



# Arquitetura de Software

Professor Gilmar Luiz de Borba

**2020 - 2**

# Conteúdo (ementa resumida)

- Introdução: o que é arquitetura de software?
- Arquitetura de software: REFERÊNCIA, MDA e SOA
- Principais padrões de projeto GOF e GRASP,
- Padrões de projeto organizacionais,
- Injeção de dependência,
- Inversão de controle,
- Localizador de serviço,
- Padrões para  
Dispositivos móveis.

# Bibliografia Básica

FREEMAN, Eric; FREEMAN Elisabeth. **Use a Cabeça (head first) Padrões de Projeto.** – Rio de Janeiro : Alta Books, 2007.

GAMMA, Erich et al. **Padrões de Projeto: soluções reutilizáveis de software orientado a objetos.** – Porto Alegre: Bookman, 2000.

HORSTMANN, Cay. **Padrões e Projetos Orientados a Objetos.** – Porto Alegre:Bookman, 2007.

LARMAN, Craig. **Utilizando UML e padrões:uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento interativo.** 3<sup>a</sup>. Ed. Porto Alegre:Bookman, 2007.

MARTIM, Robert C. Princípios, padrões e práticas ágeis em C#. Porto Alegre:Bookman, 2011.

PAGE-JONES, Meilir. Projeto Estruturado de Sistemas. São Paulo:McGraw-Hill, 1998.



# Bibliografia Básica

## *GlobalCode*

<http://www.globalcode.com.br/noticias/ApostilaDesignPatterns>

F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns.  
John Wiley & Sons, 1996.

## OASIS - SOA

[https://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=soa-rm](https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm)

## OMG - OBJECT MANAGEMENT GROUP

<http://www.omg.org/>

# Atividades Teóricas

*De 1 a 10: Introdução*

*De 11 a 21: Modelo MVC*

*De 22 a 43: SOA e MDA*

*De 44 a 52: Introdução Patterns e Padrão Singleton*

*De 53 a 57: Padrão Factory*

*De 58 a 64: Padrão Decorator*

*De 65 a 66: Padrão Adapter*

*De 67 a 69: Padrão Strategy*

*De 70 a 72: Padrão Observer*

*De 73 a 81: UML*

*A entrega das questões deverá ser feita no final do semestre letivo.*

# Atividades Práticas

## Implementação em linguagem JAVA

Padrão Singleton

Padrão Factory

Padrão Decorator

Padrão Adapter

Padrão Strategy

Padrão Observer

Uma aplicação distribuída (RMI / Webservice)

## Desenho com UML

Um diagrama de pacotes Padrão Factory

Um diagrama de componente

Um diagrama de Implantação

# Seminário

Baseado em artigos sobre os assuntos desenvolvidos na disciplina.

O seminário será realizado a partir de plenárias com assuntos distribuídos em equipes.

O seminário será realizado pouco antes do final do semestre.

# Considerações Adicionais

Distribuição de pontos

Atividades em Sala de Aula

Atividades no Laboratório

...

# Introdução e Conceitos

# *Engenharia de Software*

“A engenharia de software é uma disciplina de engenharia relacionada com todos os aspectos da produção de software, desde os estágios iniciais de especificação do sistema, até sua manutenção, depois que este entrar em operação.”

(Sommerville, Ian, 2011, 5).

# *Projeto de Arquitetura de Software*

“Sistemas grandes são sempre decompostos em subsistemas que fornecem algum conjunto de serviços relacionados. O processo inicial de projeto, que consiste em identificar esses subsistemas e estabelecer um framework para o controle e a comunicação de subsistemas, é denominado PROJETO DE ARQUITETURA.” (Sommerville, Ian, 2011, 161).

# *Projeto de Arquitetura de Software*

Durante o processo de elaboração do projeto de arquitetura o Arquiteto de Software deve responder a algumas perguntas, como:

# *Projeto de Arquitetura de Software*

- 1 – Existe uma arquitetura genérica?
- 2 – Quais estilos de arquitetura são apropriados?
- 3 – Qual será a abordagem básica do sistema?
- 4 – Como controlar as operações das partes do sistema?
- 5 – Como a arquitetura será avaliada?
- 6 – Como a arquitetura será documentada?

# *Arquitetura*

Trata os elementos que compõem uma estrutura e consequentemente o relacionamento entre esses elementos. A arquitetura procura organizar o espaço e os elementos que compõem esse espaço de forma estética e organizada.

(n/a)

# Arquitetura de Software

Se refere aos componentes de software (pacotes, classes, componentes, serviços etc.), como eles se relacionam, quais são suas propriedades internas e externas e como devem ser documentados.

# Arquitetura (Software)

“A arquitetura de um produto expressa uma divisão desse produto em subsistemas e outros componentes de nível mais baixo, as interfaces entre esses componentes e as interações através das quais eles realizam as funções do produto atendendo aos requisitos [...].”

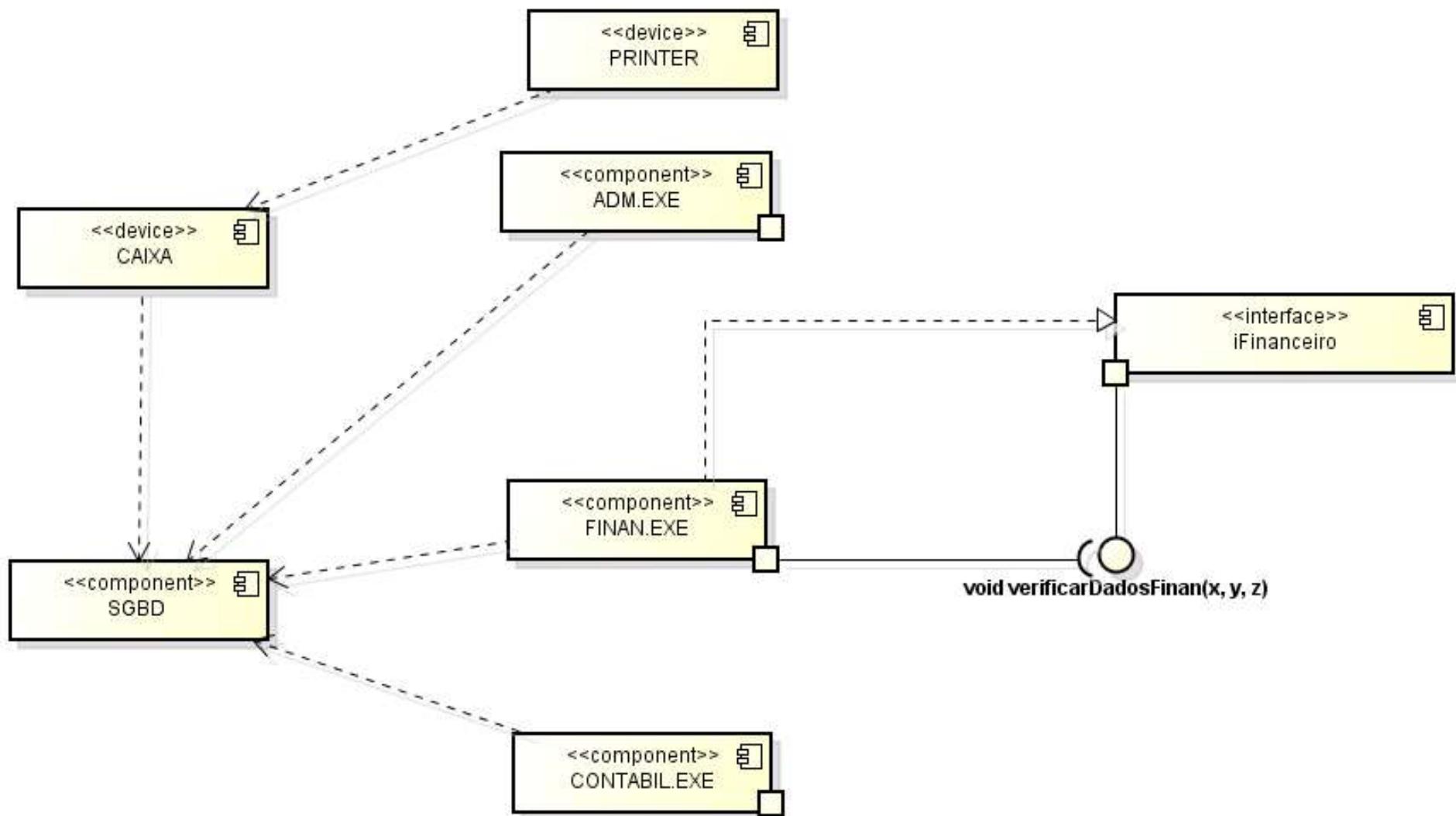
(Sommerville, Ian, 2007, 152).

# Arquitetura de Software

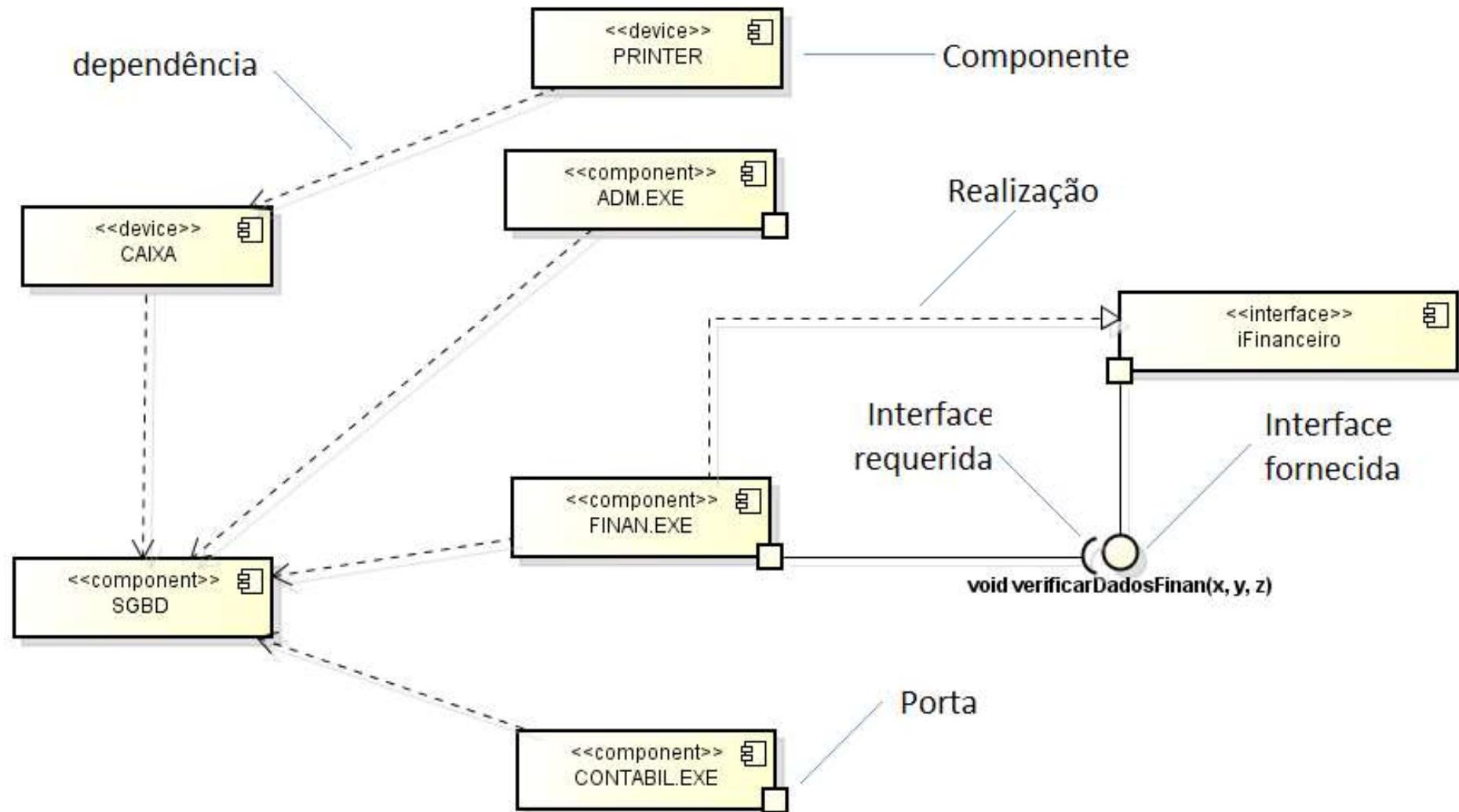
No RUP, a arquitetura de um sistema de software (em um determinado ponto) é a organização ou a estrutura dos componentes significativos do sistema que interagem por meio de interfaces, com elementos constituídos de componentes e interfaces sucessivamente menores.

*No RUP a arquitetura é tratada na disciplina de Modelagem de Negócios*

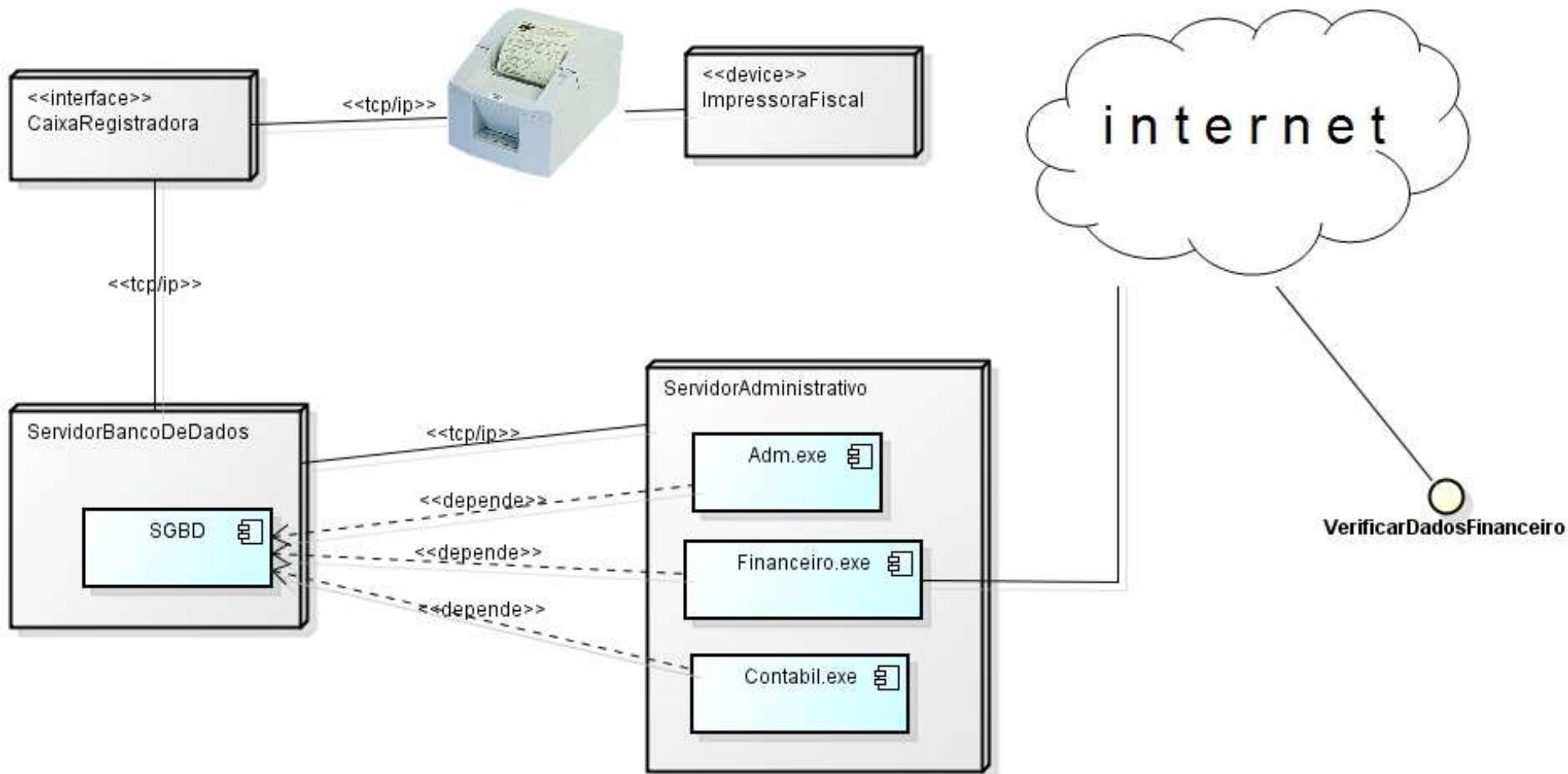
# RUP - Diagrama de Componente



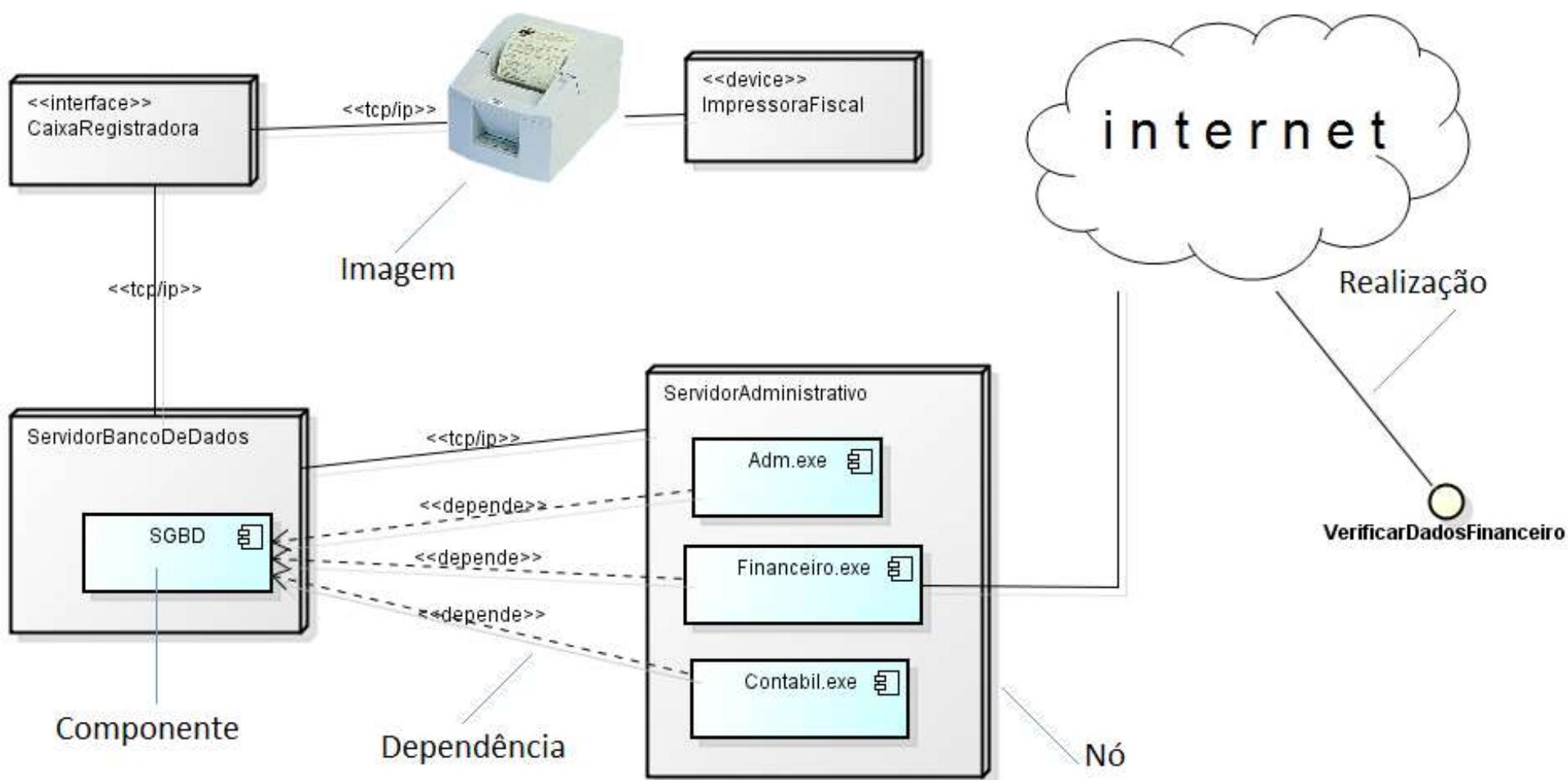
# RUP - Diagrama de Componente



# RUP - Diagrama de Implantação



# RUP - Diagrama de Implantação



# Arquitetura de Software

O artigo Working Group on Architecture da IEEE define a arquitetura como "o conceito de nível mais alto de um sistema em seu ambiente" [IEEE98]

# Arquitetura de Software

Arquitetura de software para um sistema é a estrutura ou estruturas do sistema, que consistem em elementos e suas propriedades visíveis externamente e as relações entre eles.

OASIS - Advanced open standards for the information society  
<https://www.oasis-open.org/committees/soa-rm/faq.php>

# Arquitetura de Software Benefícios

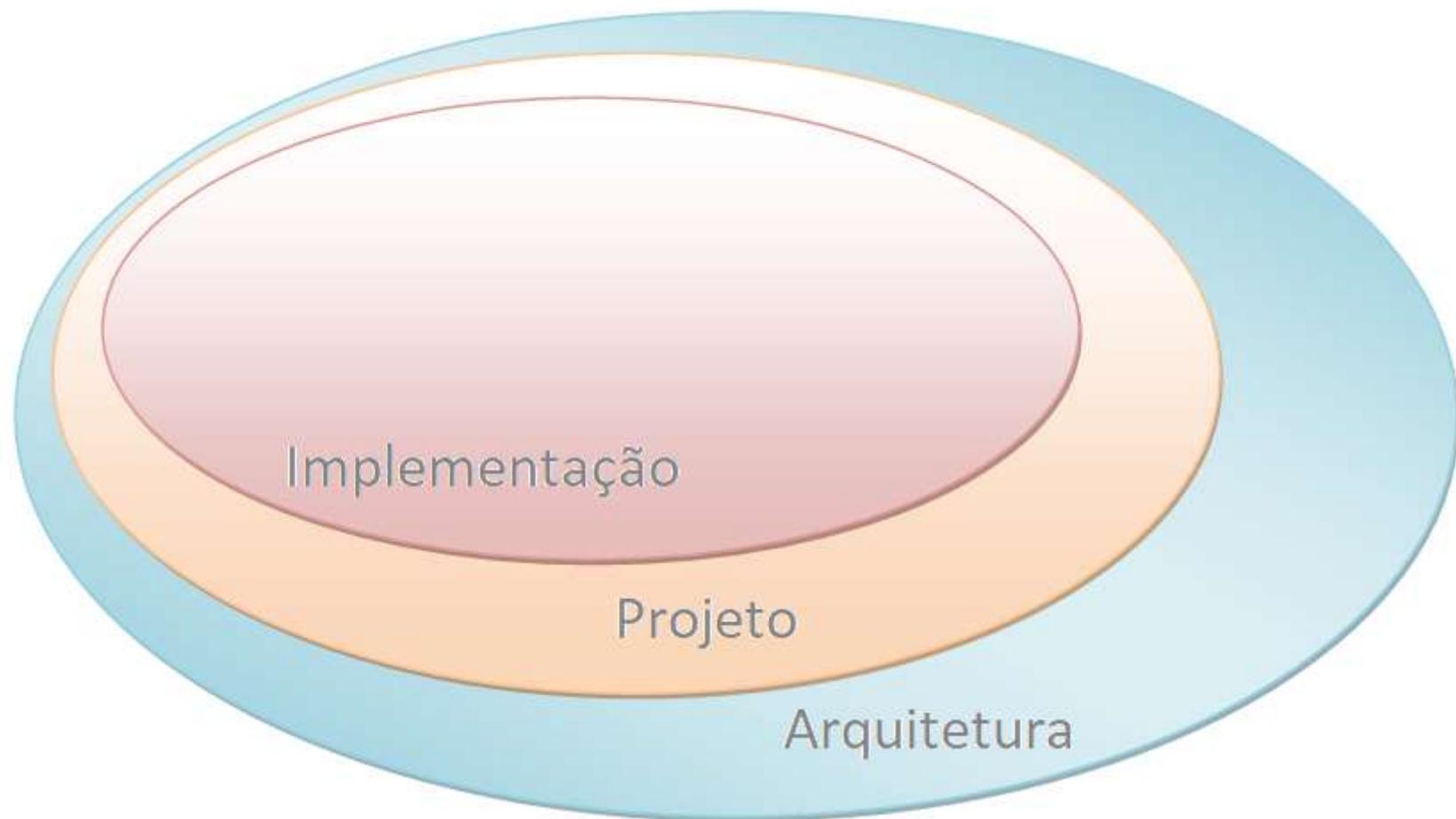
- Reconhecer os componentes do software,
- Facilitar a tomada de decisão sobre alternativas de projeto,
- Analisar e descrever as propriedades de um sistema complexo,
- Facilitar o entendimento geral do sistema,
- Usar notações padronizadas para facilitar a comunicação.

Na Arquitetura de Software são aplicados:

Princípios de projetos  
(qualidade e padronização),

Padrões de projetos  
(qualidade e padronização).

# Arquitetura de Software - Contexto



# Arquitetura de Software - Processo



# O arquiteto de software

"O arquiteto ideal deve ser uma pessoa erudita, um matemático, familiarizado com estudos históricos, um estudioso aplicado de filosofia, conhecedor de música, que não desconheça medicina, detentor de saber jurídico e familiarizado com astronomia e cálculos astronômicos." (RUP).

Marcus Vitruvius Pollio

(born c. 80–70 BC, died after c. 15 BC)

# O arquiteto de software (RUP)

O arquiteto de software deve ter grande conhecimento geral, maturidade, visão ampla e experiência para a identificar problemas, deve se expressar com sensatez, conhecer o domínio do problema e os requisitos, ser um líder nato, ter boa comunicação, ser proativo [...]. São algumas atividades do arquiteto: priorizar os casos de uso, construir e avaliar provas de conceito arquitetural, desenvolver guias de programação, desenvolver guias de design, fazer a análise arquitetural.

# *Modelo de referência da Arquitetura*



Surge a partir de um processo de amadurecimento das solução em função dos requisitos expostos de forma mais genérica ou abstrata.

É baseado em experiências anteriores dos Analistas de Negócio em conjunto com os demais *stakeholders*.

É construído a partir da “modularização” das partes considerando a cooperação entre estas.

# Modelos de referência

“Modelos de referência são mais abstratos e descrevem uma classe maior de sistemas. Eles são um meio de informação para os projetistas sobre a estrutura geral dessa classe de sistema. Os modelos de referência são geralmente derivados de um estudo de domínio de aplicação. Eles representam uma arquitetura idealizada que inclui todas as características que esses sistemas podem incorporar.” (Sommerville, Ian, 2011, 173).

# Modelos de referência

*"Um modelo de referência é um quadro abstrato para entender relacionamentos significativos entre as entidades de algum ambiente e para o desenvolvimento de padrões ou especificações consistentes que suportem esse ambiente."*

OASIS - Advanced open  
standards for the information society.  
<https://www.oasis-open.org/committees/soa-rm/faq.php>

# **Modelos de referência**

"Um exemplo:

## ***Modelo de referência da Internet das Coisas (IoT)***

*Camada de Colaboração e processo*

*Camada de aplicação*

*Camada de abstração de dados*

*Computação de fronteira*

*Camada de conectividade*

*Camada física de dispositivos e Controles*

# Arquitetura de Referência

É uma arquitetura de software que abrange o domínio de aplicação, legislação, normas, considerando as regras de negócio, estilos ou padrões arquiteturais, melhores práticas de desenvolvimento de software.

# Arquitetura de Referência

“[...] conjunto de documentos que apresentam as estruturas e integrações recomendadas de produtos e serviços de TI para formar uma solução. [...] incorpora as melhores práticas aceitas no setor, normalmente sugerindo o método de fornecimento ideal de determinadas tecnologias.”

HP Hewlett Packard

Disponível em: <https://www.hpe.com/br/pt/what-is/reference-architecture.html>

# Arquitetura de Referência

Um exemplo:

## *Arquitetura de referência IoT*

*Camada WEB/Portal, Dashboard e API Management*

*Camada de Processamento e análise de eventos*

*Camada de agregação e mensagens*

*Camada de comunicação Computação de fronteira*

*Camada de conectividade*

*Camada de dispositivos*

# Arquitetura de Software

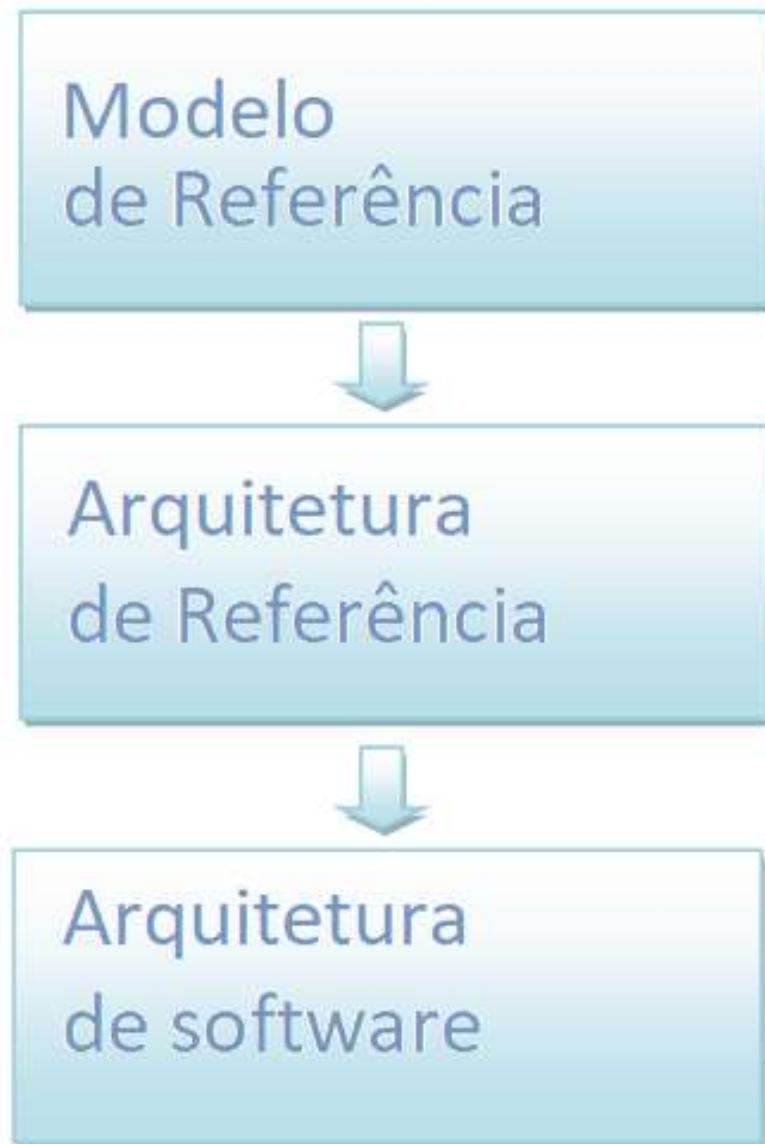
Um exemplo:

## *Arquitetura de Software IoT*

*Apresentação dos componentes e  
relacionamentos entre estes:*

*Access Point, Gateway, dispatcher messages, IoT Server, DB, User Interface, third part software etc.*

# *Arquiteturas - Relacionamento*



## Considerações adicionais

Há várias maneiras de projetar software, historicamente as aplicações *Monolíticas* foram as primeiras, em seguida surgiram as aplicações com a arquitetura do tipo *Cliente-Servidor*, posteriormente surgiram as aplicações em "N" *camadas*, *MVC*, arquitetura de *Objetos Distribuídos* com os padrões como *CORBA*, *SOA* e *MDA*.

# Arquitetura de Software

Nos últimos anos muitos artigos acadêmicos trouxeram informações relacionadas à arquitetura de sistema que envolve tecnologias relacionadas à *Computação Móvel*, *Computação “Pervasiva”* e *Ubíqua*.

Os princípios e *Padrões de Projetos* são técnicas de grande importância usadas para a implementação dessas tecnologias.

# Exercícios

- (01) Conceituar Engenharia de Software
- (02) Diferenciar Projeto de Arquitetura de Software e Arquitetura de Software.
- (03) Cite três diagramas da UML usados de maneira mais específica durante o processo de construção da arquitetura de um sistema.
- (04) Apresentar cinco benefícios da Arquitetura de Software.
- (05) No RUP, existe alguma relação entre a disciplina de Modelagem de Negócio e a Arquitetura de Software? Justifique.
- (06) Nas metodologias Ágeis (XP, SCRUM etc.) como é desenvolvida a Arquitetura de Software? Justifique.
- (07) Cite algumas características do Arquiteto de Software.
- (08) Diferenciar requisitos Funcionais e Não Funcionais.
- (09) Qual é a principal diferença entre os modelos de arquitetura Cliente-Servidor e N camadas?
- (10) Qual é o seu entendimento sobre a computação MÓVEL, PERVASIVA E UBÍQUA?

# Arquiteturas Básicas

*A arquitetura de Sistemas de Informação (ASI) envolve um vários elementos com o objetivo de mapear a organização no que se refere aos elementos envolvidos no processo de desenvolvimento e implantação de um Sistema de Informação.*

- *Recursos de informação*
- *Desenvolvimento de Sistemas*
- *Datawarehouse e datastore*
- *Infra-estrutura tecnológica*
- *Dados e Negócios*

# *Arquitetura ASI*

*A arquitetura de Sistemas de Informação (ASI) envolve um vários elementos com o objetivo de mapear a organização no que se refere aos elementos envolvidos no processo de desenvolvimento e implantação de um Sistema de Informação.*

- *Recursos de informação*
- *Desenvolvimento de Sistemas*
- *Datawarehouse e datastore*
- *Infra-estrutura tecnológica*
- *Dados e Negócios*

# *Arquitetura ASI*

1. *Surge como forma de contribuir para o processo de desenvolvimento e implantação de SI.*
2. *Possui uma visão mais abrangente, incluindo a perspectiva de negócios, a visão organizacional, os próprios SI, a tecnologia disponível e os usuários envolvidos. (Tait et ai, 1998).*
3. *Dinamiza processos de negócios;*
4. *Reduz complexidade dos sistemas;*
5. *Capacita integração na empresa através do compartilhamento de dados e capacita a evolução mais rápida para novas tecnologias.*

# Arquitetura ASI

O modelo de Zachman é estrutura básica para a arquitetura Corporativa. Possibilita, de uma maneira, estruturada, visualizar e definir uma empresa, a partir de uma matriz com colunas e linhas e dentro desse contexto (colunas e linhas) a transformação de uma ideia abstrata em uma instanciação.

	Why	How	What	Who	Where	When
Contextual	Goal List	Process List	Material List	Organisational Unit & Role List	Geographical Locations List	Event List
Conceptual	Goal Relationship	Process Model	Entity Relationship Model	Organisational Unit & Role Relationship Model	Locations Model	Event Model
Logical	Rules Diagram	Process Diagram	Data Model Diagram	Role Relationship Diagram	Locations Diagram	Event Diagram
Physical	Rules Specification	Process Function Specification	Data Entity Specification	Role Specification	Location Specification	Event Specification
Detailed	Rules Details	Process Details	Data Details	Role Details	Location Details	Event Details

# Arquitetura ASI

## Modelo Gifford

Segundo Tait et ai, (1998), O modelo Gifford (1992) propõe uma arquitetura que aplica hardware com foco na atividade de negócio. Essa arquitetura se compõe de três dimensões:

- Os componentes;
- Os ciclo de desenvolvimento;
- A correlação entre as várias plataformas;

Dentro dos componentes devem ser modelados: os dados (Diagrama Entidade-Relacionamento - DER) e funções. associados ao hardware necessário. A correlação entre as várias plataformas envolve o hardware, o sistema operacional, os protocolos de comunicação de dados e. as ferramentas de aplicações e de bancos de dados.

# *Arquitetura ASI*

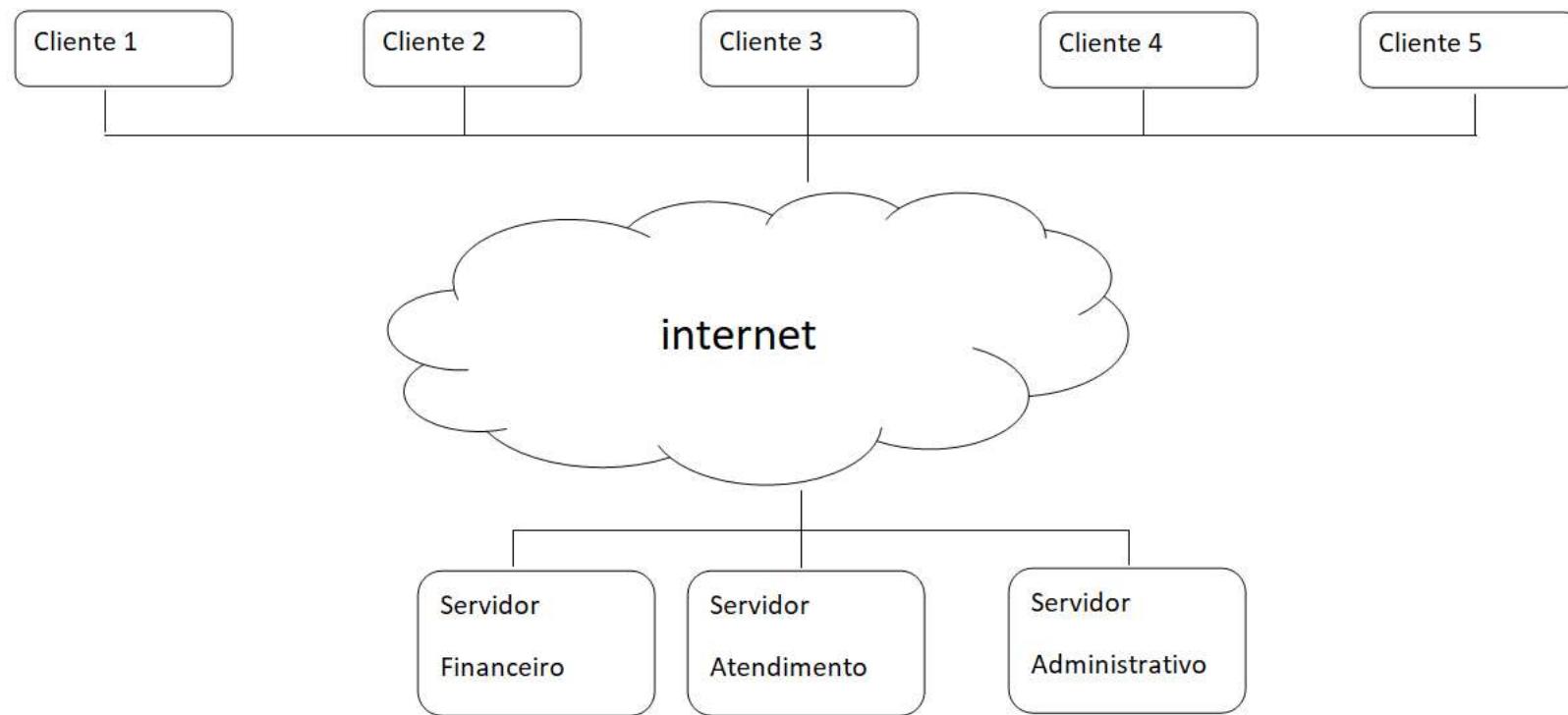
## *Considerações Finais*

*A arquitetura ASI surgiu de forma a contribuir para o processo de desenvolvimento e implantação. A ASI ao longo da sua evolução dos Sistemas de Informação assumindo uma visão abrangente , que inclui: negócios, visão organizacional e evolução das tecnologias de Informação e Comunicação.*

*A arquitetura ASI aborda portanto processo de desenvolvimento, implantação dos Sistemas de informação, relacionando hardware, software e recursos de redes.*

# Arquitetura (modelo) cliente-servidor

*“O modelo de arquitetura cliente-servidor é um modelo em que o sistema é organizado como um conjunto de serviços e servidores e clientes associados que acessam e usam os serviços.” (Sommerville, 2011, 166)*



# *Arquitetura (modelo) cliente-servidor*

- 1 – Um conjunto de servidores oferecem serviços para outros subsistemas. Por exemplo servidor de impressão.*
- 2 – Um conjunto de clientes solicitam serviços oferecidos pelos servidores.*
- 3 – Uma rede permite acessar esses serviços.*

*Os clientes precisam saber os nomes dos servidores disponíveis e os serviços que eles fornecem. No entanto os servidores não precisam saber a identidade dos clientes ou quantos clientes existem.  
(Sommerville, 2011, 166)*

# Arquitetura (modelo) cliente-servidor

*Modelo Cliente-Servidor de 2 camadas: duas abordagens:*

Modelo cliente-magro



Modelo cliente-gordo



# *Arquitetura (modelo) em camadas*

*“A criação de camadas é uma das técnicas mais comuns que os projetistas de software usam para quebrar em pedaços um sistema complexo de software [...].*

*(Fowler, 2006, 37).*

# *Arquitetura (modelo) em camadas*

“O modelo em camadas de uma arquitetura [...] organiza um sistema em camadas, cada uma das quais fornecendo um conjunto de serviços. Cada camada pode ser imaginada como uma máquina abstrata cuja linguagem de máquina é definida pelos serviços oferecidos pela camada.” (Sommerville, 2011, 167)

# *Arquitetura (modelo) em camadas*

**Camada de sistema de gerenciamento de configuração**

**Camada de sistema de gerenciamento de objetos**

**Camada de sistema de banco de dados**

**Camada de sistema Operacional**

# *Arquitetura (modelo) em camadas*

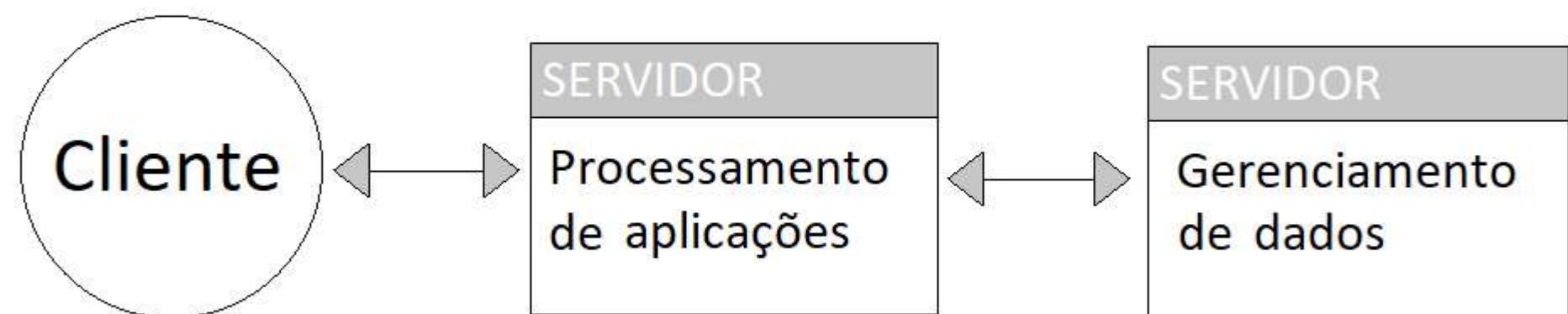
Exemplos de modelos em camadas:

- 1 – Modelo de referência OSI de protocolos de rede.
- 2 – Modelo de 3 camadas para ambiente de apoio à programação ADA.
- 3 - *Model-view-controller* (MVC)

# *Arquitetura (modelo) em camadas*

*Arquitetura Cliente-Servidor de 3 camadas:*

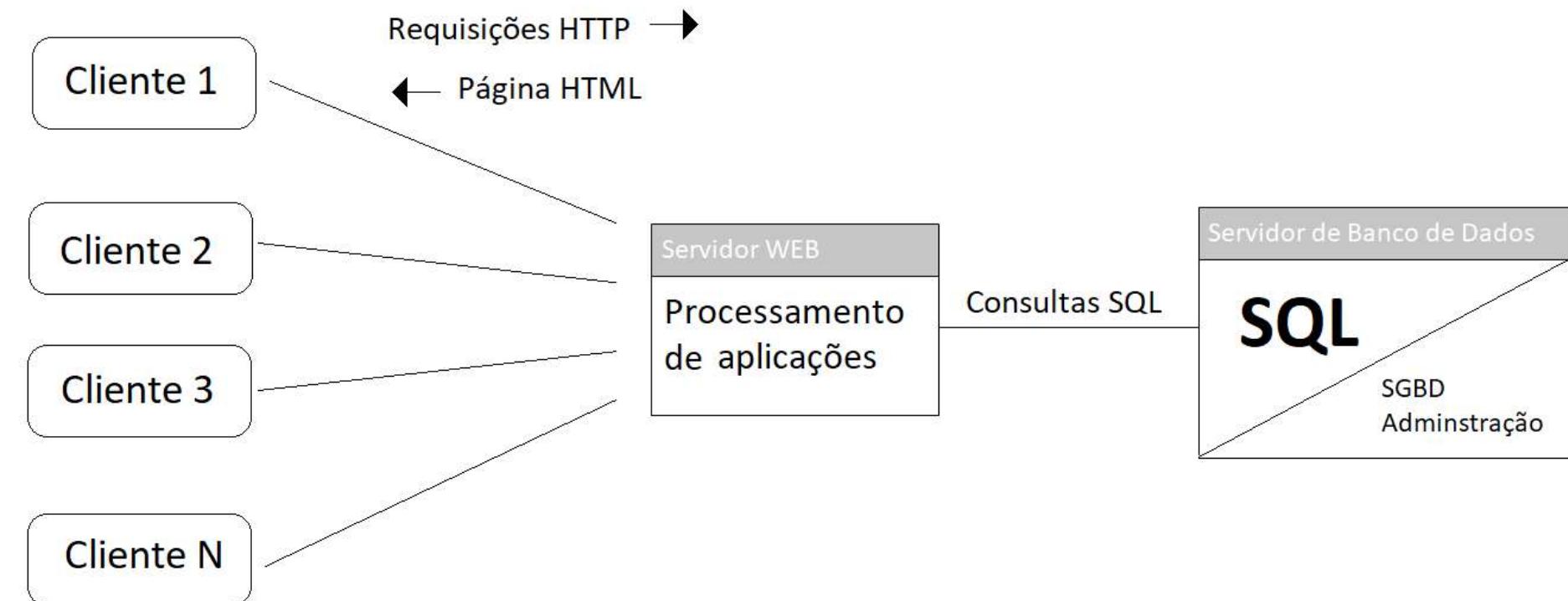
## Apresentação



c:\prof\_gilmar\_borba\java\bd\imagens\

# Arquitetura (modelo) em camadas

Arquitetura Cliente-Servidor de 3 camadas: exemplo:





# Arquitetura

## *MVC*

## **MVC**(Padrão Modelo-Visão-Controle)

Vimos que, de um modo geral, a arquitetura de software procura definir os elementos de software e como eles interagem entre si. A arquitetura de software possui vários elementos, tais como: elementos oriundos do domínio do problema, interfaces, elementos de comunicação, elementos de persistência, entre outros. Devido à complexidade das aplicações, algumas arquiteturas foram pré-definidas, algumas delas estão embutidas em frameworks.

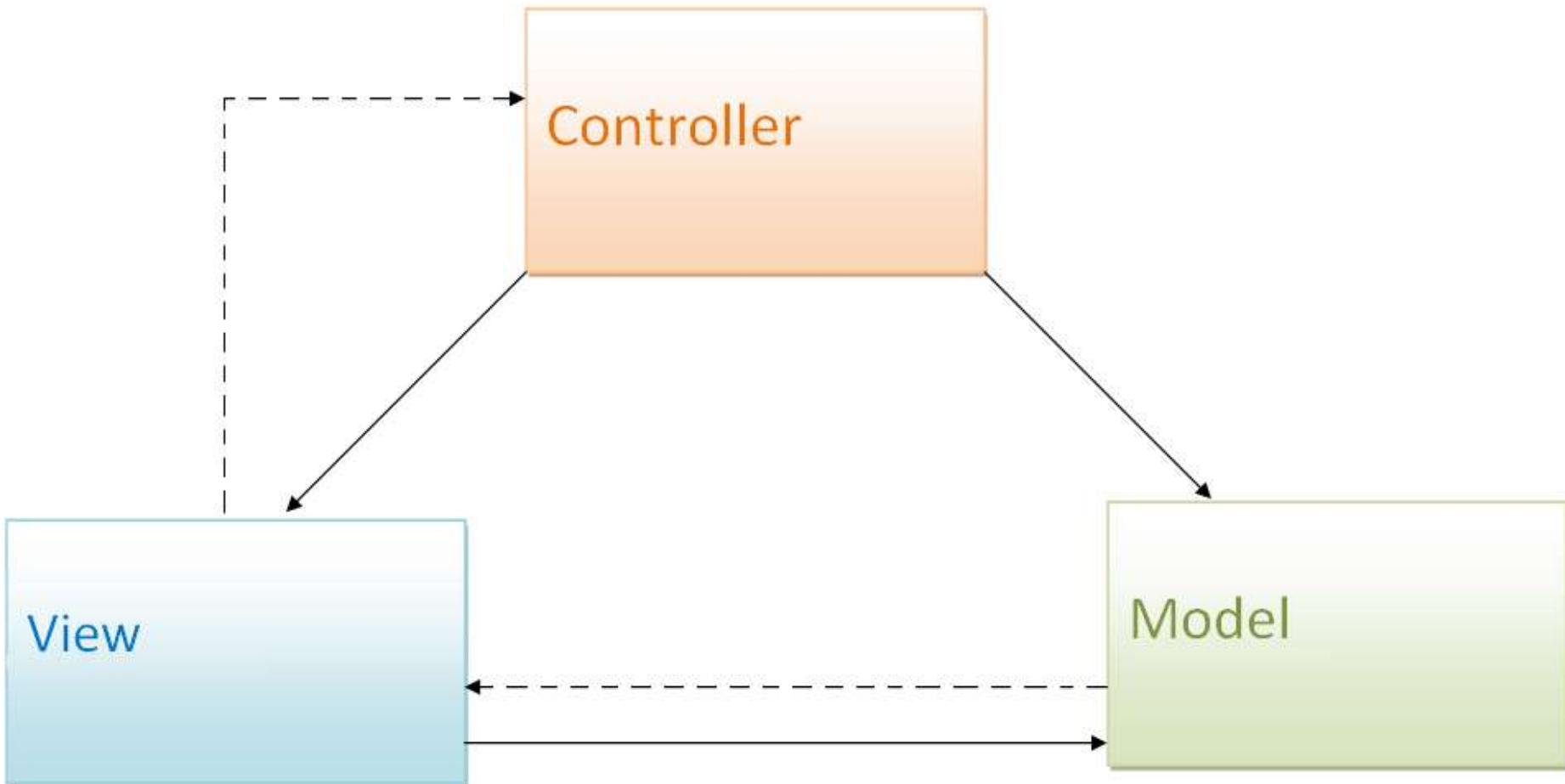
# **MVC** (*Padrão Modelo-Visão-Controle*)

Um dos modelos arquiteturais mais conhecidos é o "padrão" arquitetural MVC (*Model-View-Controller*).

O Padrão MVC possibilita a partição do projeto em camadas bem definidas, cada uma dessas camadas possui alta coesão, ou seja, executa somente o que foi definido para ela.

# **MVC (Padrão Modelo-Visão-Controle)**

## Objetos View, Controller e Model



# **MVC (Padrão Modelo-Visão-Controle)**

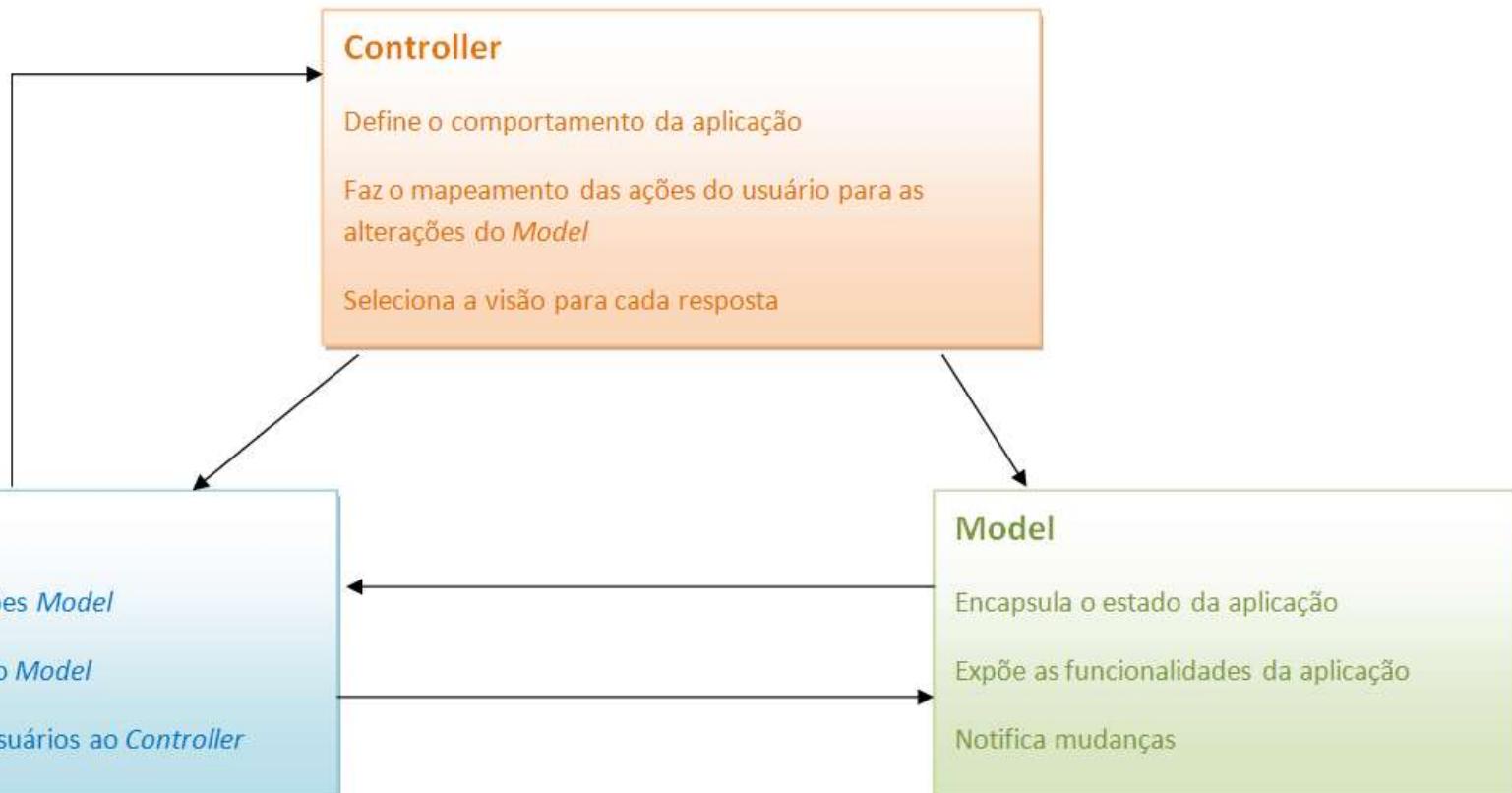
## Benefícios

- 1 - Separar as regras de negócios e interface com o usuário.
- 2 - Reusar a interface para outras aplicações.
- 2 - Fácil comunicação entre as camadas através do Controlador.
- 3 - Facilitar o reuso das classes.
- 4 - Facilitar a manutenção do sistema.
- 5 - Usar recursos básicos da OO como alta coesão e baixo acoplamento.
- 6 - Usado em ambientes diferentes como: *desktop, web e mobile*.

# **MVC (Padrão Modelo-Visão-Controle)**

- 1 - **Controller**: interpreta as entradas de dados, fazendo o mapeamento das ações do usuário em comandos que são enviados para o modelo (*Model*) ou camada de visualização (*View*).
- 2 - **Model**: responde as requisições altera o estado de acordo com essas requisições. O modelo é o centro da arquitetura, pois é ele, que traz a modelagem do problema a ser resolvido.
- 3 - **View**: apresenta as informações para o usuário, a partir do recebimentos das instruções do *Controller*. Também retorna comunicação com o modelo e com o controlador para reportar o seu estado.

# MVC (Padrão Modelo-Visão-Controle)



c:\prof\_gilmar\_borba\java\bd\imagens\

# **MVC** (*Padrão Modelo-Visão-Controle*)

Exemplos de implementação MVC (Java):

- 1 – JSF ,
- 2 – *Struts*,
- 3 – *Spring MVC*,
- 4 – *Play Framework*,
- 5 – *Tapestry*.

# **MVC** (*Padrão Modelo-Visão-Controle*)

“MVC é um paradigma para delimitar seu código em segmentos funcionais para que seu cérebro não exploda. Para obter reusabilidade, mantenha esses limites claros, **Modelo** de um lado, **Visualização** de outro e **Controlador** no meio.

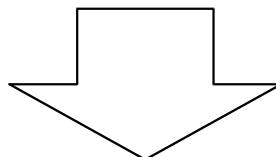
(Freeman, Head first design patterns, 2005, 2007, 420).



# Arquitetura de *Sistemas Distribuídos*

# Arquitetura cliente-servidor

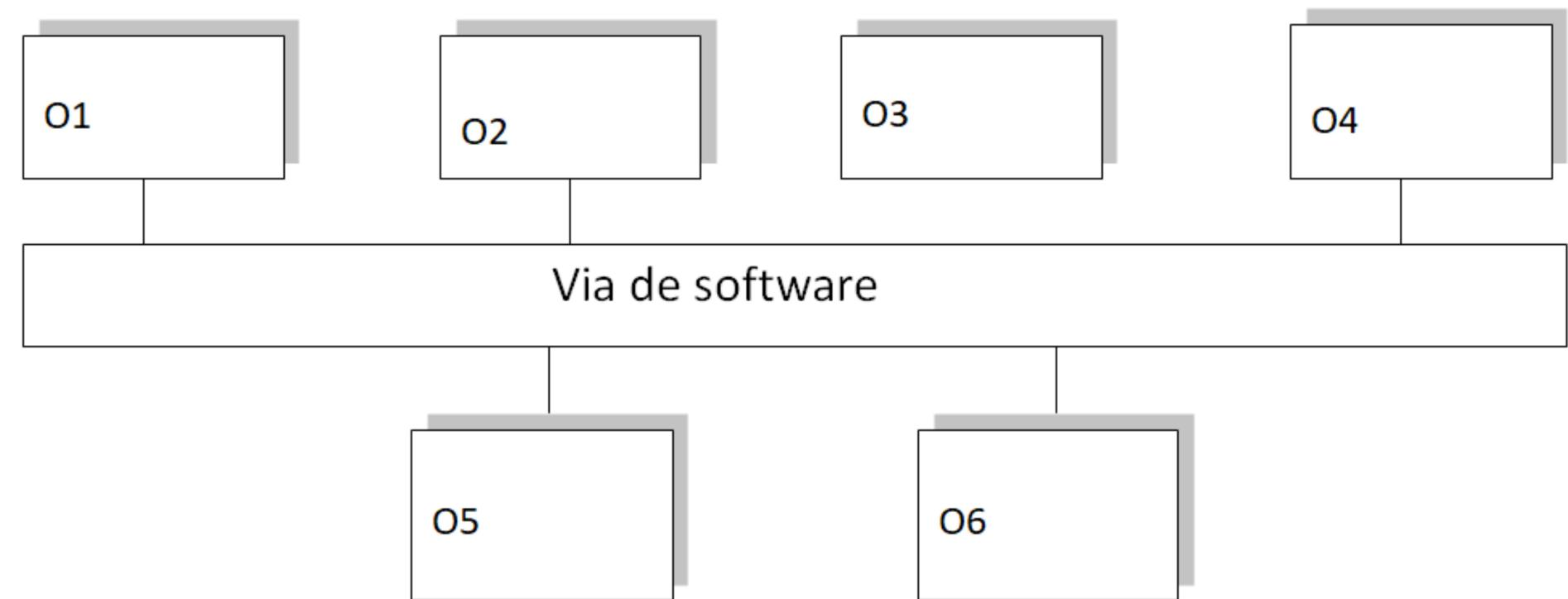
No modelo Cliente-Servidor [...] os clientes e os servidores são diferentes, os clientes recebem serviços dos servidores e não de outros clientes, os servidores podem atuar como clientes quanto ao recebimento de serviços de outros servidores, mas eles não solicitam serviços de clientes [...]” (*Sommerville, 2011, 183*).



“Uma abordagem mais genérica para o projeto de sistemas distribuídos é remover a distinção entre cliente e servidor e projetar a arquitetura de sistemas como uma arquitetura de objetos distribuídos. Em uma arquitetura de objetos distribuídos os componentes [...] são objetos que fornecem uma interface para um conjunto de serviços fornecidos.” (*Sommerville, 2011, 183*).

# Arquitetura cliente-servidor

Os objetos podem ser distribuídos entre uma série de computadores na rede através de um *middleware*, denominado “requisitor” de objetos, seu papel é fornecer uma interface transparente contínua entre os objetos.

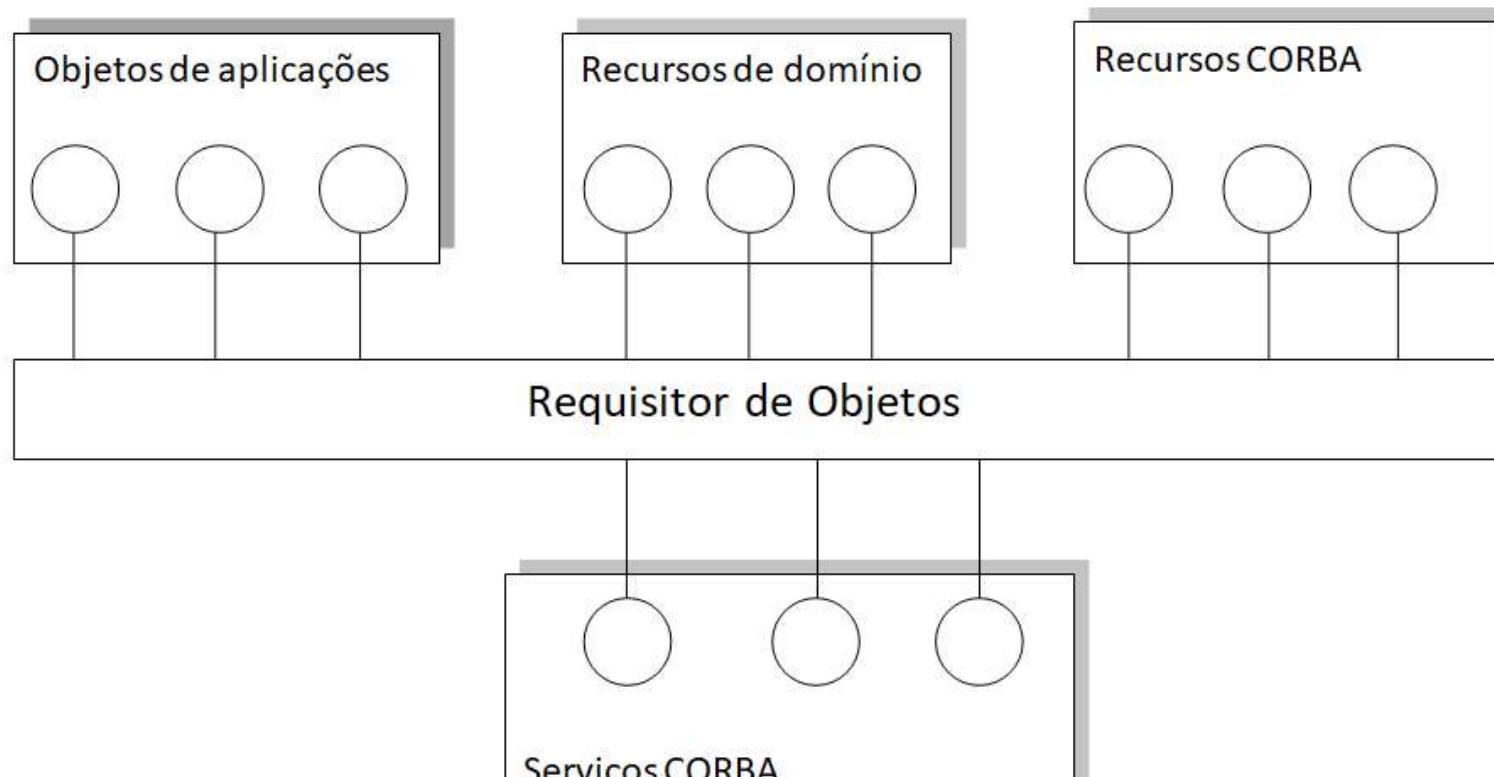


# Arquitetura cliente-servidor

O *middleware* fornece funcionalidade permitindo aos objetos entre os diferentes computadores trocando informações de dados e de controle.

Exemplos: **EJB**, **ACTIVE X**, **CORBA** e **COM**.

<http://www.omg.org/>, <http://www.corba.org/>



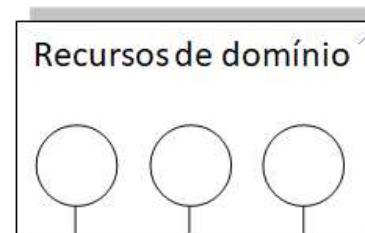
# Arquitetura cliente-servidor

## Detalhamento CORBA

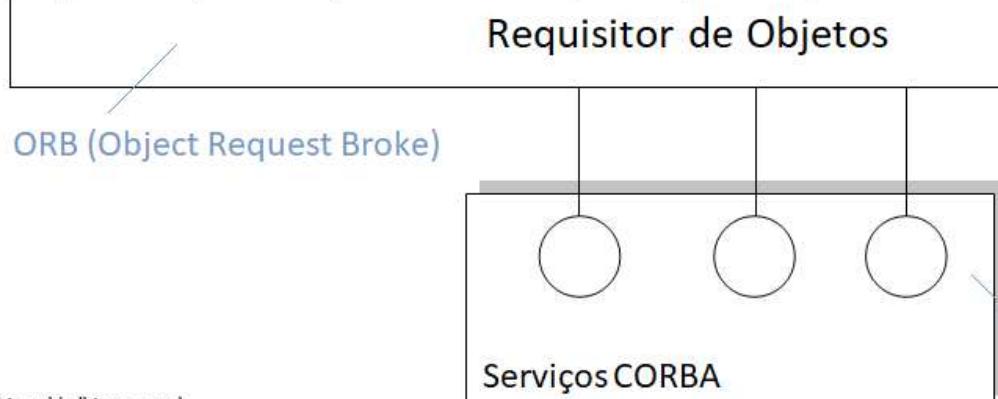
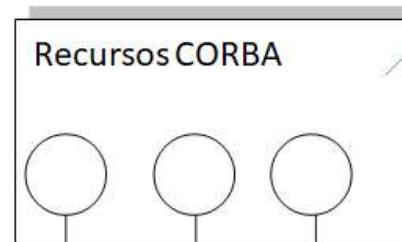
Um objeto CORBA encapsula estado para um objeto da aplicação (usa uma linguagem específica).



Componentes verticais da fronteira da aplicação



Componentes horizontais (de propósito geral)



ORB localiza o objeto que fornece o serviço, prepara e envia o serviço solicitado

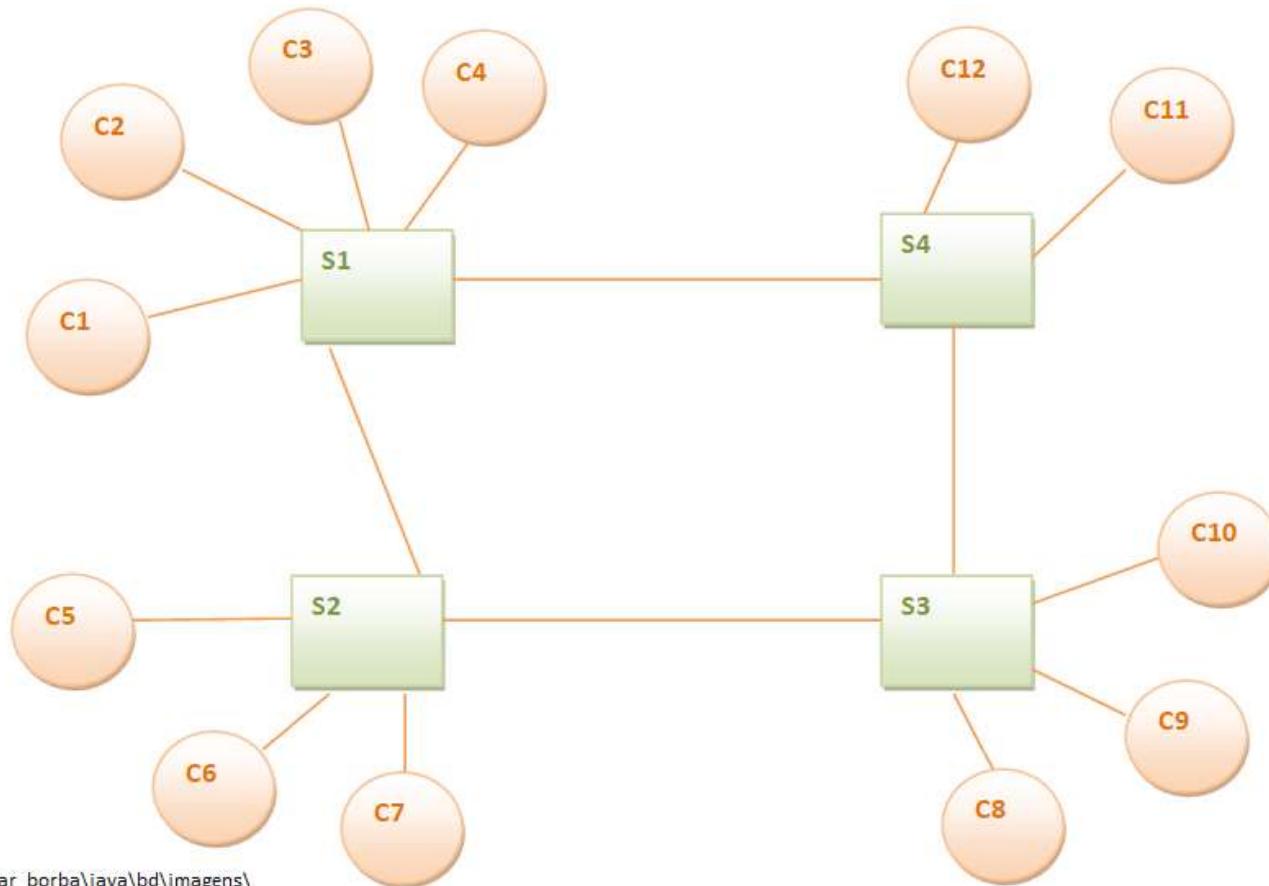
Conjunto de serviços que podem ser solicitados pela aplicações distribuídas.

# *Arquitetura cliente-servidor*

Vantagens do modelo de objetos distribuídos:

- 1 – Os objetos que fornecem serviços podem ser executados em qualquer nó da rede. A distinção entre cliente magro ou gordo passa ser irrelevante.*
- 2 – É uma arquitetura aberta, pois permite que novos recursos sejam adicionados quando necessário.*
- 3 – O sistema é flexível e “escalonável”. Novos objetos podem ser adicionados quando a carga do sistema aumentar sem interromper outros objetos.*
- 4 – É possível reconfigurar o sistema dinamicamente com objetos que migram através da rede.*

# Arquitetura distribuída cliente-servidor



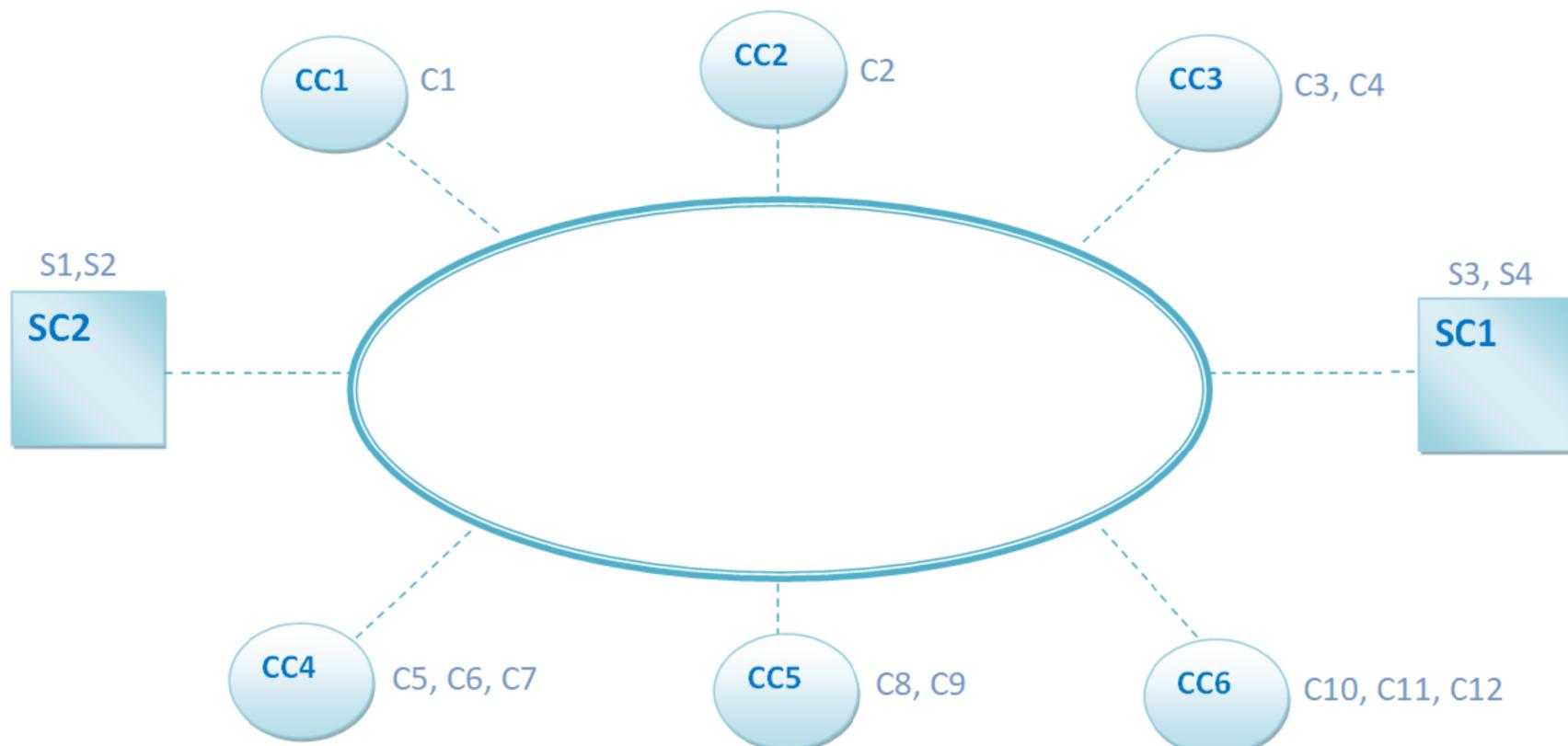
**MODELO LÓGICO DE UMA ARQUITETURA CLIENTE SERVIDOR DISTRIBUÍDA.**

S1, S2, S3, S4 = Processo servidor.

C1, C2, C3, C4 = Processo cliente

(Sommerville, 2011, 180)

# Arquitetura distribuída cliente-servidor



c:\prof\_gilmar\_borba\java\bd\imagens\

## **MODELO FÍSICO DE UMA ARQUITETURA CLIENTE-SERVIDOR DISTRIBUÍDA**

*SC1, SC2, SC3, SC4 = Computador servidor*

*CC1, CC2 ... CC6 = Computadores clientes*

# Exercícios



- (11) Dentro do contexto do modelo cliente-servidor, qual é a diferença entre os modelos: *thin client - fat server* e *fat client - thin server*.
- (12) Forneça um exemplo de sistema modelo/arquitetura em camadas.
- (13) Qual é a principal diferença entre o modelo tradicional cliente-servidor de duas camadas e o modelo de 3 camadas?
- (14) O que significa dizer que no modelo MVC as camadas possuem alta coesão?
- (15) No modelo MVC, qual é a função do controlador (Controller)?
- (16) Tecnicamente, o que ocorre no modelo MVC quando o cliente aciona o botão em uma interface cliente?

# Exercícios



- (17) Tecnicamente, o que ocorre no modelo MVC quando o "model" recebe a mensagem do controlador?
- (18) Citar 3 benefícios do modelo MVC
- (19) No modelo MVC qual parte é considerada o centro desse modelo? Justifique.
- (20) Qual é a diferença entre uma arquitetura cliente-servidor tradicional e a arquitetura cliente-servidor distribuída?
- (21) Destacar 3 vantagens do modelo de objetos distribuídos.

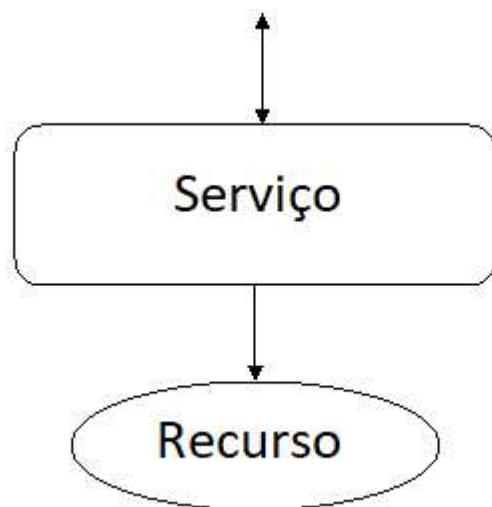


# SOA

# **SOA (Service-Oriented Architecture)**

É uma arquitetura independente de fornecedores onde os serviços são fornecidos aos componentes a partir de um protocolo de comunicação em uma rede.

Entende-se como **serviço** uma funcionalidades do sistema que pode ser acessada remotamente e atualizada de forma independente.



**Recurso** pode ser entendido como qualquer tipo de aplicativo ou armazenamento de dados usado em uma organização.

# **SOA (Service-Oriented Architecture)**

É um paradigma para organização e utilização de recursos distribuídos que estão sob o controle de domínios proprietários.

OASIS

<https://www.oasis-open.org/committees/soa-rm/faq.php>

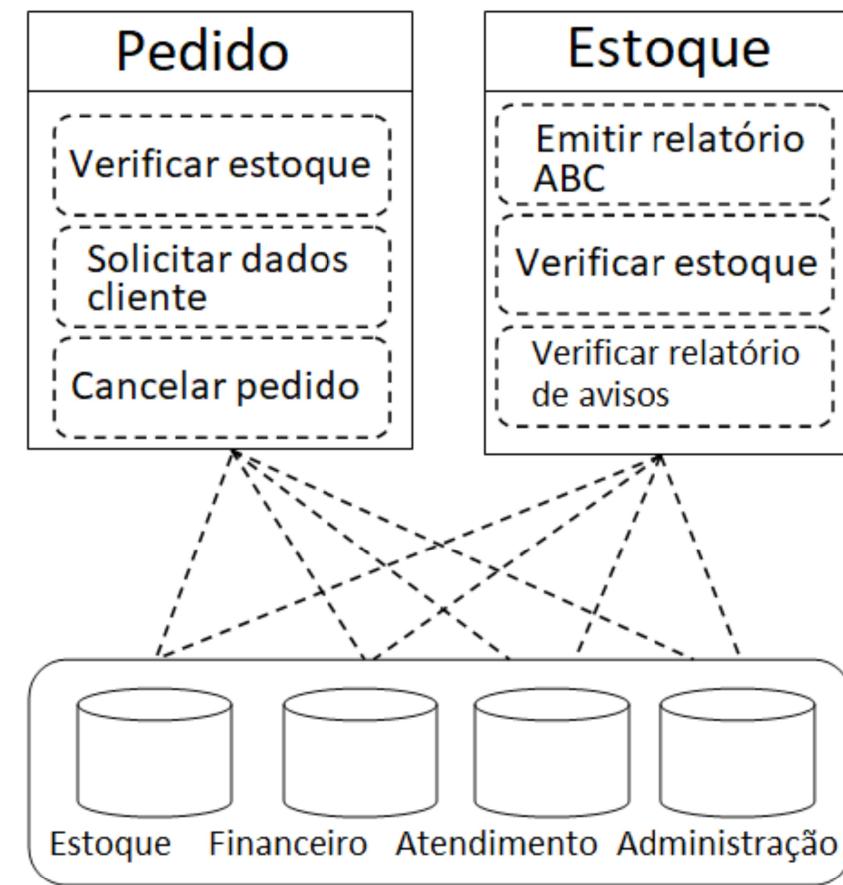
## **Características**

- 1 - Independente de linguagem/tecnologia
- 2 - Focos nos processos da empresa
- 3 – Integração (dos processos e infraestrutura)
- 5 – Modelo que auxilia na produtividade
- 6 – Usa o conceito de baixo acoplamento.

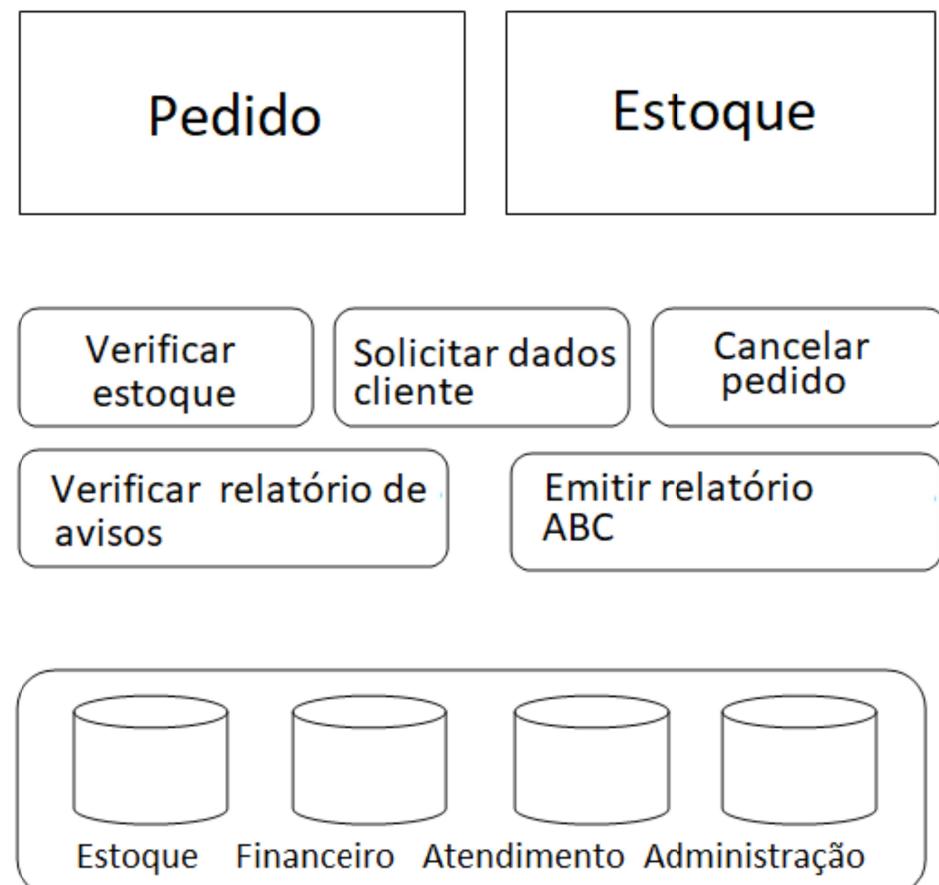
# **SOA (Service-Oriented Architecture)**

Comparativo: modelo tradicional X SOA

## APLICAÇÃO TRADICIONAL

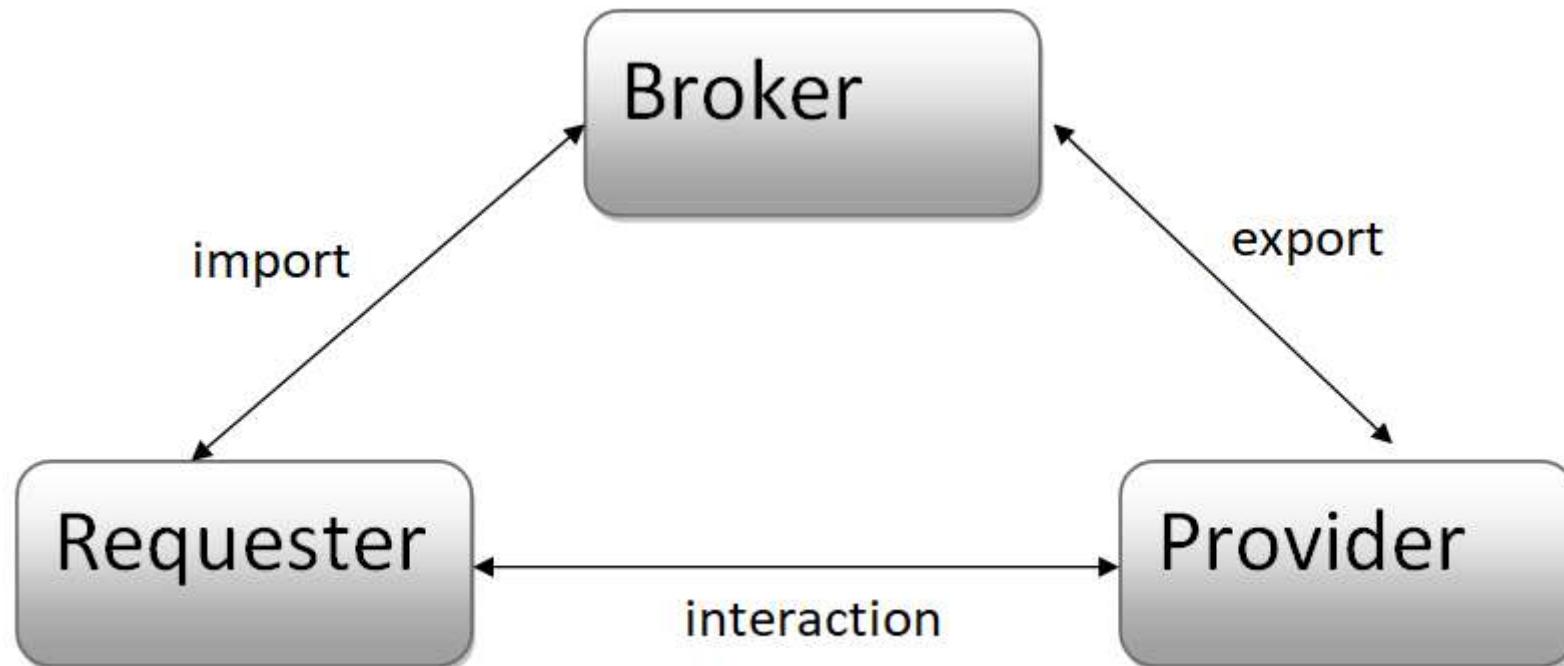


## APLICAÇÃO SOA



# **SOA (Service-Oriented Architecture)**

## Funcionamento – Primeiros Modelos/Arquitetura

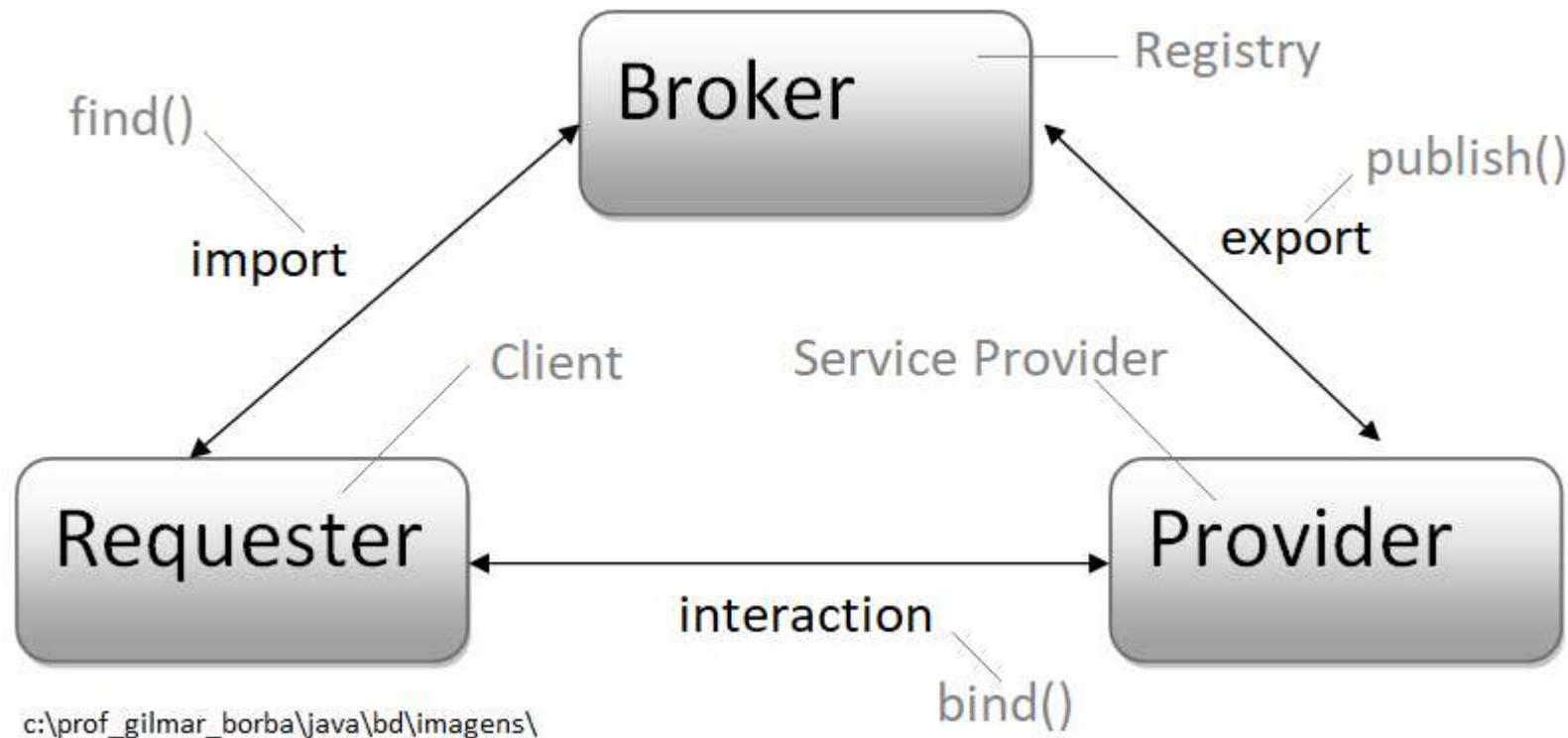


c:\prof\_gilmar\_borba\java\bd\imagens\

Origem do modelo triangular

# **SOA (Service-Oriented Architecture)**

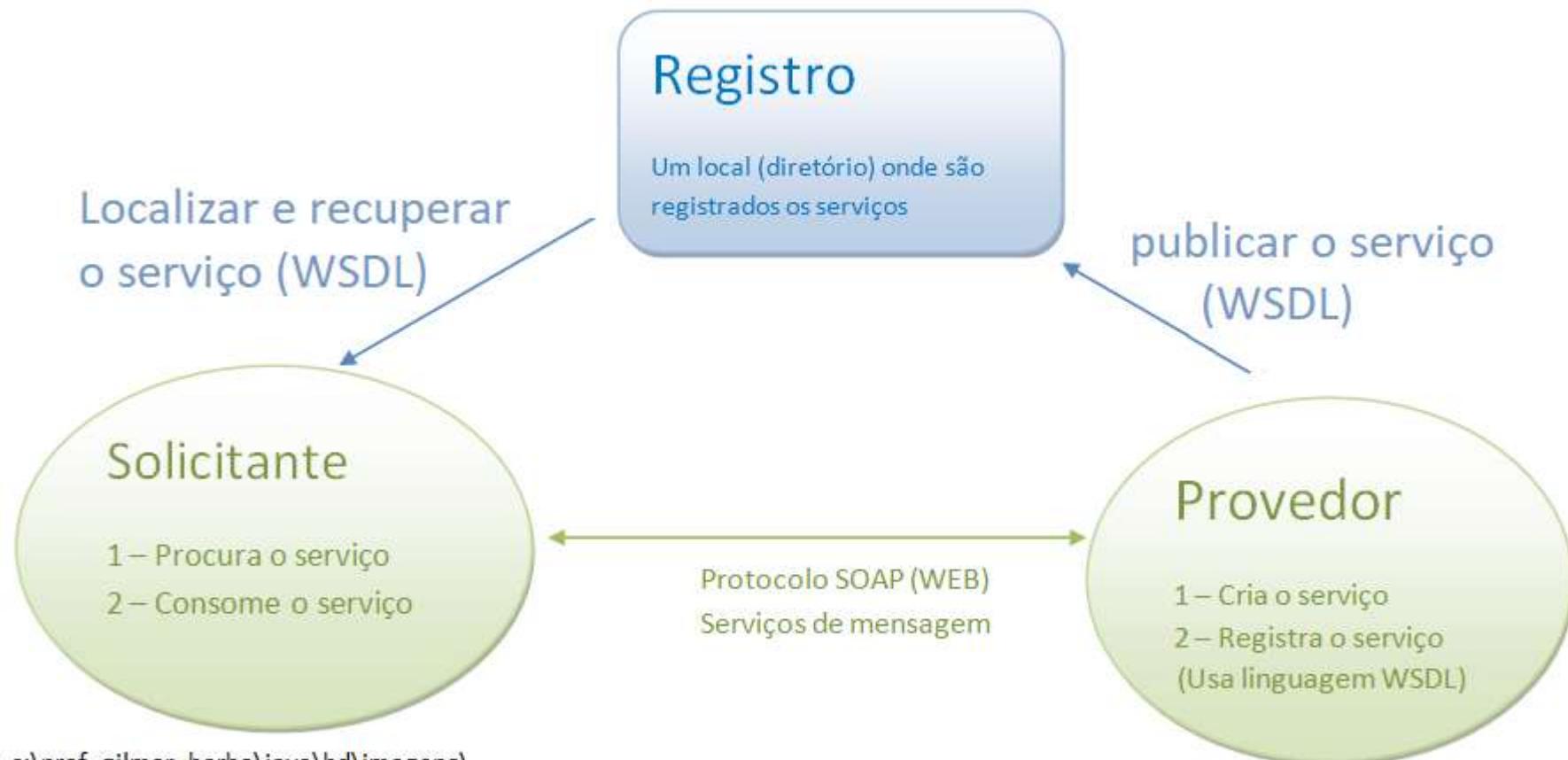
## Funcionamento – Primeiros Modelos/Arquitetura



Origem do modelo triangular

# **SOA (Service-Oriented Architecture)**

## Funcionamento – Modelo/Arquitetura Triangular



c:\prof\_gilmar\_borba\java\bd\imagens\

O modelo triangular é baseado em WEB Services.

# **SOA (Service-Oriented Architecture)**

## ELEMENTOS DO SOA

1 - Aplicação frontend

2 – Serviços

3 - Repositório de serviços

4 - Barramento de serviços



## Benefícios

- 1 – Maior agilidade pois responde rápido às mudanças
- 2 – Possibilita o reuso (de classes, métodos, infraestrutura)
- 3 - É independente de uma tecnologia específica.
- 4 – Maior agilidade no processo de desenvolvimento.
- 5 – Redução de riscos no projeto e implementação.
- 6 – Pode ser usado para dar sobrevida aos legados.

# **SOA (Service-Oriented Architecture)**

A comunidade responsável pela especificação do SOA é a OASIS (Organization for the Advancement of Structured Information Standards) que especifica a Arquitetura de Referência, os melhores princípios, padrões e práticas relacionados aos serviços de computação distribuída.

<https://www.oasis-open.org/committees/soa-rm/faq.php>

# **SOA (Service-Oriented Architecture)**

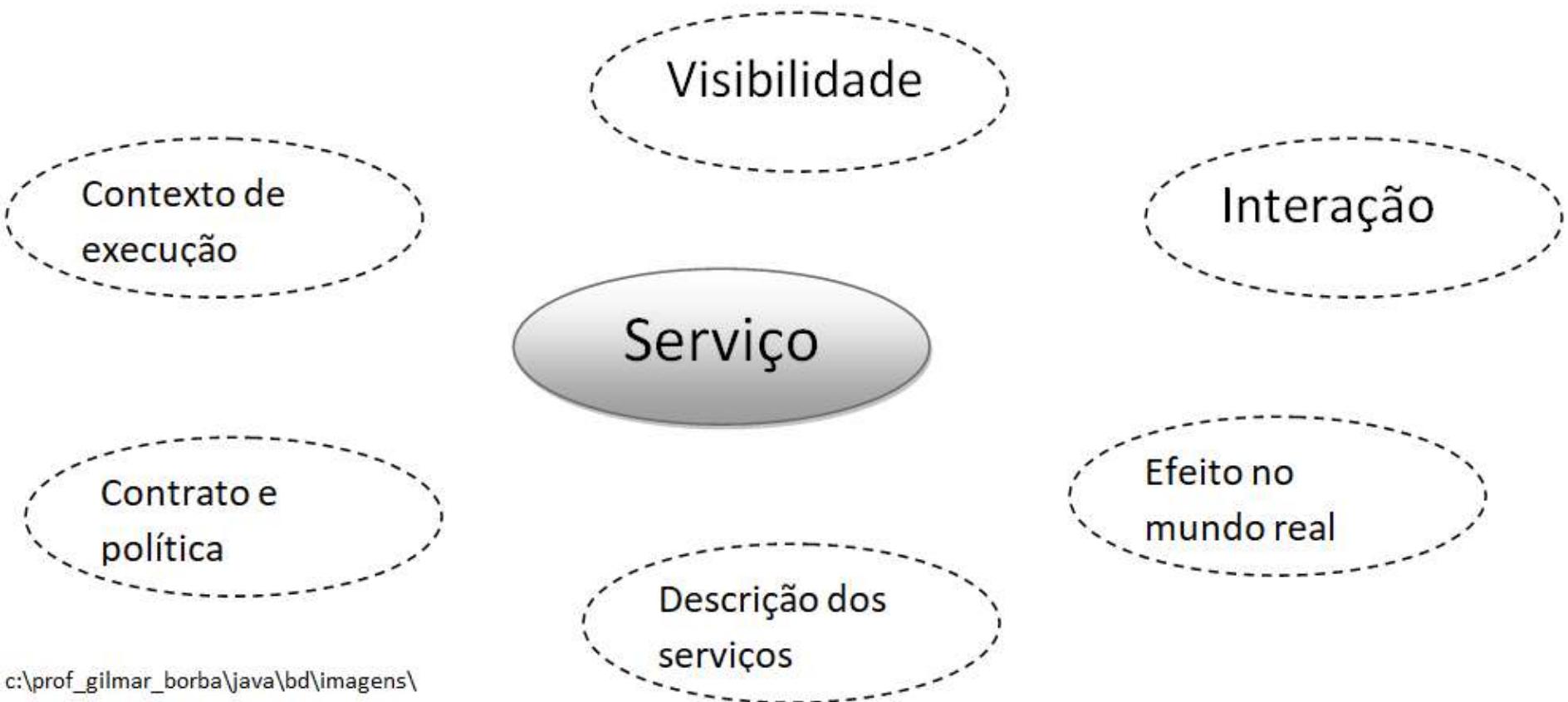
O modelo de referência

Segundo a OASIS um modelo de referência é

“É um framework abstrato para o entendimento dos relacionamentos significativos das entidades de um ambiente para o desenvolvimento de padrões ou especificações consistentes que suportem esse ambiente.”

# **SOA (Service-Oriented Architecture)**

Conceitos importantes no contexto da dinâmica de serviços trazidos no modelo de referência da OASIS



# **SOA (Service-Oriented Architecture)**

## Diferenças entre a OO e SOA

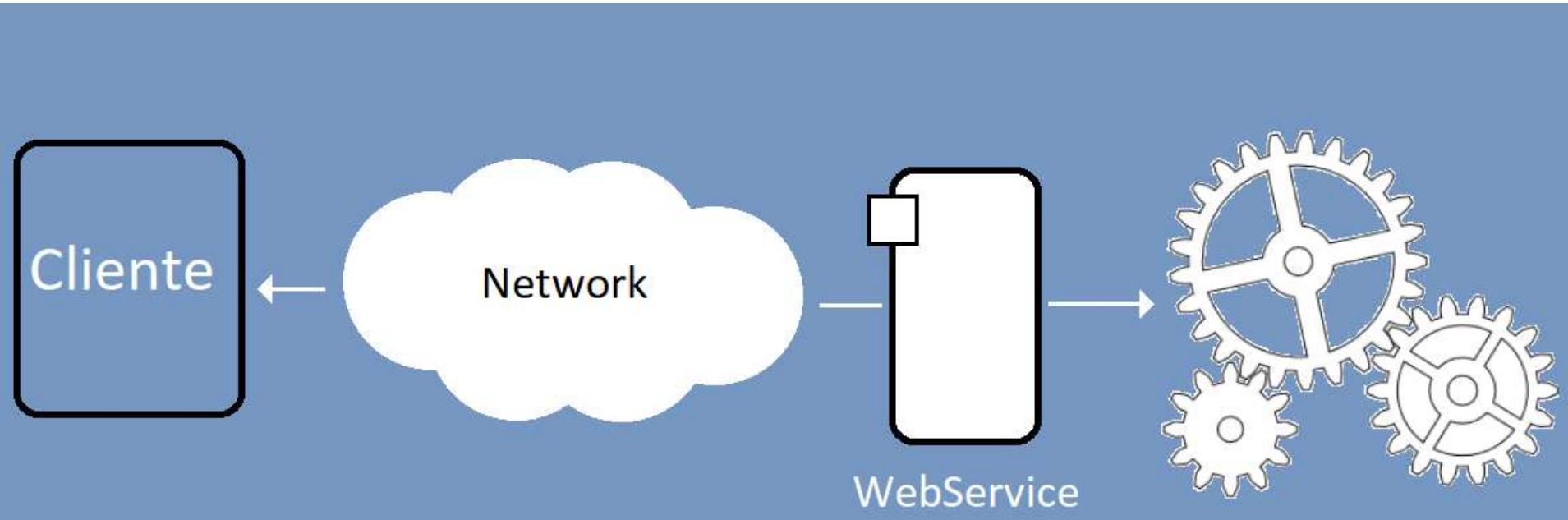
OO	SOA
<p>1 - Tem foco no empacotamento dos dados e operações, a estrutura de dados é interna (classes).</p> <p>2 - Os dados e os métodos pertencem aos objetos</p>	<p>1 - Tem o foco na tarefa ou função do negócio.</p> <p>2 - Proporciona o acesso aos métodos.</p> <p>3 - A estrutura de dados não é interna mas realizada a partir dos parâmetros.</p> <p>4 - A semântica das funcionalidades são colocadas de maneira mais clara (serviços)</p>

c:\prof\_gilmar\_borba\java\bd\imagens\

# WebServices

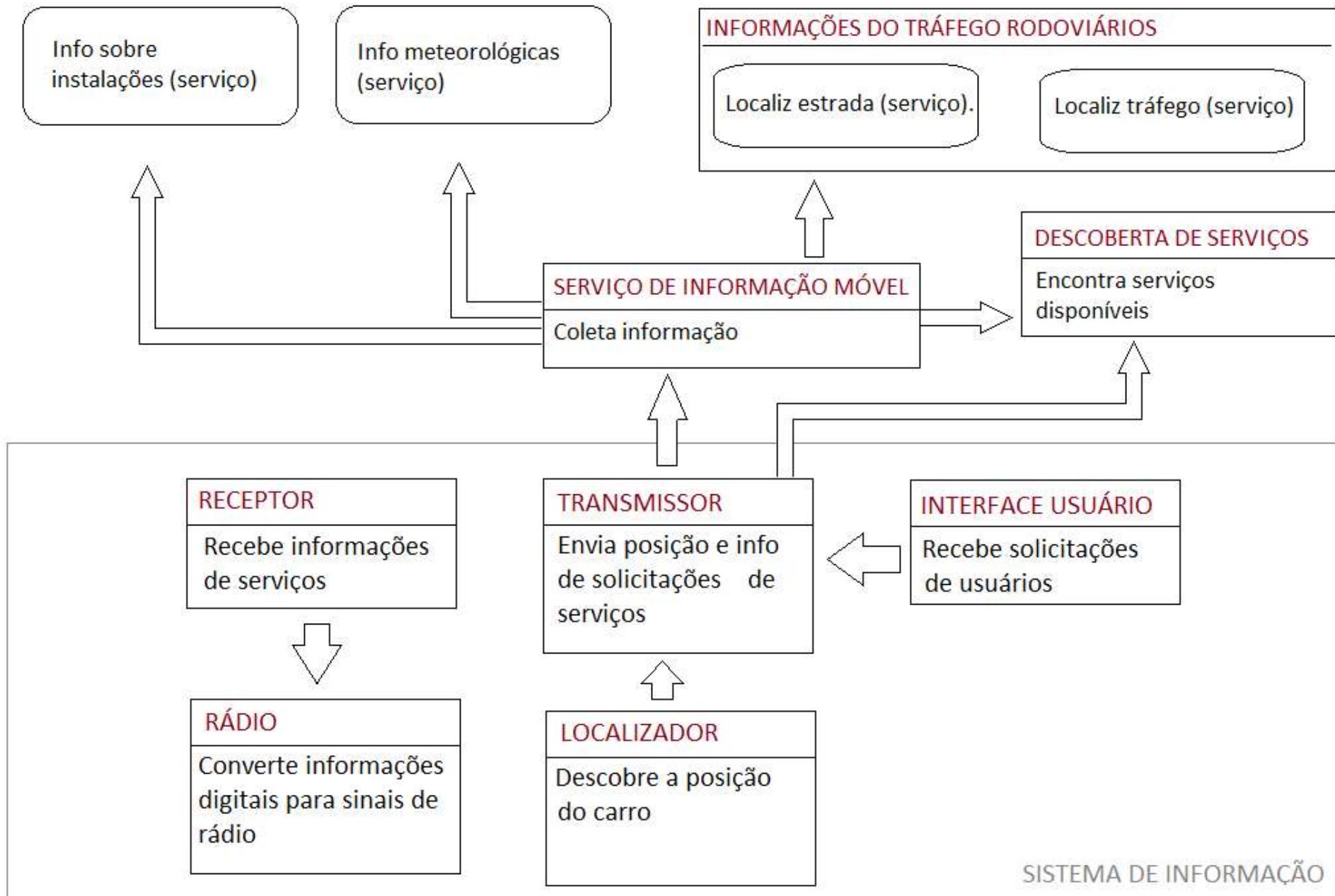
(SOA)

São componentes de aplicações baseados em XML, auto descriptivos e auto contidos que se comunicam a partir de protocolos abertos. São usados por outras aplicações e podem ser descobertos com UDDI.  
*(W3Schools)*



## **Características**

- 1 – Usa protocolos padrões internet (HTTP, XML etc.)*
- 2 – Não depende de linguagem, plataforma, SO etc.*
- 3 - Baixo acoplamento entre consumidor e provedor.*
- 4 – Expõe serviços na quantidade certa de lógica de negócio.*





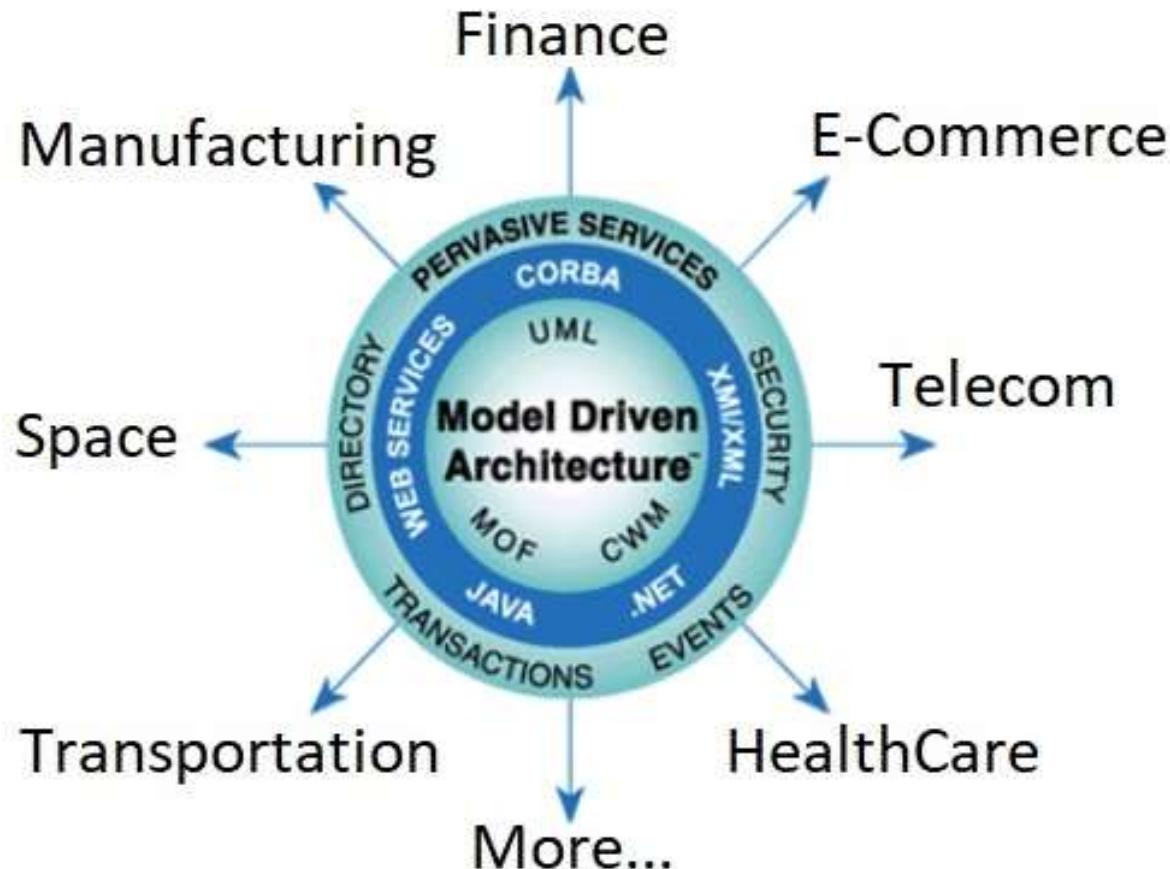
# MDA

# ***MDA (Model Driven Architecture)***

O *Model Driven Architecture* da OMG oferece uma abordagem aberta, neutra do fornecedor, de forma a atender os constantes desafios de mudanças de negócios e tecnologia. Com base nos padrões estabelecidos da OMG, o MDA separa a lógica de negócios e aplicativos da tecnologia de plataforma subjacente. [...]. (<http://www.omg.org/mda/>).

# MDA (Model Driven Architecture)

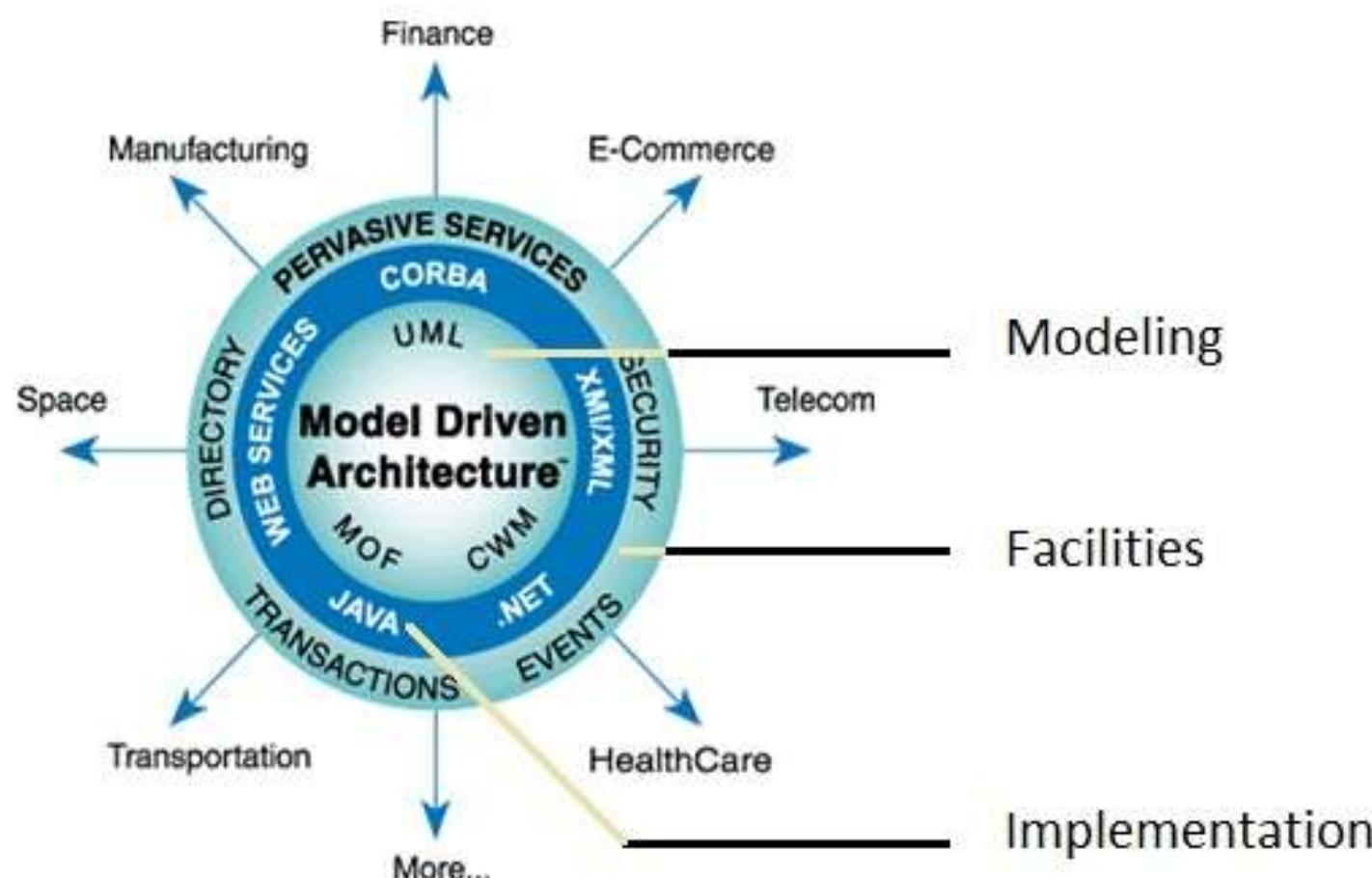
## Áreas e tecnologias relacionadas



<http://ticsoftware.com/Solutions/BluAge/FAQ.aspx>

# MDA (Model Driven Architecture)

## Áreas e tecnologias relacionadas



# **MDA (Model Driven Architecture)**

O MDA é dividido em três etapas ou modelos:

- 1 - Modelo de alto nível de abstração, independente de qualquer tecnologia, PIM (*Platform Independent Model*).
- 2 - Um modelo mais complexo oriundo da transformação do modelo da etapa 1. esse novo modelo é denominado PSM (*Platform Specific Model*). Esse modelo é mais específico como por exemplo um EJB (*Enterprise Java Beans*) ou modelo de banco de dados.
- 3 - O terceira modelo é gerado na última etapa, é a transformação do PSM em código. Trata-se de uma transformação trivial que pode ser automatizada facilmente.

# **MDA (Model Driven Architecture)**

O MDA é dividido em três etapas ou modelos:



# **MDA** (*Model Driven Architecture*)

## Benefícios

Produtividade: O processo de transformação dos modelos é feito de maneira padronizada e uma única vez.

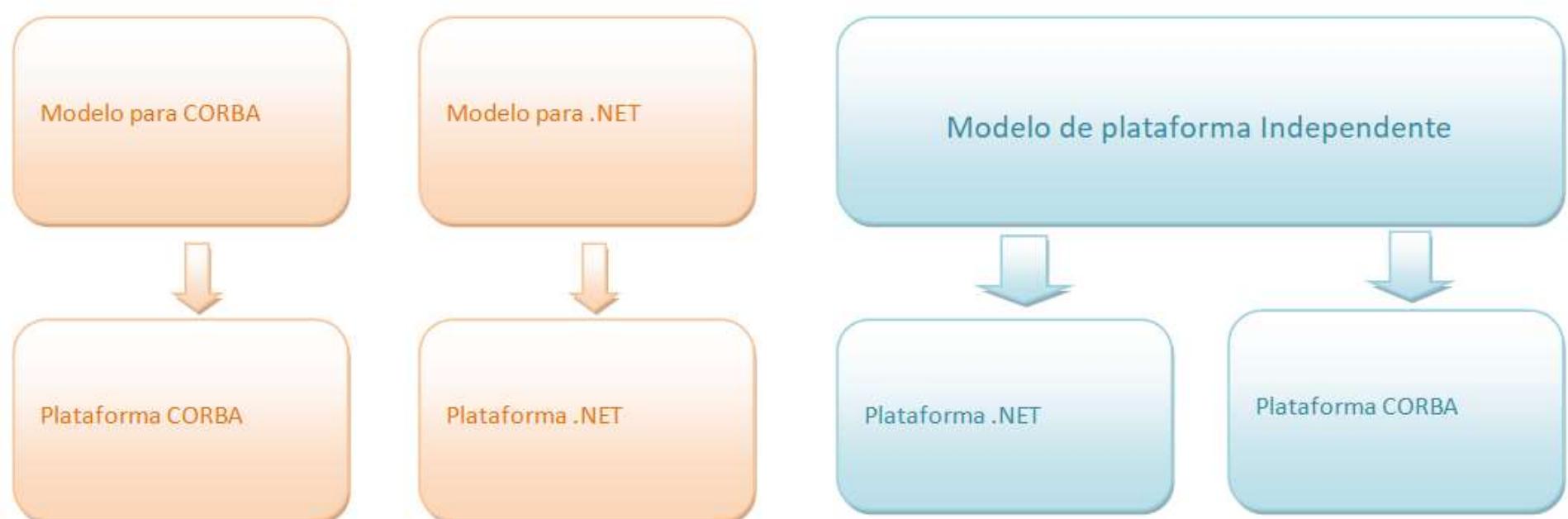
Portabilidade: O modelo inicial (PIM) pode dar origem a vários modelos específicos (PSM) e podem ser independentes de plataforma.

Interoperabilidade: Uma vez que os artefatos são gerados para plataformas diferentes, podem ser usados em padrões abertos.

# **MDA (Model Driven Architecture)**

Plataforma dependente

**MDA**



# **MDA (Model Driven Architecture)**

## **Tecnologias relacionadas**

UML (*Unified Modeling Language*),  
MOF (*Meta-Object Facility*),  
XML (*Extensible Markup Languag*),  
XMI (*Metadata Interchange*),  
EDOC (*Enterprise Distributed Object Computing - EDOC*),  
SPEM (*Software Process Engineering Metamodel*),  
CWM (*Common Warehouse Metamodel*).

# ***MDA (Model Driven Architecture)***

A MDA é a tentativa da OMG para solucionar os problemas de integração a partir de especificações de interoperabilidade independente do fabricante.

# Exercícios

- (22) Definir a arquitetura SOA.
- (23) No contexto da arquitetura SOA definir Serviço.
- (24) No contexto da arquitetura SOA definir Recurso.
- (25) Destacar três características da arquitetura SOA.
- (26) Destacar quatro benefícios da arquitetura SOA.
- (27) Explicar o conceito VISIBILIDADE no contexto da dinâmica de serviços trazidos no modelo de referência SOA da OASIS.
- (28) Explicar o conceito INTERAÇÃO no contexto da dinâmica de serviços trazidos no modelo de referência SOA da OASIS.
- (29) Explicar o conceito EFEITO NO MUNDO REAL no contexto da dinâmica de serviços trazidos no modelo de referência SOA da OASIS.

# Exercícios

- (30) Explicar o conceito DESCRIÇÃO DOS SERVIÇOS no contexto da dinâmica de serviços trazidos no modelo de referência SOA da OASIS.
- (31) Explicar o conceito CONTRATO E POLÍTICA no contexto da dinâmica de serviços trazidos no modelo de referência SOA da OASIS.
- (32) Explicar o conceito CONTEXTO DE EXECUÇÃO no contexto da dinâmica de serviços trazidos no modelo de referência SOA da OASIS.
- (33) Explicar o conceito CONTEXTO DE EXECUÇÃO no contexto da dinâmica de serviços trazidos no modelo de referência SOA da OASIS.
- (34) Diferenciar o uso da orientação a objetos no paradigma OO e no SOA.
- (35) Descrever *Webservice* segundo a definição da W3school.
- (36) Como a padronização de linguagem é feita na construção dos Webservices?

# Exercícios

- (37) Com relação a descrição de *webService* da W3School o que significa dizer que os Webservices são auto contidos?
- (38) Com relação a descrição de *webService* da W3School o que significa dizer que os Webservices são auto descritivos?
- (39) Com relação a descrição de *webService* da W3School o que significa dizer que os Webservices usam protocolos abertos?
- (40) Com relação a descrição de *webService* da W3School o que significa dizer que os Webservices podem ser descobertos com UDDI?
- (41) Definir MDA?
- (42) Como é definido o MDA?
- (43) Quais são os benefícios trazidos pelo MDA?

# Princípio e Padrões de Projeto

# Padrões GoF

Erich Gamma, Richard Helm, Ralph  
Johnson, John Vlissides.

Livro:

*Design Patterns: Elements of Reusable Object-Oriented Software.* 1.ed. Estados Unidos da América:  
*Addison-Wesley, 1995.*

# O que é um padrão de projeto?

É uma técnica usada para identificar, abstrair, documentar e padronizar os diversos aspectos recorrentes do projeto de software orientado a objetos, de modo a torná-los reutilizáveis em outros projetos.

Os padrões de projeto fornecem um vocabulário compartilhado com outros desenvolvedores. Quando você tem um vocabulário, pode se comunicar mais facilmente com outros desenvolvedores e inspirar aqueles que não conhecem os padrões a começar a aprendê-los. Isso também eleva o seu raciocínio sobre as arquiteturas, permitindo que você pense no nível do padrão e não no nível de código. (FREEMAN, Eric; FREEMAN Elizabeth, 2007).

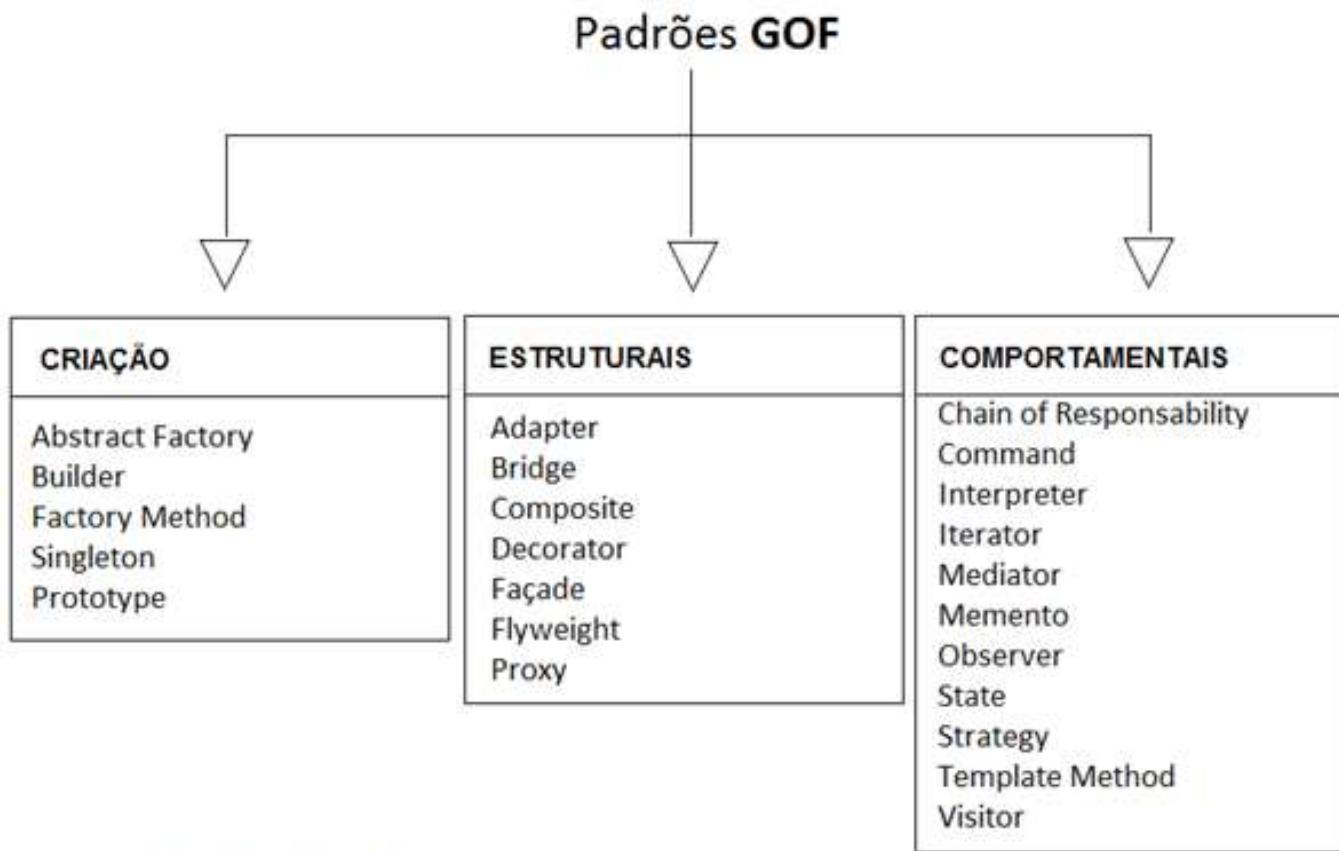
“Os padrões de projeto são descobertos e não inventados!!”

# Como os padrões de projeto são apresentados?

- . **Intenção.** Uma descrição sobre o padrão.
- . **Motivação.** Um exemplo ou descrição de como o padrão surgiu.
- . **Aplicabilidade.** Aplicação do padrão de projeto.
- . **Estrutura.** Um diagrama da UML que represente o padrão.
- . **Participantes.** Descrição dos componentes do diagrama.
- . **Colaborações.** Descrição das colaborações dos componentes.
- . **Conseqüências.** Vantagens e desvantagens do padrão.
- . **Implementação.** Codificação de exemplo do padrão.

Apresentação dos padrões de projeto segundo Eric Gamma, Richard Helm, Ralph Johnson e John Vlissides (GoF).

# Categorias



c:\prof\_gilmar\_borba\java\bd\imagens\

**CRIAÇÃO:** lidam com problemas de criação de objetos.

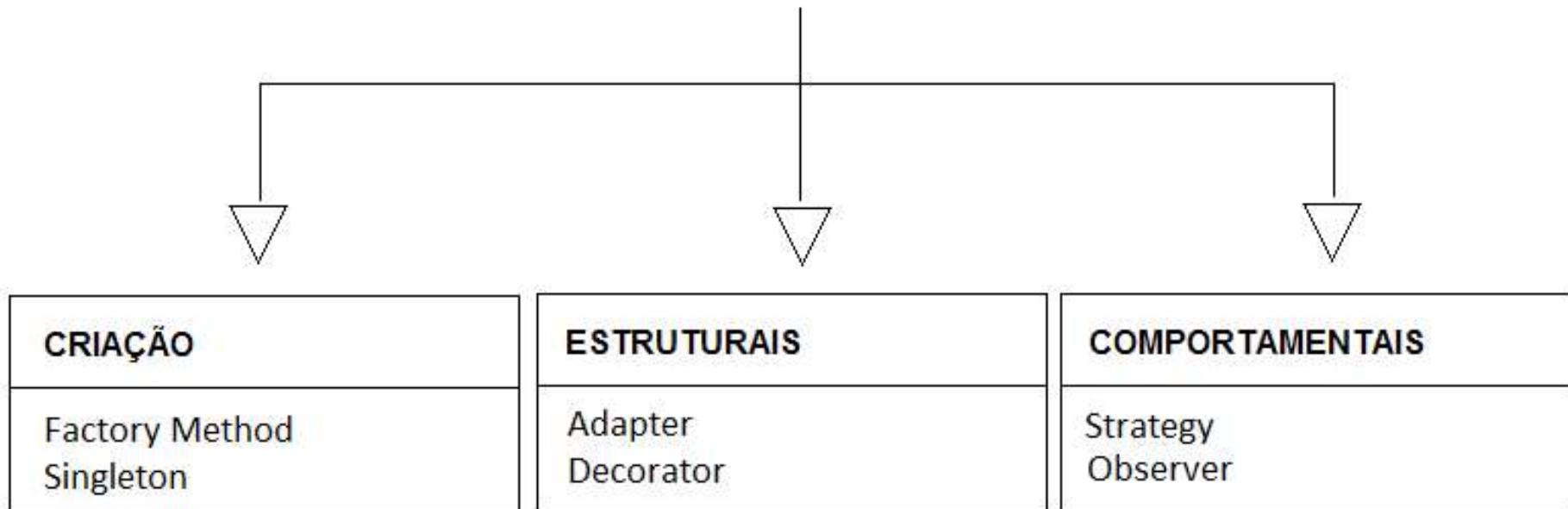
**ESTRUTURAIS:** lidam com problemas de relacionamento entre os objetos.

**COMPORTAMENTAIS:** lidam com problemas de responsabilidades entre os objetos.

# Discutiremos os padrões:

(Observando os itens: intenção, motivação, aplicabilidade, vantagens e desvantagens, diagrama de classe e exemplo prático implementado em uma linguagem orientada a objetos – JAVA)

## Padrões GOF



# O PADRÃO SINGLETON

# O PADRÃO SINGLETON

INTENÇÃO:

Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma.

(GAMMA et al., 2000:130)

# O PADRÃO SINGLETON

## MOTIVAÇÃO:

É importante para algumas classes ter uma, e apenas uma, instância. Por exemplo, embora possam existir muitas impressoras em um sistema, deveria haver somente um spooler de impressora. Da mesma forma, deveria existir somente um sistema de arquivos e um gerenciador de janelas. [...] Um sistema de contabilidade será dedicado a servir somente a uma companhia.

(GAMMA et al., 2000:130)

É uma convenção para garantir que um objeto seja criado uma única vez. Fornece um ponto de acesso global, assim como uma variável global mas sem as suas desvantagens.

# O PADRÃO SINGLETON

## APLICABILIDADE:

- 1- Quando é necessário ter uma única instância de uma classe e essa instância tiver que dar acesso aos clientes através de um ponto bem conhecido.
- 2- A única instância tiver de ser extensível através de subclasses possibilitando aos clientes usar essa instância sem alterar o seu código.

(GAMMA et al., 2000:130).

# O PADRÃO SINGLETON

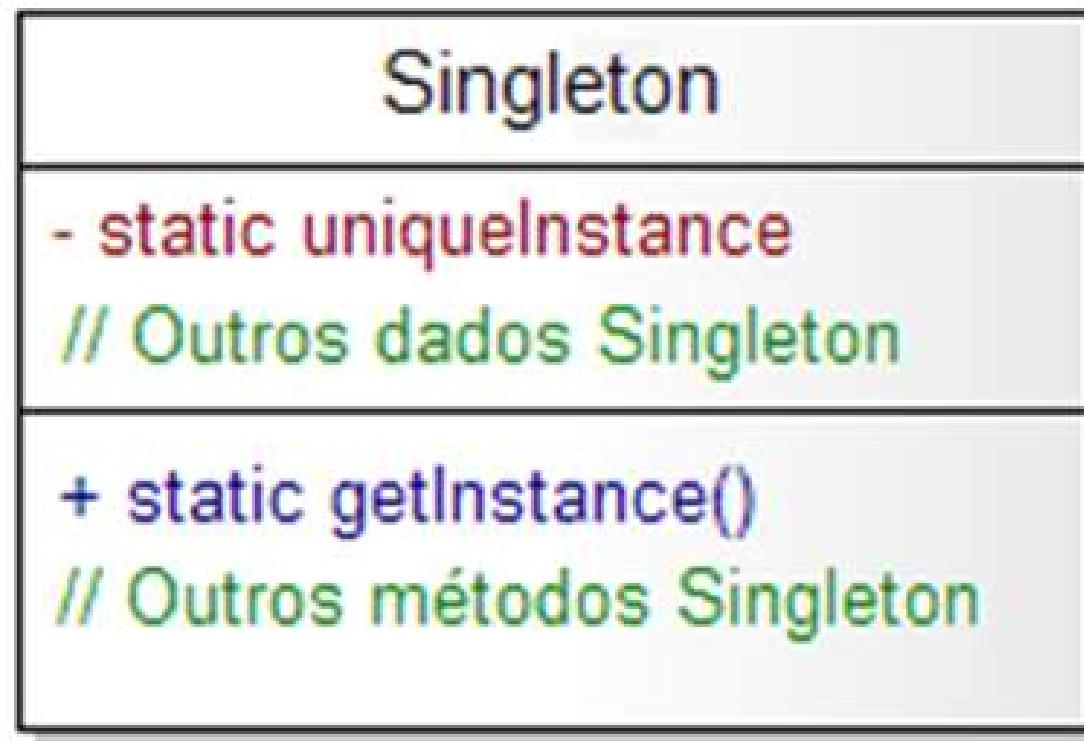
## VANTAGENS:

- 1- Acesso controlado a uma única instância.
- 2- Evitar o uso de variáveis globais que armazenam instâncias únicas.
- 3- Maior flexibilidade uma vez que permite a mesma abordagem para controlar a quantidade de instâncias que a aplicação utiliza.

(GAMMA et al., 2000:130).

# O PADRÃO SINGLETON

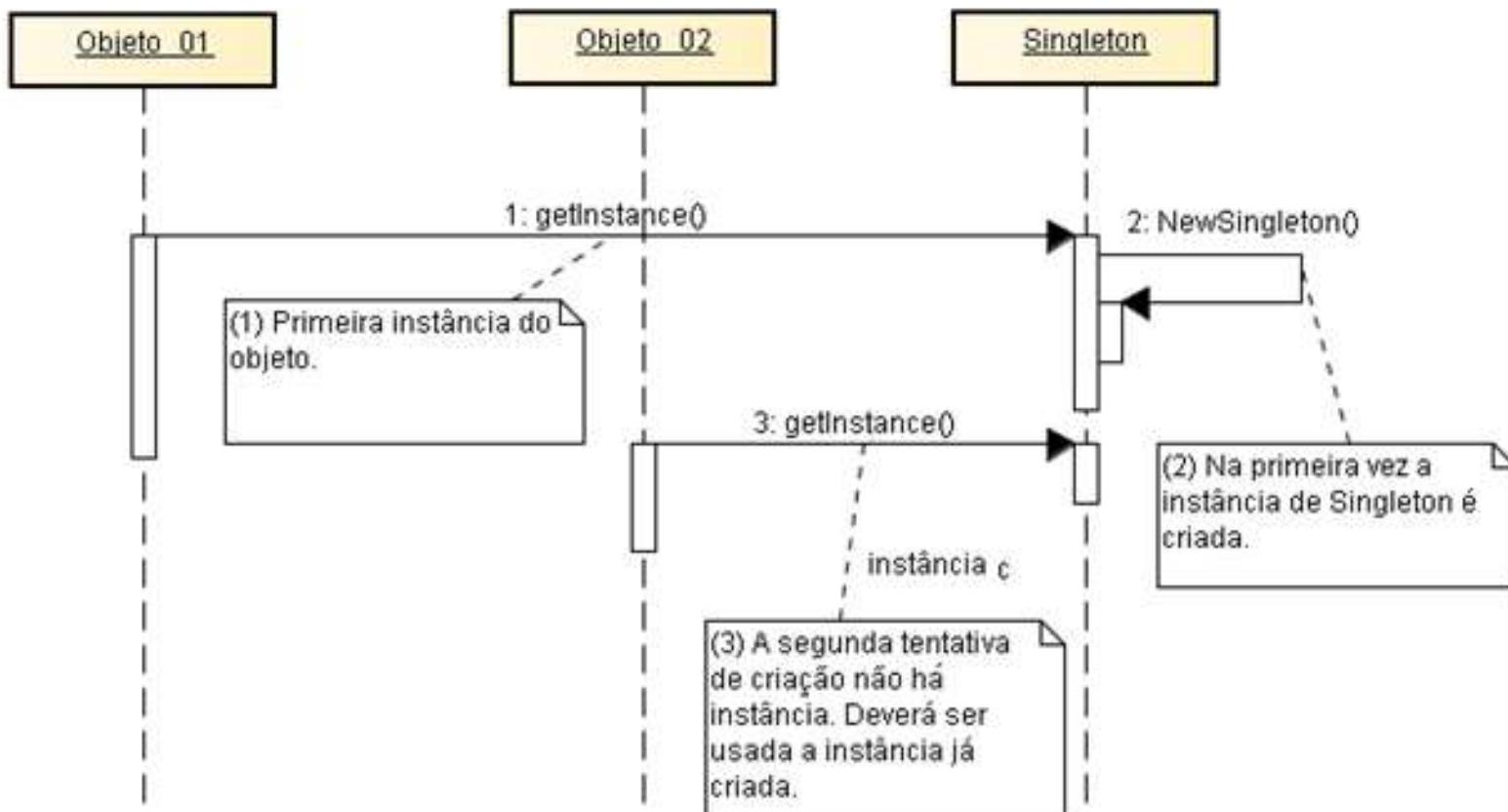
## ESTRUTURA (Diagrama de Classe)



c:\prof\_gilmar\_borba\java\bd\imagens\

# O PADRÃO SINGLETON

## ESTRUTURA (Diagrama de Sequência)



c:\prof\_gilmar\_borba\java\bd\imagens\



## Por que ter apenas uma instância de uma classe?

Há objetos que devem ser únicos em uma aplicação, para evitar problemas de violação de acesso, uso excessivo de recursos ou comportamento inadequado.

# O PADRÃO SINGLETON

## CODIFICAÇÃO (Parcial)

```
public class Singleton {  
  
    private static Singleton uniqueInstance;  
  
    private Singleton() {  
        System.out.println( "Objeto criado!" );  
    }  
    public static Singleton getInstance() {  
        if (uniqueInstance == null)  
            uniqueInstance = new Singleton();  
        return uniqueInstance;  
    }  
}
```

# O PADRÃO SINGLETON

## OBSERVAÇÕES:

- Garante que uma única instância pode ser acessada de forma global.
- Garante que nenhuma outra instância seja criada.
- É útil para o caso de objetos grandes e caros  
(dificuldades de múltiplas instâncias)
- Construtores JAVA sempre retornam novos objetos e nunca  
um objeto já criado.
- A classe pode ser final c / construtor *private*. Evita criação  
da instância da classe.
- Desvantagem: métodos "static" perdem as vantagens de  
implementar interfaces.

# O PADRÃO SINGLETON

## O princípio da Responsabilidade Única

**SRP** (*Single responsibility Principle*)

De acordo com Page-Jones (1988), a coesão é a medida da força de uma associação funcional de atividades de processamento (normalmente dentro de um único módulo). (PAGE-JONES, 1988:381). A coesão de um módulo ou classe pode ser verificada de vários pontos de vista, como por exemplo: funcional, lógico, temporal etc.

Martin (2011), define o Princípio da Responsabilidade Única:

uma classe deve ter apenas um motivo para mudar.

A ideia é: uma classe, uma responsabilidade!

## SRP (Single responsibility Principle)

Ao implementar uma classe, é importante que esta tenha uma "mentalidade única", ou seja, uma única responsabilidade. Quando uma classe possui muitos objetivos (ou responsabilidades) é importante decompor esses objetivos de forma a ter classes preferencialmente atômicas, Martin (2011) esclarece essa questão:

[...] cada responsabilidade é um eixo de mudança. Quando os requisitos mudarem, a alteração se manifestará por meio de uma mudança na responsabilidade entre as classes. Se uma classe assumir mais de uma responsabilidade, ela terá mais de um motivo para mudar.  
(MARTIM, 2011: 135).

[...] Se uma classe tem mais de uma responsabilidade, as responsabilidades se tornariam acopladas. Mudanças em uma responsabilidade podem prejudicar ou inibir a capacidade da classe de cumprir as outras. Esse tipo de acoplamento leva a projetos frágeis que estragam de maneiras inesperadas quando alterados.  
(MARTIM, 2011: 135).

# Exercícios



- (44) Definir padrões de Projeto.
- (45) Explicar a frase: “Os padrões de projeto são descobertos e não inventados!!”
- (46) Quais são as categorias dos padrões de projeto? Explique cada uma delas.
- (47) Por que aplicar o padrão *Singleton* nas aplicações? Exemplificar.
- (48) como os padrões de projeto são apresentados?
- (49) O que acontece se a visibilidade do construtor da classe *Singleton* for alterada para *public*?
- (50) Qual é a melhor prática para tornar a classe *Singleton* para *Thread Safe*?
- (51) Explicar o princípio de projeto da Responsabilidade Única.
- (52) Qual é a relação entre o padrão de projeto *Singleton* e o princípio da Responsabilidade única?



# O PADRÃO

# FACTORY

# METHOD

# O PADRÃO FACTORY METHOD

Define uma interface para criar um objeto, mas permite às classes decidir qual classe instanciar. O *Factory Method* permite a uma classe deferir a instanciação para subclasses. (FREEMAN, 2007:121).

Ou seja:

Os clientes nunca criam objetos diretamente, eles pedem a fábrica para fazer isso por eles.

# O PADRÃO FACTORY METHOD

Define uma interface para criar um objeto, mas permite às classes decidir qual classe instanciar. O *Factory Method* permite a uma classe deferir a instanciação para subclasses.

(FREEMAN, 2007:121)

# O PADRÃO FACTORY METHOD

## INTENÇÃO:

Definir uma interface para criar um objeto, mas deixar as suas subclasses decidirem que classe instanciar. O *Factory Method* permite adiar a instanciação para subclasses.

(GAMMA et al., 2000:112).

Também conhecido como *Virtual Constructor*.

# O PADRÃO FACTORY METHOD

## APLICABILIDADE:

- 1- Uma classe que não pode antecipar a classe que deve criar.
- 2- Uma classe quer que suas subclasses especifiquem os objetos que criam.
- 3- Classes delegam responsabilidades para suas subclasses e você quer encontrar o conhecimento de qual subclasse auxiliar que é a classe delegada.

(GAMMA et al., 2000:113).

# O PADRÃO FACTORY METHOD



## MOTIVAÇÃO:

### EXEMPLO:

No caso de um framework com duas abstrações: classes do tipo *Application* e *Document* (ambas abstratas). Os clientes devem “especializá-las” para executar sua funcionalidade específica, por exemplo: `desenharAplicação()` e `desenharDocumento()`.

### PROBLEMA:

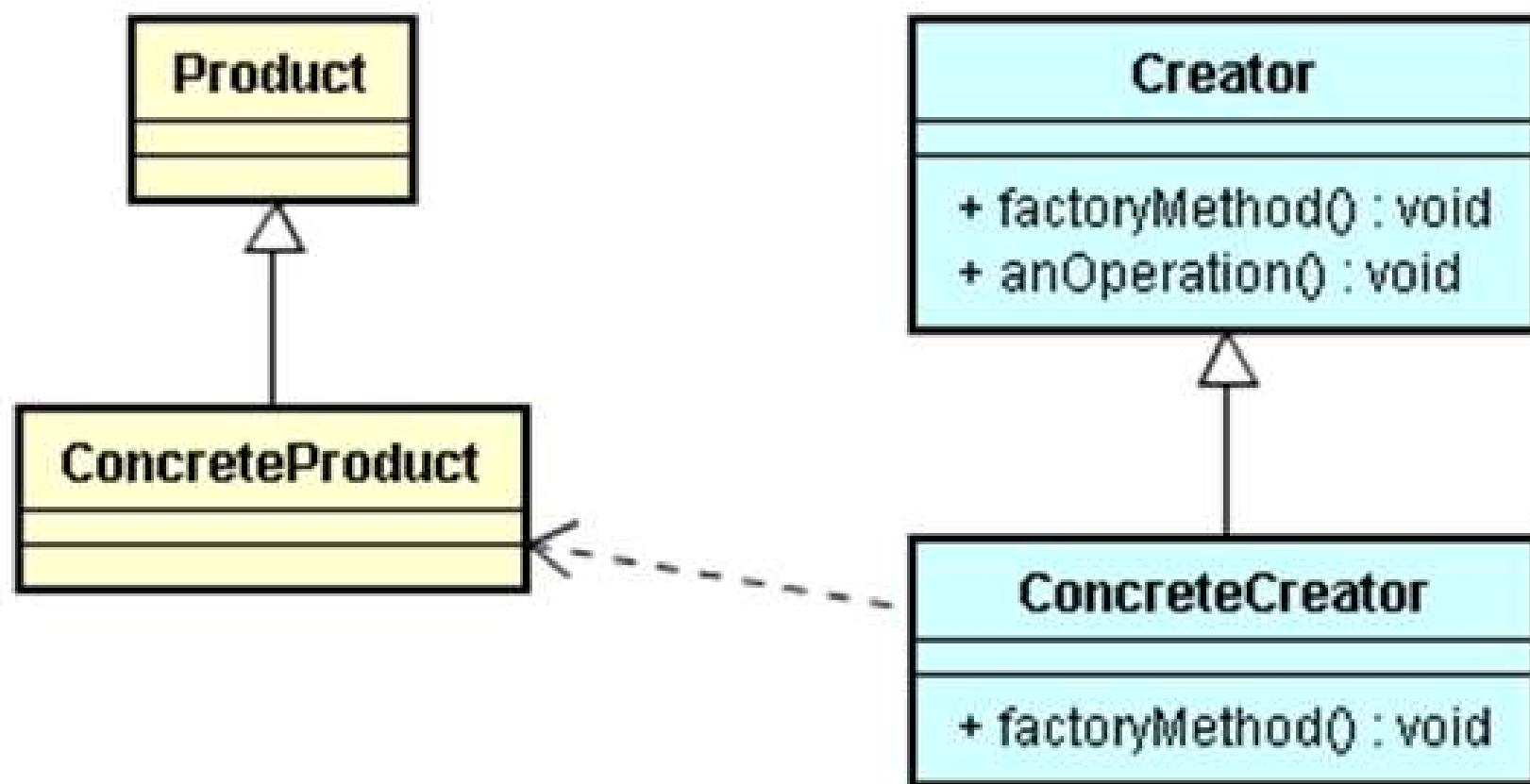
Um problema: uma vez que subclasse de *Document* possui especificidades da aplicação, *Application* não conhece a subclasse a ser instanciada, ou seja, *Application* sabe quando mas não qual *Document* deve criar.

**SOLUÇÃO:** Usar o padrão *Factory Method* para encapsular o conhecimento de qual subclasse *Document* criar e consequentemente mover este conhecimento para fora do framework.

(GAMMA et al., 2000:112).

# O PADRÃO FACTORY METHOD

## ESTRUTURA (Diagrama de Classes)

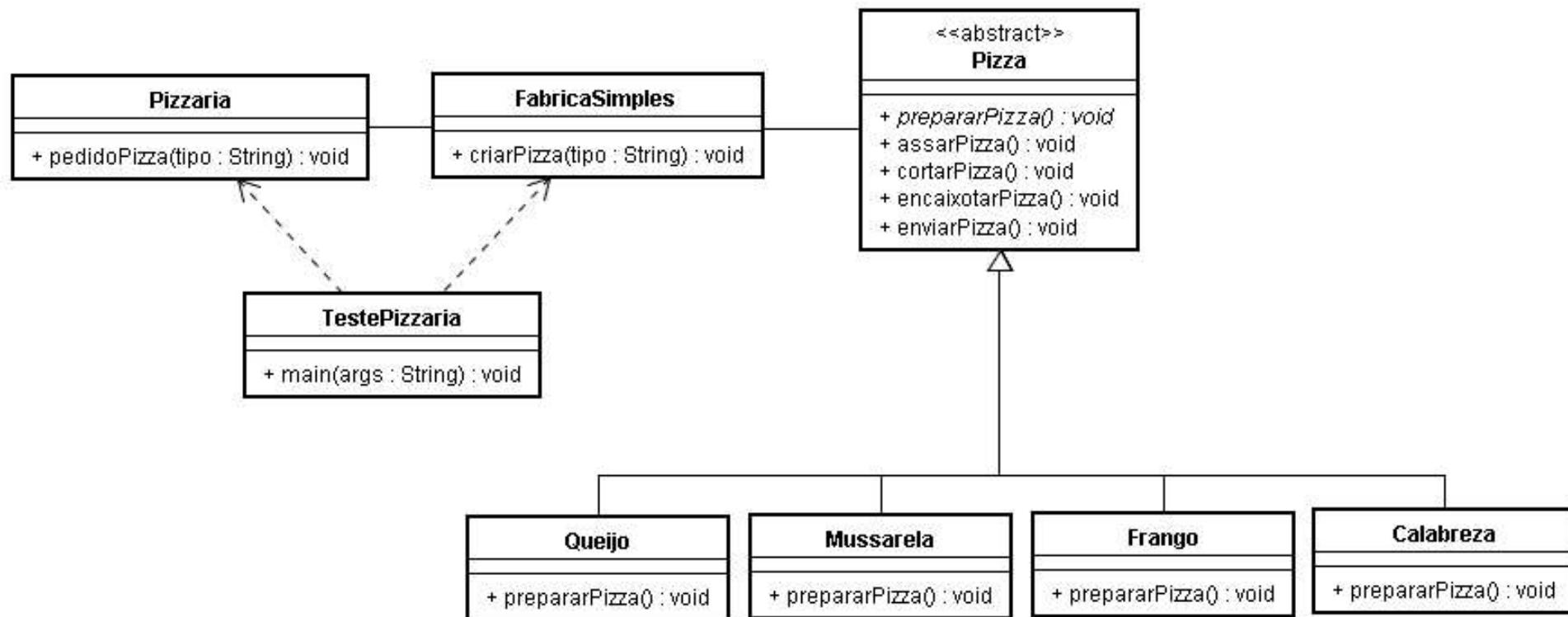


c:\prof\_gilmar\_borba\java\bd\imagens\

(GAMMA et al., 2000:113).

# O PADRÃO FACTORY METHOD

## EXEMPLO (Implementação)



# O PADRÃO FACTORY METHOD

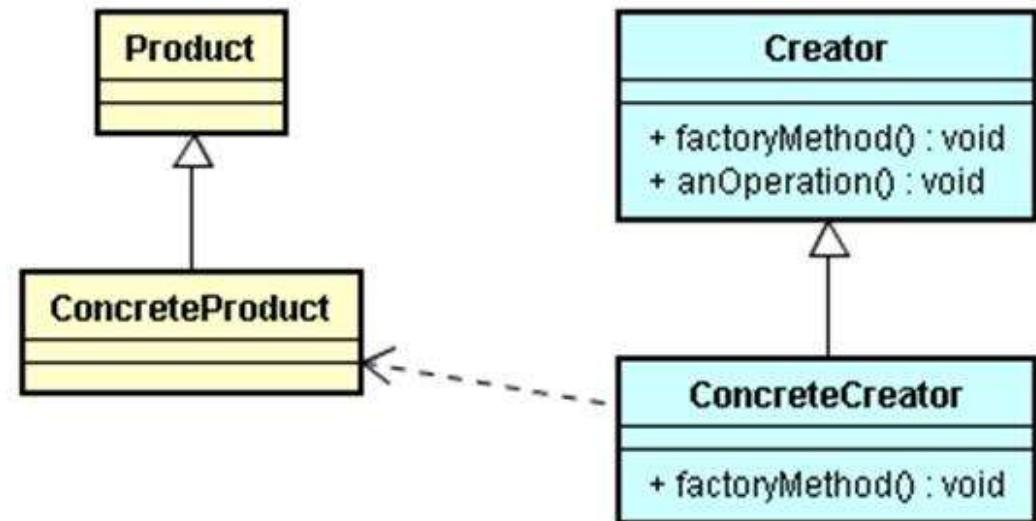
## Vantagens:

Não requer o acoplamento de classes específicas para aplicação (em nível de código), lida apenas com a superclasse ou interface, a fim de conhecer os métodos e atributos de uma forma genérica, permitindo manipular as subclasses ou implementações de maneira específica.

(GAMMA et al., 2000:112).

Promove maior flexibilidade para as classes uma vez que o “método-fábrica” funciona como uma conexão para que uma das subclasses possa estender um objeto.

(GAMMA et al., 2000:112).



# O PADRÃO FACTORY METHOD

## Consequências/Desvantagens

Há possibilidade, em alguns casos, de criar subclasses que suportam o método-fábrica para prover funcionalidades a classes paralelas oferecendo assim maior portabilidade.

Essa é uma solução viável para casos específicos, pois tornaria improutivo e "caro" a sua implementação.

(GAMMA et al., 2000, pags.114-115).

# O PADRÃO FACTORY METHOD

## CODIFICAÇÃO (Parcial)

```
public Pizza criarPizza(String tipo) {
    Pizza pizza = null;
    if (tipo.equalsIgnoreCase("Calabresa")) {
        pizza = new Calabresa();
    }
    else if (tipo.equalsIgnoreCase("Queijo")) {
        pizza = new Queijo();
    }
    else if (tipo.equalsIgnoreCase("Frango")) {
        pizza = new Frango();
    }
    else if (tipo.equalsIgnoreCase("Mussarela")) {
        pizza = new Mussarela();
    }
    return pizza;
}
```

```
public class FabricaMoto {
    public static Moto getMoto(String tipoMoto) {
        if (tipoMoto.equalsIgnoreCase("Honda")) {
            return new Honda();
        }
        if (tipoMoto.equalsIgnoreCase("Suzuki")) {
            return new Suzuki();
        }
        if (tipoMoto.equalsIgnoreCase("Yamaha")) {
            return new Yamaha();
        }
        if (tipoMoto.equalsIgnoreCase("Kawasaki")) {
            return new Kawasaki();
        }
        else
            return null;
    }
}
```

c:\prof\_gilmar\_borba\java\bd\imagens\

# Exercícios

- (53) Definir o padrão de projeto *Factory Method*.
- (54) Em que momento é viável a criação de uma fábrica?
- (55) Quais são as vantagens em criar uma fábrica?
- (56) Explicar o princípio de projeto Aberto Fechado.
- (57) O Padrão de projeto Fábrica viola algum princípio de projeto?



# O PADRÃO DECORATOR

# O PADRÃO DECORATOR

## O Princípio Aberto-Fechado (Open-Closed Principle)

Entre; estamos abertos. Fique à vontade para estender nossas classes com qualquer comportamento novo que desejar. Se suas necessidades ou requisitos mudarem (e sabemos que isso vai acontecer), vá adiante e crie suas próprias extensões.  
(FREEMAN, 2007:87).

Infelizmente estamos fechados. É isso mesmo, gastamos muito tempo deixando esse código correto e sem erros, por isso não podemos deixá-los alterar o código existente. Ele deve permanecer fechado a modificações. Se você não gostar fale com o gerente.  
(FREEMAN, 2007:87).

Cuidado ao escolher as áreas de código que precisam ser estendidas; aplicar o Princípio Aberto-fechado EM TODO LUGAR é um desperdício desnecessário e pode gerar um código complexo e de difícil compreensão.  
(FREEMAN, 2007:87).

# O PADRÃO DECORATOR

O padrão *Decorator* anexa responsabilidades adicionais a um objeto de maneira dinâmica. Os decoradores fornecem uma alternativa flexível para a subclasse estender funcionalidade. (FREEMAN, 2007:100).

# O PADRÃO DECORATOR

## INTENÇÃO:

Dinamicamente, agregar responsabilidades adicionais a um objeto. Os decoradores fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades. (GAMMA et al., 2000:170).

Também conhecido como Wrapper.

# O PADRÃO DECORATOR

## APLICABILIDADE:

- 1- Adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, ou seja, sem afetar outros objetos.
- 2- Quando a extensão através do uso de subclasses não é prática.
- 3- Para proporcionar um número menor de subclasses.

# O PADRÃO DECORATOR



## VANTAGENS:

- 1- Maior flexibilidade do que a herança.
- 2- Evita classes sobre carregadas na parte superior da hierarquia de classes.

## DESVANTAGENS:

- 1- Grande quantidade de pequenos objetos.

(GAMMA et al., 2000:173).

# O PADRÃO DECORATOR



## Fraco/Baixo Acoplamento

- . Minimização de dependências.
- . Maximização do reuso.

### O que é o acoplamento?

É o grau de ligação entre dois objetos/classes, ou seja, uma medida que mostra o quanto uma classe está ligada a outra, ou, possui conhecimento de outra classe.

Um acoplamento fraco mostra que uma classe não é dependente de outras classes

### Problemas do alto/forte acoplamento:

- . Há problemas de reuso da classe;
- . Mudanças em uma classe acarretarão em mudanças em outras classes;
- . Há dificuldades em entender uma classe separadamente, pois ela depende de outras.

# O PADRÃO DECORATOR

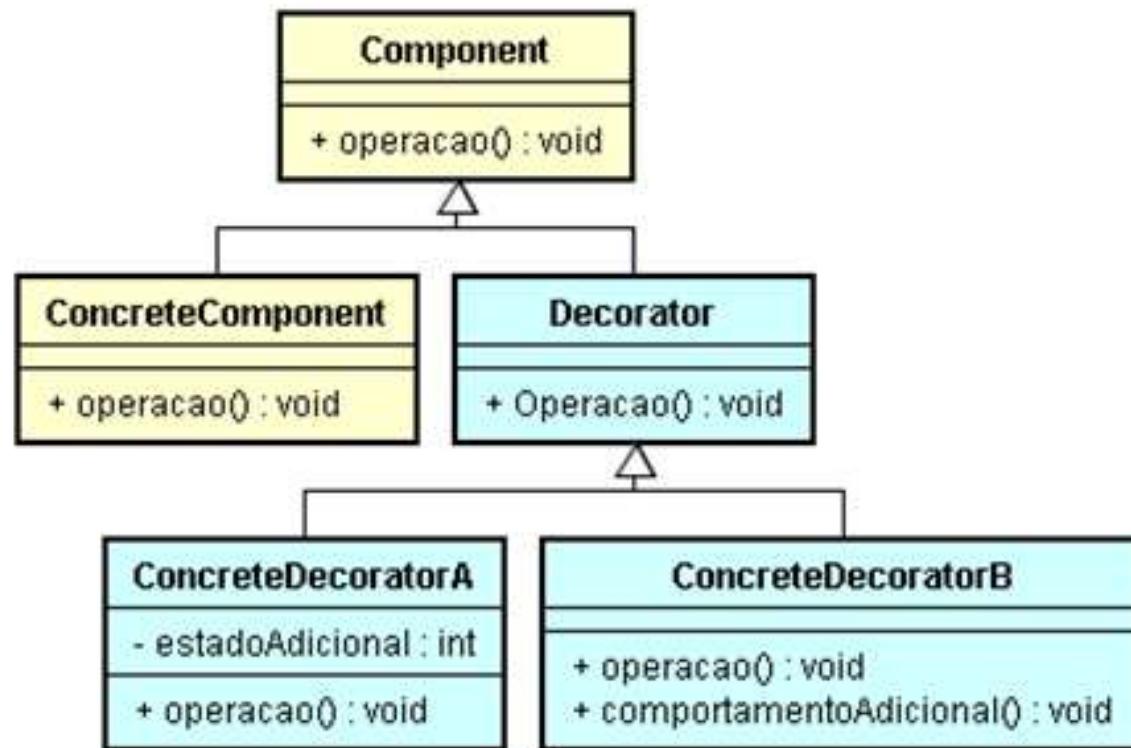
## O Princípio Aberto-Fechado (Open-Closed Principle)

As classes devem estar abertas para a extensão, mas fechadas para modificação.

Nosso objetivo é permitir que as classes sejam facilmente estendidas para incorporar um novo comportamento sem modificar o código existente. O que conseguimos se fizermos isso? Projetos que são resistentes a mudanças e suficientemente flexíveis para assumir novos recursos para atender aos requisitos que mudam.  
(FREEMAN, 2007:87).

# O PADRÃO DECORATOR

## Diagrama de classe



c:\prof\_gilmar\_borba\java\bd\imagens\

(GAMMA et al., 2000:172).

# O PADRÃO DECORATOR

```
import javax.swing.*;
public class PadraoDecorador {
    static public void main(String args[]) {
        int numeros[] = {0,1,2,3,4,5,6,7,8,9};
        JTextArea saida = new JTextArea(5,10);
        JScrollPane rolagem = new JScrollPane(saida);
        saida.setText("VETOR DE NÚMEROS\n");
        for(int i=0;i<=9;i++) {
            saida.append("numeros["+i+"]="+numeros[i]+\n");
        }
        JOptionPane.showMessageDialog(null,rolagem);
        System.exit(0);
    }
}
```

c:\prof\_gilmar\_borba\java\bd\imagens\

# O PADRÃO DECORATOR

```
import javax.swing.*;
public class PadraoDecorador {
    static public void main(String args[]) {
        int numeros[] = {0,1,2,3,4,5,6,7,8,9};
        JTextArea saida = new JTextArea(5,10);
        JScrollPane rolagem = new JScrollPane(saida);
        saida.setText("VETOR DE NÚMEROS\n");
        for(int i=0;i<=9;i++) {
            saida.append("numeros["+i+"]="+numeros[i]+\n");
        }
        JOptionPane.showMessageDialog(null,rolagem);
        System.exit(0);
    }
}
```

Caixa de Texto:  
Componente a ser  
decorado.

# Exercícios



- (58) Qual é a principal função do padrão de projeto *Decorator*?
- (59) Qual é a relação entre a Herança e o padrão de projeto *Decorator*?
- (60) Quando usar o padrão *decorator*?
- (61) O que você entende por Composição?
- (62) Qual é o papel da composição no padrão *Decorator*?
- (63) O que você entende por acoplamento?
- (64) Qual é a relação entre o padrão *decorator* e o acoplamento?

# O PADRÃO ADAPTER

# O PADRÃO ADAPTER

O padrão *Adapter* converte a interface de uma classe para outra interface que o cliente espera encontrar. O adaptador permite que classes com interfaces incompatíveis trabalhem juntas.

(FREEMAN, 2007:100).

# O PADRÃO ADAPTER

## Entendendo o padrão Adapter

Tomada da parede européia



A tomada europeia expõe uma interface para alimentação elétrica.

Adaptador para CA



O adaptador faz a conversão de uma interface para outra.

Pino americano para CA



O laptop americano espera uma outra interface.

# O PADRÃO ADAPTER



## INTENÇÃO:

Converter a interface de uma classe em outra interface, esperada pelos clientes. O *Adapter* permite que classes com interfaces incompatíveis trabalhem em conjunto - o que, de outra forma, seria impossível.

Também conhecido como **Wrapper** (assim como o padrão *Decorator* é um padrão empacotador).

(GAMMA et al., 2000:140).

# O PADRÃO ADAPTER

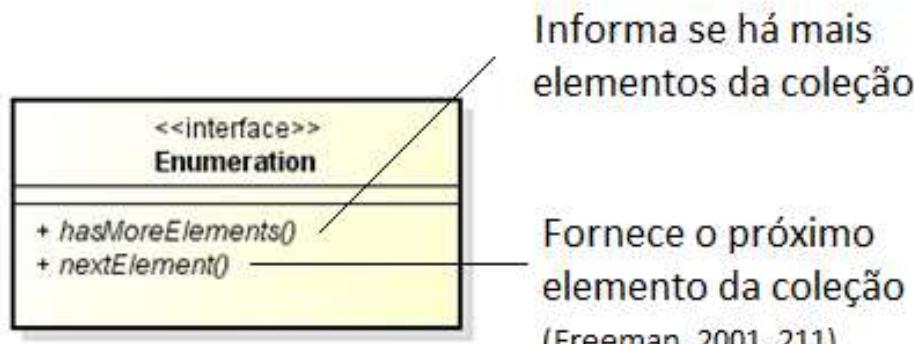
Na prática:

Dificuldades em modificar, por exemplo, um componente, devido à falta do seu código fonte. Nesse caso, cria-se um segundo componente que se acoplará ao antigo e fornecerá as funcionalidades esperadas pelos novos componentes.

# O PADRÃO ADAPTER

## Na prática:

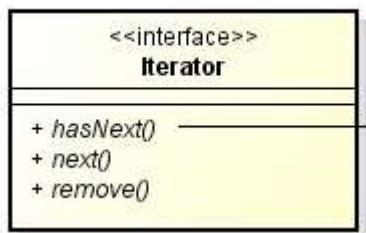
Se você lida com Java há algum tempo, provavelmente se lembra que os primeiros tipos de coleções (Vector, Stack, Hastable e alguns outros), implementam um método chamado `elements()`, que retorna uma enumeração. A interface Enumeration permite que você acesse seqüencialmente os elementos de uma coleção sem saber especificamente como eles são gerenciados. (FREEMAN, 2007:211).



Informa se há mais elementos da coleção

Fornece o próximo elemento da coleção  
(Freeman, 2001, 211)

c:\prof\_gilmar\_borba\java\bd\imagens\



Assim como o método `hasMoreElements()` da interface `Enumeration`, esse método informa se você já examinou todos os itens da coleção.

(Freeman, 2001, 211)

remove um item da coleção.

Fornece o próximo elemento da coleção.

Quando a Sun introduziu as suas classes Collections mais recentes, elas começaram a usar uma interface `Iterator` que, como `Enumeration`, permite realizar a iteração através de um conjunto de itens numa coleção, mas também acrescenta a capacidade de remover itens.  
(FREEMAN, 2007:211).

c:\prof\_gilmar\_borba\java\bd\imagens\

# O PADRÃO ADAPTER

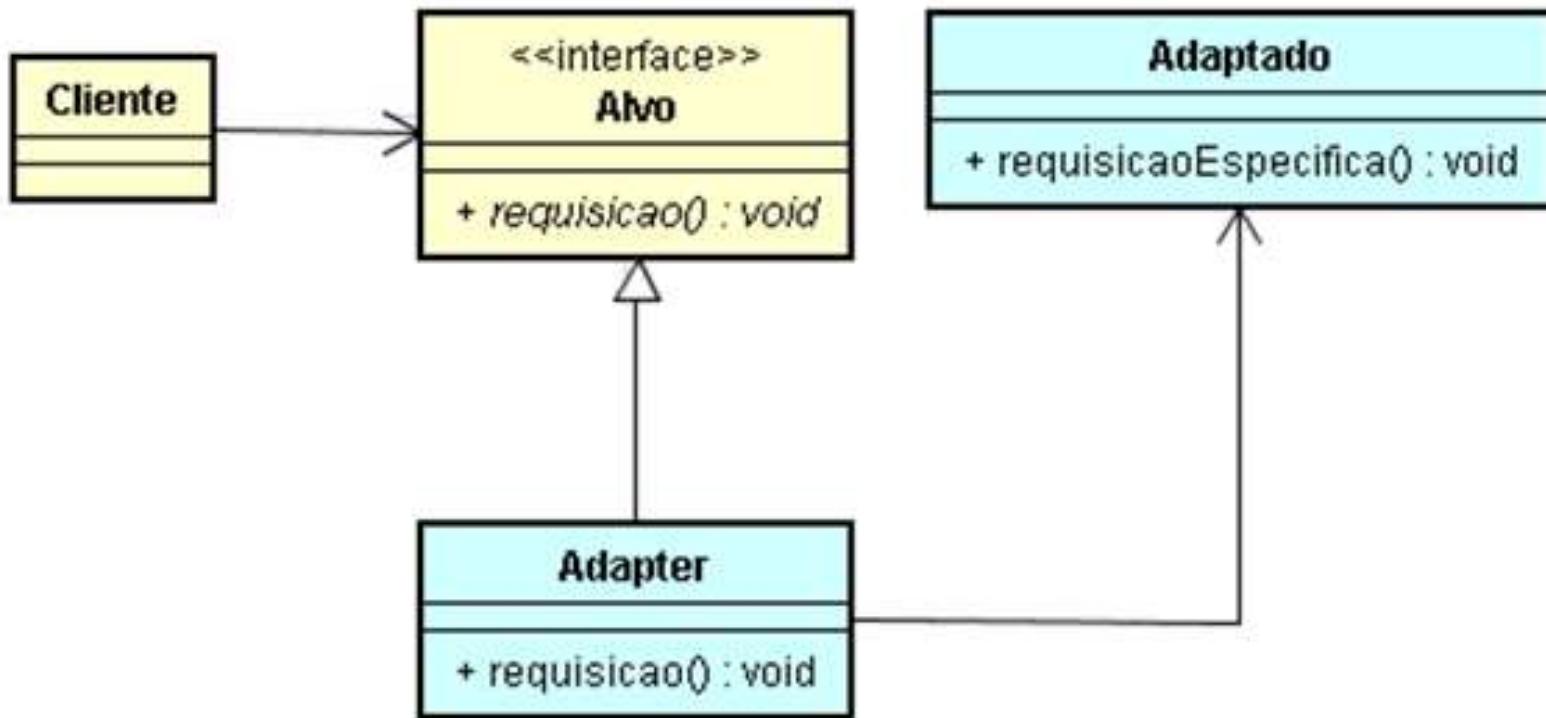
## APLICABILIDADE:

- 1- Quando é necessário usar uma classe existente e a interface não corresponder à necessidade.
- 2- Quando é necessário usar várias subclasses existentes e for impraticável adaptar essas interfaces criando subclasses para cada uma.

(GAMMA et al., 2000:142).

# O PADRÃO ADAPTER

Diagrama de classe



# O PADRÃO ADAPTER

O PADRÃO DECORATOR X ADAPTER (FREEMAN, 2007:214)

## DECORATOR

*Eu sou importante, meu trabalho é sobre responsabilidade - quando um decorator está envolvido você sabe que novos comportamentos ou responsabilidades serão adicionados ao seu projeto.*

*[...] quando temos que decorar uma interface grande, nem queira saber a quantidade de código que precisamos usar.*

*[...] Não pense que nós recebemos toda a fama; às vezes eu sou apenas um decorator que está sendo envelopado por quem sabe quantos outros decoradores. [...]*

*[...] também fazemos isso, só que permitimos que novos comportamentos sejam adicionados às classes sem alterar o seu código existente. Continuo achando que os adaptadores são apenas decoradores enfeitados. [...]*

## ADAPTER

*Vocês querem toda a fama, enquanto nós adaptadores, ficamos nas trincheiras fazendo o trabalho sujo: a conversão de interfaces. [...]*

*Experimente ser um adaptador quando é preciso juntar várias classes para fornecer a interface que o seu cliente está esperando. isso sim é que é dureza.*

*Pois, quando os adaptadores estão fazendo o seu trabalho, nossos clientes nem sabem que nós existimos. Pode ser um trabalho ingrato.*

*Não, não, de jeito nenhum. Nós sempre convertemos a interface daquilo que envelopamos, vocês nunca fazem isso. Eu diria que um decorator é como um adaptador, só que não muda a interface!*

...

# Exercícios



(65) Definir o padrão *Adapter*.

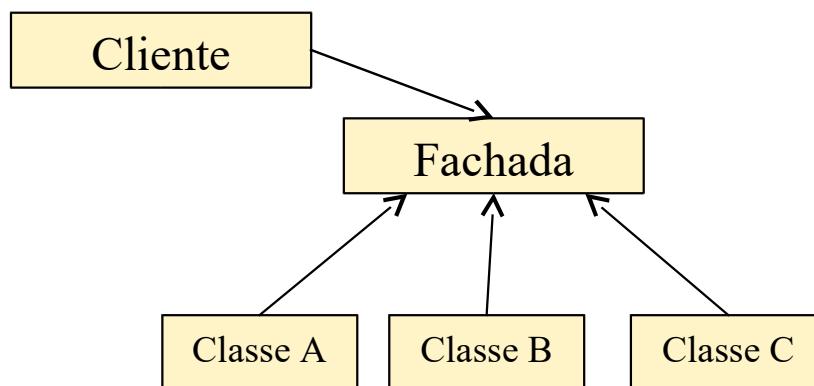
(66) Discorrer sobre o acoplamento no padrão *Adapter*.

# O PADRÃO FACADE E AS INTERFACES

Aproveitando que estamos falando de interfaces (durante a discussão do padrão Adapter) ...

O Padrão Facade fornece uma interface unificada para um conjunto de interfaces em um sistema. A fachada define uma interface de nível mais alto que facilita a utilização do subsistema. (FREEMAN, 2001:221).

Enquanto o padrão adapter converte a interface de uma classe para algo que o cliente está esperando, o padrão Facade (Fachada) oculta toda a complexidade de uma ou mais classes através de uma fachada, mais clara e resumida. O padrão fachada usa **o princípio do conhecimento mínimo**, ou seja, haverá uma classe, na estrutura onde a fachada for implementada, que resumirá todos os componentes para o cliente, mantendo o cliente simples e flexível.



**Conhecimento mínimo:** o Cliente só tem um amigo, ou seja, só conhece a Fachada. Para Freeman (2007), na OO, ter somente um amigo é uma coisa BOA!

• • •



## IMPORTANTE

- . Verificar a descrição, motivação e os objetivos do padrão.
- . Verificar como os padrões se agrupam e se relacionam.
  - . Identificar o que é variável no projeto.
  - . Analisar/criticar a codificação/exemplo do padrão.

Não esquecer que os padrões solucionam problemas específicos de projeto.

# O PADRÃO STRATEGY

# O PADRÃO STRATEGY

Define uma família de algoritmos e encapsula cada um deles de forma a os tornar intercambiáveis. O encapsulamento, neste caso, reduz as consequências indesejadas de alterações de código. A estratégia permite ainda que o algoritmo varie independente dos clientes que o utilizam, isto é feito priorizando a composição ao invés da herança e programando para interface ao invés da implementação.

(FREEMAN, 2007:42)

# O PADRÃO STRATEGY

## INTENÇÃO:

Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

(GAMMA et al., 2000:292).

O padrão Strategy é conhecido também como padrão **Policy**.

# O PADRÃO STRATEGY



## APLICABILIDADE:

- 1- Classes relacionadas são diferentes apenas no que se refere ao seu comportamento.
- 2- Há dados no algoritmo, os quais os clientes não deveriam conhecer.
- 3- Eliminar comandos condicionais da linguagem. É uma alternativa ao uso de comandos condicionais para a seleção de comportamentos específicos.

# O PADRÃO STRATEGY



## VANTAGENS:

- 1- Criação de família de algoritmos relacionados.
- 2- Oferece mais uma opção para o uso de subclasses.
- 3- As estratégias eliminam comandos condicionais da linguagem à medida que agrupam algoritmos similares.
- 4- Permite escolher diferentes tipos de implementação para o mesmo comportamento.

## DESVANTAGENS:

- 1- Aumento da quantidade de objetos criados.

# O PADRÃO STRATEGY



## Resumindo:

- Encapsular o que varia.
- Dar prioridade à composição ao invés da herança.
- Programar para a interface e não para a implementação.



## A constante no desenvolvimento de Software

[...] qual é a única coisa com a qual podemos contar sempre no desenvolvimento de software?

(FREEMAN, 2007:30).



## A constante no desenvolvimento de Software

[...] qual é a única coisa com a qual podemos contar sempre no desenvolvimento de software?

O A Ç A Y E T J A

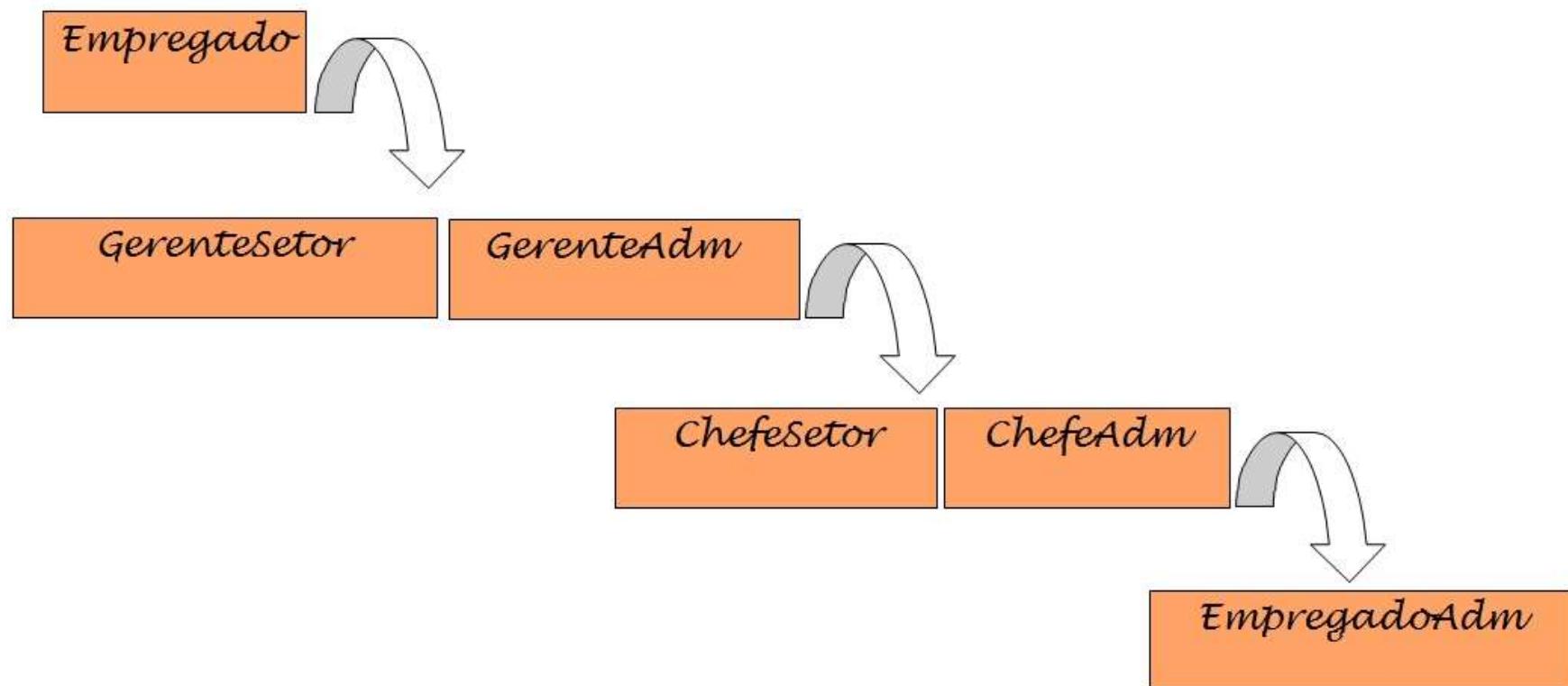
(Use um espelho para ver a resposta)

## O problema da herança!

- A herança, na programação OO, requer um bom planejamento da hierarquia de classes.
- Uma herança mal planejada pode causar a quebra do encapsulamento e aumentar o acoplamento dos futuros objetos, comprometendo o funcionamento e a manutenção do sistema.
- Ao alterar uma classe Empregado, essa alteração pode causar problemas na classe Gerente Administrativos que por sua vez causará problema na classe Chefe Administrativo que causará impactos na classe Empregado Administrativo e assim sucessivamente.
- O recurso da herança à primeira vista é excelente, mas há problemas nessa característica da programação OO.

# O PADRÃO STRATEGY

O problema da herança!

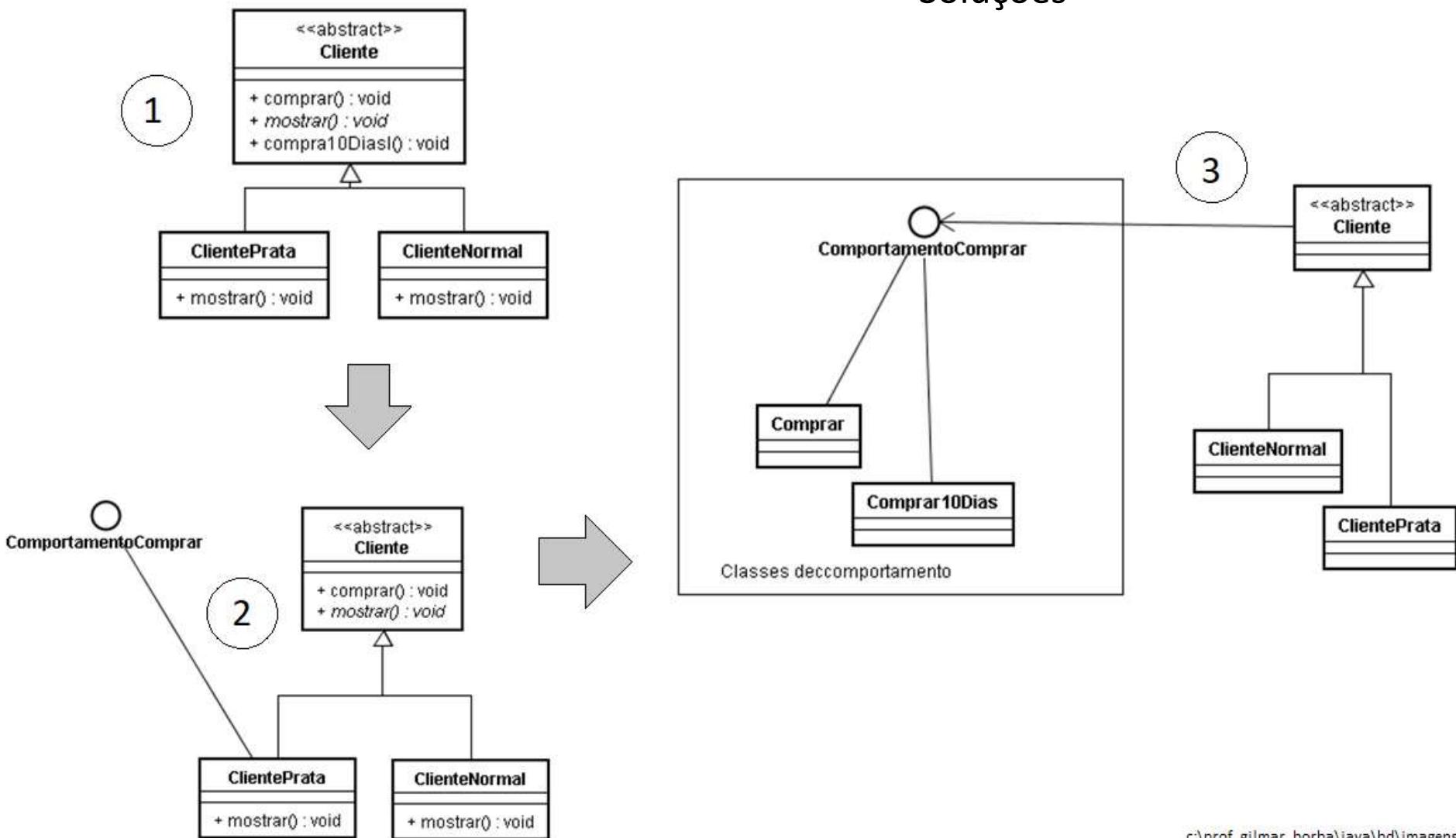


## Na prática!

- Temos **comportamentos** de classes que podem ser aplicados em diferentes situações.
- Temos classes com **comportamentos** (métodos) complexos (if/else ...).
- Temos classes com **comportamentos** comuns com pequenas diferenças entre elas.
- Temos famílias com estruturas similares mas que se diferenciam no **comportamento**.
- Temos algoritmos (**comportamentos**) complexos e o cliente não precisa conhecer essa complexidade.

# O PADRÃO STRATEGY

## Soluções





# Discussão final

(baseado em Freeman (2007))

**Devemos primeiro implementar o aplicativo e verificar o que muda para aplicar a estratégia?**

O ideal é antecipar essa ideia em tempo de projeto, planejando essa flexibilidade antes mesmo da antecipação.



## Discussão final

(baseado em Freeman (2007))

**É comum ter classes que representam apenas comportamentos?**

Sim, as classes representam “coisas” (estados) e métodos (comportamentos). Mas até um comportamento pode ter estado e método.

(invariância ...)

## Discussão final

(baseado em Freeman (2007))

**Ao “juntar” clientes que compram em 10 dias, 20 dias, ou em outros tipos de promoções, o que estamos na verdade querendo dizer com isso?**

A ideia do relacionamento É-UM é interessante, poderíamos criar classes que herdam comportamentos de compras, mas já vimos os problemas da herança. Quando juntamos classes com comportamentos diferentes em um módulo de comportamento, como fizemos, estamos, de certa maneira, compondo esses vários comportamentos.

Estamos usando relacionamentos TEM-UM.



## Discussão final

(baseado em Freeman (2007))

A ideia é dar prioridade para a  
composição!

# Exercícios

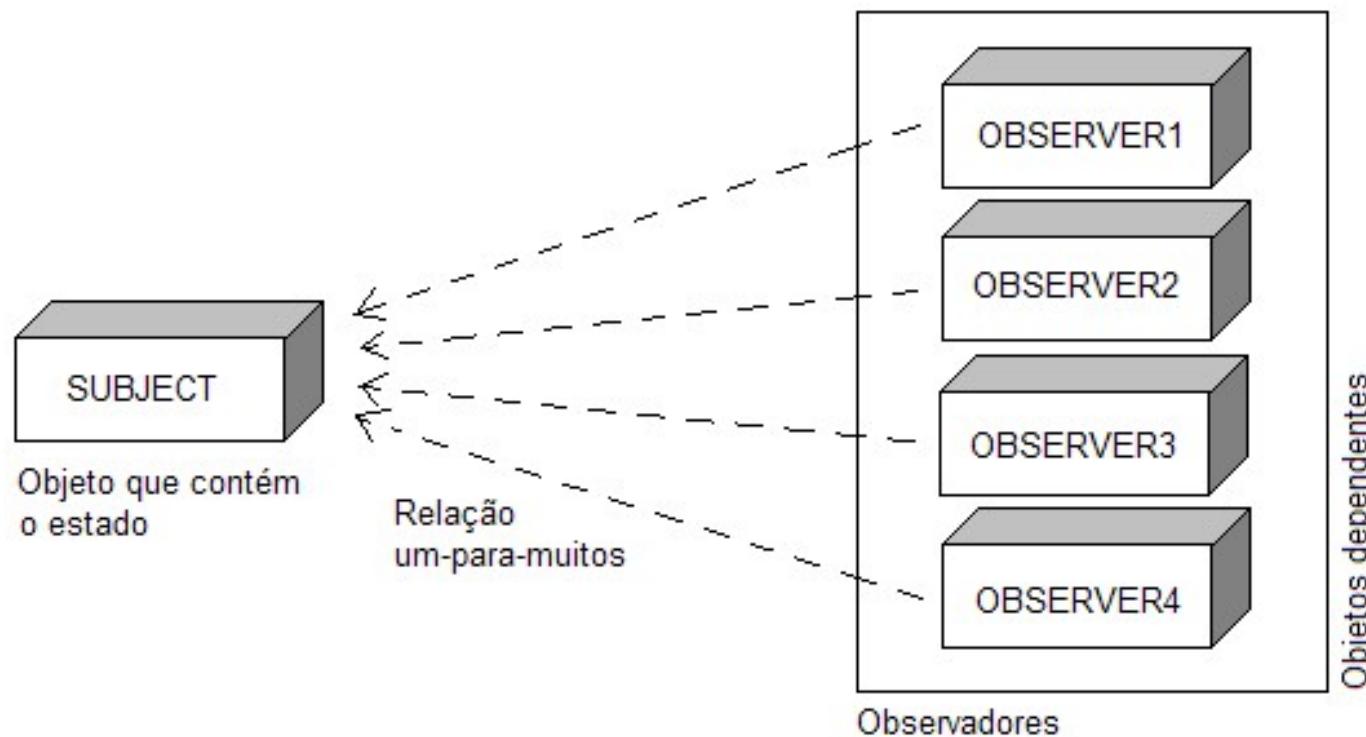
- (67) Quais são as principais características do padrão de projeto *Strategy*?
- (68) Ao dizer que o padrão *Strategy* "elimina comandos condicionais da linguagem", qual princípio de projeto se relaciona a essa afirmativa?
- (69) Qual é a principal desvantagem do padrão *Strategy*?



# O PADRÃO OBSERVER

# O PADRÃO OBSERVER

Define uma dependência um-para-muitos entre os objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados automaticamente. (FREEMAN, 2007:60)



# O PADRÃO OBSERVER

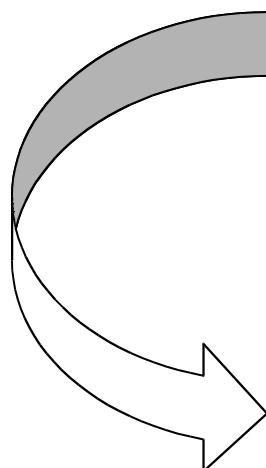
Não deixe de saber quando algo de importante acontece! Há um padrão que mantém os objetos atualizados quando algo importante acontece. O padrão Observer é um dos mais usados no JDK. O padrão observer usa as relações um-para-muitos e ligações leves.

(FREEMAN, 2007:51).

# O PADRÃO OBSERVER

Quando dois objetos estão levemente ligados, podem interagir, mas sabem muito pouco um do outro.

O padrão ***Observer*** fornece um design de objeto onde os sujeitos e observadores são levemente ligados.



- 1 – A única coisa que o sujeito sabe sobre um observador é que ele implementa uma certa interface.  
(A interface Observer).
- 2 – Podemos adicionar novos observadores a qualquer momento.  
(o sujeito só depende da lista de objetos que implementam a interface).
- 3 – Nunca precisamos modificar o sujeito para adicionar novos tipos de observadores.  
(basta implementar a interface).
- 4 – Podemos reutilizar sujeitos ou observadores independentes uns dos outros.  
(... Os dois não estão fortemente ligados).
- 5 – Alterações no sujeito ou num observador não irão afetar o outro.  
(eles estão levemente ligados e cumprem a obrigação de implementar as interfaces)

(FREEMAN, 2007:62).

# O PADRÃO OBSERVER

## INTENÇÃO:

Definir uma dependência *um-para-muitos* entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.

(GAMMA et al., 2000:292).

Também conhecido como **Dependents, Publisher-Subscribe**.

# O PADRÃO OBSERVER

## APLICABILIDADE:

- 1- Usar esse padrão quando uma abstração possuir dois aspectos, um dependente do outro.
- 2- Quando uma mudança em um objeto exigir mudança em outro(s) objeto(s).
- 3- Quando um objeto deverá ser capaz de notificar outros objetos, mesmo sem saber quem são esses objetos (comunicação do tipo broadcast).
- 4- Para implementar atualizações inesperadas.

# O PADRÃO OBSERVER

## CONSEQUÊNCIAS (Vantagens e Desvantagens):

### 1- Acoplamento abstrato entre *Subject* e *Observer*.

[...] por não serem fortemente acoplados, *Subject* e *Observer* podem pertencer a diferentes camadas em um sistema. Um *subject* de nível mais baixo pode comunicar-se com um observador de nível mais alto, desta maneira mantendo intacta as camadas do sistema.

[...] se *Subject* e *Observer* forem agrupados, então o objeto resultante deve cobrir duas camadas (e violar a estrutura de camadas) [...]

### 2 - Comunicação do tipo broadcast.

[...] O *Subject* não se preocupa com quantos objetos interessados existem; sua única responsabilidade é notificar seus observadores.

### 3- Atualizações inesperadas.

[...] Como um observador não tem conhecimento da presença dos outros, eles podem ser cegos para o custo global de mudança do *subject*.

# O PADRÃO OBSERVER

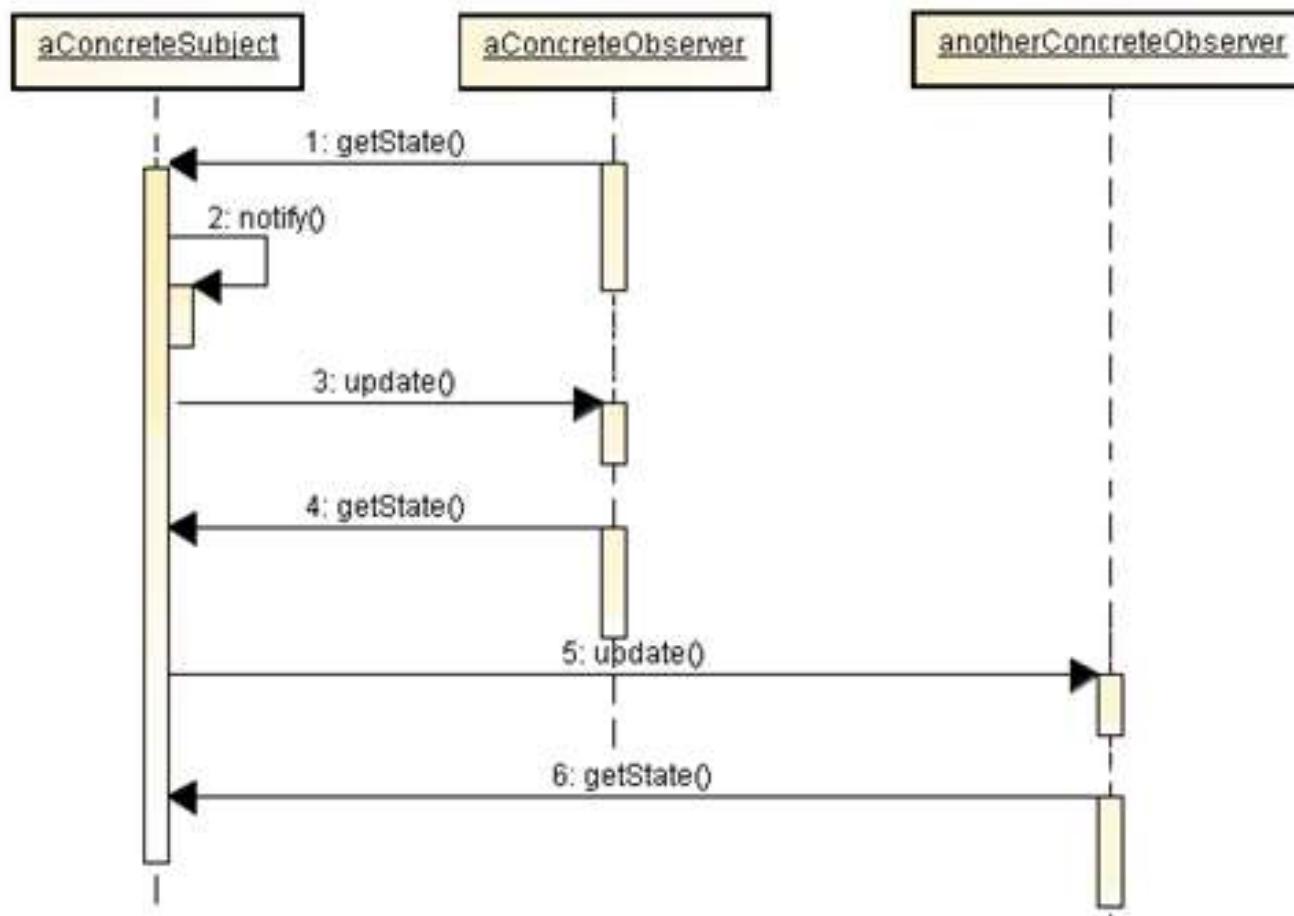
## Diagrama de classes



- 1- Cada sujeito pode ter muitos observadores.
- 2- Os objetos usam a interface subject para se registrarem como observadores.
- 3-Todos os observadores precisam implementar a interface Observer. Esta interface tem apenas um método (update()).

# O PADRÃO OBSERVER

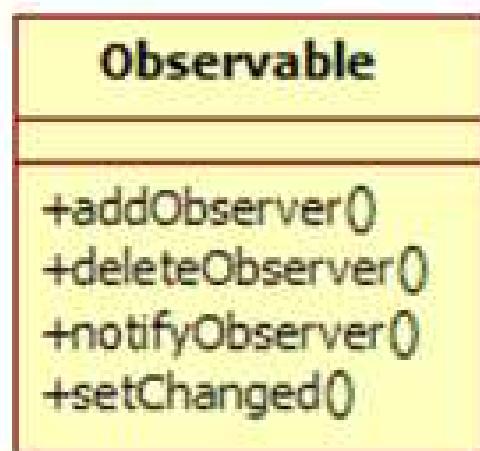
Diagrama de sequência



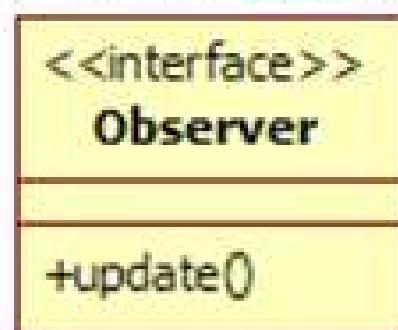
# O PADRÃO OBSERVER

Implementando o padrão Observer interno do JAVA

`java.util.Observable`



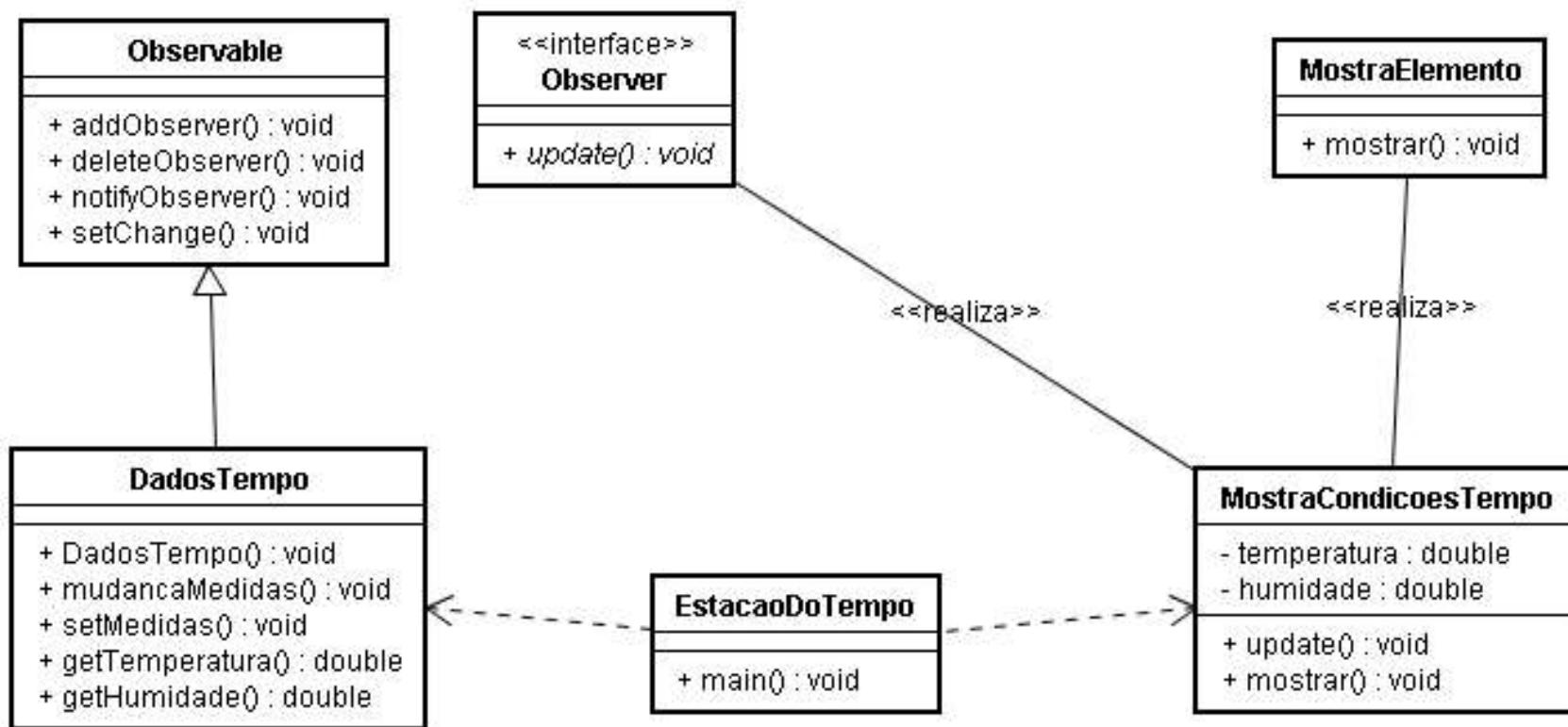
`java.util.Observer`



`c:\prof_gilmar_borba\java\bd\imagens\`

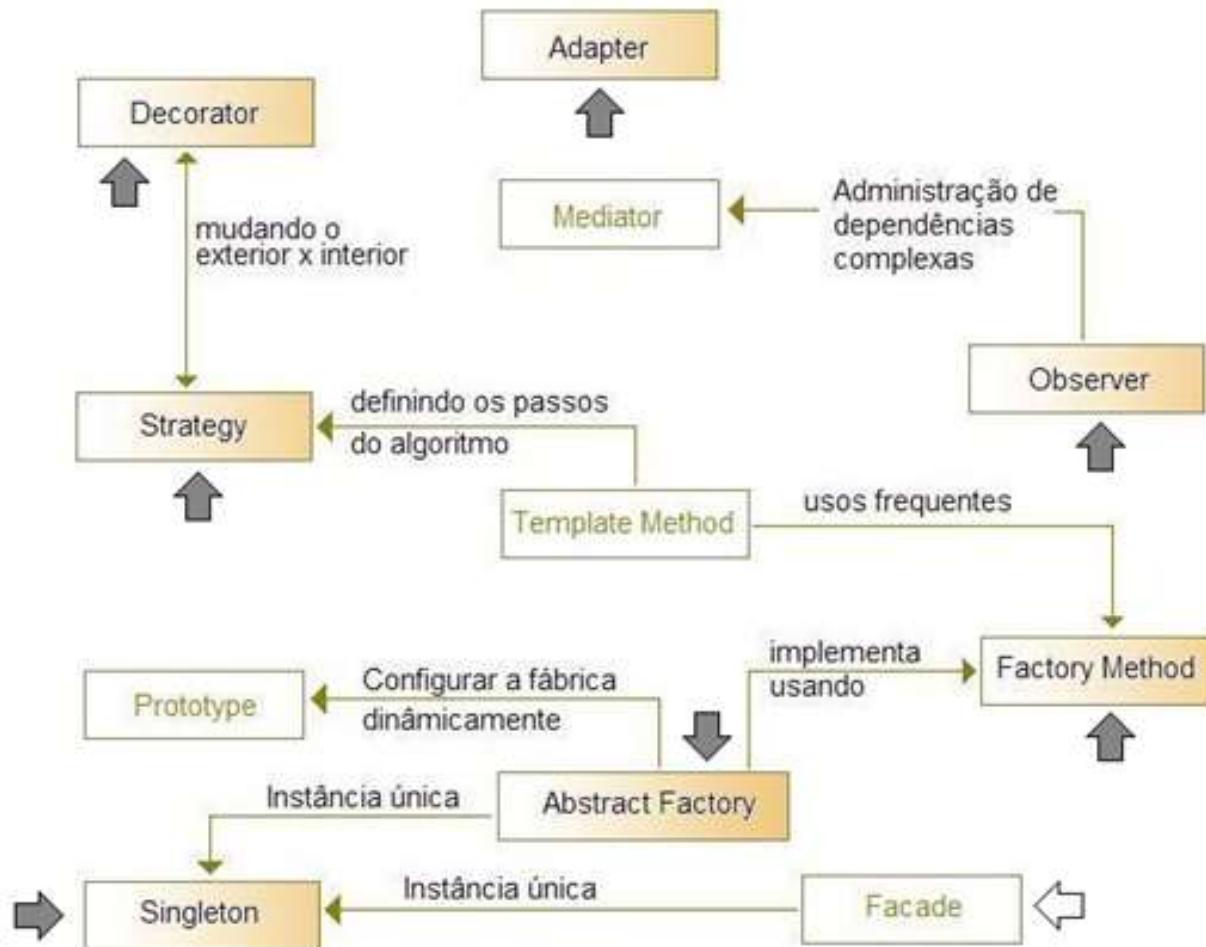
# O PADRÃO OBSERVER

Implementando o padrão *Observer* interno do JAVA (prática)



# O PADRÃO OBSERVER

## Relacionamento entre os padrões discutidos



# O PADRÃO OBSERVER – CONSIDERAÇÕES FINAIS

- Atualmente há mais padrões do que os que foram incluídos no livro da GOF.
- Importante é sempre oferecer recursos de expansão.
- Buscar sempre a simplicidade, não se entusiasme. Se houver uma solução simples que não usa um padrão fique com ela.
- Os padrões não são regras, são ferramentas que podem ser adaptadas ao seu problema.

(FREEMAN, 2007:476)

Discutimos alguns padrões da GOF, consulte também os padrões GRASP (*General Responsibility Assignment Software Patterns*), uma boa fonte de consulta pode ser encontrada em: LARMAN, (2007).

Padrões GRASP: Especialista na Informação, Criador, Coesão Alta, Acoplamento Baixo, Controlador, Polimorfismo, Indireção, Invenção Pura (*Pure Fabrication*) e Variações Protegidas.

# Exercícios



- (70) Qual é a principal desvantagem do padrão *Strategy*?
- (71) Qual princípio de projeto se adéqua à frase: "Quando dois objetos estão levemente ligados, podem interagir, mas sabem muito pouco um do outro"?
- (72) Como as notificações em broadcast ocorrem no processo do padrão *Observer*?

# UML

## Diagramas:

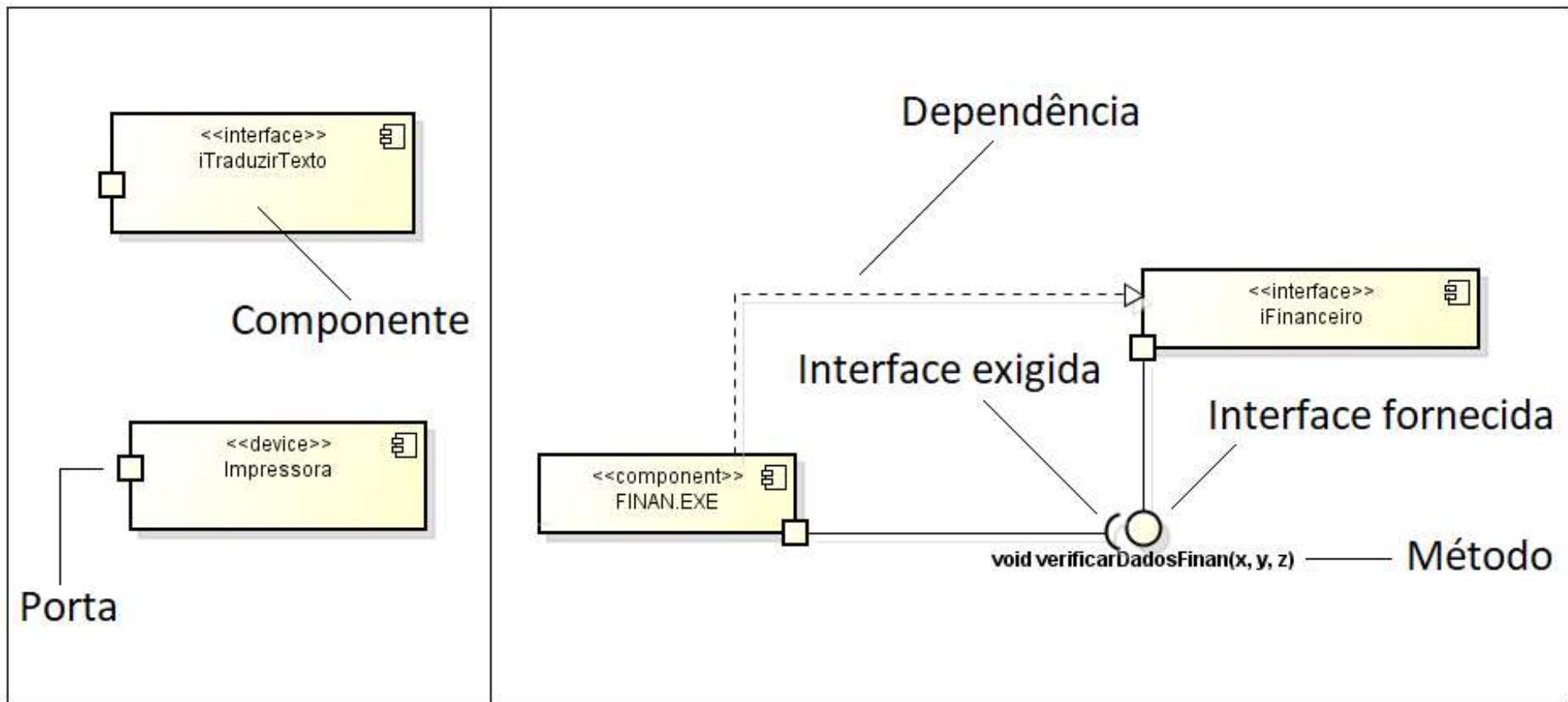
## Pacote / Componentes e Implantação

## Diagrama de Componentes

Representa um módulo físico do código, é o empacotamento físico do código. Os diagramas de componentes representam as dependências entre os componentes de software e os artefatos a eles relacionados (.exe, scripts, dlls etc.)

# Diagrama de Componentes

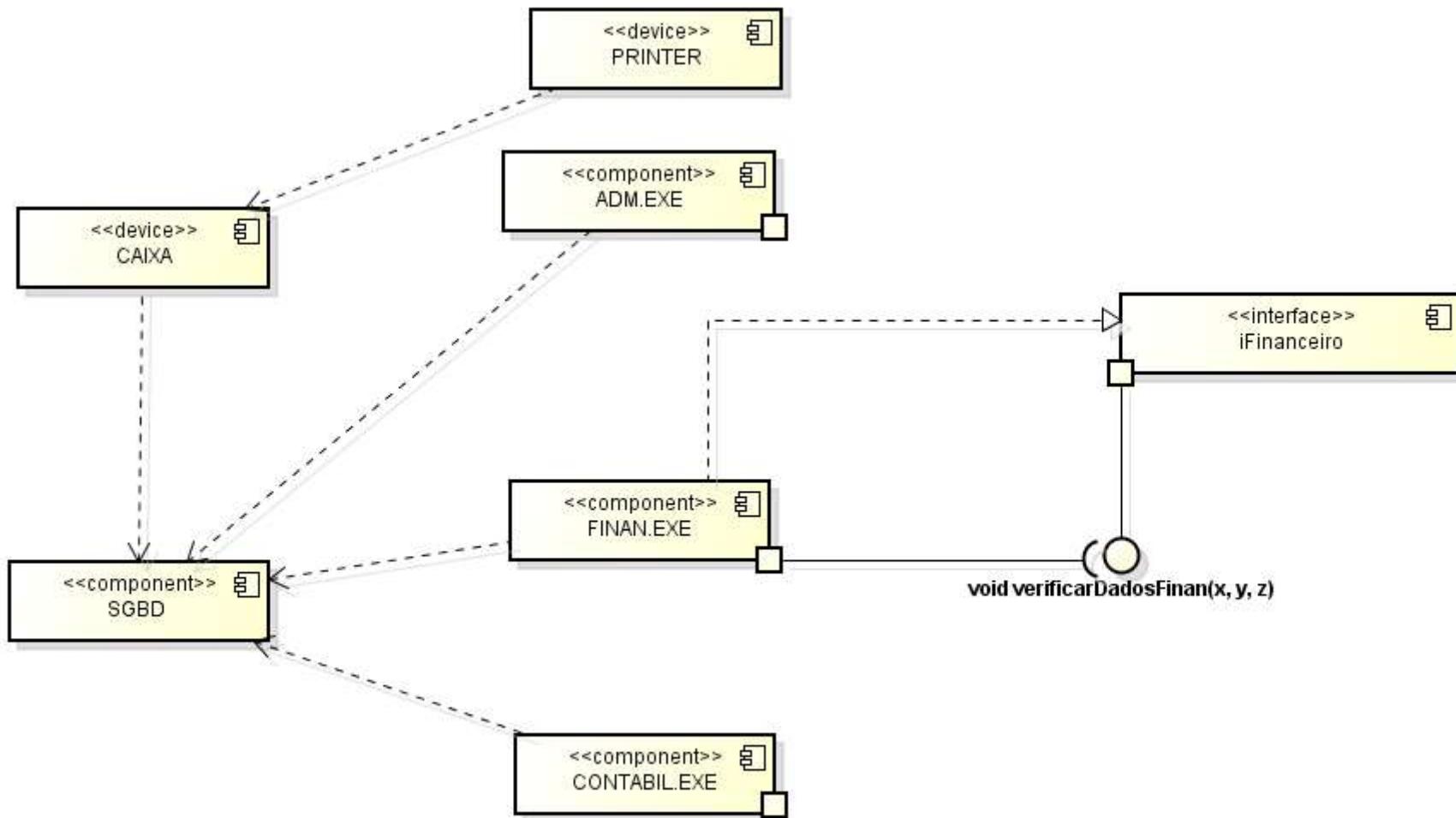
Exemplos de componentes:



c:\prof\_gilmar\_borba\java\bd\imagens\

# Diagrama de Componentes

Exemplo:

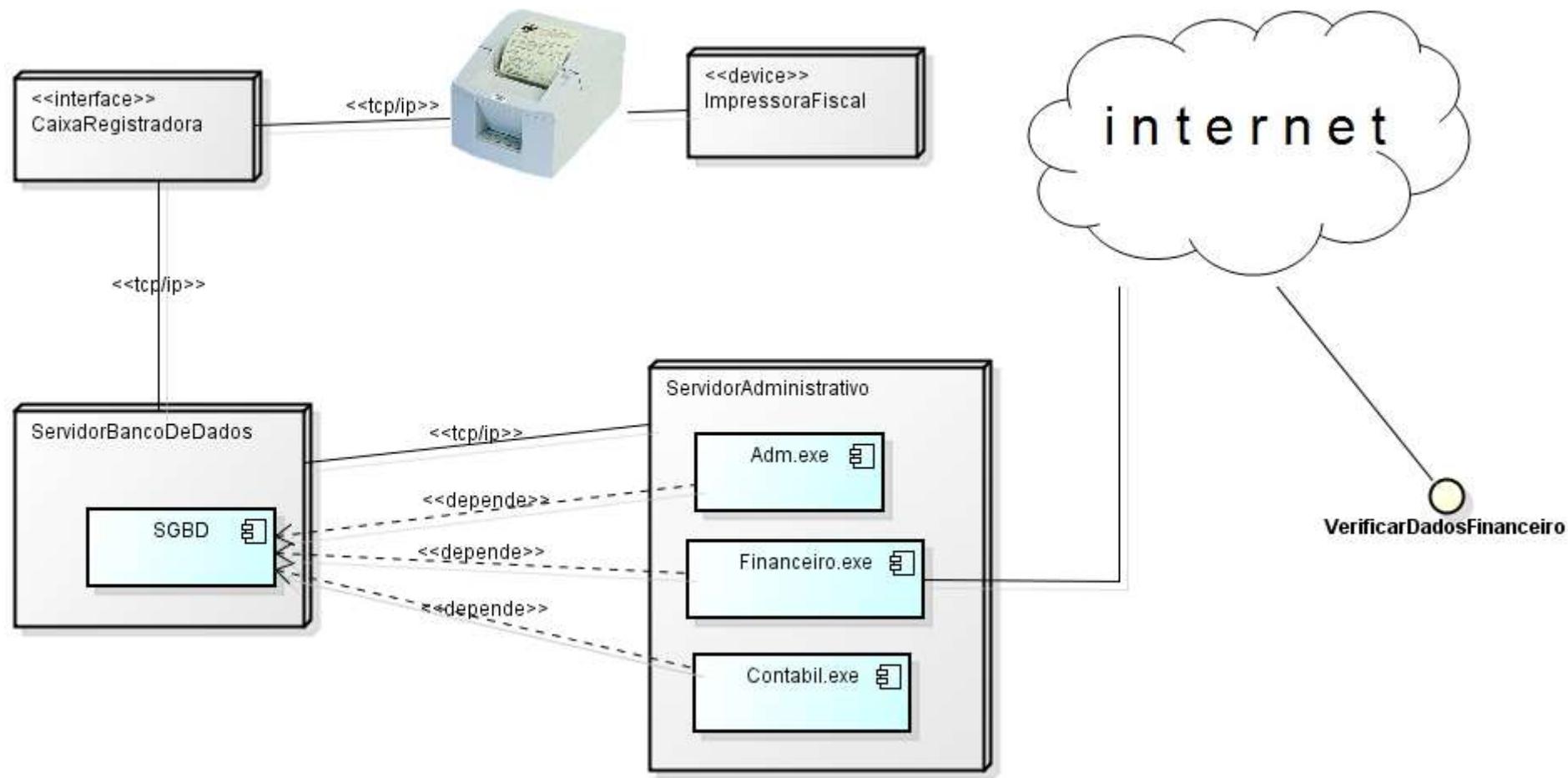


## Diagrama de Implantação

Mostra as relações físicas entre os componentes de software e hardware no sistema. Mostra os acessos e a movimentação dos objetos em um ambiente distribuído. Exemplo:

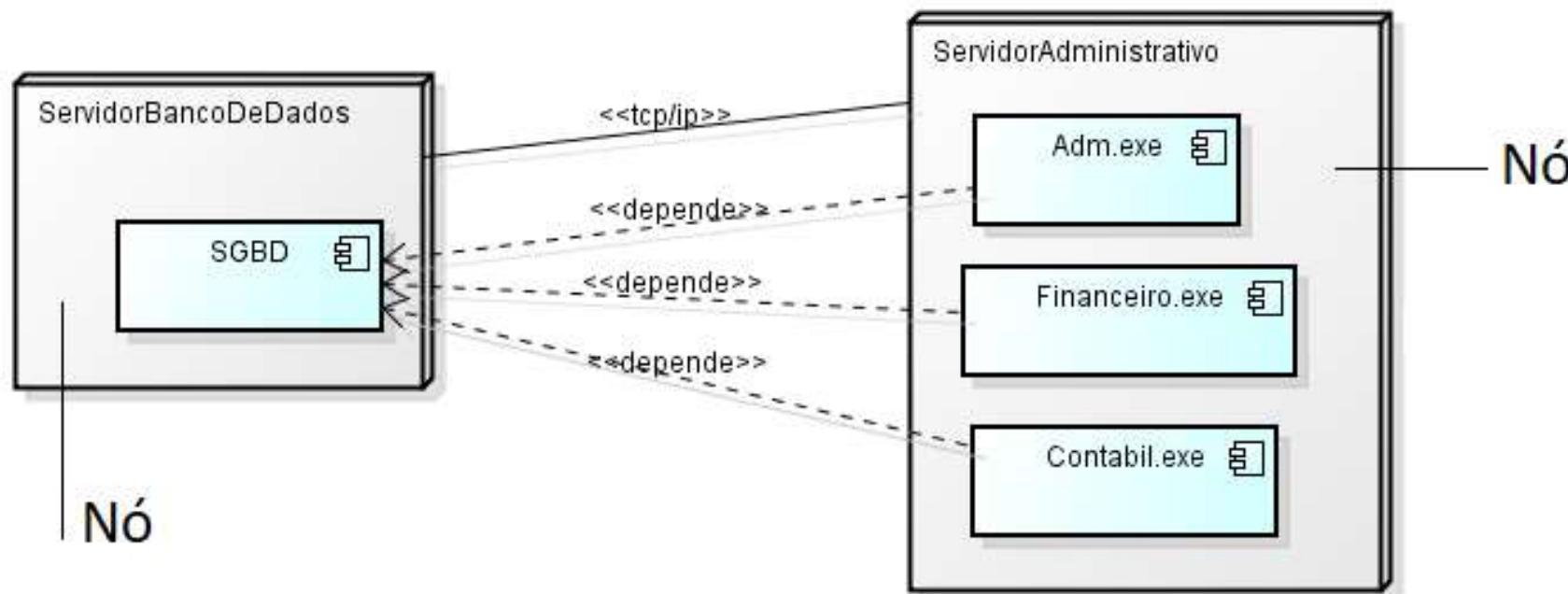
# Diagrama de Implantação

Exemplo:



# Diagrama de Implantação

**Nó** Cada nó neste diagrama representa algum tipo de unidade computacional, na maioria dos casos uma parte do hardware: um sensor, um host etc.



# O DIAGRAMA DE PACOTES

## O que são pacotes?

- São recursos usados para organizar o trabalho.
- É a principal construção na UML para agrupamento de elementos do modelo.
- Pacotes podem conter outros pacotes (hierarquia).

*Possui a mesma funcionalidade de uma pasta no Windows. Organiza e agrupa elementos em um mesmo container.*

# O DIAGRAMA DE PACOTES



**Podem ser inseridos dentro dos pacotes:**

- Elementos possuídos pelo próprio pacote.*
- Elementos importados de outros pacotes.*
- *Elementos acessados de dentro do pacote.*

# O DIAGRAMA DE PACOTES

## Namespaces

Os pacotes oferecem o recurso de namespace.

O *namespace* é um recurso usado para identificar um elemento dentro de uma pacote. Quando inserimos um elemento dentro de um pacote, seu nome se torna exclusivo nesse pacote, mesmo que outra cópia desse mesmo elemento seja alocado em um outro pacote.



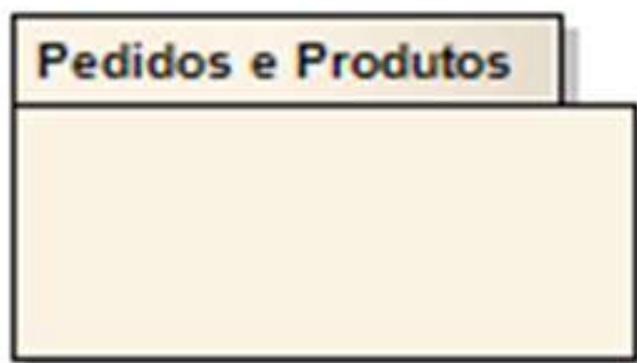
## Notação

*Um pacote é representado por um símbolo de pasta, onde o nome do pacote é inserido na guia superior do símbolo da pasta ou opcionalmente a identificação do pacote pode ser colocada no meio do símbolo.*

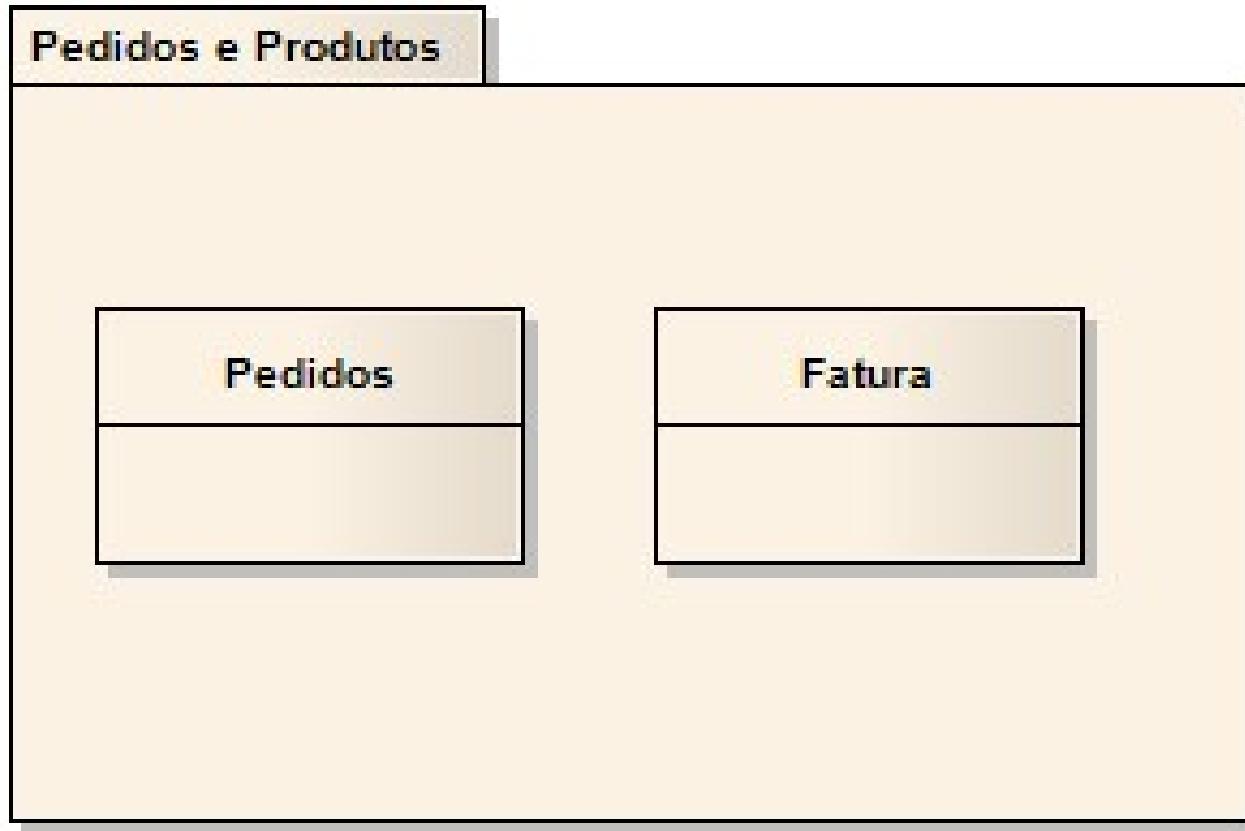
# O DIAGRAMA DE PACOTES

## Notação

Um pacote é representado por um símbolo de pasta, onde o nome do pacote é inserido na guia superior do símbolo da pasta ou opcionalmente a identificação do pacote pode ser colocada no meio do símbolo.



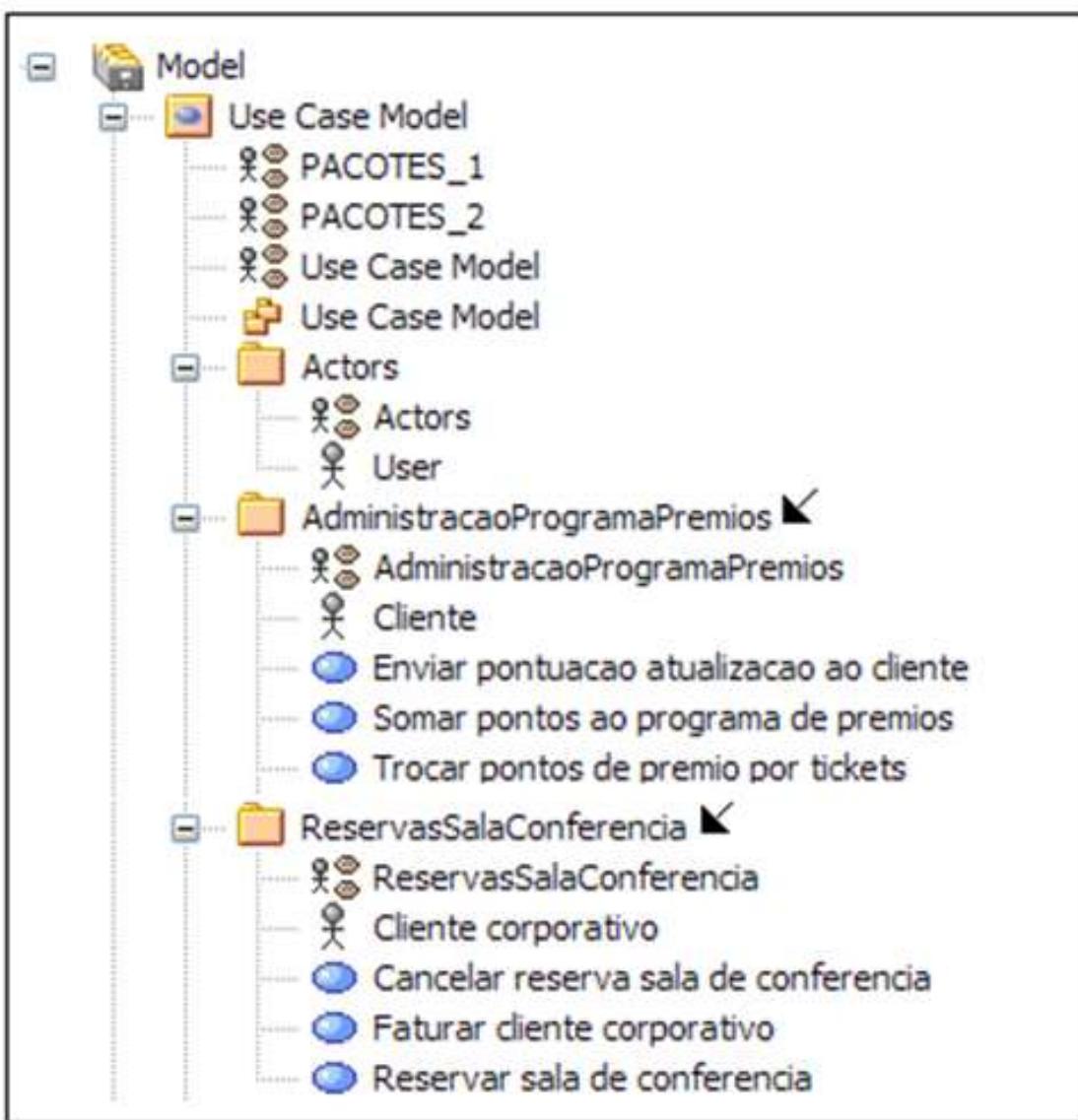
# O DIAGRAMA DE PACOTES



## *Aninhamento de pacotes*

Pode haver outros pacotes dentro de um pacote. O aninhamento é infinito. Cada pacote oferece um namespace exclusivo a todos os elementos dentro dele. Um pacote também pode conter vários diagramas de todo e qualquer tipo UML.

# O DIAGRAMA DE PACOTES



c:\prof\_gilmar\_borba\java\bd\imagens\

# A UML E A ARQUITETURA DE SOFTWARE

O desenho arquitetônico de um software pode ser apresentado em duas visões básicas: lógica e física. A UML oferece recursos para essa finalidade. A decomposição hierárquica do sistema em módulos lógicos, por exemplo, pode ser apresentada no diagrama de **Pacotes** e o diagrama de **Componentes**, esse último também pode apresentar componentes físicos. A especificação da topologia de hardware (dispositivos físicos e conexões) são expressos no diagrama de **Implantação**.

# Exercícios



- (73) Definir UML. Destacar o objetivo principal da UML.
- (74) Quais são as situações usadas para representar o sistema através de um diagrama de componentes?
- (75) Qual é a diferença entre um componente e um artefato?
- (76) No diagrama de componente, o que é uma interface fornecida?
- (77) No diagrama de componente, o que é uma interface exigida?
- (78) Na UML, o que são pacotes?
- (79) O que é um Namespace?
- (80) O que é um diagrama de implantação?
- (81) Qual é a finalidade do diagrama de implantação?

# *Bibliografia*

## *Referências Principais*

Sommerville, Ian . Engenharia de Software, 9a. edição. 2011. Capítulo 6.

Sommerville, Ian . Engenharia de Software, 8a. edição. 2007. Capítulos 11, 12 e 13.

Fowler Martin. Padrões de Arquitetura de aplicações corporativas. – Porto Alegre : Bookman. 2006.

# *Referências Relevantes*

## *GlobalCode*

<http://www.globalcode.com.br/noticias/ApostilaDesignPatterns>

F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996.

## OASIS - SOA

[https://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=soa-rm](https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm)

## OMG - OBJECT MANAGEMENT GROUP

<http://www.omg.org/>

# Referências Relevantes

FREEMAN, Eric; FREEMAN Elisabeth. **Use a Cabeça (head first) Padrões de Projeto.** – Rio de Janeiro : Alta Books, 2007.

GAMMA, Erich et al. **Padrões de Projeto: soluções reutilizáveis de software orientado a objetos.** – Porto Alegre: Bookman, 2000.

HORSTMANN, Cay. **Padrões e Projetos Orientados a Objetos.** – Porto Alegre:Bookman, 2007.

LARMAN, Craig. **Utilizando UML e padrões:uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento interativo.** 3<sup>a</sup>. Ed. Porto Alegre:Bookman, 2007.

MARTIM, Robert C. Princípios, padrões e práticas ágeis em C#. Porto Alegre:Bookman, 2011.

PAGE-JONES, Meilir. Projeto Estruturado de Sistemas. São Paulo:McGraw-Hill, 1998.

**Arquitetura de Sistemas de Informação Análise Comparativa de Modelos.** Tania Fatima Calvi Tait; Roberto C. S. Pacheco; Aline França de Abreu.  
Disponível em: <http://www.scielo.br/pdf/prod/v9n1/v9n1a06.pdf>  
Acessado em: fevereiro de 2019.