



# Análise e Projeto OO

Fernando Pedrosa – [fpedrosa@gmail.com](mailto: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](http://www.wthreex.com/rup)

# Análise OO

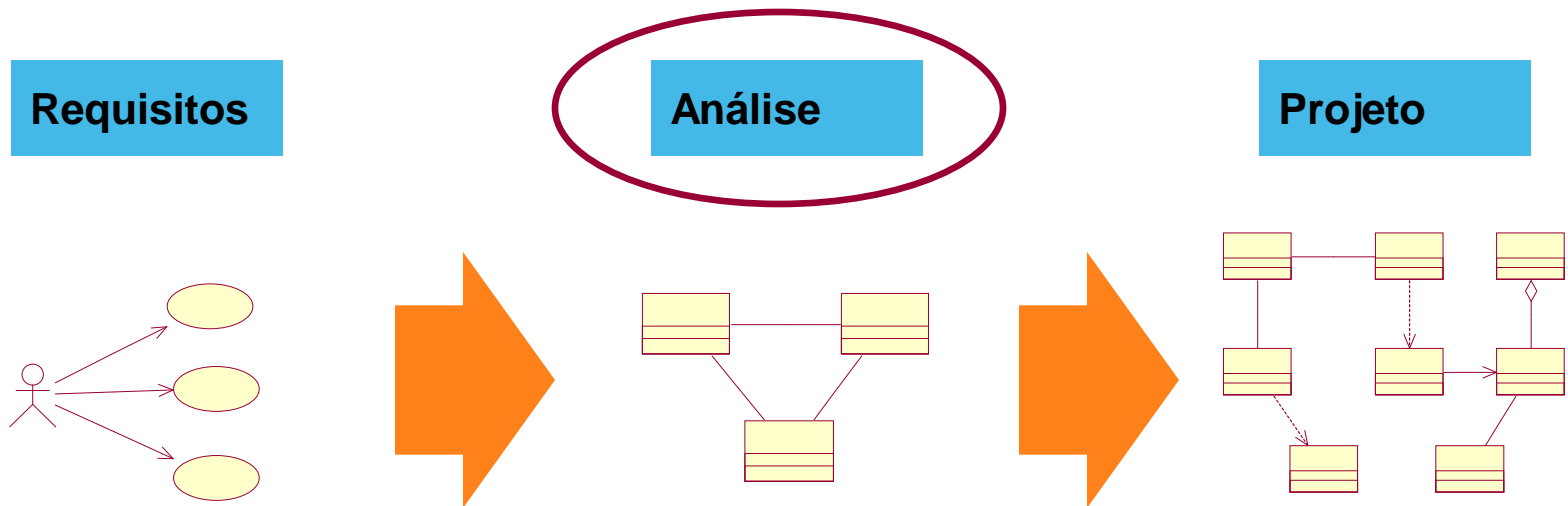
# Análise Orientada a Objetos

- ▶ 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?

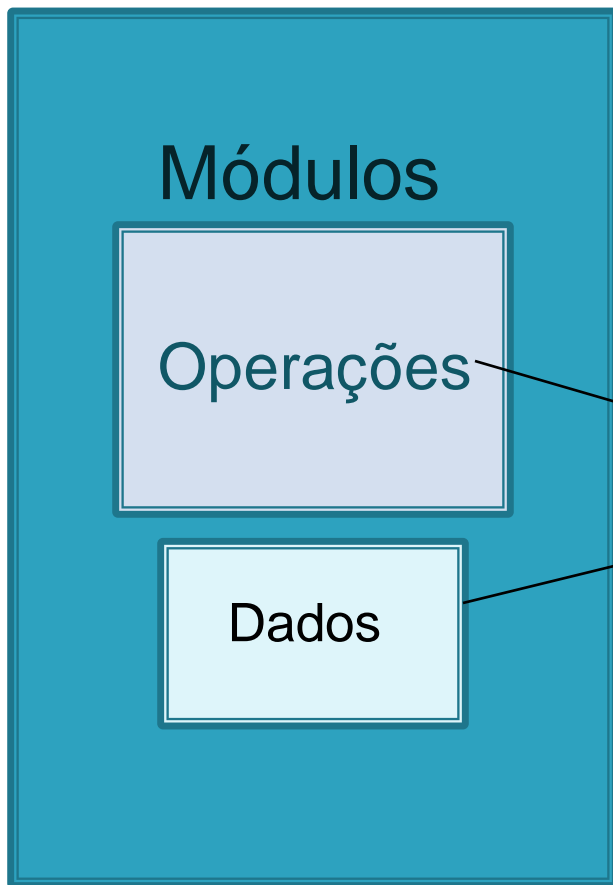
- ▶ Para entendermos Análise OO, precisamos estudar o Paradigma OO
- ▶ Vamos estudar
  - Objetos
  - Classes
  - Comportamentos
  - Atributos
  - Relacionamentos
  - ...

# Paradigma OO

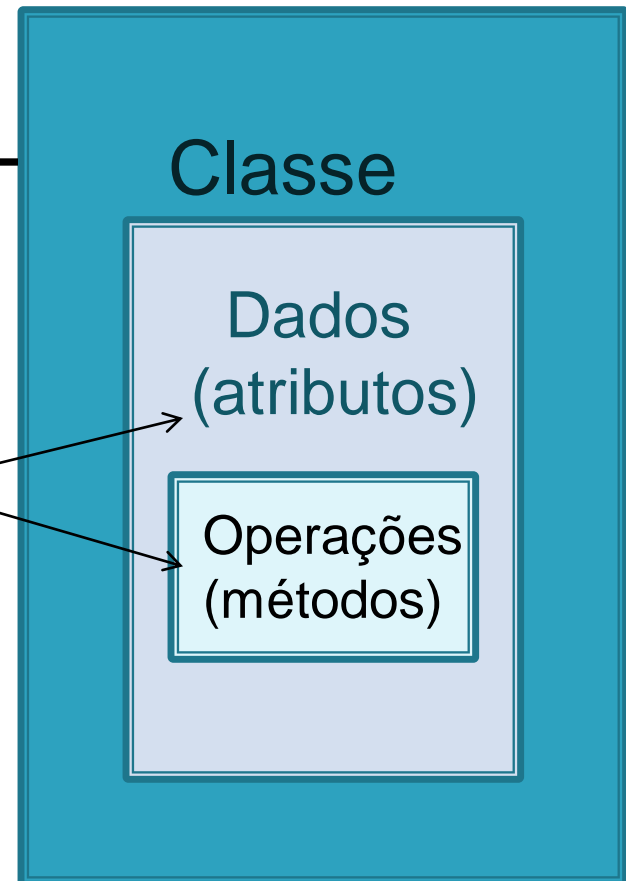
# Paradigma OO

- ▶ 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)

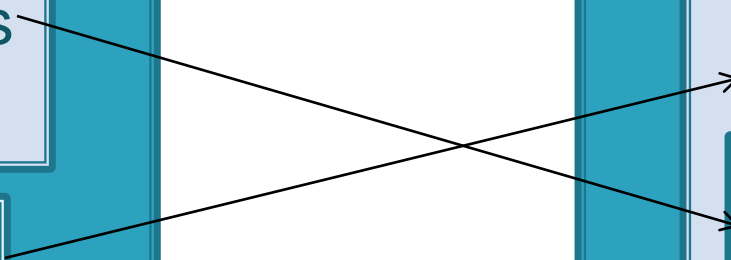
## Foco na Estrutura



## 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

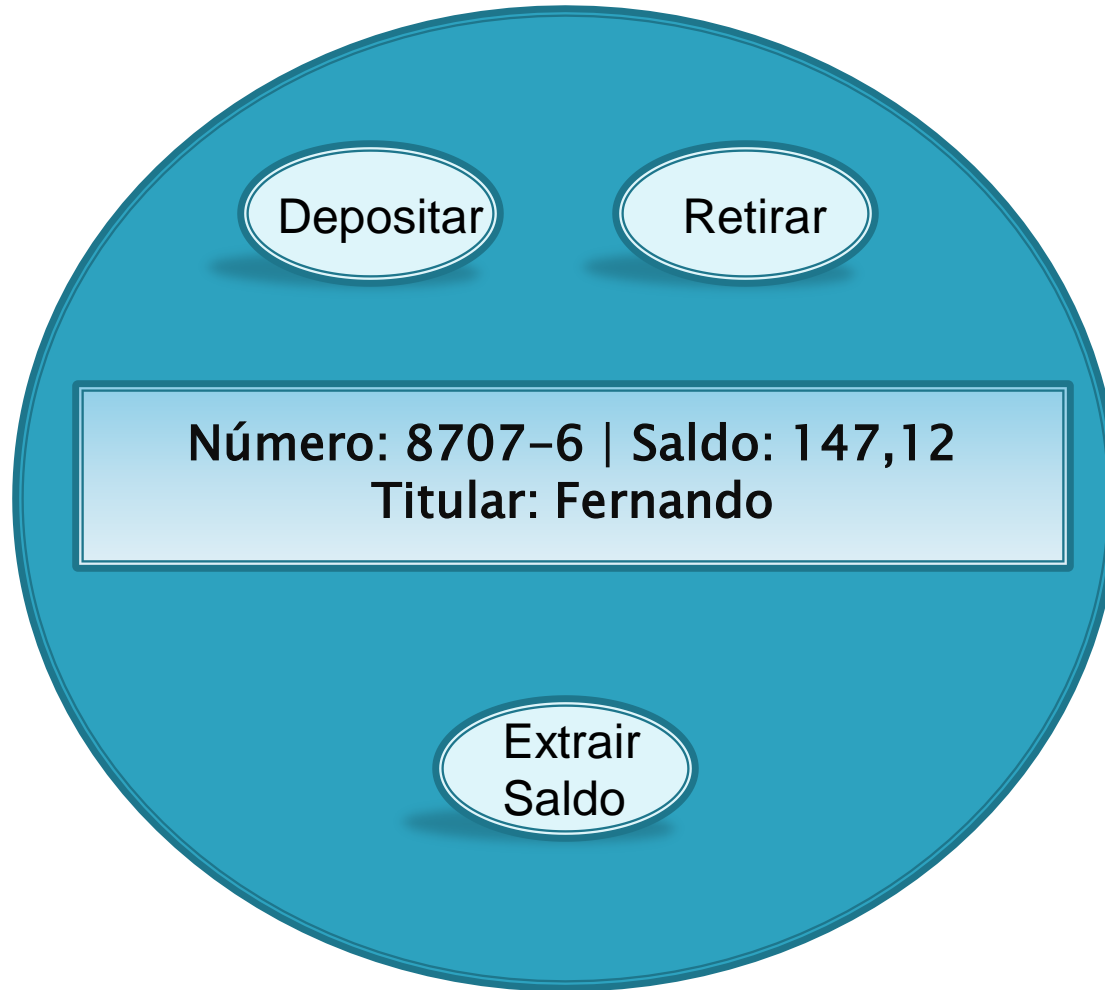
## ▶ Estado

- 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

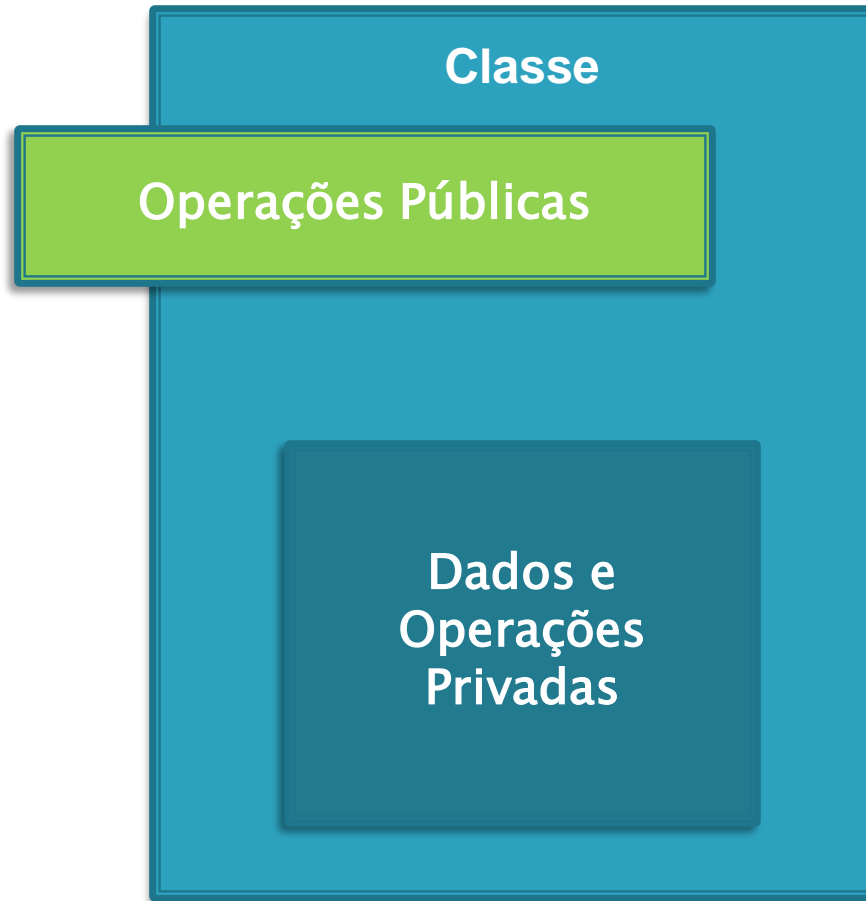
# Componentes de uma Classe

- ▶ 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

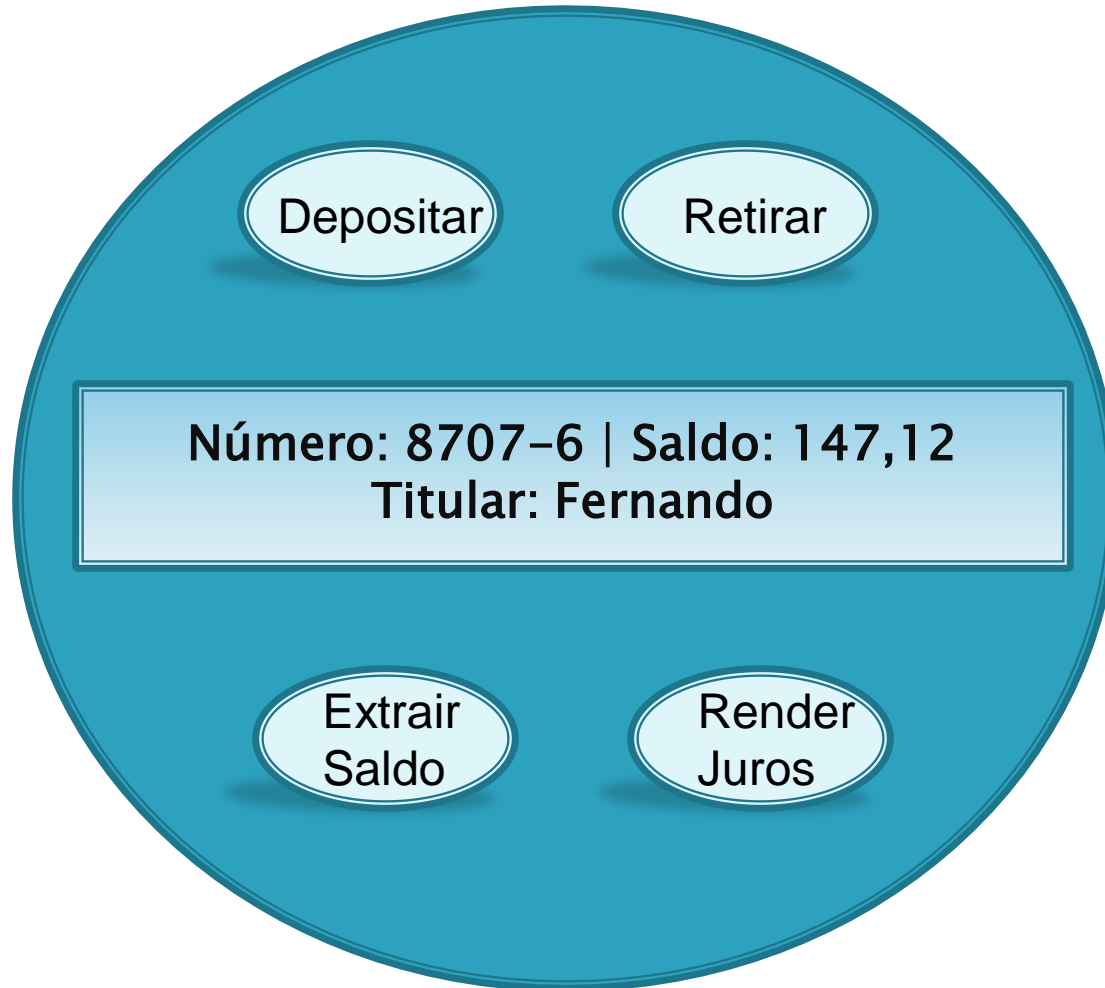
# Encapsulamento



# 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...

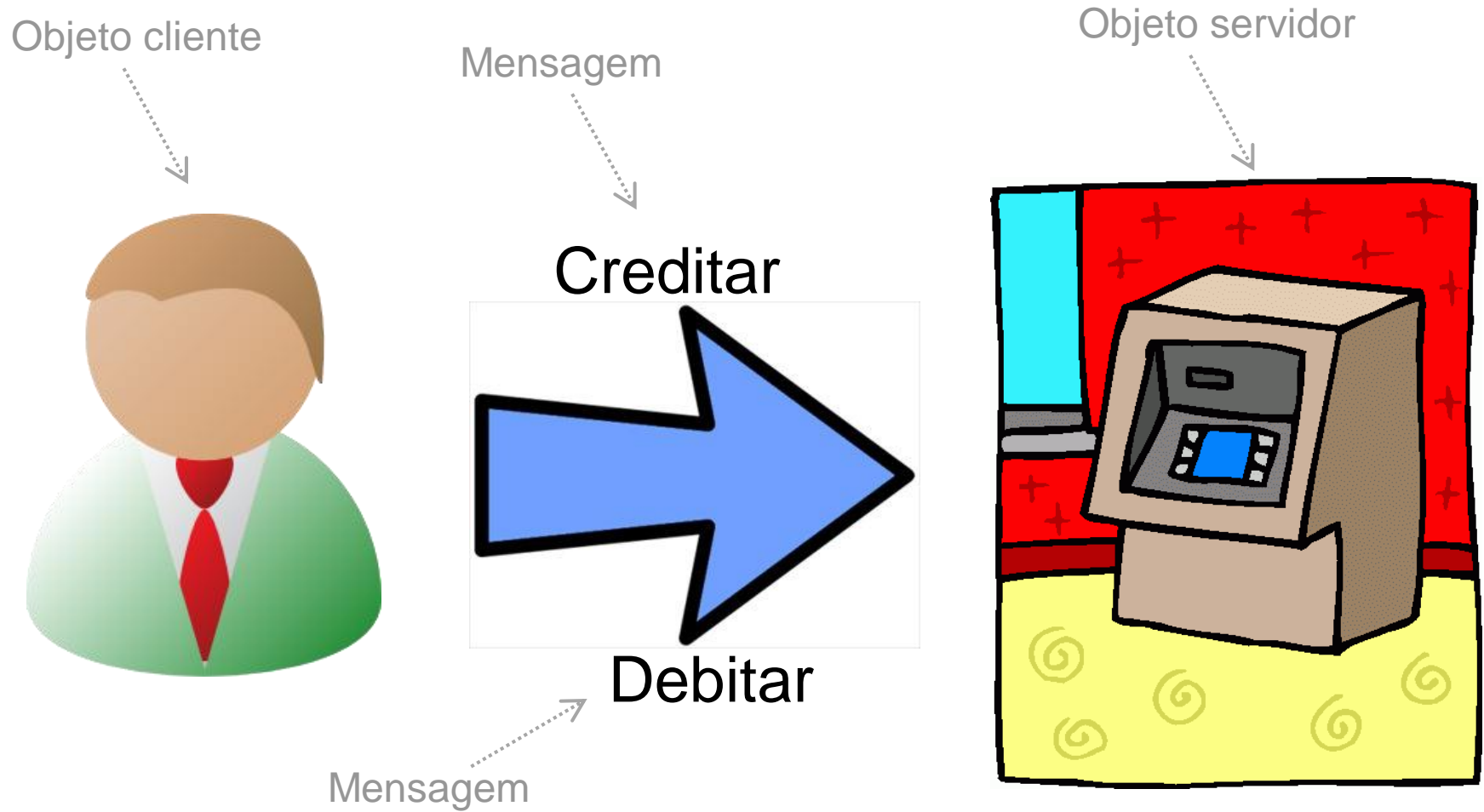
# Objeto Poupança



# 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

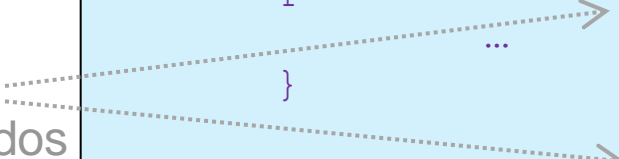
- ▶ “Várias formas”
- ▶ Denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem
- ▶ Dois tipos
  - Polimorfismo Estático ou Sobrecarga
  - Polimorfismo Dinâmico ou Sobreposição

# 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

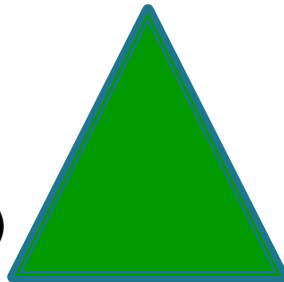
# Polimorfismo Dinâmico

Forma calcularArea()



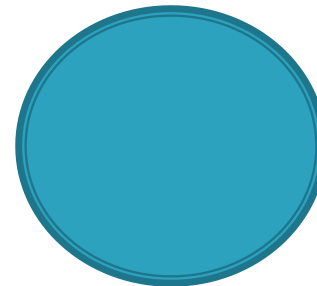
Triângulo

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?

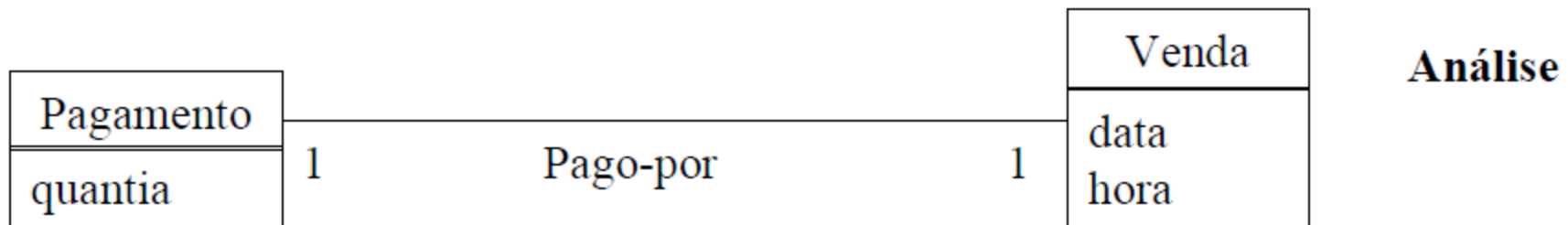
- ▶ É um modelo de objetos que descreve a realização dos casos de uso e que funciona como uma abstração do Modelo de Design
- ▶ Contém as classes de análise (Modelo de Classes) e qualquer artefato associado
- ▶ O Modelo de Classes evolui durante as iterações do projeto, incrementando novos detalhes às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

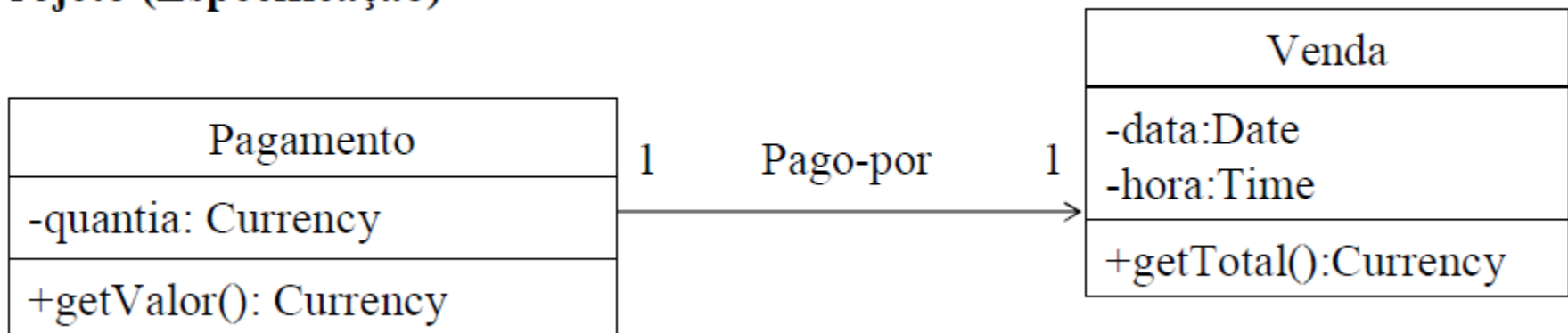
- ▶ Representa as classes no domínio do negócio em questão
- ▶ Não leva em consideração restrições inerentes à tecnologia específica a ser utilizada na solução de um problema



# 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

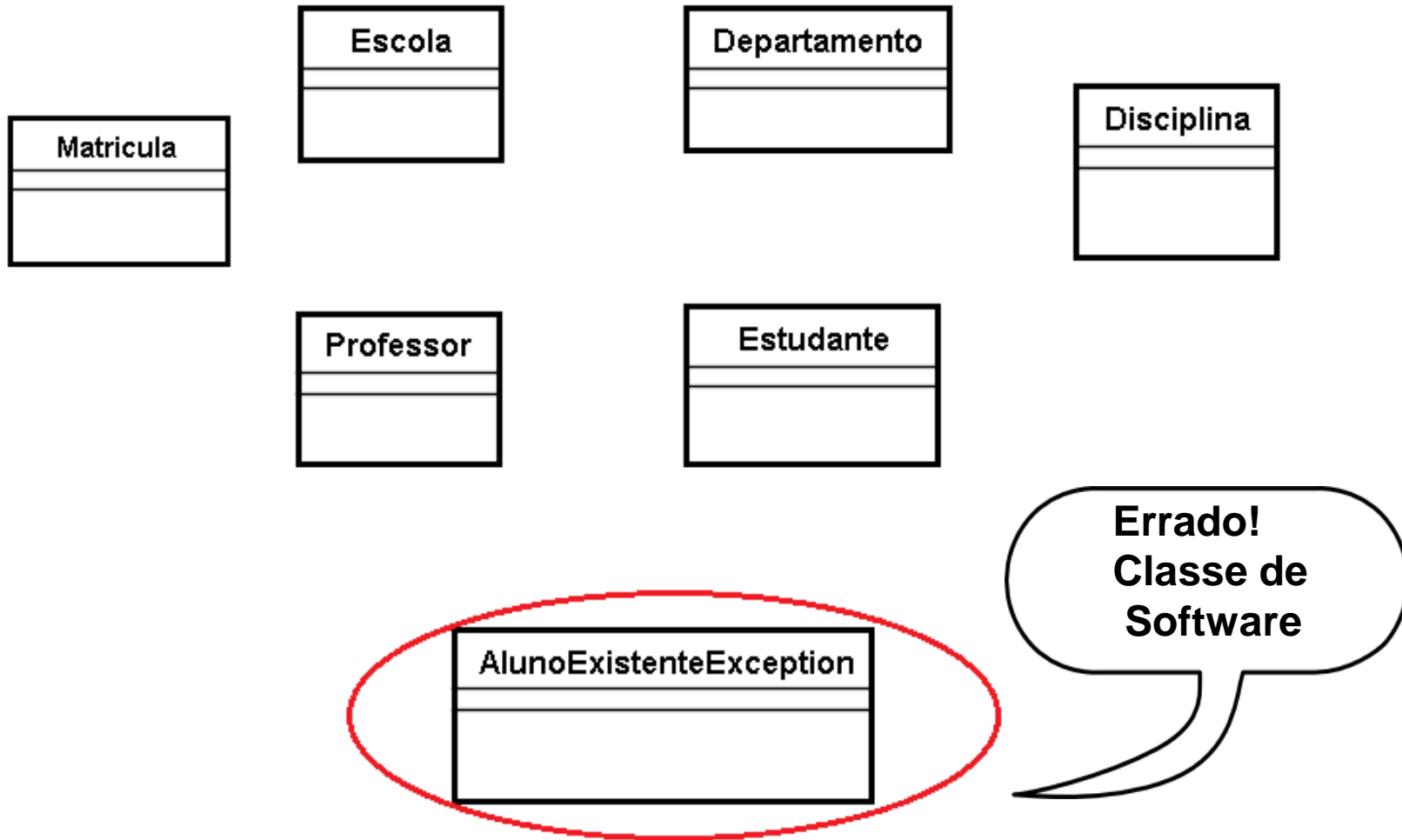
```
public class Pagamento {  
    private Currency quantia;  
    ...  
    public Currency getValor() {  
        return quantia;  
    }  
}  
  
    public class Venda {  
        private Date data;  
        private Time hora;  
        private Pagamento pagamento;  
        ...  
        public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

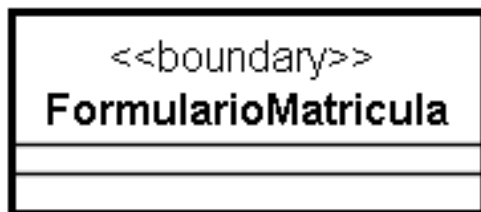


# Passo 1: Identificando classes

- ▶ 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

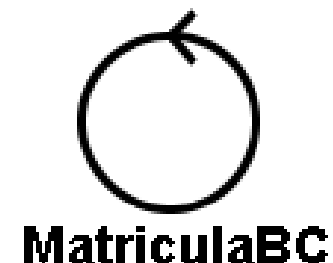


# Classes de Controle

- ▶ 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**:

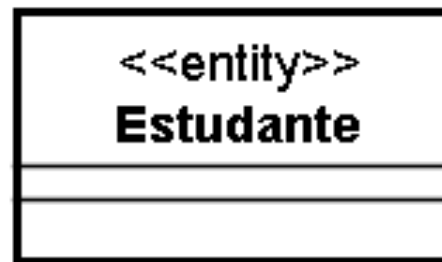


ou



# 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:

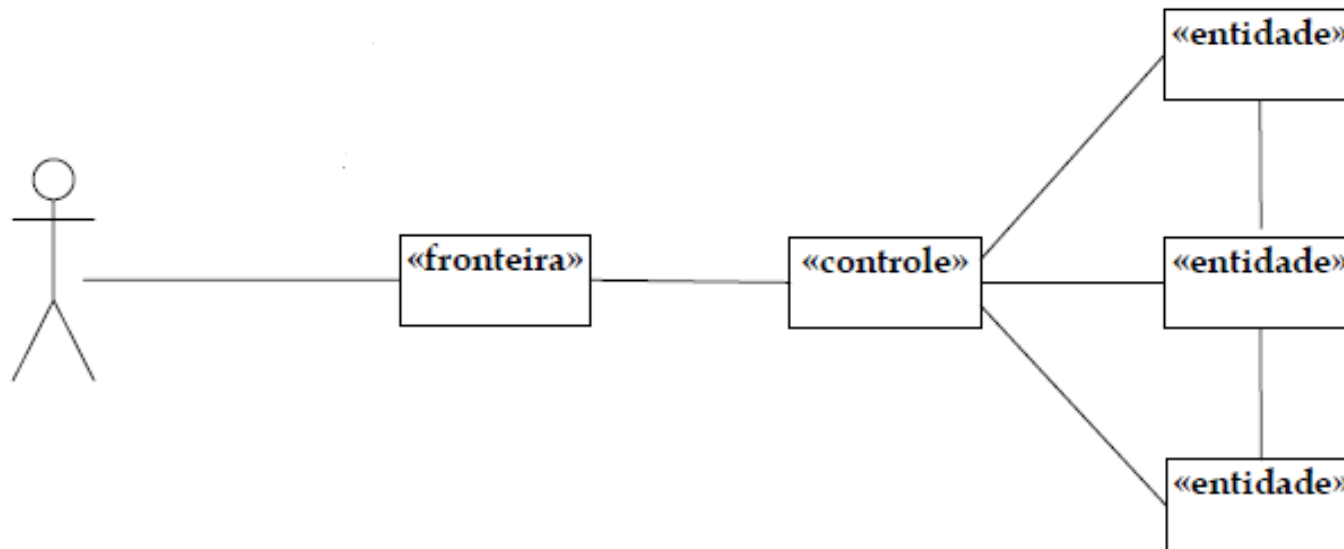


ou



# Fronteira x Controle x Entidade

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



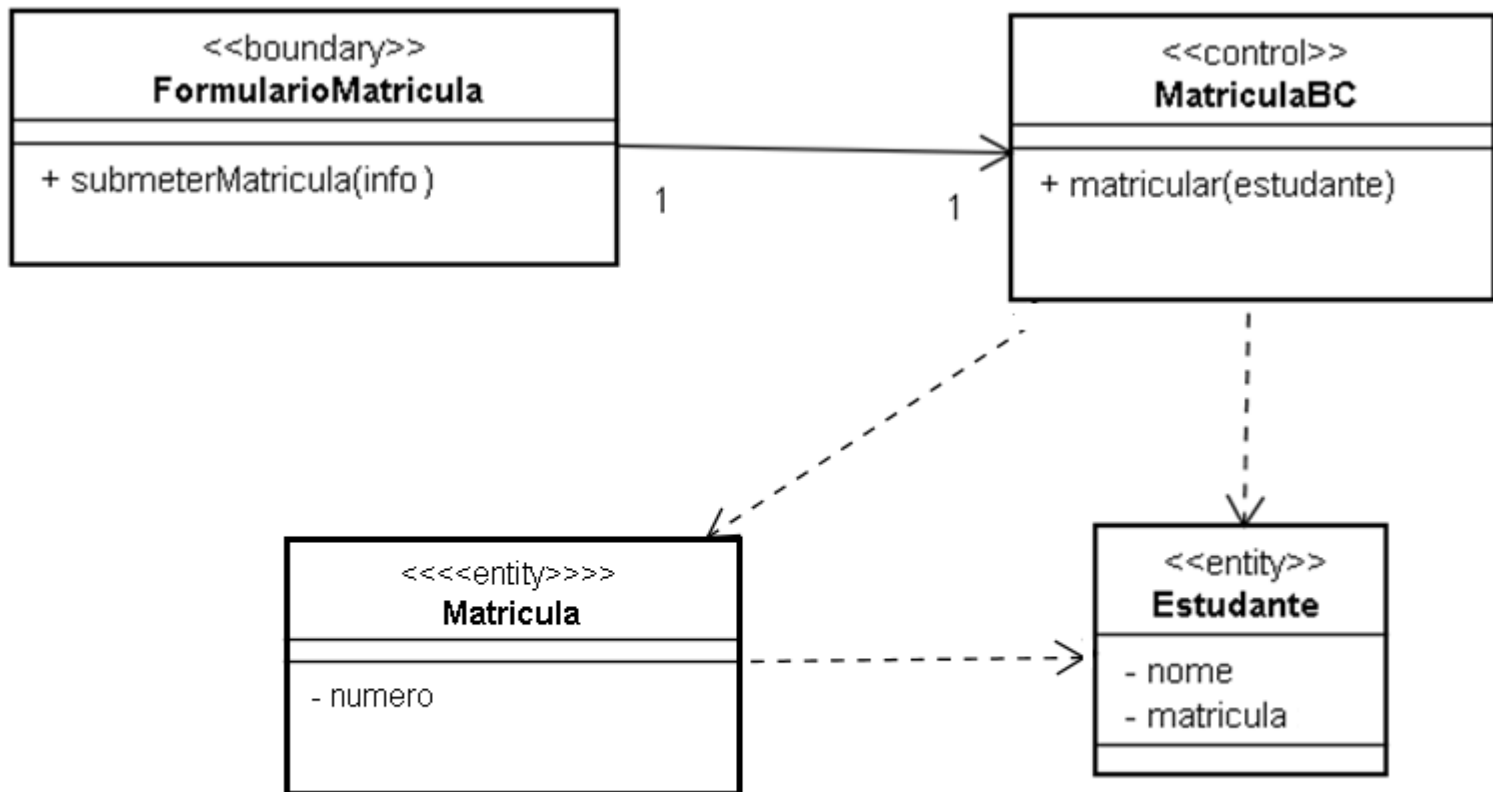
- ▶ Lembra algo? **MVC!**

# 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

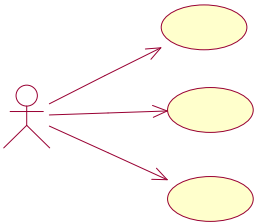
- ▶ Após a etapa de análise, temos o primeiro modelo do sistema
  - Definimos “o que” o software deve fazer
- ▶ Queremos agora detalhar este modelo, para gerarmos facilmente a implementação do sistema
  - Definimos “COMO” o software atenderá os requisitos analisados
- ▶ Este modelo é chamado de Modelo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

# Análise x Projeto

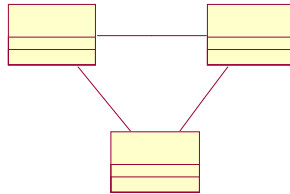
- ▶ Abstrato x Concreto
- ▶ Independente x Dependente da tecnologia de implementação
- ▶ Simples x Detalhado
- ▶ Modelos por caso de uso x Unificação em um único modelo

# Projeto Orientado a Objetos

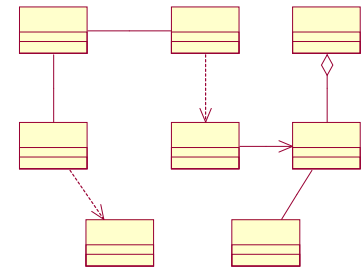
**Requisitos**



**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)

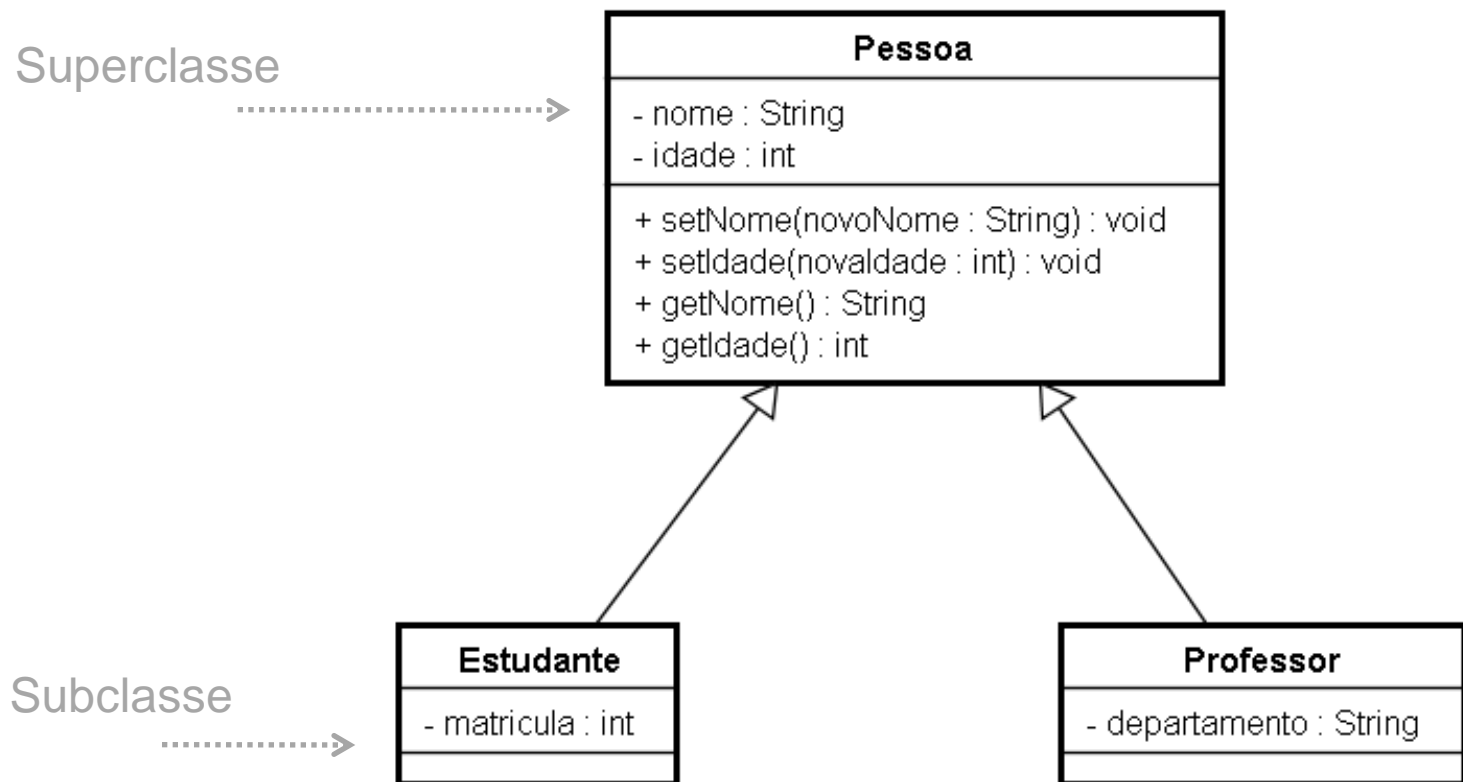
# Refinando o Modelo de Classes

# 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

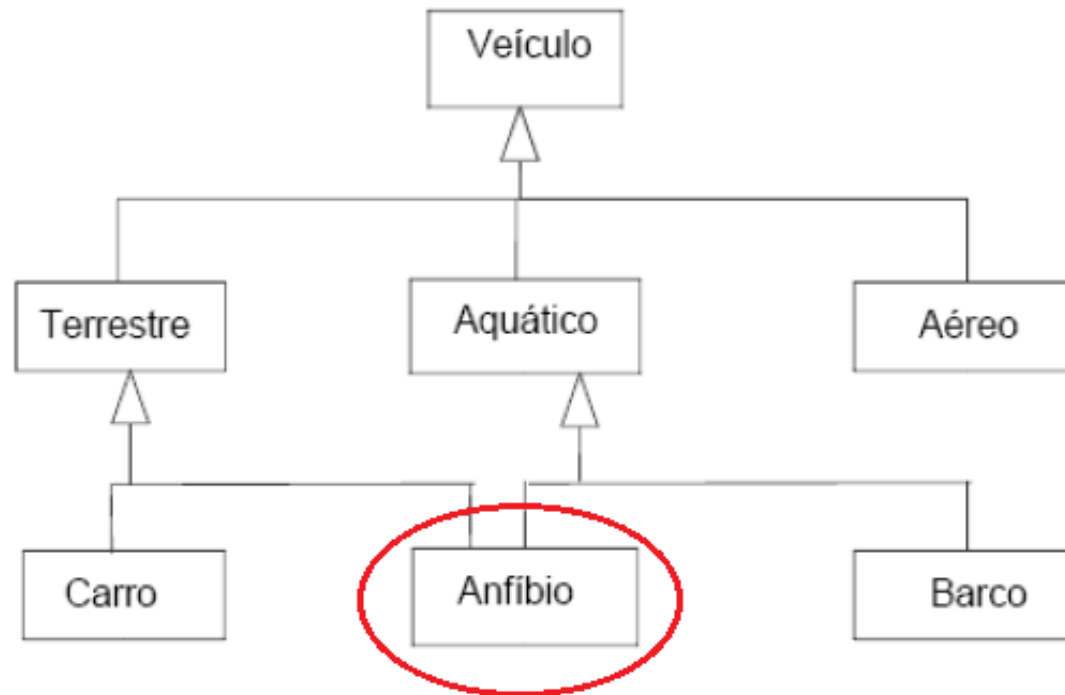
# Herança Simples

- ▶ 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.

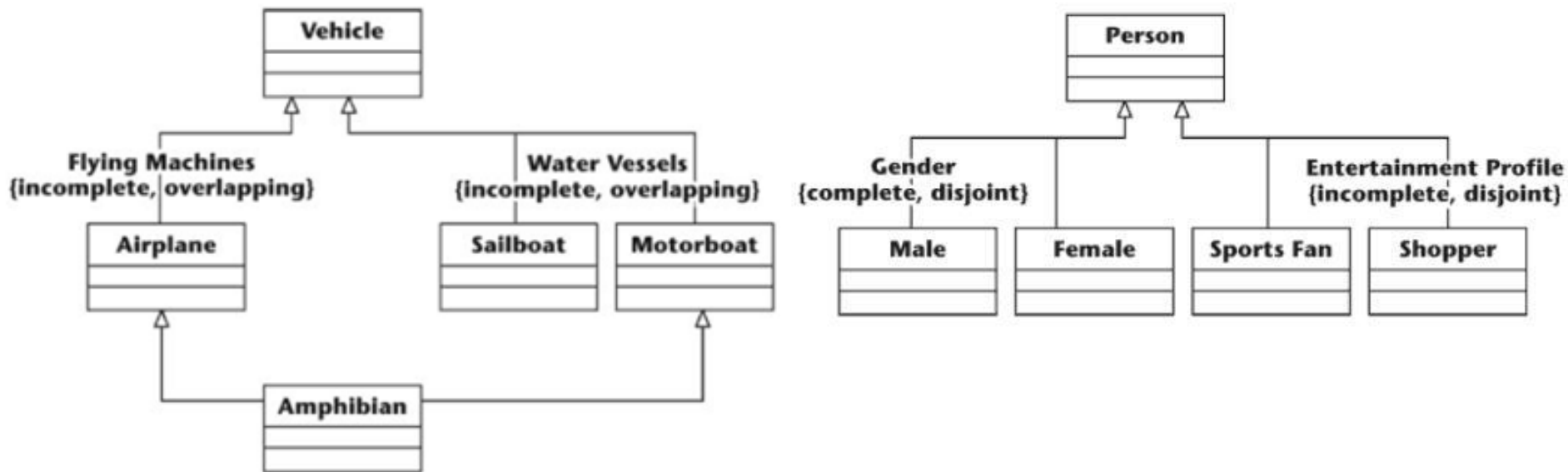
## ▶ Disjunta

- 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.

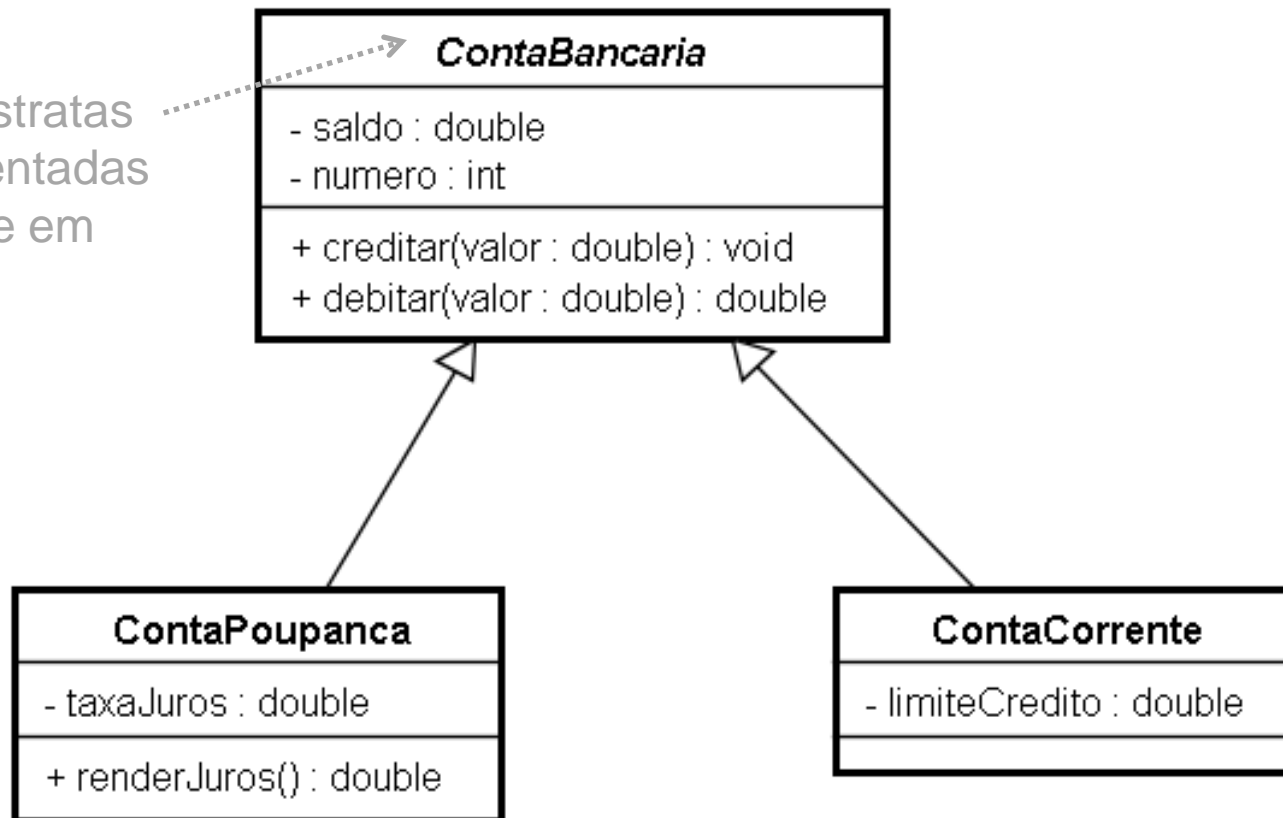


# Classe Abstrata

- ▶ 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*

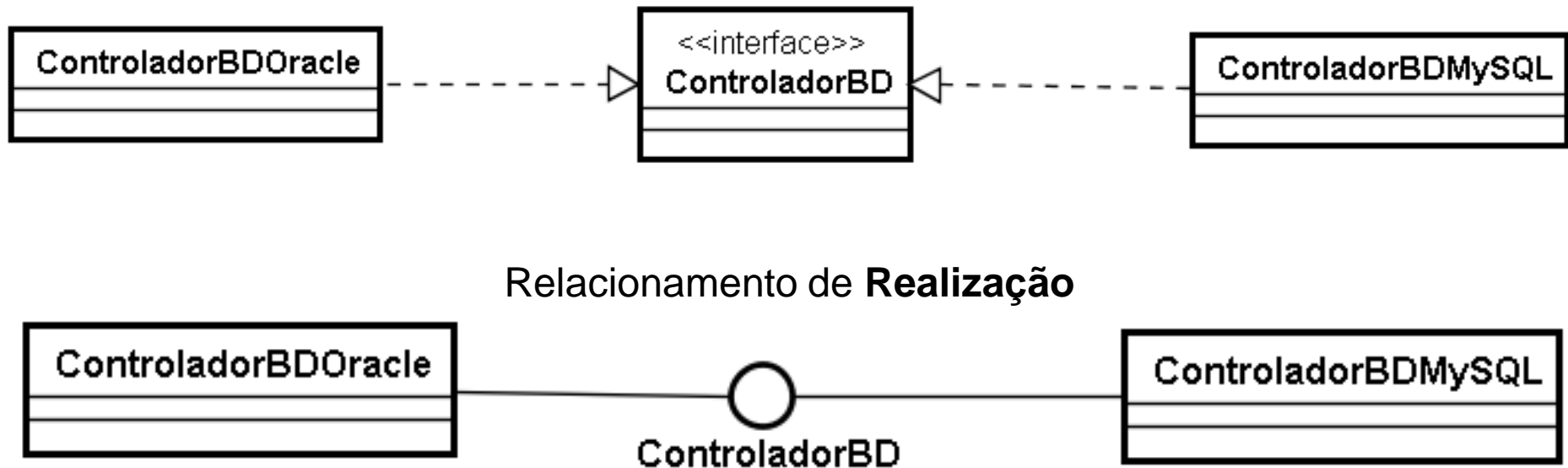


# Interface

- ▶ 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

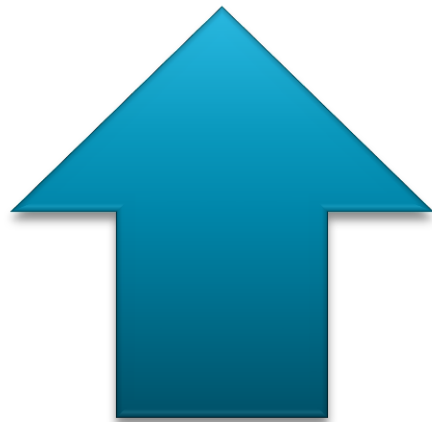
- ▶ Arquitetura de software é a organização ou a estrutura dos componentes significativos do sistema que interagem por meio de interfaces
- ▶ É composta de:
  - Componentes de software
  - Suas propriedades visíveis externamente
  - O relacionamento entre os componentes
- ▶ Forma a espinha dorsal para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

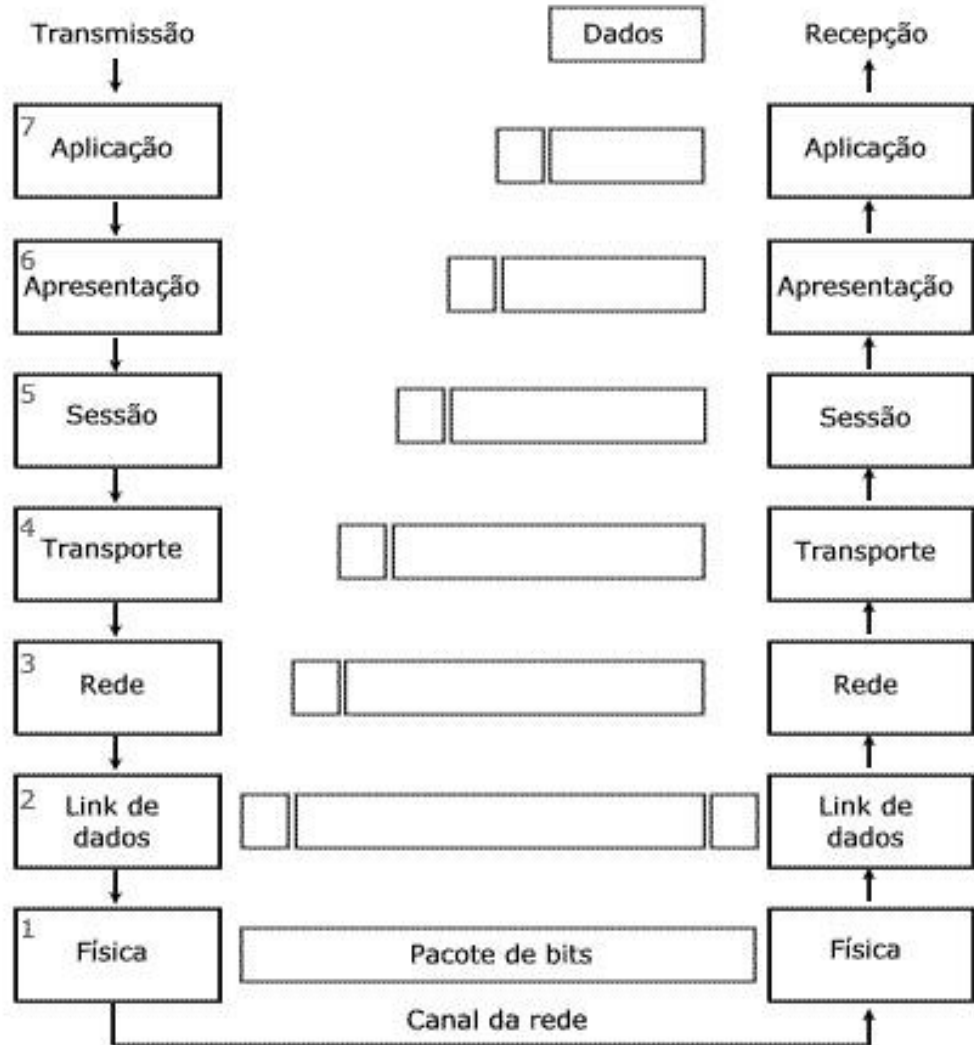
- ▶ Separação de responsabilidades
- ▶ Decomposição de complexidade
- ▶ Encapsulamento de implementação
- ▶ Maior reuso e extensibilidade

## 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

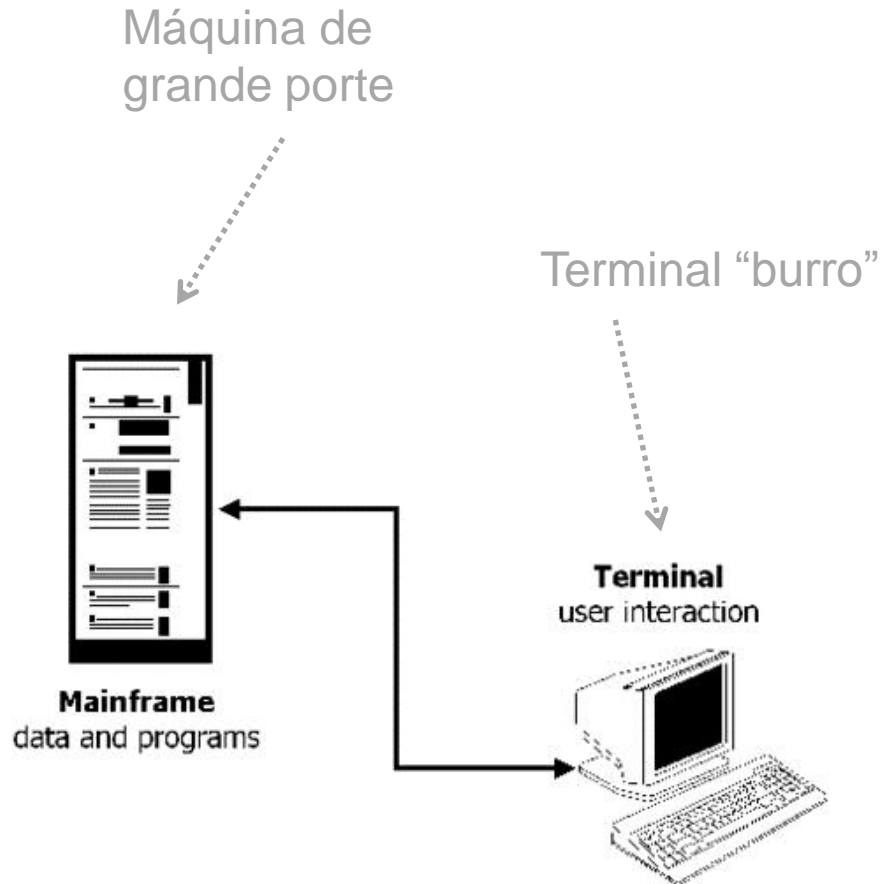
## ▶ Arquitetura Monolítica

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

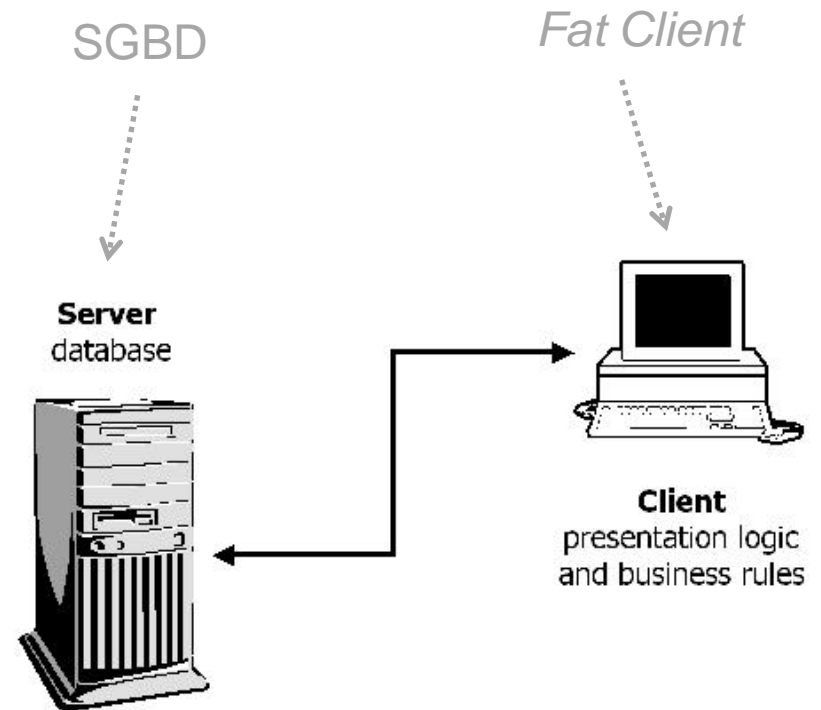
## ▶ Arquitetura em duas camadas (*two-tier*)

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

# Arquitetura Monolítica



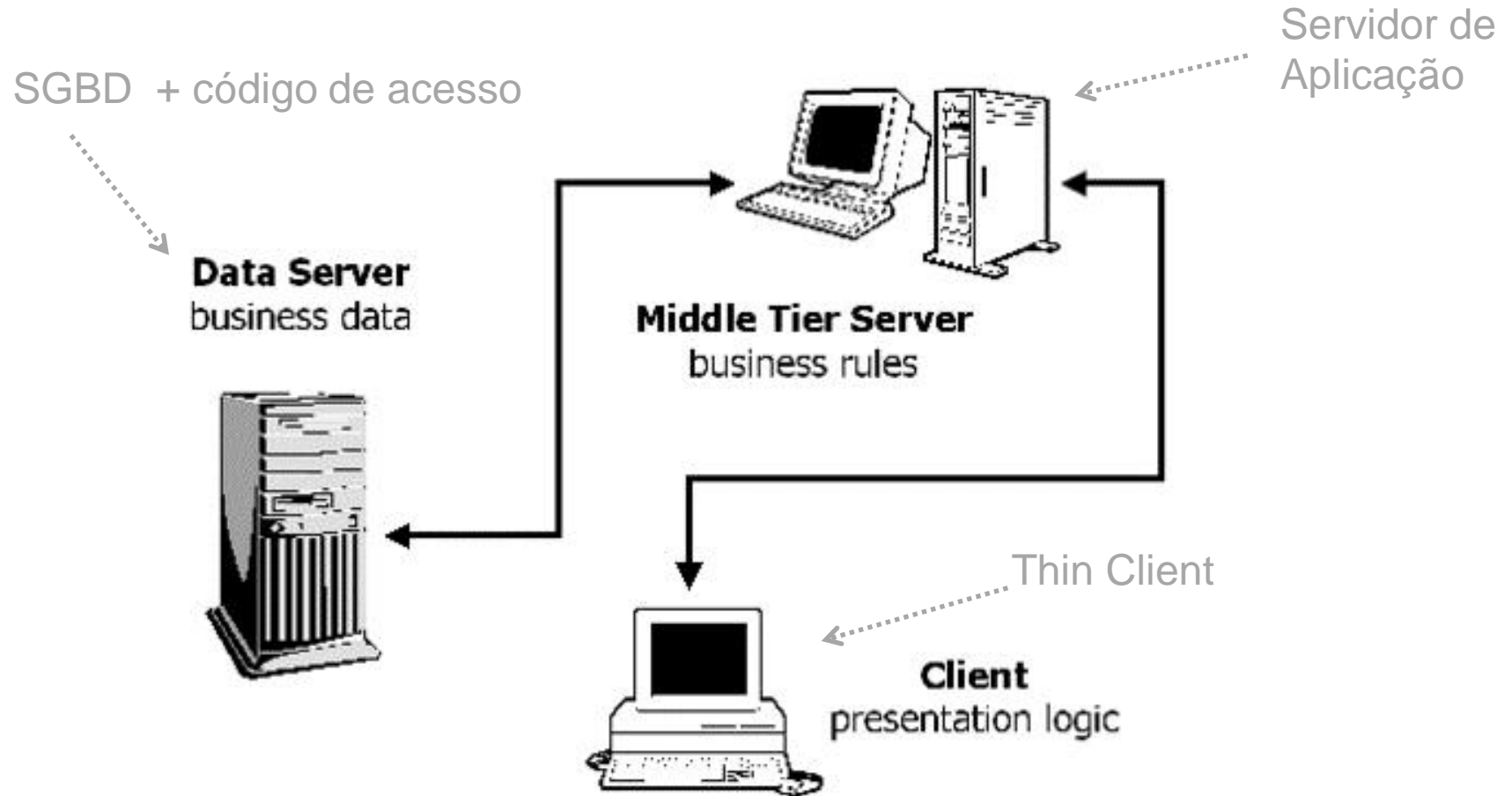
# Two-tier Architecture



# 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

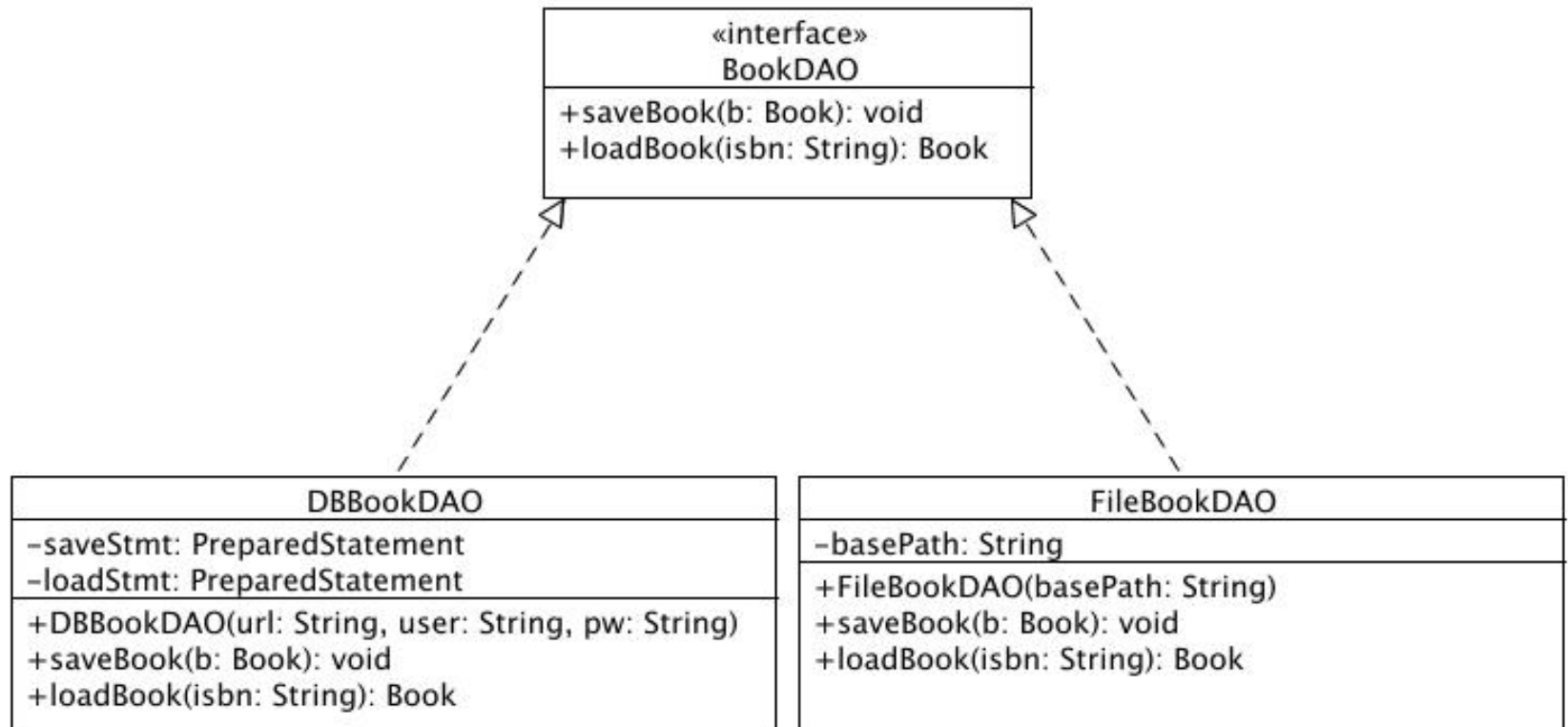
- ▶ Coordena a aplicação, processa comandos, toma decisões lógicas, faz avaliações e implementa as regras de negócio
- ▶ É **inerente** ao domínio (negócio) da aplicação
- ▶ Vários protocolos podem ser utilizados para ligar esta camada às outras duas
  - Sockets, HTTP, TCP/IP, etc. (Apresentação)
  -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

# Camada de Acesso a Dados

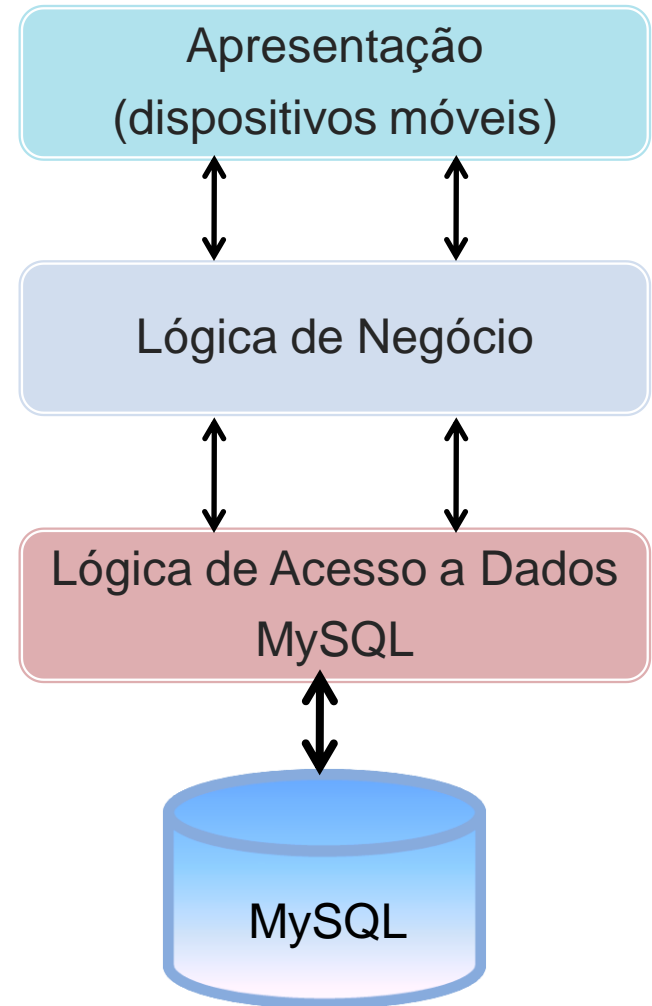
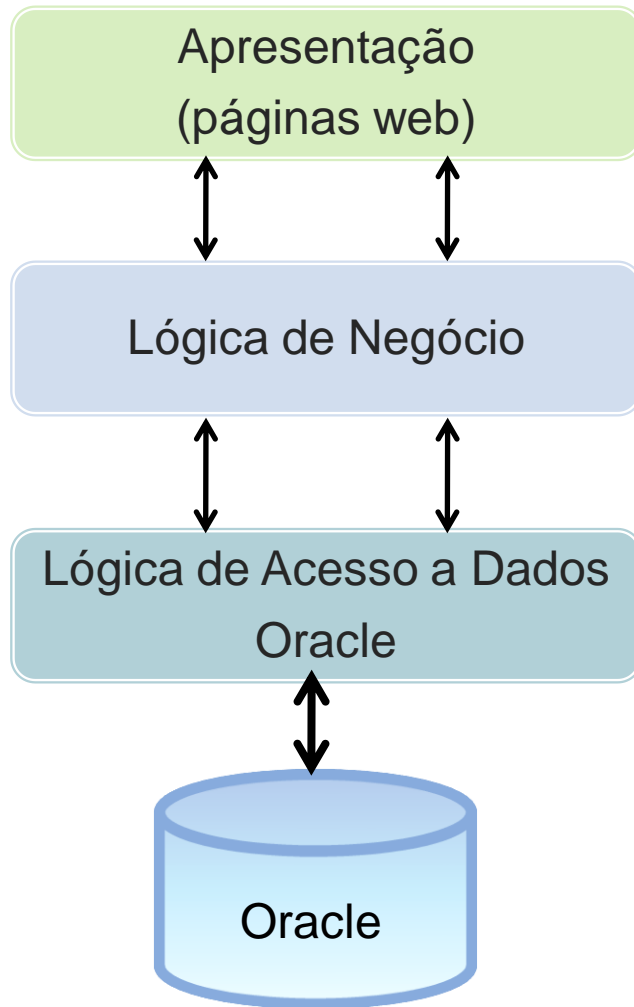
- ▶ Contém o código responsável por armazenar e recuperar dados de uma base de dados ou sistema de arquivos
- ▶ Normalmente há uma sub-camada (interface) dentro desta camada que abstrai o mecanismo de persistência
  - O famoso padrão DAO (*Data Access Object*) é utilizado aqui.

# Camada de Acesso a Dados

## *Data Access Object*

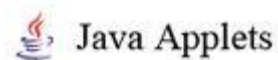


# Exemplo – substituição de camadas



# Exemplo – camadas e seus protocolos/tecnologias

## Apresentação



## Lógica do Negócio



## Acesso a Dados



# Exercícios [4]

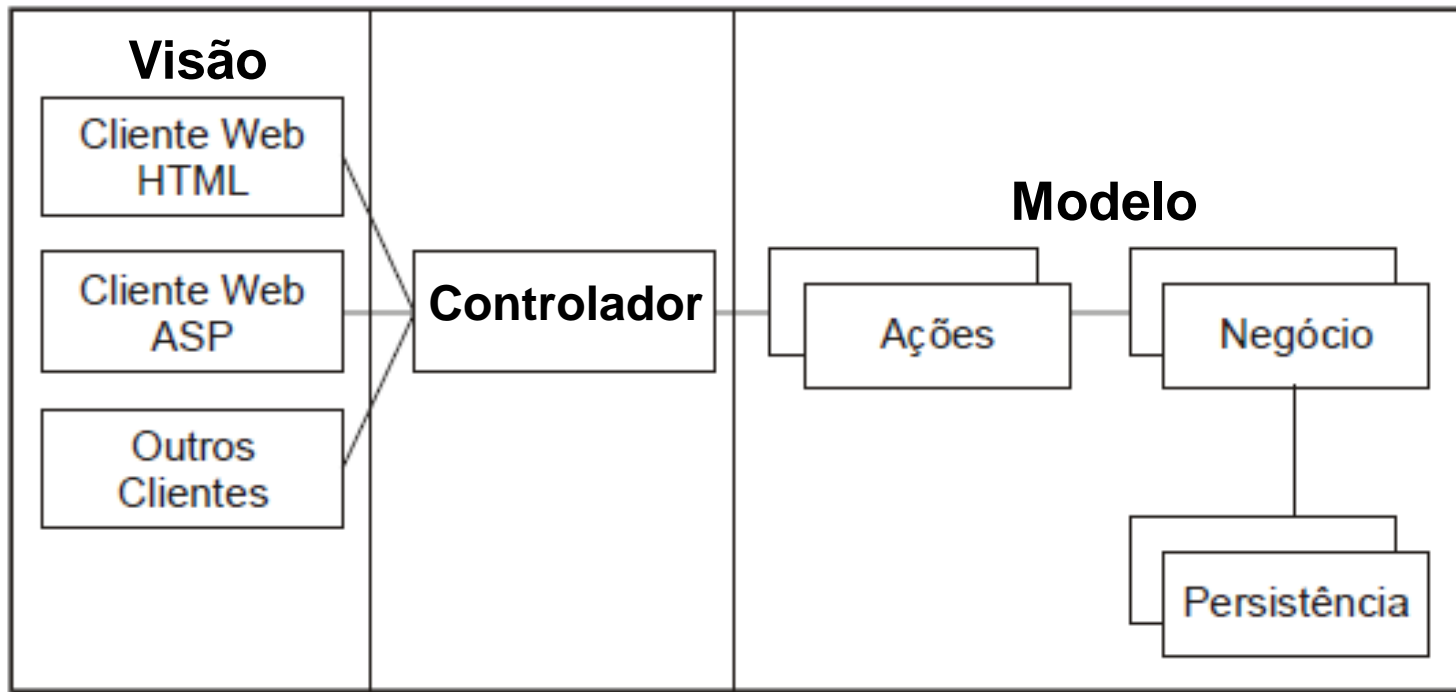
(PETROBRAS – CESGRANRIO 2010)

[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

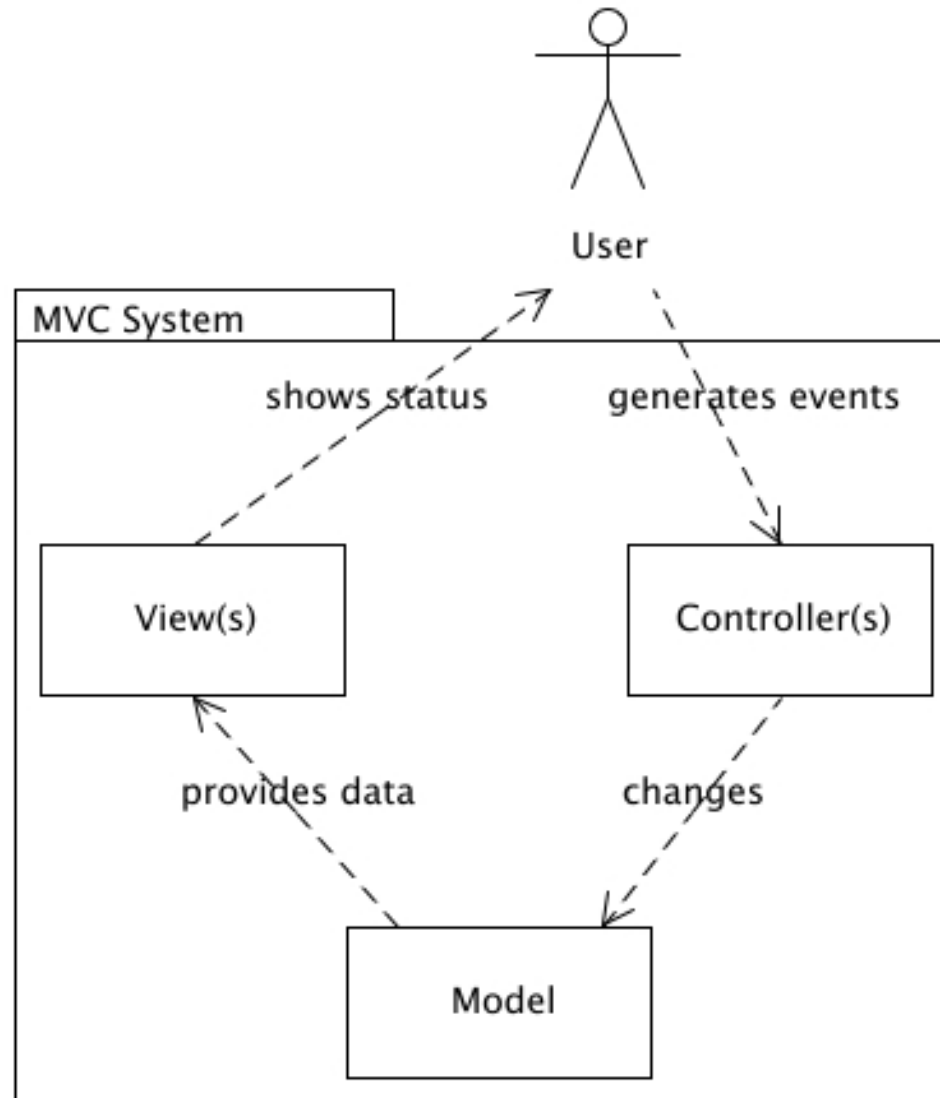
# Camada de Controle

- ▶ 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

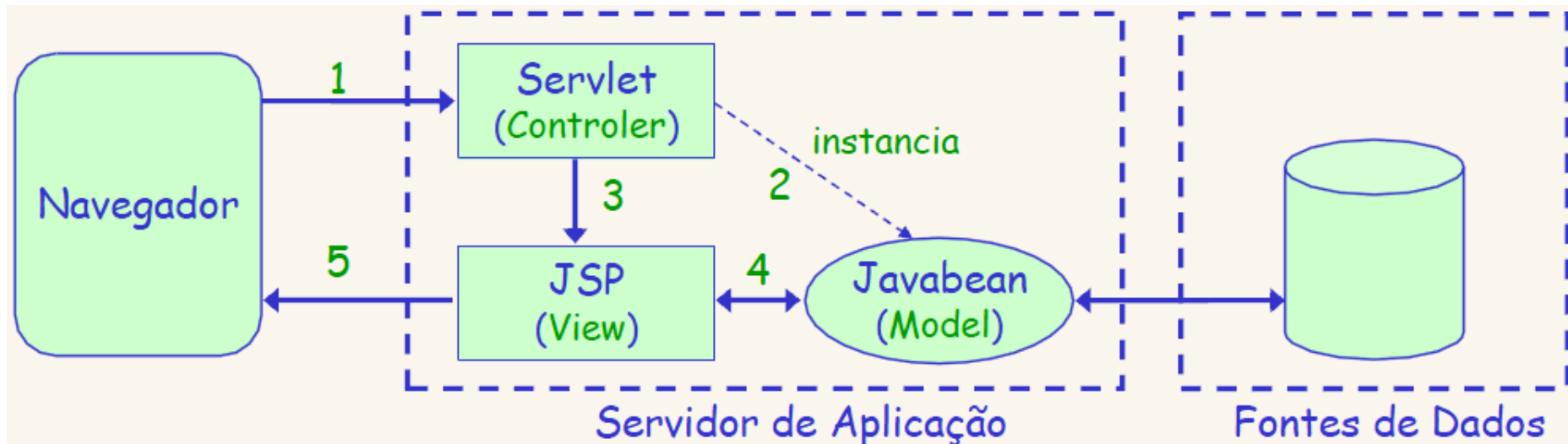
# Interações entre as camadas MVC



# 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:

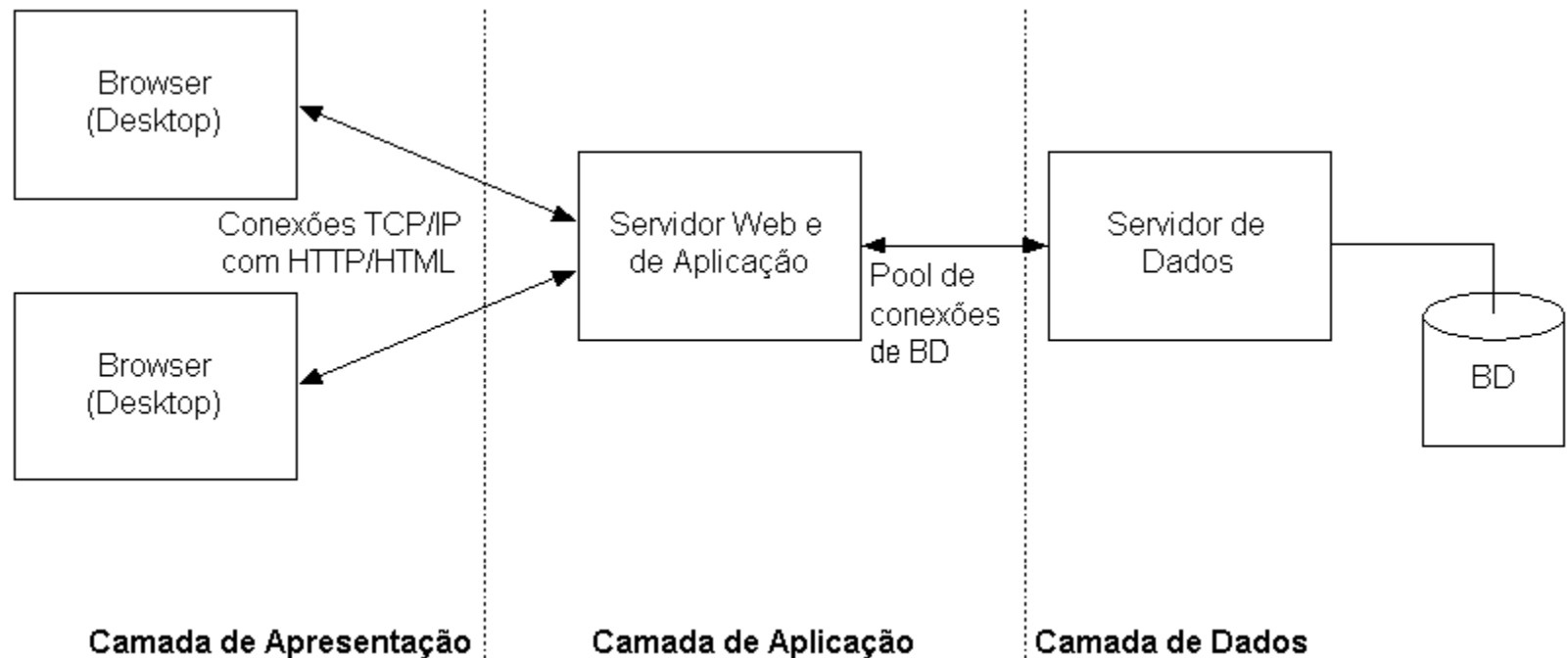
- (A) Todos os dados e a lógica do negócio para processá-los devem ser representados na camada Controller.
- (B) A camada Model pode interagir com a camada View para converter as ações do cliente em ações que são compreendidas e executadas na camada Controller.
- (C) A camada View é a camada responsável 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*)

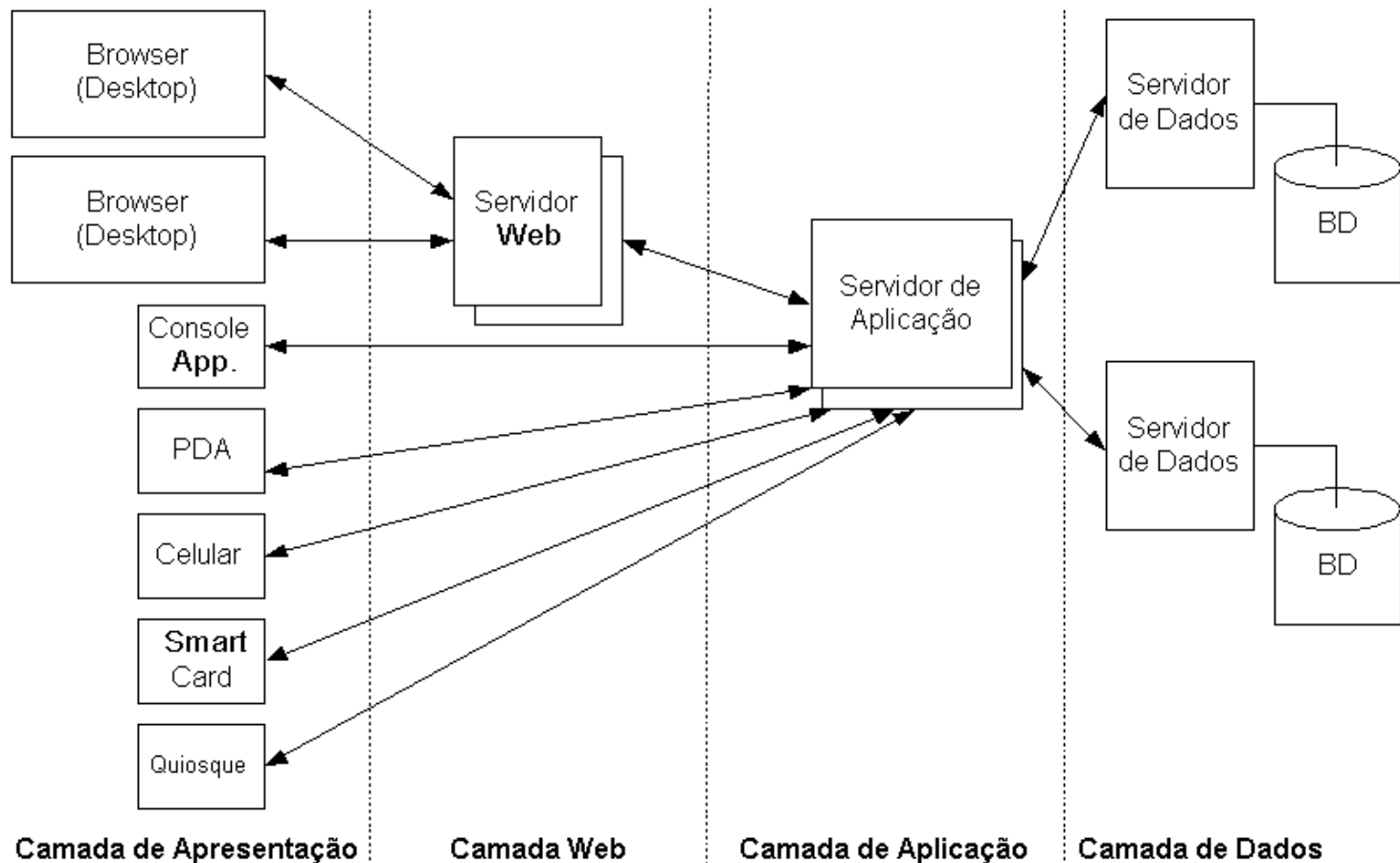
# Arquitetura WEB (três camadas)

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



# Arquitetura WEB (n-camadas)

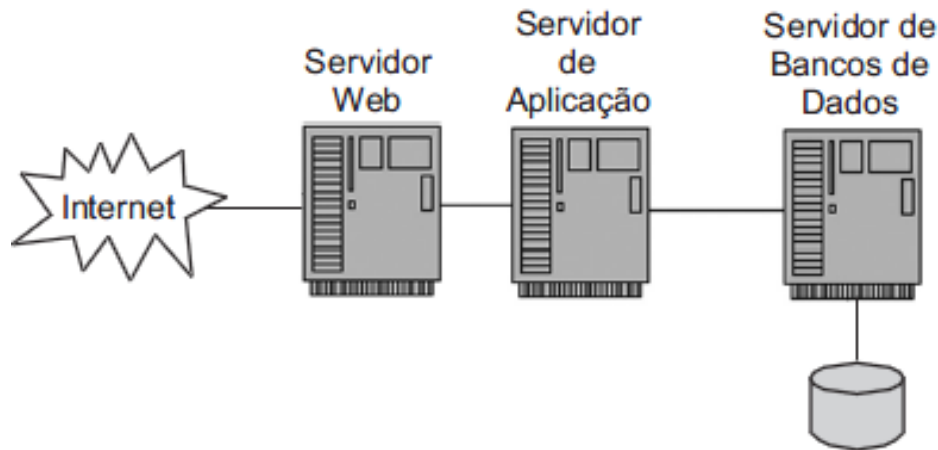
Mas podemos ter mais camadas (flexibilidade)



# 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