



Análise e Projeto OO

Fernando Pedrosa – fpedrosa@gmail.com

Bibliografia

- ▶ **Sommerville, Ian.** Software Engineering. **Editora:** Addison Wesley.
- ▶ **Pressman, Roger S.** Software Engineering: A Practitioner's Approach. **Editora:** McGraw–Hill.
- ▶ **RUP** – www.wthreex.com/rup

Análise OO

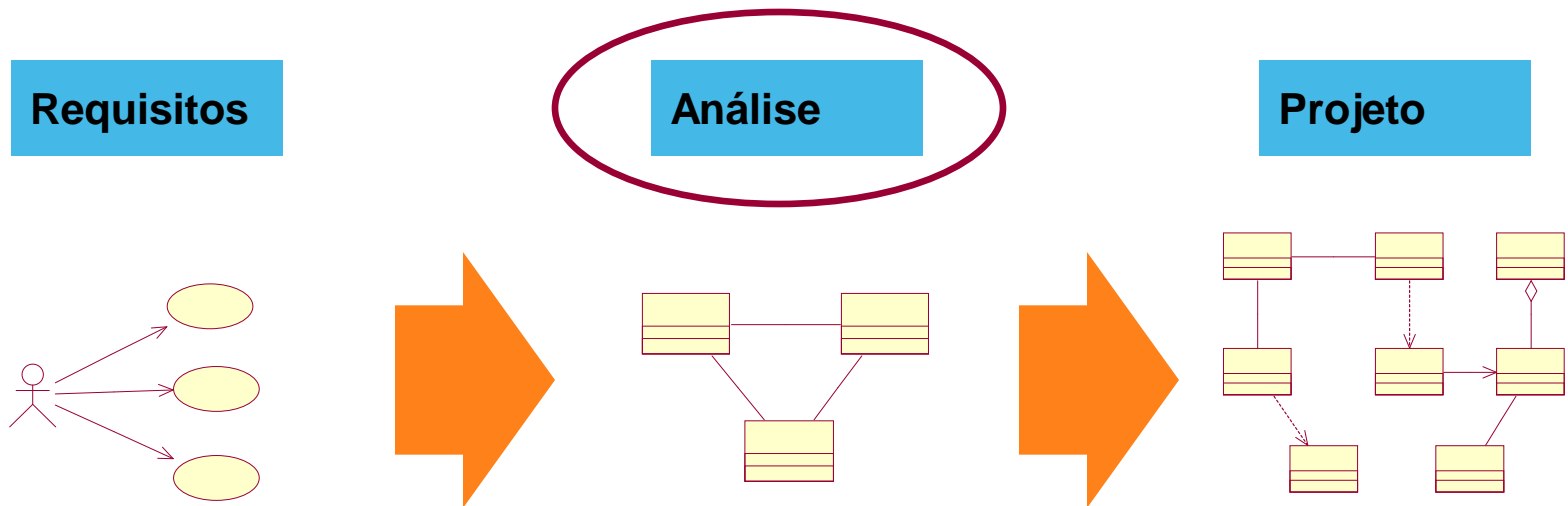
Análise Orientada a Objetos

- ▶ Após a etapa de requisitos, temos os documentos de requisitos e os casos de uso em mãos
- ▶ Queremos agora gerar um primeiro modelo do sistema a partir dos casos de uso
- ▶ Este modelo é chamado de **modelo de análise**
- ▶ No RUP, toda a atividade de análise é guiada por Casos de Uso

Objetivos Específicos

- ▶ Entender o problema a ser tratado antes de partir para a solução
- ▶ Encontrar os elementos que vão compor o software, suas funções, dados e relacionamentos
- ▶ Mas, nesta etapa, a tecnologia de implementação e RNFs são ignorados
- ▶ Segundo o RUP, é atividade **opcional**

Contexto



Casos de Uso x Análise OO

casos de uso

Descritos na linguagem do cliente

Visão externa do sistema

Captura as funcionalidades do sistema

Estruturado por casos de uso

análise

Descrito na linguagem dos desenvolvedores

Visão interna do sistema

Mostra como as funcionalidades podem ser realizadas

Estruturado por classes e pacotes

Mas o que são classes?

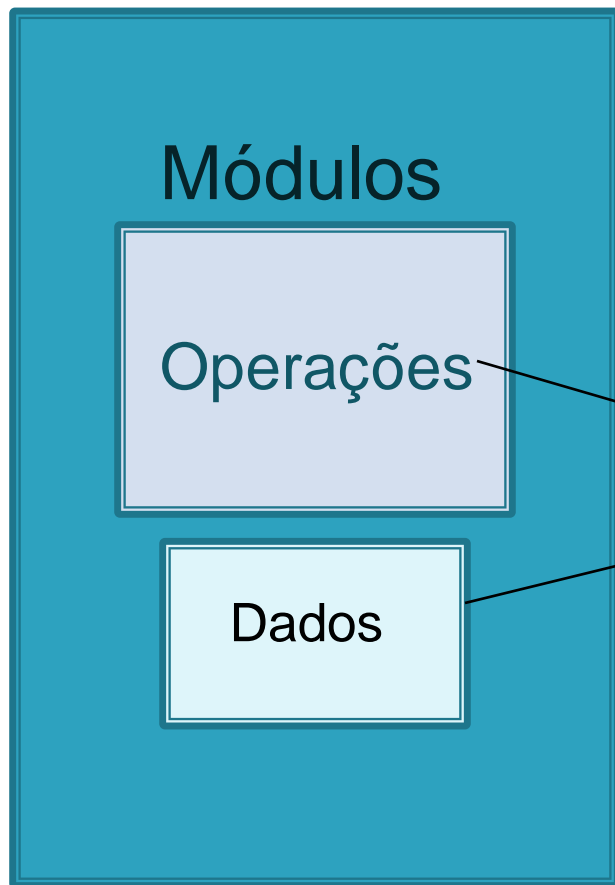
- ▶ Para entendermos Análise OO, precisamos estudar o Paradigma OO
- ▶ Vamos estudar
 - Objetos
 - Classes
 - Comportamentos
 - Atributos
 - Relacionamentos
 - ...

Paradigma OO

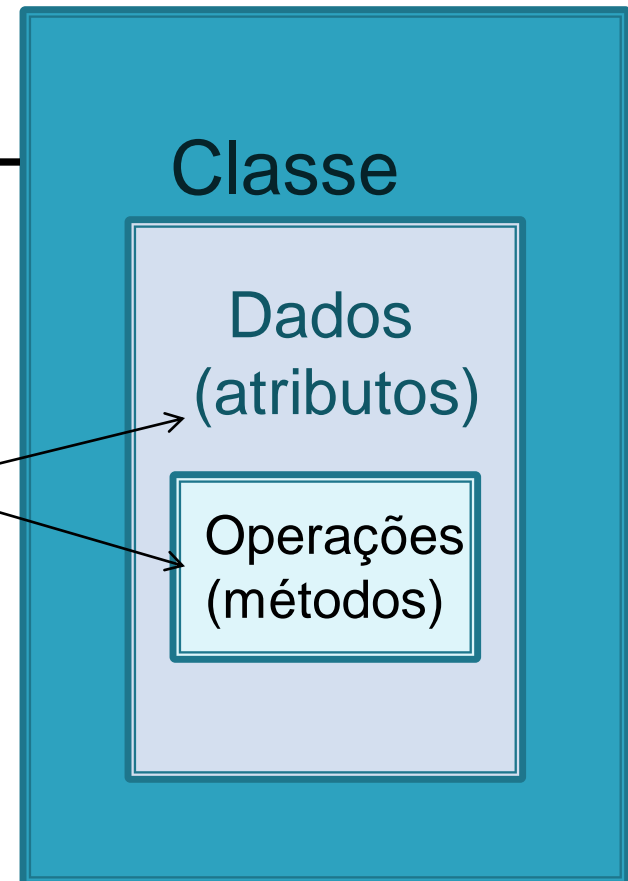
Paradigma OO

- ▶ Enfoque tradicional: compreensão do sistema como um conjunto de programas que executam processos sobre dados
- ▶ Enfoque OO: o sistema é uma coletânea de **objetos** que interagem entre si, com características próprias representadas por dados (atributos) e operações (métodos)

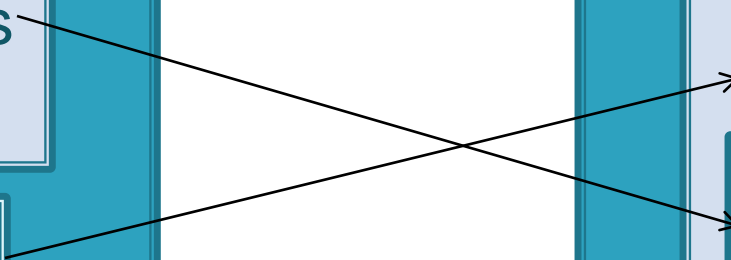
Foco na Estrutura



Foco no Objeto



Horizontal line connecting the two main structures.



Objetos

- ▶ No Paradigma OO, a idéia é olhar o mundo real como se tudo pudesse ser representado por objetos
- ▶ Um objeto é a representação de qualquer coisa que você queira modelar em um programa
- ▶ Vantagens
 - Facilidade de manutenção
 - Extensibilidade
 - Maior reuso

Componentes de um Objeto

▶ Identidade

- Propriedade do objeto que o distingue de todos os outros objetos

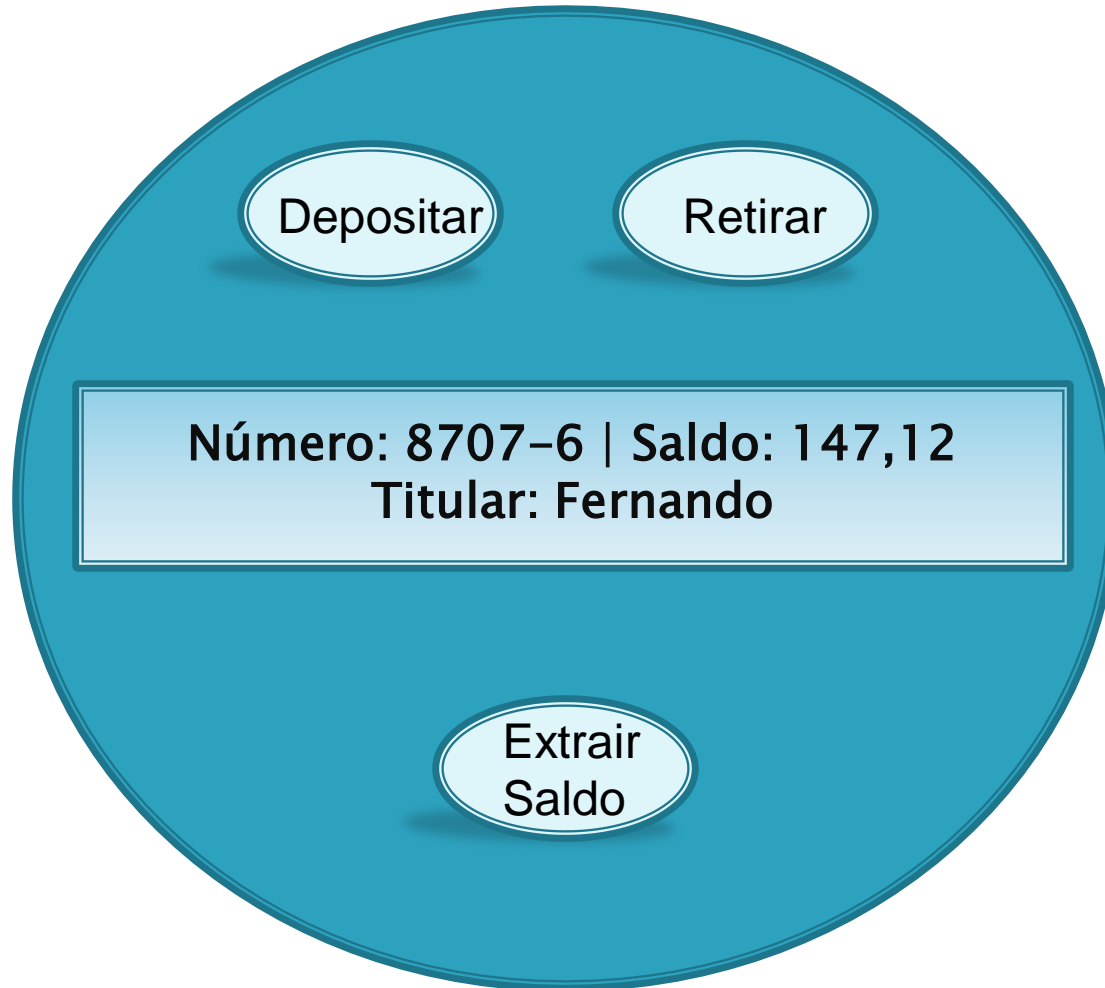
▶ Estado

- Reflete os valores correntes dos atributos do objeto em determinado momento

▶ Comportamento

- Como o objeto reage em termos de mudança de estado e troca de mensagens
- Conjunto de atividades externamente observáveis do objeto

Objeto Conta Bancária



Classes

- ▶ Classes podem ser pensadas como *templates* ou “moldes” para objetos
- ▶ Consistem de um conjunto de objetos do mesmo tipo, com as mesmas características (operações e propriedades)
- ▶ Objetos são **instâncias** de classes

Classe versus Objetos

Objeto:
Conta do Fernando

Número: 8707
Saldo: 147.42
Titular: Fernando

Objeto:
Conta da Eliane

Número:
123456
Saldo: 770.77
Titular: Eliane

Classe
Conta

Objeto:
Conta do Ricardo

Número: 654321
Saldo: 10000.00
Titular: Ricardo

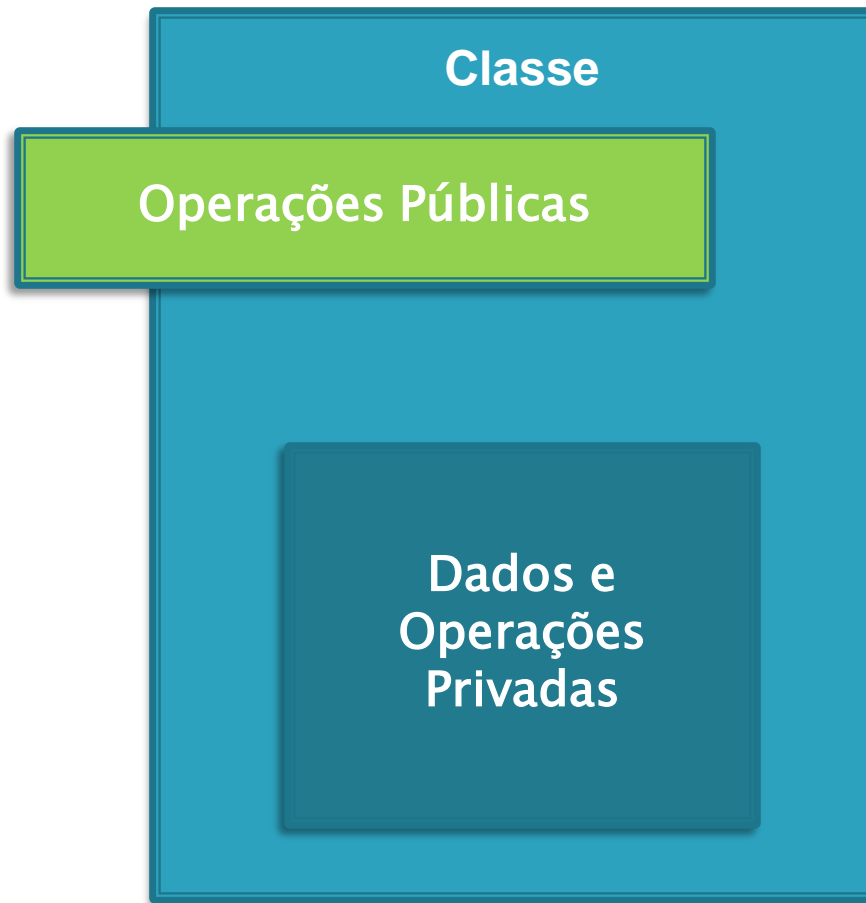
Componentes de uma Classe

- ▶ Propriedades (atributos)
 - Características pertencentes a todos os objetos da classe
 - Armazenam a informação sobre o estado dos objetos
- ▶ Operações (métodos)
 - Funções ou serviços oferecidos pela classe
 - Métodos são usados para implementar o comportamento dos objetos

Encapsulamento

- ▶ É o mecanismo da OO para esconder os detalhes internos de implementação dos objetos do mundo externo
- ▶ Principais benefícios
 - Redução dos impactos propagados a partir de mudanças
 - Diminuição do acoplamento (dependência) entre classes e objetos
 - Simplificação das interfaces entre os objetos

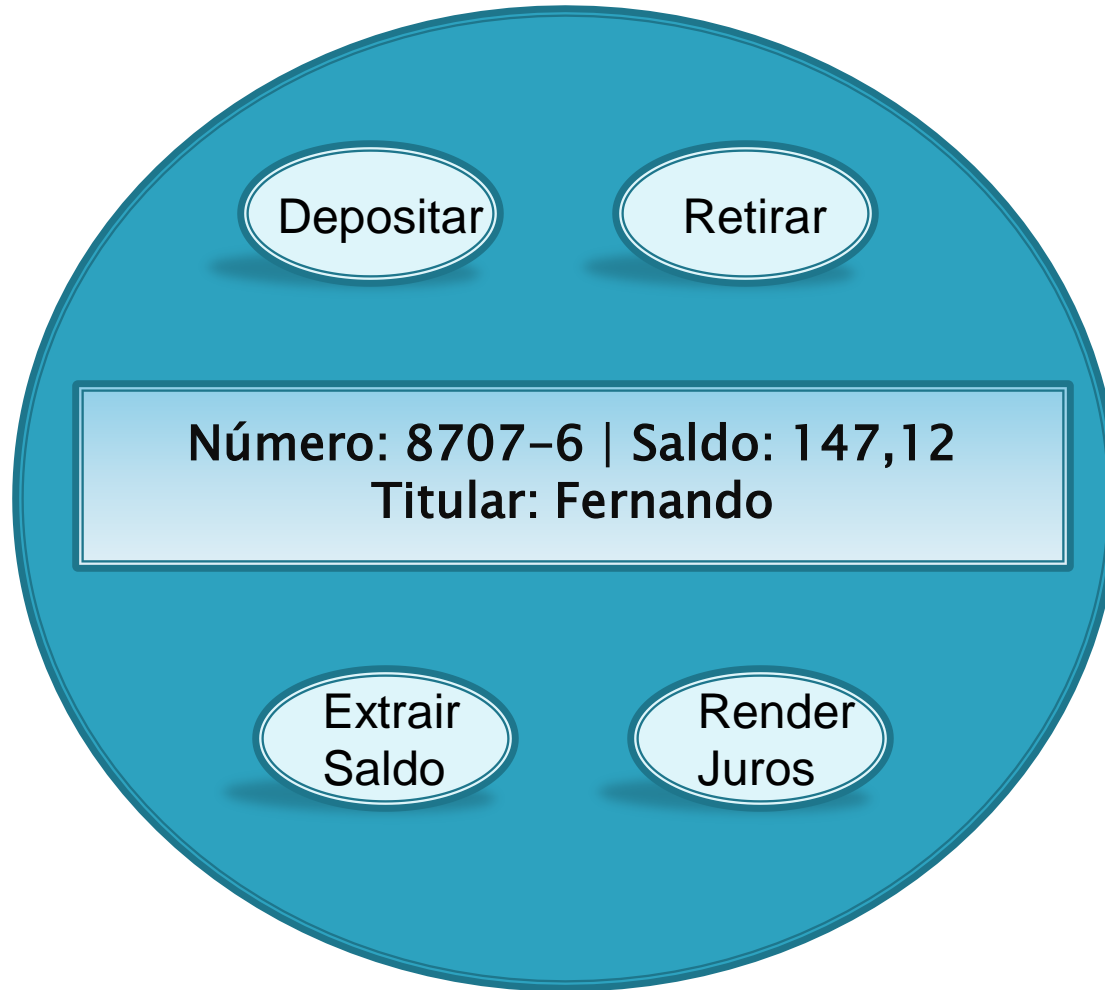
Encapsulamento



Herança

- ▶ É o mecanismo da OO que permite criar novas classes a partir de classes já existentes, reutilizando seus atributos e comportamentos
- ▶ Benefícios
 - Reuso
 - Extensibilidade
- ▶ Ex: Conta Corrente, Poupança, Aplicação...

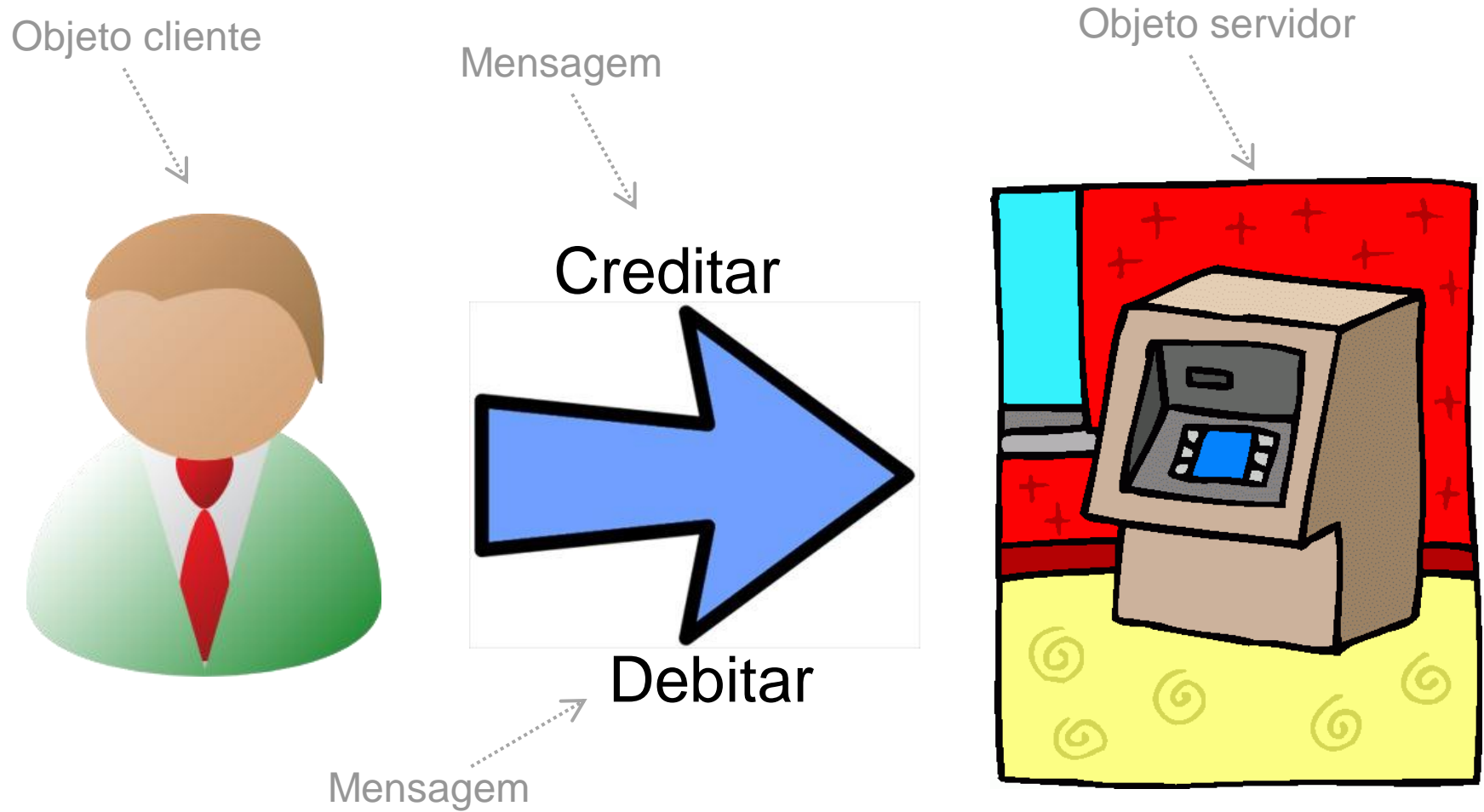
Objeto Poupança



Mensagens

- ▶ Objetos se comunicam através de mensagens
- ▶ Uma mensagem é uma operação que um objeto realiza em outro
 - Na prática, significa um objeto invocando um método de outro
- ▶ Objetos devem se comunicar **apenas** através de mensagens (boa prática)

Mensagens



Polimorfismo

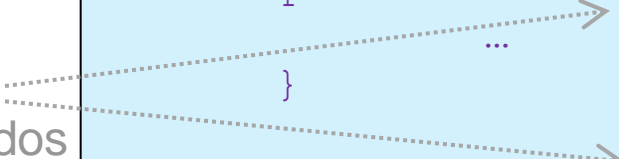
- ▶ “Várias formas”
- ▶ Denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem
- ▶ Dois tipos
 - Polimorfismo Estático ou Sobrecarga
 - Polimorfismo Dinâmico ou Sobreposição

Polimorfismo Estático

- ▶ Sobrecarga (overload)
- ▶ A mesma operação implementada várias vezes na mesma classe
- ▶ A escolha depende da assinatura dos métodos sobrecarregados

Métodos
sobrecarregados

```
public class Graphic{  
    ...  
    public void draw (int x, int y) {  
        ...  
    }  
    public void draw (int x, int y, int z) {  
        ...  
    }  
}
```



Polimorfismo Dinâmico

- ▶ Sobreposição ou Sobrescrita (override)
- ▶ Acontece na herança, quando a subclasse sobrepõe o método original.
- ▶ O método é escolhido em tempo de execução e não em tempo de compilação (**Ligação Dinâmica**)
- ▶ A escolha depende do tipo do objeto que recebe a mensagem

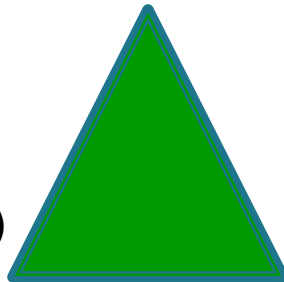
Polimorfismo Dinâmico

Forma calcularArea()



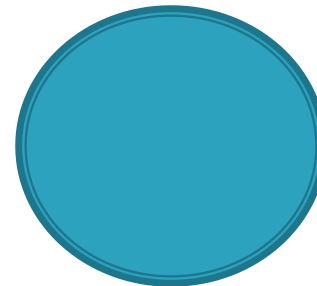
Triângulo

Base
Altura
calcularArea()



Círculo

Raio
calcularArea()



Polimorfismo Dinâmico

```
public abstract class Forma
{
    private double area;

    public abstract double
        calcularArea();
}
```

```
public class Triangulo
    extends Forma
{
    private int altura;
    private int base;
    public double calcularArea()
    {
        return (base*altura)/2
    }
}
```

```
public class Circulo
    extends Forma
{
    double raio;
    public double calcularArea()
    {
        return 3.14*raio*raio;
    }
}
```


Modelo de Análise (Domínio)

O que é?

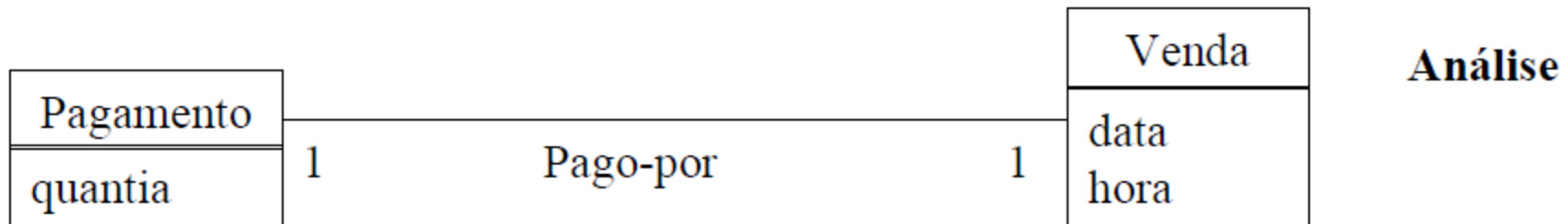
- ▶ É um modelo de objetos que descreve a realização dos casos de uso e que funciona como uma abstração do Modelo de Design
- ▶ Contém as classes de análise (Modelo de Classes) e qualquer artefato associado
- ▶ O Modelo de Classes evolui durante as iterações do projeto, incrementando novos detalhes às classes

Modelo de Classes: Evolução

- ▶ Há três níveis sucessivos de detalhamento
- ▶ **Análise**
 - Modelo de Classes de Análise (Domínio)
- ▶ **Especificação (Projeto)**
 - Modelo de Classes de Especificação
- ▶ **Implementação**
 - Modelo de Classes de Implementação

Modelo de Classes de Análise

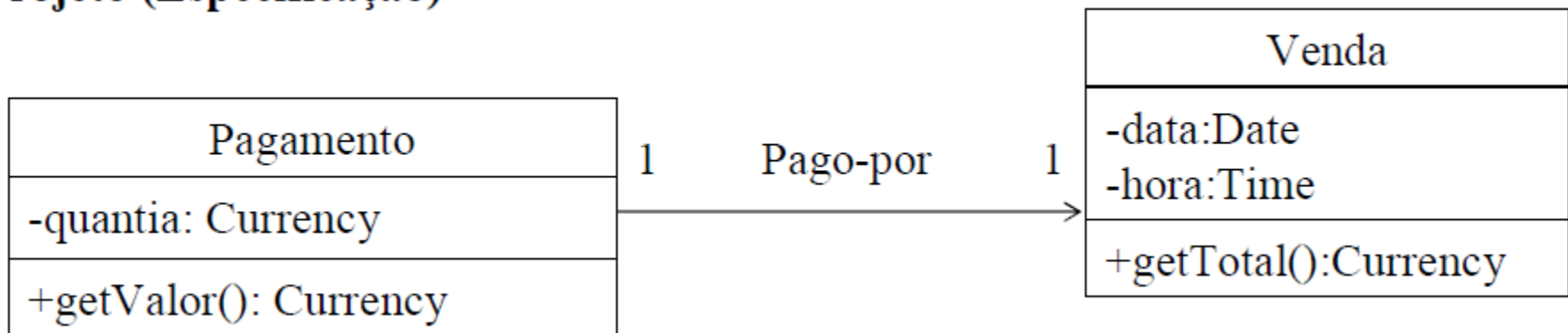
- ▶ Representa as classes no domínio do negócio em questão
- ▶ Não leva em consideração restrições inerentes à tecnologia específica a ser utilizada na solução de um problema



Modelo de Classes de Especificação

- ▶ Obtido através da adição de detalhes ao modelo anterior, conforme a solução de software escolhida

Projeto (Especificação)



Modelo de Classes de Implementação

- Implementação das classes em alguma linguagem de programação

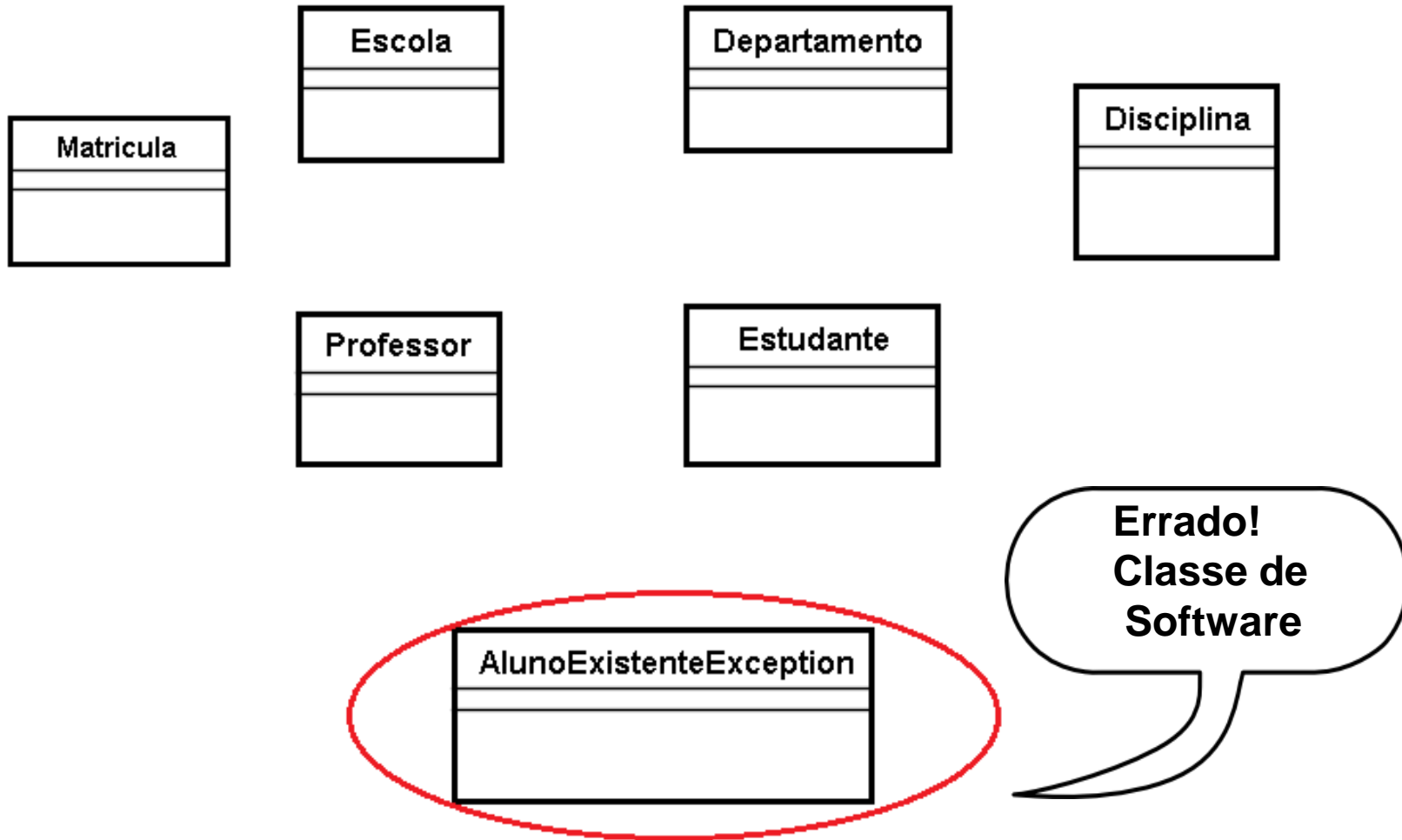
```
public class Pagamento {  
    private Currency quantia;  
    ...  
    public Currency getValor() {  
        return quantia;  
    }  
}  
  
    public class Venda {  
        private Date data;  
        private Time hora;  
        private Pagamento pagamento;  
        ...  
        public Currency getTotal() {  
            return pagamento.getValor();  
        }  
    }
```

Atividades da Análise OO

Passos da Atividade de Análise

- ▶ Passo 1: Identificar as classes
 - Desenhar diagramas de classes conceituais
 - Identificar persistência
- ▶ Passo 2: Identificar responsabilidades
- ▶ Passo 3: Identificar atributos
- ▶ Passo 4: Identificar relacionamentos

Exemplo: Sistema Escola

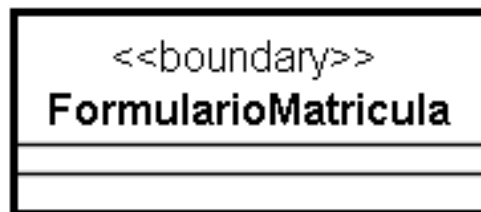


Passo 1: Identificando classes

- ▶ Para cada caso de uso, identificamos três tipos de classes
 - Fronteira
 - Controle
 - Entidade
- ▶ As classes devem ser **conceituais**
 - Apenas idéias abstratas do mundo real
- ▶ Vamos identificar as classes de análise para o Caso de Uso “Matricular Aluno”

Classes de Fronteira

- ▶ Utilizadas para modelar a interação entre um ator e o sistema
- ▶ Para cada ator, é identificada pelo menos uma classe de fronteira para permitir sua interação com o sistema
- ▶ Dependem do **ambiente** (visão)
- ▶ Representação:



ou

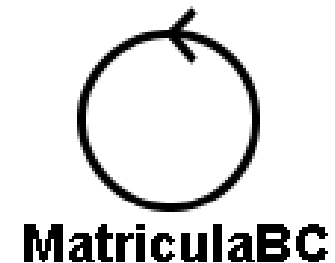


Classes de Controle

- ▶ Objetos responsáveis por controlar a lógica de execução correspondente a cada caso de uso
- ▶ Geralmente são do tipo
 - **Controlador**: intermediam objetos de classe de fronteira com o objeto de controle Cadastro
- ▶ **Representação**:

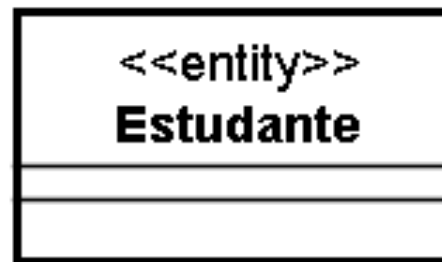


ou



Classes de Entidade

- ▶ Representam a informação que é manipulada ou processada pelo caso de uso
- ▶ Vêm do domínio do negócio
- ▶ Normalmente armazenam informações persistentes
- ▶ Representação:

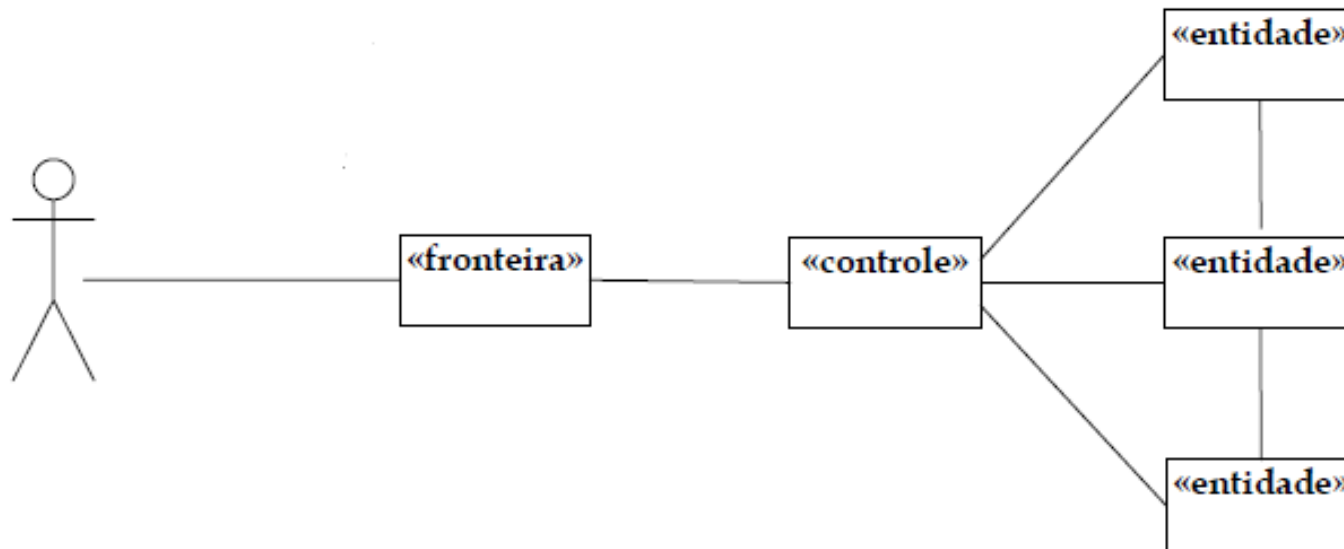


ou



Fronteira x Controle x Entidade

- ▶ Em resumo, as classes podem ser modeladas assim:



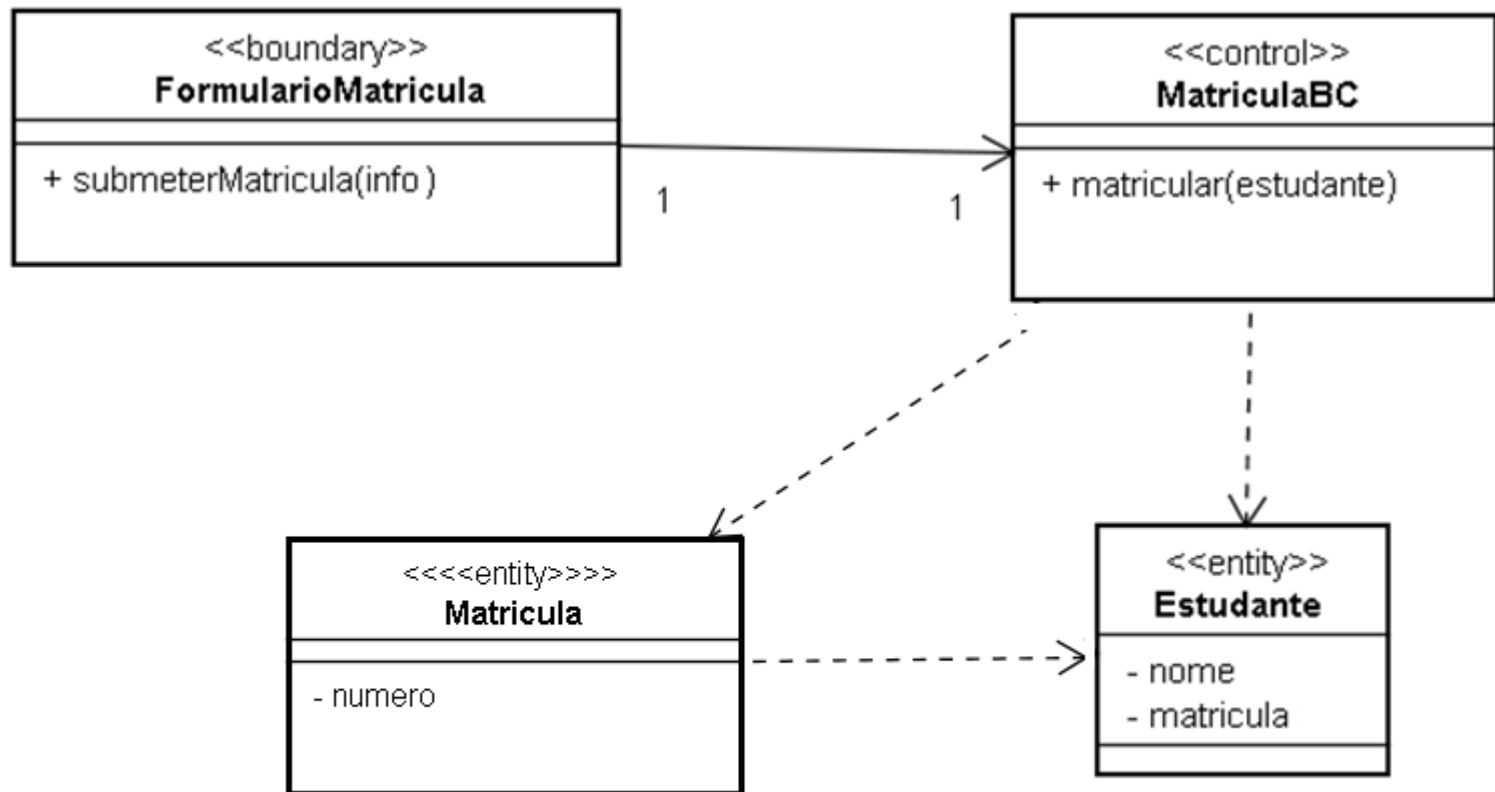
- ▶ Lembra algo? **MVC!**

Completando o Diagrama de Classes

- ▶ Passo 2: Identificar responsabilidades
 - Descobrir quais operações serão fornecidas pelas classes
 - Diagramas de interação (sequência, interação, etc.) podem ser úteis aqui
- ▶ Passo 3: Identificar atributos
 - Nesta etapa, ainda não é necessário identificar o tipo dos atributos
- ▶ Passo 4: Identificar relacionamentos
 - Associações, dependências, etc.

Diagrama de Classes: “Matricular Aluno”

Para cada Caso de Uso levantado na etapa de Requisitos, deve-se identificar as classes de Fronteira, Controle e Entidade e organizá-las em diagramas de classes, que vão compor o Modelo de Análise



Exercícios [1]

(TSE CESPE 2006) [54] Acerca da análise e do projeto orientados a objetos assinale a opção correta.

- A) Um modelo de análise é menos abstrato que um de projeto e as classes em um modelo de análise não podem ser conceituais. As classes na análise podem modelar objetos persistentes, mas não transientes.
- B) Uma importante responsabilidade da análise é definir a arquitetura do sistema, dividindo-o em subsistemas. Um subsistema expõe serviços via interfaces, que devem ser especificadas na análise.
- C) Uma classe descreve objetos com as mesmas responsabilidades, relacionamentos, operações, atributos e semântica. As instâncias de uma classe têm, portanto, os mesmos valores para os seus atributos.
- D) Um modelo de análise pode realizar casos de uso. A realização de um caso de uso descreve interações entre objetos. Na UML, essas realizações podem ser documentadas via diagramas de colaboração.

Exercícios [1]

(BNDES – CESGRANRIO 2009)

[61–I] O conceito de herança possibilita a especialização de comportamentos pré-existentes em classes ancestrais.

[61–III] Uma das desvantagens da herança é a criação de dependência entre as classes envolvidas.

[61–IV] De acordo com a ideia do encapsulamento, é desejável, do ponto de vista de um objeto, que seus atributos internos estejam protegidos contra modificações diretas e que o acesso seja realizado por meio de métodos específicos (setters e getters).

[61–V] Polimorfismo está relacionado à vinculação dinâmica de mensagens e sobrescrita de métodos, sendo que o método correto a ser chamado só será definido em tempo de execução e dependerá do tipo da instância do objeto referenciado pela mensagem.

Exercícios [1]

(PETROBRAS – CESPE 2007)

[100] Em um modelo de análise, as classes de fronteira modelam interações entre o sistema e os atores. Cada classe de fronteira deve estar relacionada a um ou mais atores. Pode-se também ter classes de entidade, as quais tipicamente modelam dados persistentes.

(ANATEL – CESPE 2006)

[96] Uma classe na análise orientada a objeto representa uma abstração que pode ser mapeada para mais de uma classe no projeto. As classes na análise podem ser fronteiras, controladoras ou entidades. Uma fronteira modela interações entre o sistema e atores, uma entidade modela apenas objetos persistentes e uma controladora só pode controlar interações entre instâncias de uma mesma classe.

Projeto OO

Projeto Orientado a Objetos

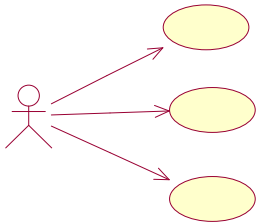
- ▶ Após a etapa de análise, temos o primeiro modelo do sistema
 - Definimos “o que” o software deve fazer
- ▶ Queremos agora detalhar este modelo, para gerarmos facilmente a implementação do sistema
 - Definimos “COMO” o software atenderá os requisitos analisados
- ▶ Este modelo é chamado de Modelo de Projeto

Análise x Projeto

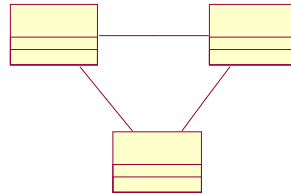
- ▶ Abstrato x Concreto
- ▶ Independente x Dependente da tecnologia de implementação
- ▶ Simples x Detalhado
- ▶ Modelos por caso de uso x Unificação em um único modelo

Projeto Orientado a Objetos

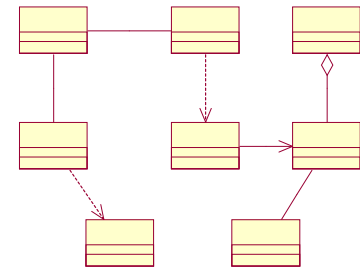
Requisitos



Análise



Projeto



Principais atividades

- ▶ Refinar o modelo de classes
 - Identificar relacionamentos de herança, classes abstratas e interfaces
 - Elaborar um diagrama de classes unificado
- ▶ Projetar Arquitetura
 - Divisão em camadas
- ▶ Projetar detalhadamente a estrutura e o comportamento interno de cada subsistema (módulos)

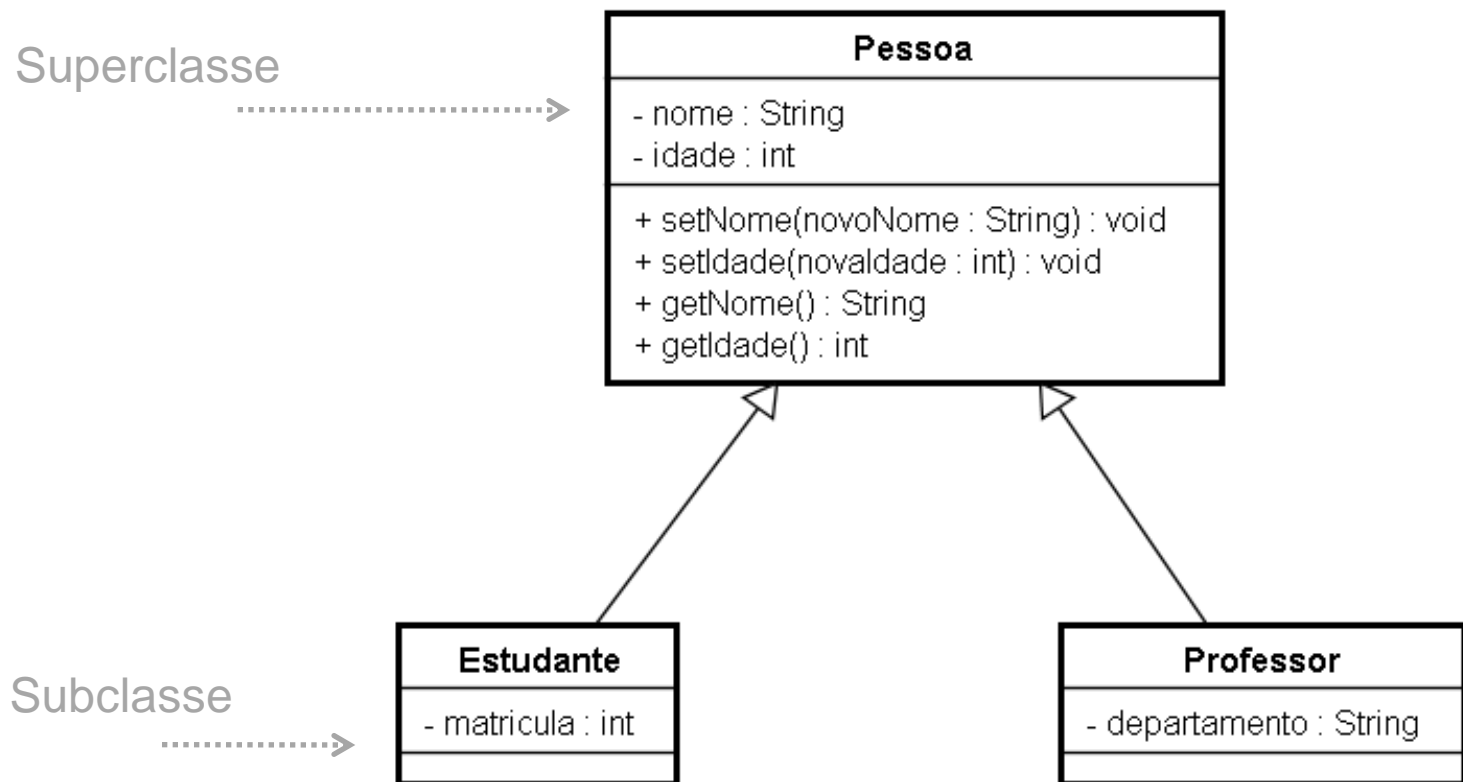
Refinando o Modelo de Classes

Generalização (Herança)

- ▶ “Uma generalização é um relacionamento entre um elemento mais geral e um elemento mais específico”
- ▶ Na modelagem de classes de projeto, são considerados aspectos relacionados ao relacionamento de herança
 - Tipos de herança (Simples x Múltipla)
 - Classes abstratas
 - Interfaces

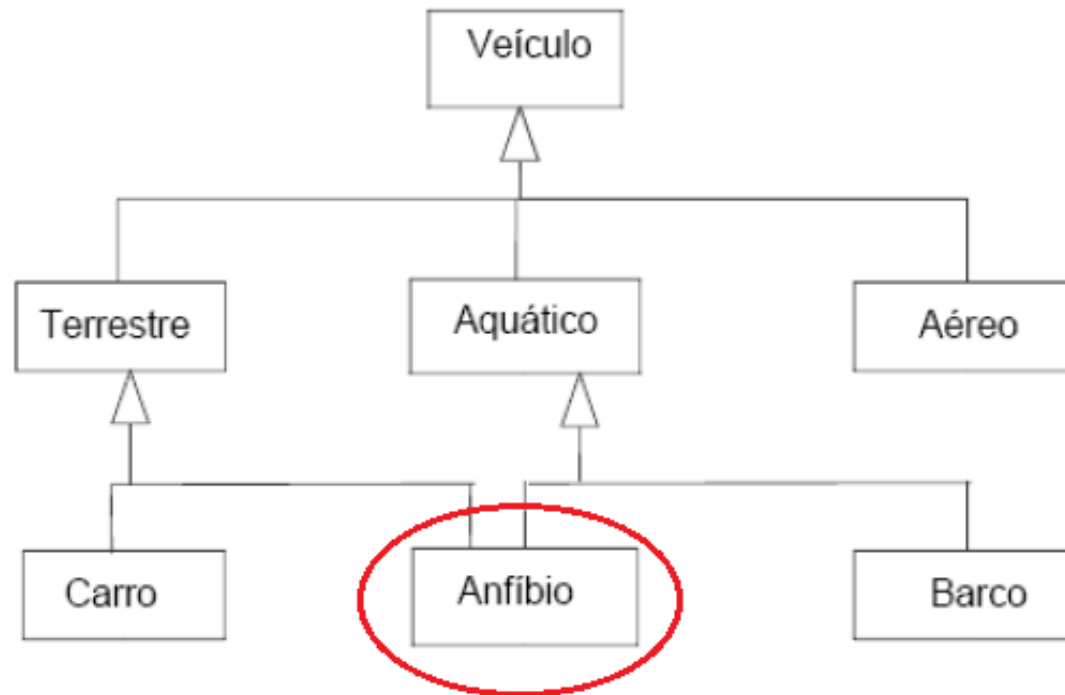
Herança Simples

- ▶ Permite criar novas classes a partir de classes existentes



Herança Múltipla

- ▶ Uma classe pode herdar de várias outras classes



Herança Múltipla

- ▶ A herança múltipla deve ser evitada
- ▶ Potenciais problemas:
 - Difícil de entender
 - Codificação confusa
 - Ambigüidade e Duplicidade de atributos
- ▶ Algumas linguagens não suportam herança múltipla (Java e Smalltalk)
 - C++ suporta!

Restrições da Generalização

▶ Incompleta

- Indica que outros subtipos podem ser adicionados no futuro. É o padrão.

▶ Completa

- Indica que todas as subclasses foram especificadas e que não é possível realizar mais sub-classificações.

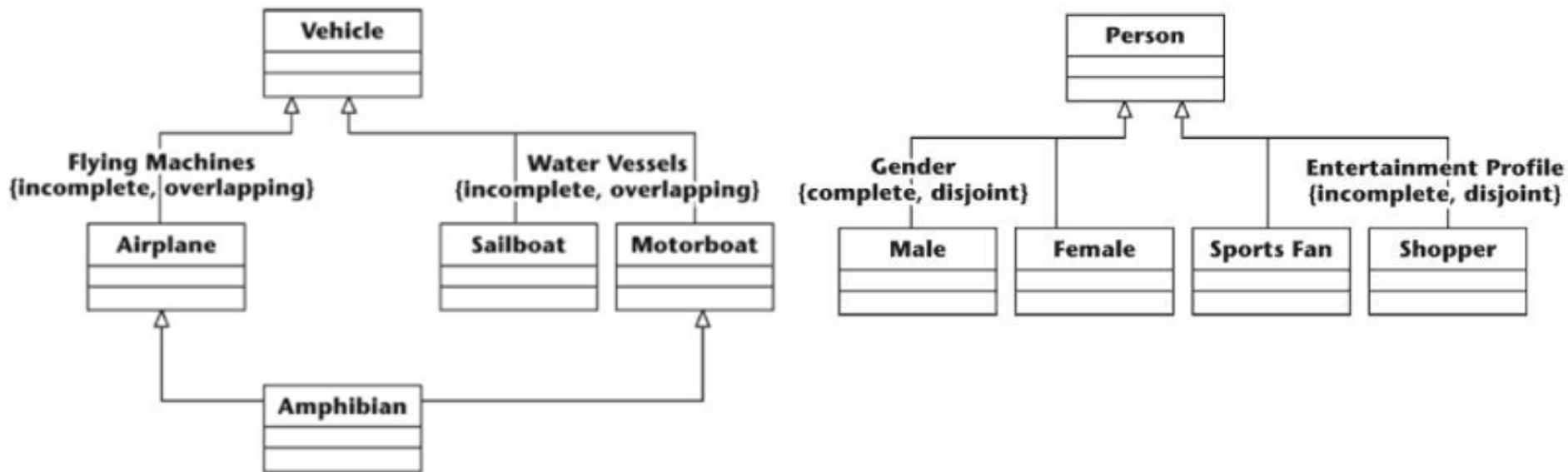
▶ Disjunta

- Significa que as classes não podem ser especializadas em uma subclasse comum, isto é, não é possível haver herança múltipla.

Restrições da Generalização

► Sobreposta (overlapping)

- É o contrário de disjoint, isto é, as subclasses podem herdar de mais de uma superclasse, ocorrendo herança múltipla.

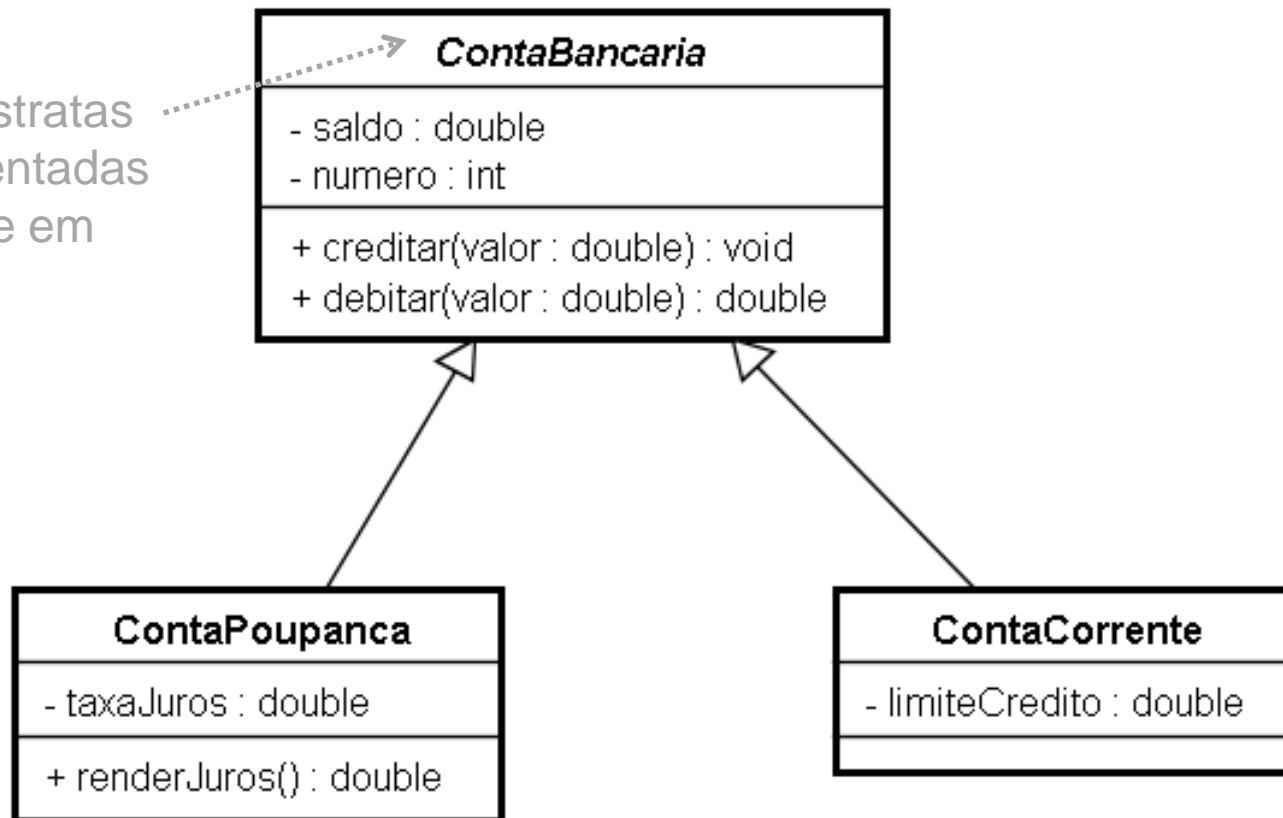


Classe Abstrata

- ▶ Representa um conceito abstrato e é utilizada para organizar uma hierarquia de generalização
- ▶ Permite que um conjunto de subclasses tenha o mesmo comportamento
- ▶ Não é projetada para gerar instâncias
- ▶ As classes dela derivadas representam implementações do conceito

Classe Abstrata

Classes abstratas
são representadas
com o nome em
itálico

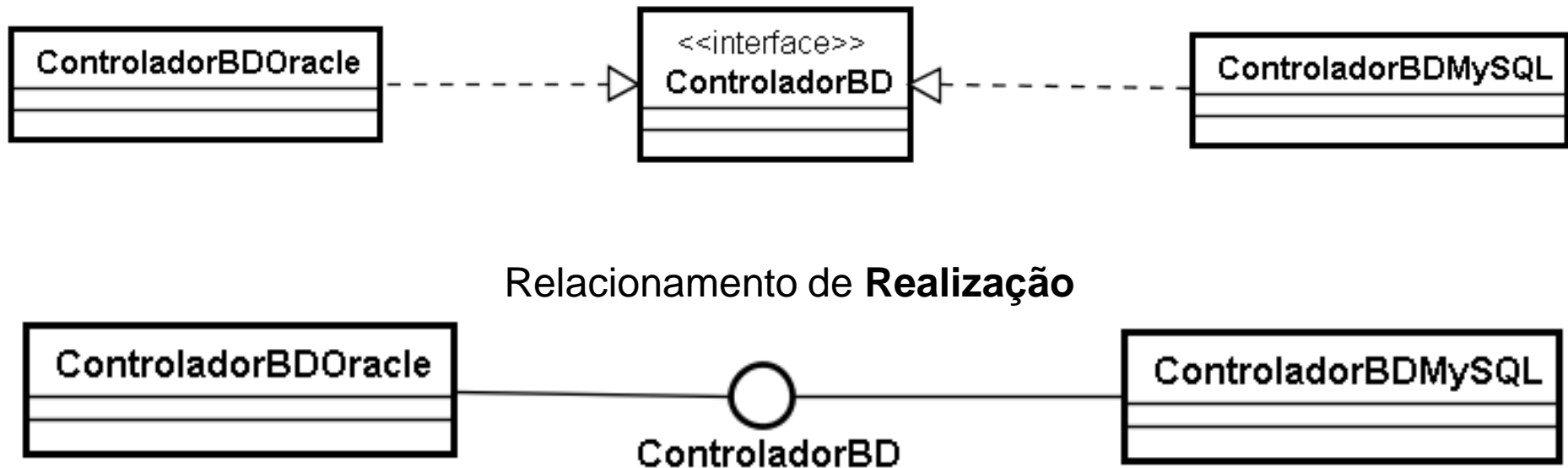


Interface

- ▶ Define um conjunto de comportamentos (operações) oferecido para uma classe ou componente
- ▶ Pode ser interpretada como um contrato de comportamento entre um objeto cliente e o fornecedor de serviços
- ▶ É dito que classes **realizam** interfaces

Interface

- ▶ Há duas notações para representar interfaces



Exercícios [2]

(PETROBRAS – CESPE 2007)

[125] Se uma classe criada por meio de herança tiver uma única classe-pai, o processo chama-se herança simples. Se tiver mais de uma classe-pai, o processo chama-se herança múltipla. Uma classe derivada pode acrescentar variáveis e métodos, possibilitando que certas operações sejam fornecidas apenas aos objetos da classe derivada.

(SAD/PE – CESPE 2010)

[47] Nas linguagens orientadas a objeto da atualidade, é comum o uso de herança múltipla, que permite a determinada classe herdar diretamente das implementações de uma ou mais classes, possibilitando mais expressividade semântica e facilitando a manipulação do sistema de tipos nessas linguagens.

Exercícios [2]

(TRE/MT – CESPE 2010)

[22-e] Classes abstratas não possuem atributos e se caracterizam por possuir métodos que podem ser criados dinamicamente quando essas classes são instanciadas.

(TJ/PA – FCC 2009)

[52] NÃO é uma das quatro restrições definidas pela UML, que podem ser aplicadas aos relacionamentos de generalização:

- (A) incomplete.
- (B) disjoint.
- (C) overlapping.
- (D) joint.
- (E) complete.

Projetando a Arquitetura do Sistema

Arquitetura do Sistema

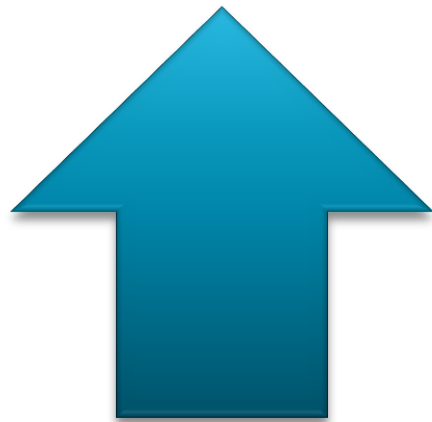
- ▶ Arquitetura de software é a organização ou a estrutura dos componentes significativos do sistema que interagem por meio de interfaces
- ▶ É composta de:
 - Componentes de software
 - Suas propriedades visíveis externamente
 - O relacionamento entre os componentes
- ▶ Forma a espinha dorsal para se construir softwares efetivos

Por que a Arquitetura é importante?

- ▶ **Comunicação entre os *stakeholders***
 - A arquitetura representa uma abstração que pode ser entendida por todas as partes interessadas
 - Serve como base para entendimento, comunicação, negociação e consenso
- ▶ **Decisões de projeto tempestivas**
 - A arquitetura representa o ponto mais precoce onde as decisões de projeto podem ser analisadas
- ▶ **Abstração “transferível” de um sistema**
 - É possível reutilizar a aplicação de uma arquitetura ao longo de vários sistemas diferentes, mas que exibam características semelhantes

O que é uma boa arquitetura?

- ▶ Uma boa arquitetura de software deve ter os seus componentes projetados com **baixo acoplamento e alta coesão**



Coesão



Acoplamento

Acoplamento

- ▶ É o grau de dependência de um determinado módulo do programa em relação a outros módulos
- ▶ O acoplamento forte entre classes significa que elas precisam conhecer detalhes internos umas das outras
- ▶ Quanto menos acoplamento (interconexões entre classes) melhor!

Desvantagens do forte acoplamento

- ▶ Mudanças em um módulo causam um efeito em cascata de mudanças em outros módulos
- ▶ A construção de um módulo se torna mais complicada devido à interdependência com outros módulos
- ▶ O reuso é prejudicado
- ▶ Os testes tornam-se mais difíceis de ser realizados

Exemplo

```
public class Pedido {  
    public String produto;  
    public int quantidade;  
    public double preço;  
    ...  
}
```

Código fortemente acoplado

```
public class Venda {  
    ...  
    public double calcularValor(Pedido pedido) {  
        double valor = pedido.preço * pedido.quantidade;  
    }  
}
```

Código fracamente acoplado

```
public class Venda {  
    ...  
    public double calcularValor(Pedido pedido) {  
        double valor = pedido.calcularTotal();  
    }  
}
```

Coesão

- ▶ É a medida do quão fortemente relacionadas são as responsabilidades de um módulo
- ▶ Queremos ter classes
 - Com a menor complexidade possível
 - Com responsabilidades claramente definidas
 - Que não executam um grande volume de trabalho
- ▶ Queremos ter a máxima coesão possível

Vantagens da alta Coesão

- ▶ Módulos de sistemas coesos são mais simples de se entender
- ▶ A manutenção do sistema torna-se mais fácil, pois as mudanças são isoladas apenas ao módulo que interessa
- ▶ A capacidade de reuso aumenta

Exemplo

Código pouco coeso

```
public class Programa {
```

```
    public void desenharTela() {  
        //implementação  
    }
```

Código de Apresentação

```
    public class reservarProduto() {  
        //implementação  
    }
```

Código de Negócio

```
    public class gravarNoBD() {  
        // implementação  
    }
```

Código de Acesso a Dados

Exercícios [3]

(SAD/PE – CESPE 2010)

[35–D] É desejável que o valor da coesão e o do acoplamento, duas importantes propriedades da arquitetura de um software, sejam maximizados durante a engenharia de software.

(FINEP – CESPE 2009)

[36–C] A característica de manutenção (manutenability) de um software é diretamente proporcional ao acoplamento apresentado por esse software e inversamente proporcional à coesão.

(BNDES – CESGRANRIO 2009)

[62–E] Os conceitos de coesão e acoplamento são complementares, ou seja, em um projeto de software, se o acoplamento for baixo, fatalmente sua coesão será alta e vice-versa.

Arquitetura em Camadas

- ▶ Uma forma de organizar a arquitetura é através de camadas de software
 - Cada camada provê um conjunto de funcionalidades em determinado nível de abstração
 - Tipicamente, uma camada de mais alta abstração depende de uma camada de mais baixa abstração, e não o contrário
 - Uma mudança em determinada camada, desde que seja mantida sua interface, não afeta as outras camadas

Arquitetura em Camadas

Vantagens

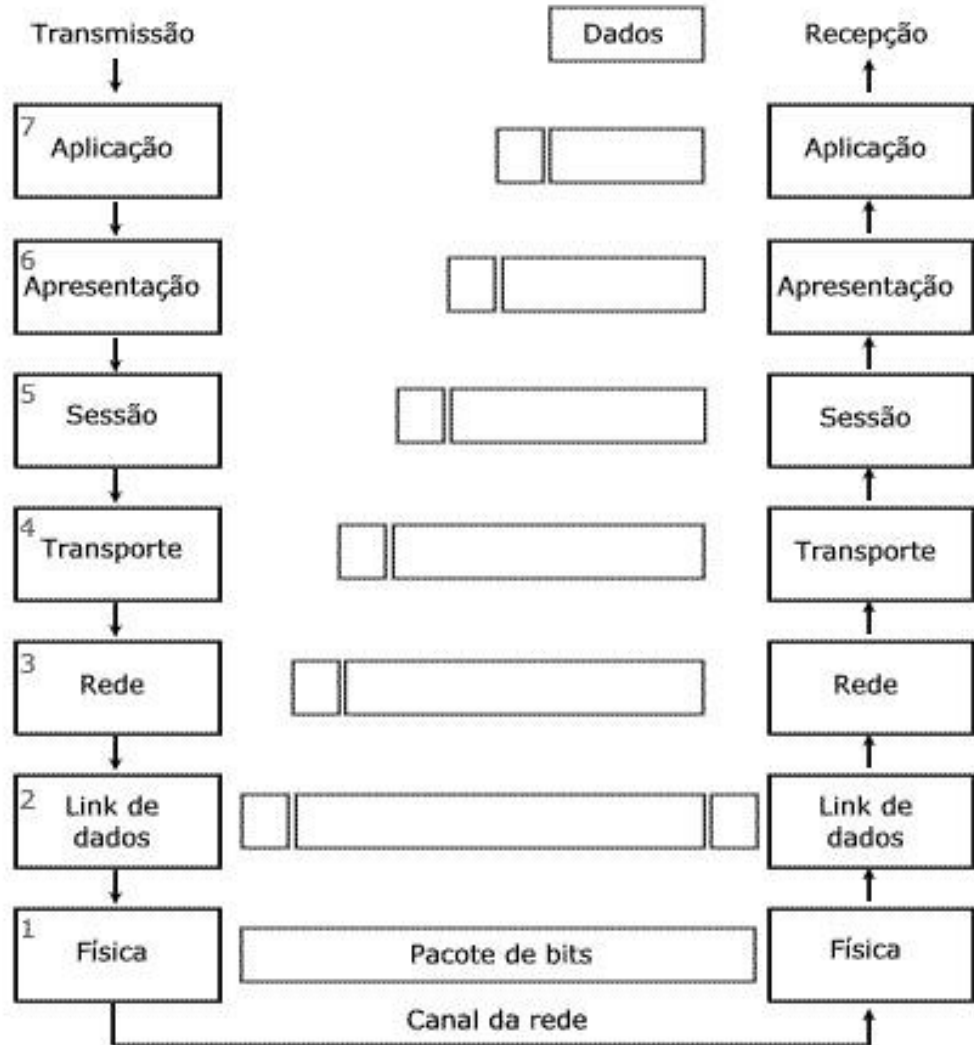
- ▶ Separação de responsabilidades
- ▶ Decomposição de complexidade
- ▶ Encapsulamento de implementação
- ▶ Maior reuso e extensibilidade

Desvantagens

- ▶ Podem penalizar a performance do sistema
- ▶ Aumento do esforço e complexidade de desenvolvimento

Arquitetura em Camadas

Exemplo: Modelo OSI



Evolução das arquiteturas em camadas

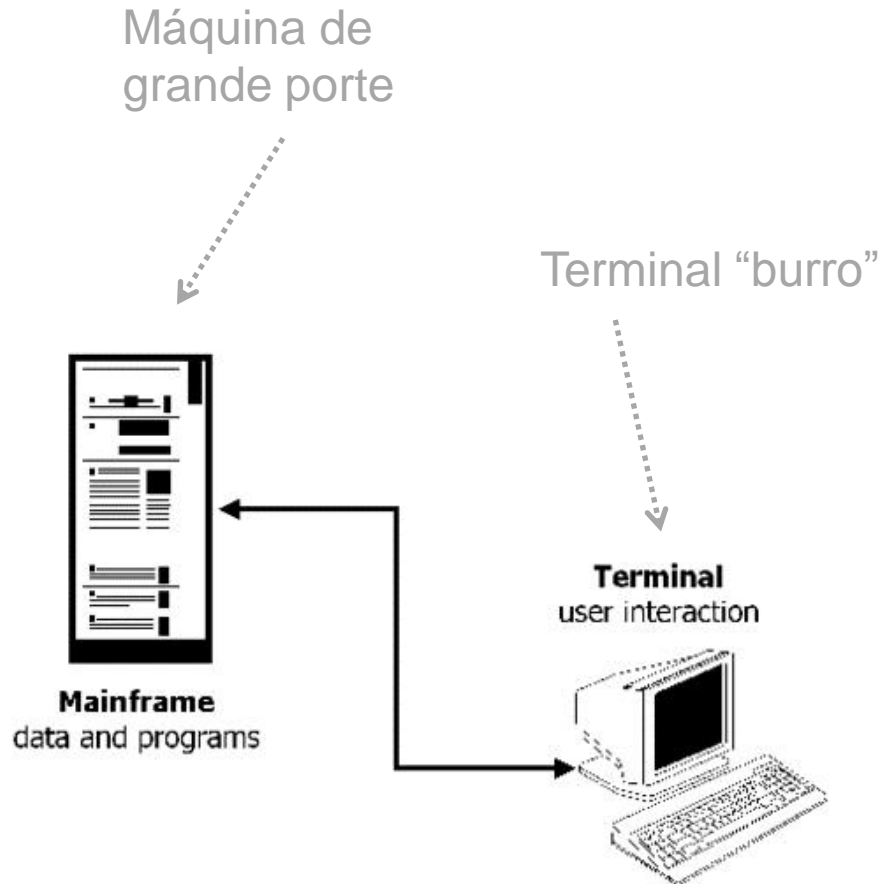
▶ Arquitetura Monolítica

- Programa e dados armazenados em uma única grande máquina – não havia camadas
- Acesso através de terminais “burros”

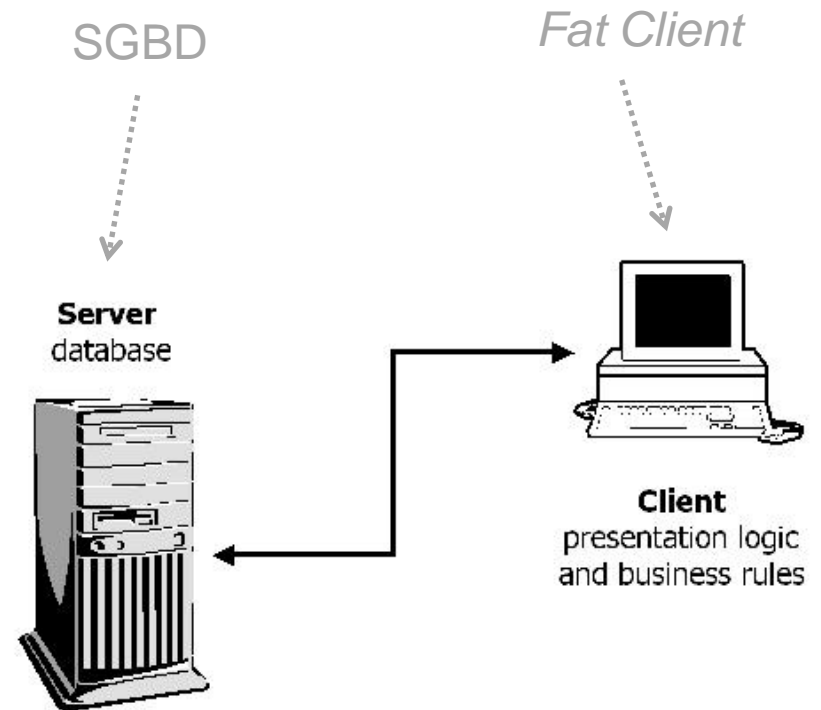
▶ Arquitetura em duas camadas (*two-tier*)

- Década de 80: surgem os PC's baratos
- Aplicação rodava na máquina cliente que interagia com um SGBD (servidor de dados)
- “*Fat client*” continha toda a lógica de apresentação, negócio e acesso a dados

Arquitetura Monolítica



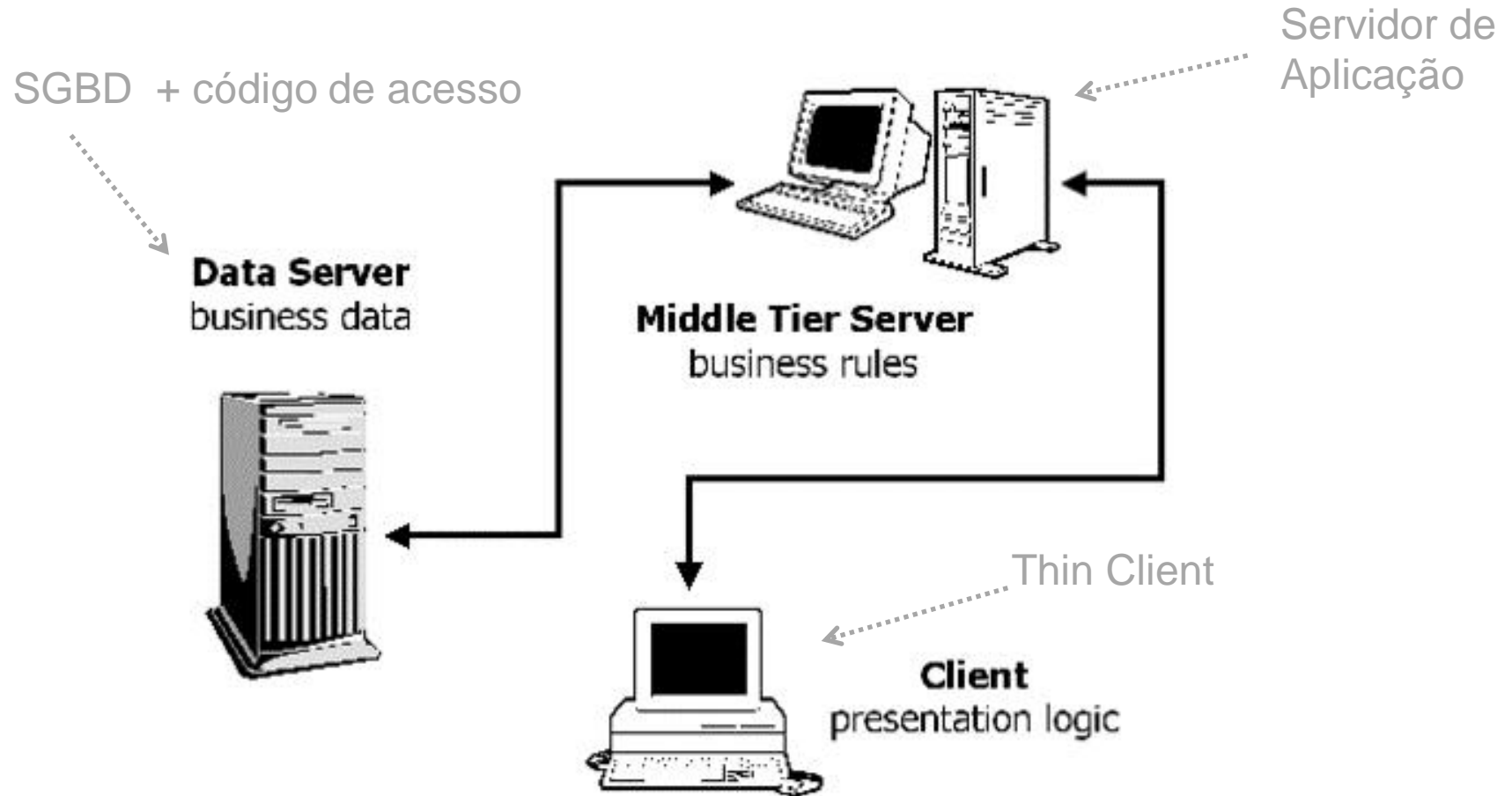
Two-tier Architecture



Arquitetura em três camadas

- ▶ Para minimizar o impacto de mudanças nas aplicações, decidiu-se separar a camada de negócio da camada de interface gráfica, gerando três camadas:
 - Camada de Apresentação
 - Camada da Lógica do Negócio
 - Camada de Acesso a Dados
- ▶ Arquitetura conhecida como *Three-tier Architecture*

Arquitetura em três camadas



Arquitetura em três camadas

Vantagens

- ▶ É mais fácil de modificar ou substituir qualquer camada sem afetar as outras
- ▶ Separar a lógica de aplicação da lógica de acesso a dados melhora o balanceamento de carga
- ▶ É mais fácil assegurar políticas de segurança na camada do servidor sem interferir nos clientes

Arquitetura em três camadas

Desvantagens

- ▶ Quanto mais camadas houver na arquitetura, maior é a tendência da performance diminuir
- ▶ O rastreamento de ponta-a-ponta em sistemas complexos com muitas camadas é uma tarefa complicada
- ▶ Requer um maior esforço de desenvolvimento

Camada de Apresentação

- ▶ Contém o código para a apresentação da aplicação (entrada e saída de dados)
 - As classes de fronteira se localizam aqui
- ▶ A camada de apresentação é altamente depende de ambiente
 - Páginas WEB (HTML, JavaScript, CSS, JSP, Applet, etc.)
 - Aplicações desktop (Windows/Linux Applications, etc.)
 - Menus baseados em texto (sistemas legados, aplicações móveis, etc.)

Camada da Lógica do Negócio

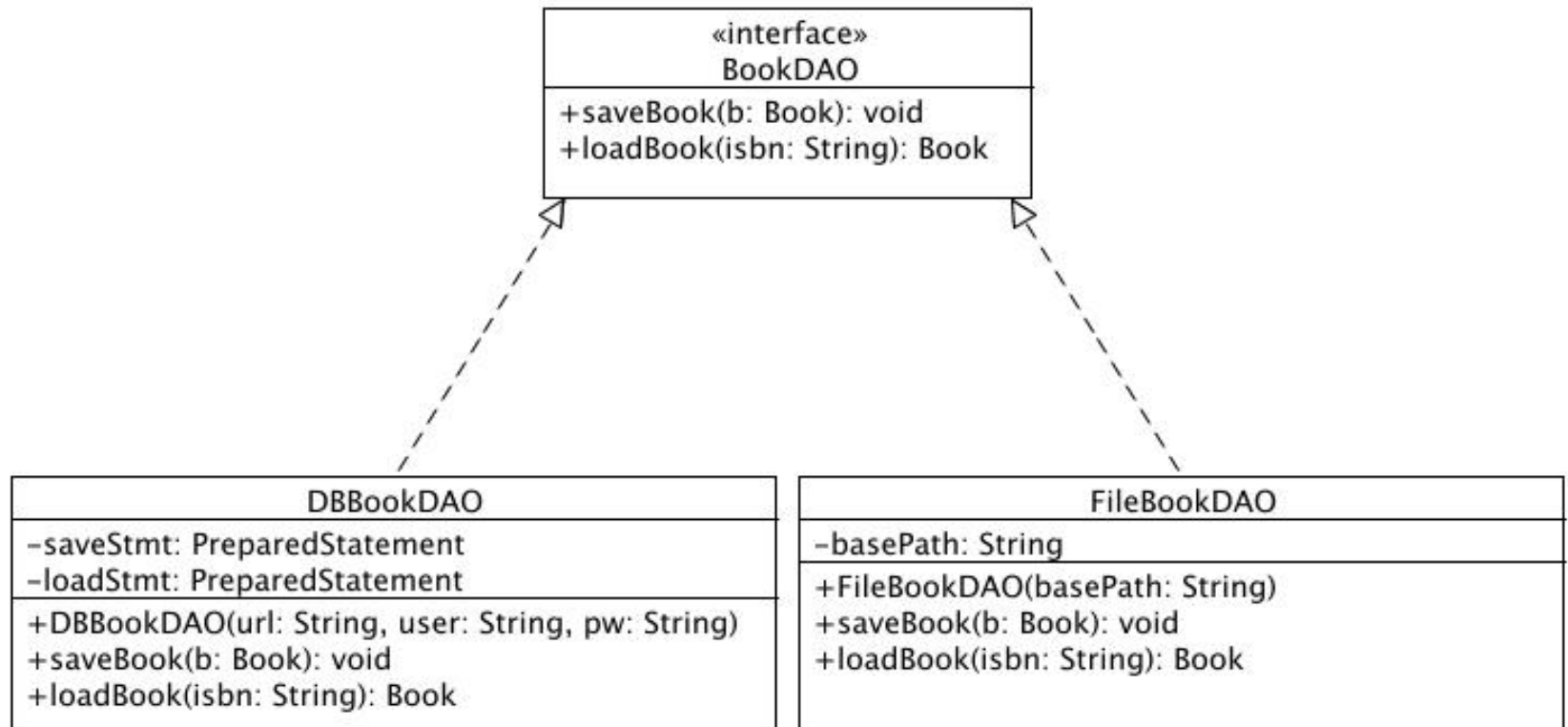
- ▶ Coordena a aplicação, processa comandos, toma decisões lógicas, faz avaliações e implementa as regras de negócio
- ▶ É **inerente** ao domínio (negócio) da aplicação
- ▶ Vários protocolos podem ser utilizados para ligar esta camada às outras duas
 - Sockets, HTTP, TCP/IP, etc. (Apresentação)
 - JDBC, LDAP, ODBC, etc. (Dados)

Camada de Acesso a Dados

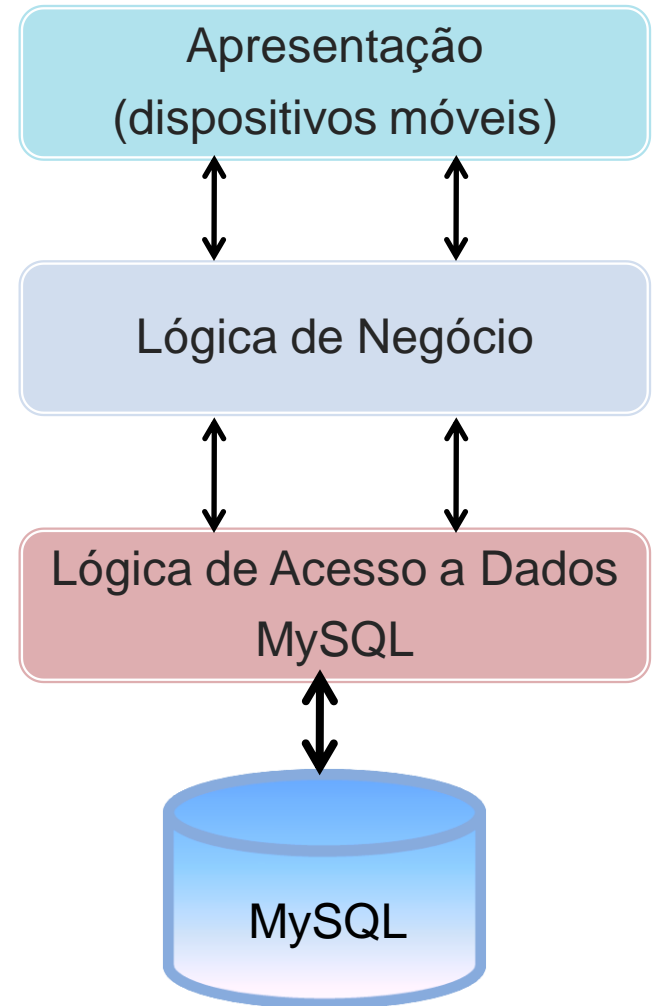
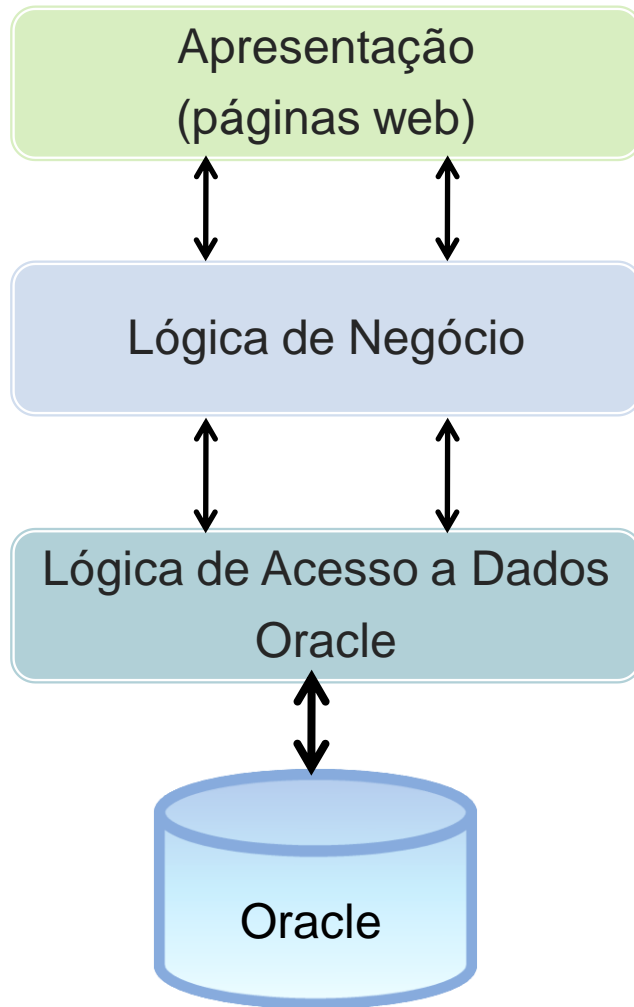
- ▶ Contém o código responsável por armazenar e recuperar dados de uma base de dados ou sistema de arquivos
- ▶ Normalmente há uma sub-camada (interface) dentro desta camada que abstrai o mecanismo de persistência
 - O famoso padrão DAO (*Data Access Object*) é utilizado aqui.

Camada de Acesso a Dados

Data Access Object

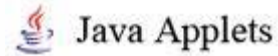


Exemplo – substituição de camadas



Exemplo – camadas e seus protocolos/tecnologias

Apresentação



Lógica do Negócio



Acesso a Dados



Exercícios [4]

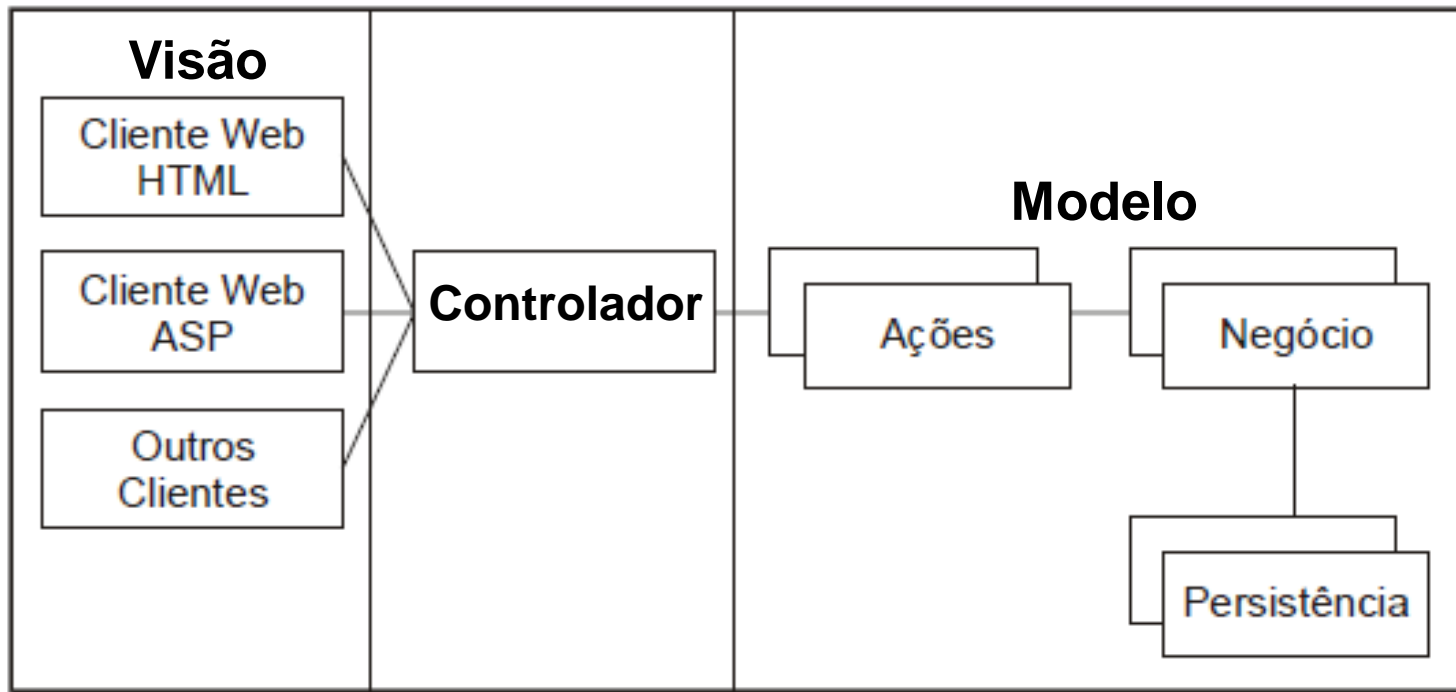
(PETROBRAS – CESGRANRIO 2010)

[7] A arquitetura de 3 camadas é comumente utilizada no desenvolvimento de aplicações para Internet. Nesse tipo de arquitetura, a lógica da aplicação é dividida entre as camadas físicas cliente, servidor de aplicação e banco de dados. NÃO é característica deste tipo de arquitetura o(a)

- (A) aumento da disponibilidade do serviço oferecido através da possibilidade de redundância dos servidores de aplicação e banco de dados.
- (B) facilidade de integração de múltiplas fontes de dados.
- (C) maior segurança, uma vez que o banco de dados não é acessado diretamente pelo cliente.
- (D) aplicação em larga escala, possibilitando o atendimento a vários clientes simultaneamente.
- (E) diminuição da complexidade e do esforço para o desenvolvimento da aplicação.

Arquitetura MVC (Model-View-Controller)

- ▶ Principal padrão de arquitetura em três camadas utilizado no mercado



Camada de Visão

- ▶ É a camada de interface com o usuário
- ▶ Responsável por receber a entrada de dados e apresentar os resultados
- ▶ Não está preocupada em como ou onde a informação foi obtida, apenas exibe a informação
- ▶ Inclui elementos de exibição no cliente
 - HTML, XML, ASP, Applets, etc.
- ▶ Pode requerer dados diretamente da camada de Modelo

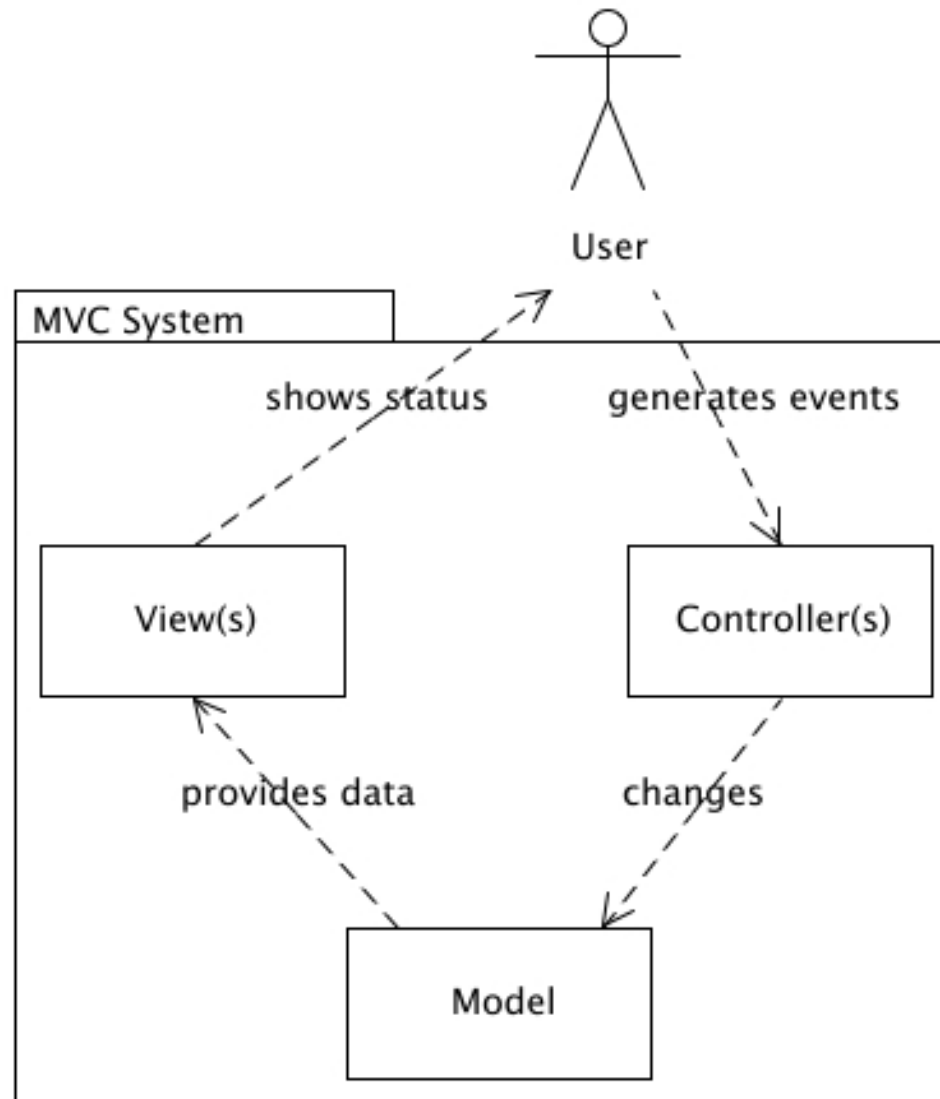
Camada de Controle

- ▶ Responsável por controlar e mapear as ações do usuário, fazendo o papel de intermediário entre a camada de Visão e de Modelo
- ▶ Atualiza o Modelo
- ▶ Seleciona a Visão

Camada de Modelo

- ▶ Responsável por modelar os dados e o comportamento por trás das regras de negócio
- ▶ Se preocupa com o armazenamento, manipulação e geração dos dados
- ▶ Objetos do Modelo são normalmente reusáveis, distribuídos, persistentes e portáteis para várias plataformas

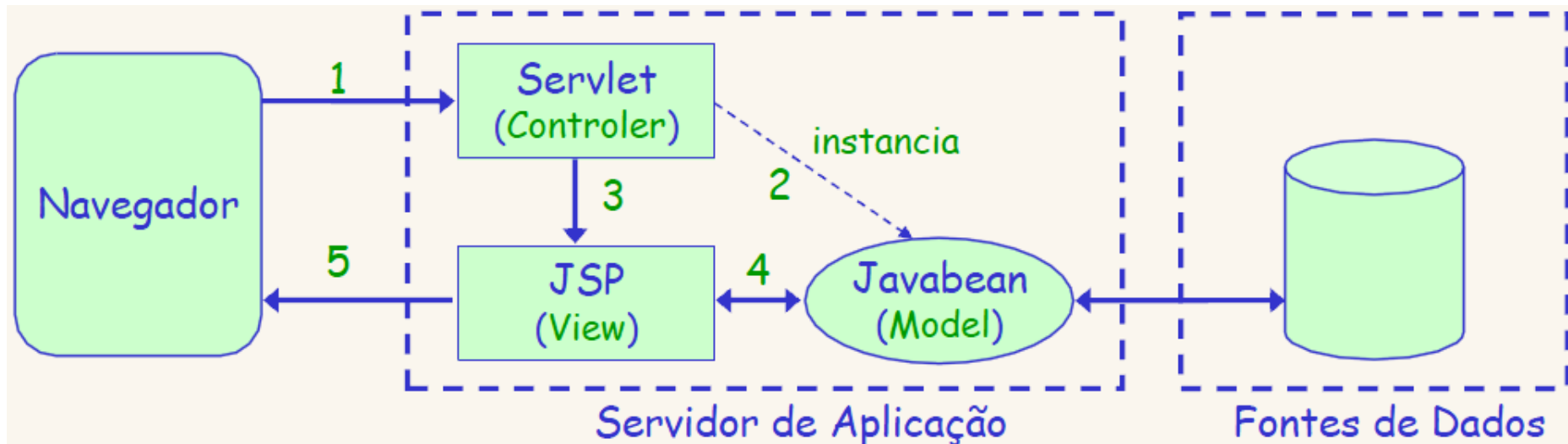
Interações entre as camadas MVC



MVC versus *Three-tier Architecture*

- ▶ A arquitetura em três camadas “pura” é linear – toda comunicação deve passar pela camada intermediária
- ▶ A arquitetura MVC é triangular – nem toda comunicação passa pelo Controlador
 - A Visão despacha atualizações para o Controlador
 - O controlador atualiza o modelo
 - **A Visão é atualizada diretamente pelo Modelo**

Arquitetura MVC na WEB (Model 2)



Exercícios [5]

(Assembléia Legislativa de SP – FCC 2010)

[42] Sobre as camadas do modelo de arquitetura MVC (Model-View-Controller) usado no desenvolvimento web é correto afirmar:

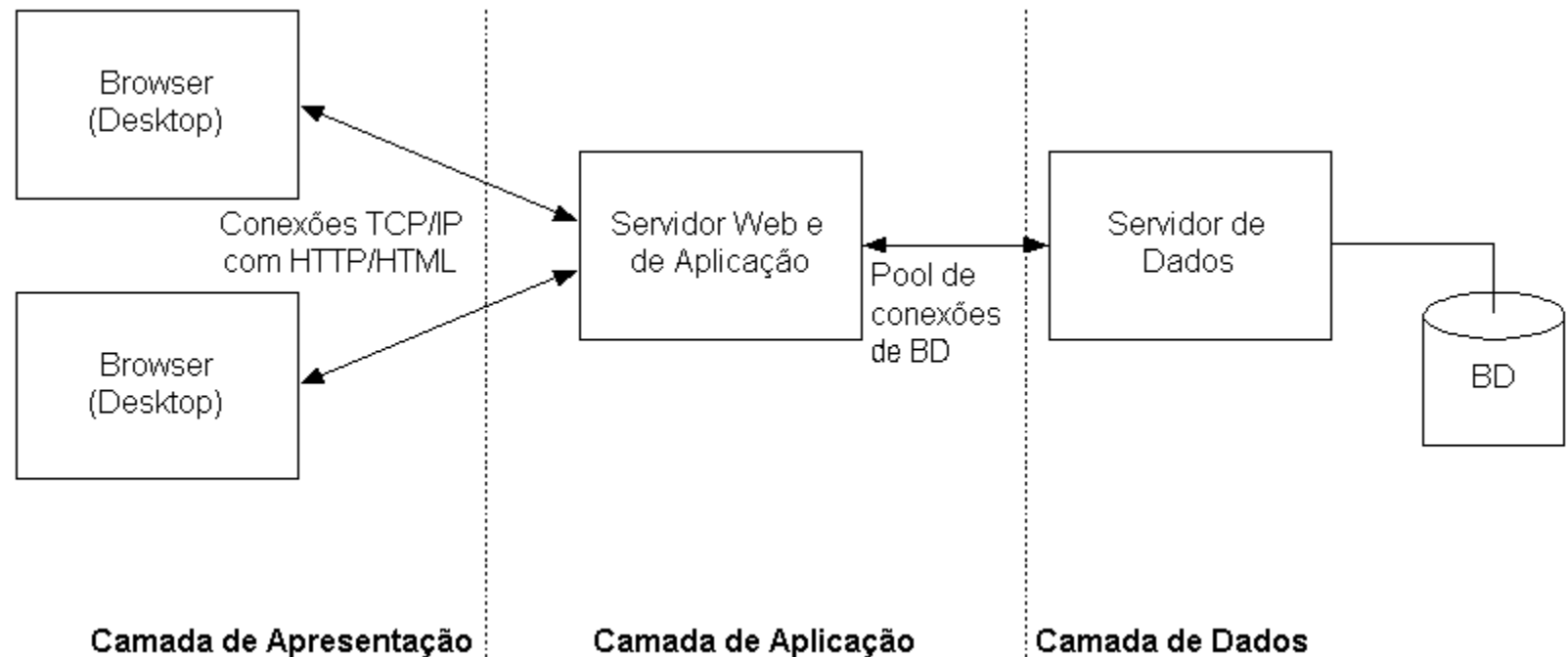
- (A) Todos os dados e a lógica do negócio para processá-los devem ser representados na camada Controller.
- (B) A camada Model pode interagir com a camada View para converter as ações do cliente em ações que são compreendidas e executadas na camada Controller.
- (C) A camada View é a camada responsável por exibir os dados ao usuário. Em todos os casos essa camada somente pode acessar a camada Model por meio da camada Controller.
- (D) A camada Controller geralmente possui um componente controlador padrão criado para atender a todas as requisições do cliente.
- (E) Em aplicações web desenvolvidas com Java as servlets são representadas na camada Model.

Arquitetura WEB

- ▶ Com o advento da WEB, o *browser* passou a ser utilizado como cliente universal
 - Evitamos instalar qualquer software em desktops, facilitando a manutenção
- ▶ O número e nome das camadas variam
 - No mínimo, costuma-se ter três camadas (pequenos volumes)
 - Mas pode haver “N” camadas, dependendo da necessidade (*N-tier architecture*)

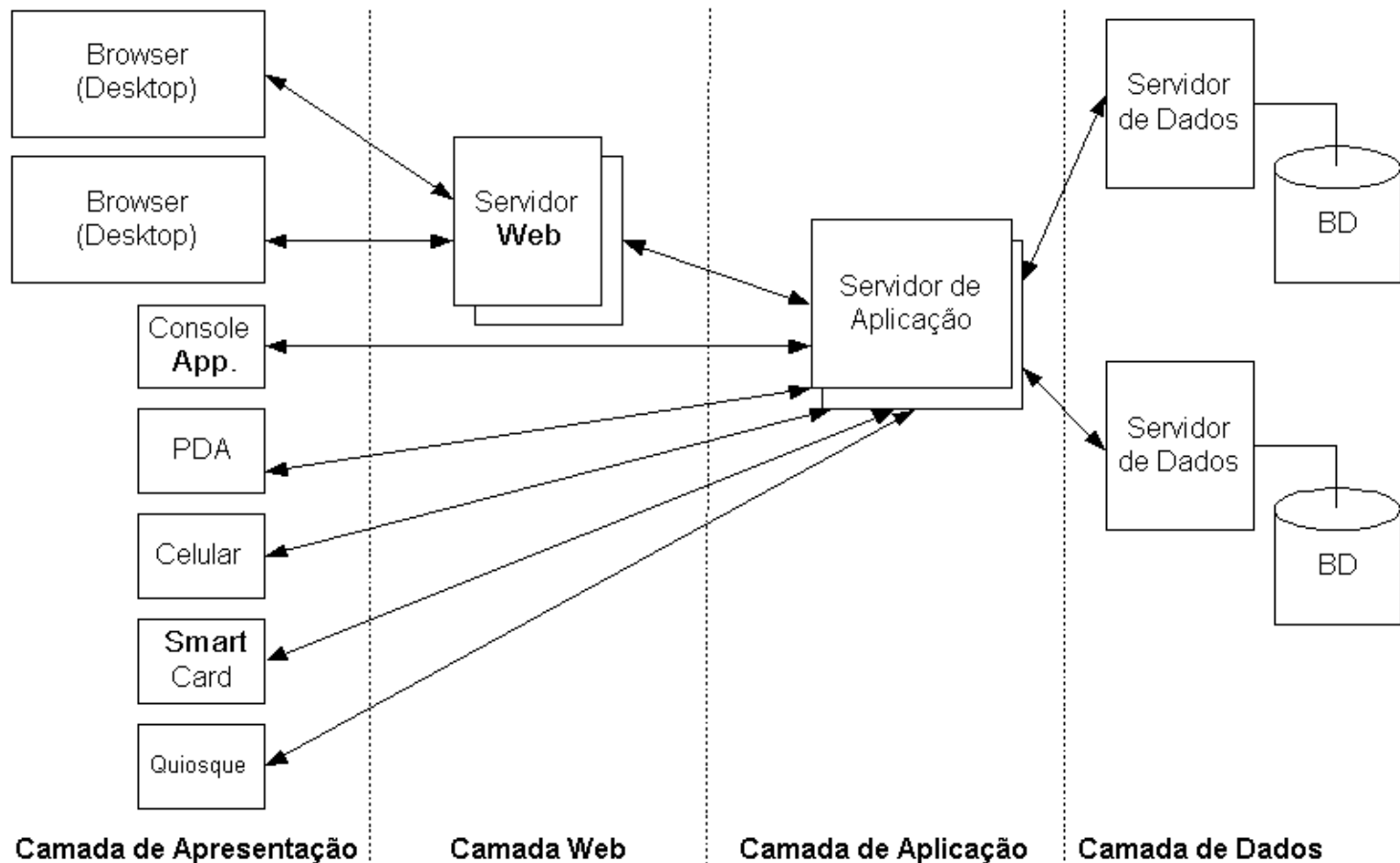
Arquitetura WEB (três camadas)

Para projetos mais simples podemos ter as camadas Web e Aplicação no mesmo local



Arquitetura WEB (n-camadas)

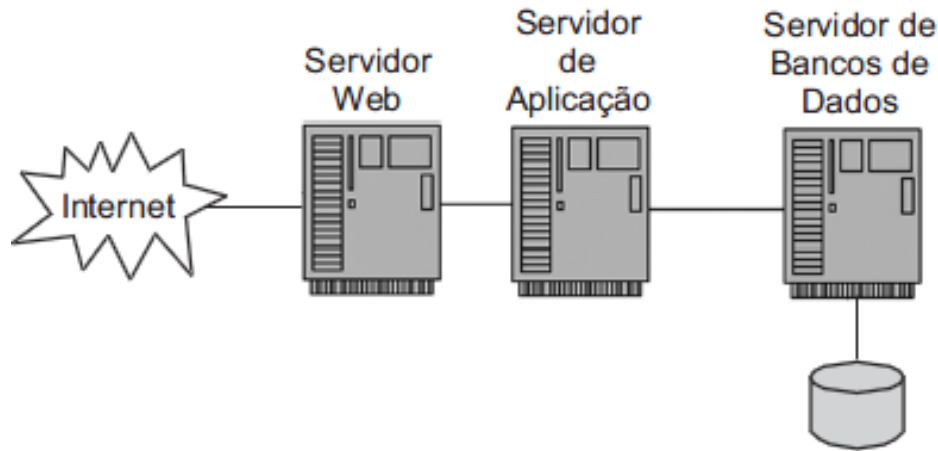
Mas podemos ter mais camadas (flexibilidade)



Exercícios [6]

(IBGE – CESGRANRIO 2010)

[61] A figura abaixo apresenta uma típica arquitetura de 3 camadas utilizada para disponibilizar sites na Internet.



Sobre essa arquitetura, são feitas as afirmativas abaixo.

I – Drivers que seguem o padrão ODBC podem ser utilizados por aplicações que estão no servidor de aplicações para acessar tabelas no servidor de Banco de dados.

Exercícios [6]

II – Se o nível de processamento aumentar, um novo servidor de aplicações pode ser colocado em uma estrutura de cluster para responder aos pedidos do servidor Web e, nesse caso, a replicação de sessão, presente em alguns servidores de aplicação, garante que um servidor assuma as funções de um servidor com problemas, sem que o usuário perceba o ocorrido.

III – Como uma boa prática na implementação de soluções distribuídas, a lógica de negócio é implementada em componentes que ficam instalados no servidor Web, sendo que o servidor de aplicações funciona como intermediário entre o servidor web e o de banco de dados gerenciando as transações.

Está(ão) correta(s) a(s) afirmativa(s)

(A) I, apenas. (B) II, apenas.

(C) III, apenas. (D) I e II, apenas.

(E) I, II e III.

Gabarito dos Exercícios

- ▶ [1] – [54] D, [61–I] C, [61–III] C, [61–IV] C, [61–V] C, [100] C, [96] E
- ▶ [2] – [125] C, [47] E, [22–e] E, [52] D
- ▶ [3] – [35–D] E, [36–C] E, [62–E] E
- ▶ [4] – [7] E
- ▶ [5] – [42] D
- ▶ [6] – [61] D

FIM



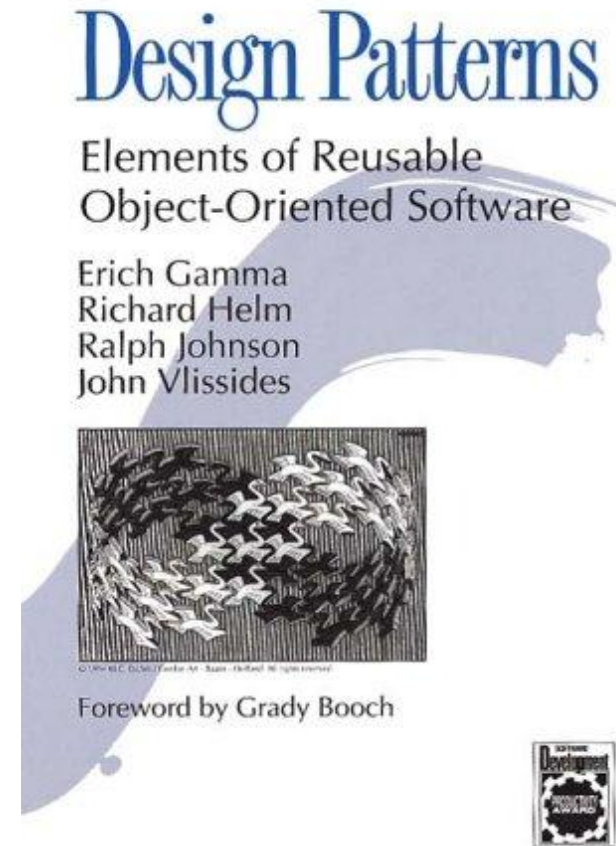
Padrões de Projeto

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 participantes 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 benefícios 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

		Propósito		
		Criação	Estrutura	Comportamento
Escopo	Classe	Factory Method	Adapter (classe)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (objeto) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

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)

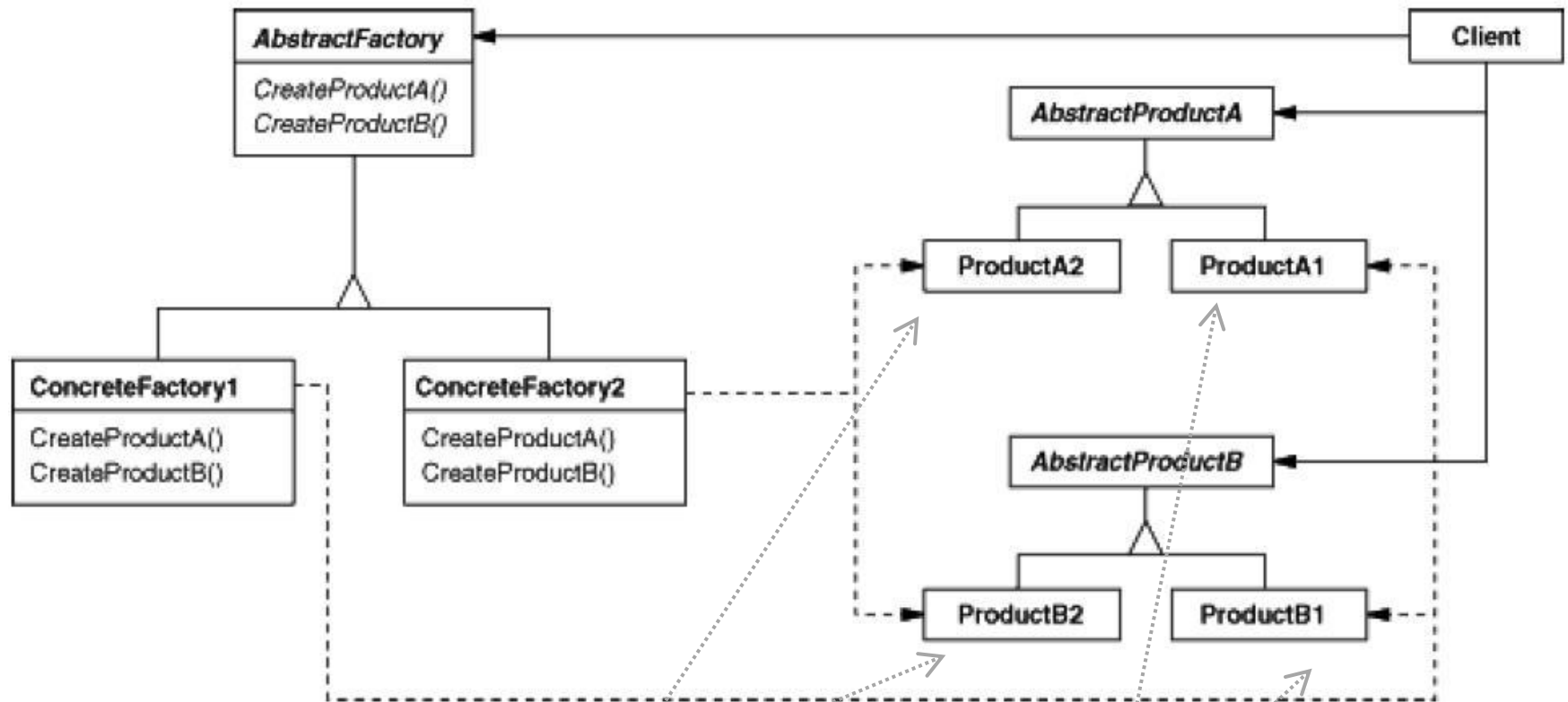
[82] O emprego de padrões de projeto reusáveis, como façade, builder e singleton, é uma prática com nível inferior de abstração, quando comparado ao emprego de estilos arquiteturais de software, como camadas, cliente-servidor e peer-to-peer.

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

Produtos concretos

```
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");  
    }  
}
```


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

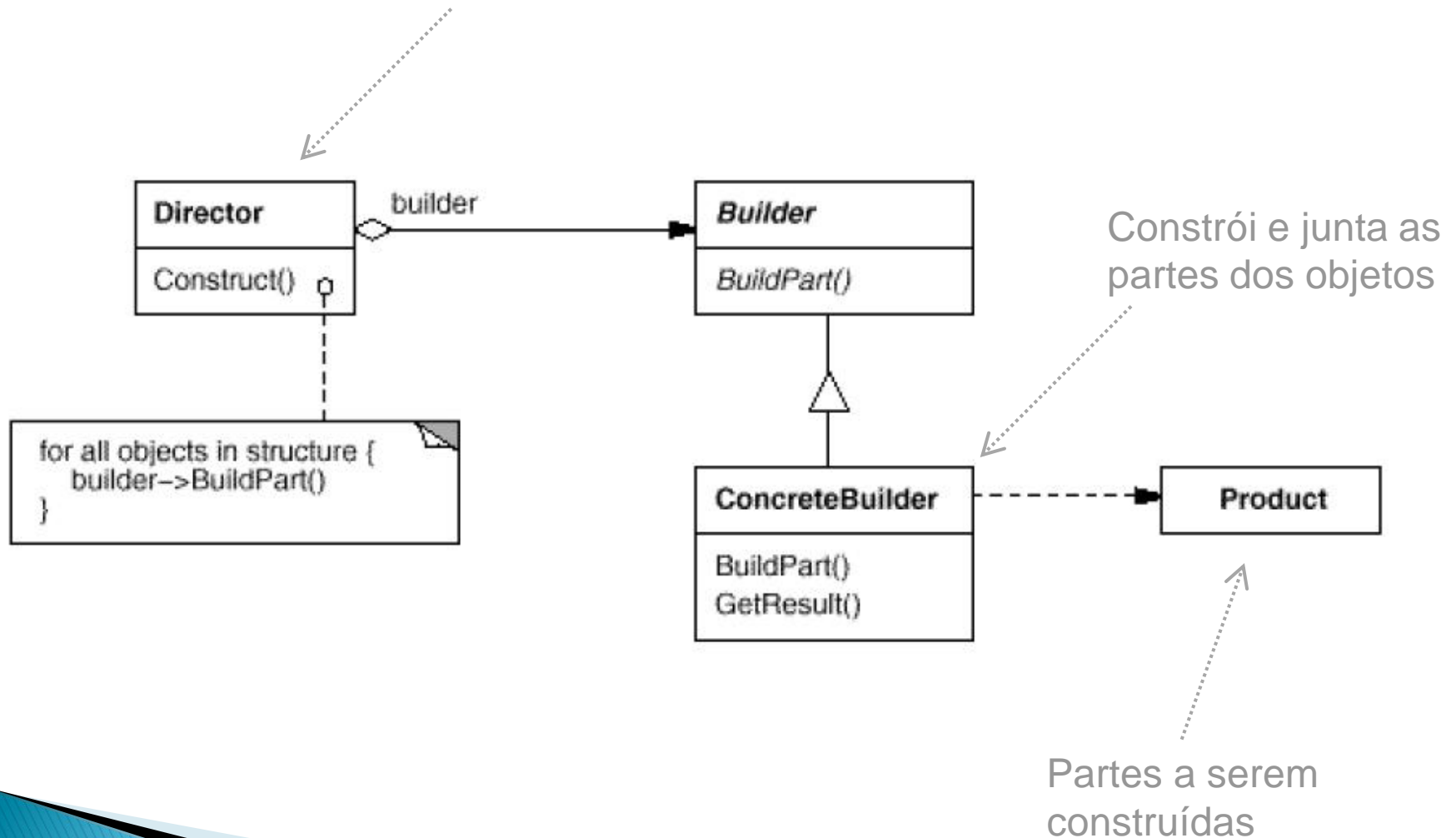
A decisão I pode ser atendida usando-se o padrão de projeto Abstract Factory;

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



Builder

Produto a ser construído

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

Partes do produto

Classe construtora
abstrata (genérica)

```
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();  
}
```

Métodos para
construir cada parte

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

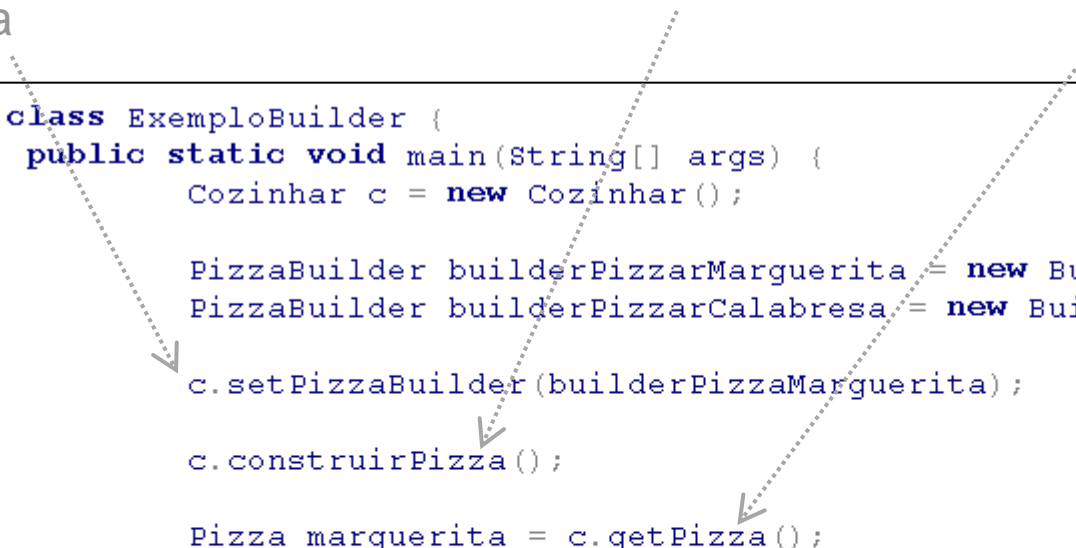
Builder (executando)

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(builderPizzarMarguerita);  
  
        c.construirPizza();  
  
        Pizza marguerita = c.getPizza();  
  
        c.setPizzaBuilder(builderPizzarCalabresa);  
  
        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.

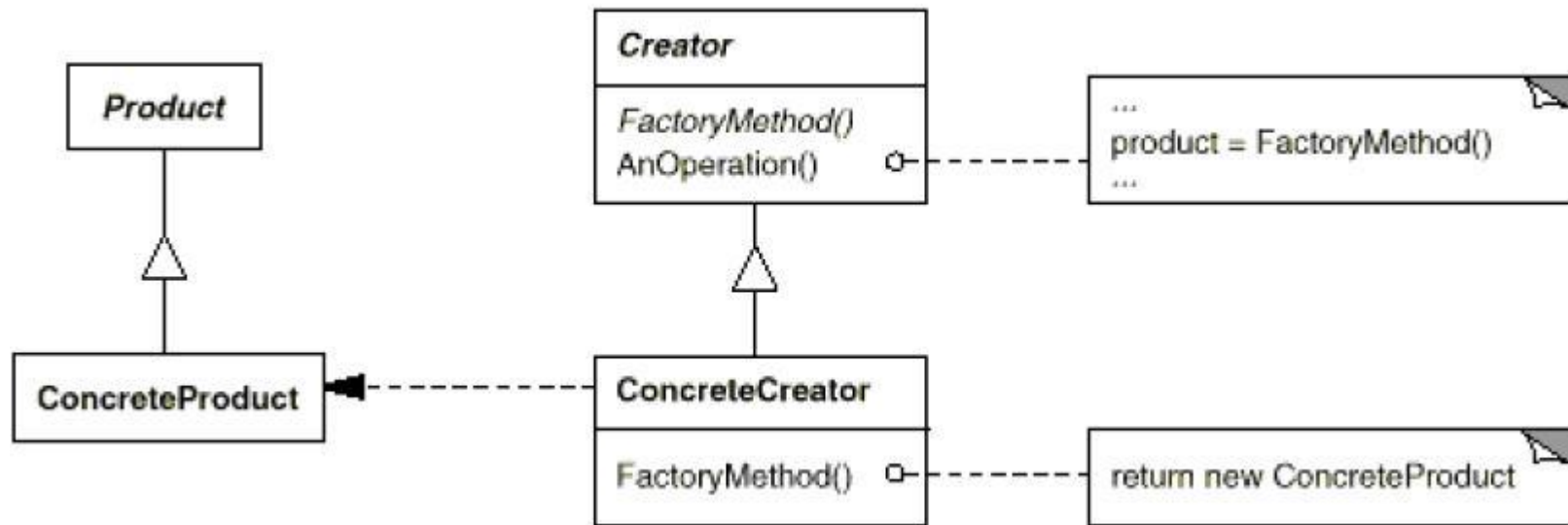
(SERPRO – CESPE 2010)

[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

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

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

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

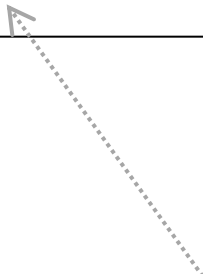
public class Mulher extends Pessoa { Produto concreto
    public Mulher(String nomeCompleto){
        System.out.println("Ola, Senhora " + nomeCompleto);
    }
}
```

```
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

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.

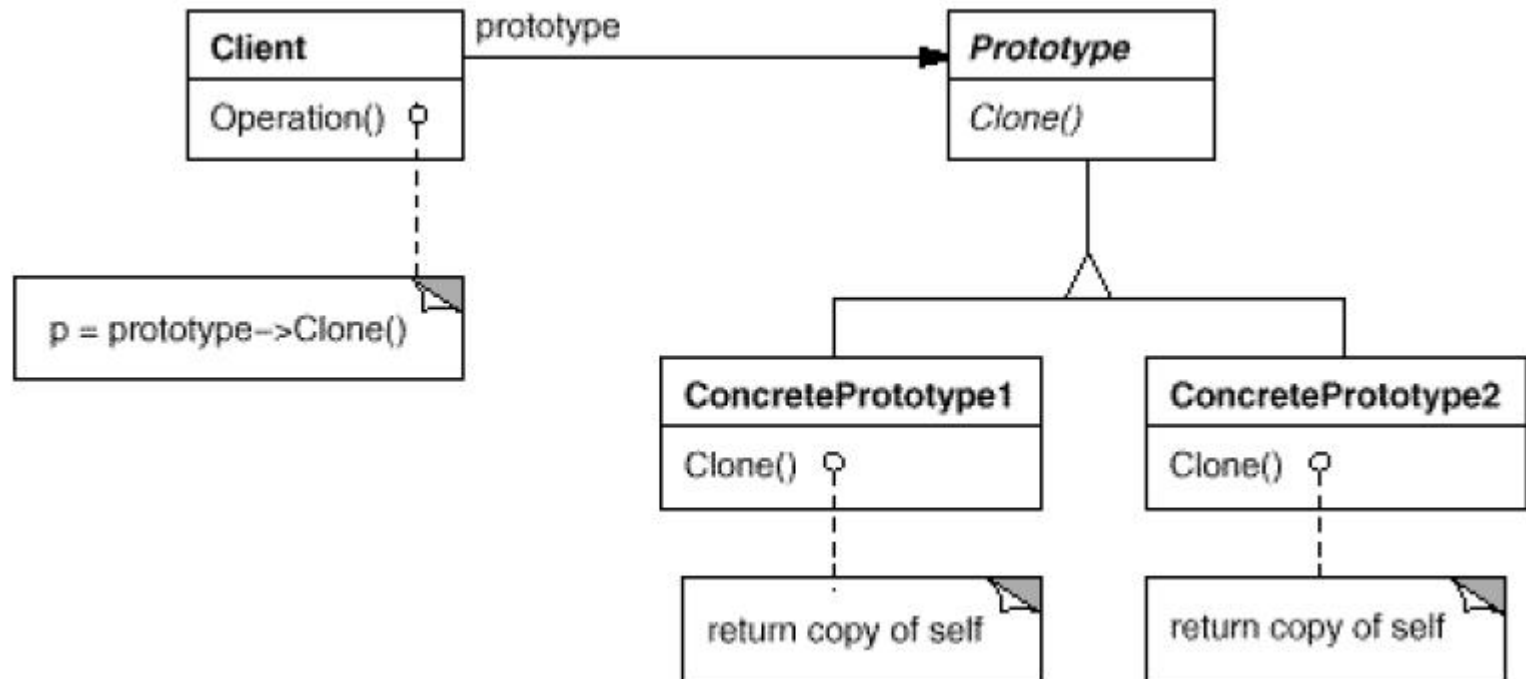
(TRE/MS – FCC 2007)

[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



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 { }
```

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();  
        }  
    }  
}
```

Protótipo abstrato

Classe cliente

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

Exercícios [5]

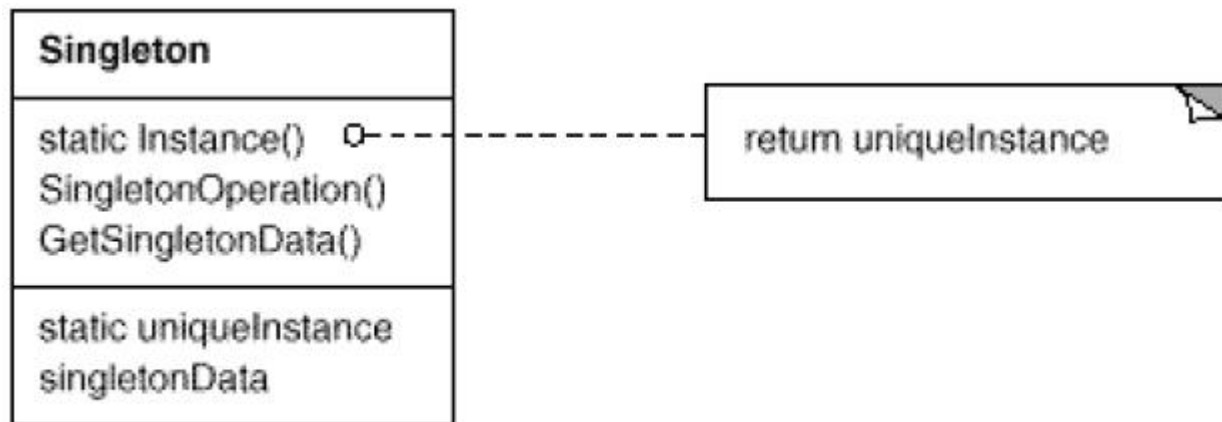
(IRB – ESAF 2006)

[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

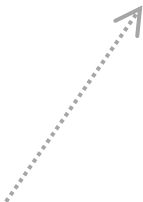
Singleton



Singleton

```
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?

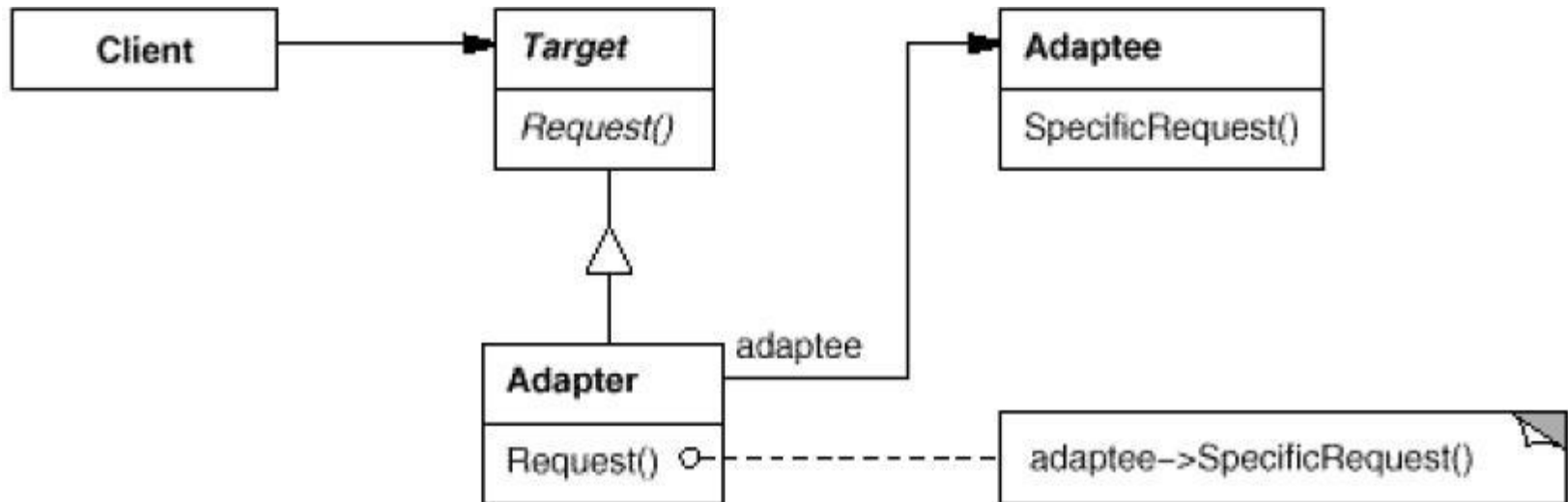
(A) Iterator (B) Visitor (C) Bridge (D) Memento (E) Singleton

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



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);  
    }  
}
```

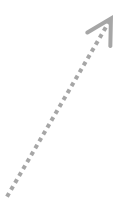
Classe alvo (target): é o que o cliente possui

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.

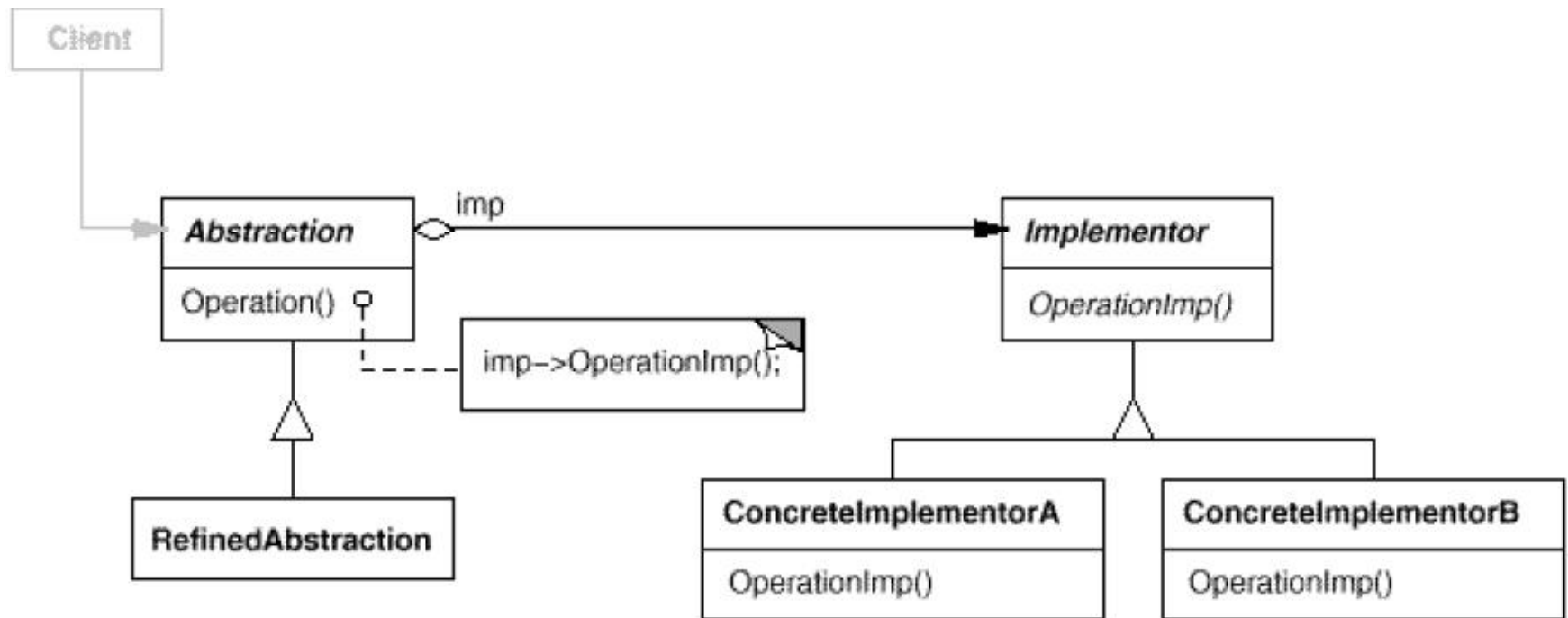
(SERPRO – CESPE 2008)

[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



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)

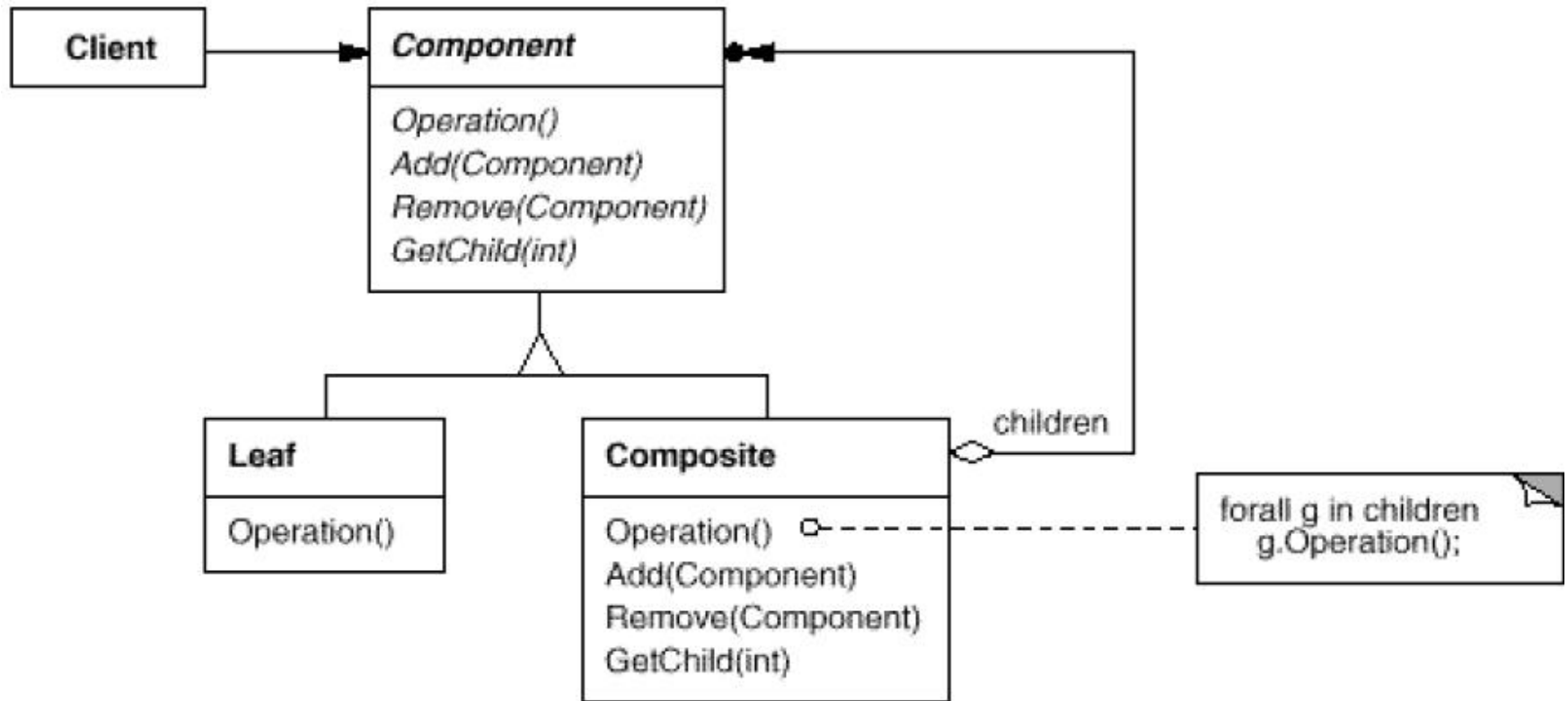
[60] Ao consultar informações a respeito dos padrões de projeto Adapter e Bridge, um Analista de Sistemas identificou uma afirmativa INCORRETA. Assinale-a.

- (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



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.

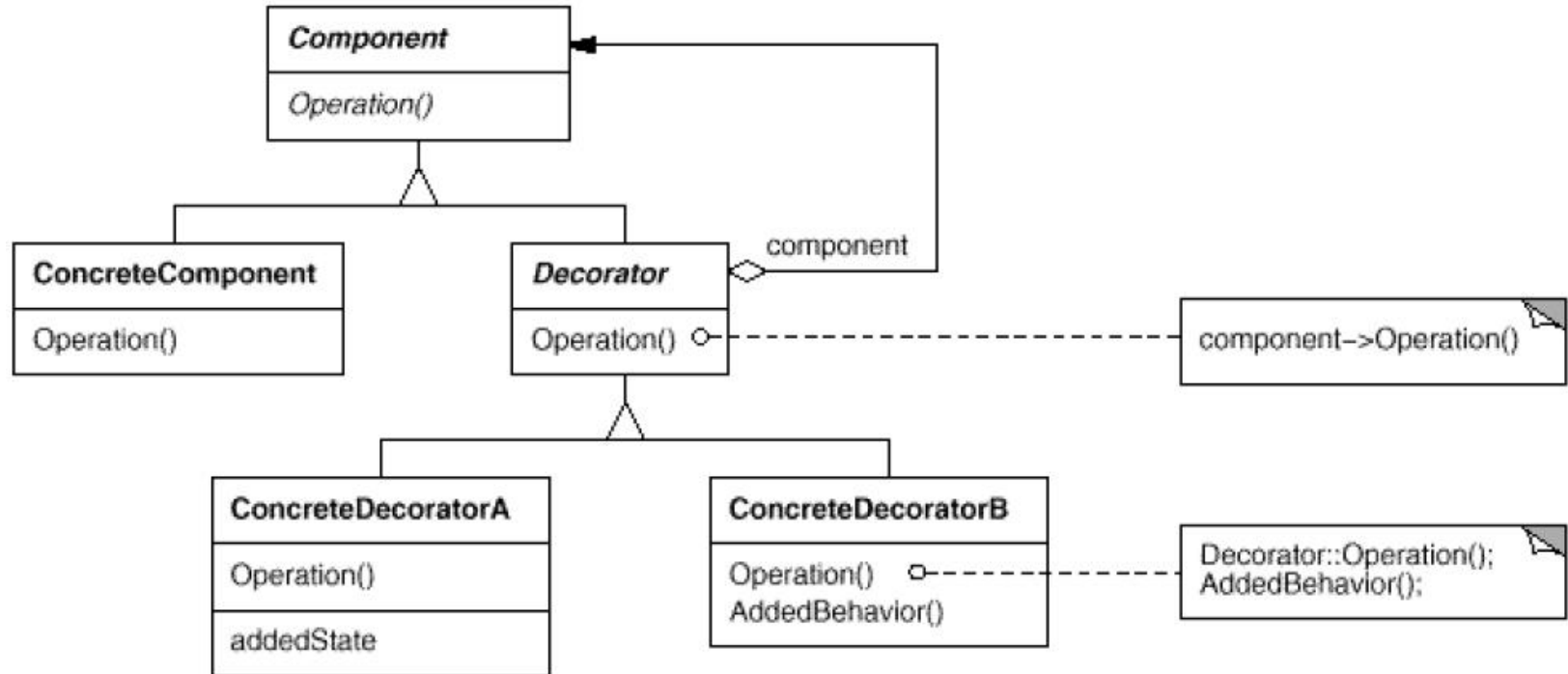
(TRE/AP – CESPE 2007)

[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



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 [1 1]

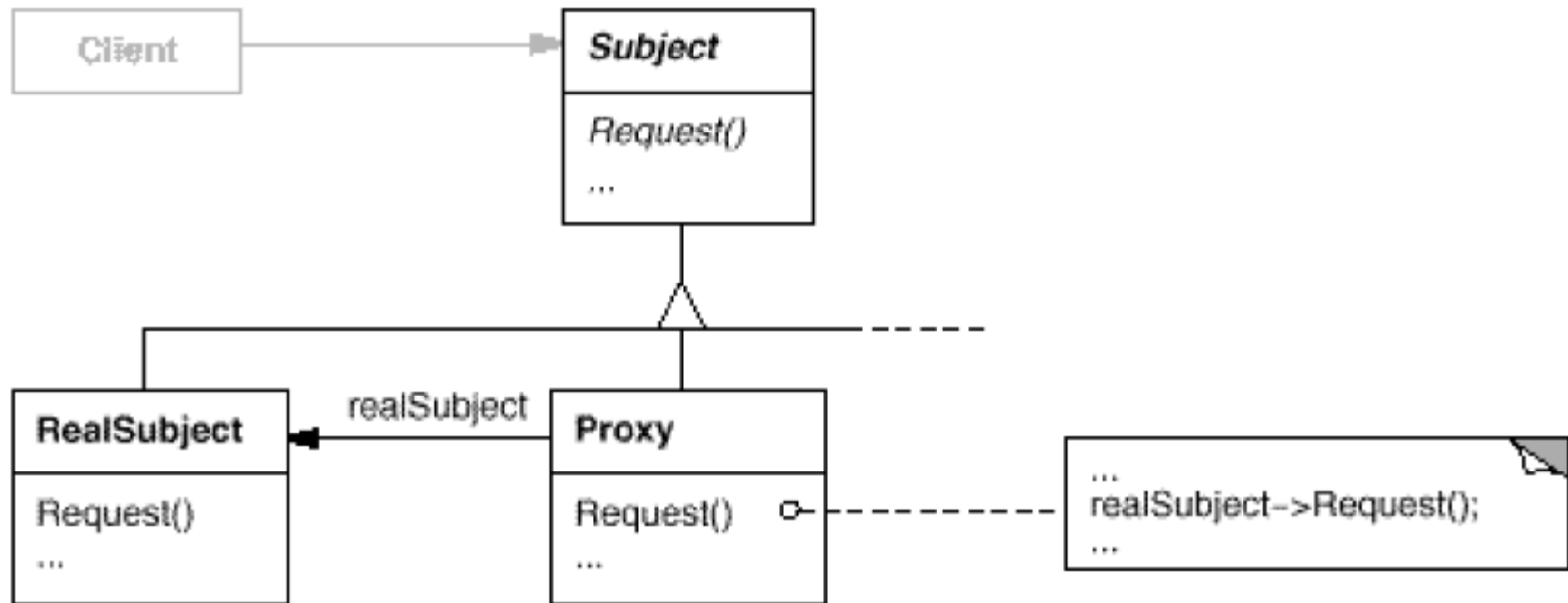
(INMETRO – CESPE 2009)

[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



Proxy

Objeto real →

Proxy ↘

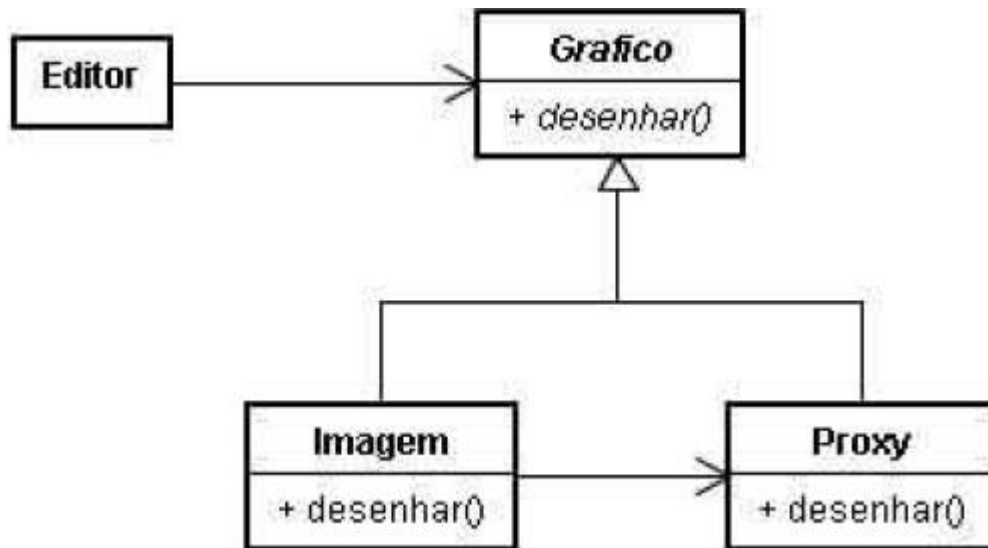
```
interface Pessoa {  
    public String getNome();  
}  
  
//Objeto real  
class PessoaImpl implements Pessoa {  
    private String nome;  
  
    public PessoaImpl(String nome) {  
        this.nome = nome;  
    }  
    public String getNome() {  
        return nome;  
    }  
}
```

```
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)

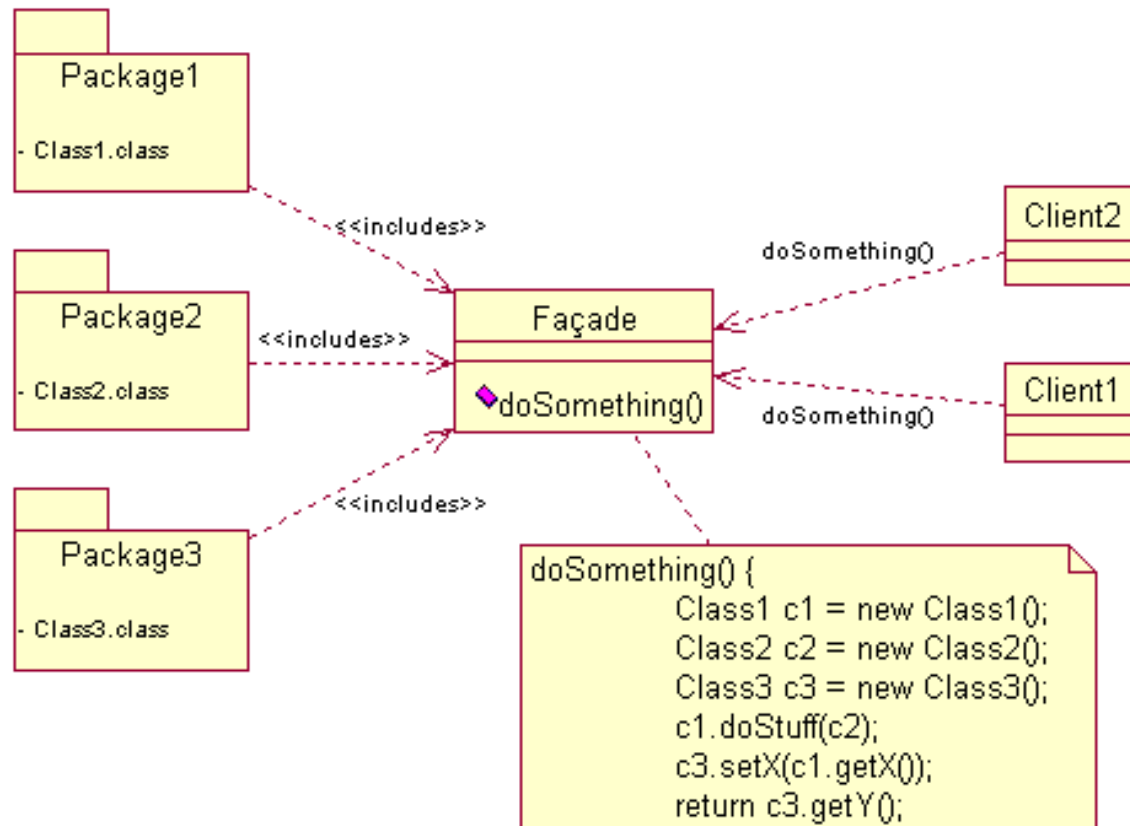
[99] O padrão proxy está corretamente documentado no seguinte diagrama UML.



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



Façade

Partes complexas,
com várias
interfaces

```
/* 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();  
    }  
}
```

```
/* 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) {  
        ...  
    }  
}
```

Interface unificada
na Façade

Exercícios [13]

(BNDES – CESGRANRIO 2009)

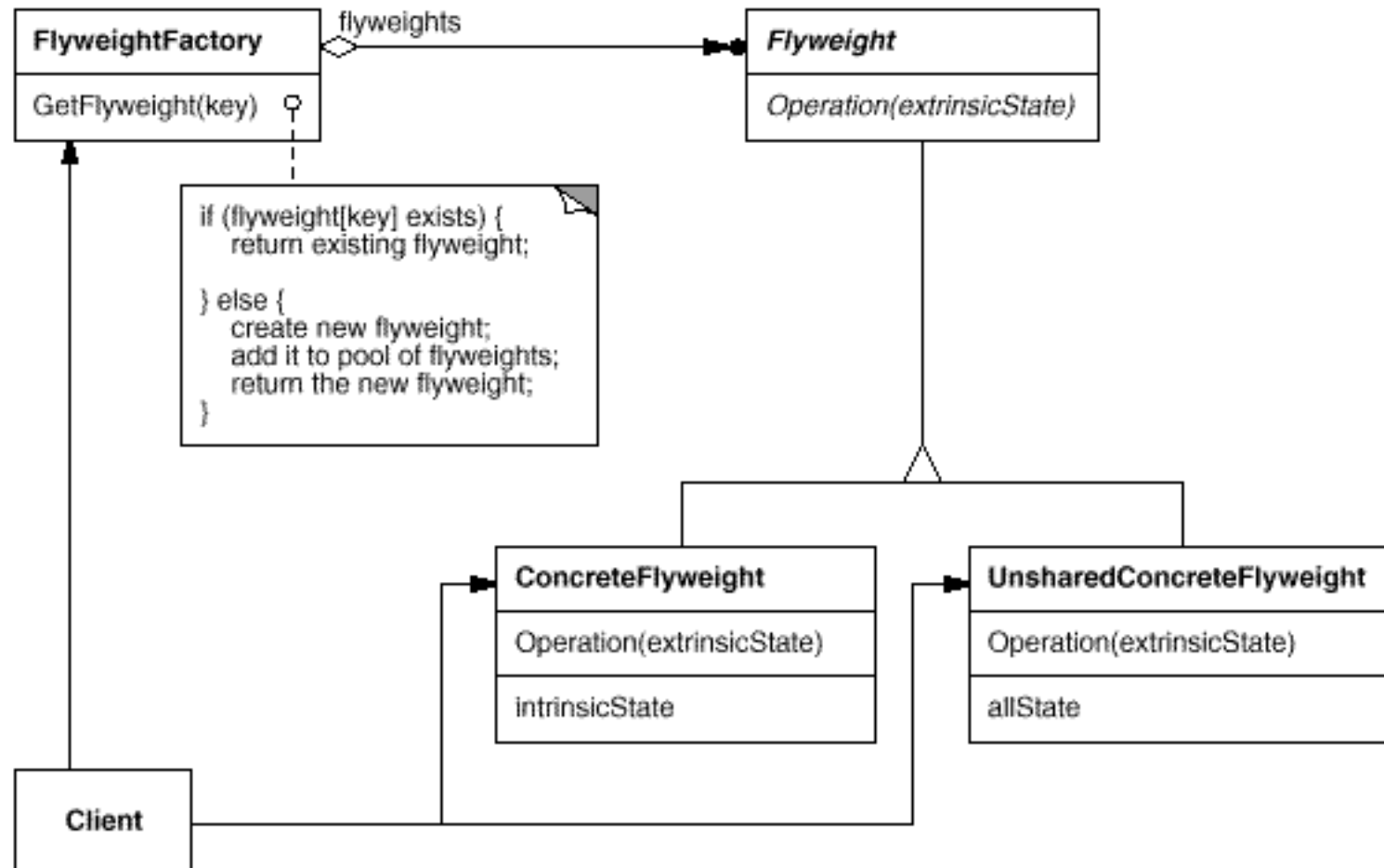
[59] O presidente de uma empresa determinou que fosse disponibilizado um sistema de vendas na Internet. No entanto, o software de controle de estoque que deve ser acessado pela aplicação de vendas é muito antigo e provê uma API (Application Programming Interface) de uso muito complicado. Para que os 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



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;  
    }  
}
```

```
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];  
    }  
}
```

← Objeto Flyweight

Factory de Flyweights



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

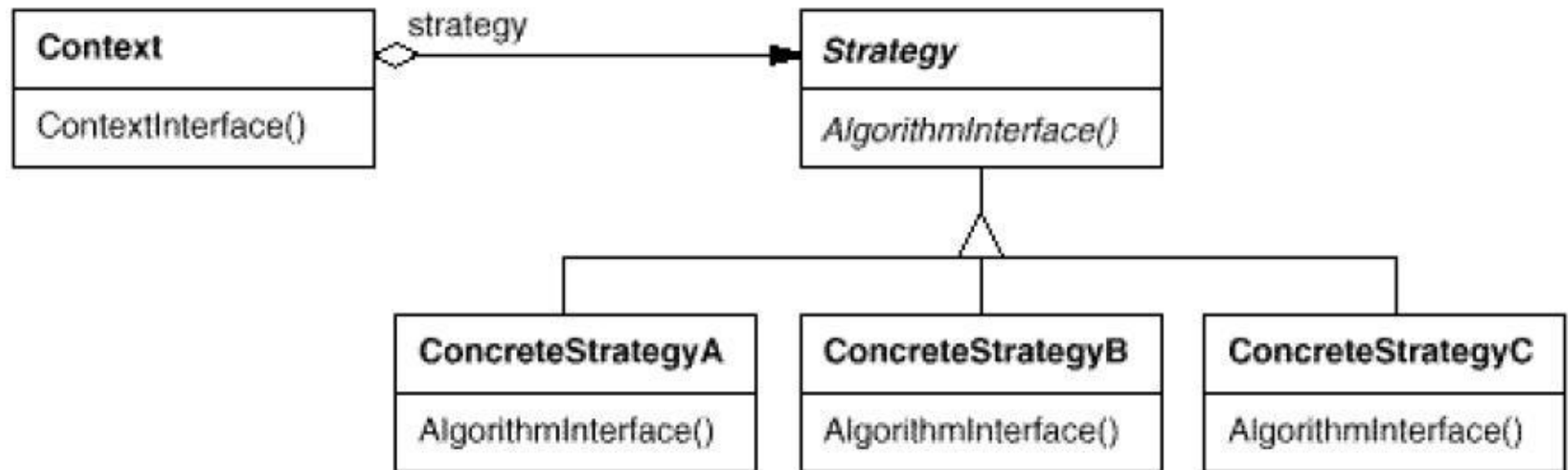
- a) Strategy
- b) Composite
- c) Flyweight
- d) State
- e) Builder

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



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 [1 5]

(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.

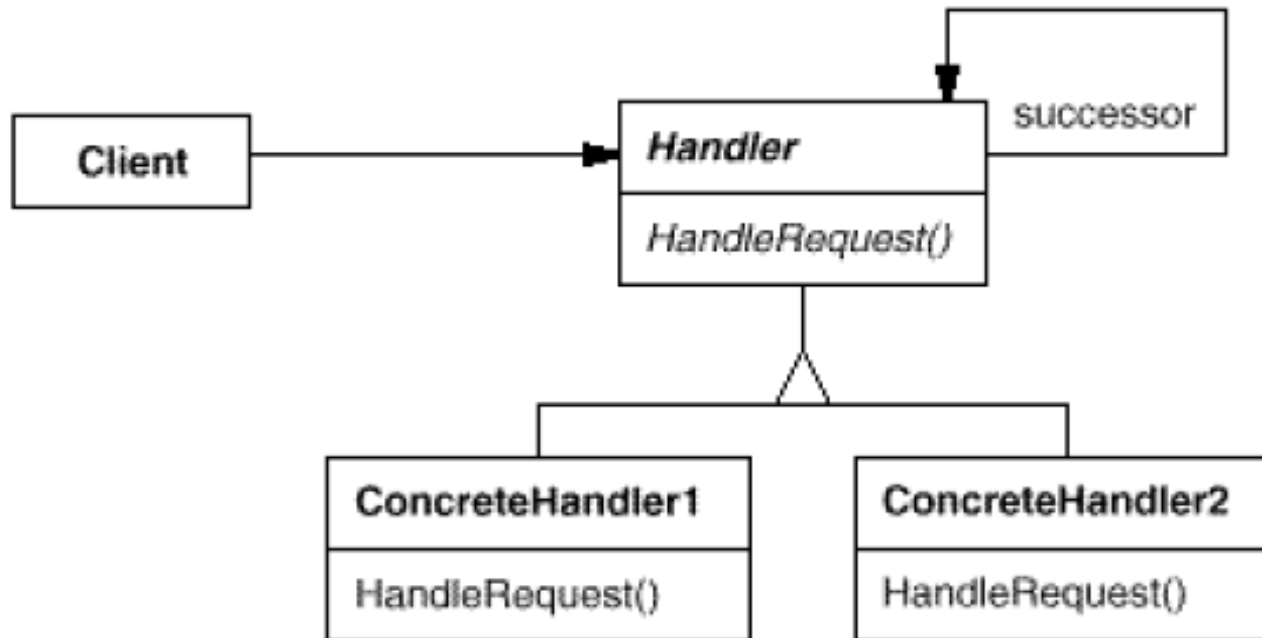
(INMETRO – CESPE 2009)

[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 cadeia 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();  
}
```

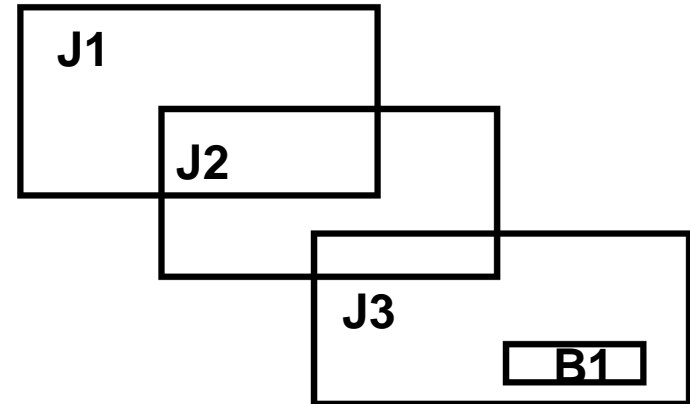
```
public class Janela implements HelpHandler {  
    //Handler concreto  
    private HelpHandler helpSuccessor;  
    public Janela (HelpHandler helpSuccessor) {  
        this.helpSuccessor = helpSuccessor;  
    }  
  
    public void handleHelp() {  
        if (temHelp) {  
            //codigo de tratamento do help...  
        } else {  
            helpSuccessor.handleHelp();  
        }  
    }  
}
```

```
public class Botao {  
    //Handler concreto  
    private HelpHandler helpSuccessor;  
    public Botao (HelpHandler 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)

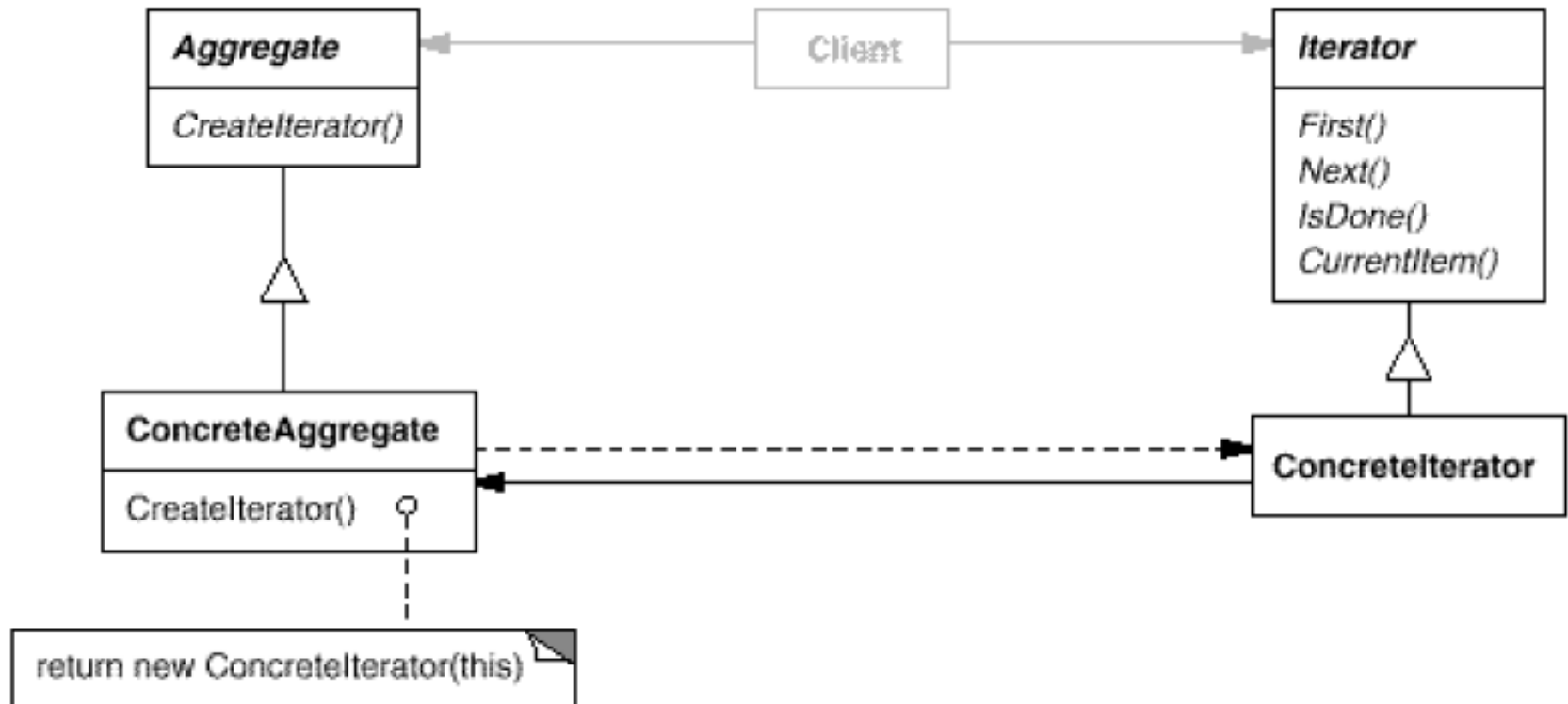
[33] Um arquiteto de software estuda que padrões de projeto são apropriados para o novo sistema de vendas de uma empresa. Ele deve considerar que o padrão

- 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



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 [1 7]

(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 [1 7]

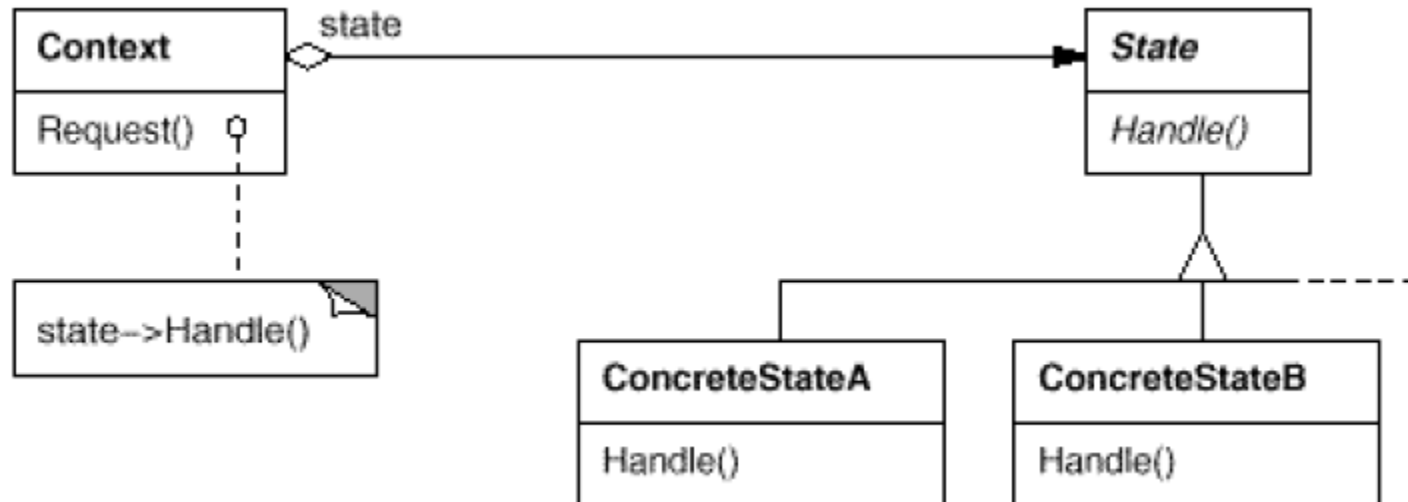
As afirmações correspondem, respectivamente, aos padrões

- 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



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

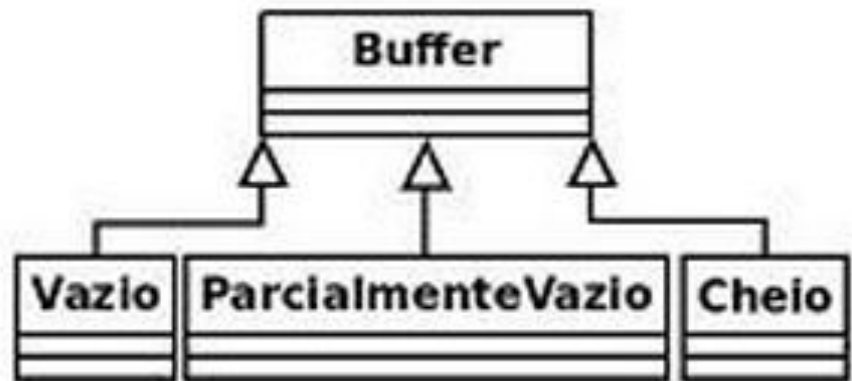
```
public class StateDemo {  
    public static void main( String arg[] ) {  
  
        //criando o objeto de contexto  
        TCPConnection conn = new TCPConnection();  
  
        //configurando o objeto para o estado CLOSED  
        TCPState stateClosed = new TCPClosed();  
        conn.setState(stateClosed);  
  
        //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 [1 8]

(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?

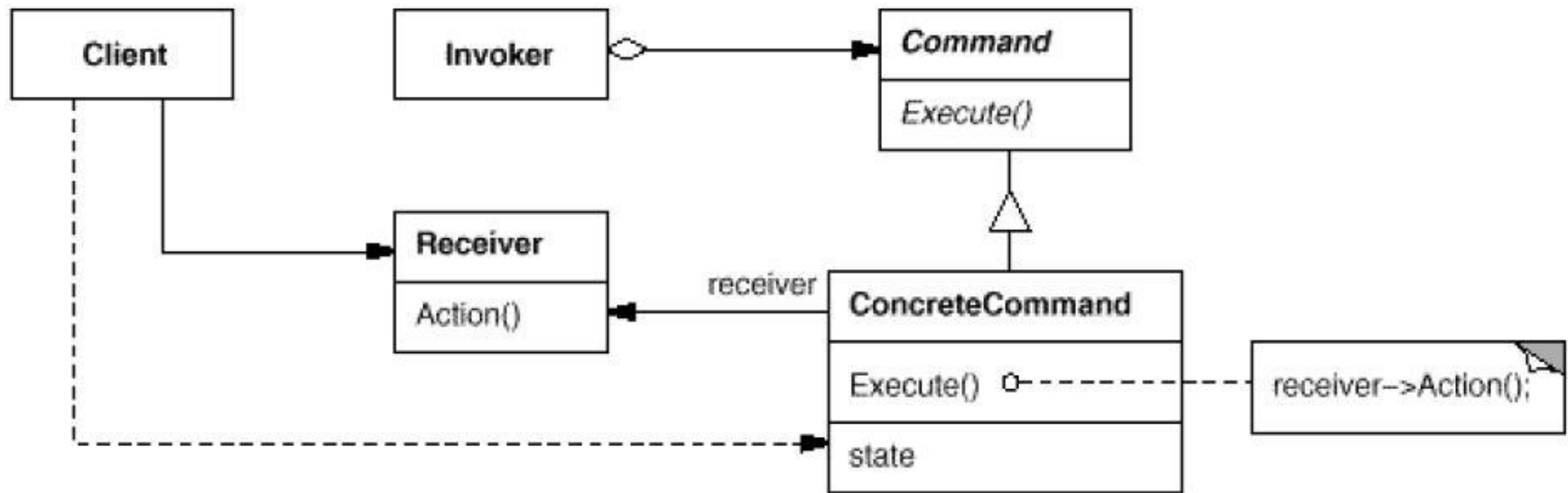
- a) Singleton.
- b) Dynamic behavior.
- c) Mediator.
- d) Composite.
- e) State.



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



Command

O invoker “control” 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]

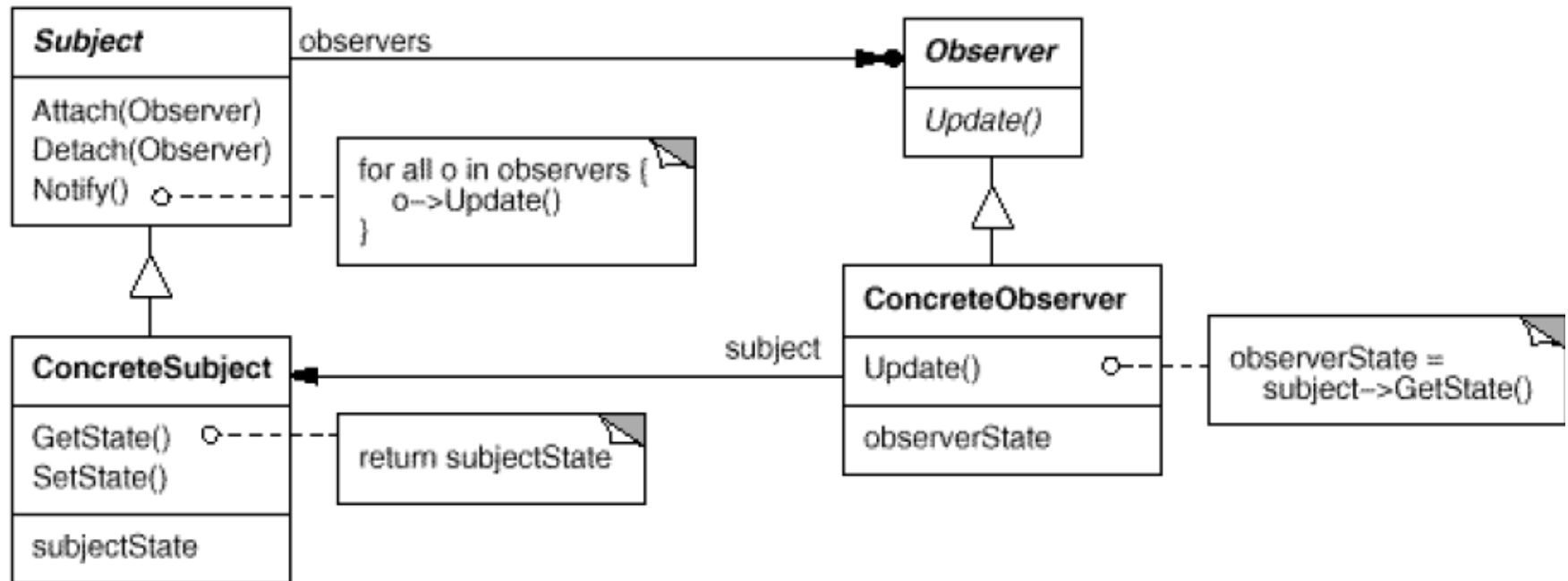
(INMETRO – CESPE 2009)

[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



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.

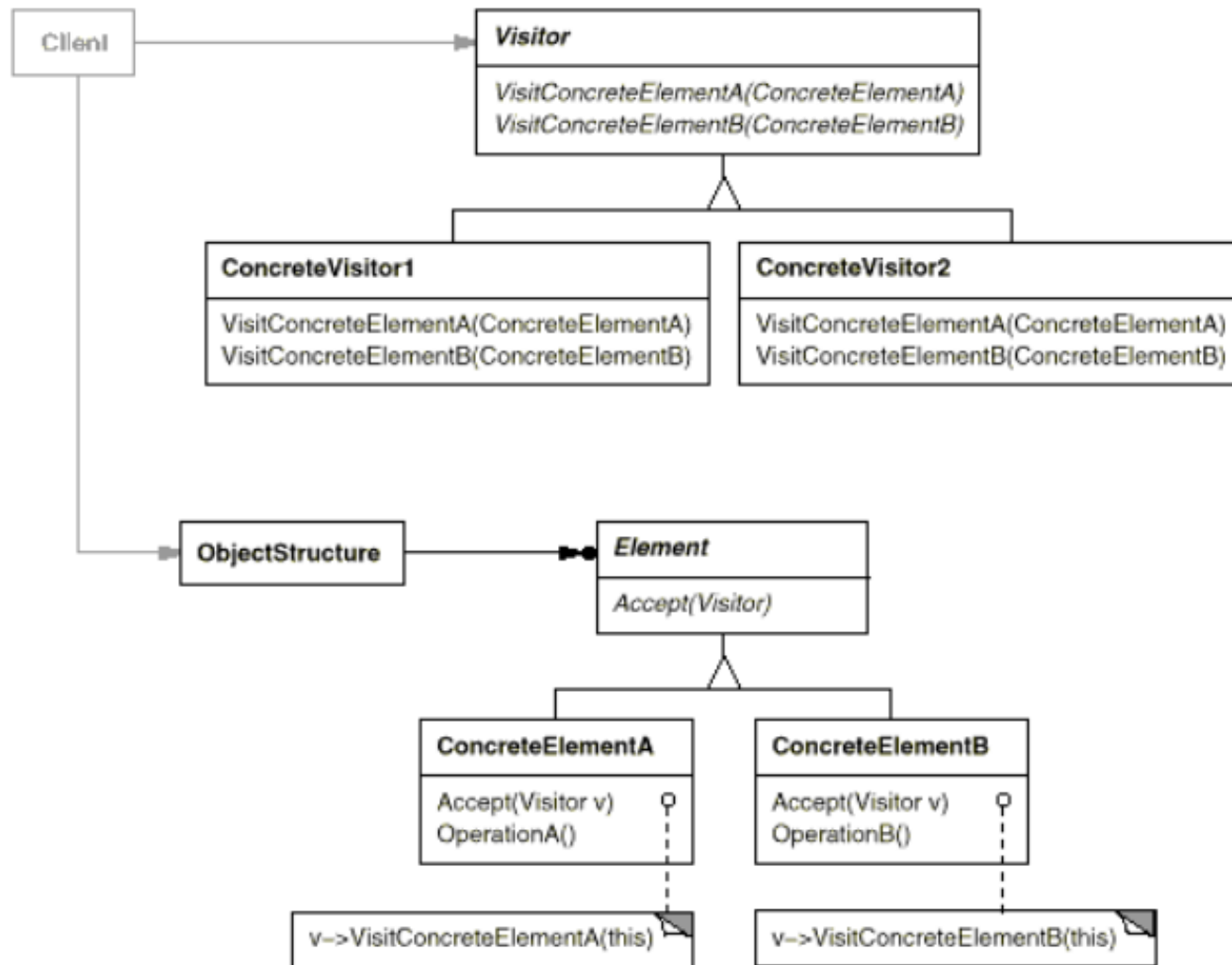
(Casa da Moeda – CESGRANRIO 2009)

[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



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]

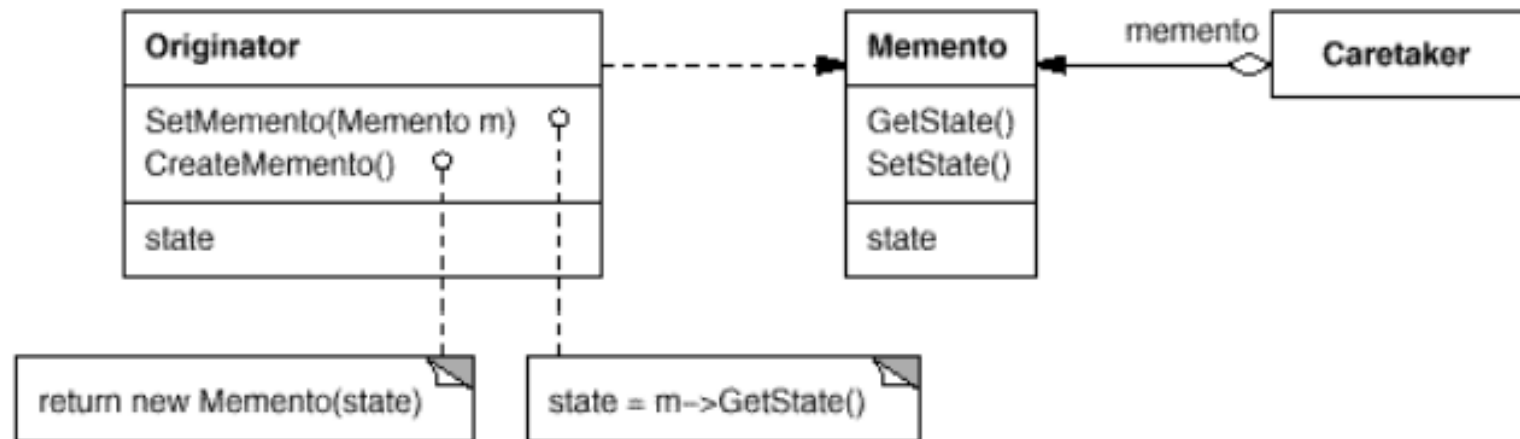
(TRE/AP – CESPE 2007)

[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



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.

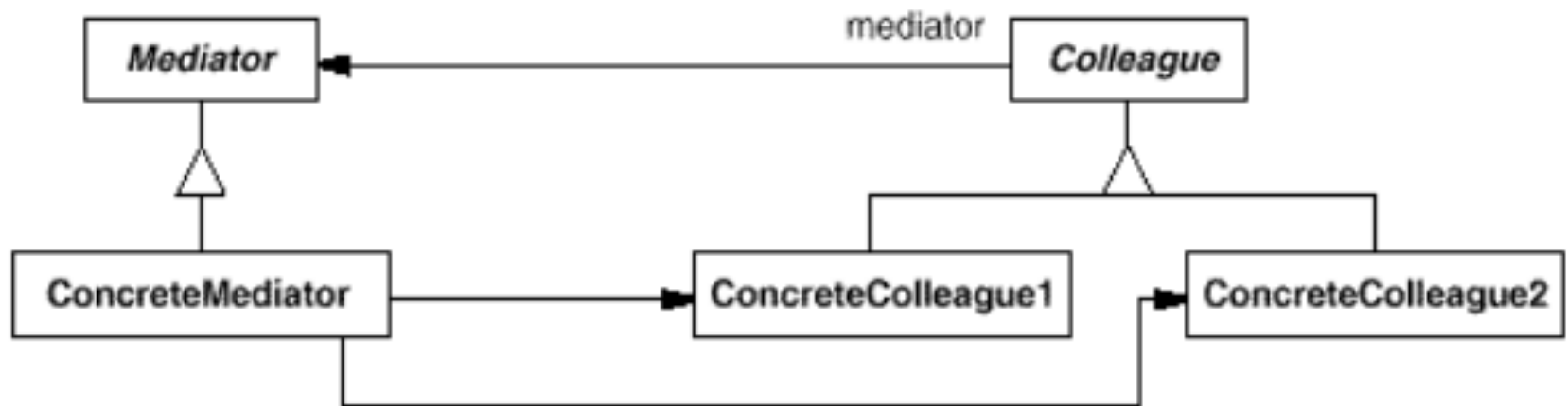
Exercícios [22]

- 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 quando: 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

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)

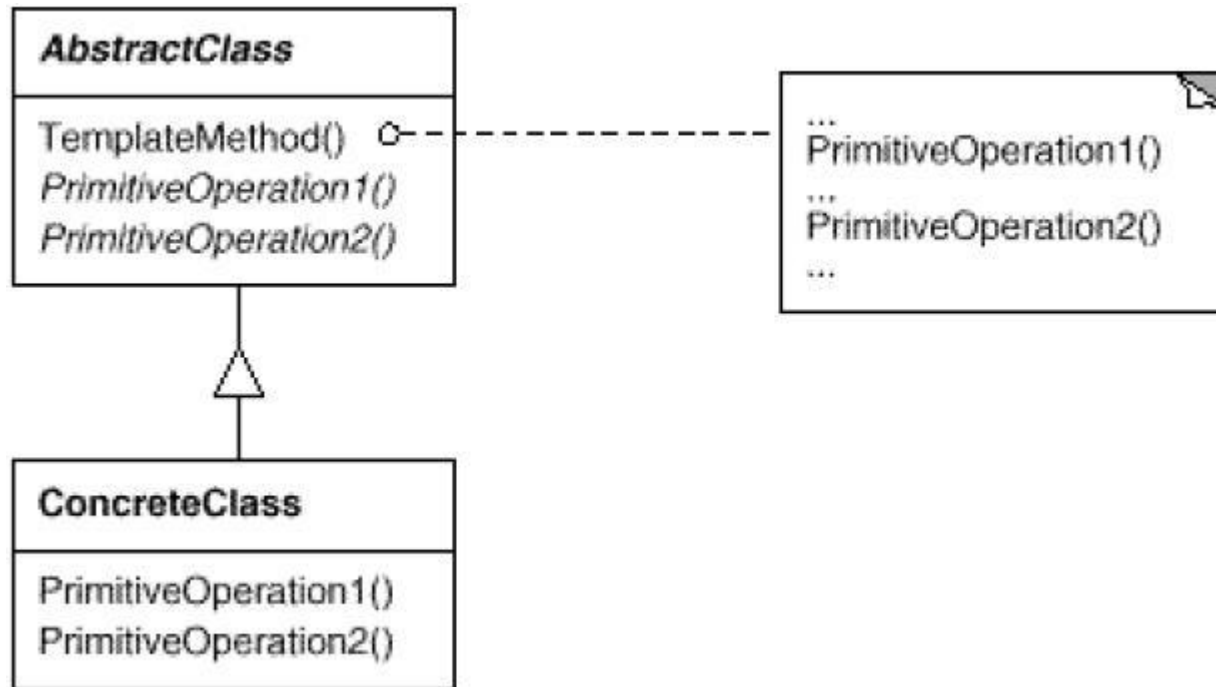
29. Definir um objeto que encapsula a forma como um conjunto de objetos interage. Promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente. Qual opção abaixo corresponde à descrição anterior?

- 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)

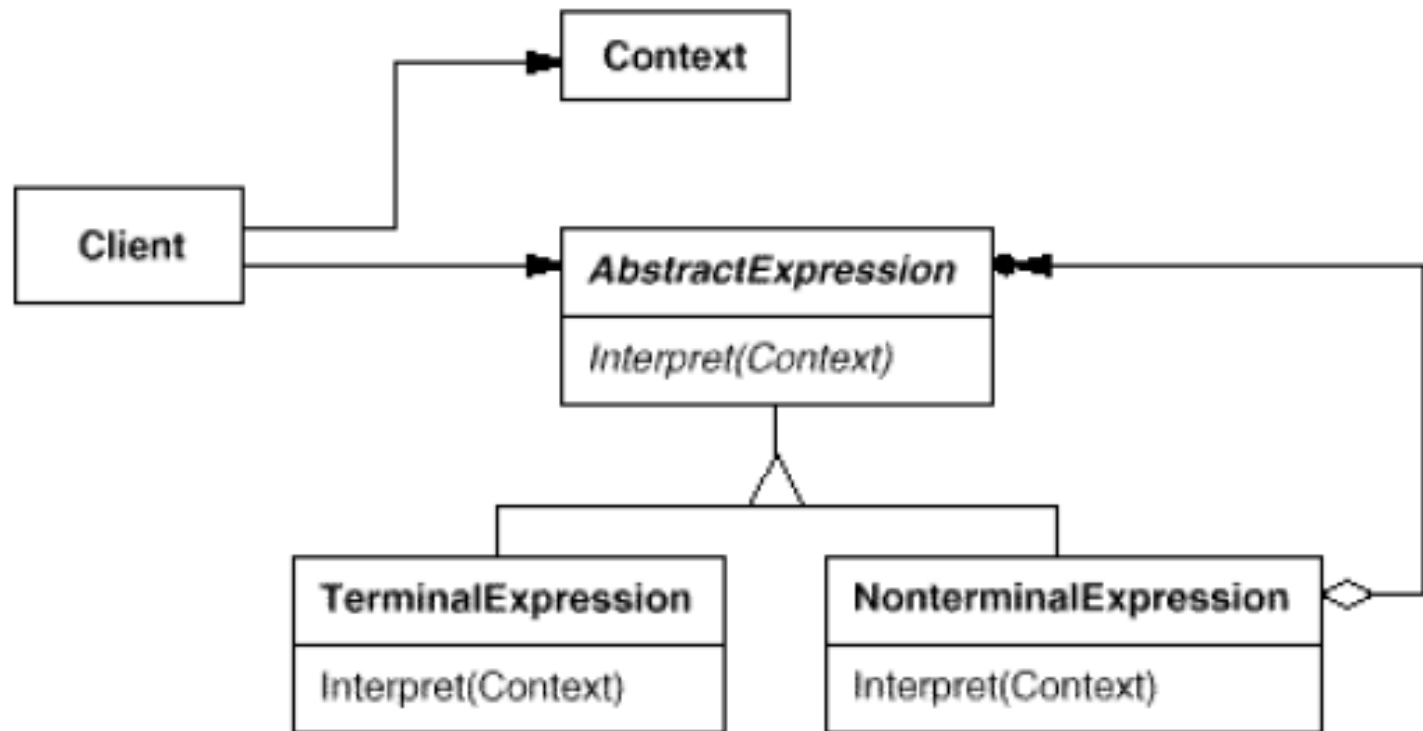
[33] Um dos participantes da equipe de desenvolvimento de um framework deve implementar uma operação em uma das classes desse framework. Seja X o nome dessa classe. Essa operação implementa um algoritmo em particular. Entretanto, há passos desse algoritmo que devem ser implementados pelos usuários do 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 frequência, 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, Decorator 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



Testes e Qualidade de Software

Fernando Pedrosa – fpedrosa@gmail.com

Bibliografia

- ▶ **Sommerville, Ian.** Software Engineering. **Editora:** Addison Wesley.
- ▶ **Pressman, Roger S.** Software Engineering: A Practitioner's Approach. **Editora:** McGraw–Hill.
- ▶ **RUP** – www.wthreex.com/rup
- ▶ **IEEE TC FTD/IFIP WG 10.4**

Contexto e Objetivo

- ▶ Uma vez que o código fonte tenha sido gerado é necessário testar para descobrir erros antes de entregar o software para o cliente
- ▶ O objetivo é projetar uma série de testes com máxima probabilidade de encontrar erros
- ▶ São utilizadas técnicas para:
 - Testar a lógica interna dos componentes
 - Testar as entradas e saídas das funções

Quem está envolvido?

- ▶ Durante os estágios iniciais do desenvolvimento, um Engenheiro de Software executa todos os testes
- ▶ Entretanto, à medida que o processo evolui, especialistas em testes podem ser envolvidos

Tipos de Manutenção de Software

- ▶ Corretiva
 - Correção de erros encontrados na verificação ou na validação
- ▶ Adaptativa
 - Adaptação a mudanças externas
- ▶ Melhoria (perfectiva)
 - Melhorias requeridas pelos usuários
- ▶ Preventiva ou de reengenharia
 - Abordagem pró-ativa com foco na melhoria da manutibilidade

Falta, Erro e Falha (IEEE)

▶ **Falta (Fault)**

- Causa de uma falha (aspecto físico)
- Exemplo: código incorreto ou faltando, defeito de hardware

▶ **Erro (Error)**

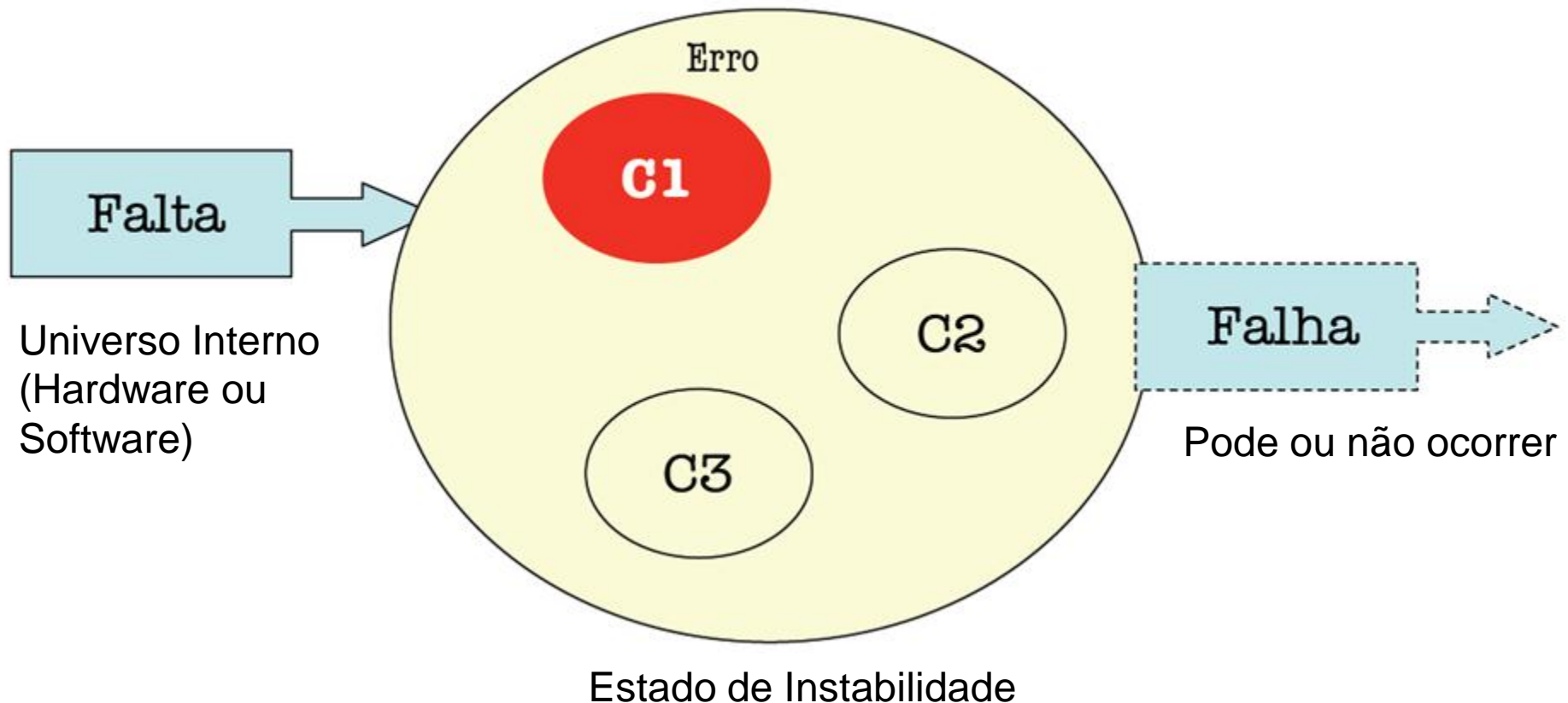
- Estado intermediário, de instabilidade (aspecto de informação)
- Pode resultar em falha, se propagado até a saída

▶ **Falha (Failure)**

- Incapacidade do software de realizar a função requisitada (aspecto externo). Manifestação observável.
- Exemplo: terminação anormal, restrição temporal violada



Falta, Erro e Falha (IEEE)



Falta, Erro e Falha (IEEE)

Como tornar os sistemas mais confiáveis?

- ▶ Prevenção de faltas
 - Especificação rigorosa
 - Proteção de Hardware
 - Ambientes e linguagens apropriados
- ▶ Tolerância a falhas
 - Replicação/Redundância
 - Isolamento do componente faltoso
 - *Hot swapping*

Exercícios [1]

(Min. Comunicações – CESPE 2008)

[109] Ao longo do desenvolvimento, artefatos produzidos podem ser revisados, objetivando garantir que os mesmos apresentem, pelo menos, a qualidade mínima especificada. Não apenas o código, mas também outros artefatos podem ser revisados. Os defeitos encontrados pelas revisões referem-se à faltas (fault), enquanto os defeitos encontrados por testes são falhas do software, pois testes avaliam a qualidade comparando o comportamento esperado com o observado.

Exercícios [1]

(TRE/BA – CESPE 2010)

[61] Segundo o IEEE, defeito é um ato inconsistente cometido por um indivíduo ao tentar entender determinada informação, resolver um problema ou utilizar um método ou uma ferramenta; erro é o comportamento operacional do software diferente do esperado pelo usuário, e que pode ter sido causado por diversas falhas; e falha é uma manifestação concreta de um defeito em um artefato de software, ou seja, é qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa.

Verificação e Validação

- ▶ É todo um processo de ciclo de vida que deve ser aplicado em cada estágio do desenvolvimento do software
- ▶ Tem dois objetivos principais:
 - O descobrimento de defeitos no sistema
 - A avaliação sobre se o sistema é útil e adequado em uma situação operacional
- ▶ V&V não garante que o software está livre de defeitos, mas apenas garante um certo nível de confiabilidade

Verificação

- ▶ Refere-se ao conjunto de atividades que garantem que o software implementa corretamente as funções especificadas

“Estamos construindo o produto de forma correta?”

“Are we building the Product Right?”

- ▶ Principais atividades
 - Inspeções (verificação estática)
 - Testes (verificação dinâmica)

Validação

- ▶ Refere-se ao conjunto de atividades que garantem que o software construído implementa o que o cliente realmente desejava

“Estamos construindo o produto certo?”

“Are we building the Right Product?”

- ▶ Principais atividades
 - Homologação, Testes de Aceitação (beta)
 - Revisões, etc.

Exercícios [2-A]

(MPOG – ESAF 2008)

[13-B] Demonstrar ao desenvolvedor e ao cliente que o software atende aos requisitos é uma meta de validação do software.

(IPEA – CESPE 2008)

[85] A verificação assegura que o produto, como fornecido, irá atender o seu uso pretendido, ou seja, que se está construindo o produto certo. E a validação confirma que os produtos de trabalho refletem de forma apropriada os requisitos que foram especificados, ou seja, que se está construindo o produto corretamente.

(TRT5 – CESPE 2009)

[75] A diferença entre verificação e validação reside no fato de que a primeira se refere ao conjunto de atividades que garante que o software realiza corretamente uma função específica, enquanto a segunda refere-se a um conjunto diferente de atividades que garante que o software que foi construído é rastreável às exigências do cliente.

Exercícios [2-A]

(MPU – FCC 2007)

[56] Considere as informações abaixo em relação ao desenvolvimento de sistemas:

- I. executar um software com o objetivo de revelar falhas, mas que não prova a exatidão do software.
- II. correta construção do produto.
- III. construção do produto certo.

Correspondem corretamente a I, II e III, respectivamente,

- a) validação, verificação e teste.
- b) verificação, teste e validação.
- c) teste, verificação e validação.
- d) validação, teste e verificação.
- e) teste, validação e verificação

Qualidade: Garantia x Controle (Padrão de mercado)

Garantia da Qualidade	Controle da Qualidade
Focada no <u>processo</u>	Focada no <u>produto</u>
Orientada a prevenção	Orientada a detecção
Exemplos: metodologias e padrões de desenvolvimento	Exemplos: checagem de requisitos, testes de software, etc.
Garante que você está fazendo as coisas da maneira correta	Garante que os <u>resultados</u> do seu trabalho estão de acordo com o esperado

Qualidade: Garantia x Controle (Sommerville & Pressman)

- ▶ Alguns autores não diferenciam estas atividades
- ▶ Para Sommerville inclui Validação e Verificação de Produtos, além dos Processos adequados
- ▶ Para Pressman inclui um espectro amplo de atividades, tais como padronização, revisões, testes, etc.

Exercícios [2-B]

(SERPRO – CESPE 2010)

[98] A garantia de qualidade tem como objetivo testar os produtos de software de modo a identificar, relatar e remover os defeitos encontrados, enquanto o controle da qualidade provê a gerência sênior da organização com a visibilidade apropriada sobre o processo de desenvolvimento.

(STJ – CESPE 2008)

[97] Um processo de gerenciamento da qualidade do projeto tipicamente visa garantir e controlar a qualidade. No controle da qualidade, são executadas atividades planejadas e sistemáticas visando garantir que o projeto empregará os processos necessários para atender aos requisitos. Por sua vez, a garantia da qualidade, diferentemente do controle de qualidade, monitora resultados do projeto a fim de determinar se eles estão de acordo com os padrões relevantes de qualidade e procura identificar meios para eliminar as causas de resultados que sejam insatisfatórios.

Inspeções de Software (verificação estática)

Inspeção de Software

- ▶ Se preocupa com a análise estática dos artefatos do sistema
- ▶ Envolve pessoas examinando os produtos de trabalho com o objetivo de encontrar anomalias e defeitos
- ▶ Pode ser suplementada por ferramentas CASE de análise estática

Inspeção x Teste

- ▶ Inspeções e Testes são técnicas complementares e não opostas
- ▶ Muitos defeitos podem ser encontrados durante uma inspeção – testes podem mascarar erros e necessitar de várias execuções
- ▶ Inspeções não podem checar requisitos não funcionais (desempenho, usabilidade, etc.)
- ▶ Inspeções checam a conformidade com a especificação e não com as reais necessidades do cliente

Vantagens

- ▶ Inspeções não necessitam da execução do sistema, portanto podem ser aplicadas antes da implementação
- ▶ Podem ser aplicadas a qualquer representação do sistema
 - Requisitos
 - Projeto,
 - Código, etc.
- ▶ Têm se mostrado uma técnica efetiva para se descobrir erros no programa

Dificuldades

- ▶ Inspeções aumentam o custo do processo de desenvolvimento, no início
- ▶ As equipes devem ser bem informadas e ter acesso a especificações precisas
- ▶ Padrões organizacionais devem ser bem definidos
- ▶ A gerência pode utilizar os achados de inspeção para avaliar (culpar) indivíduos

Inspeções do programa

- ▶ São abordagens formais para documentar as revisões do código
- ▶ A intenção é de **detecção de defeitos, apenas** (e não correção)
- ▶ Defeitos podem ser
 - Lógicos (caminhos incorretos, cálculos errados, etc.)
 - Anomalias no código (variáveis não inicializadas, laços incompletos, etc.)
 - Não conformidade com padrões

Inspeções manuais do programa

Procedimento:

- ▶ Um resumo do sistema é apresentado à equipe de inspeção
- ▶ Código e documentos associados são distribuídos à equipe com antecedência
- ▶ A inspeção acontece e os erros descobertos são anotados, de acordo com um *checklist*
- ▶ Uma nova inspeção pode ser necessária

Inspeções automatizadas

- ▶ Inspeções de código automatizadas utilizam **Analísadores Estáticos**
 - São ferramentas CASE que analisam o código fonte do sistema
 - Buscam e relatam erros em potencial
 - São bastante efetivos como um auxílio às inspeções, mas não a substituem
- ▶ Têm um melhor custo-benefício para linguagens fracamente tipadas, onde o compilador detecta poucos erros

Inspeções automatizadas

- ▶ Exemplos de possíveis análises:
 - Análise de controle de fluxo
 - Checa a codificação de loops, código inalcançável, etc.
 - Análise de dados
 - Detecta variáveis não inicializadas, variáveis que nunca são utilizadas, etc.
 - Análise de interfaces
 - Checa a consistência de declarações de rotinas e procedimentos e como são utilizados

Exercícios [3]

(STJ – CESPE 2008)

[73] Inspeções e walkthroughs podem fazer parte de um processo de verificação e validação, sendo realizadas por equipes cujos membros têm papéis definidos. Quando da inspeção de um código, uma lista de verificação de erros (checklist) é usada. O conteúdo da lista tipicamente independe da linguagem de programação usada.

(TSE – CESPE 2006)

[63–A] Inspeções e walkthroughs podem ser usadas para revisar artefatos. Uma walkthrough requer mais tempo de preparação dos revisores do que uma inspeção, também exige que seja feito o acompanhamento das soluções dos problemas identificados e a coleta de métricas associadas à revisão.

Exercícios [3]

(TSE – CESPE 2006)

[63–B] Em uma inspeção, os participantes têm papéis definidos. O moderador conduz reuniões e os inspetores devem, durante as reuniões, descrever os problemas identificados e soluções para os mesmos.

(DETRAN – CESPE 2009)

[97] O processo de validação tem por objetivo estabelecer com os clientes confiança quanto ao funcionamento adequado de um software. Enquanto inspeções de software ou revisões por pares são consideradas validação estática, o teste consiste em uma técnica dinâmica de validação de software. Os termos estático ou dinâmico são relativos à necessidade ou não do software ser executado.

Testes (verificação dinâmica)

Teste

- ▶ Processo de executar um programa com o **objetivo de encontrar erros.**
- ▶ Um bom caso de teste é aquele que tem alta probabilidade de achar um erro ainda não descoberto.
- ▶ Um teste de sucesso é aquele que descobre um erro ainda não descoberto.



O processo de testes **não** pode garantir que o software está livre de erros, ele pode apenas mostrar que erros e defeitos de software estão presentes.

Princípios

- ▶ Testes devem ser rastreáveis aos requisitos do cliente
- ▶ Testes devem ser planejados muito antes de serem executados
- ▶ O princípio de Pareto se aplica a Testes
 - 80% dos erros vão acontecer em 20% dos componentes
- ▶ Testes devem começar “pequenos” e progredir para pedaços maiores

Princípios

- ▶ Testes exaustivos não são possíveis
 - É impossível executar todos os caminhos existentes em um programa
- ▶ Para terem o máximo de efetividade, testes devem ser, preferencialmente, conduzidos por terceiros

O que constitui um “bom” teste?

- ▶ Um bom teste deve ter alta probabilidade de encontrar erros
- ▶ Um bom teste não deve ser redundante
 - Todo teste deve ter um propósito diferente
- ▶ Um bom teste deve ser “representativo” na sua classe
- ▶ Um bom teste não deve ser nem muito simples nem muito complexo [Pressman]

Abordagens de Testes

Abordagens de Testes

- ▶ Abordagem Funcional (“caixa preta”)
 - Focada nas entradas e saídas especificadas nos requisitos funcionais
- ▶ Abordagem Estrutural (“caixa branca”)
 - Focada nas estruturas internas dos procedimentos do sistema
- ▶ Abordagem Mista (“caixa cinza”)
 - É um meio termo entre caixa preta e branca
 - Algum conhecimento sobre as estruturas internas é utilizado para executar testes mais “bem informados”

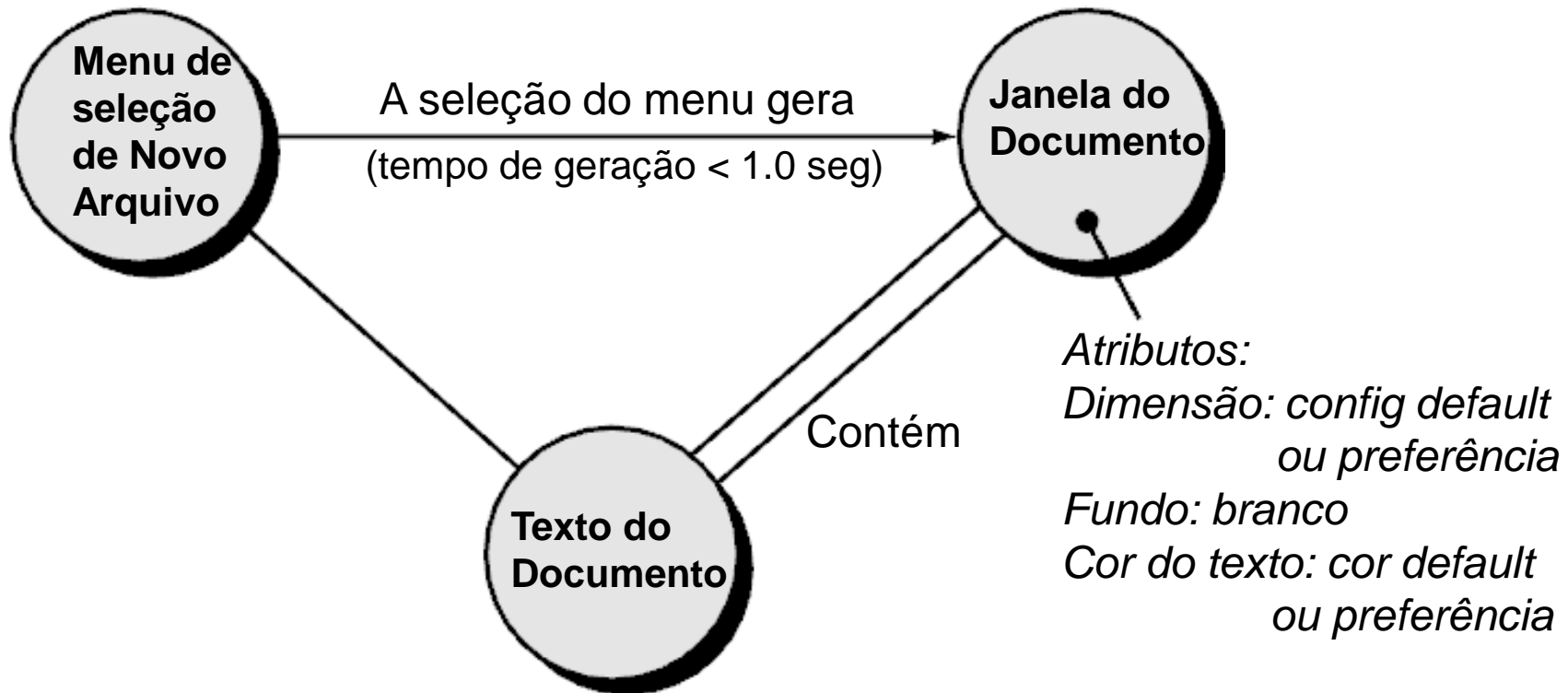
Abordagem Caixa Preta

- ▶ Baseada em prós e pós condições
- ▶ Geralmente utilizada nas etapas posteriores da disciplina de testes
- ▶ Busca erros nas seguintes categorias (dentre outras):
 - Funções incorretas ou inexistentes
 - Erros de comportamento ou desempenho
 - Erros de inicialização e término, erros de interface
- ▶ Principais técnicas: testes baseados em grafos, matriz ortogonal, análise de valores limítrofes, particionamento de equivalências

Testes baseados em grafos

- ▶ *Graph-based testing methods*
- ▶ Toda aplicação é construída por “objetos”
- ▶ A técnica identifica todos estes objetos e gera gráficos para representá-los
- ▶ Os objetos e relacionamentos são testados para descobrir erros e comportamentos inesperados

Testes baseados em grafos (exemplo: editor gráfico)



Particionamento de Equivalências

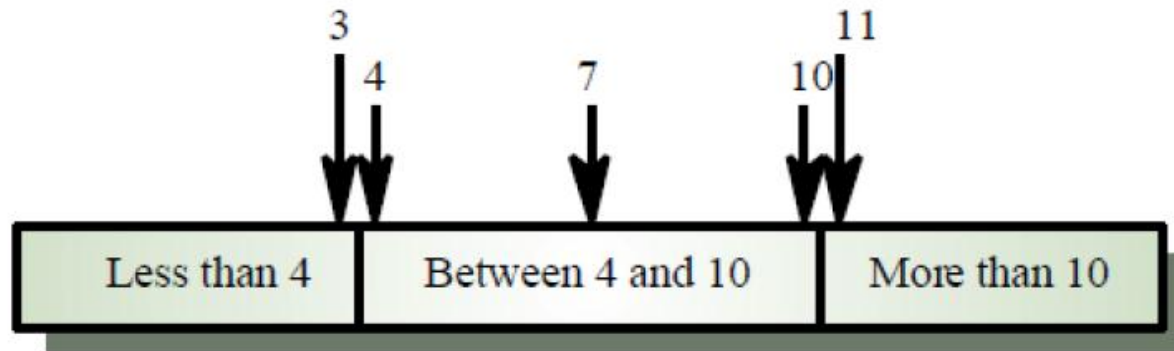
- ▶ Técnica baseada em dividir o domínio de entradas de um programa em classes de dados
- ▶ Cada classe de dados é chamada de “partição de equivalência”
- ▶ Casos de Teste são gerados considerando cada partição de equivalência

Particionamento de Equivalências

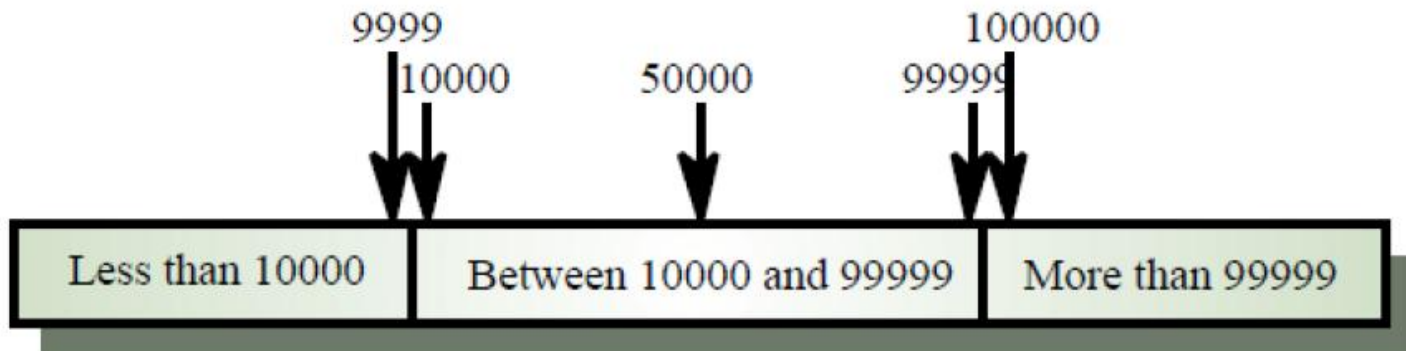
- ▶ Como guia, deve-se considerar entradas do tipo:
 - Faixa de Valores
 - 1 partição válida, 2 inválidas
 - Valor específico
 - 1 partição válida, 2 inválidas
 - Conjunto específico de valores
 - 1 partição válida, 1 inválida
 - Entrada *booleana*
 - 1 partição válida, 1 inválida

Particionamento de Equivalências

Exemplos (faixas de valores)



Number of input values



Input values

Análise de valores limítrofes

- ▶ É sabido que a maioria dos erros acontece nos limites do domínio de entrada, e não no “centro”
- ▶ Os testes devem ser gerados considerando esses valores “limítrofes”
 - Números máximos e mínimos
 - “Um a mais do que o máximo”
 - “Um a menos do que o mínimo”
- ▶ É utilizada em conjunto com a técnica de Particionamento de Equivalência

Exercícios [4]

(TRE/BA – CESPE 2010)

[79] Teste funcional é uma técnica para se projetar casos de teste na qual o programa ou sistema é considerado uma caixa-preta e, para testá-lo, são fornecidas entradas e avaliadas as saídas geradas.

(CEF – CESPE 2010)

[57-C] Relativamente ao modelo de caixa-cinza, o teste de software caixa-preta apresenta maior dependência no trabalho entre implementador e testador.

(CEF – CESPE 2010)

[57-D] O teste de software caixa-preta, relativamente ao modelo de caixa-cinza, apresenta maior dependência de conhecimento a respeito dos programas-fontes a serem testados.

Exercícios [4]

(TRT5 – CESPE 2008)

[74] Entre os tipos de testes de caixa preta, encontram-se o teste baseado em grafos; o particionamento de equivalência; a análise de valor-limite; e o teste de matriz ortogonal.

(MPOG – ESAF 2008)

[13-C] o particionamento de equivalência é uma maneira estratégica de aplicar testes de software.

Abordagem Caixa Branca

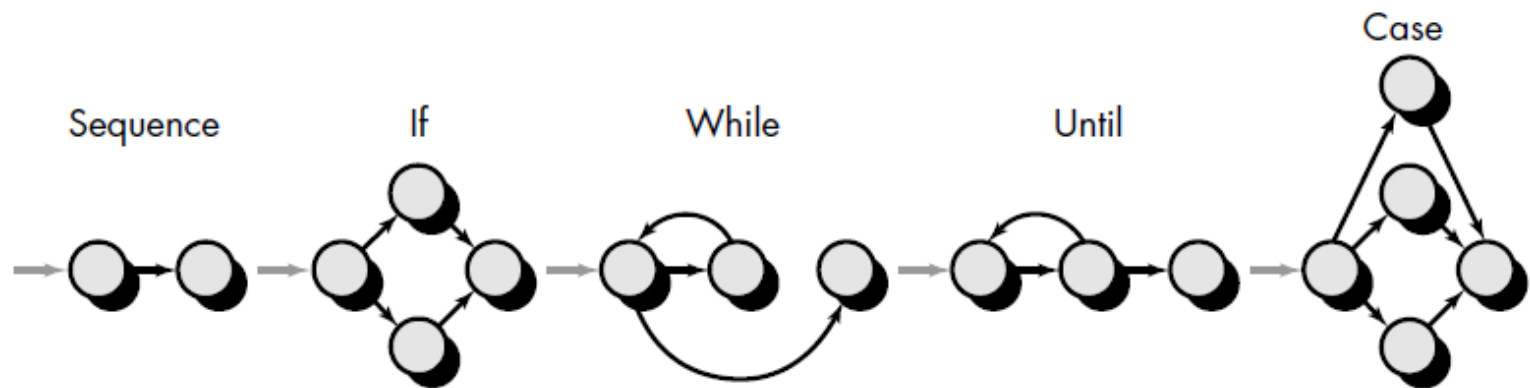
- ▶ Os testes são gerados a partir de uma análise dos caminhos lógicos possíveis de serem executados
- ▶ É necessário conhecimento do funcionamento interno dos componentes do software
- ▶ Objetivos
 - Garantir que todos os caminhos independentes de um módulo sejam executados pelo menos uma vez

Abordagem Caixa Branca

- ▶ **Objetivos (continuação)**
 - Realizar todas as decisões lógicas para valores falsos e verdadeiros
 - Executar laços dentro dos valores limites
 - Avaliar as estruturas de dados internas
- ▶ **Principais técnicas**
 - Testes de caminhos
 - Testes de estruturas de controle (laços, estruturas condicionais, etc.)
 - Complexidade ciclomática (métrica)

Complexidade Ciclomática

- ▶ Métrica que fornece uma medida quantitativa da complexidade lógica de um programa
- ▶ Quando usada no contexto de testes caixa branca, denota o número de caminhos possíveis dentro de um módulo
 - Nos dá uma idéia do limite superior necessário!



Exercícios [5]

(SERPRO – CESPE 2010)

[85] Com relação ao emprego de técnicas para a realização de testes de software, é correto afirmar que haverá maior diminuição da dependência de acesso às especificações arquiteturais de um sistema se o testador empregar a técnica de caixa-branca (white-box), em vez das técnicas de caixa-cinza (gray-box) e de caixa-preta (black-box).

(CEF – CESPE 2010)

[57-B] O aumento na medida de complexidade ciclomática de um programa introduz mudanças significativas no refinamento de uma abordagem do tipo caixa-preta.

Exercícios [5]

(CEF – CESPE 2010)

[57-E] À medida que avança o nível de integração dos módulos de um software, mais viável se torna a adoção do método de caixa-branca para desenho do teste de software.

(CEF – CESPE 2010)

[57-A] Para aderência à abordagem de software caixa-branca, podem ser empregados testes de fluxo de controle e de dados, que não são apoiadores diretos em testes de caixa-preta.

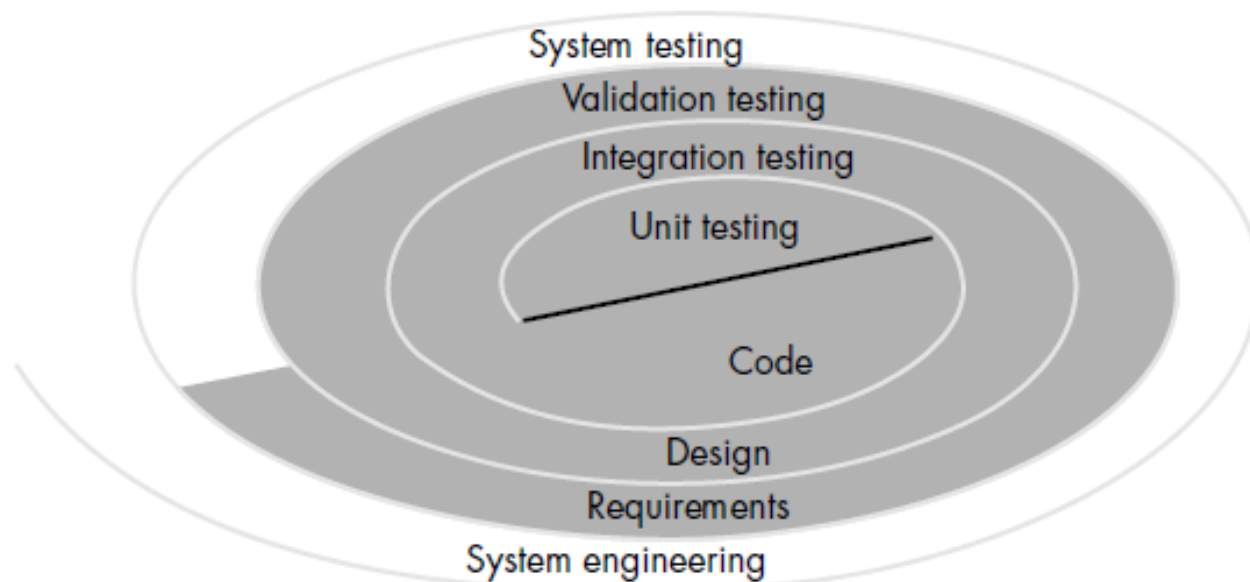
(MPOG – ESAF 2008)

[13-D] O teste estrutural é uma estratégia que se baseia na análise da especificação de um programa para ajudar na seleção de casos de teste.

Estágios (estratégias) de Testes

Estágios (estratégias) de Teste

- ▶ Os testes geralmente são agrupados de acordo com o momento no qual eles são executados ou pelo nível de especificidade do teste



Teste de Unidade

- ▶ Primeiro nível de testes, onde componentes individuais são testados para assegurar que os mesmos operam de forma correta
- ▶ Componentes podem ser:
 - Métodos individuais dentro de um objeto
 - Classes com vários atributos e métodos
 - Componentes que tenham sua estrutura interna bem conhecida
- ▶ Ferramenta mais utilizada (Java): JUnit

JUnit

- ▶ Framework de código aberto, que provê o ambiente necessário para testar códigos em Java
- ▶ A maioria das IDEs (Eclipse, Netbeans, JDeveloper, etc.) incorpora-o dentro do seu ambiente, tornando o uso mais fácil

Componentes de um teste JUnit

- ▶ Classe a ser testada
 - `Calculadora.java`
- ▶ Classe contendo os casos de teste
 - É quem realmente executa os testes na classe a ser testada
 - `CalculadoraTest.java`
- ▶ Chamador (driver) dos testes
 - Chama as classes contendo os testes
 - Normalmente é utilizado automaticamente através das IDE's

Exemplo

Classe a ser testada

```
public class Calculadora {
```

```
    public static int somar (int a, int b) {  
        return a + b;  
    }
```

```
    public static int multiplicar (int a, int b) {  
        return a * b;  
    }
```

```
    ...
```

```
}
```

```
import junit.framework.*;
```

```
public class CalculadoraTest extends TestCase {
```

```
    public void testSomar() {
```

```
        int a = 1;
```

```
        int b = 2;
```

```
        int total = 3;
```

```
        int resultado = 0;
```

```
        resultado = Calculadora.somar(a,b);
```

```
        assertEquals(resultado,total);
```

```
    }
```

Classe contendo os
casos de teste

Assertiva

Exercícios [6]

(PETROBRAS – CESGRANRIO 2010)

[13-A] A utilização de testes unitários faz com que seja desnecessária a fase de testes de aceitação, pois é garantido que o software passou por todos os testes de funcionamento necessários.

(PETROBRAS – CESGRANRIO 2010)

[13-B] A utilização de testes unitários ajuda a verificar se alterações introduzidas por um dos membros de uma equipe de tamanho médio ou grande não causaram efeitos colaterais em módulos de outros membros da equipe.

Exercícios [6]

(SERPRO – CESPE 2010)

[83] A atividade de teste unitário de software é, conforme os modelos de ciclo de vida de software vigentes, realizada de forma mais eficaz no escopo de implementação e da construção de software – nas quais a codificação de uma unidade executável de software é feita –, quando comparada à situação em que o teste unitário é realizado simultaneamente ao teste de integração.

(TRE/MT – CESPE 2010)

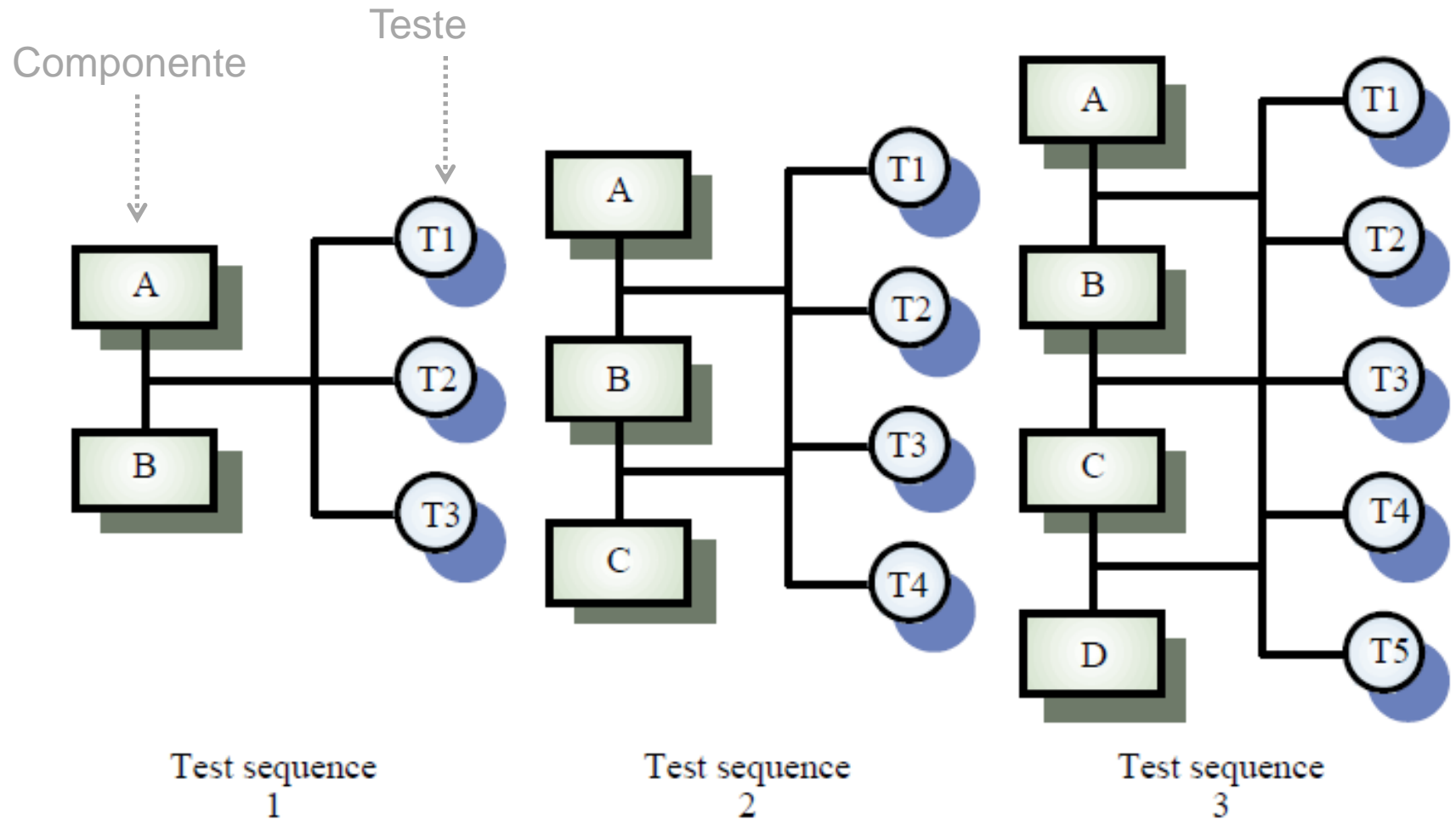
[40–A] JUnit é um framework open–source (arcabouço livre) para escrever e executar testes automaticamente, sem necessidade de escrever código adicional.

[40–C] A classe AssertEquals(a,b) compara dois valores. O teste é executado com sucesso se a.equals(b).

Teste de Integração

- ▶ Integra e testa os componentes de um sistema com o objetivo de encontrar problemas durante suas interações
- ▶ Integração Top-Down
 - Desenvolve o esqueleto do sistema e o preenche com os componentes do sistema
- ▶ Integração Bottom-Up
 - Integra os componentes de infraestrutura e depois adiciona componentes funcionais
- ▶ Para facilitar a localização de erros, sistemas devem ser integrados gradualmente

Integração gradual



Exercícios [7]

(IPEA – CESPE 2008)

[58] Os testes de integração verificam se os componentes do sistema funcionam em conjunto, se os componentes são chamados corretamente e se os componentes transferem dados corretos via suas interfaces. Nesses testes, os componentes são testados interligados; podem ser necessários drivers e stubs para simular componentes ainda não implementados; e, em sistemas de software orientados a objeto, os stubs podem ser classes.

(TRE/PR – CESPE 2009)

[83] Nos testes de integração, realizados antes dos testes unitários, os componentes são construídos e testados separadamente.

Teste de Aceitação (Validação)

- ▶ Ao final dos testes de integração, o software está consolidado e iniciam-se os testes de aceitação
- ▶ A finalidade é demonstrar a conformidade com os requisitos de software
- ▶ O ambiente utilizado deve ser o mais próximo possível do ambiente real
- ▶ Os mais comuns são Testes Alfa ou Beta

Teste Alfa

- ▶ É virtualmente impossível para o desenvolvedor prever como o software será utilizado pelo usuário
- ▶ São necessários testes conduzidos pelo cliente nas instalações do desenvolvedor
- ▶ O desenvolvedor anota os erros e problemas que possam ocorrer
- ▶ Acontece em ambiente **controlado**

Teste Beta

- ▶ É conduzido em um ou mais locais do cliente, pelo usuário final do produto
- ▶ O desenvolvedor geralmente não está presente
- ▶ O usuário anota todos os problemas que possa encontrar e os envia para o desenvolvedor frequentemente
- ▶ Acontece em ambiente **real**

Teste de Sistema

- ▶ Software é apenas um elemento de todo um sistema computacional maior
- ▶ É necessário testá-lo no ambiente operacional, onde são considerados
 - Hardware e Pessoas
 - Processos e informações
 - Outros sistemas, etc.
- ▶ São conduzidos em um ambiente completo e integrado, por várias pessoas (não só os desenvolvedores)

Exercícios [8]

(TRE/MT – CESPE 2010)

[39–D] O teste alfa é conduzido pelo cliente em seu ambiente de uso final.

(TSE – CESPE 2006)

[55–C] Os testes são realizados em várias fases de um desenvolvimento. Testes de unidade são de baixo nível, testes de sistema são executados após os de integração, testes beta empregam apenas desenvolvedores.

(EPE – CESGRANRIO 2010)

[43] Um novo sistema de informação interno de uma empresa está sendo testado por um grupo restrito de usuários, fora do ambiente dos desenvolvedores. Isso caracteriza o teste

(A) de unidade. (B) de usabilidade.

(C) alfa. (D) beta.

(E) de stress.

Tipos de Testes

Teste de Regressão

- ▶ Cada vez que um módulo é adicionado ao sistema, o software muda
 - Novas entradas e saídas surgem
 - Podem ser executados novos caminhos
 - A lógica de controle muda
- ▶ Teste de regressão visa a executar um subconjunto de testes que já foram executados com o intuito de garantir que as mudanças não propagaram efeitos indesejados

Smoke Testing (Teste de Fumaça)

- ▶ O termo é usado em várias áreas e refere-se ao primeiro teste realizado depois de integrar os componentes
- ▶ Em Software, é aplicado após cada montagem (build) do produto para verificar sua funcionalidade básica
- ▶ O intuito deve ser o de encontrar erros que têm a maior probabilidade de atrasar o projeto (*show stopper errors*)

Teste de Recuperação

- ▶ Muitos sistemas devem se recuperar de falhas e voltar ao processamento dentro de um tempo pré-estabelecido
- ▶ Teste de recuperação força o software a falhar de várias formas e verifica que a recuperação foi feita de forma adequada
- ▶ A recuperação pode ser automática ou manual

Teste de Segurança

- ▶ Tem o objetivo de verificar que mecanismos de proteção implementados no sistema irão, de fato, protegê-lo de ataques
- ▶ O testador tenta penetrar no sistema
- ▶ Dado tempo suficiente, um teste de segurança irá penetrar o sistema
 - É papel do desenvolvedor do sistema assegurar que isto custe mais caro que os ganhos obtidos

Teste de Carga (estresse)

- ▶ Inicialmente, testes caixa branca e caixa preta tem o intuito de testar o sistema sob condições normais
- ▶ Testes de carga visam a confrontar programas com situações anormais
 - Têm caráter destrutivo
 - Até onde ele aguenta?
- ▶ Podem ser estressados
 - Memória (vários objetos criados)
 - I/O (muitas interrupções por segundo)
 - Disco (busca por dados), etc.

Teste de Desempenho

- ▶ Pode ocorrer durante todos os estágios de testes
- ▶ Visa a garantir que o sistema atende aos níveis de desempenho e tempo de resposta acordados com o usuário e definidos nos requisitos

Teste de Usabilidade

- ▶ Avalia o sistema do ponto de vista do usuário final
- ▶ Enfatiza o seguinte:
 - Fatores humanos
 - Estética
 - Consistência na interface do usuário
 - Ajuda online e contextual
 - Assistentes e agentes (wizards)
 - Documentação do usuário
 - Material de treinamento

Exercícios [9]

(PRODEST – CESPE 2006)

[101] O XP é um processo que visa a um desenvolvimento ágil e portanto não recomenda os testes de unidade, pois eles consomem muitos recursos. Durante o desenvolvimento, o primeiro teste recomendado é o smoke test que foca os detalhes de funcionamento. O smoke test é realizado após as unidades serem integradas. Após o smoke test, é realizado o teste de sistema.

(TRT16 – FCC 2009)

[48] Há um tipo de teste que vislumbra a “destruição do programa” por meio de sua submissão a quantidades, frequências ou volumes anormais que é o teste

(A) de recuperação. (B) de configuração. (C) beta. (D) de desempenho. (E) de estresse.

Debugging

- ▶ É o processo que resulta na remoção de um erro encontrado
- ▶ Ocorre como uma consequência de um teste de sucesso (um teste que encontrou erros)
- ▶ Envolve formular uma hipótese sobre o comportamento do sistema e testar essa hipótese para achar os erros

O processo de debugar



Abordagens de debugging

▶ Força Bruta

- Popula-se o sistema com escritas ao console para tentar encontrar algum traço de erro durante a execução
- É o método menos eficiente de todos

▶ *Backtracking*

- Começando a partir de onde o erro ocorreu, deve-se rastrear o código manualmente até a fonte do erro

▶ Eliminação de causa

- Uma “hipótese de causa” é elaborada e os dados relacionados ao erro são utilizados para prová-la

Exercícios [10]

(TRE/MT – CESPE 2010)

[43] Existem várias maneiras de se depurar (debug) programas. Algumas delas envolvem conhecimento, prática e bom senso do programador. Acerca de pontos que são importantes para depurar programas, julgue os itens a seguir.

- I É possível encontrar falhas nos programas por meio da reprodução do erro em testes.
- II Quanto maior a entrada de dados nos testes, mais simples é encontrar o problema e mais fácil é encontrar a solução da falha.
- III Em um programa modular, o processo de encontrar falhas requer uma menor variação de informações de entrada, de modo que o programador possa encontrar o módulo com erros.
- IV A passagem de parâmetros para variáveis auxiliares evita o uso de Break points.

Exercícios [10]

V A análise estruturada é a melhor maneira de encontrar erros em programação orientada a objetos.

Estão certos apenas os itens

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

Testes segundo o RUP

Plano de Testes

- ▶ Define metas e objetivos dos testes no escopo da iteração (ou projeto)
- ▶ Explicita a abordagem adotada, os recursos necessários e os produtos liberados
- ▶ Determina a estrutura (framework) na qual os papéis de testes funcionarão
- ▶ Assim como em outros documentos, é necessário ganhar a aprovação e aceitação dos envolvidos

Caso de Teste

- ▶ Tem a finalidade de identificar e comunicar as condições específicas nas quais as funcionalidades serão testadas
- ▶ Define um conjunto de:
 - Entradas de teste
 - Condições de execução
 - Resultados esperados



Estrutura do Caso de Teste

- ▶ Descrição do Caso de Teste
 - Descreve o objetivo e o escopo do teste

Condições de execução:

- ▶ Pré-Condições
 - É o estado obrigatório do sistema antes do início do teste
- ▶ Entradas
 - Estímulos específicos aplicados durante o teste
- ▶ Pontos de observação
 - Observações específicas a serem feitas

Estrutura do Caso de Teste

- ▶ Pontos de Controle
 - Identifica os pontos em que pode ocorrer mudança do fluxo de controle
- ▶ Resultados esperados
 - Condições observáveis esperadas após a execução do teste. Pode incluir resposta positivas e negativas (erros, falhas, etc.)
- ▶ Pós-condições
 - Estado ao qual o sistema deve retornar para permitir a execução dos testes subsequentes

Caso de Teste a partir de Caso de Uso

- ▶ É necessário desenvolver casos de teste para cada cenário do caso de uso
- ▶ Cenário (instâncias do caso de uso): caminhos que percorrem o fluxo básico, alternativo, de exceção, etc.



Caso de Teste a partir de Caso de Uso

- ▶ Após cada caminho possível do caso de uso mostrado, é possível identificar cenários do caso de uso através da combinação do fluxo básico com fluxos alternativos

Cenário 1	Fluxo Básico			
Cenário 2	Fluxo Básico	Fluxo Alternativo 1		
Cenário 3	Fluxo Básico	Fluxo Alternativo 1	Fluxo Alternativo 2	
Cenário 4	Fluxo Básico	Fluxo Alternativo 3		
Cenário 5	Fluxo Básico	Fluxo Alternativo 3	Fluxo Alternativo 1	
Cenário 6	Fluxo Básico	Fluxo Alternativo 3	Fluxo Alternativo 1	Fluxo Alternativo 2
Cenário 7	Fluxo Básico	Fluxo Alternativo 4		
Cenário 8	Fluxo Básico	Fluxo Alternativo 3	Fluxo Alternativo 4	

Caso de Teste a partir de Caso de Uso

- ▶ É possível obter casos de teste para cada cenário através da identificação da condição específica que causará a execução deste cenário

Exemplo (fluxo alternativo 3):

“Ocorrerá esse fluxo de eventos se o valor digitado no Passo 2 acima, "Digitar o Valor da Retirada" for maior que o saldo atual da conta. O sistema exibirá uma mensagem de aviso e, depois, retornará ao Passo 2 do fluxo básico "Digitar o Valor da Retirada" acima, onde o cliente do banco poderá digitar um novo valor de retirada”

Caso de Teste a partir de Caso de Uso

- ▶ Com estas informações, podemos especificar alguns casos de teste para o fluxo alternativo número 3

ID de Caso de Teste	Cenário	Condição	Resultado Esperado
TC x	Cenário 4	Passo 2 - Valor da Retirada > Saldo da Conta	Retorna ao Passo 2 do fluxo básico
TC y	Cenário 4	Passo 2 - Valor da Retirada < Saldo da Conta	Não executa o Fluxo Alternativo 3, segue o fluxo básico
TC z	Cenário 4	Passo 2 - Valor da Retirada = Saldo da Conta	Não executa o Fluxo Alternativo 3, segue o fluxo básico

Na prática, teríamos uma lista de TC's parecida com isso...

ID do TC	Cenário/Condição	Senha	No da Conta	Valor Digitado (ou escolhido)	Valor na Conta	Valor no Caixa Eletrônico	Resultado Esperado
CW1.	Cenário 1 - Retirada em Dinheiro Bem-sucedida	4987	809 - 498	50.00	500.00	2,000	Retirada em dinheiro bem-sucedida. Saldo da conta atualizado para 450,00
CW2.	Cenário 2 - Caixa Eletrônico sem Dinheiro	4987	809 - 498	100,00	500,00	0,00	Opção Retirada em Dinheiro indisponível, fim do caso de uso
CW3.	Cenário 3 - Fundos insuficientes no caixa eletrônico	4987	809 - 498	100,00	500,00	70,00	Mensagem de aviso, retorno ao Passo 6 do Fluxo Básico - Digitar o Valor
CW4.	Cenário 4 - Senha Incorreta (> 1 nova tentativa)	<u>4978</u>	809 - 498	n/a	500,00	2.000	Mensagem de aviso, retorno ao Passo 4 do Fluxo Básico, Digitar a Senha
CW5.	Cenário 4 - Senha Incorreta (= 1 nova tentativa)	<u>4978</u>	809 - 498	n/a	500,00	2.000	Mensagem de aviso, retorno ao Passo 4 do Fluxo Básico, Digitar a Senha
CW6.	Cenário 4 - Senha Incorreta (= sem novas tentativas)	<u>4978</u>	809 - 498	n/a	500,00	2.000	Mensagem de aviso, cartão retido, fim do caso de uso

Exercícios [1 1]

SAD/PE (CESPE 2010)

[56] A respeito do plano de teste, um registro do processo de planejamento de testes de software, assinale a opção correta.

- A) O processo de planejamento de testes é usualmente descrito em um plano de testes.
- B) Um plano de teste de software é um registro da execução de um caso de teste de software.
- C) A automação de um teste de integração é mais facilmente empreendida que a de um teste de módulo.
- D) A produção de scripts de teste deve preceder a eventual construção de casos de teste.
- E) Ao se inspecionar o conteúdo de um plano de testes, devem-se encontrar, entre outras, as seguintes descrições: escopo de testes, abordagens de teste, recursos para realização dos testes e cronograma das atividades de teste a serem realizadas.

Gabarito dos Exercícios

[1] – [109] C, [61] E

[2-A] – [13-B] E, [85] E, [75] C, [56] C

[2-B] – [98] E, [97] E

[3] – [73] E, [63-A] E, [63-B] E, [97] C

[4] – [79] C, [57-C] E, [57-D] E, [74] C, [13-C] E

[5] – [85] E, [57-B] E, [57-E] E, [57-A] C, [13-D] E

[6] – [13-A] E, [13-B] E, [83] C, [40-A] E, [40-C] E

[7] – [58] C, [83] E

[8] – [39-D] E, [55-C] E, [43] D

[9] – [101] E, [48] E

[10] – [43] B

[11] – [56] E

FIM



UML

Fernando Pedrosa – fpedrosa@gmail.com

Bibliografia

- ▶ **Boch, Jacobson, Rumbaugh; UML – Guia do Usuário; Editora: Elsevier; Ano: 2006**
- ▶ **Martin Fowler; UML Essencial; Editora: Bookman; Ano: 2004**

Unified Modeling Language

Linguagem de Modelagem Unificada

▶ Linguagem

- Usada para expressar e comunicar idéias
- Não é uma metodologia!

▶ Modelagem

- Descrever um sistema em um alto nível de abstração

▶ Unificada

- UML se tornou o padrão mundial para modelagem de sistemas – www.omg.org

O que é a UML?

- ▶ Linguagem gráfica para especificar, visualizar, construir e documentar os artefatos de software
- ▶ Vantagens
 - Usa notação gráfica: mais clara que a linguagem natural (imprecisa) e código (muito detalhado)
 - Ajuda a obter uma visão geral do sistema
 - **Não** é dependente de tecnologia
 - Diminui a fragmentação, aumenta a padronização

Evolução



Industrialização

Padronização

Unificação

Fragmentação

Ano Versão

2011: UML 2.4

2010: UML 2.3

2009: UML 2.2

2003: UML 2.0

2001: UML 1.4

1999: UML 1.3

1997: UML 1.0, 1.1

1996: UML 0.9 & 0.91

1995: Unified Method 0.8

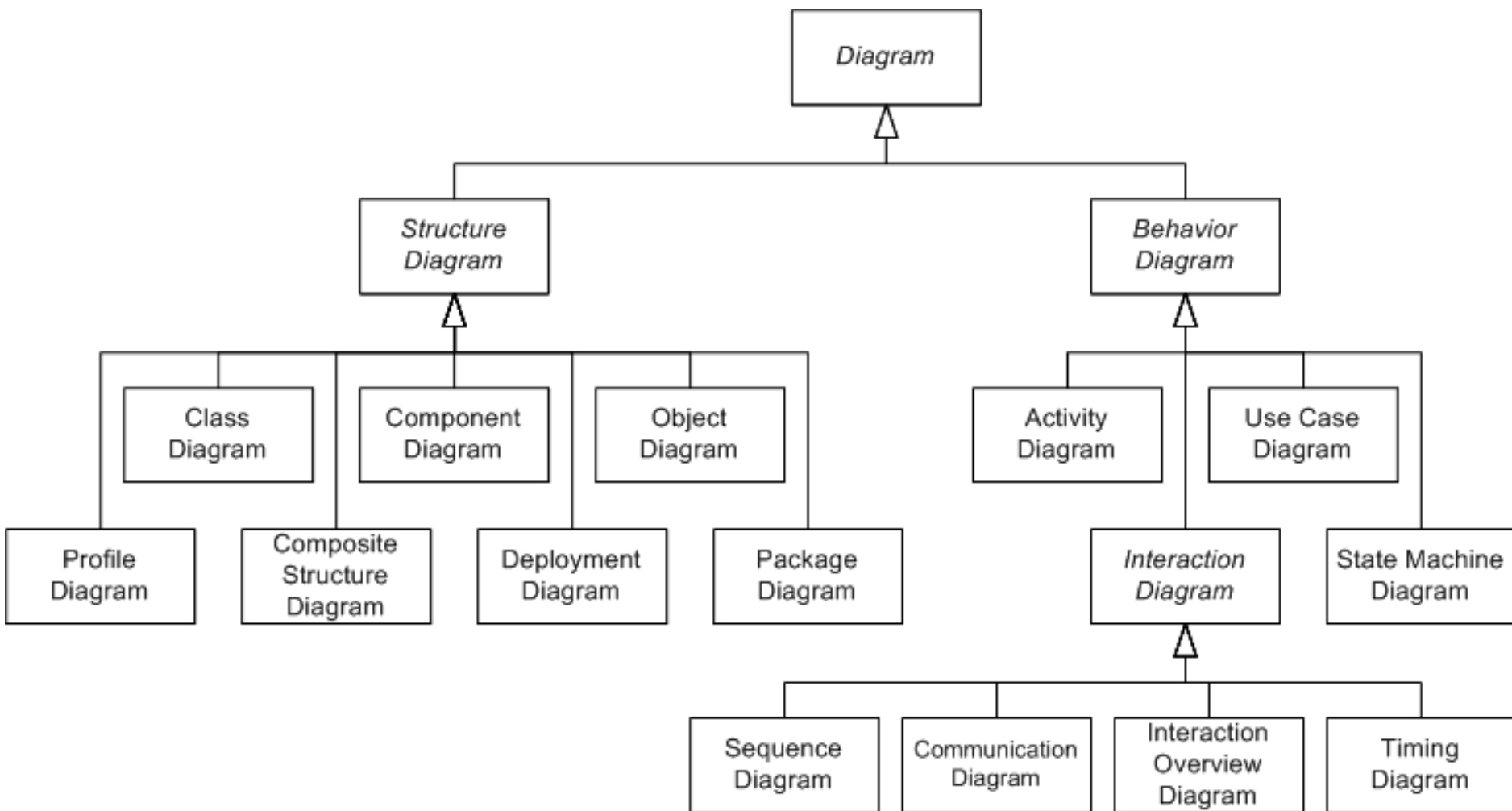
Outros
métodos

Booch (OOAD)

OMT (Rumbaugh)

OOSE (Jacobson)

UML 2.2



Tipos de Diagramas

▶ Diagramas estruturais

- Mostram a estrutura estática do sistema e suas partes em diferentes níveis de abstração e como elas se relacionam
- Não utilizam conceitos relacionados ao tempo

▶ Diagramas comportamentais

- Mostram a natureza dinâmica dos objetos do sistema, que pode ser descrita como uma série de mudanças no sistema com o passar do tempo

Exercícios [1]

(Governo do ES – CESPE 2009)

[78] UML é um método para desenvolvimento de software que foi proposto para ser aplicado à análise e projeto de software orientados a objetos.

(EMBASA – CESPE 2009)

[94] Os diagramas em UML podem ser estáticos ou dinâmicos. O diagrama de classes é um exemplo de um diagrama dinâmico.

(SERPRO – CESPE 2008)

[101] UML (universal modelling language) é uma linguagem de modelagem proprietária que pode ser utilizada no desenvolvimento de sistemas de maneira intuitiva para visualização de objetos.

Exercícios [1]

(CGU – ESAF 2008)

[31] – A linguagem de Modelagem Unificada (UML) emergiu como notação de diagramação de padrão, de fato e de direito, para a modelagem orientada a objetos. Desta forma, a sentença que conceitua apropriadamente a UML, segundo o OMG–Object Management Group, é

- a) um método para especificar e modelar os artefatos dos sistemas.
- b) um processo de especificação e modelagem de sistemas orientados a objeto.
- c) uma linguagem para implementar os conceitos da orientação a objetos.
- d) uma linguagem visual para especificar, construir e documentar os artefatos dos sistemas.
- e) um método comum para a representação da orientação a objetos.

Diagramas Estruturais (estáticos)

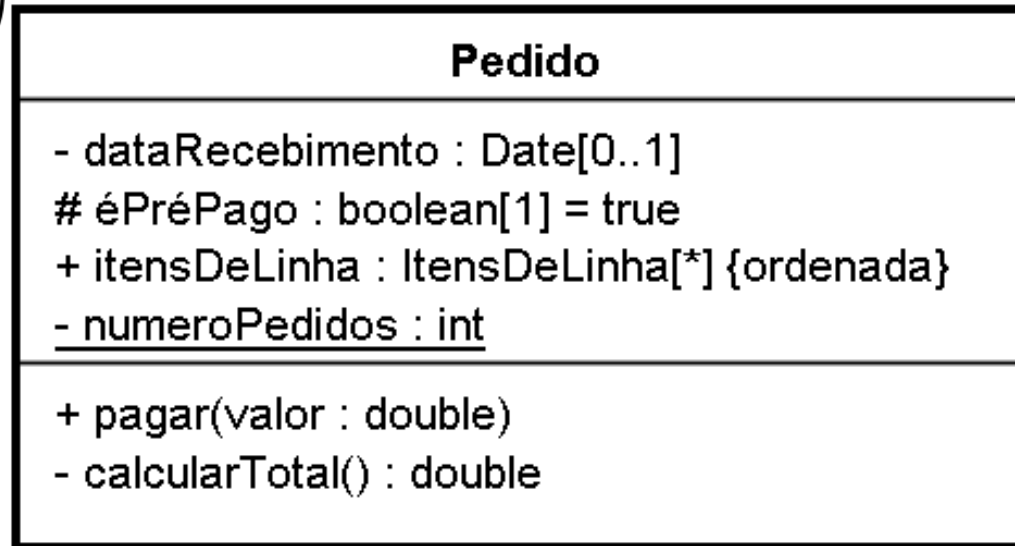
- ▶ Diagrama de Classes
- ▶ Diagrama de Objetos
- ▶ Diagrama de Componentes
- ▶ Diagrama de Pacotes
- ▶ Diagrama de Implantação
- ▶ Diagrama de Estrutura Composta
- ▶ Diagrama de Perfis (UML 2.2)

Diagrama de Classes

- ▶ É um diagrama estático da UML que reúne os elementos mais importantes de um sistema orientado a objetos
- ▶ Exibe um conjunto de classes, interfaces e seus relacionamentos
- ▶ As classes especificam tanto as propriedades quanto os comportamentos dos objetos

Estrutura da Classe

Propriedades
(Atributos)

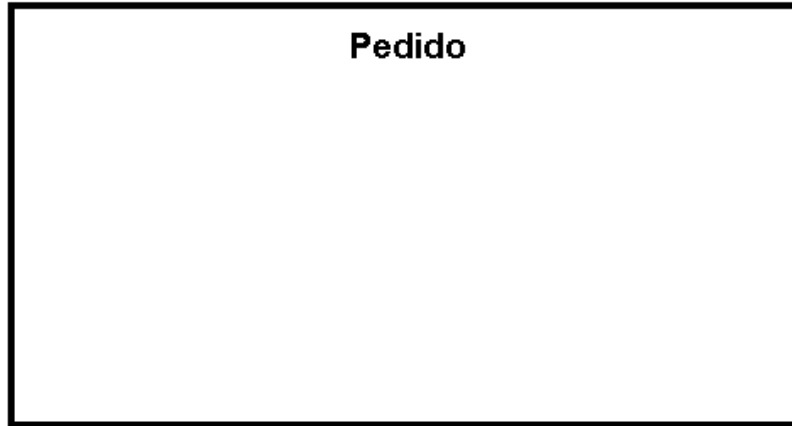


Nome da Classe

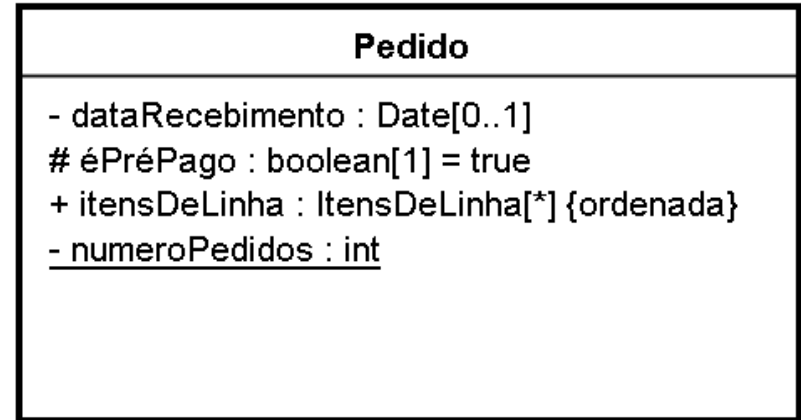
Operações

Três formas de representação

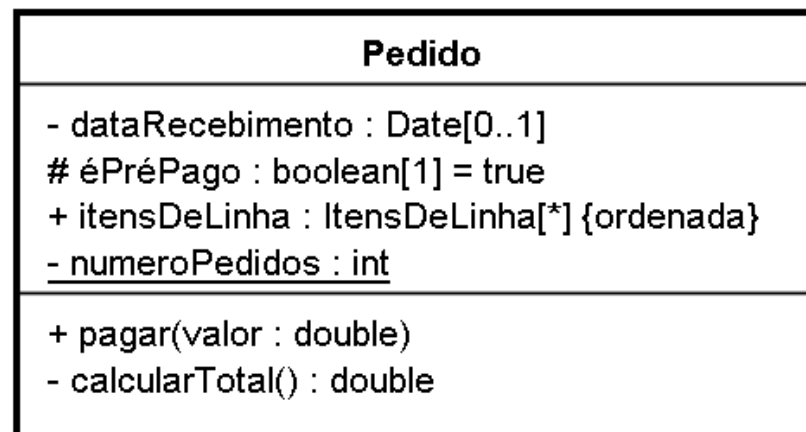
Nome

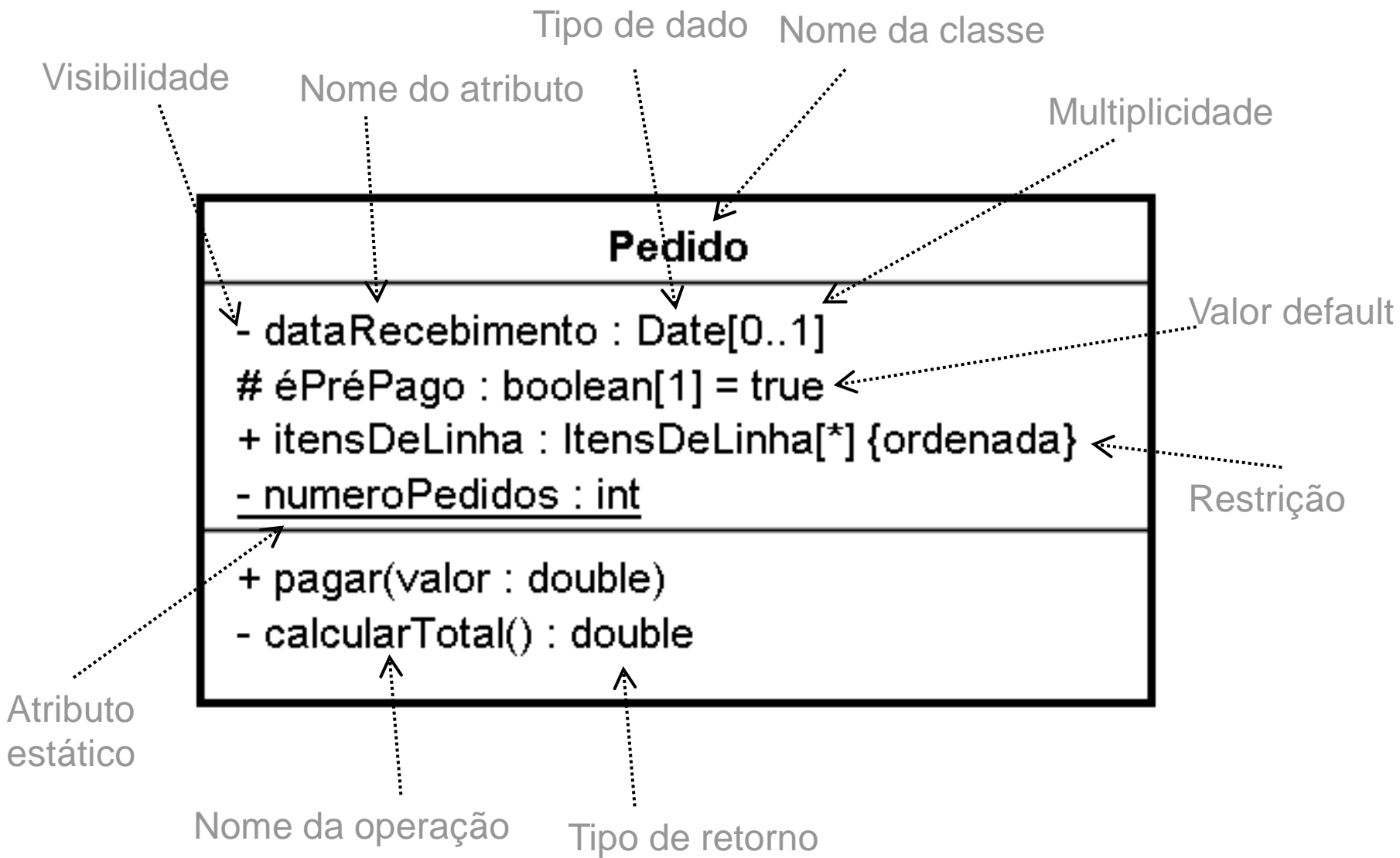


Nome + Atributos



Nome + Atributos + Operações





Atributos

Notação completa:

Visibilidade nome: tipo [multiplicidade] = valor_default {lista de restrições}

Nome: corresponde ao nome do atributo

Tipo: domínio do atributo

Multiplicidade: indicação de quantos objetos podem preencher a propriedade [min..max]

Valor Default: valor do atributo, caso ele seja omitido no momento da criação

Restrição: permite indicar propriedades adicionais.
{readOnly}, {ordered}, {unique}, etc.

Atributos – Escopo

- ▶ As propriedades (atributos) podem ter dois tipos de escopo
 - **Escopo de instância:** cada objeto tem o seu próprio valor para o atributo. É o escopo *default* da UML.
 - **Escopo de classe (estático):** o valor do atributo é comum a todos os objetos daquela classe. Para denotar este escopo o atributo deve ser sublinhado.

Operações

Notação completa:

Visibilidade nome (lista de parâmetros): tipo-de-retorno {lista restrições}

Nome: corresponde ao nome da operação

Lista de parâmetros: são os parâmetros da operação.

Tipo de retorno: o tipo de dado retornado pela operação

Restrição: permite indicar propriedades adicionais.

ex: {query}.

Operações abstratas e estáticas

- ▶ Operações abstratas, ou seja, que não têm uma implementação específica, devem ser escritas em *itálico*
- ▶ Operações estáticas devem ser escritas com fonte sublinhada

Modificadores de Visibilidade

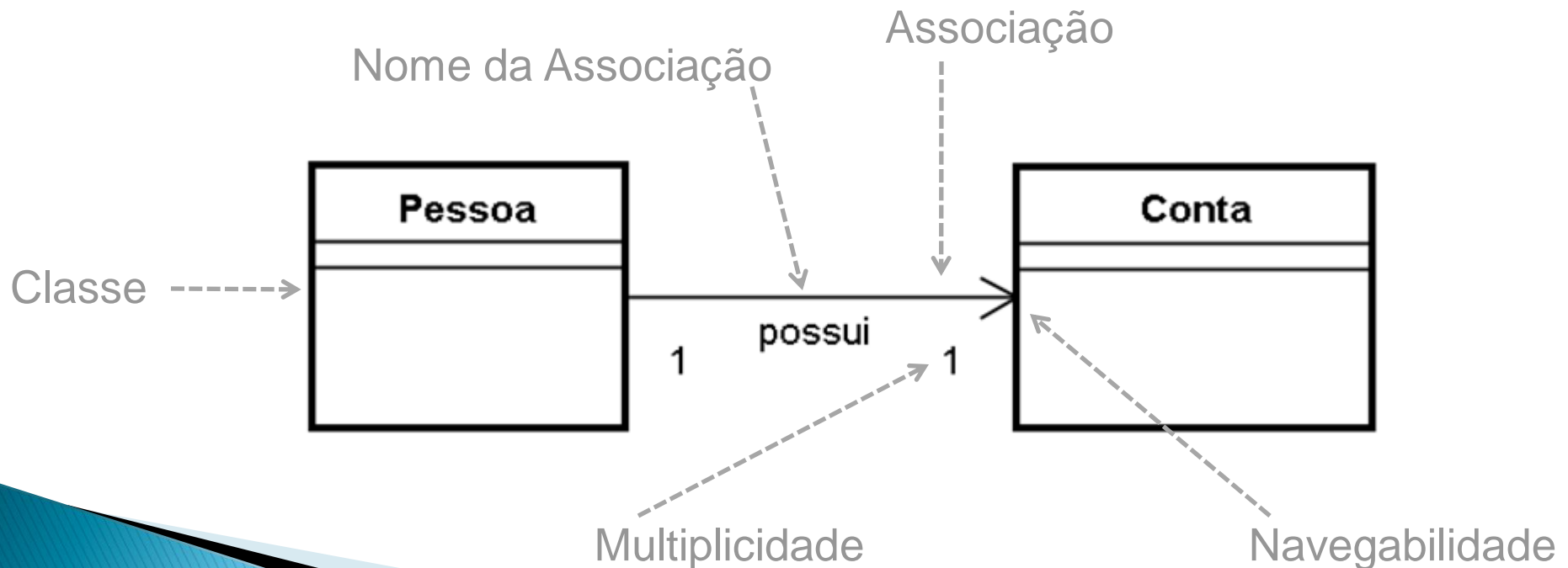
- ▶ Público (+)
 - O elemento é visível por qualquer classe
- ▶ Protegido (#)
 - O elemento é visível na própria classe e pelas subclasses da classe
- ▶ Pacote (~)
 - O elemento é visível apenas pela própria classe ou dentro do pacote onde a classe está localizada
- ▶ Privado (-)
 - O elemento é visível apenas pela própria classe

Relacionamentos

- ▶ Relacionamentos ligam classes entre si, criando relações lógicas
- ▶ Podem ser de:
 - **Associação**
 - Simples
 - Agregação
 - Composição
 - **Generalização**
 - **Dependência**
 - **Realização**

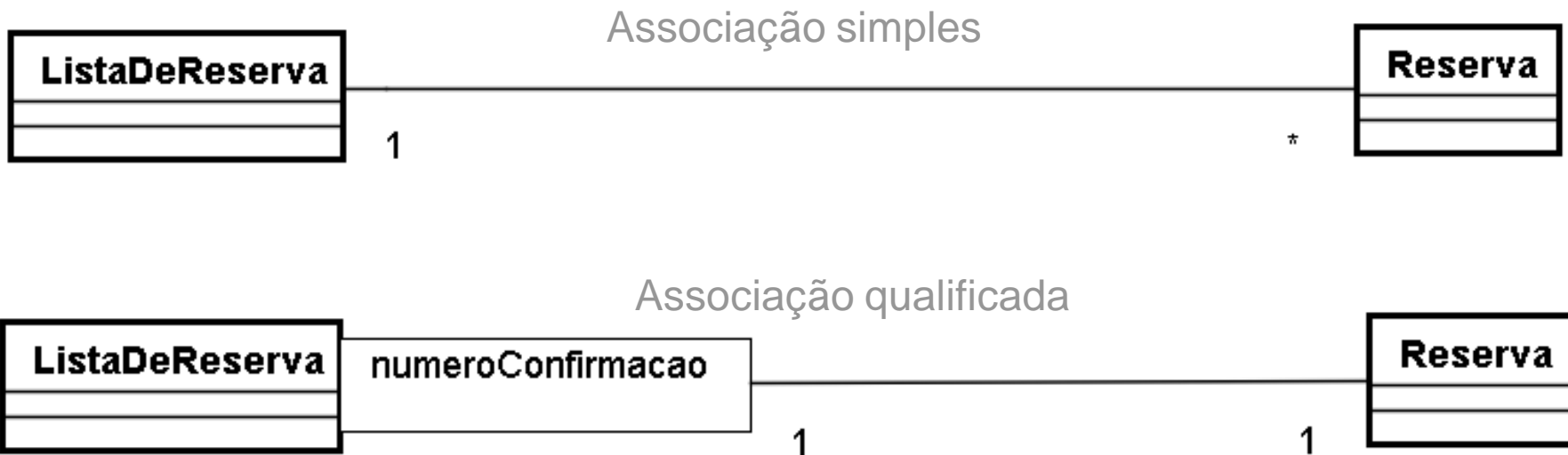
Associação simples

- ▶ Indica que objetos de um elemento estão ligados a objetos de outro elemento
- ▶ A navegabilidade pode ser unidirecional ou bidirecional



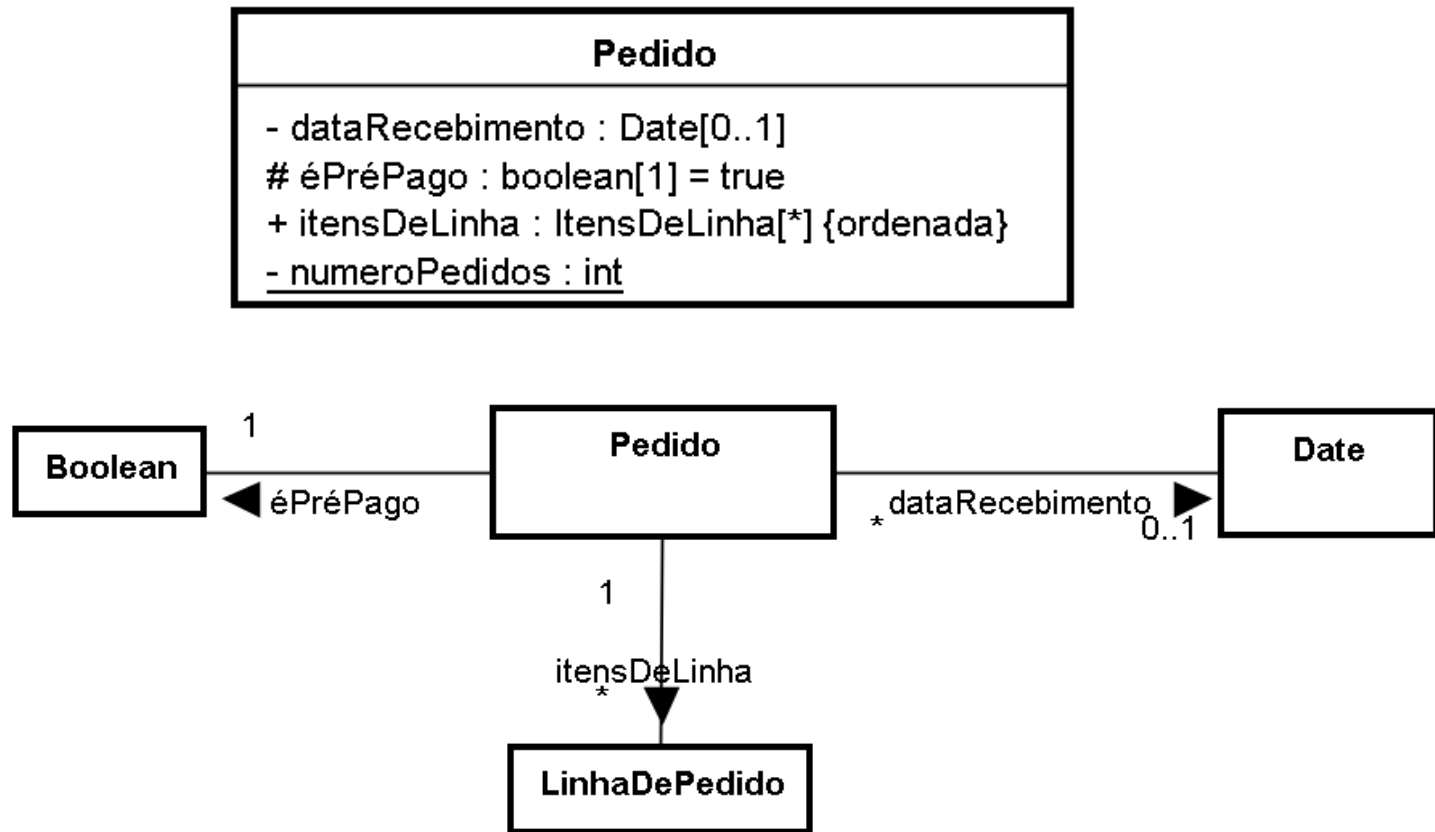
Associação qualificada

- Um qualificador de associação é um atributo do elemento-alvo capaz de identificar uma instância dentre as demais



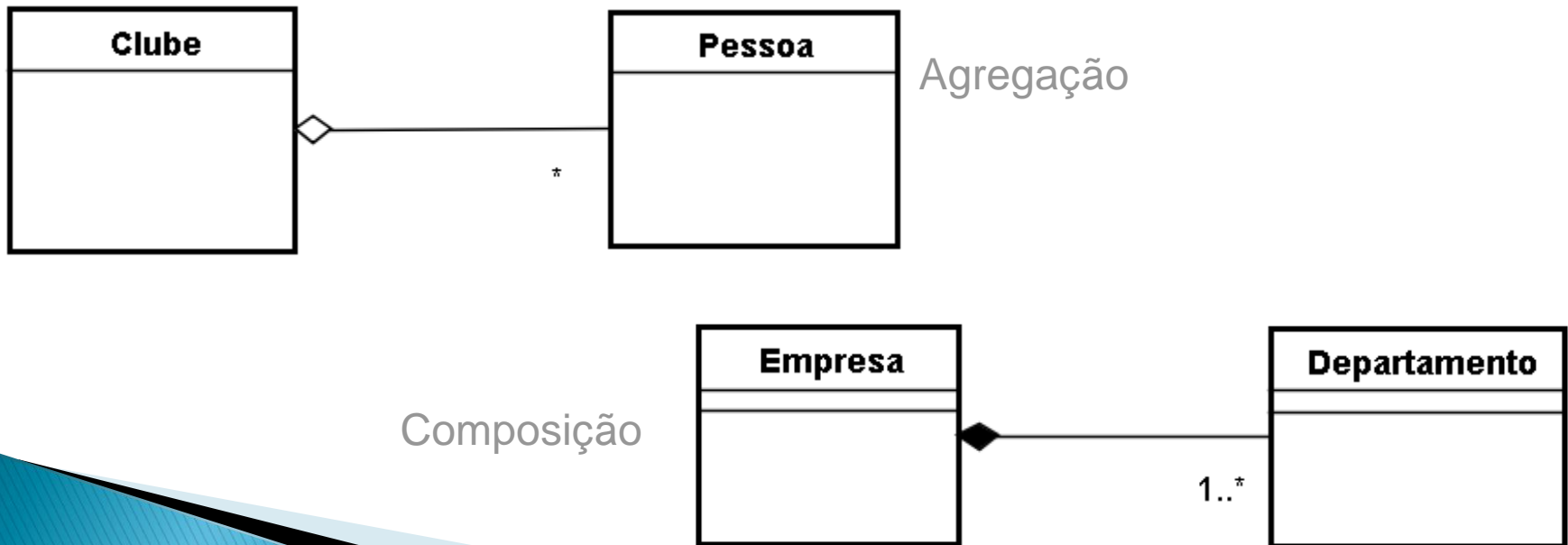
Associação x Atributos

Uma associação pode mostrar as mesmas informações que um atributo



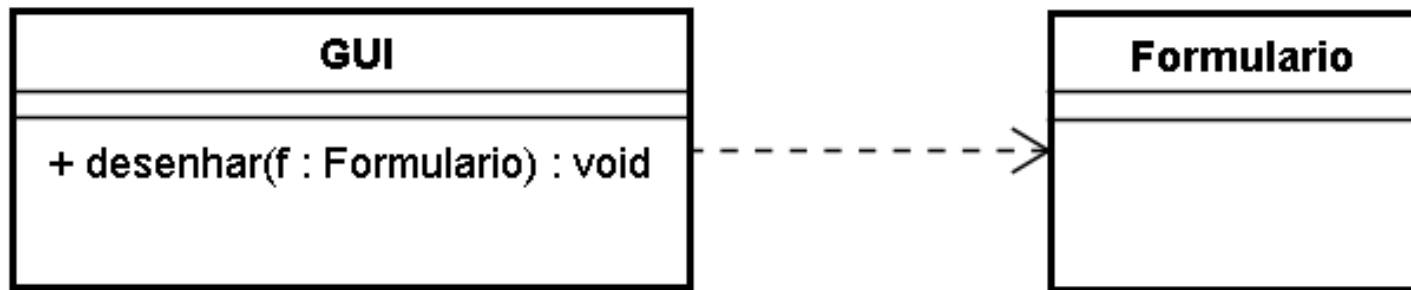
Agregação e Composição

- ▶ Relacionamentos “todo–parte”
- ▶ **Agregação**: a parte existe sem o todo
- ▶ **Composição**: o todo controla o ciclo de vida da parte, e ela não pode ser compartilhada em outros relacionamentos



Dependência

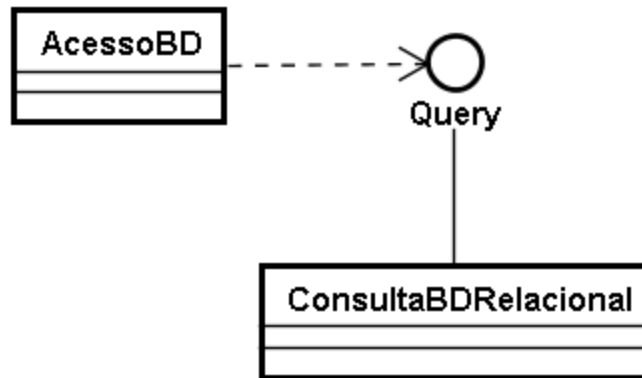
- ▶ Indica que mudança em um elemento pode causar mudanças no outro (uso)



```
public class GUI {  
  
    public void desenhar(Formulario f) {  
        f.pintarBotao();  
        f.pintarMenu();  
        f.pintarJanelas();  
        ...  
    }  
}
```


Dependência

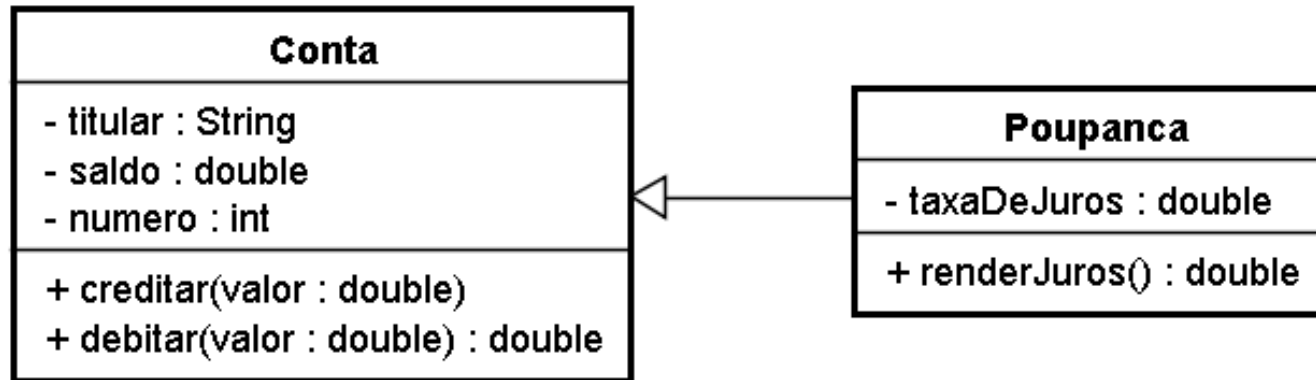
- ▶ Pode ocorrer, também, entre uma classe e uma interface



```
public class AcessoBD {  
    private Conexao conexao;  
  
    public void consultar(Query query) {  
        query.iniciarConsulta();  
        ...  
    }  
}
```

Generalização

- Relacionamento “é um tipo de”

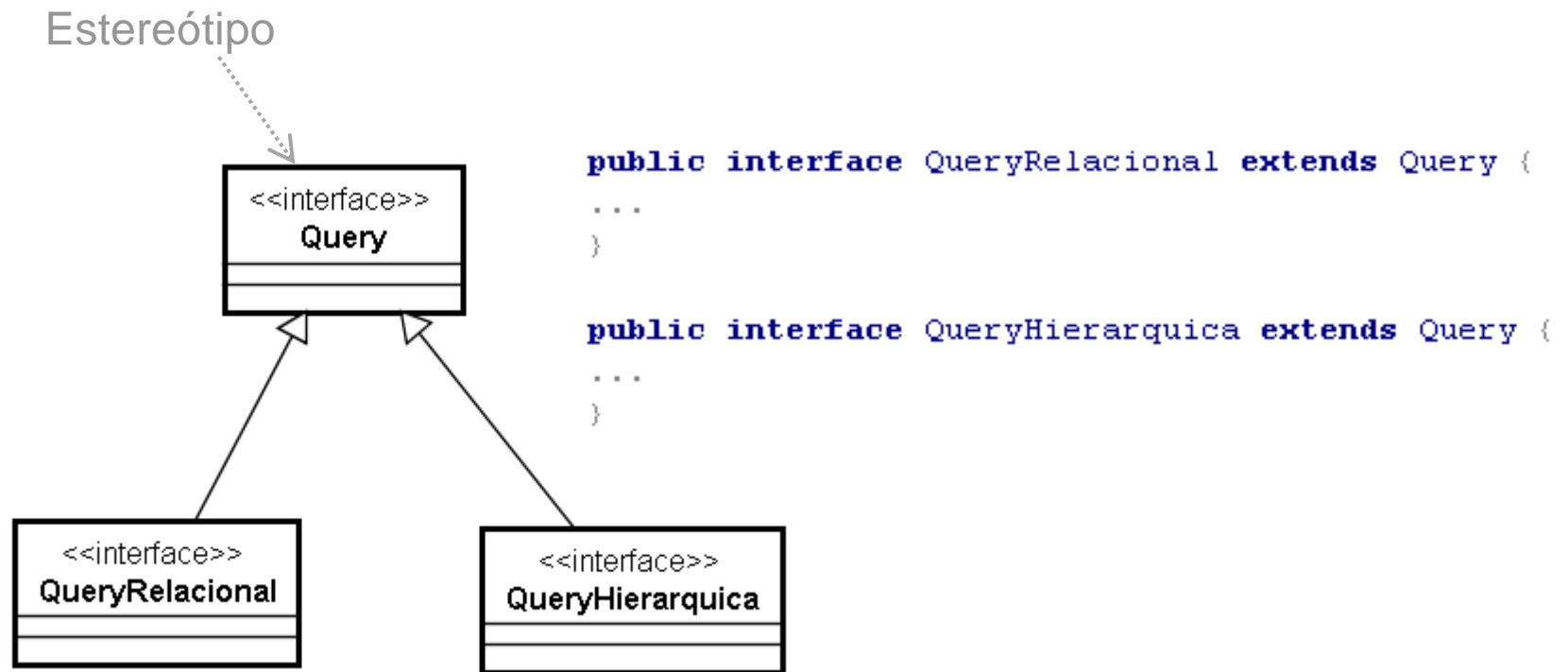


```
public class Conta {
    private String titular;
    private double saldo;
    private int numero;
    public void creditar(double valor) {...}
    public double debitar (double valor) {...}
}

public class Poupanca extends Conta {
    private double taxaDeJuros;
    public double renderJuros () {...}
}
```

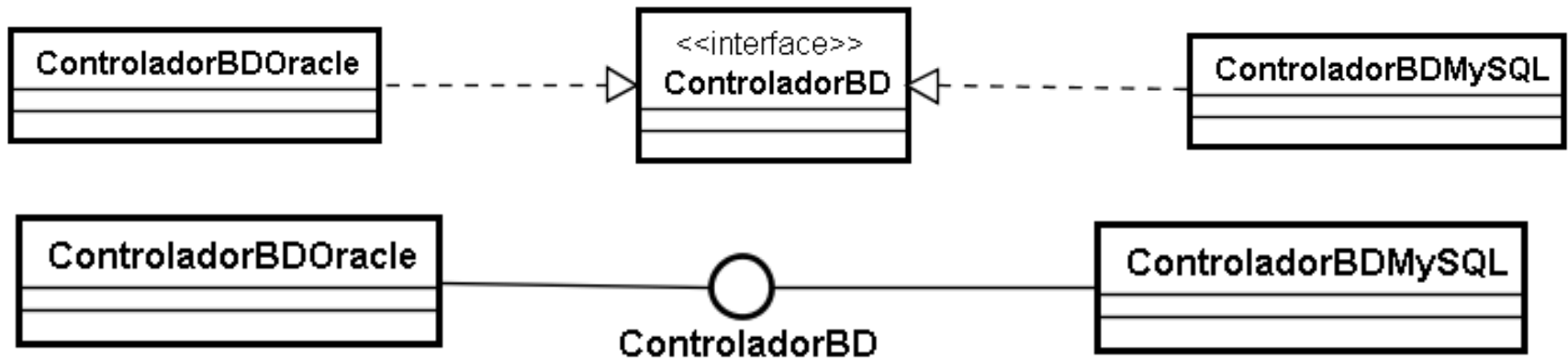
Generalização

- Pode ocorrer, também, entre interfaces



Realização (Interfaces)

- ▶ Há várias notações para realizações

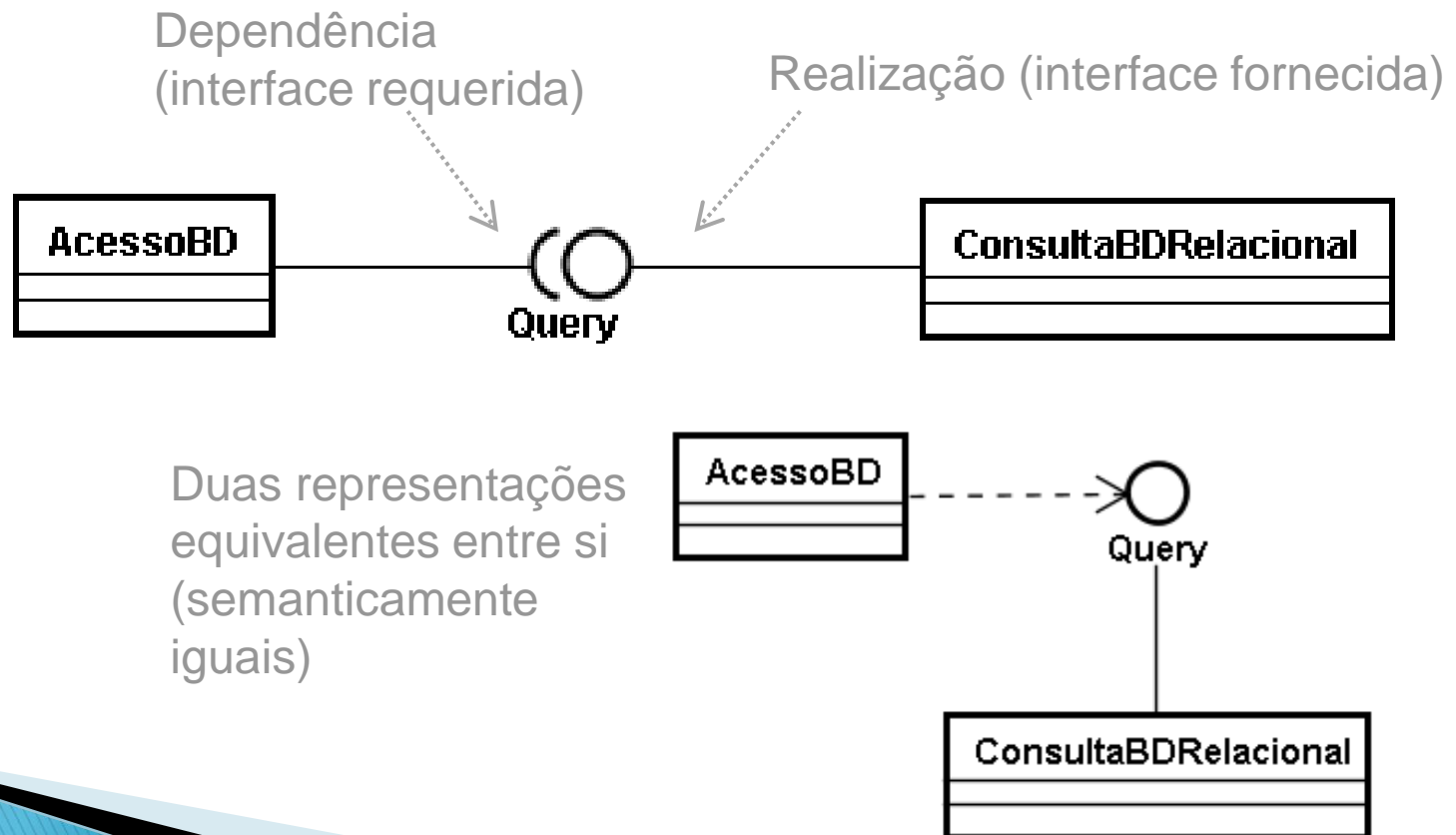


```
public interface ControladorBD {  
  
    public void insert(String SQL);  
    public Object select(String SQL);  
    ...  
}
```

```
public class ControladorOracle  
    implements ControladorBD {  
  
    public void insert(String SQL) {  
        ...  
    }  
    public Object select(String SQL) {  
        ...  
    }  
}
```

Realização (Interfaces)

- ▶ A notação “bola-soquete” (UML 2.0) é utilizada para modelar uma dependência e uma realização entre duas classes e uma interface



Exercícios [2]

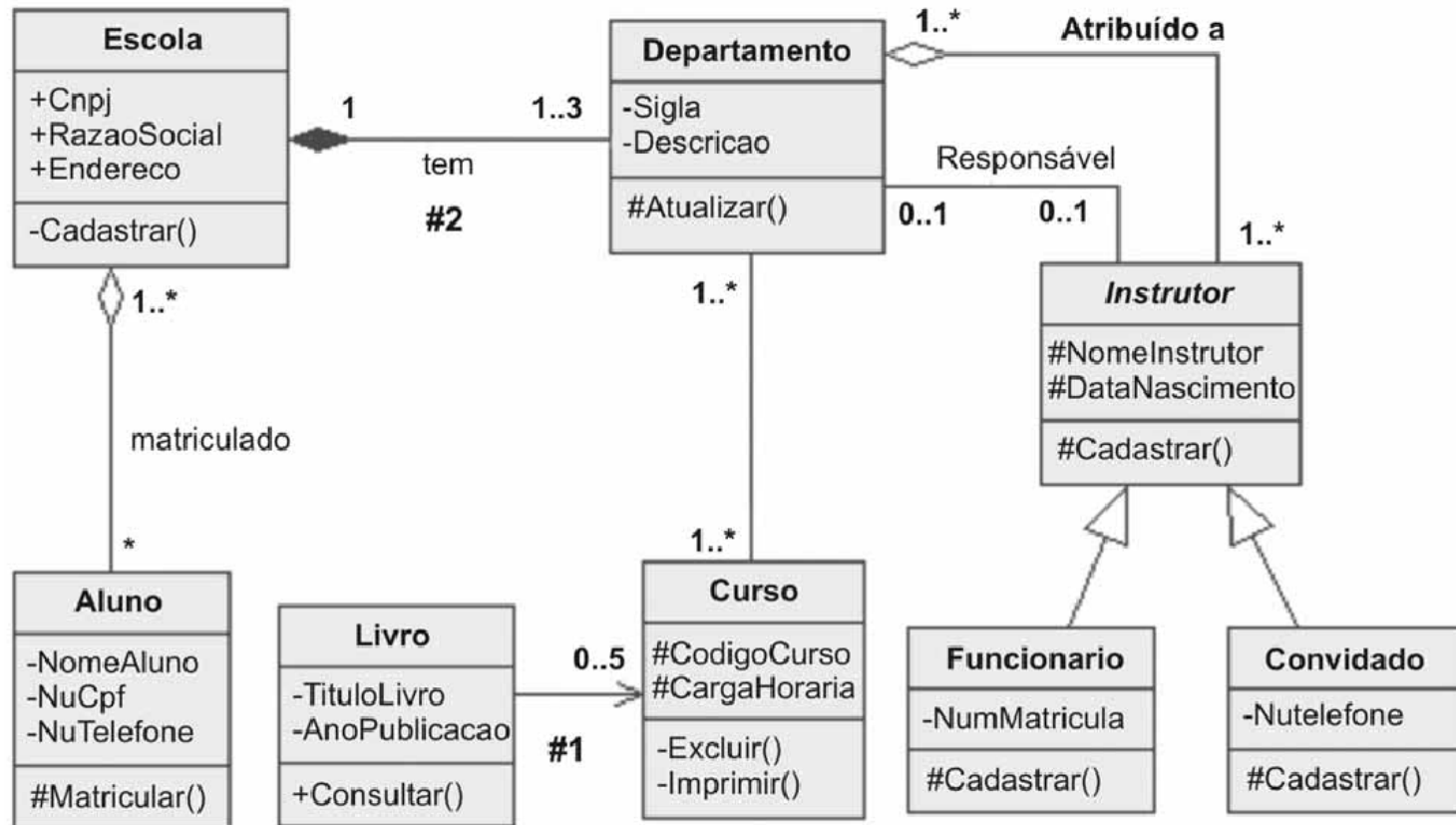
(MPE/AM – CESPE 2008)

[88] Na UML, uma agregação é um relacionamento que estabelece que uma classe define objetos que são parte de um objeto definido por outra classe.

[89] Em um diagrama de classes UML, uma associação entre classes pode ser documentada em termos da multiplicidade da associação.

Exercícios [2]

(TCU – CESPE 2009)



Exercícios [2]

[108] O método #Cadastrar() da classe Instrutor tem visibilidade do modo protegido tal que somente a classe possuidora Instrutor pode utilizá-lo.

[109] Na associação do tipo agregação composta identificado por #2, uma instância da classe Departamento pertence exclusivamente a uma única instância composta em Escola, e um objeto da classe Escola pode relacionar-se com, no máximo, três objetos da classe Departamento.

[110] Instrutor é uma superclasse abstrata; assim, o método #Cadastrar() oferece uma assinatura, que, no entanto, está incompleta, devendo ser implementada pelos métodos de mesmo nome nas suas classes-filhas.

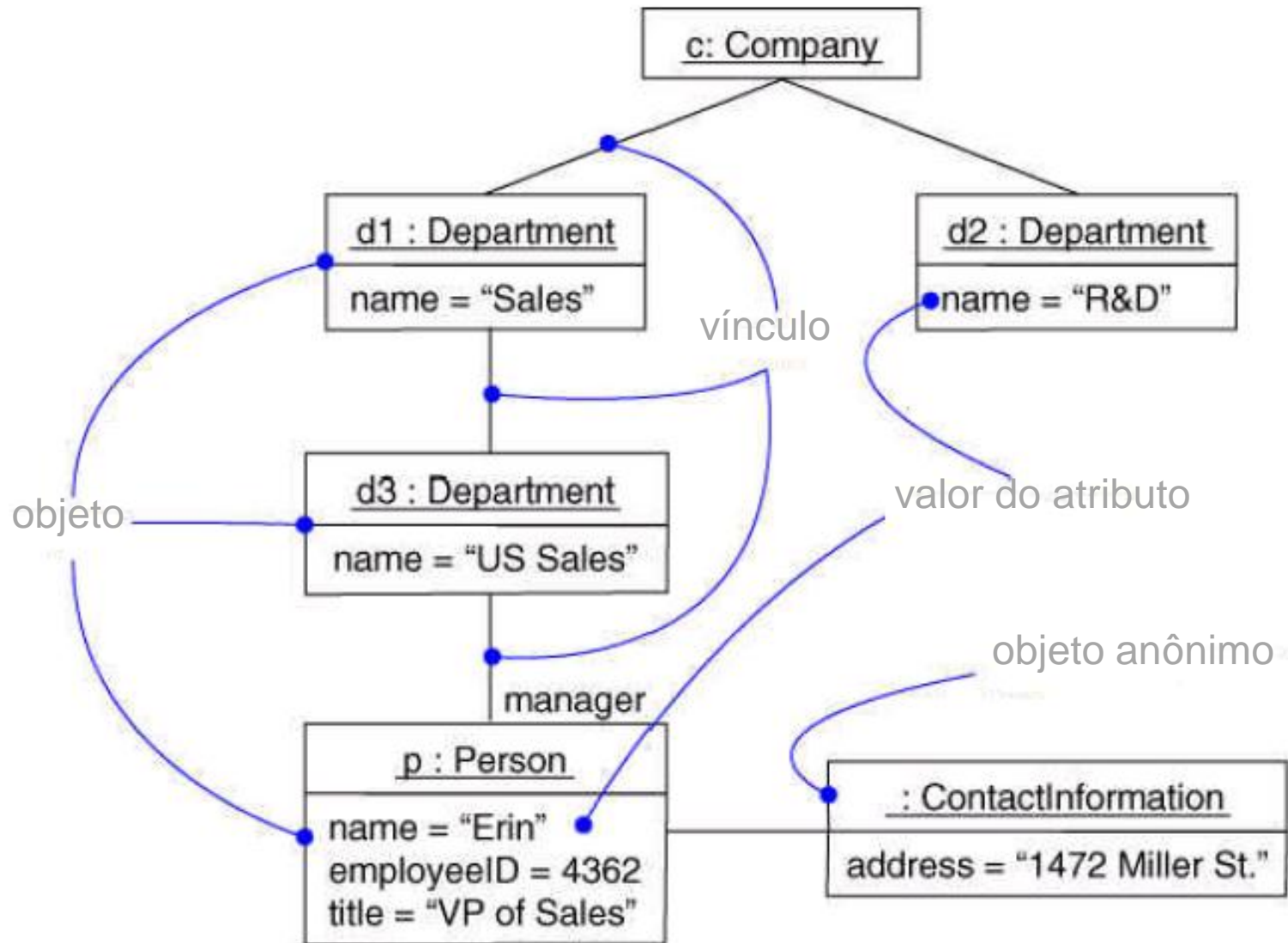
Diagramas Estruturais (estáticos)

- ▶ Diagrama de Classes
- ▶ **Diagrama de Objetos**
- ▶ Diagrama de Componentes
- ▶ Diagrama de Pacotes
- ▶ Diagrama de Implantação
- ▶ Diagrama de Estrutura Composta
- ▶ Diagrama de Perfis (UML 2.2)

Diagrama de Objetos

- ▶ O diagrama de objetos representa uma fotografia do sistema em um dado momento
- ▶ Mostra os vínculos entre os objetos conforme estes interagem e os valores dos seus atributos
- ▶ Pode ser visto como uma “instância” do diagrama de classe

Diagrama de Objetos



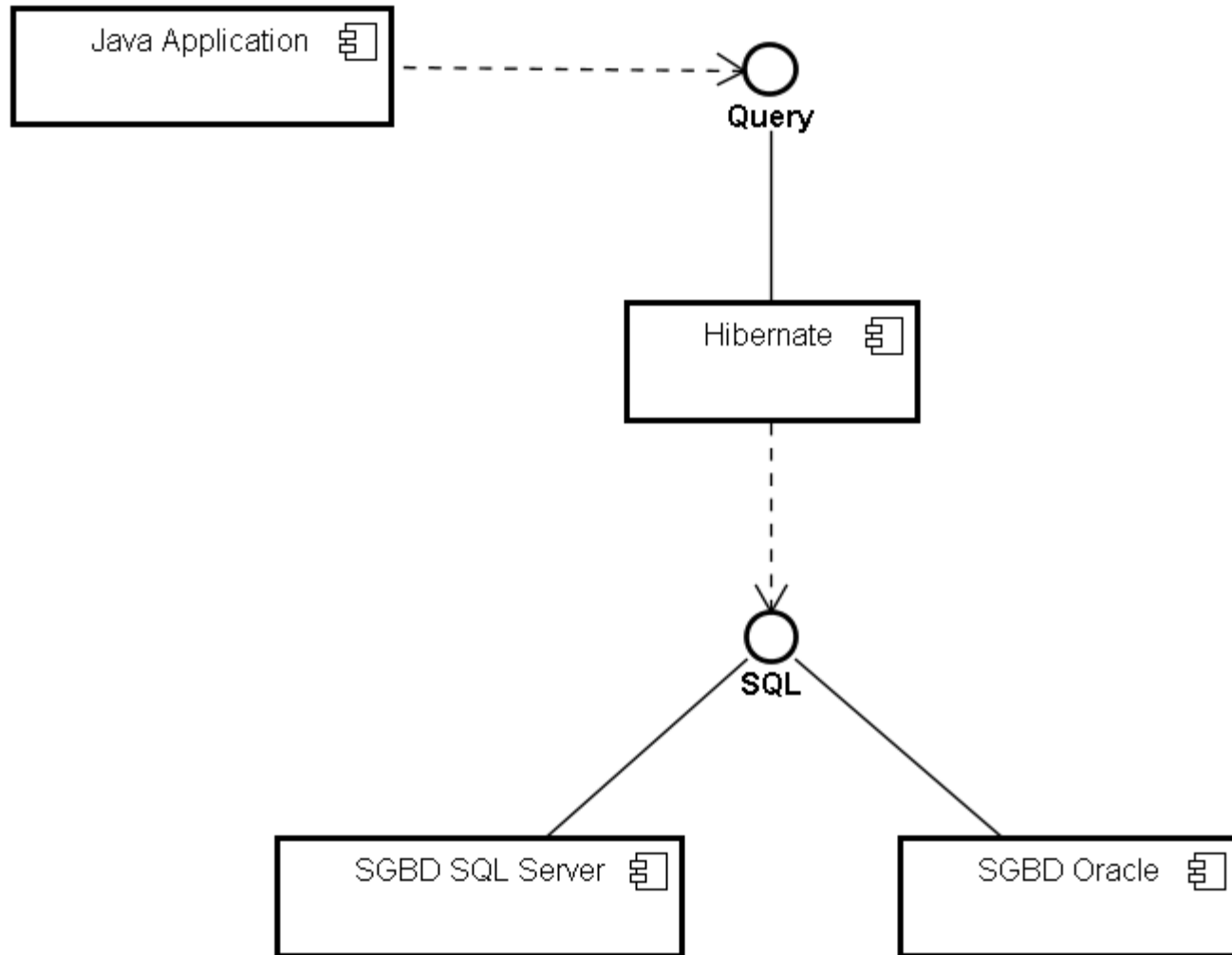
Diagramas Estruturais (estáticos)

- ▶ Diagrama de Classes
- ▶ Diagrama de Objetos
- ▶ **Diagrama de Componentes**
- ▶ Diagrama de Pacotes
- ▶ Diagrama de Implantação
- ▶ Diagrama de Estrutura Composta
- ▶ Diagrama de Perfis (UML 2.2)

Diagrama de Componentes

- ▶ Modela o sistema em termos de componentes e seus relacionamentos através de interfaces
- ▶ Decompõe o sistema em subsistemas que detalham a estrutura interna
- ▶ Alguns componentes existem em tempo de ligação, outros em tempo de execução

Diagrama de Componentes



Diagramas Estruturais (estáticos)

- ▶ Diagrama de Classes
- ▶ Diagrama de Objetos
- ▶ Diagrama de Componentes
- ▶ **Diagrama de Pacotes**
- ▶ Diagrama de Implantação
- ▶ Diagrama de Estrutura Composta
- ▶ Diagrama de Perfis (UML 2.2)

Diagrama de Pacotes

- ▶ Pacotes são estruturas que permitem agrupar qualquer construção da UML em estruturas de alto nível
- ▶ Pode mostrar:
 - Pacotes e suas dependências
 - Interfaces entre os pacotes
 - Generalizações entre pacotes

Diagrama de Pacotes

- ▶ Duas representações possíveis

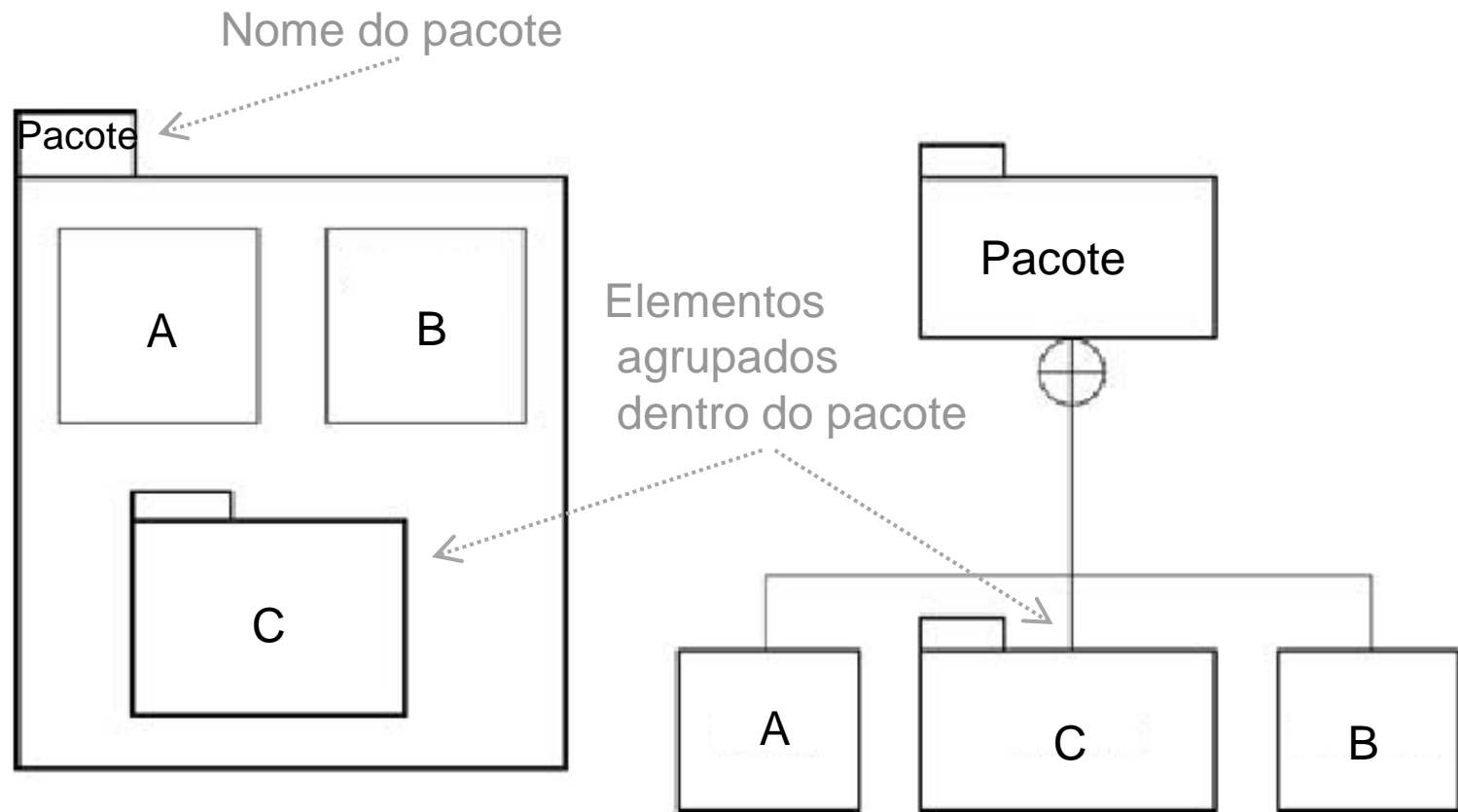
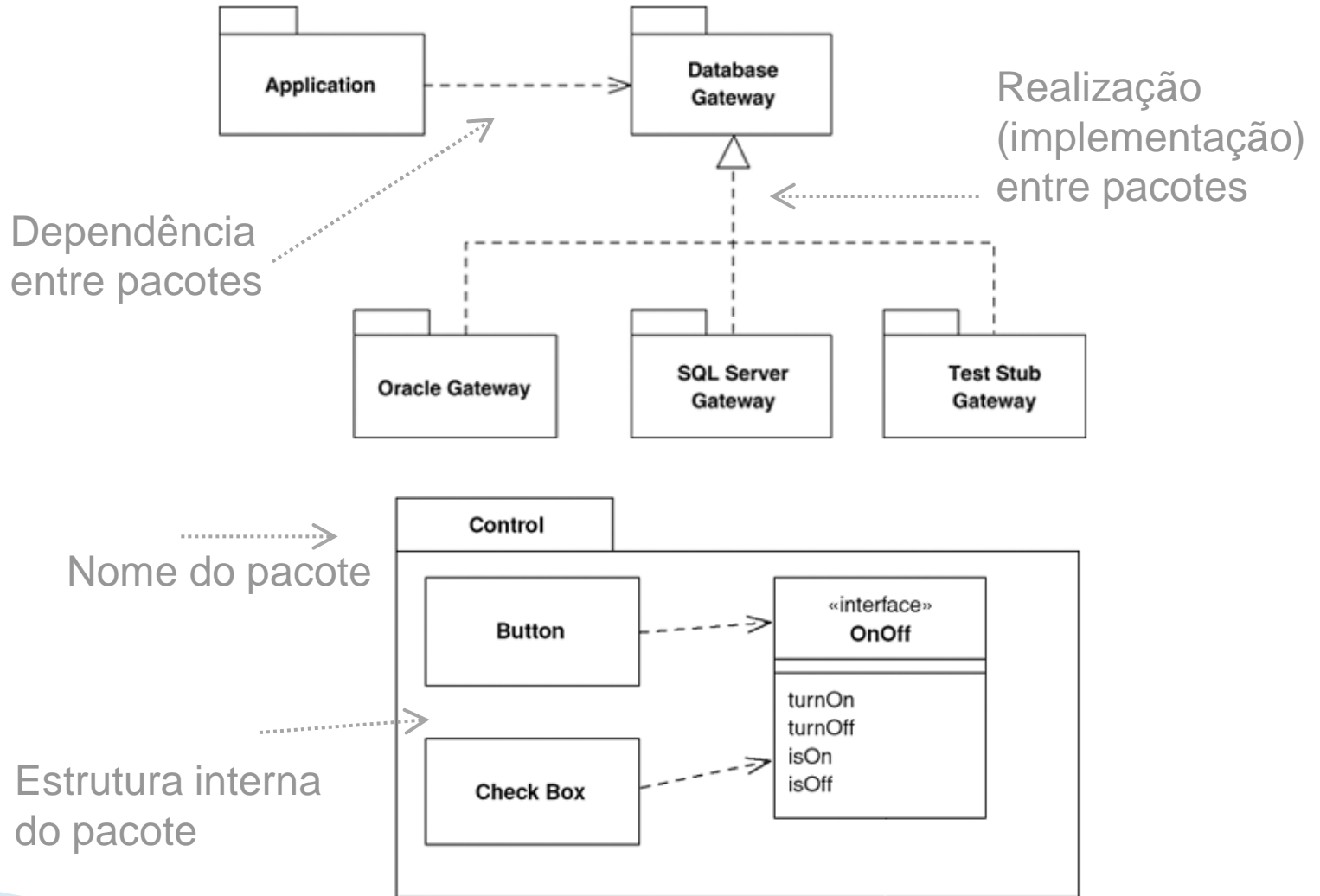


Diagrama de Pacotes



Exercícios [3]

(UNIPAMPA – CESPE 2010)

[106] O diagrama de objetos está amplamente associado ao diagrama de classes, sendo que o primeiro consiste em uma instância do segundo, em determinado momento da execução, ou seja, um diagrama de objetos descreve os objetos, os métodos, os atributos e seus valores, além dos vínculos entre os objetos, sendo ambos diagramas estruturais.

(TRE/TO – CESPE 2007)

[40] Um diagrama de componentes permite mostrar componentes de um sistema e as dependências entre eles. As dependências entre os componentes podem ser, por exemplo, dependências de compilação ou de comunicação.

Exercícios [3]

(EMBASA – CESPE 2009)

[96] O objetivo principal de um diagrama de pacotes é agrupar os pacotes em classes. Esse tipo de diagrama pode usar dependências.

(TJ/PE – FCC 2007)

[40] Diagrama de Pacotes descreve os pacotes ou pedaços do sistema, como o sistema é dividido em agrupamentos lógicos e mostra as dependências entre estes.

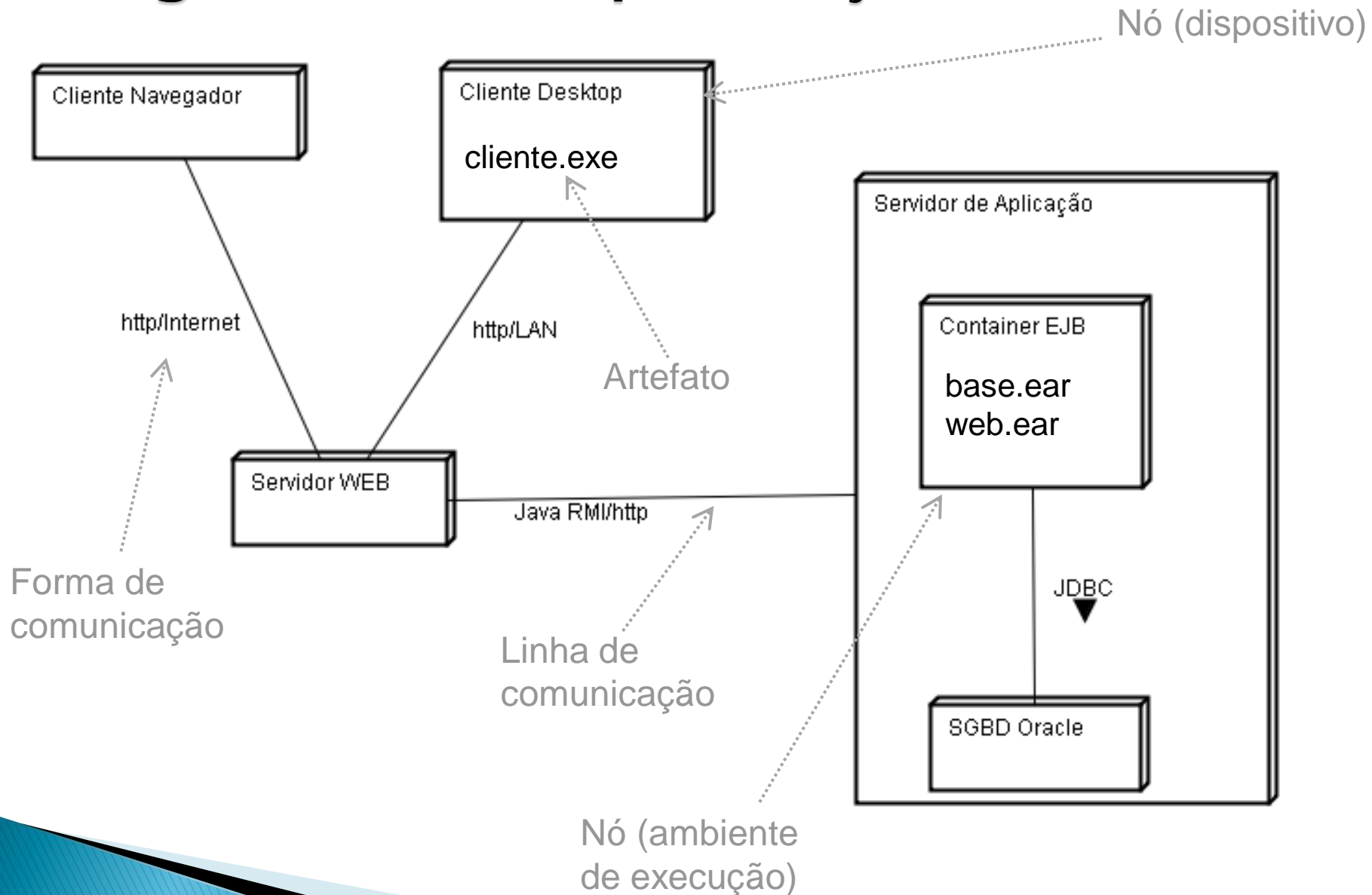
Diagramas Estruturais (estáticos)

- ▶ Diagrama de Classes
- ▶ Diagrama de Objetos
- ▶ Diagrama de Componentes
- ▶ Diagrama de Pacotes
- ▶ **Diagrama de Implantação**
- ▶ Diagrama de Estrutura Composta
- ▶ Diagrama de Perfis (UML 2.2)

Diagrama de Implantação

- ▶ Modela a configuração física do sistema, revelando que pedaços de software rodam em que equipamentos de hardware
- ▶ Inclui
 - Nós
 - Dispositivos (Hardware)
 - Ambientes de Execução
 - Artefatos
 - Código fonte, Código binário
 - Executáveis, etc.

Diagrama de Implantação



Exercícios [4]

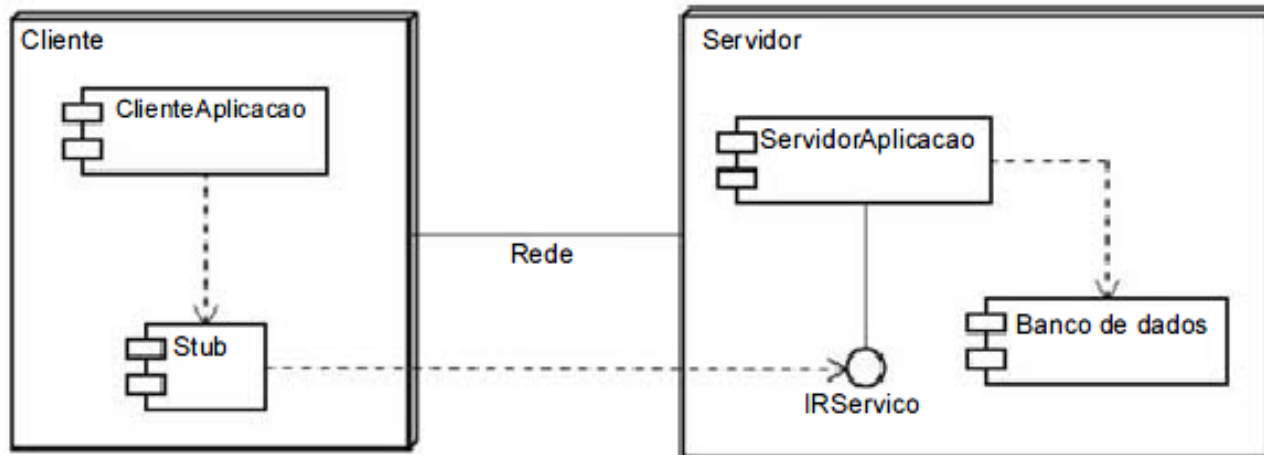


Figura III

(STJ – CESPE 2008)

[54] No diagrama da figura III, há dois nós interligados, que representam duas unidades computacionais; há cinco componentes distribuídos entre os nós; um destes implementa uma interface e um outro depende dessa interface; ClienteAplicacao depende de Stub; ServidorAplicacao depende de Banco de dados.

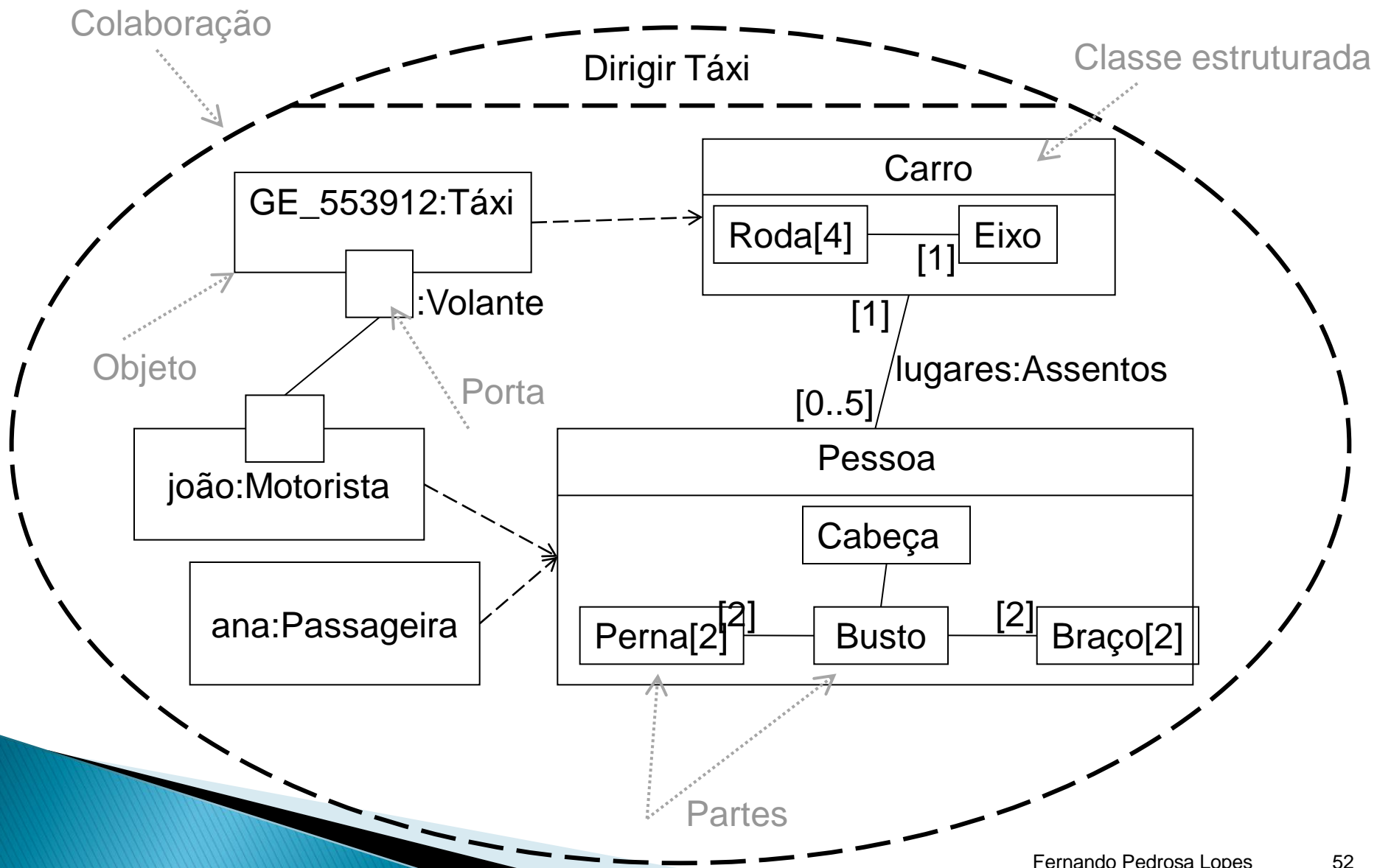
Diagramas Estruturais (estáticos)

- ▶ Diagrama de Classes
- ▶ Diagrama de Objetos
- ▶ Diagrama de Componentes
- ▶ Diagrama de Pacotes
- ▶ Diagrama de Implantação
- ▶ **Diagrama de Estrutura Composta**
- ▶ Diagrama de Perfis (UML 2.2)

Diagrama de Estrutura Composta

- ▶ É utilizado para modelar colaborações entre interfaces, objetos ou classes
- ▶ Pode ser usado para descrever
 - Estruturas de partes interconectadas
 - Estruturas de instâncias interconectadas
- ▶ **Parte:** representa o conjunto de uma ou mais instâncias contidas em outro elemento
- ▶ **Porta:** ponto de interação entre os elementos

Diagrama de Estrutura Composta



Exercícios [5]

(UNIPAMPA – CESPE 2010)

[105] Na UML 2.0, o diagrama de estrutura composta (composite structure diagram) descreve a estrutura interna de um classificador modelando as colaborações, no qual uma colaboração descreve uma visão de um conjunto de instâncias que cooperam entre si para executar uma função específica entre instâncias de classes, objetos ou interfaces.

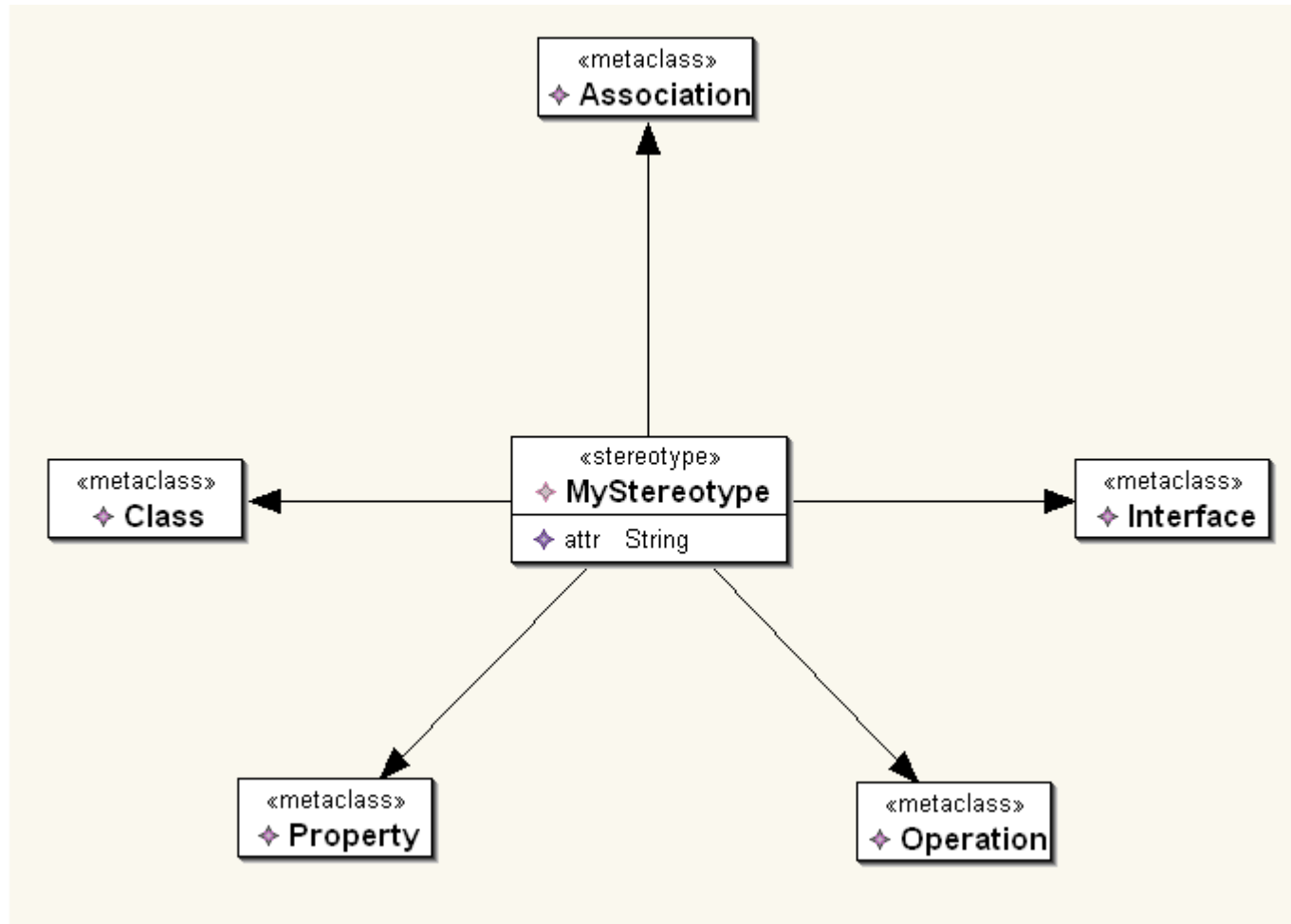
Diagramas Estruturais (estáticos)

- ▶ Diagrama de Classes
- ▶ Diagrama de Objetos
- ▶ Diagrama de Componentes
- ▶ Diagrama de Pacotes
- ▶ Diagrama de Implantação
- ▶ Diagrama de Estrutura Composta
- ▶ **Diagrama de Perfis (UML 2.2)**

Diagrama de Perfis (Profile Diagram)

- ▶ É um diagrama auxiliar que permite definir tipos padronizados de estereótipos, valores rotulados e restrições
- ▶ A UML define o mecanismo de perfis como um “mecanismo leve de extensão” da linguagem
- ▶ Permite adaptar os modelos UML para diferentes plataformas e domínios

Diagrama de Perfis (Profile Diagram)



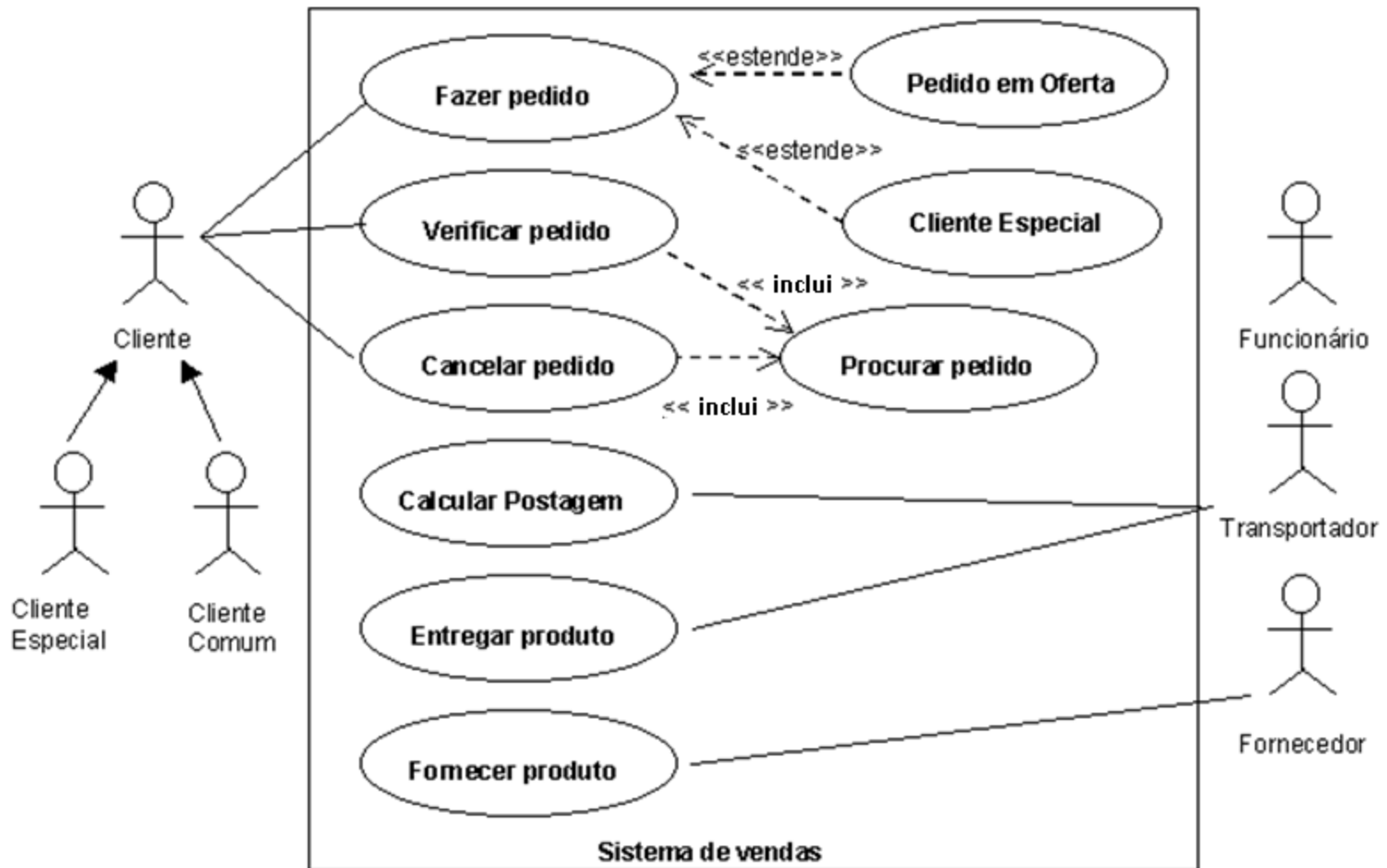
Diagramas Comportamentais (dinâmicos)

- ▶ Diagrama de Casos de Uso
- ▶ Diagrama de Atividade
- ▶ Diagrama de Máquina de Estados
- ▶ Diagramas de Interação
 - Diagrama de Sequência
 - Diagrama de Comunicação
 - Diagrama de Tempo
 - Diagrama de Interação Geral

Diagrama de Casos de Uso

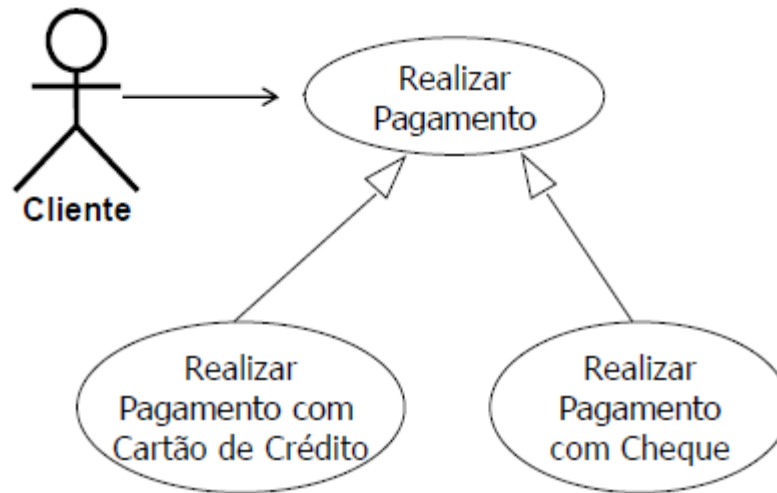
- ▶ Contém um conjunto de casos de uso e modela interações entre
 - Atores e o sistema
 - O próprio sistema
- ▶ Descreve um conjunto de cenários
- ▶ Captura os requisitos do usuário
- ▶ Delimita o escopo do sistema

Diagrama de Caso de Uso



Generalizações entre Casos de Uso

- ▶ O filho herda o comportamento do pai, podendo adicionar e redefinir passos em pontos arbitrários do comportamento original



Inclusão e Extensão

▶ Inclusão

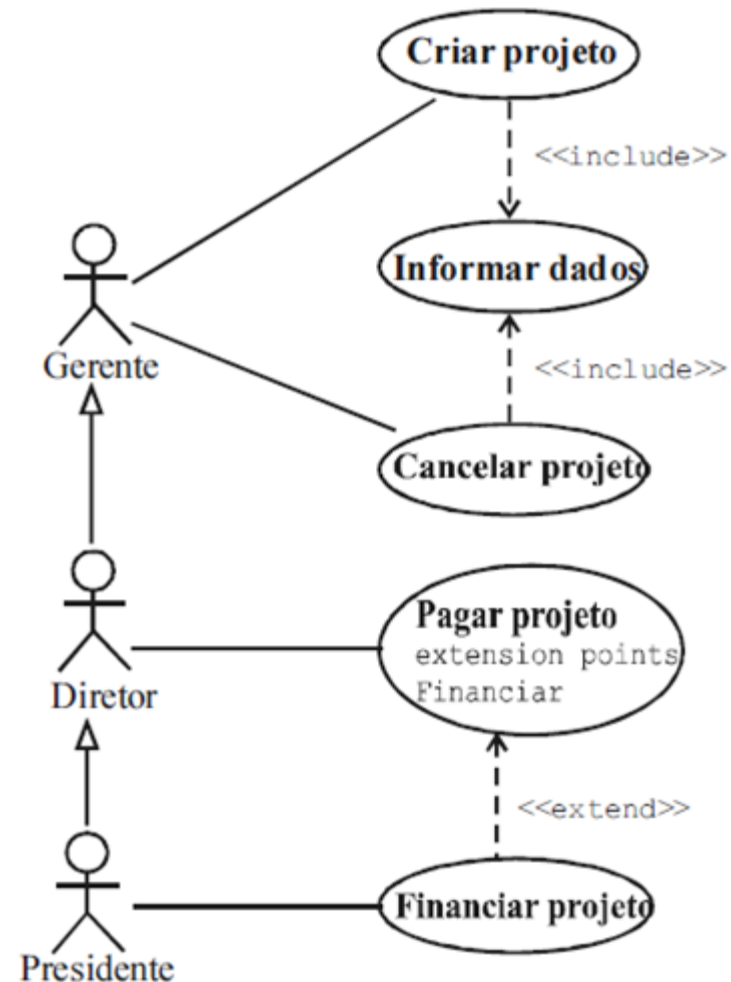
- Use quando o mesmo comportamento se repete em mais de um Caso de Uso e o processo de realizar X **sempre** envolve realizar Y pelo menos uma vez

▶ Extensão

- Use quando você quiser modelar um comportamento **opcional** de um Caso de Uso

Generalização entre Atores

- ▶ Use quando um ator (filho) é um **tipo de** outro ator mais genérico (pai)
- ▶ Exemplo:



Tipos de Casos de Uso

- ▶ **Concreto**
 - É iniciado por um ator e constitui um fluxo completo de eventos
- ▶ **Abstrato: nunca é instanciado diretamente**
 - Casos de Uso abstratos geralmente são:
 - Incluídos em outros Casos de Uso
 - Estendidos de outros Casos de Uso
 - Generalizações de outros Casos de Uso
- ▶ **Atores “enxergam” apenas casos de uso concretos**

Exercícios [6]

(EMBASA – CESPE 2009)

[95] Um diagrama de casos de uso descreve um cenário que mostra as funcionalidades do sistema do ponto de vista do usuário. É comum o uso de atores nesse diagrama.

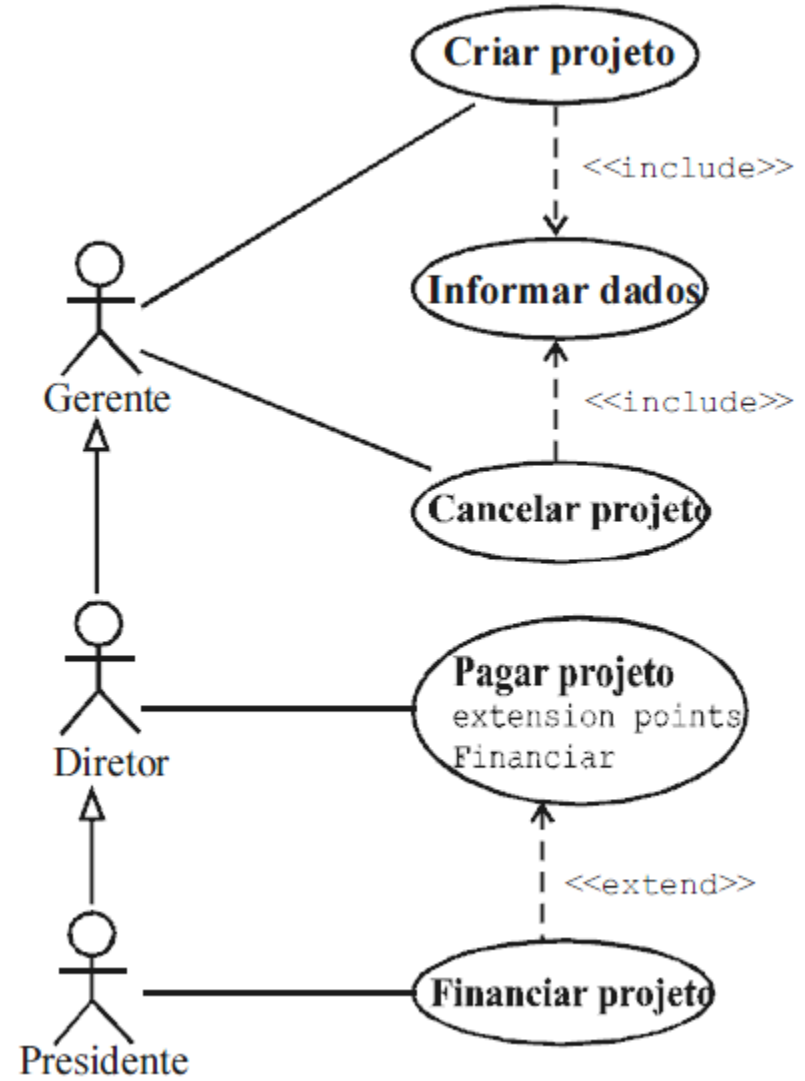
(MPE/AM – CESPE 2008)

[85] Em um diagrama de casos de uso da UML, um ator é definido como um usuário humano do sistema.

Exercícios [6]

(MPE/RR – CESPE 2008)

[87] No diagrama UML ao lado, o ator Presidente está relacionado ao caso de uso Criar projeto; o caso de uso Informar dados contém comportamento comum a dois casos de uso; o caso de uso Pagar projeto estende o comportamento Financiar projeto e Cancelar projeto é abstrato.



Diagramas Comportamentais (dinâmicos)

- ▶ Diagrama de Casos de Uso
- ▶ **Diagrama de Atividade**
- ▶ Diagrama de Máquina de Estados
- ▶ Diagramas de Interação
 - Diagrama de Sequência
 - Diagrama de Comunicação
 - Diagrama de Tempo
 - Diagrama de Interação Geral

Diagrama de Atividade

- ▶ Descreve lógicas de procedimento, processos de negócio e fluxos de trabalho
- ▶ Permite que seja mostrado que entidade é responsável por cada ação no diagrama, com uso de raias (*swimlanes*)
 - Quem faz o quê?

Diagrama de Atividade

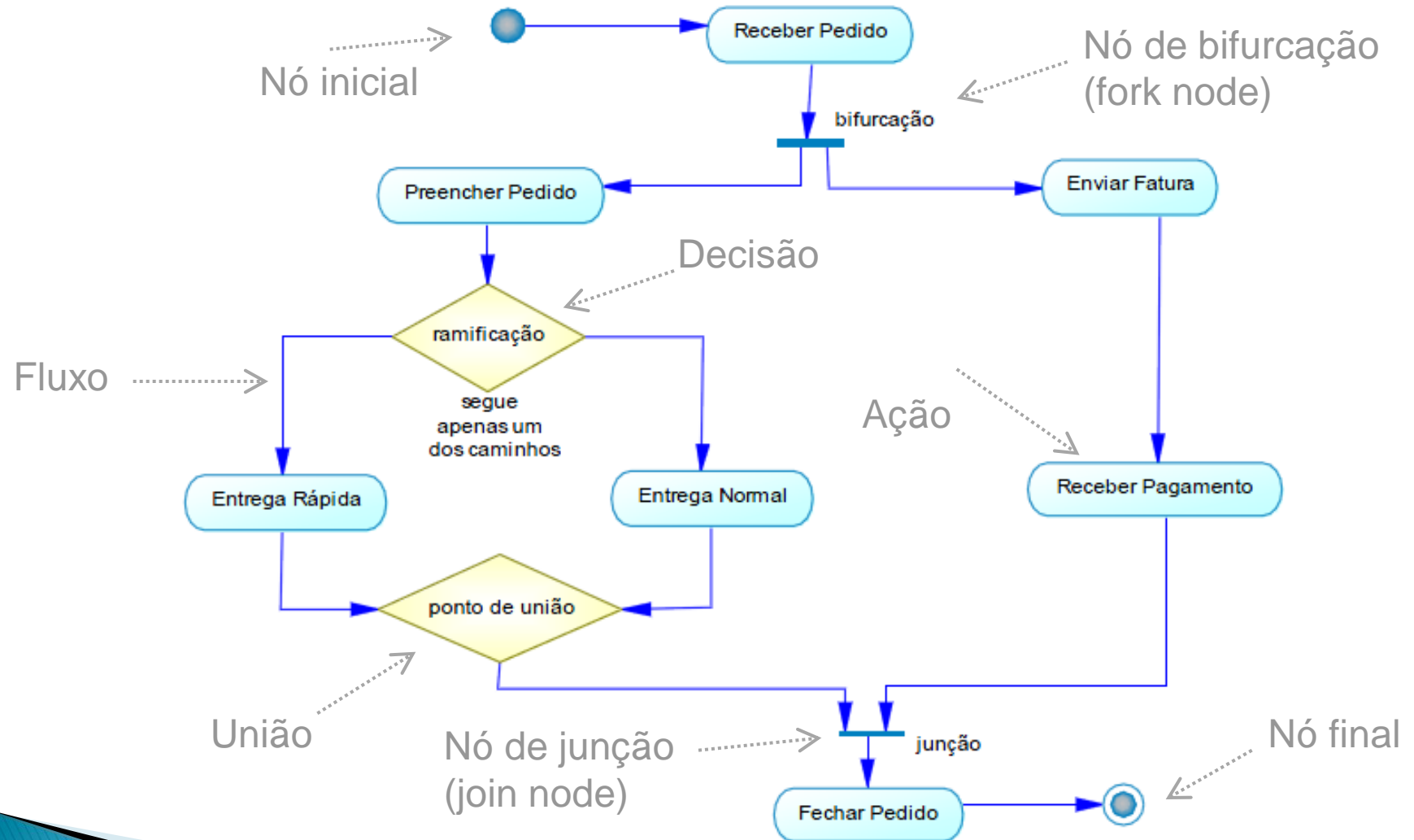
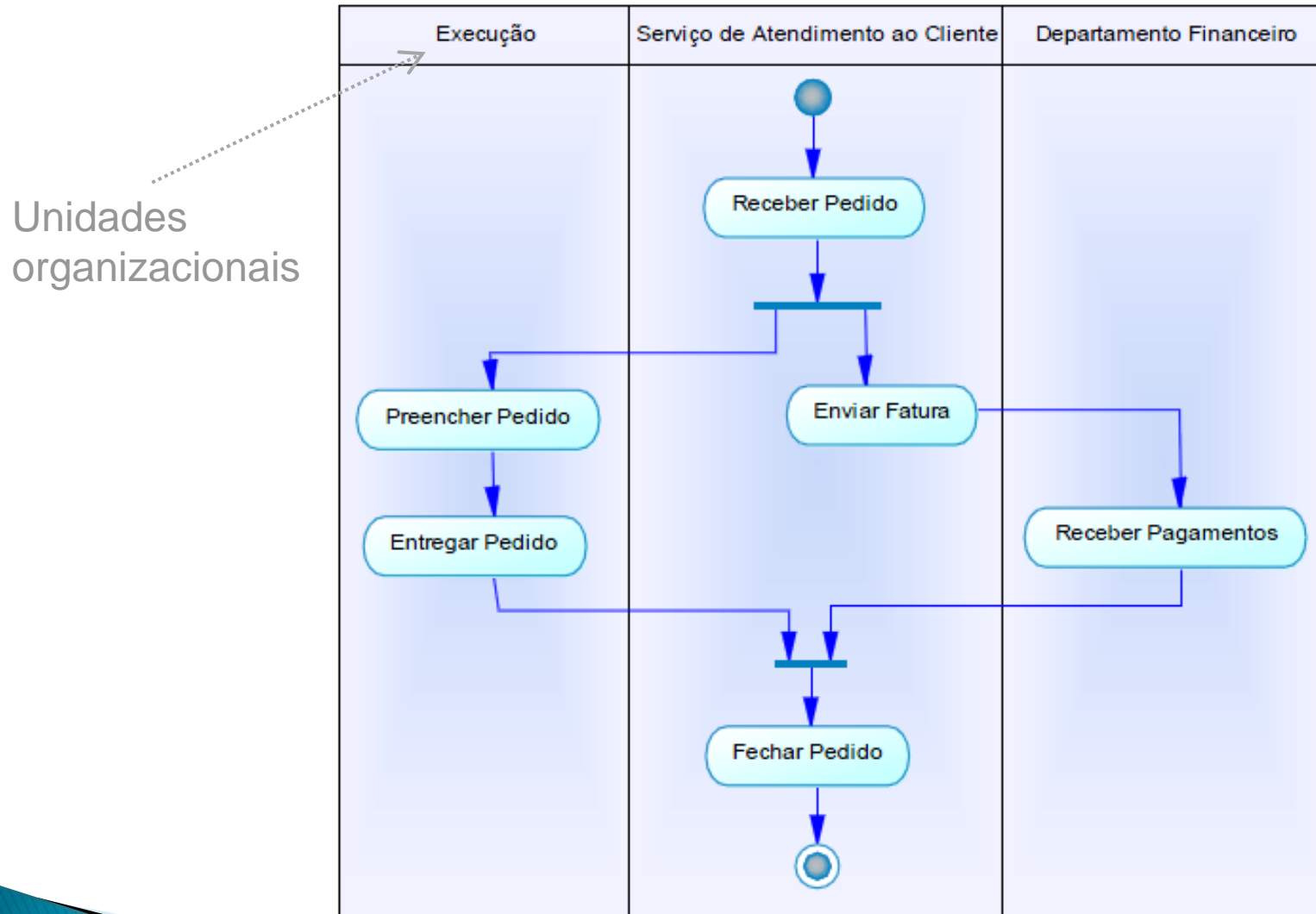


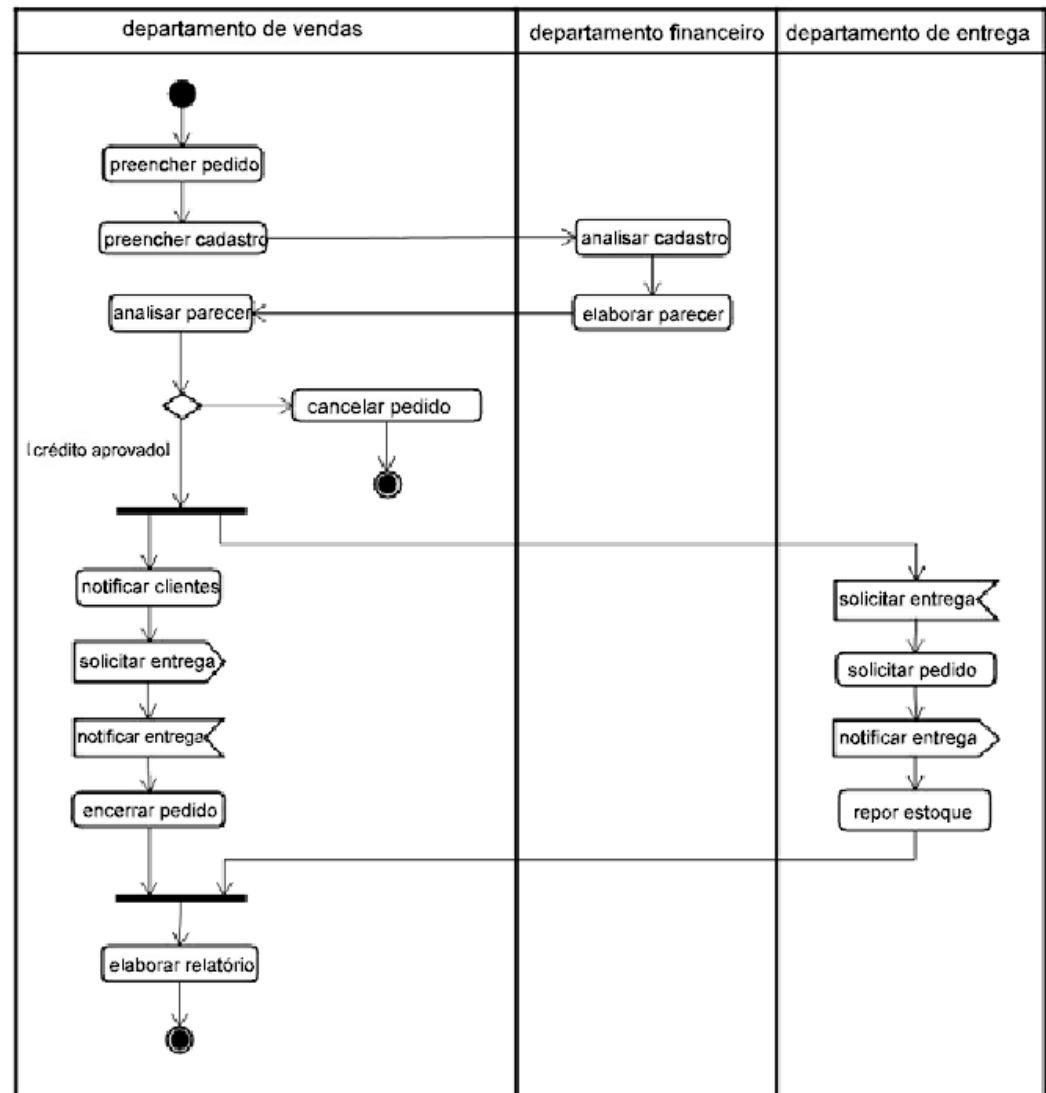
Diagrama de Atividade (swimlanes)



Exercícios [7]

(PETROBRAS – CESPE 2007)

Com referência ao diagrama de atividades UML ao lado, julgue os itens a seguir.



Exercícios [7]

[88] No diagrama, há duas raias, um estado inicial e dois finais. Por estarem em raias distintas, a atividade Preencher cadastro pode ser realizada em paralelo à atividade Analisar cadastro. Na decisão representada pelo losango, apenas uma condição de guarda é especificada, o que torna o diagrama incorreto.

[89] A atividade Notificar cliente pode ser executada em paralelo à atividade Entregar produto, mas a atividade Encerrar pedido não pode ser executada em paralelo à atividade Repor estoque. A atividade Elaborar relatório será executada após ser concluída a atividade Encerrar pedido ou Repor estoque.

Diagramas Comportamentais (dinâmicos)

- ▶ Diagrama de Casos de Uso
- ▶ Diagrama de Atividade
- ▶ **Diagrama de Máquina de Estados**
- ▶ Diagramas de Interação
 - Diagrama de Sequência
 - Diagrama de Comunicação
 - Diagrama de Tempo
 - Diagrama de Interação Geral

Diagrama de Máquina de Estados

- ▶ Mostra os vários estados possíveis por quais um objeto pode passar
- ▶ Um objeto muda de estado quando acontece algum evento interno ou externo ao sistema
- ▶ Através da análise das transições entre os estados, pode-se prever todas as possíveis operações realizadas, em função de eventos que podem ocorrer

Elementos

▶ Estados

- Situações na vida de um objeto na qual ele satisfaz uma condição ou realiza alguma atividade

▶ Transições

- Estados são associados através de transições
- Transições têm eventos associados
 - Sintaxe: **evento [condição]/ação**

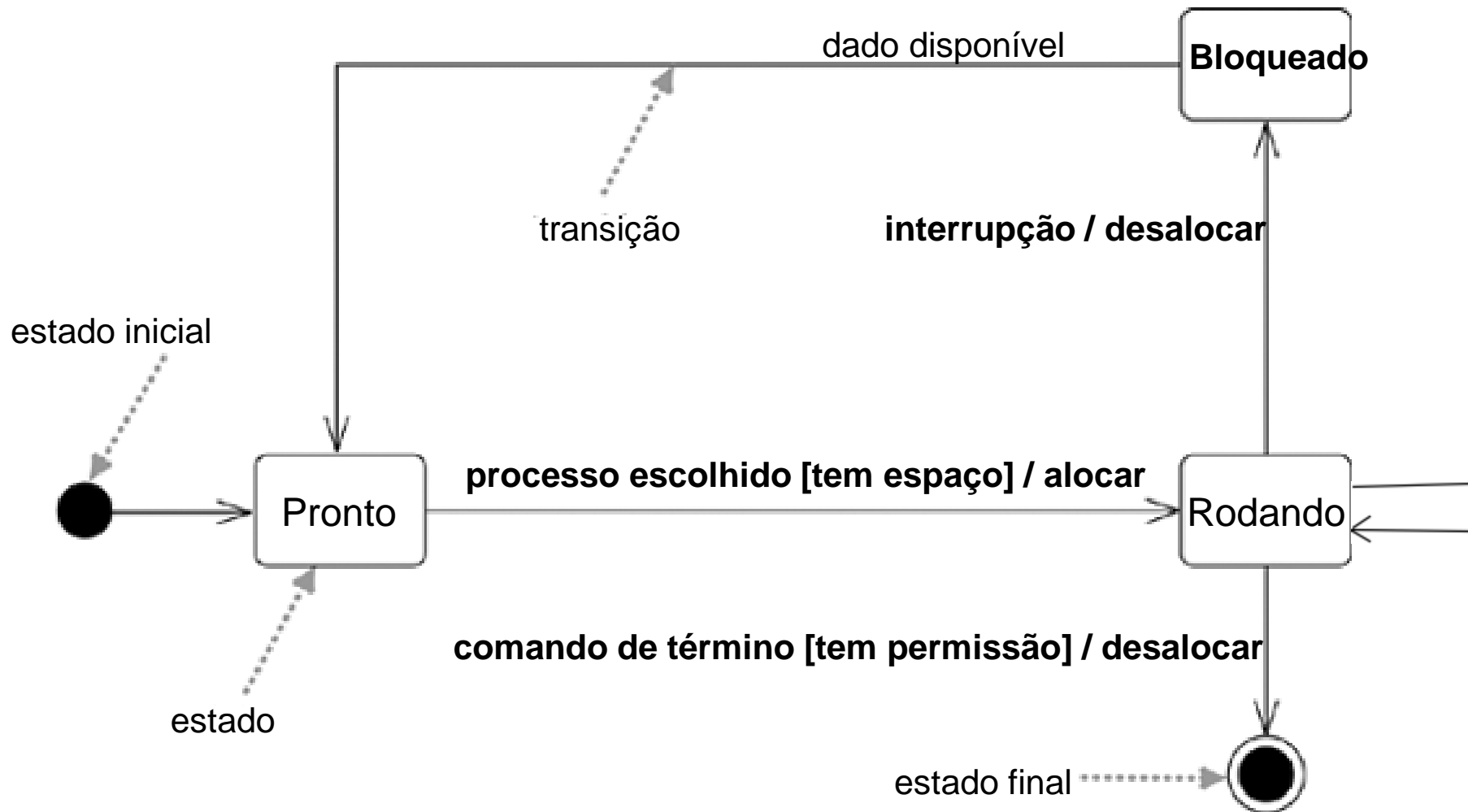
▶ Ações

- Ao passar de um estado para o outro o objeto pode realizar ações

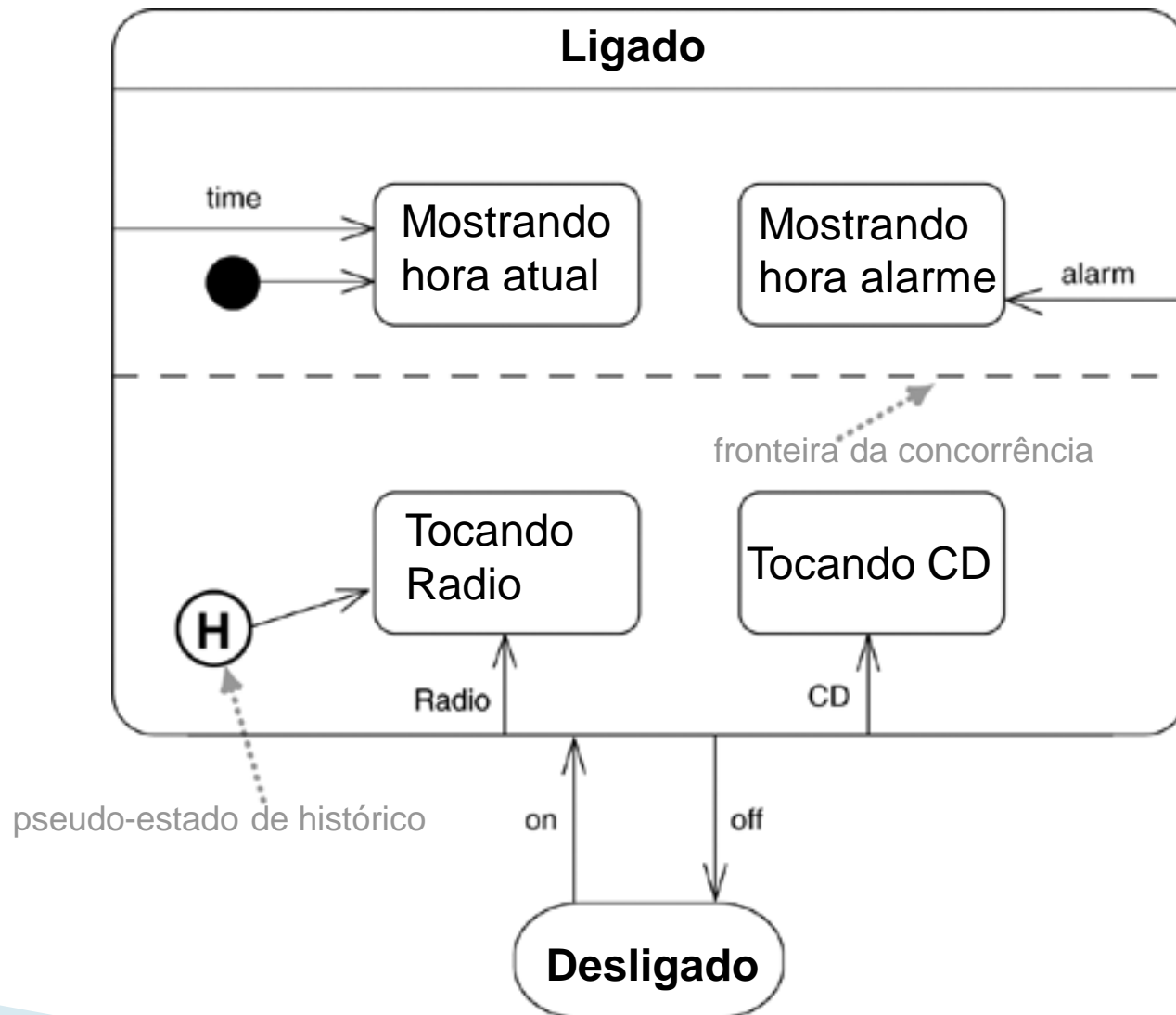
▶ Atividades

- Executadas durante um estado

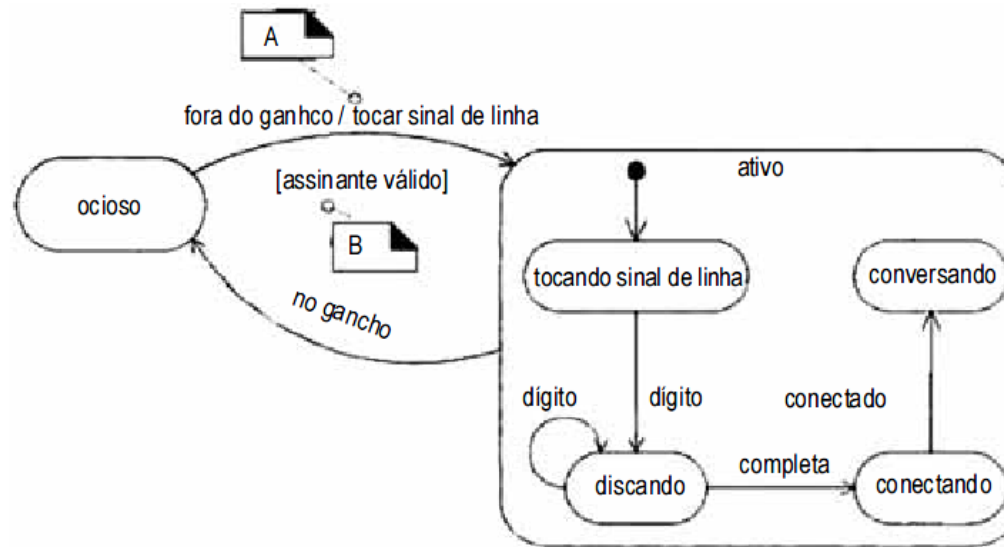
Ex: Escalonamento de Processos



Estados aninhados e concorrentes



Exercícios [8]



(EMBASA – CESPE 2009)

A figura acima é um exemplo de diagrama de transição de estados, que permite modelar como o sistema responde a eventos internos e externos, especificando o que acontece quando o evento ocorre. Ele é útil para modelar o comportamento de sistemas de tempo real, já que tais sistemas lidam com estímulos do ambiente. A respeito desse assunto e da figura acima, julgue os próximos itens.

Exercícios [8]

[73] É possível criar um diagrama de transição de estados que descreva o ciclo de vida de um objeto em níveis de detalhe arbitrariamente simples ou complexos, dependendo das necessidades, pois não há a obrigação de ilustrar todos os eventos possíveis.

[74] Na figura, A associa-se a uma ação de guarda, e B, a uma ação de transição.

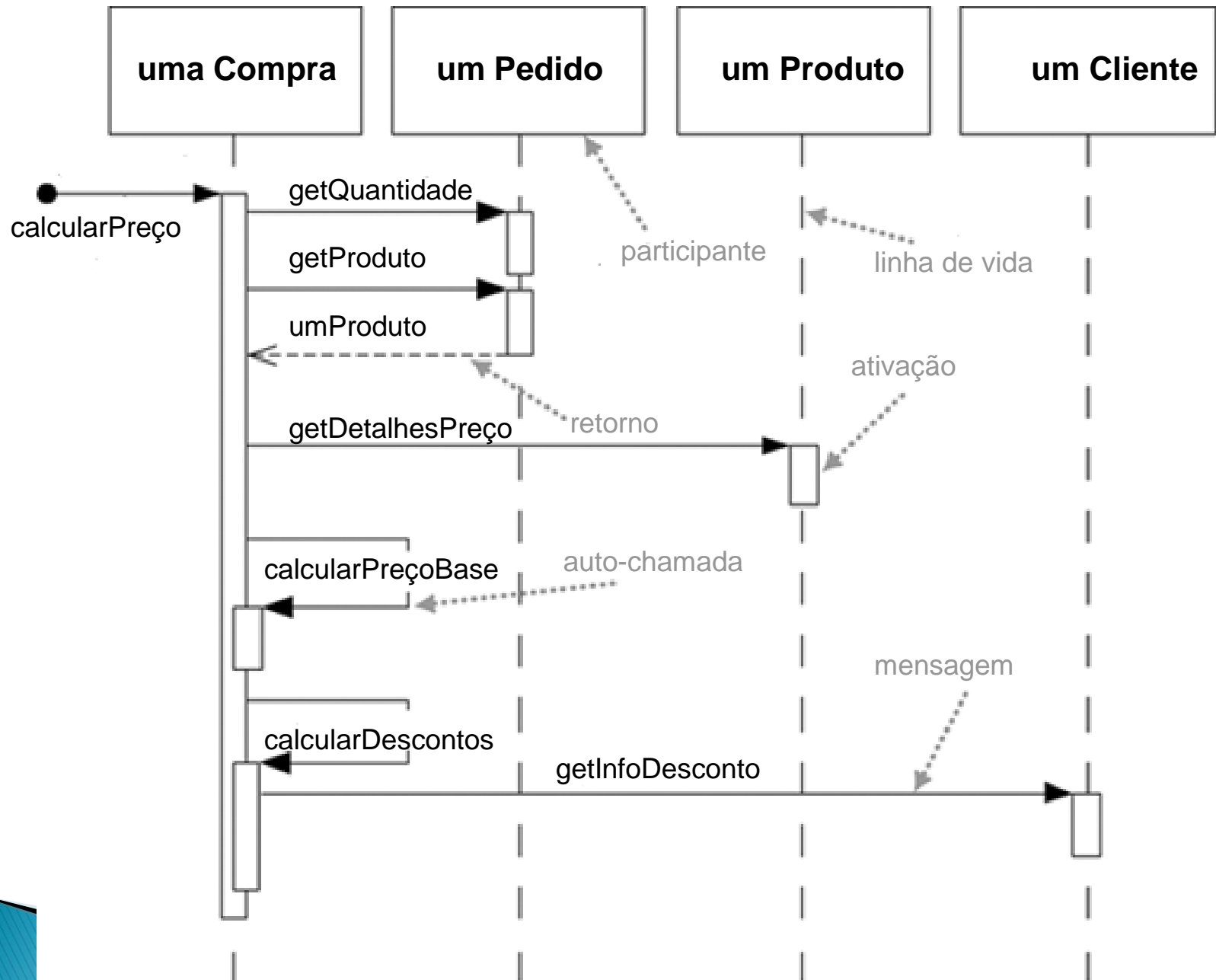
[97] Um diagrama de estado é capaz de mostrar os estados possíveis de um objeto. Além disso, pode mostrar as transações responsáveis pelas suas mudanças de estado.

Diagramas Comportamentais (dinâmicos)

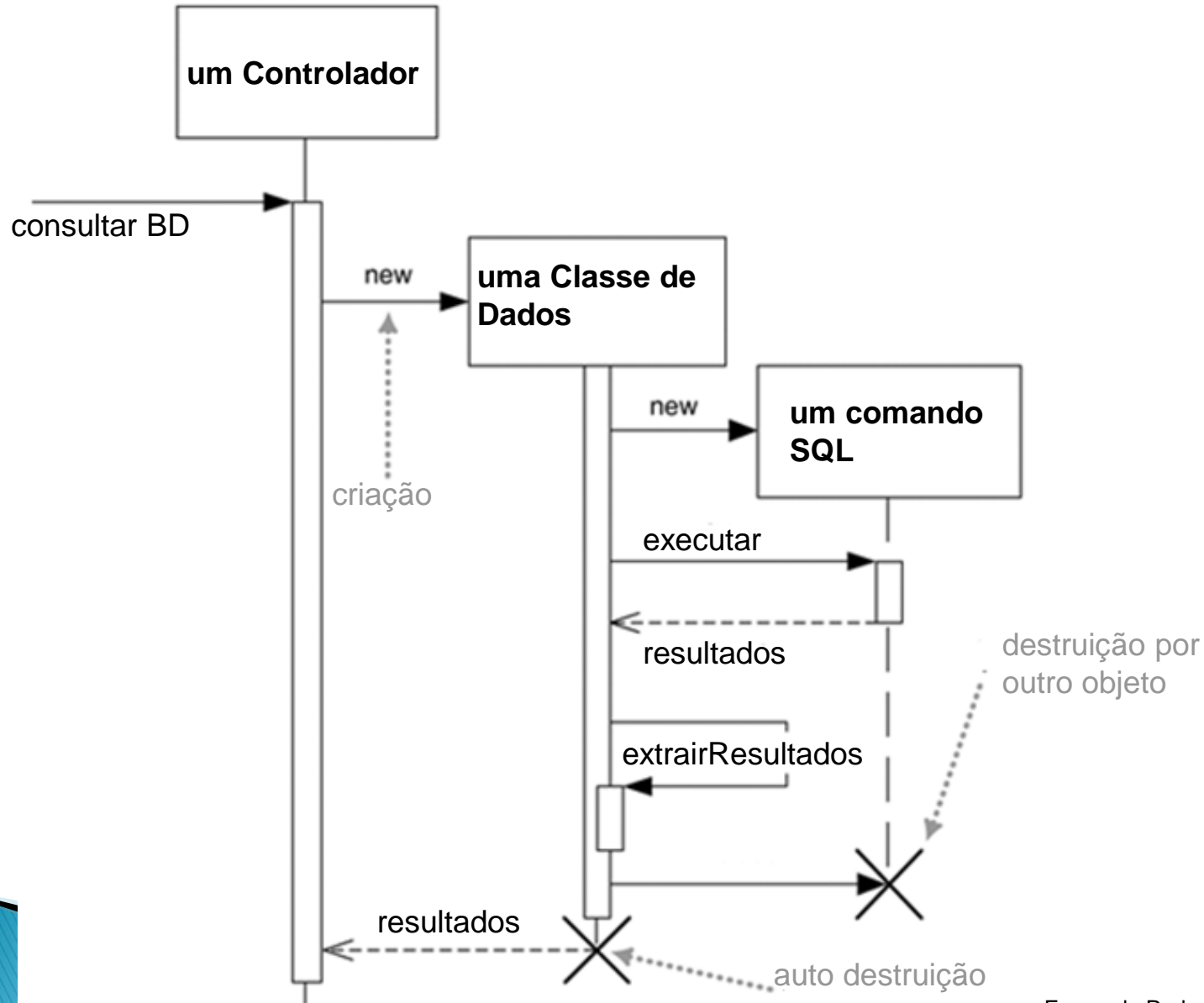
- ▶ Diagrama de Casos de Uso
- ▶ Diagrama de Atividade
- ▶ Diagrama de Máquina de Estados
- ▶ **Diagramas de Interação**
 - Diagrama de Sequência
 - Diagrama de Comunicação
 - Diagrama de Tempo
 - Diagrama de Interação Geral

Diagrama de Sequência

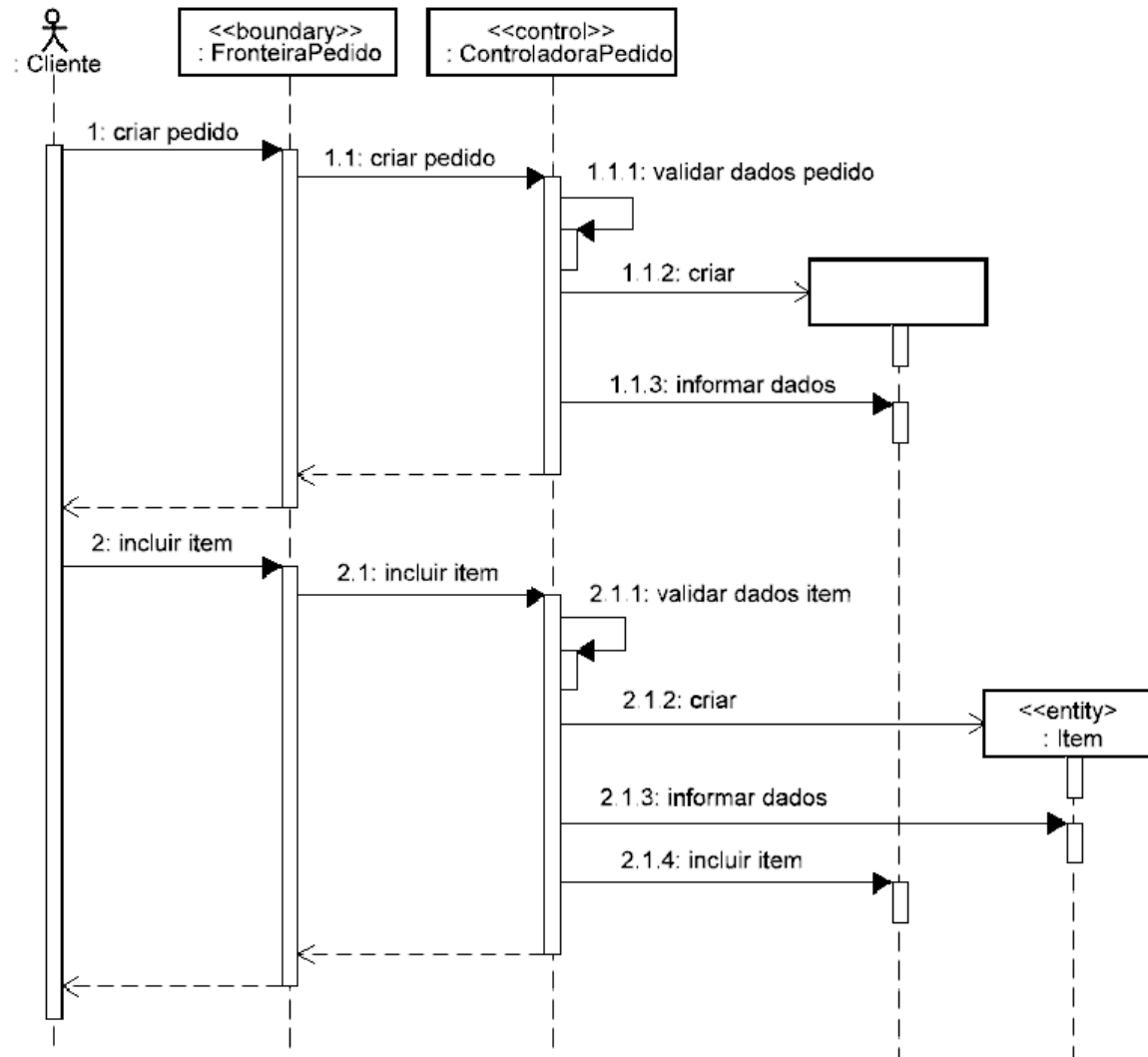
- ▶ Captura o comportamento de um determinado cenário
- ▶ Mostra os objetos e as mensagens trocadas entre eles
- ▶ Enfatiza a **ordem temporal** das mensagens
- ▶ É o diagrama mais utilizado na etapa de Projeto OO (solucionar o problema)



Criação e destruição de objetos



Exercícios [9]



Exercícios [9]

(PETROBRAS – CESPE 2007)

Julgue os itens a seguir, relativos ao diagrama de seqüência UML apresentado acima.

[91] Dois objetos existiam antes da interação e dois foram criados durante a interação. As setas da instância de ControladoraPedido para a instância de FronteiraPedido são retornos de mensagens. Um dos objetos tem nome Pedido e outro, Item. No diagrama, encontram-se representadas as linhas da vida dos objetos e as áreas de ativação das mensagens.

[92] São assíncronas as mensagens da instância de FronteiraPedido para a de ControladoraPedido. Há um erro no diagrama, pois uma instância de uma classe não pode enviar mensagens para ela mesma.

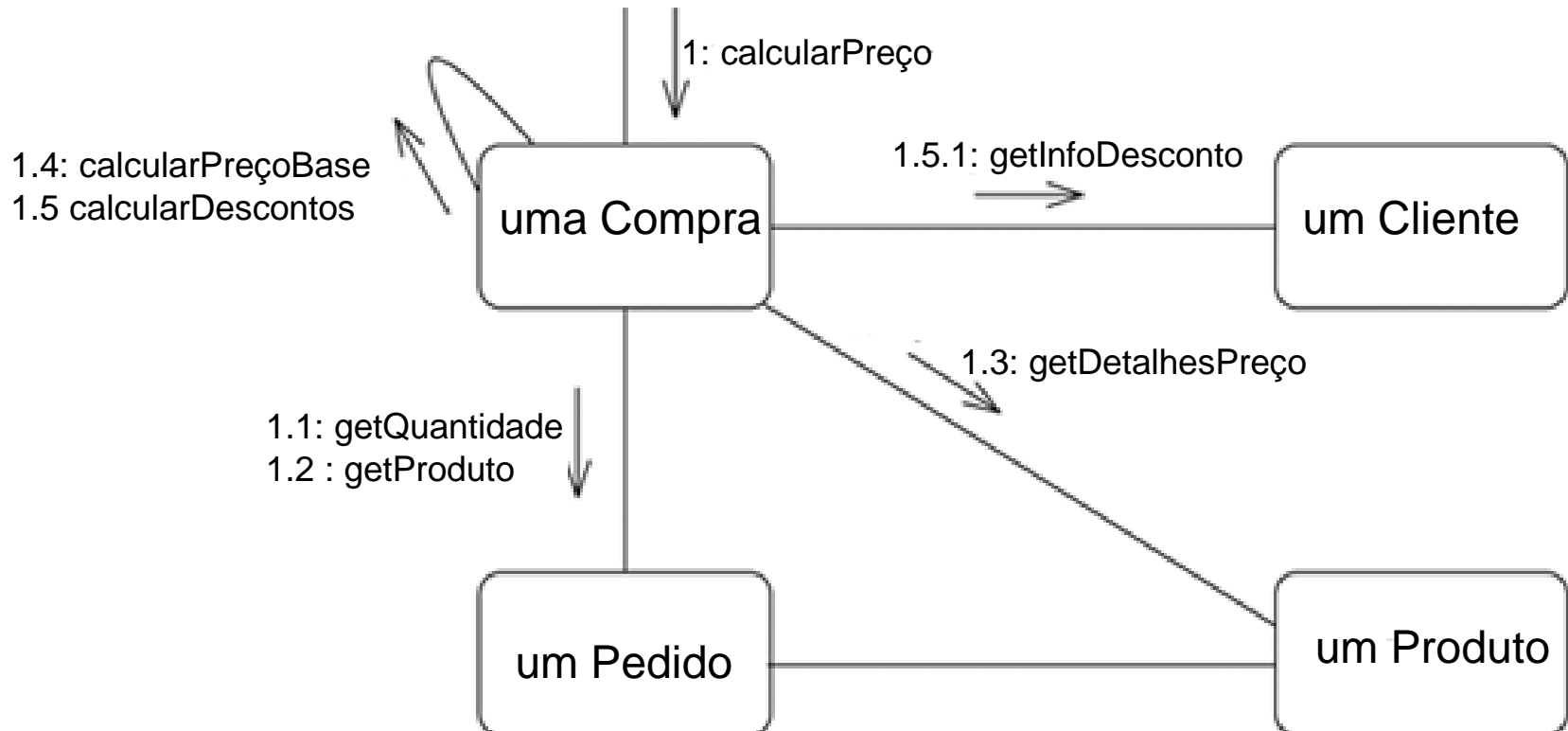
Diagramas Comportamentais (dinâmicos)

- ▶ Diagrama de Casos de Uso
- ▶ Diagrama de Atividade
- ▶ Diagrama de Máquina de Estados
- ▶ **Diagramas de Interação**
 - Diagrama de Sequência
 - **Diagrama de Comunicação**
 - Diagrama de Tempo
 - Diagrama de Interação Geral

Diagrama de Comunicação

- ▶ Captura o comportamento de um determinado cenário
- ▶ Mostra os objetos e as mensagens trocadas entre eles
- ▶ Enfatiza a **ordem estrutural** das mensagens (relacionamentos entre objetos)
- ▶ É equivalente ao diagrama de sequência

Diagrama de Comunicação



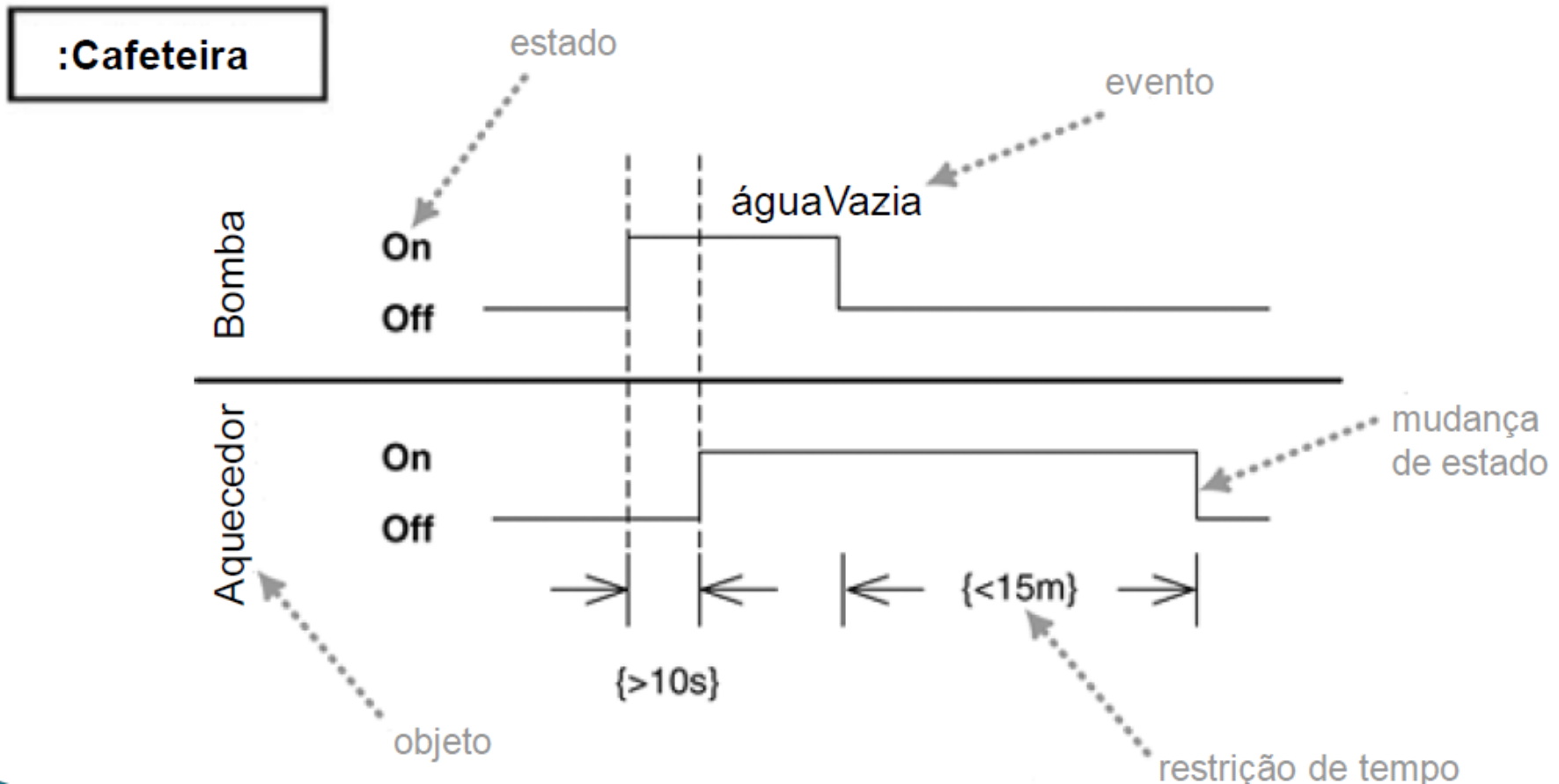
Diagramas Comportamentais (dinâmicos)

- ▶ Diagrama de Casos de Uso
- ▶ Diagrama de Atividade
- ▶ Diagrama de Máquina de Estados
- ▶ **Diagramas de Interação**
 - Diagrama de Sequência
 - Diagrama de Comunicação
 - **Diagrama de Tempo**
 - Diagrama de Interação Geral

Diagrama de Tempo

- ▶ Captura o comportamento de objetos ao longo do tempo e a duração na qual eles permanecem em determinados estados
- ▶ O foco se dá nas **restrições de tempo** das interações
- ▶ É uma mistura entre o diagrama de sequência e o diagrama de máquina de estados

Diagrama de Tempo



Diagramas Comportamentais (dinâmicos)

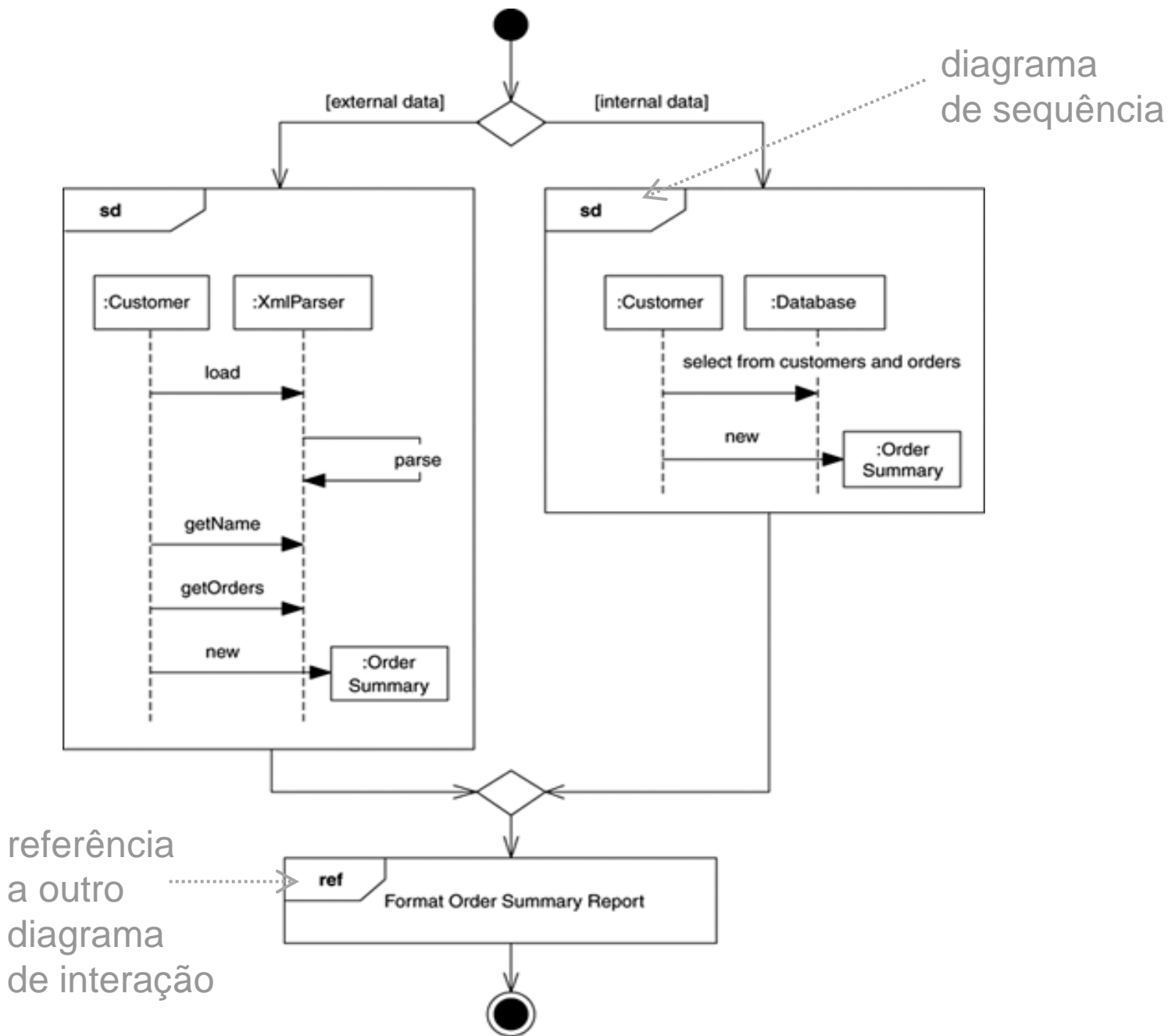
- ▶ Diagrama de Casos de Uso
- ▶ Diagrama de Atividade
- ▶ Diagrama de Máquina de Estados
- ▶ **Diagramas de Interação**
 - Diagrama de Sequência
 - Diagrama de Comunicação
 - Diagrama de Tempo
 - **Diagrama de Interação Geral**

Diagrama de Interação Geral

- ▶ Fornece uma visão geral do controle de fluxo entre objetos
- ▶ É uma mistura entre diagramas de sequência e diagramas de atividade

No exemplo, se o Cliente for externo, os dados são buscados de um XML. Se for interno, os dados são buscados de um banco de dados. A sequência destes dois fluxos é detalhada. Ao final, é gerado um relatório

Diagrama de Interação Geral



Object Constraint Language

- ▶ Linguagem que faz parte da UML e tem o objetivo de desenvolver modelos mais precisos
- ▶ Uma **restrição** (constraint) atua sobre um ou mais valores de um modelo orientado a objetos
- ▶ Vantagens
 - Modelos mais completos, consistentes e precisos
 - Comunicação sem ambigüidade
 - Sintaxe e semântica formais

Object Constraint Language

- ▶ Exemplos de restrições (regras de um sistema de Universidade)
 - “A avaliação de supervisores acadêmicos deve ser maior que a nota dos seus supervisionados”
 - “A bolsa escolar dos alunos depende da sua avaliação acadêmica”
- ▶ Estas regras podem ser escritas em OCL
- ▶ E podem ser transformadas em
 - Código
 - *Scripts* de bancos de dados
 - Outros modelos, etc.

Gabaritos dos Exercícios

- ▶ [1] 78 E, 94 E, 101 E, 31 D
- ▶ [2] 88 C, 89 C, 108 E, 109 C, 110 X (E)
- ▶ [3] 106 E, 40 C, 96 E, 40 C
- ▶ [4] 54 E
- ▶ [5] 105 C
- ▶ [6] 95 C, 85 E, 87 E
- ▶ [7] 88 E, 89 E
- ▶ [8] 73 C, 74 E, 97 C
- ▶ [9] 91 E, 92 E

FIM