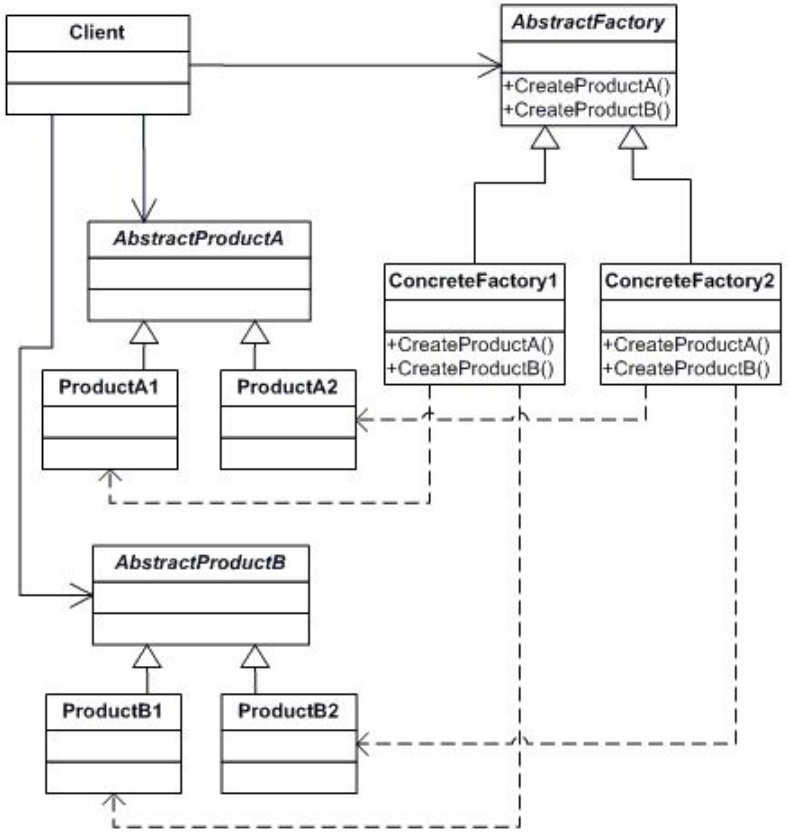
Abstract Factory: Exemplo

```
void foo(ProtocolFactory pf) {
   ....
   Socket s = pf.createSocket();
   ....
   Port p = pf.createPort();
   ....
}
....
class TCPFactory implements ProtocolFactory {
   public Socket createSocket() { return new TCPSocket(); }
   public Port createPort() { return new TCPPort(); }
}
....
ProtocolFactory pf = new TCPFactory();
foo(pf);
....
```

Abstract Factory: Quando usar?

- Quando se quer tornar um sistema independente dos "produtos" que ele cria (ou depende para funcionar)
- No exemplo, tais "produtos" são os protocolos que o sistema utiliza para comunicação. Ao adotar o padrão Abstract Factory:
  - Sistema apenas manipula as interfaces dos "produtos" que ele usa (Socket, Port etc)
  - Implementações concretas desses "produtos" são fornecidas por subclasses de uma "Abstract Factory"
  - Assim, sistema não sabe os "produtos" concretos que ele usa
- Veja que, nesta explicação, produtos são objetos

Abstract Factory: Diagrama de Classes

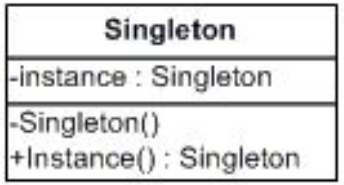


- AbstractProductA: Socket
  - ProductA1: TCPSocket
  - ProductA2: UDPSocket
- AbstractProductB: Port
  - ProductB1: TCPPort
  - ProductB2: UDPPort
- AbstractFactory: ProtocolFactory
  - ConcreteFactory1: TCPFactory
  - ConcreteFactory2: UDPFactory
- TCPFactory:
  - CreateSocket()
  - CreatePort()
- UDPFactory:
  - CreateSocket()
  - CreatePort()

(3) Singleton

- Usado para garantir que uma classe possui uma única instância
- Exemplo: servidor com uma única instância (de impressora, de nomes etc)

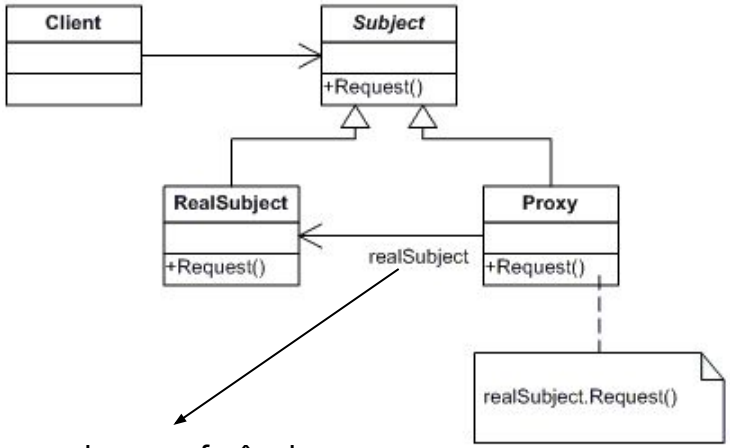
```
class MyServer {
  private static MyServer instance;
  private MyServer() {}     // evita que cliente use new
   MyServer()
  public static MyServer getInstance()
    if(instance == null)
      Instance = new MyServer();
    return instance;
  }
}
```

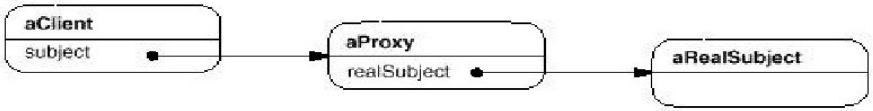


- Uso: `MyServer s = MyServer.getInstance();`

(4) Proxy

- Cria um “intermediário” que controla o acesso a um objeto base
- Cliente não possui uma referência direta para o objeto base, mas sim para o proxy; por sua vez, proxy possui uma referência para o objeto base
- Proxy implementa as mesmas interfaces do objeto base





Funcionamento: cliente invoca método do proxy; proxy realiza uma “determinada tarefa” e eventualmente repassa a requisição do cliente para o objeto base

Exemplos de Proxy (e sua função)

- Remote proxy: comunicação com um cliente remoto (isto é, encapsular protocolos e detalhes de comunicação; às vezes, chamado de stub
- Virtual proxy: alocação de memória por demanda para objetos que consomem muita memória
- Protection proxy: controlar o acesso ao objeto base por parte de diferentes clientes
- Cache proxy: armazenar em um cache resultados de métodos idempotentes
- Outros usos (requisitos não-funcionais): tolerância a falhas, persistência, distribuição de carga, sincronização etc

(5) Adapter

- Converte a interface de uma classe para outra interface, esperada pelos seus clientes; objetivo: permitir que classes trabalhem juntas, o que não seria possível antes devido à incompatibilidade de interfaces
- Suponha que uma classe possua o seguinte método copy:

```
Copyable[] copy (Copyable[] v);
```

- A interface Copyable possui o seguinte método:

```
boolean isCopyable();
```

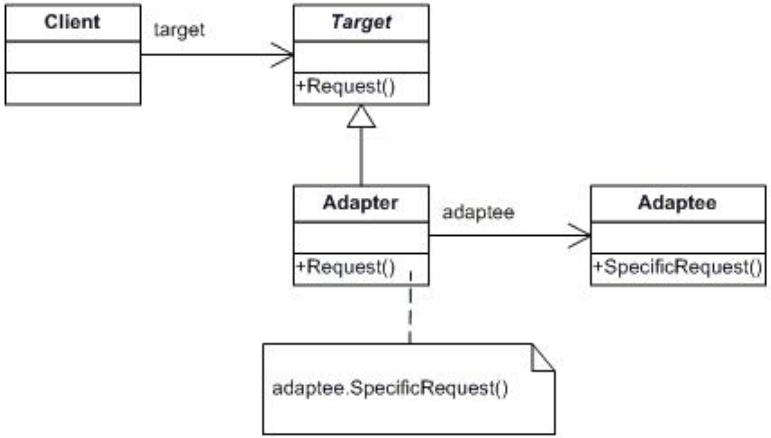
- copy apenas copia elementos de v sobre os quais isCopyable retorna true
- Agora suponha que precisamos copiar objetos Document, e que não dispomos do código desta classe; e ela não implementa a interface Copyable, mas implementa um método equivalente chamado isValid()

Um Adapter é a Solução para o problema anterior

```
class DocumentAdapter implements Copyable { // classe adaptadora
  private Document document;
  public DocumentAdapter (Document document) {
    this.document = document;
  }
  public boolean isCopyable() {
    return document.isValid();
  }
}
```

- Evidentemente, devemos inserir no vetor a ser copiado objetos do tipo DocumentAdapter e não do tipo Document

Adapter: Diagrama de Classes

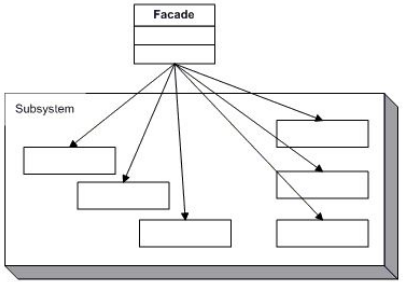


- Analogia com o diagrama de classes do padrão:
  - Target: Copyable
  - Request: isCopyable()
  - Adapter: DocumentAdapter
  - Adaptee: Document
  - SpecificRequest: isValid

(6) Façade

- Define uma interface de mais alto nível, a qual torna a utilização de um componente mais fácil; suponha o exemplo de um compilador:

```
Scanner s = new Scanner(“exemplo1.java”);
Parser p = new Parser(s);
AST ast = p.parse();
......
JVMCodeGenerator jvm = new JVMCodeGenerator(ast)
jvm.code()
```



- Logo, esse código requer conhecimento dos subsistemas internos de um compilador; seria mais fácil se existisse um único método:

```
void compila(String nomeArq)
```

que realizasse tais tarefas; ele seria uma “fachada” para o compilador

(7) Decorador

- Adicionam dinamicamente novas responsabilidades a um objeto
- Constituem uma alternativa a subclasses quando se precisa adicionar novas funcionalidades em uma classe
- São transparentes aos clientes do objeto decorado
- Suponha a seguinte interface:

```
interface Channel {
  public void send(byte[] m) throws NetworkException;
  public byte[] recv() throws NetworkException;
}
```

- Suponha uma classe TCPChannel:

```
class TCPChannel implements Channel { .... }
```

Exemplo **sem** Decorador

- Suponha um TCPChannel que compacte/descompacte os dados enviados/recebidos:

```
class TCPZipChannel extends TCPChannel {
  public void send(byte[] m) throws NetworkException {
    m = “compactação do vetor m”
    super.send(m);
  }
  public byte[] recv() throws NetworkException {
    byte[] m = super.recv();
    m = “descompactação do vetor m”
    return m;
  }
}
```

Exemplo **sem** Decorador

- Suponha um TCPChannel com um buffer associado (dados são escritos em um buffer; quando buffer cheio são enviados)

```
class TCPBufferChannel extends TCPChannel { .... }
```

Exemplo **sem** Decorador

- Algumas combinações possíveis:
  - TCPZipChannel → TCPChannel
  - TCPBufferedChannel → TCPChannel
  - TCPBufferedZipChannel → TCPZipChannel → TCPChannel
  - TCPLogChannel → TCPChannel
  - TCPLogBufferedZipChannel → TCPBufferedZipChannel → TCPZipChannel → TCPChannel
  - UDPZipChannel → UDPChannel
  - UDPBufferedChannel → UDPChannel
  - UDPBufferedZipChannel → UDPZipChannel → UDPChannel
  - UDPLogChannel → UDPChannel
  - UDPLogBufferedZipChannel → UDPBufferedZipChannel → UDPZipChannel → UDPChannel
- **Problema**: extensão via herança é estática e inflexível

Solução com Decorador

- Suponha um TCPChannel que compacte/descompacte os dados enviados/recebidos:

```
Channel = new ZipChannel (new TCPChannel());
```

- Suponha um TCPChannel com um buffer associado:

```
Channel = new BufferChannel (new TCPChannel());
```

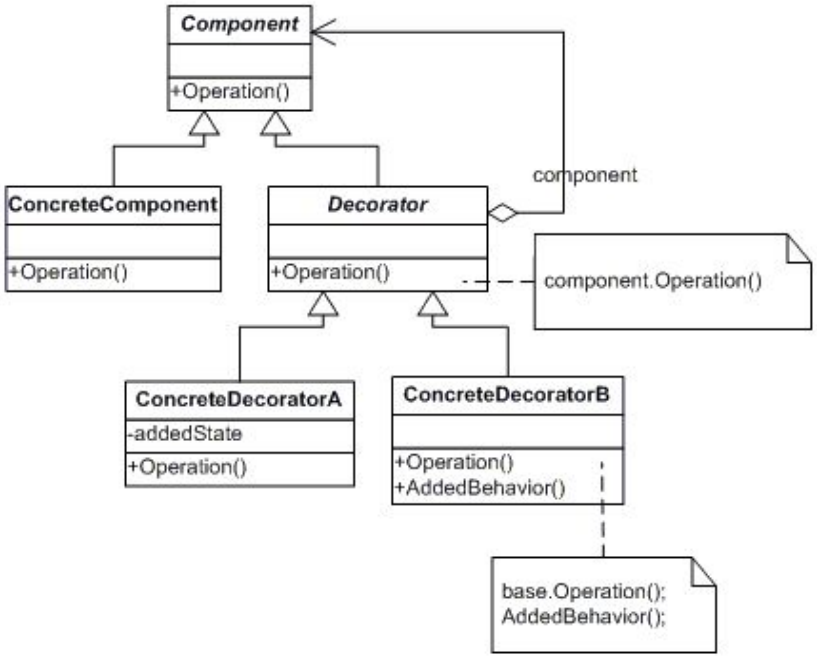
- Suponha um UDPChannel com um buffer associado:

```
Channel = new BufferChannel (new UDPChannel());
```

- Suponha um TCPZipChannel com um buffer associado:

```
Channel= new BufferChannel (new ZipChannel (new TCPChannel());
```

Decorador: Diagrama de Classes



Analogia com exemplo dos canais:

Component: Channel

ConcreteComponent: TCPChannel, UDPChannel

Decorator: ChannelDecorator

ConcreteDecoratorA: ZipChannel

ConcreteDecoratorB: BufferChannel

Decorador: Implementação (1)

```
interface Channel {
  public void send(byte[] m) throws NetworkException;
  public byte[] recv() throws NetworkException;
}

class ChannelDecorator implements Channel {
  protected Channel channel;
  public ChannelDecorator(Channel ch){
     this.channel = ch;
  }
  public void send(byte[] m) throws NetworkException {
     channel.send(m);
  }
  public byte[] recv() throws NetworkException {
     return channel.recv();
  }
}
```

Decorador: Implementação (2)

```
public class ZipDecorator extends ChannelDecorator {
   public ZipDecorator(Channel c) {
      super(c);
   }
   public void send(byte[] m) throws NetworkException {
      m = “compactação do vetor m”
      super.channel.send(m);
   }
   public byte[] recv() throws NetworkException {
      byte m[] = super.channel.recv();
      m= “descompactação do vetor m”
      return m;
   }
}
```

(8) Iterador

- Disponibilizam modos de acesso a objetos compostos sem expor a representação interna de tais objetos
- Objetos compostos: listas, árvores, tabelas hash etc
- Encapsulam código para percorrer uma estrutura de dados
- Muitas vezes, deseja-se percorrer uma estrutura de dados sem conhecer seu tipo concreto
- Para isso, basta que o cliente conheça a interface desta estrutura
- Assim, pode-se mudar de estrutura de dados (de uma lista para uma árvore, por exemplo), sem afetar seus clientes

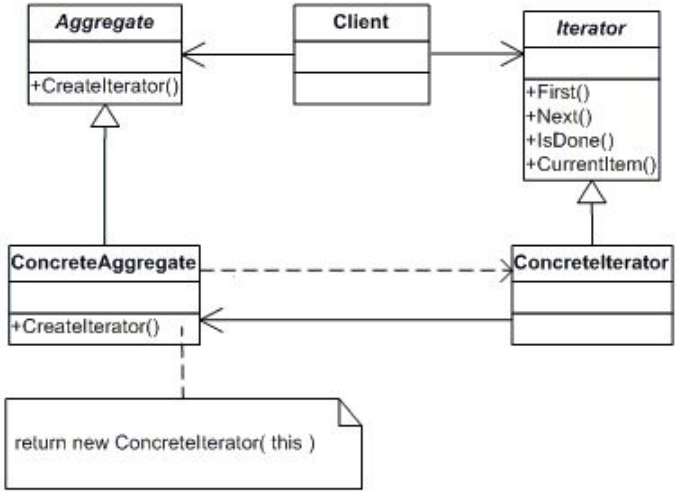
Iteradores

- Disponibilizam uma interface para percorrer diversas estruturas de dados
- Permitem que vários caminhamentos sobre uma estrutura estejam ativos

Iterador: Exemplo

```
void foo(List list) {
  Iterator iterator = list.createIterator(list);
  iterator.first();
  while (!iterator.IsDone()) {
    Object item = iterator.CurrentItem();
    // Code here to process item.
    iterator.next();
  } ...
}

class LinkedList implements List {

   ...

   Iterator createIterator (LinkedList list) {

     return new LinkedListIterator(list);

   }
...}
```

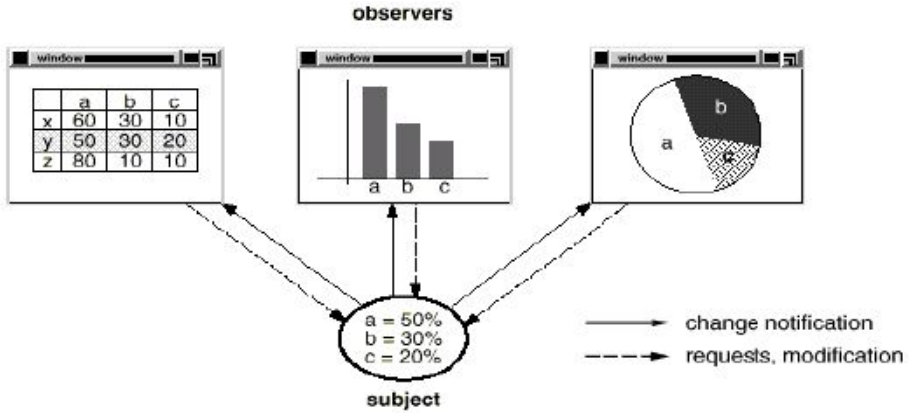
Iterador: Diagrama de Classes



- Analogia com exemplo anterior:
  - Aggregate: List
  - ConcreteAggregate: LinkedList
  - ConcreteIterator: LinkedListIterator

(9) Observador

- Define uma dependência do tipo um-para-muitos entre objetos, de forma que quando um objeto tem seu estado alterado, todos os seus dependentes são notificados
- Usado quando a mudança do estado de um objeto (chamado de Subject) requer que outros objetos sejam notificados (chamados de Observers)
- Uso muito comum em aplicações com interfaces gráficas



Fonte: GoF book

Observador: Exemplo

```
public interface Observer {
  public void update(Subject s);
}
public class Subject {
  private List observers = new LinkedList();
  public void addObserver(Observer observer) {
    observers.add(observer);
  }
  public void removeObserver(Observer observer) {
    observers.remove(observer);
  }
  public void notifyObservers() {
    Iterator it = observers.iterator();
    while (it.hasNext()) {
      Observer obs= (Observer) it.next();
      obs.update(this);
    }
  }
}
```

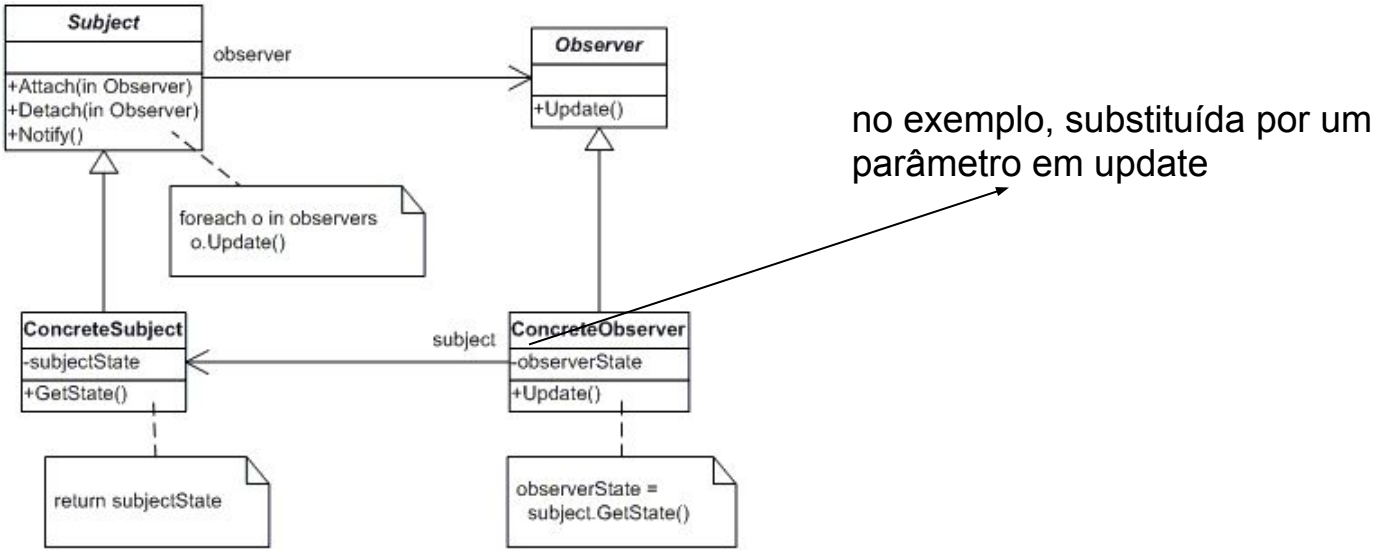
Observador: Exemplo (cont.)

```
public class Temperature extends Subject  {
  private double value;
  public double getValue() {
    return value;
  }
  public void setValue(double value) {
    this.value = value;
    notifyObservers();
  }
}

public class TermometerCelsius implements Observer  {
  public void update(Subject s) {
    double value = ((Temperature) s).getValue();
    System.out.println("Celsius: " + value);
  }
}
```

Observador: Exemplo (cont.)

```
public class TermometerFahrenheit implements Observer {
  public void update(Subject s) {
    double value = 1.8 * ((Temperature) s).getValue() + 32;
    System.out.println("Fahrenheit: " + value);
  }
}

public class TesteTemperature {
  public static void main(String[] args) {
    Temperature t = new Temperature(10);
    t.addObserver(new TermometerCelsius ());
    t.addObserver(new TermometerFahrenheit ());
    t.setValue(100);
  }
}
Saída:
Celsius: 100.0
Fahrenheit: 212.0
```

Observador: Diagrama de Classes



- Analogia com exemplo da Temperatura e dos Termômetros:
  - Subject e Observer
  - ConcreteSubject: Temperature
  - ConcreteObserver: TermometerCelsius, TermometerFahrenheit etc

(10) Strategy

- Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis.
- Permite parametrizar os algoritmos usados por uma classe
- Recomendado quando uma classe é usuária de um certo algoritmo, por exemplo, um algoritmo de ordenação; como existem diversos algoritmos de ordenação, não se quer codificar todos eles dentro desta classe

Motivação: Qual o problema com essa lista?

```
class MyList {

     private ArrayList list; // dados de uma lista

     ... // metodos de uma lista: add, delete, search

     public sort() {
       ... // ordena lista usando Quicksort
     }
}
```

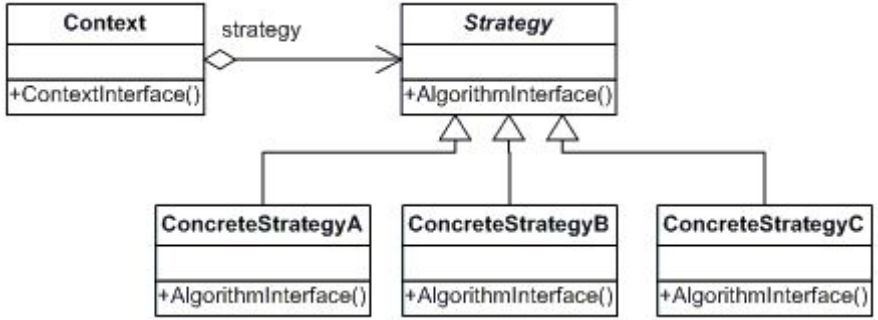
Strategy: Exemplo

```
class MyList {
     private ArrayList list; // dados de uma lista

     private SortStrategy strategy;

     public void setSortStrategy (SortStrategy strategy) {
       this.strategy = strategy;
     }

     public sort() {
       strategy.sort(list);
     }
}
```

Strategy: Exemplo

```
abstract class SortStrategy {   // interface comum aos algoritmos
    abstract void sort (ArrayList list);
}

class QuickSortStrategy extends SortStrategy {
    void sort (ArrayList list) { .... }
}

class ShellSortStrategy extends SortStrategy {
    void sort (ArrayList list) { .... }
}
```

Strategy: Diagrama de Classes



- Analogia com exemplo anterior:
  - Context: MyList
  - Strategy: SortStratety
  - ConcreteStrategy: QuickSortStratety, ShellSortStratety etc

(11) Template Method

- Às vezes em uma classe só conseguimos definir o template de um algoritmo (ou método); algumas partes do método são variáveis e somente podem ser implementadas em subclasses
- Suponha um framework de testes (como o JUnit, que veremos mais adiante neste curso). Suponha que classes de teste tenham que herdar de TestCase, a qual possui o seguinte método que vai executar os testes:

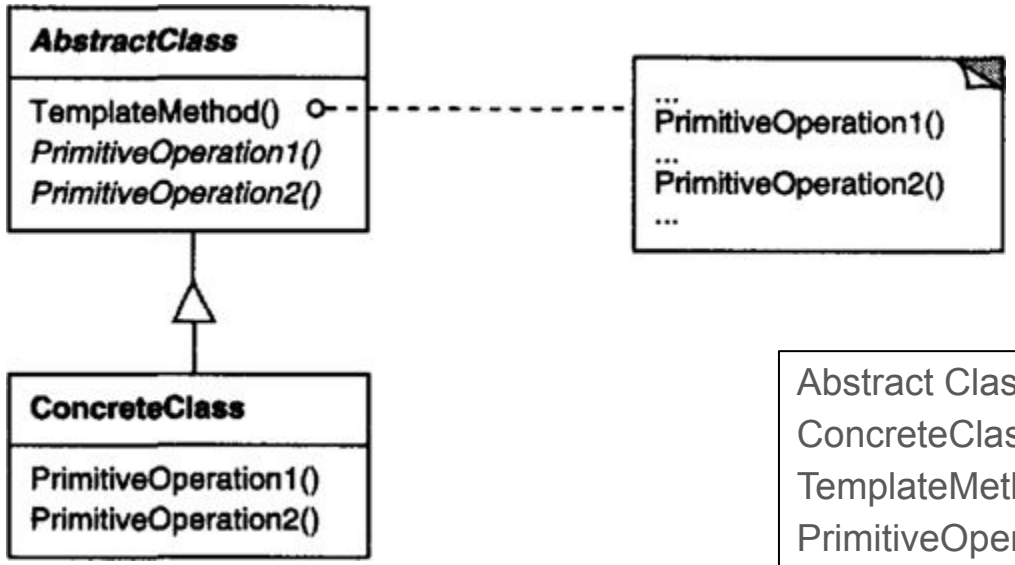
```
public void run() { // run = template method
    setUp();          // método abstrato
    runTest();        // método abstrato
    tearDown();       // método abstrato
}
```

- "Run" é um template method; os três métodos que ele chama são abstratos e são implementados em subclasses de TestCase

Template Method: Definição

- Define o esqueleto de um algoritmo; e posterga a implementação de alguns passos para métodos abstratos, implementados em subclasses
- Ou seja, permite que subclasses "customizem" um algoritmo (ou método), mas sem mudar a estrutura geral do mesmo
- Classe concreta herda o Template Method, mas sendo um template, ele não está pronto para uso; falta ainda implementar alguns métodos
- Se for interessante, a classe abstrata pode prover uma implementação default para alguns dos métodos "variáveis" (quando isso acontece, tais métodos com comportamento default são chamados de hook methods). Esse comportamento default pode ser simplesmente não fazer nada.

Template Method: Diagrama de Classes



Abstract Class: TestCase
ConcreteClass: MyTestCase
TemplateMethod: run()
PrimitiveOperation1: SetUp()
PrimitiveOperation2: runTest()
etc

Template Methods e Frameworks (1)

- Templates methods são muito usados na implementação de frameworks
- Frameworks são aplicações semi-prontas; normalmente, eles incluem inclusive o método main da aplicação.
- Exemplo de frameworks: Ruby on Rails (desenvolvimento Web) ou Angular, React ou Vue (desenvolvimento de front-ends em JavaScript)
- Frameworks são diferentes de bibliotecas, pois essas são passivas, isto é, o controle está sempre com a aplicação, que decide quando chamar os métodos da biblioteca
- Exemplo de biblioteca: java.util, que inclui várias classes (ArrayList, Set, Stack, Hashtable etc) para implementação de estruturas de dados.

Template Methods e Frameworks (2)

- Ao contrário de bibliotecas, frameworks são "ativos"
- Eles se valem do que se chama de Inversão de Controle ou Princípio de Hollywood (*don't call us, we'll call you*), para chamar métodos da aplicação
- Frozen spots: são as funcionalidades (ou o código) de um framework que não pode ser customizado; ele está implementado e pronto
- Hot spots: são as funcionalidades (ou o código) de um framework que pode ser customizado. Exemplo: o que fazer ao clicar em um botão da interface gráfica (evidentemente, este código não está pronto em um framework para construção de interfaces gráficas; mas cabe ao o usuário do framework implementá-lo).

Template Methods e Frameworks (3)

- No caso específico de frameworks, Template Methods podem ser usados para implementar Inversão de Controle
- Especificamente, para que uma superclasse (do framework) chame um método de uma subclasse (da aplicação)
- Esse recurso de linguagens OO, chamado de despacho dinâmico ("dynamic dispatching"), é muito interessante e poderoso
- Ele permite que "código mais antigo" chame "código mais novo":
  - Código antigo: código da classe base (TestCase, em nosso exemplo)
  - Código novo: código da subclasse que herda de TestCase
  - Claro que TestCase foi implementado antes de nossa subclasse

(12) Visitor

- Permite "adicionar" uma nova operação em uma família de objetos, sem que seja preciso modificar as classes dos mesmos
- Suponha uma classe Veiculo
- Suponha diversas subclasses de Veiculo: Carro, Onibus, Motocicleta etc
- Suponha que pretendemos disponibilizar em Veiculo um método "print", que vai imprimir os dados do Veiculo
- Problema: requisitos de print são "flexíveis"
    - Às vezes, quero imprimir na tela; às vezes, quero salvar em arquivo
    - Às vezes, quero imprimir msgs em português, às vezes, em inglês
    - Às vezes, quero imprimir só um "resumo" dos dados de um Veiculo; às vezes, quero imprimir os dados completos

Alternativa ao Uso do Padrão Visitor

- Tornar "print" um método abstrato em Veiculo
- E também abstrato em Carro, Onibus, Motocicleta etc
- Logo, ele seria implementado em subclasses
- Problema: cria facilmente uma explosão de subclasses
    - **class** CarroPrintOnScreen
    - **class** VeiculoPrintOnScreen
    - **class** OnibusPrintOnScreen
    - **class** CarroPrintOnFile
    - etc

Solução Usando um Visitor

```
interface Visitor {
   void visit(Carro c);
   void visit(Onibus o);
   void visit(Motocicleta m);
}

class VeiculoVisitorPrint {
   void visit(Carro c) { .... } // print de carro, conforme requisitado
   void visit(Onibus o) { .... } // idem, para Onibus
   void visit(Motocicleta m) { .... } // idem, para Motocicleta
}
```

Solução Usando um Visitor (cont.)

```
class Carro extends Veiculo {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    } ...
}

class Onibus extends Veiculo {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    } ...
}

// Idem para Motocicleta
```

Usando um Visitor (por exemplo, no main)

- Contexto:
  - Todas as instâncias de Veiculo estão em uma lista
  - Queremos fazer uma operação (isto é, "visitar") todos esses veículos
  - Para isso Veiculo aceita (accept) "visitas"
  - Porém, accept requer que seja passado o Visitor correto
  - No caso, uma instância de VeiculoVisitorPrint

```
VeiculoVisitorPrint myVisitor = new VeiculoVisitorPrint();
...
for (Veiculo v: list) {
  v.accept(myVisitor);
}
```

Mais uma solução alternativa ...

- O seguinte código poderia ser uma alternativa, mais simples do que Visitor
- Porém, ele **não** compila em Java:

```
VeiculoVisitorPrint myVisitor = new VeiculoVisitorPrint();
for (Veiculo veic: list) {
    myVisitor.visit(veic);
}
```

Visitor e Double Dispatch (1)

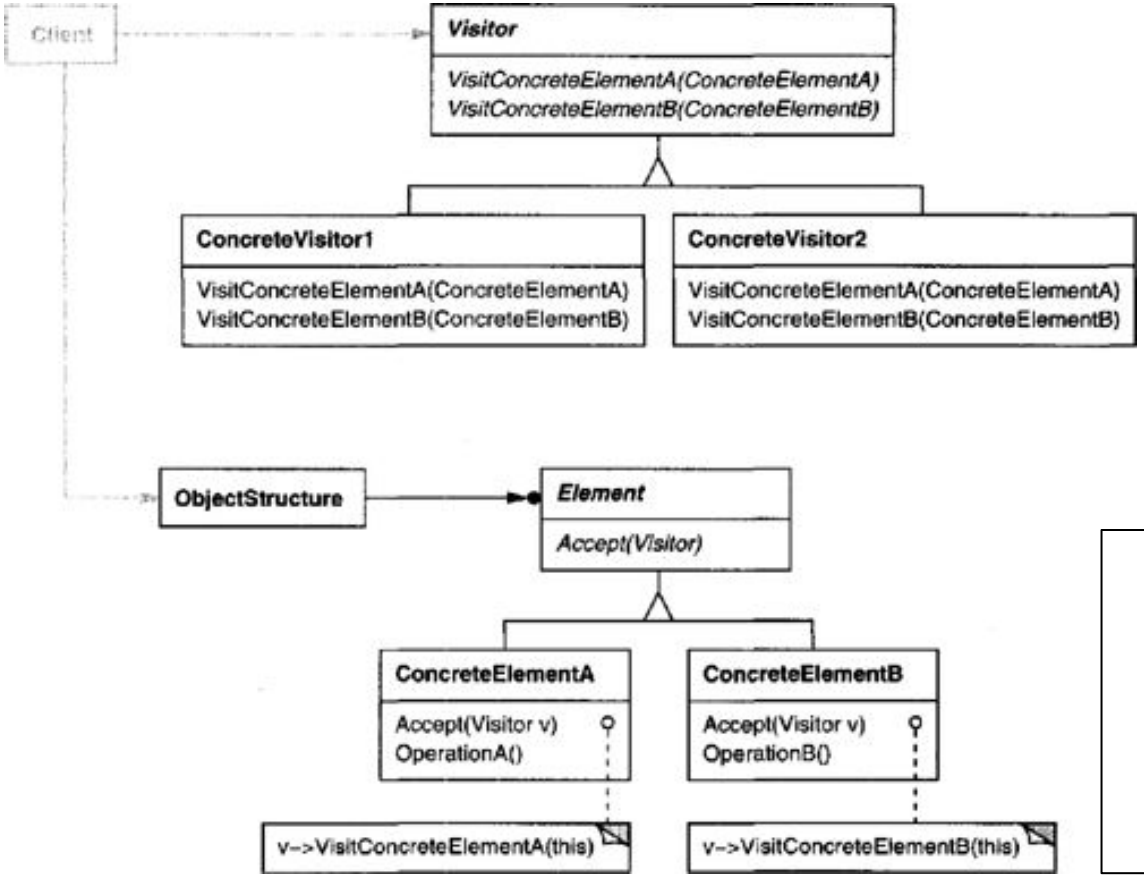
- No exemplo anterior, fica claro que o método visit a ser chamado depende do tipo do visitor (myVistor) e do tipo do objeto que se deseja visitar (veic)
- Porém, Java usa "single dispatching", isto é, apenas o tipo do target object (myVisitor) é usado para escolher o método a ser chamado
- O código, para compilar, exigiria que Java suportasse "double dispatch"
- Double dispatch: quando o tipo do target object e de um dos parâmetros de chamada são usados para escolher o método que será invocado

```
VeiculoVisitorPrint myVisitor = new VeiculoVisitorPrint();
for (Veiculo veic: list) {
    myVisitor.visit(veic);
}
```

Visitor e Double Dispatch (2)

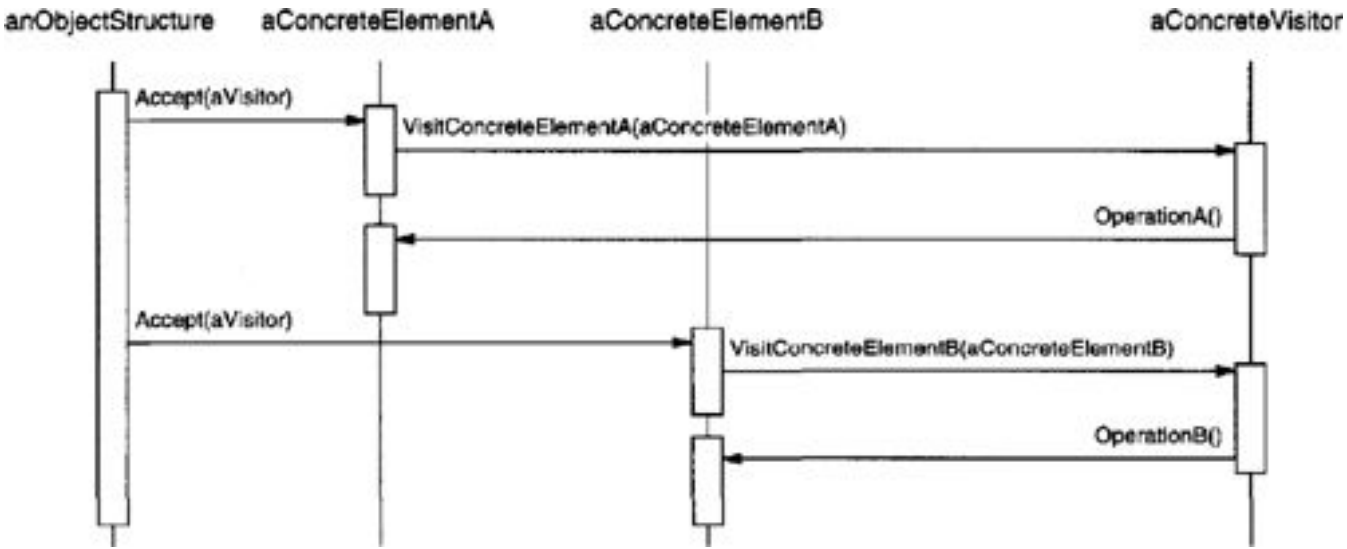
- Na verdade, o padrão Visitor pode ser entendido como uma maneira de simular "double dispatch" em Java
- Curiosidade: algumas linguagens suportam double dispatch diretamente, como Smalltalk e CLOS (mas não é o caso de Java, C++, C# etc).

Visitor: Diagrama de Classes



- Visitor: Visitor
- ConcreteVisitor1: VeiculoVisitorPrint
- ObjectStructure: list
- Element: Veiculo
- ConcreteElementA: Carro
- ConcreteElementB: Onibus

Visitor: Diagrama de Sequência



Visitor: Consequências

- Visitors facilitam a adição de um método em uma hierarquia de classes
- Um Visitor "congrega" operações relacionadas (no exemplo, um print de dados de Veiculos); poderia também existir um segundo Visitor, com outras operações (ex.: calcular imposto de veículos vendidos em MG)
- Por outro lado, a adição de uma nova classe na hierarquia (exemplo: Caminhao), implica que todos os Visitors (concretos) terão que ser atualizados, com um novo método: visit(Caminhao)
- Por fim, Visitors podem quebrar "encapsulamento"; por exemplo, Veiculo pode ter que implementar métodos públicos expondo seu estado interno (campos privados) para que os Visitors tenham acesso a eles.

ENADE 2011 (CC)

Um Padrão de Projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizáveis.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. **Padrões de Projeto-Soluções Reutilizáveis de Software Orientado a Objetos**. Porto Alegre: Bookman, 2000.

Em relação a Padrões de Projeto, analise as afirmações a seguir.

I. *Prototype* é um tipo de padrão estrutural.

II. *Singleton* tem por objetivos garantir que uma classe tenha ao menos uma instância e fornecer um ponto global de acesso para ela.

III. *Template Method* tem por objetivo definir o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses.

IV. *Iterator* fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

É correto apenas o que se afirma em

A) I.
B) II.
C) I e IV.
D) II e III.
E) III e IV.

ENADE 2011 (CC)

Um Padrão de Projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizáveis.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. **Padrões de Projeto-Soluções Reutilizáveis de Software Orientado a Objetos**. Porto Alegre: Bookman, 2000.

Em relação a Padrões de Projeto, analise as afirmações a seguir.

I. *Prototype* é um tipo de padrão estrutural.

II. *Singleton* tem por objetivos garantir que uma classe tenha ao menos uma instância e fornecer um ponto global de acesso para ela.

III. *Template Method* tem por objetivo definir o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses.

IV. *Iterator* fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

É correto apenas o que se afirma em

A) I.
B) II.
C) I e IV.
D) II e III.
X) III e IV.

ENADE 2011 (SI)

QUESTÃO 20

Padrões de criação (*creational patterns*) abstraem a forma como objetos são criados, tornando o sistema independente de como os objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto. Há dois temas recorrentes nesses padrões. Primeiro, todos encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema. Segundo, ocultam o modo como essas classes são criadas e montadas. Tudo que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas. Os padrões de criação são classificados em *Abstract Factory, Builder, Factory Method, Prototype* e *Singleton*.

GAMMA, E. *et al.* **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading, MA: Addison-Wesley, 1994.(adaptado)

O padrão *Abstract Factory* é usado quando

A o sistema deve ser independente da maneira como seus produtos são criados, relacionados e representados.

B o algoritmo de criação de um objeto deve ser independente das suas partes e da maneira como ele é montado.

C houver uma única instância de uma classe e esta for acessada a partir de um ponto de acesso conhecido.

D classes delegam responsabilidade a alguma das subclasses, e deseja-se localizar qual é a subclasse acessada.

E as classes utilizadas para instanciação são especificadas em tempo de execução ou carregadas dinamicamente.

ENADE 2011 (SI)

QUESTÃO 20

Padrões de criação (*creational patterns*) abstraem a forma como objetos são criados, tornando o sistema independente de como os objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto. Há dois temas recorrentes nesses padrões. Primeiro, todos encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema. Segundo, ocultam o modo como essas classes são criadas e montadas. Tudo que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas. Os padrões de criação são classificados em *Abstract Factory, Builder, Factory Method, Prototype* e *Singleton*.

GAMMA, E. *et al.* **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading, MA: Addison-Wesley, 1994.(adaptado)

O padrão *Abstract Factory* é usado quando

A o sistema deve ser independente da maneira como seus produtos são criados, relacionados e representados.

B o algoritmo de criação de um objeto deve ser independente das suas partes e da maneira como ele é montado.

C houver uma única instância de uma classe e esta for acessada a partir de um ponto de acesso conhecido.

D classes delegam responsabilidade a alguma das subclasses, e deseja-se localizar qual é a subclasse acessada.

E as classes utilizadas para instanciação são especificadas em tempo de execução ou carregadas dinamicamente.