

# ANÁLISE DE SISTEMAS

## Orientação a Objetos e UML



Livro Eletrônico

**Presidente:** Gabriel Granjeiro

**Vice-Presidente:** Rodrigo Calado

**Diretor Pedagógico:** Erico Teixeira

**Diretora de Produção Educacional:** Vivian Higashi

**Gerência de Produção de Conteúdo:** Magno Coimbra

**Coordenadora Pedagógica:** Élica Lopes

Todo o material desta apostila (incluindo textos e imagens) está protegido por direitos autorais do Gran Cursos Online. Será proibida toda forma de plágio, cópia, reprodução ou qualquer outra forma de uso, não autorizada expressamente, seja ela onerosa ou não, sujeitando-se o transgressor às penalidades previstas civil e criminalmente.

**CÓDIGO:**

230705433104



**PATRÍCIA QUINTÃO**

Mestre em Engenharia de Sistemas e computação pela COPPE/UFRJ, Especialista em Gerência de Informática e Bacharel em Informática pela UFV. Atualmente é professora no Gran Cursos Online; Analista Legislativo (Área de Governança de TI), na Assembleia Legislativa de MG; Escritora e Personal & Professional Coach. Atua como professora de Cursinhos e Faculdades, na área de Tecnologia da Informação, desde 2008. É membro: da Sociedade Brasileira de Coaching, do PMI, da ISACA, da Comissão de Estudo de Técnicas de Segurança (CE-21:027.00) da ABNT, responsável pela elaboração das normas brasileiras sobre gestão da Segurança da Informação. Autora dos livros: Informática FCC – Questões comentadas e organizadas por assunto, 3ª. edição e 1001 questões comentadas de informática (Cespe/UnB), 2ª. edição, pela Editora Gen/Método. Foi aprovada nos seguintes concursos: Analista Legislativo, na especialidade de Administração de Rede, na Assembleia Legislativa do Estado de MG; Professora titular do Departamento de Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia; Professora substituta do DCC da UFJF; Analista de TI/Suporte, PRODABEL; Analista do Ministério Público MG; Analista de Sistemas, DATAPREV, Segurança da Informação; Analista de Sistemas, INFRAERO; Analista – TIC, PRODEMGE; Analista de Sistemas, Prefeitura de Juiz de Fora; Analista de Sistemas, SERPRO; Analista Judiciário (Informática), TRF 2ª Região RJ/ES, etc.

**GRAN**  
CONCURSOS

O conteúdo deste livro eletrônico é licenciado para gi soares - , vedada, por quaisquer meios e a qualquer título, a sua reprodução, cópia, divulgação ou distribuição, sujeitando-se aos infratores à responsabilização civil e criminal.

# SUMÁRIO

Apresentação .....	5
<b>Orientação a Objetos e UML .....</b>	<b>6</b>
Objetos.....	6
Classes .....	6
Abstração.....	8
Interfaces .....	9
Aspectos do Paradigma de Orientação a Objetos .....	10
Classificação .....	10
Identidade .....	11
Ênfase na Estrutura de Objetos .....	11
Compartilhamento (Herança).....	11
Encapsulamento.....	11
Polimorfismo.....	13
Análise .....	15
Análise Orientada a Objetos .....	15
Modelagem .....	16
Modelagem Orientada a Objetos .....	17
Utilização da UML (Unified Modeling Language – Linguagem Unificada para Modelagem).....	17
Diagramas na UML .....	18
Diagrama de Classes .....	19
Diagrama de Objetos .....	25
Diagrama de Componentes.....	26
Diagrama de Pacotes .....	26
Diagrama de Implantação (Diagrama de Instalação).....	27
Diagrama de Estrutura Composta .....	28
Diagrama de Perfil .....	28

Diagramas de Atividades . . . . .	29
Diagramas de Caso de Uso . . . . .	30
Diagrama de Sequência . . . . .	35
Diagramas de Máquina de Estados (ou Diagrama de Estados) . . . . .	37
Diagrama de Comunicação . . . . .	38
Diagramas de Interação Geral . . . . .	38
Diagramas de Tempo . . . . .	39
A Arquitetura de um Sistema . . . . .	40
Padrões de Projeto (ou Design Patterns) . . . . .	41
Conceitos Básicos dos Padrões de Projeto . . . . .	42
Observação: o Padrão Model-View-Controller . . . . .	43
Princípios SOLID . . . . .	44
Princípios GRASP (General Responsibility Assignment Software Patterns) . . . . .	44
Domain Driven Design (DDD) . . . . .	45
Arquitetura Hexagonal (Portas e Adaptadores) . . . . .	50
<b>Resumo . . . . .</b>	<b>52</b>
<b>Questões Comentadas em Aula . . . . .</b>	<b>56</b>
<b>Questões de Concurso . . . . .</b>	<b>59</b>
<b>Gabarito . . . . .</b>	<b>68</b>
<b>Gabarito Comentado . . . . .</b>	<b>69</b>
<b>Referências . . . . .</b>	<b>93</b>

## APRESENTAÇÃO

Olá, querido (a) amigo(a), tudo bem?



Quer conquistar a sua vitória? Então, tenha uma postura grandiosa! Todos que venceram grandes desafios se alimentavam com pensamentos **positivos, engrandecedores**, de **esperança** e de **coragem**.

Mesmo diante das adversidades, conserve a elegância e uma autoimagem positiva. Essa autoimagem é como se fosse um combustível que impulsionará as suas ações.  
#FicaaDica #OSegredo

Rumo então à aula sobre **Orientação a Objetos, UML, Design patterns, princípios SOLID, princípios GRASP, Domain-Driven Design (DDD), arquitetura hexagonal (portas e adaptadores)** e tópicos relacionados.

Em caso de dúvidas, acesse o fórum do curso ou entre em contato.

Um abraço.

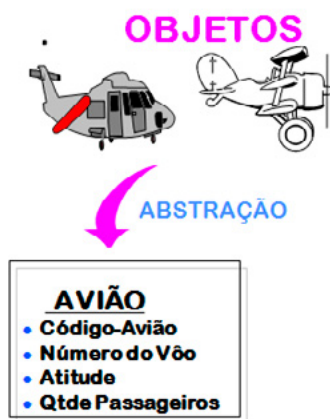
# ORIENTAÇÃO A OBJETOS E UML

## OBJETOS

**Representam elementos do mundo real.** É uma **abstração** de conjunto de coisas do mundo real.

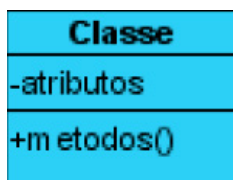
Possuem:

- **Atributos (estado);**
- **Operações (comportamento).**
  - O único acesso aos dados desse objeto é por meio de suas **operações**.



## CLASSES

- Permitem que sejam representados no mundo computacional elementos do mundo real, ou seja, do problema para o qual o software está sendo desenvolvido.
- Elas **permitem descrever um conjunto de objetos** que **compartilhem os mesmos atributos, operações, relacionamentos e semântica**, e representam o principal bloco de construção de um software orientado a objetos.



- Representam os **tipos de objetos existentes no modelo**, e são descritas a partir de seus **atributos, métodos e restrições**.

A figura seguinte apresenta a simbologia para uma **Classe** chamada ContaBancaria, utilizando a **linguagem UML (Unified Modeling Language - Linguagem Unificada para Modelagem)**.



Figura. Classe ContaBancaria

Com as classes definidas, precisam-se especificar quais são seus **ATRIBUTOS** (**propriedades que caracterizam um objeto**). Por exemplo, uma entidade conta bancária possui como atributos o número e o saldo. É bastante simples identificar os atributos de cada classe, basta identificar as **características que descrevam sua classe no domínio do problema em questão**. Cabe destacar que os atributos identificados devem estar alinhados com as necessidades do usuário para o problema.

A próxima figura apresenta a classe ContaBancaria com alguns de seus atributos.

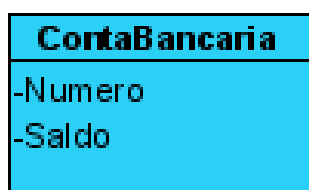


Figura. Classe ContaBancaria com alguns atributos

**Coesão** e **acoplamento** são **dois critérios importantes** para se **analisar** a **qualidade** da **interface pública de uma classe**.

- Sobe a **coesão**, a **interface pública** será considerada **coesa** se todos os seus recursos estiverem relacionados ao **conceito que a classe representa**.
- Já o **acoplamento** é o conceito de uma **classe depender da outra**.

Nesse contexto, deve-se buscar:

- **máxima coesão**; e
- **remoção de acoplamento desnecessário**.

Identificadas as **classes** e seus **atributos**, o próximo passo é a identificação das **OPERAÇÕES** de cada classe, também chamadas de métodos ou **serviços**.

Fazendo um paralelo com objetos do mundo real, **operações** são **ações que o objeto é capaz de efetuar**. Dessa forma, ao procurar por operações, devem-se identificar ações que o objeto de uma classe é responsável por desempenhar dentro do escopo do sistema que será desenvolvido.

A figura seguinte apresenta algumas operações da classe ContaBancaria. Ao contrário dos atributos, normalmente operações são públicas, permitindo sua utilização por outras classes e objetos.

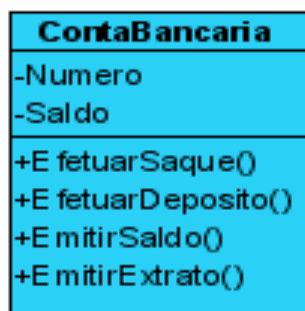


Figura. Classe ContaBancaria com seus atributos e operações

## DIRETO DO CONCURSO

**001.** (FCC/MPE-PB/ANALISTA DE SISTEMAS/2015) Apresenta um conceito correto associado à Análise e Projeto Orientado a Objetos (OO):

Objeto é uma descrição generalizada que descreve uma coleção de métodos semelhantes e encapsula dados e abstrações procedurais necessárias para descrever alguma classe do mundo real.



A assertiva destaca o conceito de **classe** e não de objeto!

As **classes** são **definições** de **atributos** e **funções** de um **tipo de objetos**. Elas são **coleções de objetos** que:

- podem ser descritos por um **conjunto básico de atributos**;
- possuem **operações semelhantes**;
- estão em uma **mesma semântica**.

Os **objetos** são a **peça-chave** do entendimento do conceito de **POO (Programação Orientada a Objetos)**. Eles são **instâncias das classes**. O **estado** e o **comportamento** de um objeto são definidos pela sua **classe**.

**Errado.**

## ABSTRAÇÃO

**Abstração** é o **processo mental de separar um ou mais elementos de uma totalidade complexa**, de forma a facilitar a sua compreensão. No nosso dia a dia utilizamos a abstração para poder trabalhar com toda a informação que o mundo nos fornece.



Rumbaugh destaca **abstração** como uma **habilidade mental que permite aos seres humanos visualizarem os problemas do mundo real com vários graus de detalhe**, dependendo do contexto corrente do problema.

Em outras palavras, **abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais**.

Em uma modelagem orientada a objetos, cabe destacar que uma **classe** é uma **abstração de entidades existentes no domínio do sistema de software**.

Um **objeto** é uma **ocorrência específica (instância) de uma classe**.

Como exemplo, imagine a abstração referente à classe **Animais**. Há várias entidades na classe Animais como Anfíbios, Répteis e Mamíferos que são também sub-classes da classe Animais, onde há **objetos** que contêm cada sub-classe como Ser humano, Jacaré e outros.

## INTERFACES

As **interfaces não** são **classes**.

Podem ser consideradas como **um conjunto de requisitos para que classes possam se adequar a ela**: se a classe estiver em conformidade com uma interface, então um determinado serviço será realizado.

As **interfaces** são como um **contrato** ou **padrão** que **descreve O QUE as classes devem fazer**, **sem** especificar **COMO** devem fazer.

## DIRETO DO CONCURSO



**002.** (CESPE/BANCO DA AMAZÔNIA/TÉCNICO CIENTÍFICO/2010) Considerando os aspectos de linguagem de programação, julgue os itens subsequentes. A abordagem embasada em objetos preocupa-se primeiro em identificar os objetos contidos no domínio da aplicação e, em seguida, em estabelecer os procedimentos relativos a eles.



A questão afirma que na **orientação a objetos** são identificados primeiro os **componentes da aplicação (classes e objetos)** e depois os **métodos**.

Como os **métodos** são de responsabilidade dos **objetos**, deve-se primeiro identificar os objetos para depois alocarem-se os métodos a eles. Logo a alternativa é verdadeira. **Certo.**

## ASPECTOS DO PARADIGMA DE ORIENTAÇÃO A OBJETOS

A expressão “**orientado a objetos**” significa que o software é organizado como uma coleção de **objetos** separados que incorporam tanto a **estrutura** quanto o **comportamento** dos componentes do sistema.

**Isso é diferente da programação convencional**, em que a estrutura e o comportamento dos dados têm **poucos** vínculos entre si.

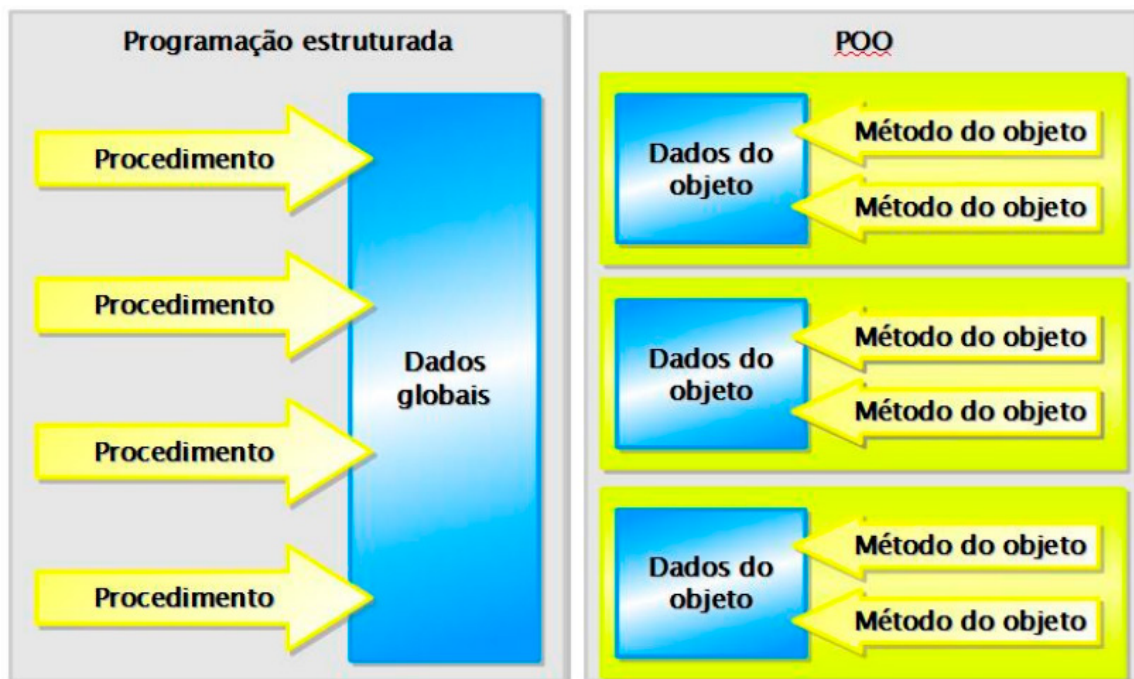


Figura. Programação estrutura x POO (ARAÚJO, R., 2018)

A **orientação a objetos** possui diferentes aspectos que a definem, as quais podem variar de acordo com o autor.

### CLASSIFICAÇÃO

**Classificação** significa que os **objetos** com a mesma estrutura de dados (chamados **atributos**) e o mesmo **comportamento** (chamados **operações** ou **métodos**) são agrupados em uma **classe**.

**Classe** é uma **abstração** que descreve propriedades importantes para uma aplicação e ignora o restante.

A escolha de classes depende da aplicação. Cada **classe descreve um conjunto possivelmente infinito de objetos individuais** e cada **objeto** é uma **instância de sua classe**.

**Uma instância da classe tem seu próprio valor para cada atributo, mas compartilha os nomes de atributos e operações com outras instâncias da mesma classe.**

## IDENTIDADE

**Identidade** é a subdivisão dos dados em entidades discretas e distintas, que são objetos. **Cada objeto tem sua própria identidade**, ou seja, **são distintos mesmo que todos os valores de seus atributos sejam idênticos**, como nome e tamanho, por exemplo.

No mundo real, um objeto limita-se a existir, mas no que se refere à linguagem de programação, **cada objeto possui um ÚNICO indicador**, pelo qual ele pode ser referenciado sem erros.

## ÊNFASE NA ESTRUTURA DE OBJETOS

O **desenvolvimento baseado em objetos** dá maior **ênfase na estrutura de dados** (em comparação com a estrutura de procedimentos) do que as metodologias tradicionais de decomposição funcional.

O desenvolvimento baseado em objetos é similar às técnicas de modelagem de informações utilizadas no projeto de bancos de dados, acrescentando o conceito de comportamento dependente da classe.

## COMPARTILHAMENTO (HERANÇA)

O **compartilhamento (herança)** da estrutura de dados e do seu comportamento permite que a estrutura comum seja compartilhada por diversas subclasses semelhantes, **sem redundâncias**.

O desenvolvimento baseado em objetos não somente permite que as informações sejam compartilhadas como também oferece a possibilidade da **reutilização** de modelos e códigos em projetos futuros.

Uma das principais **vantagens** das **linguagens baseadas em objetos** é o **compartilhamento de código com utilização de herança**, a qual reduz o trabalho de codificação e mostra o conceito proveniente de que as diferentes operações são, na realidade, as mesmas.

## ENCAPSULAMENTO

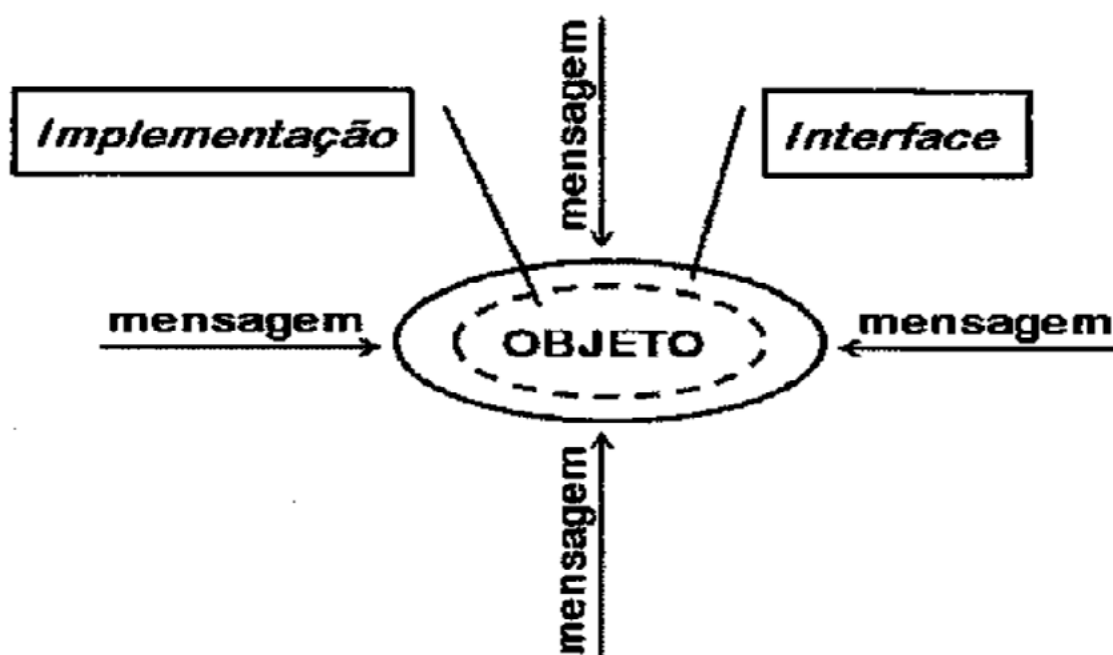
O **encapsulamento**, também conhecido como **ocultamento de informações**, **consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, dos detalhes internos da implementação daquele objeto**.

O encapsulamento impede que um programa se torne tão dependente que uma pequena modificação possa causar grandes efeitos de propagação.

**DIRETO DO CONCURSO**



**003.** (CEPERJ/IPEM-RJ/ANALISTA DE SISTEMAS/2010) No que tange aos paradigmas da Orientação a Objetos (OO), um princípio está diretamente relacionado às operações realizadas por um objeto e ao modo como as operações são executadas, constituindo uma forma de restringir o acesso ao comportamento interno de um objeto. Nesse processo, um objeto que precise da colaboração de outro para realizar alguma tarefa deve enviar uma mensagem a este último. Além disso, separa os aspectos externos de um objeto dos detalhes internos da implementação. Considerando esse contexto, observe a figura abaixo.



O princípio da OO é conhecido por:

- a) Compartilhamento
- b) Acoplamento
- c) Herança
- d) Polimorfismo
- e) Encapsulamento



**Encapsular** é colocar em uma cápsula. Por mais estranho que pareça, pense que você poderia colocar todos os itens desejados em uma cápsula (como em um remédio) e proteger os componentes internos da ação externa.

Em Orientação a Objetos é bem semelhante. A ideia de **encapsular** é **criar uma proteção para os itens do objeto de forma que somente por meio das interfaces criadas possa**

**ocorrer o acesso às estruturas internas.** Isto cria uma proteção para os itens do objeto, que passam a ter o acesso controlado por meio da interface.

**O objetivo é separar os aspectos externos (O QUE faz) dos aspectos internos (COMO faz):**

- Aspectos externos = interface, contrato;
- Aspectos internos = implementação.

O **encapsulamento** pode ser visto como um complemento da abstração:

- A **abstração** foca o comportamento observável de um objeto.
- **Encapsulamento** enfoca a implementação que origina este comportamento.

Além disso, ele – o encapsulamento – promove uma maior estabilidade, uma vez que clientes do objeto só conhecem sua interface e que podemos alterar a implementação de uma operação sem afetar o restante do sistema.

**Letra e.**

---

## POLIMORFISMO

O termo **Polimorfismo** vem do grego e são duas palavras: Poli (muitas) e Morfos (Formas), ou seja, significa “**muitas formas**”. É quando a MESMA operação pode atuar de modos diversos em classes diferentes. Uma operação é uma ação (ou transformação) que um objeto executa (ou a que ele está sujeito). Uma implementação específica de uma operação por uma determinada classe é chamada de **método**.

Pode haver mais de um método para a implementação de um operador baseado em objetos, pois ele é polimórfico.

Ao se chamar uma **operação**, **não** é necessário considerar quantas implementações de uma determinada operação existem. O polimorfismo transfere a responsabilidade da decisão de qual implementação deve ser utilizada da rotina de chamada para a hierarquia de classes.

**Obs.: Polimorfismo** é a característica na qual **os mesmos atributos ou métodos podem ser utilizados por objetos distintos e com implementações distintas.**

DevMedia (2020) cita alguns **exemplos**:

- 1) Podemos assumir que uma bola de futebol e uma camisa da seleção brasileira são artigos esportivos, no entanto o cálculo deles em uma venda é realizado de formas diferentes.
- 2) Podemos dizer que uma classe chamada Vendedor e outra chamada Diretor podem ter como base uma classe chamada Pessoa, com um método chamado CalcularVendas. Se este método (definido na classe base) se comportar de maneira diferente para as chamadas feitas a partir de uma instância de Vendedor e para as chamadas feitas a partir de uma

instância de Diretor, ele será considerado um método polimórfico, ou seja, um método de várias formas.

## DIRETO DO CONCURSO

**004.** (FCC/TRF 3ª REGIÃO/ANALISTA JUDICIÁRIO/2019) O Polimorfismo, um dos Pilares da Programação Orientada a Objetos – POO,

- a) ocorre quando uma classe tem um relacionamento do tipo “1 para” com outra classe e isso implica no modo como a definição das classes devem ocorrer nas aplicações.
- b) consiste em esconder os atributos da classe de quem for utilizá-la. Isso se deve a: 1 – para quem for usar a classe não a use de forma errada; e 2 – para que implementação seja feita por meio dos métodos get e set.
- c) permite que um mesmo método possa ter vários comportamentos e a definição de qual comportamento será executado se dá pelo valor diferente de um de seus atributos.
- d) **é um conceito que permite que as características bem como as operações, de um modo global, possam ser repassadas para várias funcionalidades da aplicação.**
- e) permite utilizar atributos e operações diferentes de uma subclasse, acrescentando ou substituindo características herdadas da classe pai.



Vamos então à análise das assertivas:

- a) Errada. A assertiva está relacionada à **cardinalidade**, **não** especificando o significado de polimorfismo.
- b) Errada. **Encapsulamento** consiste em esconder os atributos da classe de quem for utilizá-la, pois não é necessário que se enxergue o código e sim o que ele faz.
- c) Certa. **Polimorfismo** permite que um mesmo método possa ter vários comportamentos e a definição de qual comportamento será executado se dá pelo valor diferente de um de seus atributos.

DevMedia (2020) destaca **polimorfismo** como um “princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, **embora apresentem a mesma assinatura, comportam-se de maneira diferente para cada uma das classes derivadas**”.

**Trata-se de um mecanismo por meio do qual selecionamos as funcionalidades utilizadas de forma dinâmica por um programa no decorrer de sua execução (DEVMEDIA, 2020).** Com o **Polimorfismo**, os mesmos atributos e objetos podem ser utilizados em objetos distintos, porém, com implementações lógicas diferentes (**DEVMEDIA, 2020**).

d) Errada. **Generalização** é permite que as características bem como as operações, de um modo global, possam ser repassadas para várias funcionalidades da aplicação.

e) Errada. A **Herança** permite utilizar atributos e operações diferentes de uma subclasse, acrescentando ou substituindo características herdadas da classe pai.

**Letra c.**

---

## ANÁLISE

Podemos definir “**análise**” como o processo de decompor o todo em suas partes componentes, examinando estas partes, conhecendo sua natureza, funções e relações.

A tarefa de construir **Sistemas de Informação** é bastante complexa, configurandose, na realidade em um processo de solução de problemas. Neste sentido, dizemos que a análise de sistemas consiste nos métodos e técnicas de avaliação e especificação da solução de problemas, para implementação em algum meio que a suporte, utilizando mecanismos apropriados. É uma etapa no processo de desenvolvimento de software.

Segundo Pressman todos os métodos de análise devem ser capazes de suportar 5 atividades:

- representar e entender o domínio da informação;
- definir as funções que o software deve executar;
- representar o comportamento do software em função dos eventos externos;
- particionar os modelos de informação, função e comportamento de maneira a apresentar os detalhes de forma hierárquica;
- prover a informação essencial em direção à determinação dos detalhes de implementação.

Para que a análise seja bem feita devemos não só entender o que vamos fazer, mas também desenvolver uma representação que permita que outros entendam o que é necessário para que o sistema que está sendo desenvolvido atinja sua finalidade.

A grande maioria dos autores advoga que não devemos levar em conta a tecnologia que empregaremos durante a análise de sistemas.

## ANÁLISE ORIENTADA A OBJETOS

O objetivo da **Análise Orientada a Objetos** é desenvolver uma série de modelos que descrevem **COMO o software irá se “comportar” para satisfazer seus requisitos**. Logo, **a preocupação maior desta fase é representar “O QUE” o sistema irá fazer, sem considerar “como”**.



A análise deve se preocupar com “o que fazer” e nunca com o “como fazer”. Para isso, fazemos a modelagem dos sistemas, utilizando abstrações.

Como na **Análise Estruturada**, constroem-se modelos que representam diversas perspectivas, descrevendo o fluxo de informações, as funções e o comportamento do sistema dentro do contexto dos elementos do sistema.

No entanto, **diferentemente da Análise Estruturada**, aplica-se aqui os conceitos do **paradigma da Orientação a Objetos** para realizar o estudo do sistema, buscando um desenvolvimento mais adequado.

Seguindo os modelos do processo de software, a **Análise Orientada a Objetos** é realizada a partir dos documentos de **Especificação de Requisitos do software**, os quais descrevem, principalmente, as funcionalidades esperadas do sistema.

Nesta fase produz-se um **conjunto de diagramas** que descrevem as características do software. **Deve-se ressaltar que o emprego dos diagramas difere de acordo com as características do sistema**. Podese ainda desenvolver um Dicionário de Dados descrevendo os componentes dos diagramas. Ao final desta fase, se possível, deve-se revisar os documentos juntamente com representantes do cliente.

Os diagramas produzidos nesta fase são modelos gráficos representativos do sistema.

Os diagramas gerados podem (e devem) ser empregados para a prototipação do sistema, podendo-se avaliar se ele cumpre realmente seus requisitos. Na Análise Orientada a Objetos, podem-se empregar diferentes diagramas da UML (**Unified Modeling Language - Linguagem Unificada para Modelagem**), de acordo com a necessidade do projeto.

Os diferentes diagramas permitem ao Analista observar o software sob diferentes aspectos – **Visões**. O emprego de cada diagrama é determinado de acordo com as características do sistema.

## MODELAGEM

Como visto, um modelo é uma **representação abstrata de algo real**. Ele tem o objetivo de imitar a realidade, para possibilitar que esta seja estudada quanto ao seu comportamento. **Os modelos permitem focalizar a atenção nas características importantes do sistema**, dando menos atenção às coisas menos importantes. Seu uso torna o estudo mais barato e seguro: é muito mais rápido e barato construir um modelo do que construir a coisa “real”.

O objetivo do modelo é **mostrar COMO será o sistema**, para permitir sua inspeção e verificação, a fim de poder receber alterações e adaptações antes de ficar pronto. **Qualquer modelo realça certos aspectos do que está sendo modelado, em detrimento dos outros**.



## MODELAGEM ORIENTADA A OBJETOS

A UML (**Unified Modeling Language - Linguagem Unificada para Modelagem**) é o instrumento que permite a modelagem do software “visualmente”, tornando fácil partir dos requisitos do sistema à implementação de uma forma amigável.

## UTILIZAÇÃO DA UML (UNIFIED MODELING LANGUAGE - LINGUAGEM UNIFICADA PARA MODELAGEM)

A **UML** está formalmente em desenvolvimento desde 1994, porém, não é uma ideia que partiu do zero. Trata-se de uma consolidação de renomadas técnicas retiradas de metodologias já existentes e bastante praticadas até então.

Devido ao grande uso de suas metodologias, **James Rumbaugh (OMT)**, **Grady Booch e Ivar Jacobson (OOSE, Objectory)** uniram-se na Rational Software para produzir um padrão que unificasse suas notações. Surgiu então, em 1995, o *Unified Method* que passou a chamarse **Unified Modeling Language**, ao ser lançada a versão 1.0 em 1997.

A **Unified Modeling Language (UML)** é a **linguagem mais utilizada atualmente para especificação e projeto de software na abordagem orientada a objetos**. A UML é o instrumento que permite a modelagem do software “visualmente”, tornando fácil partir dos requisitos do sistema à implementação de uma forma amigável.

Com a adesão maciça de empresas como IBM, Microsoft, HP, Oracle e outras, a UML ganhou status de padrão e foi adotada pelo OMG (*Object Management Group, www.omg.org*) como **linguagem padrão para modelagem de sistema orientada a objetos em 1997**.

O OMG é uma organização mantida por diversas empresas de renome internacional que administra o padrão UML. A função do OMG é organizar e divulgar as especificações da linguagem, controlando as solicitações do que deve ou não ser incluído na arquitetura da linguagem.

Os principais **objetivos** da UML são:

- fornecer aos usuários uma **linguagem de modelagem visual** expressiva e pronta para o uso, visando o desenvolvimento de modelos de produtos de software;
- fornecer mecanismos para apoiar conceitos essenciais;
- ser independente de linguagens de programação e processos de desenvolvimento;
- encorajar o crescimento no número de ferramentas orientadas a objeto no mercado;
- suportar conceitos de desenvolvimento de nível mais elevado tais como colaborações, estrutura de trabalho, padrões e componentes;
- integrar no projeto as melhores práticas de desenvolvimento Orientado a Objetos.

A **UML abrange todas as etapas da produção de software**, mas principalmente é utilizada para traduzir os requerimentos do sistema (em alto nível e mais próximos do

usuário) em componentes codificáveis (mais próximos da aplicação). Mesmo estando entre essas duas camadas, a UML pretende ser fácil de entender para todos os envolvidos. A UML é uma linguagem, e como tal, é um meio de comunicação. Por meio de diagramas gráficos é mais fácil discutir e visualizar as ideias e soluções entre a equipe, ou com o usuário.

A utilização da UML permite visualizar melhor as fronteiras de um sistema e suas funções principais utilizando atores e casos de uso, ilustrar a realização de casos de uso com diagramas de interação, representar a estrutura estática de um sistema utilizando diagramas de classe, modelar o comportamento de objetos com diagramas de estado, e ainda, representar a arquitetura de implementação física com diagramas de componente e de implantação.

A UML é, então, uma **linguagem destinada a visualizar, especificar, construir e documentar sistemas complexos de software**. Estes sistemas podem ser empregados em diferentes áreas, como: telecomunicações, vendas, bancos, Web, entre outros.

## DIAGRAMAS NA UML

Um **diagrama** é a representação gráfica de um conjunto de elementos do sistema sob uma determinada perspectiva, criando uma projeção do sistema. Logo, ele **representa um modelo (representação abstrata) de algo real**.

Com os diagramas, visamos imitar a realidade, para possibilitar que esta seja estudada quanto ao seu comportamento. Esses **modelos permitem focalizar a atenção nas características importantes do sistema**, dando menos atenção às coisas menos importantes. Seu uso torna o estudo mais barato e seguro: é muito mais rápido e barato construir um modelo do que construir a coisa “real”.

A UML propõe os seguintes diagramas:

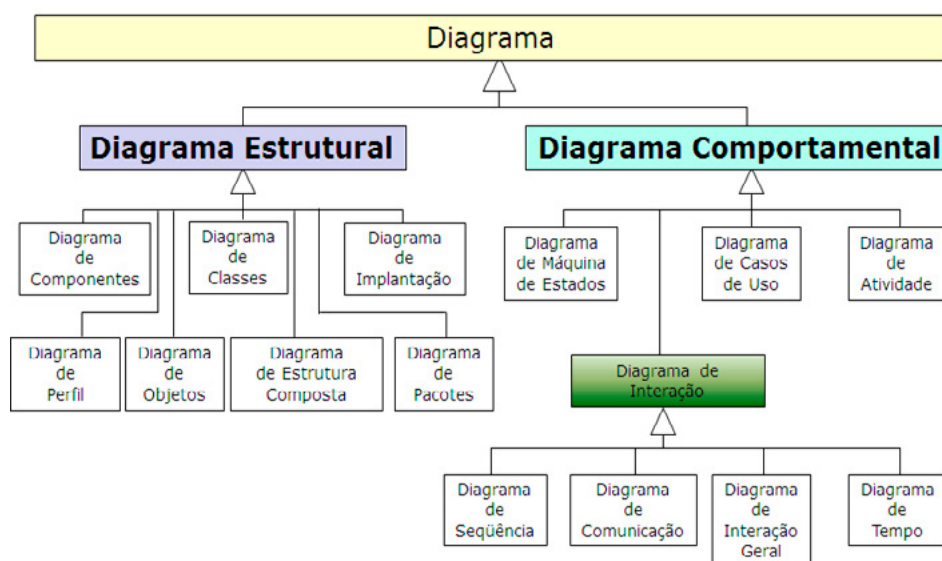


Figura. Especificação 2.2 da UML. Disponível em: <http://www.omg.org/spec/UML/>

Os **diagramas estruturais** representam o **aspecto estático do software sob a perspectiva de diversas abstrações** (Componentes, Classes, Implantação etc.). Tem-se então uma estrutura que **não se altera com o passar do tempo**, ou seja, nos diagramas estruturais não se tem a figura do tempo na representação.

Nos **diagramas comportamentais** tem-se a figura do tempo, porque neles o objetivo é **representar aspectos dinâmicos do software**, ou seja, **como o software interage ao longo do tempo**. Os **diagramas de interação** que são uma subclasse dos diagramas comportamentais visam apresentar a **interação entre objetos dentro do contexto de software**.

Vamos então ao estudo destes **diagramas**.

## DIAGRAMA DE CLASSES

No **Diagrama de Classes** representa-se a estrutura **estática** do sistema. O diagrama é considerado **estático** porque a estrutura descrita é sempre válida em qualquer ponto do ciclo de vida do sistema. Este diagrama é considerado o principal da OOA. Geralmente, a implementação do software é feita baseandose no **Diagrama de Classes**.

**Obs.:** O **Diagrama de Classes** apresenta as classes, suas interfaces, restrições, tipos e características.

O diagrama de classes da UML é o resultado de uma combinação dos diagramas propostos pela OMT, Booch e vários outros métodos. Consiste em uma **estrutura lógica estática** em uma superfície de duas dimensões mostrando uma coleção de elementos declarativos de modelo, como **classes**, **tipos** e seus **respectivos conteúdos e relações**.

### CLASSES

Uma **classe** é a descrição de um grupo de **objetos** com propriedades similares (**atributos**), comportamento comum (**operações ou métodos**), relacionamentos comuns com outros objetos (**associações**) e semânticas idênticas.

**Obs.:** **Classe é a abstração de objetos** com características em comum.

As **classes individuais são representadas na UML como um retângulo sólido com um, dois ou três compartimentos**. O primeiro compartimento é para o **nome da classe** e é obrigatório, o segundo e o terceiro compartimento são opcionais e podem ser usados para listar respectivamente os **atributos** e as **operações** definidas para a classe.

A figura a seguir apresenta uma classe na notação UML.

Nome da Classe
Atributo atributo: tipo do dado atributo: tipo do dado = valor inicial ...
Operação operação (lista de argumentos) ...

**Classes** são componentes do contexto no qual o software está inserido. Analisando-se a Especificação de requisitos, deve-se identificar todos os nomes ou cláusulas nominais. Estes nomes são possíveis classes, que podem ser:

- Entidades Externas;
- “Coisas”;
- Ocorrências;
- Funções;
- Unidades Organizacionais;
- Lugares; e
- Estruturas.

Após identificar as possíveis classes, deve-se **analisá-las para determinar aquelas que realmente são válidas**, de acordo com as seguintes informações:

- Informação Retida;
- Serviços necessários;
- Múltiplos Atributos;
- Atributos Comuns;
- Operações Comuns;
- Requisitos Essenciais.

Os **atributos** descrevem um objeto de uma classe em um certo domínio, contendo informações de estado que precisam ser lembradas. Eles podem também ser atômicos ou agrupamentos de elementos de dados.

**Um atributo é a menor unidade que em si possui significância própria e é inter-relacionada com o conceito lógico da classe à qual pertence.** Ele é uma característica da classe e **não** possui comportamento, pois **não** é objeto. Nomes de atributos são substantivos simples ou frases substantivas, devendo ser únicos dentro da classe. Cada atributo deve ter uma definição clara e concisa.

Cada objeto tem um valor para cada atributo definido dentro de sua classe. Uma assinatura de atributo representa o tipo de informação que será armazenado no atributo.

Para identificar os atributos de uma classe determine os **nomes** da especificação de requisitos que se relacionam às características de objetos.

Pergunta básica: Quais itens de dados (composto e/ou elementares) definem plenamente esse **objeto** no **contexto** em que ele está inserido?

Os **métodos** realizam diferentes **ações** nos objetos. Existem 3 categorias básicas:

- operações que modificam e descrevem dados;
- operações que realizam computação;
- operações de controle.

Para que os **métodos** de uma classe possam ser identificados devem-se analisar as ações executadas por objetos das classes (**verbos**) da especificação de requisitos.

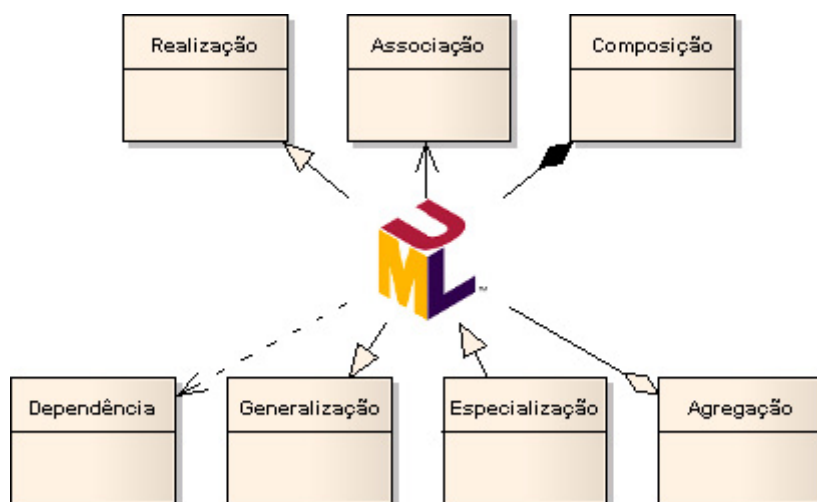
Tem-se na representação de uma classe em UML alguns modificadores (**Público, Protegido, Pacote e Privado**), que têm o objetivo de atribuir o **nível de visibilidade dos atributos e operações de uma classe**.

Modificador	Classe	Subclasse	Pacote	Todos
(+) Público	X	X	X	X
(#) Protegido	X	X		
(~) Pacote	X		X	
(-) Privado	X			

## RELACIONAMENTOS

Os **relacionamentos** **ligam as classes/objetos entre si criando relações lógicas entre estas entidades**.

Os **relacionamentos** tratados em um **diagrama de classes** são:



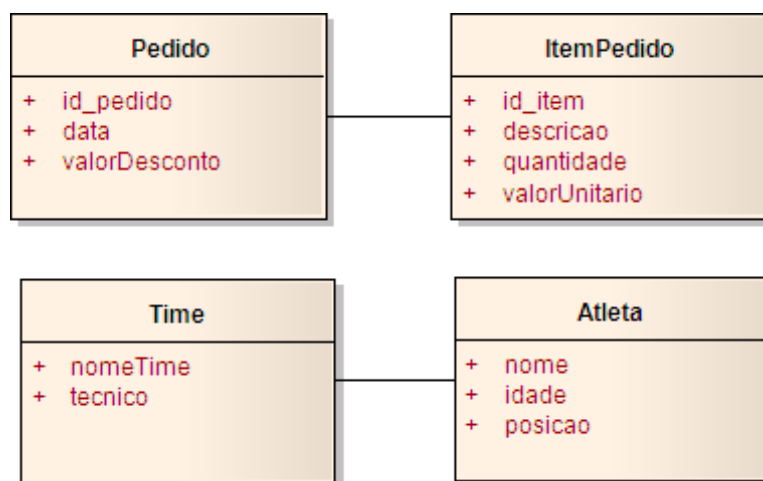
- **Relacionamento de associação:** ocorre **entre objetos de uma classe que se associam a objetos de outra classe**.

**Obs.: Associação:** é uma **conexão entre classes, e também significa que existe uma conexão entre objetos daquelas classes**. Em UML, uma associação é definida com um relacionamento que descreve uma série de ligações, em que a ligação é definida como a semântica entre as duplas de objetos ligados.

**Obs.: Uma associação representa que duas classes possuem uma ligação (link) entre elas**, significando, por exemplo, que elas “conhecem uma a outra”, “estão conectadas com”, “para cada X existe um Y” e assim por diante.

Essa associação pode ocorrer de forma **simples, qualificada, por agregação ou por composição**.

1. A forma **simples** tem um relacionamento mais forte que na **dependência**, ou seja, temos a instanciação de um elemento, **de maneira que a instância de um elemento está ligada à instância de outro elemento**.



**Obs.: O relacionamento de associação** pode apresentar uma ligação **sem** a utilização de setas.

2. Na **associação qualificada** temos um relacionamento de associação simples, mas com um **qualificador** que é um elemento que identifica uma instância.

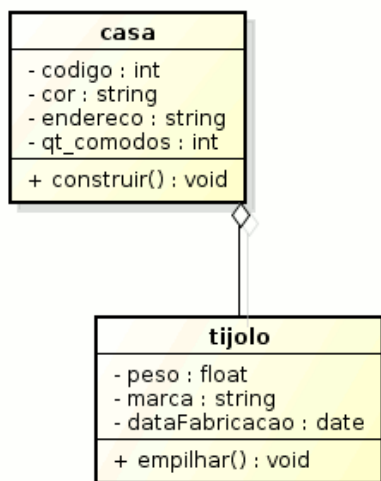


Tem-se aqui uma associação qualificada entre Pedido e item, na qual o qualificador é produto.

3. Na **agregação** nós temos uma associação forte, na qual **o todo está relacionado às partes de forma independente**.

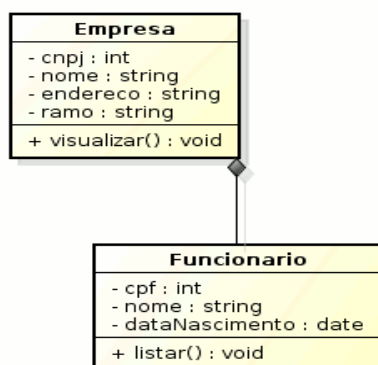
**Obs.: Agregação:** tipo especial de associação, em que um objeto é parte do outro.

Nota: Aqui temos um relacionamento, em que as partes tem existência por si só, logo a parte existe sem o todo.



Aqui tem-se um **relacionamento de agregação** entre casa e tijolo, casa é o todo e tijolo é a parte. Pergunta: o tijolo pode existir sem a casa? Obviamente sim, logo tem-se uma agregação.

4. Temos ainda um tipo de agregação bem mais forte: a **composição**. Na composição tem-se **o todo relacionado com as partes, porém de forma dependente**. Nota: aqui temos um relacionamento, em que as partes **não podem existir por si só**, logo a parte não existe sem o todo.



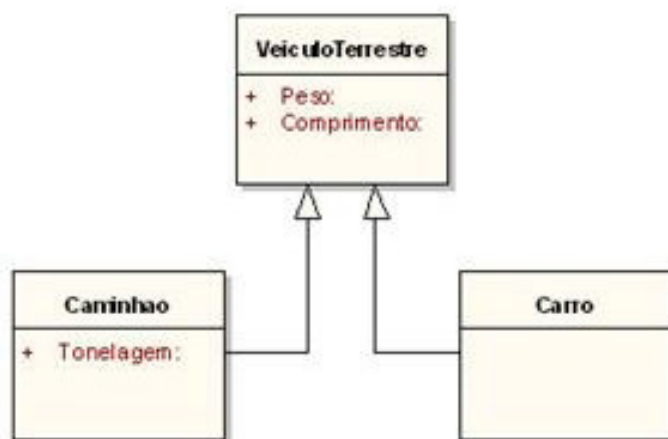
Tem-se aqui o exemplo de composição entre Empresa e Funcionário. Pergunta: O funcionário de uma empresa, pode existir se a empresa não existir? Não, claro que não! O funcionário somente é funcionário, porque a respectiva empresa existe.



Note que na **agregação** temos um relacionamento representado por uma linha com um diamante aberto, enquanto na **composição** temos um diamante fechado.

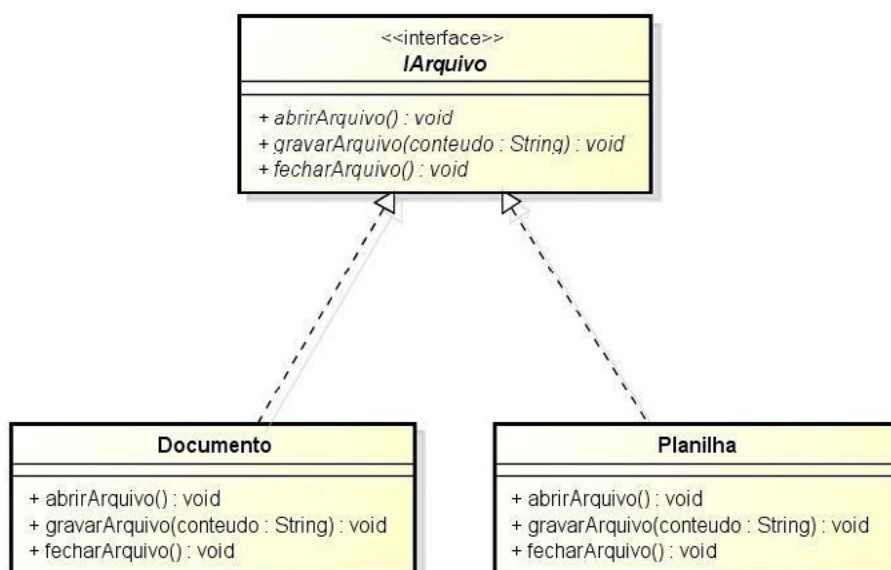
- **Generalização/Especialização:** responsável por **abstrair** o conceito de que **uma subclasse é uma especialização de uma classe e que uma classe é uma generalização de uma subclasse**, sendo que **toda instância da subclasse é uma instância da classe**.

Tem-se aqui o **relacionamento de herança**. Exemplo: Veículo Terrestre é uma generalização de Caminhão. Da mesma forma, Caminhão é uma especialização de Veículo Terrestre.



**Obs.: Generalização:** é um relacionamento de um elemento mais geral e outro mais específico. O elemento mais específico pode conter apenas informações adicionais. Uma instância (um objeto é uma instância de uma classe) do elemento mais específico pode ser usada onde o elemento mais geral seja permitido.

- **Relacionamento de dependência:** ocorre entre **elementos que dependem um do outro**. Exemplo: uma classe depende de um método de outra classe.

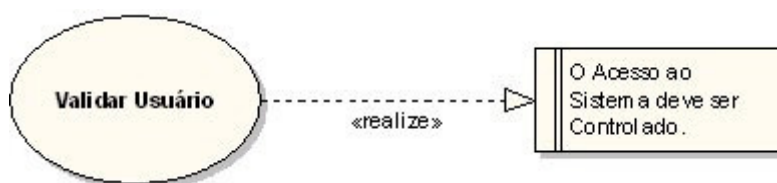




Observe pelo exemplo que a classe **documento** depende da classe **arquivo**, uma mudança por exemplo em `abrirArquivo()` na classe **arquivo**, poderia impactar a classe **documento**.

O relacionamento de dependência é apresentado por uma **seta tracejada** que aponta para uma classe ou interface.

- **Relacionamento de realização:** acontece entre elementos **em que um executa o comportamento especificado pelo outro**. É como se um elemento especificasse o que será executado e o outro elemento executasse.



Todos estes relacionamentos são mostrados no **diagrama de classes** juntamente com suas estruturas internas, que são os **atributos** e **operações**. O diagrama de classes é considerado **estático** já que a estrutura descrita é sempre válida em qualquer ponto do ciclo de vida do sistema.

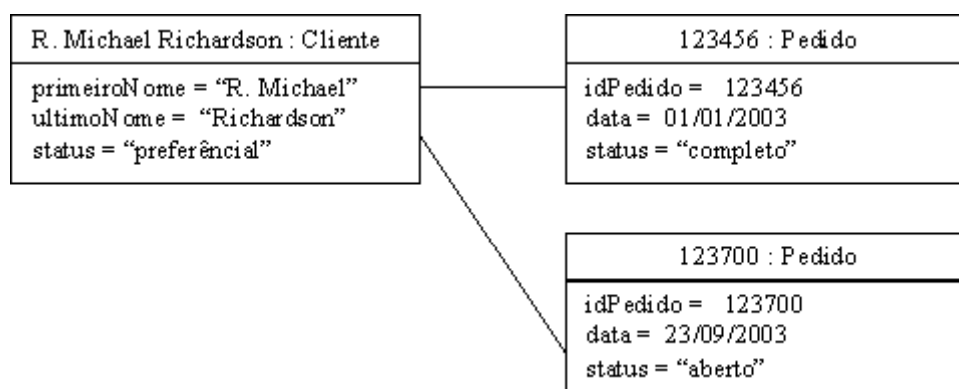
## DIAGRAMA DE OBJETOS

O **diagrama de objetos** (também chamado de **diagrama de instâncias**) é um diagrama estrutural e **nasceu a partir do diagrama de classes**.

Tem-se no diagrama de objetos a **personalização das instâncias e seus respectivos valores**, ou seja, no diagrama de objetos temos uma especificação maior das instâncias, ao invés das classes.

O **diagrama de objetos** é utilizado principalmente para mostrar os objetos e suas respectivas ligações, isso nos ajuda a **entender o domínio do sistema como um todo**.

Veja a seguir um exemplo de diagrama de objetos. Temos aqui o cliente Michael Richardson e dois pedidos, sendo que R. Michael Richardson está associado aos objetos 123456 e 123700, ambos da classe: Pedido.



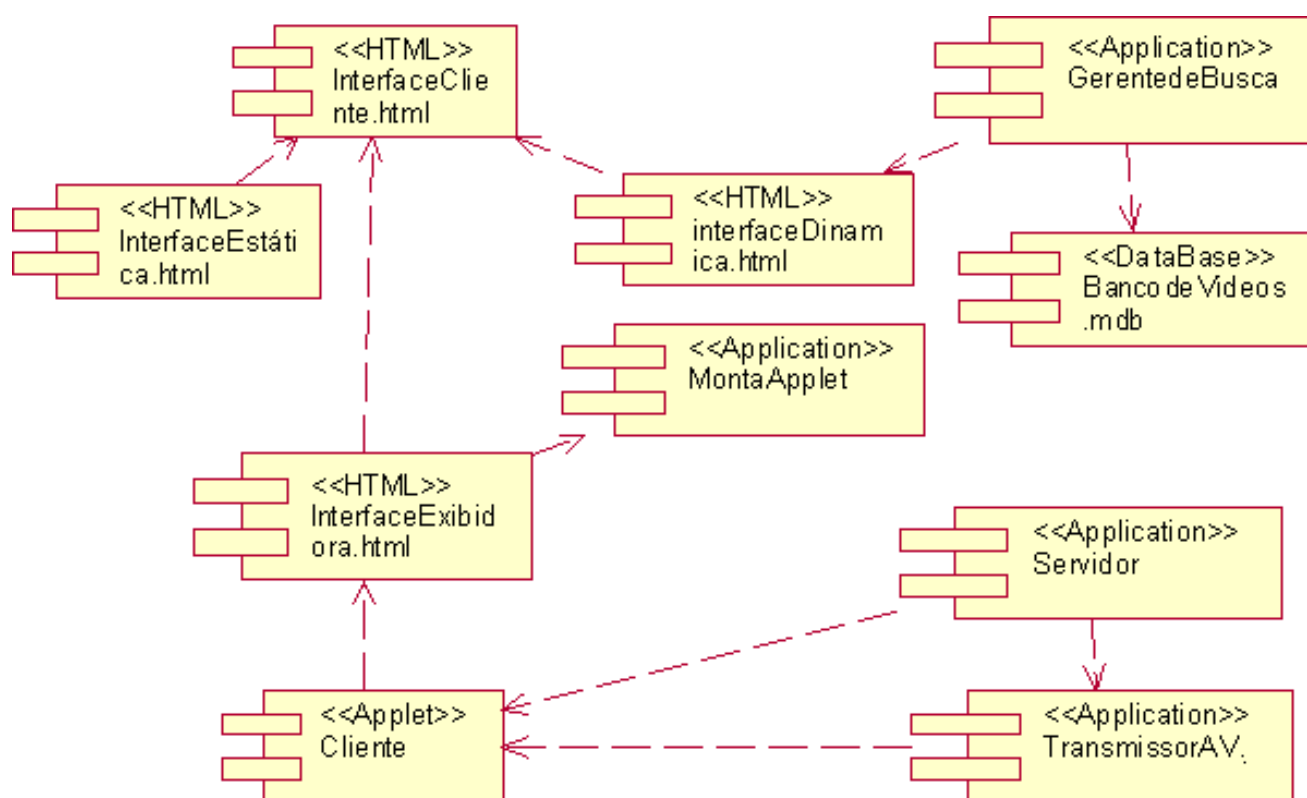
O exemplo anterior mostra um **diagrama de objetos**, responsável por modelar as instâncias das classes contidas no diagrama de classes. Tem-se então um conjunto de objetos e seus vínculos (relacionamentos).

## DIAGRAMA DE COMPONENTES

No **diagrama de componentes** tem-se uma **representação funcional do sistema, através da modelagem dos componentes, iterações e interfaces**.

**Obs.: Componente:** uma unidade independente dentro de um sistema, como um módulo, por exemplo.

Como exemplo, o **diagrama de componentes** destacado a seguir espelha o **sistema de uma locadora**:

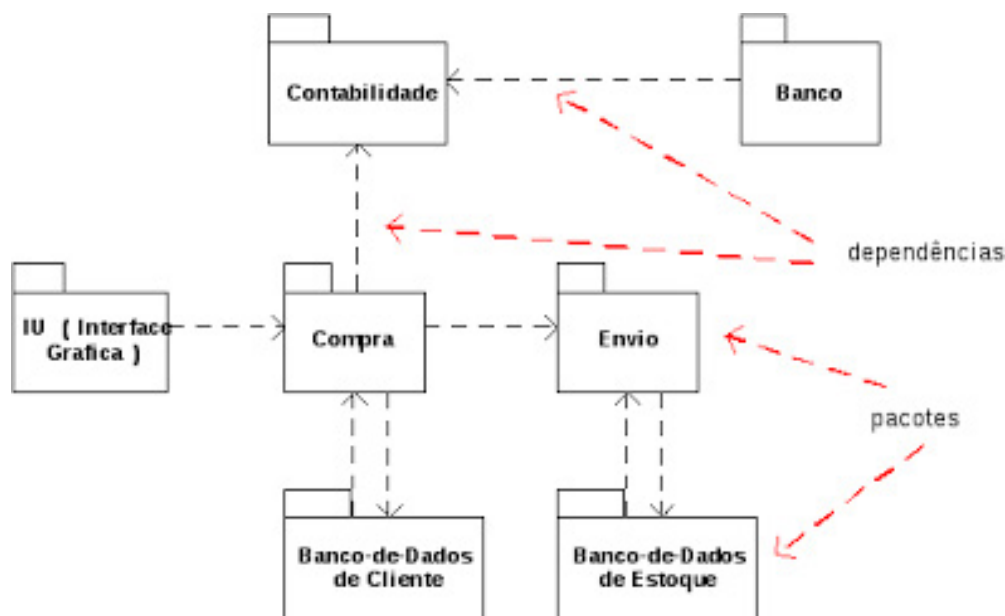


## DIAGRAMA DE PACOTES

Representa a **modelagem dos pacotes e seus relacionamentos**.

**Pacote** é um agrupamento que reúne elementos da UML em unidades de alto nível. Se temos um agrupamento de classes, temos um pacote de classes, por exemplo.

Podemos também ter **subpacotes** que são divisões de pacotes. O diagrama de pacote visa apresentar uma **arquitetura de sistema agrupada por pacote**, conforme o exemplo seguinte:

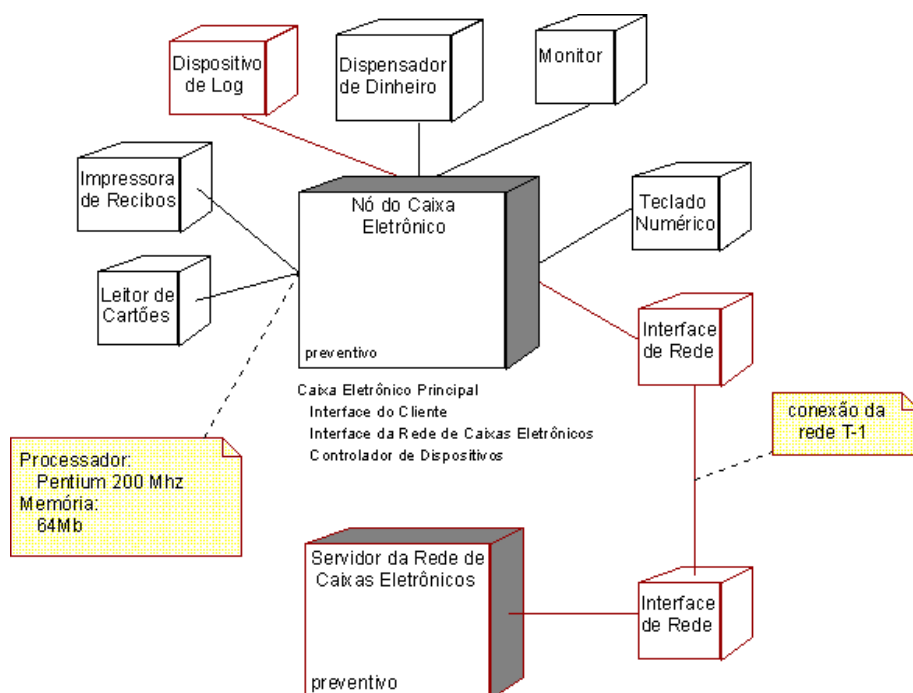


## DIAGRAMA DE IMPLANTAÇÃO (DIAGRAMA DE INSTALAÇÃO)

Modela a parte física do sistema apresentando os componentes de hardware no contexto dado.

O diagrama de implantação também é chamado de diagrama de instalação, tendo em vista que apresenta a estrutura física do sistema.

É bastante utilizado pela equipe de configuração do sistema, tendo-se em vista que nele os aspectos de hardware são considerados.

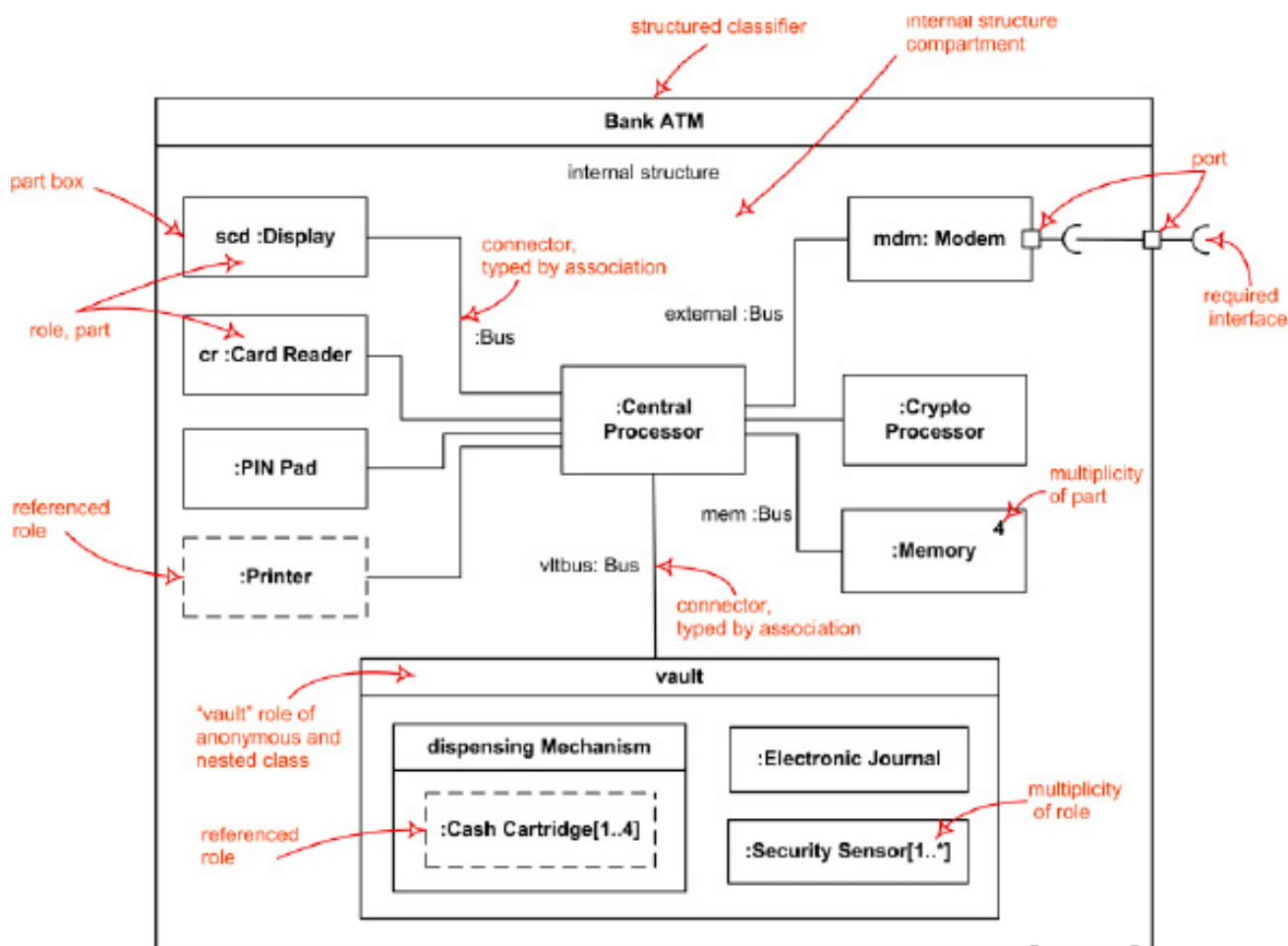


## DIAGRAMA DE ESTRUTURA COMPOSTA

**Modela as colaborações entre as interfaces, componentes e classes por funcionalidade.**

No diagrama de estrutura composta pode-se ver como cada estrutura coopera com as outras estruturas de acordo com as **funcionalidades**.

É bastante utilizado quando precisamos entender o relacionamento estrutural de forma mais detalhada.

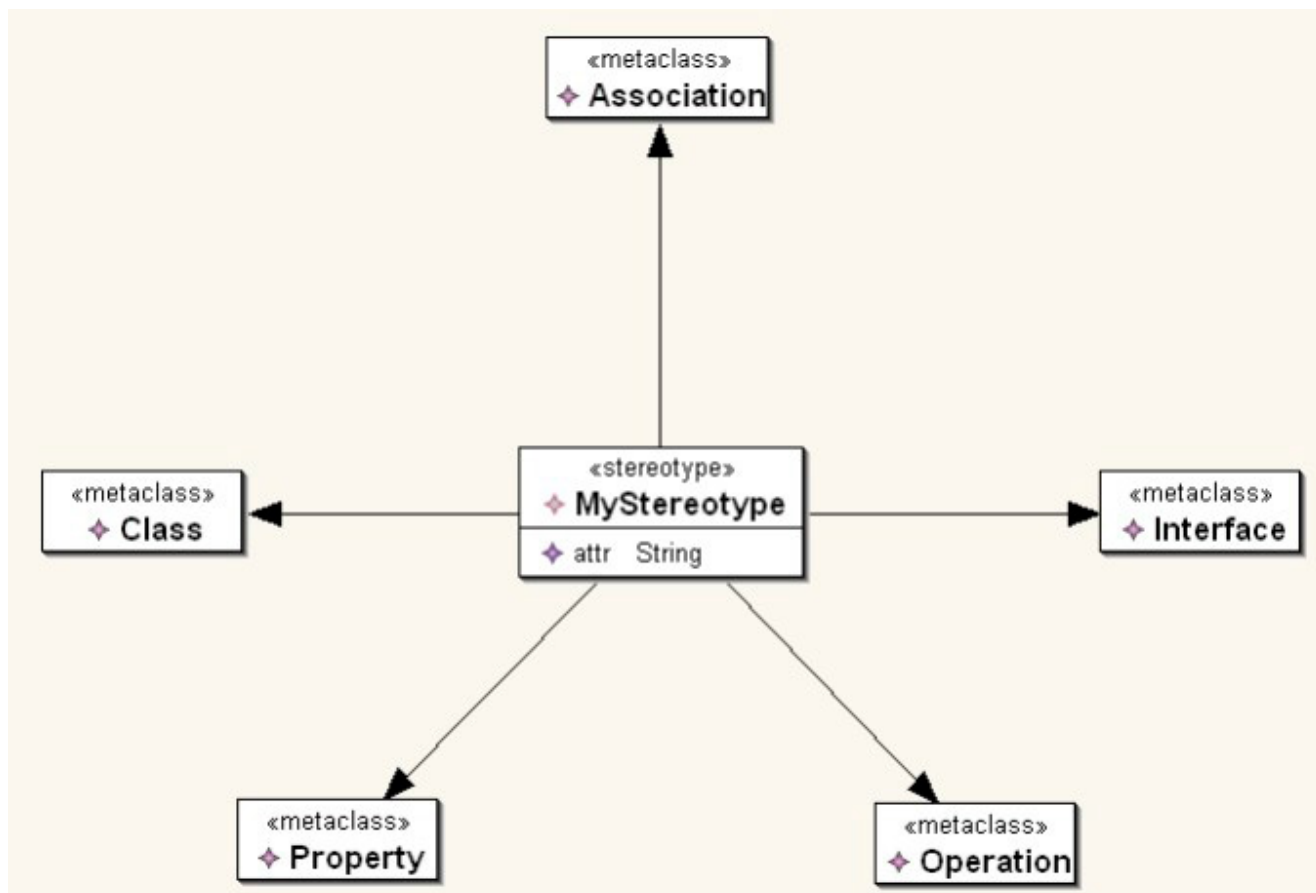


## DIAGRAMA DE PERFIL

Utilizado para **alinhar o conjunto de estereótipos predefinidos com o que precisamos representar enquanto estereótipo.**

Por exemplo, quero apresentar um elemento que representa um DataCenter, logo eu desenho o que digo ser um DataCenter e designo ele como um <<datacenter>>.

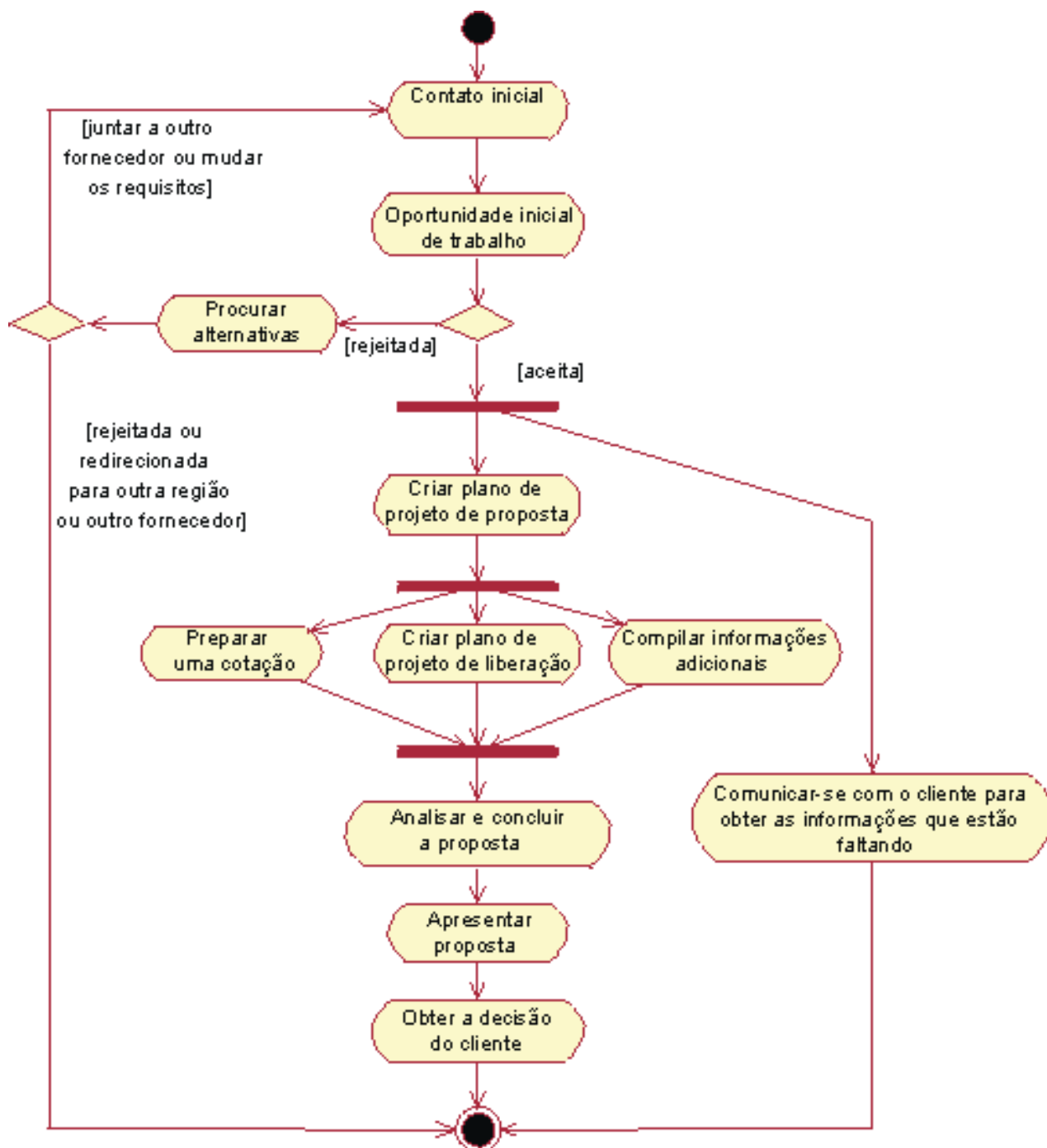
Através do **diagrama de perfil** podemos definir novos elementos na UML, permitindo estender os diagramas existentes através da customização de novas estruturas conforme a necessidade de modelagem.



## DIAGRAMAS DE ATIVIDADES

Modela a **lógica do fluxo de negócio do sistema**. É como se estivéssemos **modelando o fluxo comportamental do sistema**.

Tem-se aqui uma **preocupação com as interações entre os processos de negócio e seus estados** conforme as ações e tomadas de decisão são executadas.



## DIAGRAMAS DE CASO DE USO

Representam um **conjunto de sequências de ações que um sistema desempenha para produzir um resultado observável de valor para um ator específico.**

**Os casos de uso descrevem a funcionalidade do sistema percebida por atores externos.**

Um ator interage com o sistema, podendo ser um usuário, dispositivo ou outro sistema.

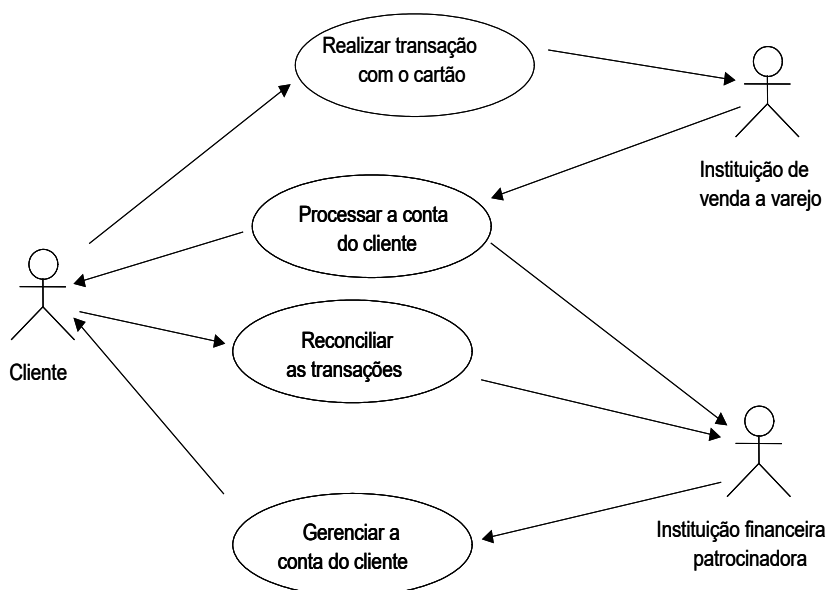
O **Diagrama de Casos de Uso** é empregado visando:

- **Definir Escopo:** um conjunto de Casos de Uso define o escopo do sistema de uma maneira simples.
- **Organizar e dividir o trabalho:** o Caso de Uso é uma importante unidade de organização do trabalho dentro do projeto. A unidade do Caso de Uso divide o trabalho da equipe entre as pessoas, fora isso, é comum dizer que o Caso de Uso está em Análise, em Programação ou em Teste. **Casos de Uso também são entregues separadamente aos usuários em conjuntos divididos em fases ou iterações no projeto.** Então, dizemos que a primeira iteração (ou entrega) terá os seguintes Casos de Uso e na segunda iteração terá os outros.
- **Estimar o tamanho do projeto:** o Caso de Uso fornece métricas para definir o tempo de desenvolvimento.
- **Direcionar os testes:** os testes do sistema (essencialmente os funcionais que são os mais importantes) são derivados do Caso de Uso. A partir dos Casos de Uso, Casos de Teste são criados para validar o funcionamento do software.

Uma das questões em aberto sobre os Casos de Uso é a confusão que fazem com o diagrama e a narrativa (texto) do Caso de Uso. Isso porque a UML define somente como deve ser o Diagrama de Casos de Uso, e não a narrativa. Desse modo, não há um consenso geral sobre como descrever a narrativa, existem muitas técnicas e é difícil julgar que uma técnica é certa e a outra errada, depende muito do projeto, dos seus processos e ferramentas que você tem à disposição.

### Componentes do Diagrama de Casos de Uso

- **Atores;**
- **Casos de Uso;** e
- **Associações** (Inclusão, Extensão e Generalização).



## ATORES

Um **Ator** é um modelo dos possíveis usuários, representando o seu **papel**. Modela uma categoria de usuários do sistema (usuário é uma instância de ator).

Para se identificar os **Atores** de um cenário de um sistema deve-se responder às seguintes perguntas:

- Quem utilizará a principal funcionalidade do sistema (atores principais)?
- Quem provê serviço externo aos sistemas (atores secundários)?
- Quem proverá suporte ao sistema em seu processamento diário?
- Quem ou o quê tem interesse nos resultados produzidos pelo sistema?
- Com quais outros sistemas o sistema irá interagir?

## CASOS DE USO

Um **caso de uso** é uma **situação específica do uso do sistema**, que constitui um curso completo de eventos iniciado por algum ator e/ou especifica a interação entre o ator e o sistema.

Características principais dos casos de uso:

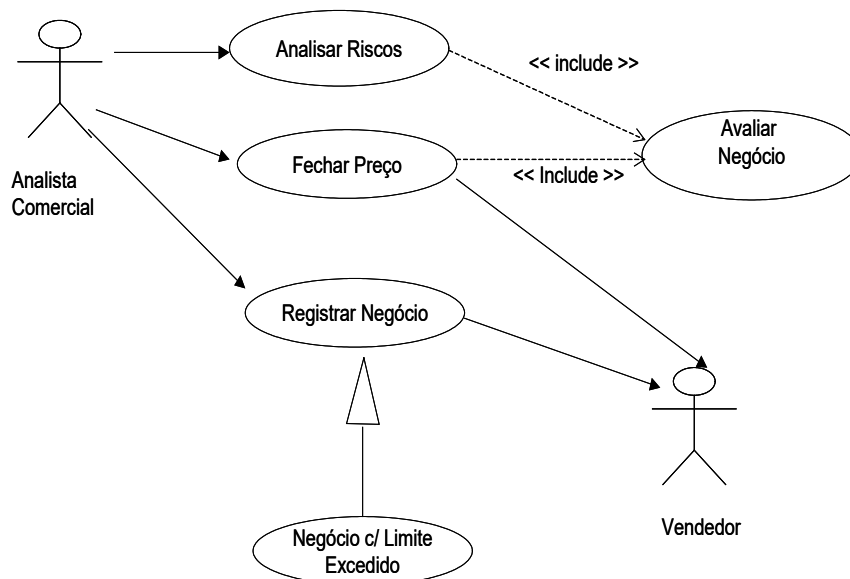
- **Um caso de uso é sempre iniciado por um ator.** Um caso de uso é sempre realizado em nome de um ator que, por sua vez, deve pedir direta ou indiretamente ao sistema tal realização;
- **Um caso de uso é completo.** Um caso de uso deve ser uma descrição completa, portanto, **não** estará completo até que o valor final seja produzido mesmo se várias comunicações ocorrerem durante uma interação; e
- **Um caso de uso provê valor a um ator.** Um caso de uso deve prover um valor tangível a um ator em resposta à sua solicitação.

Para se identificar os Casos de Uso de um cenário de um sistema deve-se analisar as ações do sistema neste cenário, tais como:

- O ator precisa ler, criar, destruir, modificar ou armazenar algum tipo de informação no sistema?
- O trabalho cotidiano do ator pode ser simplificado ou tornado mais eficiente por meio de novas funções no sistema?
- O ator tem de ser notificado sobre eventos no sistema ou ainda notificar o sistema em si? Quais são as funções que o ator necessita do sistema?
- O que o ator necessita fazer?
- Quais são os principais problemas com a implementação atual do sistema?
- Quais são as entradas e as saídas, juntamente com a sua origem e destino, que o sistema requer?



A próxima figura apresenta um exemplo de diagrama de Casos de Uso.



## ASSOCIAÇÕES

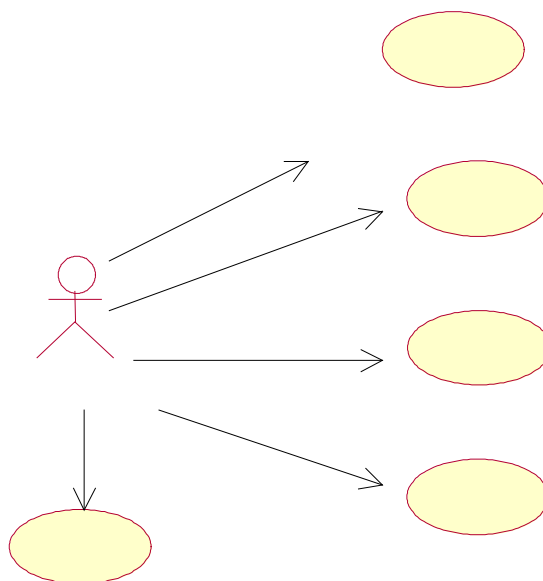
Existem as seguintes **associações** entre casos de uso:

- **Inclusão:** um caso de uso inclui outro, se um passo do mesmo “chama” o outro caso de uso. Pode-se representar também que uma parte do comportamento é semelhante em mais de um caso de uso. No exemplo anterior, tanto “Analisar Riscos” quanto “Fechar Preço” requerem a função “Avaliar Negócio”.
- **Generalização:** um caso de uso é semelhante a outro, mas faz um pouco a mais, apresentando pequenas diferenças. No exemplo, “Negócio com Limite Excedido” realiza uma função a mais que “Registrar Negócio”.
- **Extensão:** descreve situações opcionais, que somente ocorrerão se determinada condição for satisfeita, a qual interrompe a execução do caso de uso. Expressa uma variação do comportamento normal, mas aplicada de forma mais controlada. Pode-se também declarar os “pontos de extensão”. Pode-se também representar serviços assíncronos que o usuário pode ativar para interromper o caso de uso base.

O desenvolvimento dos **Diagramas de Casos de Uso** é baseado na especificação de requisitos e segue as seguintes **atividades**:

- estabelecer o **objetivo** do caso de uso;
- identificar o **limite** do sistema em termos de seus agentes externos (atores);
- identificar **cada ator pelo papel** que representa na interação com o sistema, por exemplo, cliente, gerente, governo, vendedor;
- identificar possíveis **associações** entre casos de uso; e
- **descrever** o caso de uso.

A figura a seguir apresenta mais um exemplo de **Diagrama de Casos de Uso**.



## DIRETO DO CONCURSO



**005.** (CEPERJ/IPEM-RJ/ANALISTA DE SISTEMAS/2010) A UML é uma linguagem de modelagem visual, podendo ser definida como um conjunto de notações e semântica correspondente para representar visualmente uma ou mais perspectivas de um sistema. Dentre os diagramas da UML, um diagrama foca os requisitos funcionais de um sistema, forçando os desenvolvedores a moldarem o sistema de acordo com o usuário, e não o usuário de acordo com o sistema, representando as especificações de uma sequência de interações entre um sistema e os agentes externos que utilizam esse sistema, por meio dos atores e relacionamentos entre eles. Esse diagrama é denominado:

- a) Casos de uso
- b) Casos de negócios
- c) Processos e funções
- d) Entidades e relacionamentos
- e) Processos e relacionamentos



A questão destaca o **diagrama de casos de uso**. Esse diagrama tem **foco nos requisitos funcionais de um sistema**.

Os **casos de uso** descrevem funcionalidades do sistema percebidas por atores externos.

Um **ator** é uma pessoa (ou dispositivo, ou outro sistema) que interage com o sistema.



Letra a.

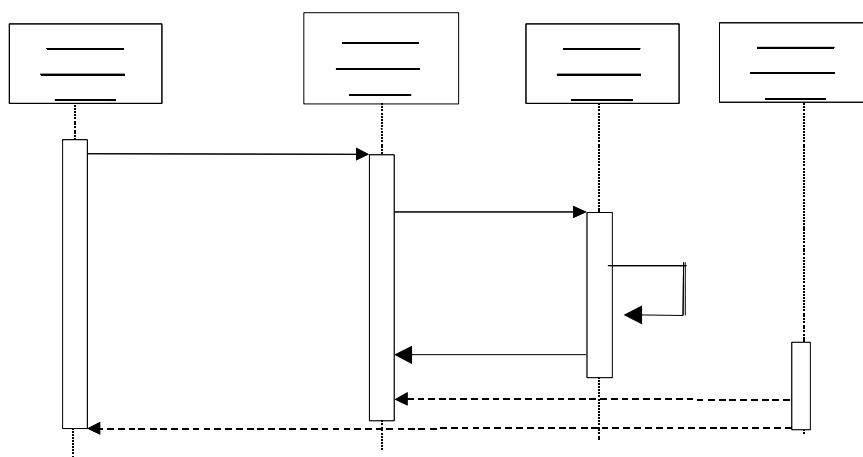
## DIAGRAMA DE SEQUÊNCIA

O **Diagrama de Sequência** **representa as interações do sistema como uma série contínua de mensagens entre os objetos, que colaboram entre si para produzir um resultado esperado**. Este diagrama é utilizado também para mostrar uma instância de um Caso de Uso.

Dentro de um **Diagrama de Sequência**, um objeto é desenhado como um retângulo no topo de uma linha vertical tracejada projetada para baixo, chamada *linha de vida do objeto*.

Cada mensagem é representada por uma linha com seta dirigida horizontalmente entre as linhas de vida dos objetos. A ordem na qual estas mensagens acontecem (fluxo de tempo) é mostrada do topo da página para baixo.

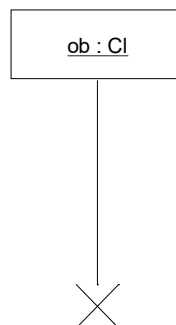
As mensagens são etiquetadas com o seu nome e podem também conter informações como condições para execução, parâmetros e informações de controle. A figura a seguir apresenta um exemplo da notação do Diagrama de Sequência.



## LINHA DE VIDA DO OBJETO

O **ciclo de vida de um objeto é representado por uma linha vertical tracejada**. Se o objeto é criado durante o diagrama, sua identificação é colocada após a mensagem de sua

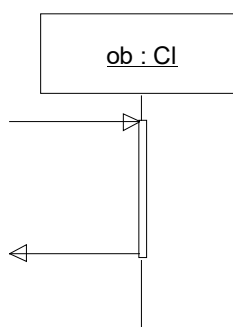
criação. Se um objeto é excluído durante o diagrama, então sua exclusão é marcada com um **X** na mensagem que causa a exclusão ou na mensagem de retorno final do objeto excluído.



## ATIVAÇÃO

**Representa o período em que o objeto está realizando uma ação.** Em fluxos procedurais, o início do símbolo de ativação coincide com a extremidade da mensagem e o final com o início da mensagem de retorno.

Para objetos concorrentes, a ativação representa a duração do período em que cada objeto realiza sua função.



## MENSAGENS

**Os objetos se comunicam por meio de mensagens**, que são setas rotuladas indicando seu emissor, receptor e conteúdo.

A figura a seguir mostra os tipos de mensagem.

**Chamada de Procedimento**



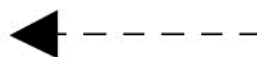
**Autodelegação**



**Mensagem Assíncrona**



**Retorno**



As mensagens podem conter uma condição, indicada pelo uso de colchetes.

## DIAGRAMAS DE MÁQUINA DE ESTADOS (OU DIAGRAMA DE ESTADOS)

Os **Diagramas de Máquinas de Estados**, ou simplesmente **Diagramas de Estados**, **descrevem o comportamento dinâmico do sistema**, **representando os estados que um objeto passa durante a execução do sistema** e **como ele muda de estado em respostas aos eventos**. Cada diagrama representa uma classe única.

Um **diagrama de estado** relaciona os possíveis estados que objetos de uma classe podem ter e quais eventos podem causar a mudança de estado (transição). Estes diagramas podem ser usados também para representar estados concorrentes.

### ESTADO

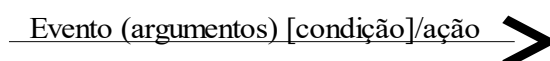
Uma **transição (condição ou interação)** durante a vida de um objeto ativa um estado no qual o objeto realiza uma ação ou espera por um evento. O diagrama começa sempre em um estado inicial e termina em um final.

Um **objeto** permanece em um estado por um tempo finito (não-instantâneo), sendo interrompido pela ocorrência de um evento. Os símbolos apresentados a seguir representam estados na notação da UML.



### TRANSIÇÃO

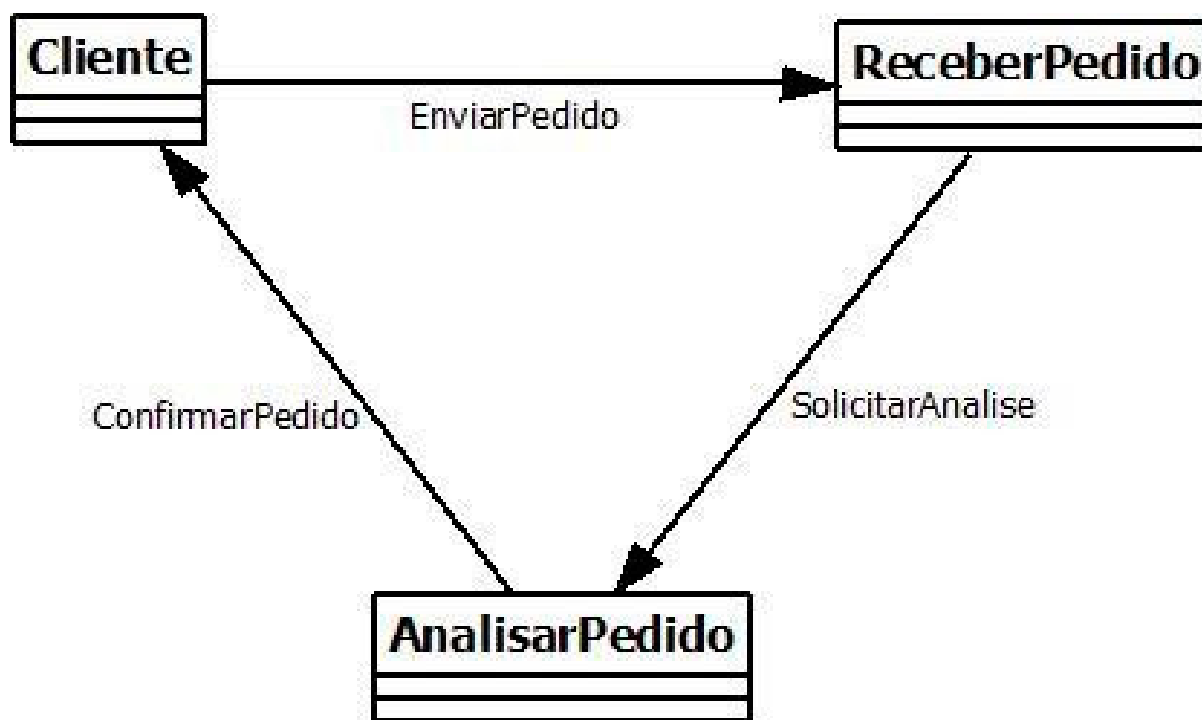
**Indica os eventos e ações necessárias para um objeto mudar de estado.** São representadas por uma seta rotulada da seguinte forma:



**Não** é sempre necessário descrever todos os itens do rótulo.

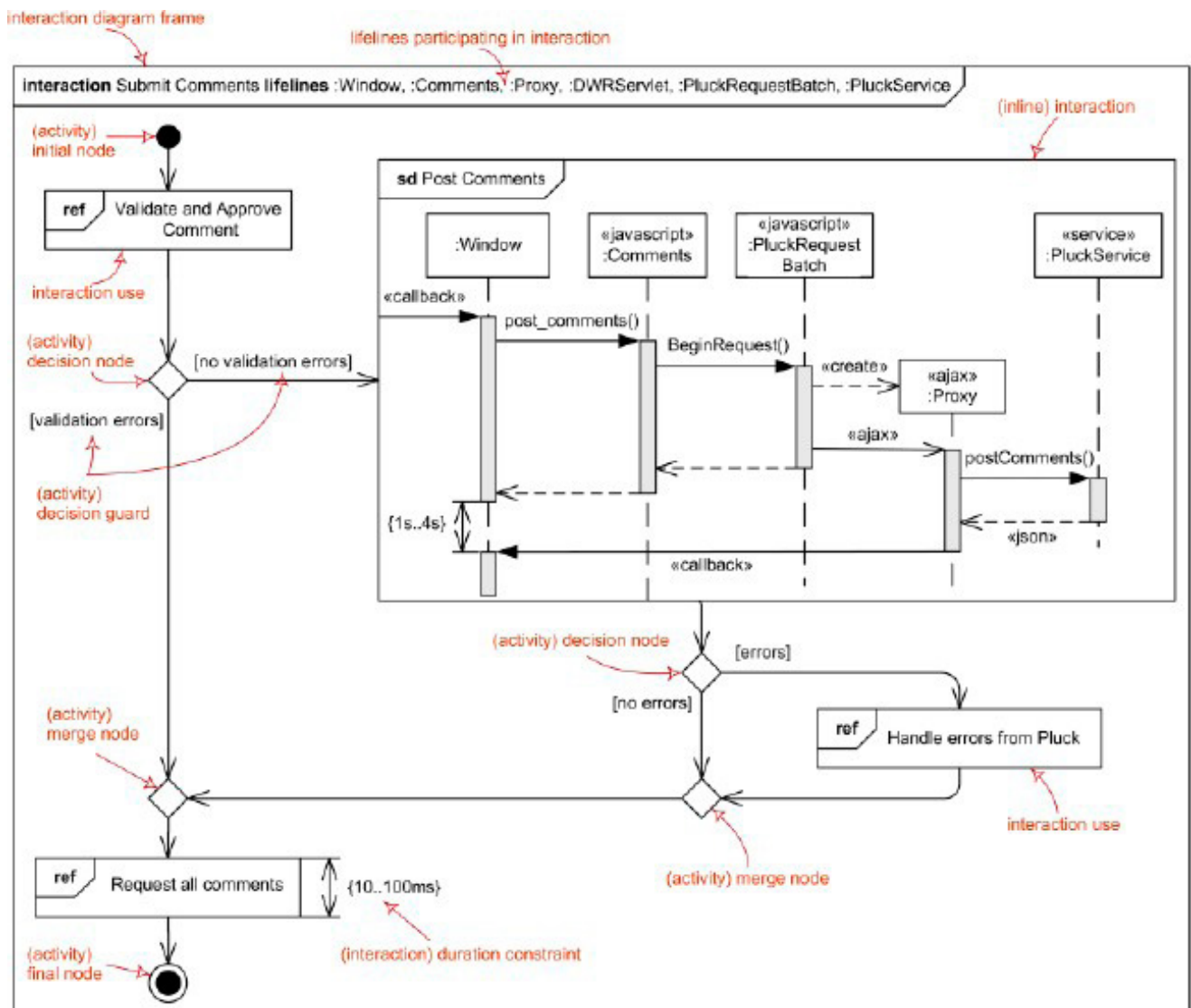
## DIAGRAMA DE COMUNICAÇÃO

É um **diagrama de interação** que lembra o diagrama de sequência, porém nele a estrutura do sistema tem maior ênfase. O **diagrama de comunicação** é chamado por alguns de diagrama de colaboração e normalmente é utilizado quando o contexto a ser modelado é mais simples, quando é mais complexo utilizamos o diagrama de sequência.



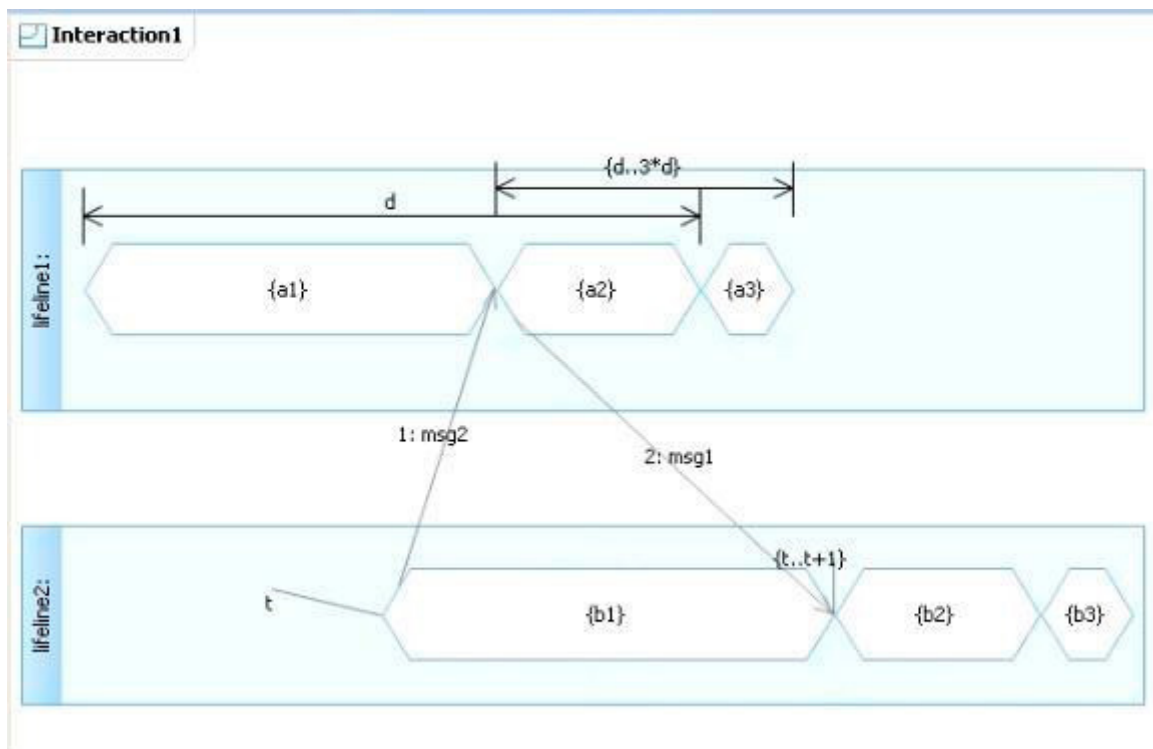
## DIAGRAMAS DE INTERAÇÃO GERAL

É um **diagrama de interação** que **une o diagrama de sequência com o diagrama de atividades**. Nele temos uma **visão geral das interações do sistema**, apresentando a **troca de mensagens entre os objetos e seus respectivos estados ao longo do tempo**.



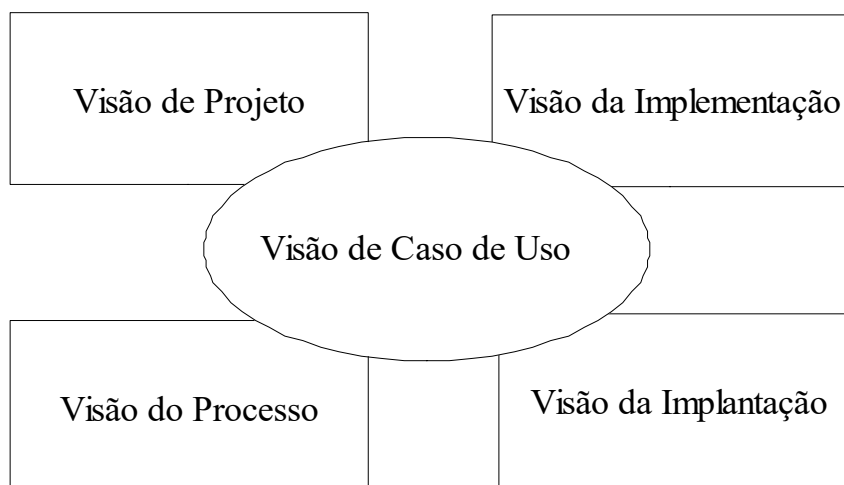
## DIAGRAMAS DE TEMPO

É um **diagrama de interação** que apresenta o **comportamento dos objetos ao longo do tempo** e os **respectivos estados de transição ao longo do tempo**.



## A ARQUITETURA DE UM SISTEMA

A UML permite avaliar um sistema sob diferentes perspectivas, conforme apresentado na figura a seguir.



**a) Visão de Casos de Uso:** descreve a **funcionalidade do sistema desempenhada pelos atores externos do sistema (usuários)**. Apresenta a visão do sistema conforme os usuários e equipe de desenvolvimento. A visão de casos de uso é central, já que seu conteúdo é base do desenvolvimento das outras visões do sistema.



**b) Visão de Projeto:** apresenta **como a funcionalidade do sistema será implementada**, ou seja, os serviços que o sistema irá oferecer aos seus usuários finais. Descreve e especifica a **estrutura estática do sistema (classes, objetos e relacionamentos)** e as **colaborações dinâmicas** quando os objetos enviarem mensagens uns para os outros para realizarem as funções do sistema.

**c) Visão de Interação:** mostra o **fluxo de controle entre várias partes de um sistema**, incluindo concorrência e sincronização. Permite assim uma melhor avaliação do ambiente onde o sistema se encontrará em relação ao desempenho, escalabilidade e *throughput*.

**d) Visão de Implementação:** mostra a **organização física (componentes e artefatos)** utilizados na montagem e fornecimento do sistema e a gerência destes componentes.

**e) Visão de Implantação:** é a **descrição da topologia (nós) em que o sistema será utilizado**, mostrando os computadores e outros periféricos e como eles se conectam.

## PADRÕES DE PROJETO (OU DESIGN PATTERNS)

Os **Padrões de Projeto** oferecem um **meio para nomear e descrever soluções abstratas para uma quantidade de problemas recorrentes**. Eles permitem que os analistas falem a mesma língua, reduzindo a quantidade de tempo gasto rascunhando em um quadro branco e permitindo um melhor desenvolvimento de aplicações.

As **duas palavras** que compõem este método de desenvolvimento de aplicação **padrão e projeto (design)** merecem atenção.

Define-se **padrão** como:

- “uma forma ou modelo proposto para imitação”;
- “um sistema coerente compreensível baseado nas inter-relações pretendidas entre as partes componentes”.

E **projeto (design)** como:

- “planejamento com propósitos determinados”;
- “um projeto mental ou esquema no qual os meios para alcançar um objetivo são estabelecidos”;
- “um rascunho preliminar ou esboço mostrando as principais características de algo que será executado”.

Em outras palavras, um **Padrão de Projeto (Design Pattern ou Padrão de Design)** é uma **solução planejada e repetida que age em um conjunto de problemas**.

Em 1995, com a publicação do livro *Padrões de Design* por Gamma, Helm, Johnson e Vlissides, também conhecido como a **“Gangue dos quatro” (GoF)**, cientistas da computação começaram a discutir os padrões de *design* de programa. Eles aplicaram as ideias atrás dos padrões de Alexander em um livro que detalha vários padrões de programação. O

livro também discute quando estes padrões de programação devem ser usados. **Padrões de Projeto (ou Padrões de Design)** contém um catálogo de padrões para linguagens orientadas a objetos.

## CONCEITOS BÁSICOS DOS PADRÕES DE PROJETO

Uma das chaves do **desenvolvimento orientado ao objeto**, de fato qualquer desenvolvimento, é sua **reutilização**. A maioria do trabalho da reutilização está concentrado na **reutilização do código**.

Os **Padrões de Projeto** permitem que se reutilize os *designs*. Desta forma, pode-se criar uma **interface padrão dos componentes, reutilizando-os quando necessário sempre que uma interface semelhante estiver disponível**. Você finaliza programando a interface de uma classe, componente ou subsistema, ao invés de sua implementação.

As linguagens de programação orientadas a objeto permitem que você herde das **superclasses**, obtendo automaticamente qualquer uma das suas qualidades. Isto é conhecido como **reutilização do tempo de design**, a partir do momento que a árvore de herança é decidida durante o *design* e desenvolvimento.

Os **padrões de design** geralmente concentram-se em outro método de reutilização chamado de *composição*. Usando a composição, um objeto tem uma referência de outro ou de vários objetos e a forma que eles trabalham juntos formam o padrão. Esta é uma **reutilização de tempo de execução**, já que se pode mudar as referências do objeto enquanto a aplicação está sendo executada.

Um **padrão** tem **quatro elementos essenciais**:

Nome do padrão	Permite aos desenvolvedores discutir os padrões de uma forma fácil sem entrar na especificação do código.
Problema	Descreve <b>quando</b> usar um padrão.
Solução	Descreve os <b>designs e as colaborações</b> do objeto que solucionam o problema.
Consequências	<b>Detalha a troca</b> envolvida na solução e indica padrões adicionais (com outras consequências) que também podem ser úteis.

No livro da **GoF**, os **padrões de design** foram divididos em três grupos básicos:

1. **Padrões criacionais (Creational Patterns)**: usados na criação de objetos. Exemplo: *factory, builder, singleton e prototype*.

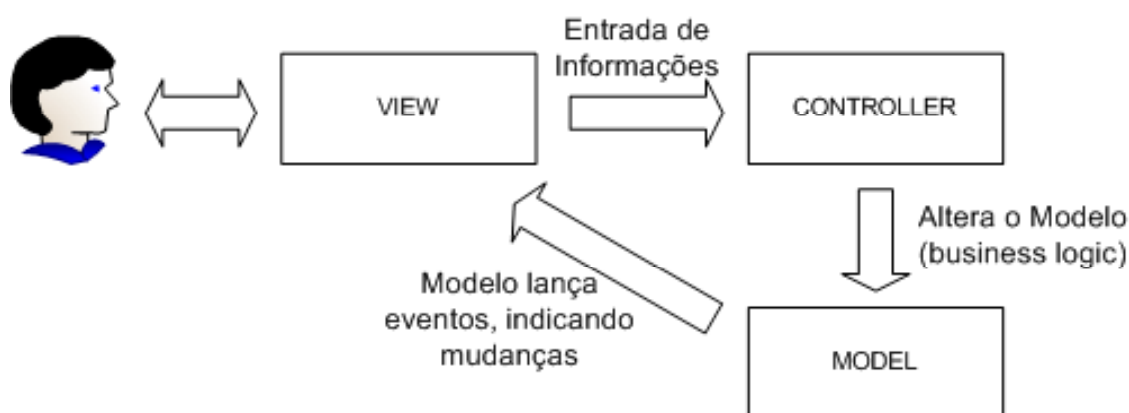
2. **Padrões estruturais (Structural Patterns)**: descrevem como os objetos interagem entre si. Em outras palavras, abordam o *framework* estrutural dos objetos. Exemplo: *bridge, adapter, composite, decorator, facad, e proxy*.

3. **Padrões comportamentais:** descrevem como você pode usar os objetos para alterar o comportamento de um sistema em tempo de execução. Exemplo: *mediator*, *chain of responsibility*, *command*, *memento*, *observer*, *state* e *strategy*.

### OBSERVAÇÃO: O PADRÃO MODEL-VIEW-CONTROLLER

O objetivo do **Padrão de Arquitetura MVC** é separar dados ou lógica de negócios (Model) da interface do usuário (View) e do fluxo da aplicação (Control). A ideia é permitir que uma mesma lógica de negócios possa ser acessada e visualizada por meio de várias interfaces. Na arquitetura MVC, a lógica de negócios (chamaremos de Modelo) não sabe de quantas nem quais interfaces com o usuário estão exibindo seu estado.

A arquitetura Model-View-Controller (MVC), originalmente usada na linguagem de programação Smalltalk, é útil para compreender como essas diferentes tecnologias J2EE se encaixam e funcionam juntas. Para as pessoas não familiarizadas com a arquitetura MVC, a ideia básica é minimizar a ligação entre os objetos em um sistema alinhando-os com um conjunto específico de responsabilidades na área dos dados permanentes e regras associadas (Model), apresentação (View) e lógica da aplicação (Controller). Isso é mostrado na figura a seguir:



Neste padrão, a aplicação é dividida em três partes:

- **Modelo (MODEL):** lógica de negócio;
- **Visão (VIEW):** camada de interface com o usuário. Nesta camada o usuário vê o estado do modelo e pode manipular a interface, para ativar a lógica do negócio;
- **Controlador (CONTROLLER):** transforma eventos gerados pela interface em ações de negócio, alterando o modelo.

O Model é responsável por manter o estado e os dados da aplicação. Pode receber e responder as consultas a partir de View e pode fornecer notificações para View quando as coisas mudam.

O Controller atualiza o Model com base na execução da lógica da aplicação em resposta aos gestos do usuário (por exemplo, botões da caixa de diálogo, solicitações de envio do formulário etc.). Também é responsável por informar a View o que exibir em resposta aos gestos do usuário.

O View é responsável pela apresentação real dos dados fornecidos por Controller.

## PRINCÍPIOS SOLID

Os **princípios SOLID** são um conjunto de **cinco princípios orientados a objetos** usados para projetar e escrever código de software mais escalável, extensível e fácil de manter.

Cada letra do acrônimo **SOLID** representa um princípio específico. São eles:

S	<b>Princípio da Responsabilidade Única (Single Responsibility Principle)</b> Uma classe deve ter apenas uma responsabilidade.
O	<b>Princípio do Aberto-Fechado (Open-Closed Principle)</b> Uma <b>classe deve estar aberta para extensão</b> , mas fechada para modificação.
L	<b>Princípio da Substituição de Liskov (Liskov Substitution Principle)</b> Um <b>objeto de uma classe derivada deve poder ser substituído por um objeto de sua classe base</b> sem afetar o comportamento do programa.
I	<b>Princípio da Segregação de Interface (Interface Segregation Principle)</b> As <b>interfaces de uma classe devem ser separadas e específicas</b> para cada cliente.
D	<b>Princípio da Inversão de Dependência (Dependency Inversion Principle)</b> Os <b>módulos de alto nível não devem depender dos módulos de baixo nível</b> . Ambos devem depender de abstrações.

Esses princípios são importantes pois ajudam a criar um **código mais flexível, adaptável e fácil de manter**. **Eles promovem a modularidade, o baixo acoplamento e a alta coesão em um sistema orientado a objetos**, o que torna o software mais fácil de estender e modificar no futuro.

## PRINCÍPIOS GRASP (GENERAL RESPONSIBILITY ASSIGNMENT SOFTWARE PATTERNS)

Padrões GRASP (General Responsibility Assignment Software Patterns) são um conjunto de princípios para a atribuição de responsabilidades em projetos de programação orientados a objetos. Esses padrões ajudam a desenvolver software com alta coesão e baixo acoplamento, o que leva a sistemas mais fáceis de manter, estender e modificar.

Os **nove padrões GRASP** são:

Especialista na Informação

Atribui responsabilidades para uma classe que contém as informações necessárias para realizar uma tarefa.

Criador	Atribui responsabilidades para uma classe que cria objetos de outras classes
Controlador	Atribui responsabilidades para uma classe que gerencia a interação entre objetos e assegura que os objetos sejam utilizados corretamente
Acoplamento Baixo	Reduz o grau de dependência entre as classes, minimizando o impacto de mudanças em uma classe sobre outras classes
Coesão Alta	Garante que as responsabilidades atribuídas a uma classe estejam altamente relacionadas e sejam interdependentes
Polimorfismo	Permite que objetos de diferentes classes respondam a uma mesma mensagem
Pure Fabrication	Cria uma classe que não representa um conceito do domínio, mas é útil para a implementação de tarefas específicas.
Indireção	Adiciona um intermediário entre dois objetos, reduzindo a dependência direta entre eles.
Variação	Define uma interface comum para um conjunto de classes relacionadas, para permitir a escolha dinâmica de uma implementação em tempo de execução.

Esses padrões são aplicáveis em várias áreas do desenvolvimento de software, como modelagem de domínio, design de classes e estruturação de interfaces de usuário.

A aplicação dos padrões GRASP pode melhorar a qualidade do software e a produtividade do desenvolvedor.

## DOMAIN DRIVEN DESIGN (DDD)

**Domain-Driven Design (DDD)** ou **Projeto Orientado a Domínio** é um padrão de modelagem de software orientado a objetos que procura **reforçar conceitos e boas práticas relacionadas à Orientação a Objetos (OO)**, como:

- alinhamento do código com o negócio;
- favorecer a **reutilização** (os blocos de construção facilitam aproveitar um mesmo conceito de domínio ou um mesmo código em vários lugares);
- mínimo de **acoplamento** (com um modelo bem feito, organizado, as várias partes de um sistema interagem sem que haja muita dependência entre módulos ou classes de objetos de conceitos distintos);
- **independência** da tecnologia.

Além dos conceitos de OO, **DDD** (ou **Projeto Orientado a Domínio**) baseia-se em **duas premissas** principais:

- o foco principal deve ser o **domínio**;
- domínios complexos devem estar baseados em um **modelo**.

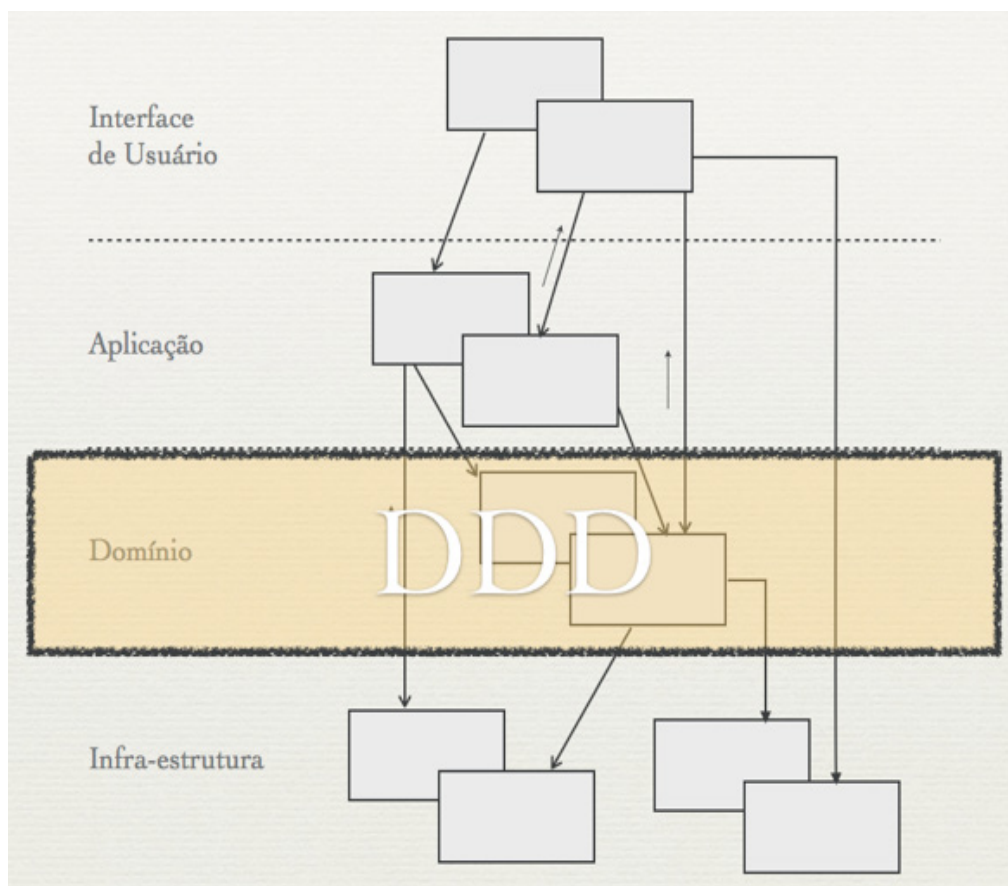


Figura. Arquitetura em camadas, utilizada para separar o domínio do resto da aplicação. Fonte: Cukier (2010)

Conforme destaca Cukier (2010), quando temos um sistema legado, com código muito bagunçado e uma interface complexa, e estamos escrevendo um sistema novo com o código razoavelmente bem feito, criamos uma camada entre esses dois sistemas, intitulada **camada anti-corrupção**. O nosso sistema novo e bem feito falará com essa camada, que possui uma interface bem feita. E a camada anti-corrupção é responsável por traduzir e adaptar as chamadas para o sistema legado, usando uma fachada interna.

Um dos princípios-chave do **Domain-Driven Design** é o uso de uma **linguagem ubíqua** com termos bem definidos, que integram o domínio do negócio e que são utilizados entre desenvolvedores especialistas de negócio

Para o DDD, o domínio é o coração do software, e é importante que as equipes de desenvolvimento trabalhem em colaboração com os especialistas do domínio para modelá-lo com precisão. Uma vez modelado, o domínio é então traduzido em código através de uma linguagem ubíqua, que é compartilhada entre especialistas de negócios e desenvolvedores.

Além do modelo de domínio, o DDD também inclui práticas e padrões para a implementação do código, como a utilização de arquiteturas hexagonal ou de camadas, a implementação de



testes automatizados e a utilização de técnicas de refatoração. **O objetivo geral do DDD é criar um software flexível, adaptável e de alta qualidade que atenda às necessidades do negócio.**

## DIRETO DO CONCURSO



**006.** (CESPE/TRE-PE/2017) O DDD (domain-driven design):

- a) consiste em uma técnica que trata os elementos de domínio e que garante segurança à aplicação em uma programação orientada a objetos na medida em que esconde as propriedades desses objetos.
- b) não tem como foco principal a tecnologia, mas o entendimento das regras de negócio e de como elas devem estar refletidas no código e no modelo de domínio.
- c) prioriza a simplicidade do código, sendo descartados quaisquer usos de linguagem ubíqua que fujam ao domínio da solução.
- d) constitui-se de vários tratadores e (ou) programas que processam os eventos para produzir respostas e de um disparador que invoca os pequenos tratadores.
- e) define-se como uma interface de domínio normalmente especificada e um conjunto de operações que permite acesso a uma funcionalidade da aplicação.



a) Errada. Esse item descreve o princípio de encapsulamento da Programação Orientada a Objetos. Esse princípio não está relacionado à **Domain-Driven Design (DDD)** ou **Projeto Orientado a Domínio**. A DDD procura reforçar conceitos e boas práticas relacionadas à Orientação a Objetos (OO), como:

- alinhamento do código com o negócio;
- favorecer a **reutilização** (os blocos de construção facilitam aproveitar um mesmo conceito de domínio ou um mesmo código em vários lugares);
- mínimo de **acoplamento** (com um modelo bem feito, organizado, as várias partes de um sistema interagem sem que haja muita dependência entre módulos ou classes de objetos de conceitos distintos);
- **independência** da tecnologia.

b) Certa. Isso mesmo! DDD não tem como foco principal a tecnologia, mas o entendimento das regras de negócio e de como elas devem estar refletidas no código e no modelo de domínio.

c) Errada. O *Domain-Driven Design* (DDD) utiliza uma **Ubiquitous Language (linguagem ubíqua)**, ou seja, um vocabulário comum entre usuários e desenvolvedores baseado no domínio do negócio.

**Obs.: Ubiquitous Language** é uma linguagem que se concentra mais nos objetivos do software do que nos meios, ou seja, na parte técnica. Isso permite maior integração entre as partes “não-técnicas”, e evita que se façam muitas traduções da linguagem técnica para uma linguagem mais simples.

d) Errada. Esse item está relacionado à Programação Orientada a Eventos e, não, a Domínio.

e) Errada. Esse item está relacionado à Programação Orientada a Objetos.

**Letra b.**

**007.** (CESPE/MPOG/2013) De acordo com os padrões de DDD (Domain-Driven Design), ao se escrever um novo sistema para também interagir com um sistema legado (considerado um código de difícil manutenção), cria-se uma camada entre os dois sistemas denominada camada anticorrupção.



Conforme visto, tem-se nesse contexto a **camada anti-corrupção**, responsável por traduzir e adaptar as chamadas para o sistema legado, usando uma fachada interna, ou seja, ela funciona como um adaptador.

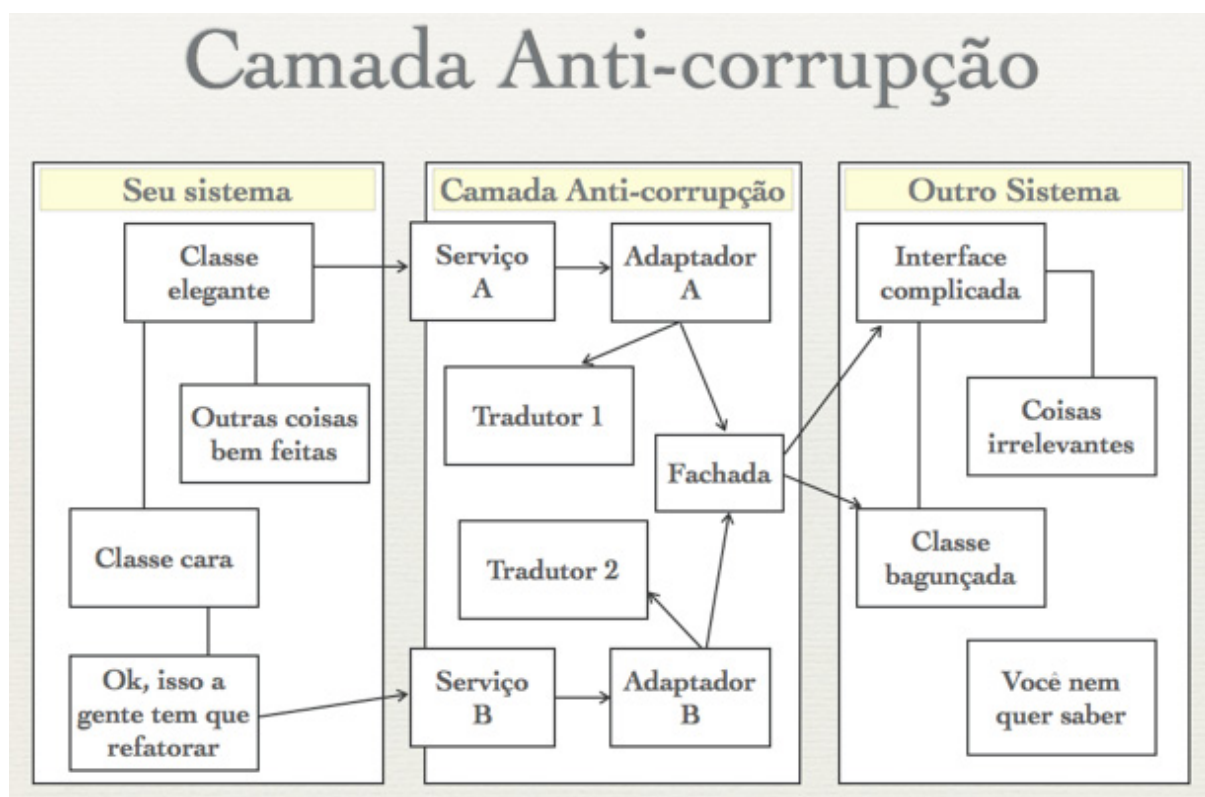


Figura. Camada Anti-corrupção. Fonte: Cukier (2010)

**Certo.**



**008.** (CESPE/BACEN/2013) A linguagem ubíqua utiliza termos que fazem parte das discussões entre os especialistas de negócio e as equipes de desenvolvimento, os quais devem utilizar a mesma terminologia na linguagem falada e no código.



O *Domain-Driven Design* (DDD) utiliza uma **Ubiquitous Language (linguagem ubíqua)**, ou seja, um vocabulário comum entre usuários e desenvolvedores baseado no domínio do negócio.

**Certo.**

**009.** (CESPE/STJ/2015) Domain-Driven Design pode ser aplicada ao processo de concepção arquitetural de um sistema de software, sendo que domain, em um software, designa o campo de ação, conhecimento e influência.



**Domain-Driven Design** ou **Projeto Orientado a Domínio** é um padrão de modelagem de software orientado a objetos que procura reforçar conceitos e boas práticas relacionadas à OO. Além dos conceitos de OO, DDD baseia-se em duas premissas principais:

- o foco principal deve ser o **domínio** (a lógica de negócios do software, seu campo de ação/atuação);
- domínios complexos devem estar baseados em um **modelo**.

**Certo.**

**010.** (CESPE/STJ/2015) Um dos princípios-chave do Domain-Driven Design é o uso de uma linguagem ubíqua com termos bem definidos, que integram o domínio do negócio e que são utilizados entre desenvolvedores especialistas de negócio.



O *Domain-Driven Design* (DDD) utiliza uma **Ubiquitous Language (linguagem ubíqua)**, ou seja, um vocabulário comum entre usuários e desenvolvedores baseado no domínio do negócio.

**Certo.**

**011.** (CESPE/SLU-DF/2019) No desenvolvimento embasado em domain-driven design, a definição da tecnologia a ser utilizada tem importância secundária no projeto.



Isso mesmo! O *Domain-Driven Design* (DDD) não tem como foco principal a tecnologia, mas o entendimento das regras de negócio e de como elas devem estar refletidas no código e no modelo de domínio.

**Certo.**

## ARQUITETURA HEXAGONAL (PORTAS E ADAPTADORES)

A **Arquitetura Hexagonal (Arquitetura de Portas e Adaptadores)** é uma abordagem para o desenvolvimento de software que **visa separar a lógica de negócios da aplicação da infraestrutura tecnológica que a suporta**. Ela é **composta por três camadas principais**:

Camada	Descrição
Camada de Domínio	Contém as regras de negócios da aplicação e as entidades que representam os objetos de domínio.
Camada de Aplicação	Responsável por orquestrar a interação entre a camada de domínio e as interfaces de entrada e saída.
Camada de Infraestrutura	Contém a lógica de infraestrutura da aplicação, como o acesso a bancos de dados, a configuração de protocolos de comunicação e o controle de fluxo.

### DIRETO DO CONCURSO

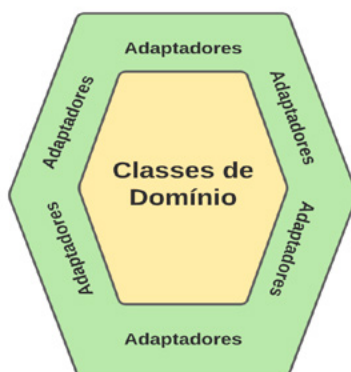


**012.** (CEBRASPE/CESPE/BANRISUL/TÉCNICO DE TECNOLOGIA DA INFORMAÇÃO/2022) A respeito de tecnologias de integração, julgue o próximo item.

Em uma arquitetura hexagonal, as classes de domínio independem das classes de infraestrutura, tecnologias e sistemas externos.



A ideia central da **Arquitetura Hexagonal** é que as interfaces de entrada e saída (também chamadas de portas) sejam definidas na camada de aplicação, de forma que **a lógica de negócios da aplicação possa ser isolada e testada independentemente da tecnologia utilizada para implementar as interfaces**. As interfaces são então adaptadas (ou conectadas) para a infraestrutura tecnológica da aplicação, através de adaptadores que implementam os protocolos de comunicação e a lógica de acesso a dados necessários para cada interface. A arquitetura pode ser representada pela imagem a seguir:



Fonte: Valente (2020)

**Essa separação entre a lógica de negócios e a infraestrutura tecnológica da aplicação** torna a Arquitetura Hexagonal uma abordagem flexível e adaptável, que **permite que a aplicação possa ser facilmente modificada e expandida sem afetar sua lógica de negócios**. Além disso, a arquitetura facilita a implementação de testes automatizados e a manutenção da aplicação, tornando-a mais robusta e escalável.

**Certo.**

---

Bem, após essa revisão inicial, vamos às questões?

## RESUMO

A **orientação a objetos** possui os **principais componentes**:

### a) Objeto:

Um objeto simplesmente é algo (coisa ou pessoa) que faz sentido no contexto de uma aplicação. O objeto é um conceito, uma abstração, algo com limites nítidos e significado em relação ao problema em causa. Eles facilitam a compreensão do mundo real e oferecem uma base real para ser implementada no computador.

**Todos os objetos têm identidade e são distinguíveis.**

**Identidade** significa que os objetos se distinguem pela sua própria existência, e **não** por suas propriedades.

O paradigma da orientação a objetos **não** possui foco nos procedimentos como nos paradigmas procedurais, e sim nos **objetos**, já que estes vão ter o papel principal nas aplicações, contendo informações de estado (atributos) e fazendo a comunicação com outros objetos, realizando uma interação entre si.

### b) Classe:

Uma **classe de objetos descreve um grupo de objetos com propriedades semelhantes (atributos), o mesmo comportamento (operações), os mesmos relacionamentos com outros objetos e a mesma semântica.**

A maioria dos objetos deriva sua individualidade das diferenças entre os valores de seus atributos e relacionamentos com outros objetos, mas também é possível a existência de objetos com idênticos valores de atributos e relacionamentos.

As linguagens de programação orientadas a objeto podem determinar a classe de um objeto em tempo de execução. A classe de um objeto é uma propriedade implícita do objeto. Ao se agrupar objetos em classe generalizam-se definições comuns, como nome da classe e nomes dos atributos, sendo armazenados uma vez por classe ao invés de por instância, isto é a **abstração**.

### Principais Componentes da Classe:

- Um **atributo** é um valor de dado guardado pelos objetos de uma classe, e cada atributo possui um valor para cada instância de objeto. Diferentes instâncias de objetos podem ter valores iguais ou diferentes para um dado atributo e é importante frisar que cada nome de atributo é único dentro de uma classe. Um atributo deve ser um puro valor de dado, e não um objeto. Diferentemente dos objetos, os valores puros de dados não têm identidade.
- O **Método** é uma função ou transformação que pode ser aplicada a objetos ou por estes a uma classe, e todos os objetos de uma classe compartilham as mesmas operações. Cada operação tem um objeto-alvo, seu comportamento depende da

classe de seu alvo. A mesma operação pode se aplicar a muitas classes diferentes, sendo assim ela é polimórfica, isto é, a mesma operação toma formas diferentes em classes diferentes. Uma operação pode ter argumentos além de um objeto-alvo. Esses argumentos servem como parâmetros para a operação, mas não afetam a escolha do método.

Quando uma operação possui métodos em várias classes, é importante que todos esses métodos tenham a mesma **assinatura**, a quantidade e tipos de argumentos e o tipo do valor resultante.

### **c) Associação:**

É uma **conexão física ou conceitual entre instâncias de objetos**.

Pode ser definida como uma lista ordenada de instâncias de objetos.

Uma associação descreve um grupo de ligações com estrutura e semântica comuns, ou seja, descreve um conjunto de potenciais ligações (da mesma maneira que uma classe descreve um conjunto de potenciais objetos), e as associações são intrinsecamente bidirecionais.

As associações muitas vezes são implementadas em linguagens de programação como ponteiros de um objeto para outro. **As associações não são simplesmente construções de banco de dados**, embora os bancos de dados relacionais sejam construídos sobre os conceitos de associações.

### **d) Multiplicidade:**

Especifica quantas instâncias de uma classe relacionam-se a uma única instância de uma classe associada. A multiplicidade restringe a quantidade de objetos relacionados.

A multiplicidade depende de pressupostos e de como são definidas as fronteiras de um problema. A mais importante distinção de multiplicidade é a existente entre “um” e “muitos”. A subestimação da multiplicidade pode restringir a flexibilidade de aplicação.

### **e) Herança:**

É o compartilhamento de atributos e operações entre classes com base num relacionamento hierárquico.

Uma classe pode ser dividida em subclasses mais definidas.

Cada subclasse herda todas as propriedades de sua superclasse e acrescenta suas próprias e exclusivas características.

As propriedades da superclasse não precisam ser repetidas em cada subclasse. A *herança* é uma das principais características (ou mesmo vantagem) dos sistemas baseados em objetos. A capacidade de identificar propriedades comuns a várias classes de uma superclasse comum e de fazê-las herdar as propriedades da sua superclasse pode reduzir substancialmente as repetições nos projetos e programas.

A UML propõe os seguintes diagramas:

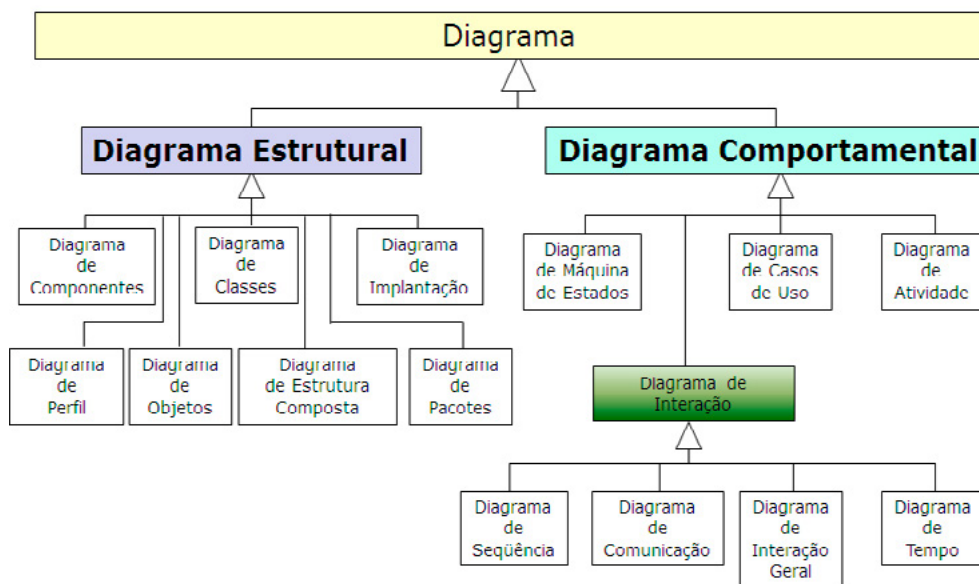


Figura. Especificação 2.2 da UML. Disponível em: <http://www.omg.org/spec/UML/>

## DOMAIN-DRIVEN DESIGN (DDD) OU PROJETO ORIENTADO A DOMÍNIO

**Domain Driven Design (DDD)** é uma abordagem para o desenvolvimento de software que **ênfatiza a importância de entender e modelar o domínio do problema que o software deve resolver**. A ideia central do DDD é que o desenvolvimento de software deve se concentrar no modelo de domínio, ou seja, na estrutura, comportamento e vocabulário relacionados às regras de negócio da aplicação.

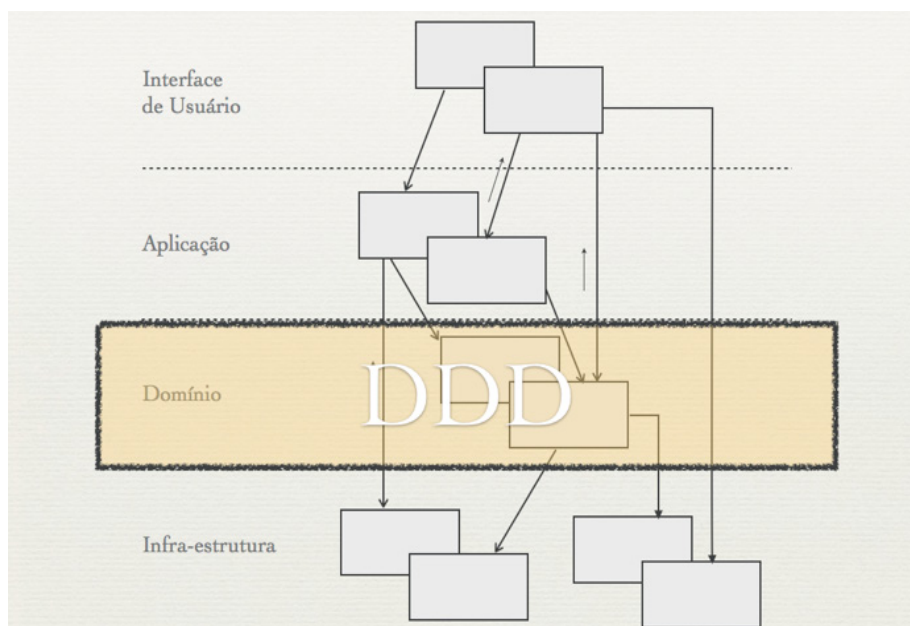


Figura. Arquitetura em camadas, utilizada para separar o domínio do resto da aplicação. Fonte: Cukier (2010)

## ARQUITETURA HEXAGONAL (ARQUITETURA DE PORTAS E ADAPTADORES)

É uma abordagem para o desenvolvimento de software que **visa separar a lógica de negócios da aplicação da infraestrutura tecnológica que a suporta**. Ela é **composta por três camadas principais**:

Camada	Descrição
Camada de Domínio	Contém as regras de negócios da aplicação e as entidades que representam os objetos de domínio.
Camada de Aplicação	Responsável por orquestrar a interação entre a camada de domínio e as interfaces de entrada e saída.
Camada de Infraestrutura	Contém a lógica de infraestrutura da aplicação, como o acesso a bancos de dados, a configuração de protocolos de comunicação e o controle de fluxo.

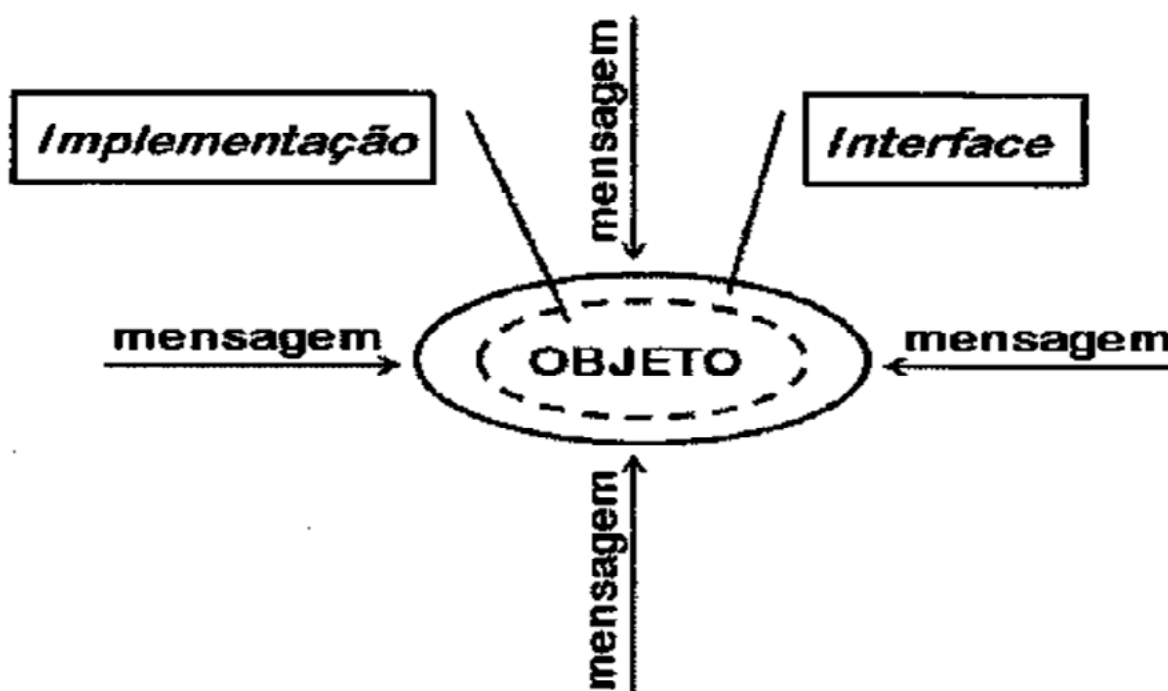
## QUESTÕES COMENTADAS EM AULA

**001.** (FCC/MPE-PB/ANALISTA DE SISTEMAS/2015) Apresenta um conceito correto associado à Análise e Projeto Orientado a Objetos (OO):

Objeto é uma descrição generalizada que descreve uma coleção de métodos semelhantes e encapsula dados e abstrações procedurais necessárias para descrever alguma classe do mundo real.

**002.** (CESPE/BANCO DA AMAZÔNIA/TÉCNICO CIENTÍFICO/2010) Considerando os aspectos de linguagem de programação, julgue os itens subsequentes. A abordagem embasada em objetos preocupa-se primeiro em identificar os objetos contidos no domínio da aplicação e, em seguida, em estabelecer os procedimentos relativos a eles.

**003.** (CEPERJ/IPEM-RJ/ANALISTA DE SISTEMAS/2010) No que tange aos paradigmas da Orientação a Objetos (OO), um princípio está diretamente relacionado às operações realizadas por um objeto e ao modo como as operações são executadas, constituindo uma forma de restringir o acesso ao comportamento interno de um objeto. Nesse processo, um objeto que precise da colaboração de outro para realizar alguma tarefa deve enviar uma mensagem a este último. Além disso, separa os aspectos externos de um objeto dos detalhes internos da implementação. Considerando esse contexto, observe a figura abaixo.



O princípio da OO é conhecido por:



- a) Compartilhamento
- b) Acoplamento
- c) Herança
- d) Polimorfismo
- e) Encapsulamento

**004.** (FCC/TRF 3ª REGIÃO/ANALISTA JUDICIÁRIO/2019) O Polimorfismo, um dos Pilares da Programação Orientada a Objetos – POO,

- a) ocorre quando uma classe tem um relacionamento do tipo “1 para” com outra classe e isso implica no modo como a definição das classes devem ocorrer nas aplicações.
- b) consiste em esconder os atributos da classe de quem for utilizá-la. Isso se deve a: 1 – para quem for usar a classe não a use de forma errada; e 2 – para que implementação seja feita por meio dos métodos get e set.
- c) permite que um mesmo método possa ter vários comportamentos e a definição de qual comportamento será executado se dá pelo valor diferente de um de seus atributos.
- d) **é um conceito que permite que as características bem como as operações, de um modo global, possam ser repassadas para várias funcionalidades da aplicação.**
- e) permite utilizar atributos e operações diferentes de uma subclasse, acrescentando ou substituindo características herdadas da classe pai.

**005.** (CEPERJ/IPEM-RJ/ANALISTA DE SISTEMAS/2010) A UML é uma linguagem de modelagem visual, podendo ser definida como um conjunto de notações e semântica correspondente para representar visualmente uma ou mais perspectivas de um sistema. Dentre os diagramas da UML, um diagrama foca os requisitos funcionais de um sistema, forçando os desenvolvedores a moldarem o sistema de acordo com o usuário, e não o usuário de acordo com o sistema, representando as especificações de uma sequência de interações entre um sistema e os agentes externos que utilizam esse sistema, por meio dos atores e relacionamentos entre eles. Esse diagrama é denominado:

- a) Casos de uso
- b) Casos de negócios
- c) Processos e funções
- d) Entidades e relacionamentos
- e) Processos e relacionamentos

**006.** (CESPE/TRE-PE/2017) O DDD (domain-driven design):

- a) consiste em uma técnica que trata os elementos de domínio e que garante segurança à aplicação em uma programação orientada a objetos na medida em que esconde as propriedades desses objetos.

- b) não tem como foco principal a tecnologia, mas o entendimento das regras de negócio e de como elas devem estar refletidas no código e no modelo de domínio.
- c) prioriza a simplicidade do código, sendo descartados quaisquer usos de linguagem ubíqua que fujam ao domínio da solução.
- d) constitui-se de vários tratadores e (ou) programas que processam os eventos para produzir respostas e de um disparador que invoca os pequenos tratadores.
- e) define-se como uma interface de domínio normalmente especificada e um conjunto de operações que permite acesso a uma funcionalidade da aplicação.

**007.** (CESPE/MPOG/2013) De acordo com os padrões de DDD (Domain-Driven Design), ao se escrever um novo sistema para também interagir com um sistema legado (considerado um código de difícil manutenção), cria-se uma camada entre os dois sistemas denominada camada anticorrupção.

**008.** (CESPE/BACEN/2013) A linguagem ubíqua utiliza termos que fazem parte das discussões entre os especialistas de negócio e as equipes de desenvolvimento, os quais devem utilizar a mesma terminologia na linguagem falada e no código.

**009.** (CESPE/STJ/2015) Domain-Driven Design pode ser aplicada ao processo de concepção arquitetural de um sistema de software, sendo que domain, em um software, designa o campo de ação, conhecimento e influência.

**010.** (CESPE/STJ/2015) Um dos princípios-chave do Domain-Driven Design é o uso de uma linguagem ubíqua com termos bem definidos, que integram o domínio do negócio e que são utilizados entre desenvolvedores especialistas de negócio.

**011.** (CESPE/SLU-DF/2019) No desenvolvimento embasado em domain-driven design, a definição da tecnologia a ser utilizada tem importância secundária no projeto.

**012.** (CEBRASPE/CESPE/BANRISUL/TÉCNICO DE TECNOLOGIA DA INFORMAÇÃO/2022) A respeito de tecnologias de integração, julgue o próximo item.

Em uma arquitetura hexagonal, as classes de domínio independem das classes de infraestrutura, tecnologias e sistemas externos.

## QUESTÕES DE CONCURSO

**013.** (IBFC/EBSERH/ANALISTA TECNOLOGIA DA INFORMAÇÃO/2022) Assinale, das alternativas abaixo, a única que identifica incorretamente sobre os conceitos básicos de programação orientada a objetos.

- a) Encapsulamento consiste na separação de aspectos internos e externos de um objeto.
- b) Interface é um contrato entre a classe e o mundo externo.
- c) Polimorfismo permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam.
- d) Herança é a sobrecarga pelo qual uma classe pode estender outro objeto ou ser estendida por outro método.
- e) Classe representa um conjunto de objetos com características afins.

**014.** (CESPE/MEC/ANALISTA DE TESTE E QUALIDADE/2015) Com relação à orientação a objetos, julgue o item subsecutivo. O foco da orientação a objetos está nos procedimentos a serem contemplados pelo sistema e nas informações que este manipulará e(ou) armazenará.

**015.** (CESPE/SERPRO/ANALISTA DESENVOLVIMENTO DE SISTEMAS/2013) Julgue os itens que se seguem, a respeito das tecnologias JSE, JME e JEE.

A herança — um princípio de orientação a objetos que permite que classes compartilhem atributos e métodos — é utilizada para reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.

**016.** (FCC/AL-RN/ANALISTA LEGISLATIVO/2013) Um dos conceitos básicos de orientação a objetos é o fato de um objeto, ao tentar acessar as propriedades de outro objeto, deve sempre fazê-lo por uso de métodos do objeto ao qual se deseja atribuir ou requisitar uma informação, mantendo ambos os objetos isolados. A essa propriedade da orientação a objetos se dá o nome de

- a) herança.
- b) abstração.
- c) polimorfismo.
- d) mensagem.
- e) encapsulamento.

**017.** (CESPE/BANCO DA AMAZÔNIA/TECNOLOGIA DA INFORMAÇÃO/2010) Julgue os itens seguintes de acordo com os conceitos do paradigma orientado a objetos. Objetos têm identidade própria. Isso garante que, mesmo tendo os mesmos valores de variáveis e pertencendo à mesma classe, dois objetos sejam considerados diferentes.

**018.** (CESPE/TRE-MT/TÉCNICO JUDICIÁRIO/2010) A estrutura interna de um objeto possui dois componentes básicos: atributos, que descrevem o estado do objeto; e métodos, que são responsáveis pela comunicação entre objetos.

**019.** (CESPE/BANCO DA AMAZÔNIA/TECNOLOGIA DA INFORMAÇÃO/2010) Na modelagem de classes, a hierarquia entre elas é representada por meio de um relacionamento chamado generalização.

**020.** (CESPE/CEHAP PB/ANALISTA DE SISTEMAS/2009) O projeto orientado a objetos encapsula atributos e serviços ou operações (comportamentos) em objetos. Os objetos comunicam-se entre si por meio de associações e os relacionamentos entre classes se dão por meio de interfaces.

**021.** (ESAF/MPOG/2008) Em Modelagem Orientada a Objetos, é correto afirmar que:

- a) uma classe deve representar uma abstração poliforme das propriedades dos objetos individuais que pertencem a um atributo.
- b) uma invariante de objeto é uma condição que toda classe desse objeto deve satisfazer para condições opcionais (quando o objeto está em equilíbrio).
- c) por meio do escopo de polimorfismo, uma classe é derivada diretamente de mais de uma subclasse.
- d) uma classe com coesão de escopo tem algumas características que são indefinidas para algumas classes dos objetos.
- e) por meio da herança múltipla, uma classe é derivada diretamente de mais de uma superclasse.

**022.** (ESAF/SEFAZ-CE/2007) A representação de classes em diagramas UML contempla os seguintes tipos básicos de informação:

- a) o nome da instância da classe e os seus objetos.
- b) o nome da classe, os seus atributos e os seus métodos.
- c) o nome da instância da classe e os seus relacionamentos.
- d) o nome da classe, os seus atributos e suas exceções.
- e) o nome da classe e suas visibilidades.

**023.** (ESAF/TRF/2006) Um mesmo nome de objeto pode ser usado para identificar diferentes objetos em uma mesma classe ou diferentes objetos em classes diferentes, evitando assim, que seja necessário usar nomes diferentes para objetos diferentes que realizam a mesma operação. A esta característica da Orientação a Objetos dá-se o nome de Polimorfismo.

**024.** (ESAF/RECEITA FEDERAL/2005) O modo para descrever os vários aspectos de modelagem pela UML é por meio do uso da notação definida pelos seus vários tipos de diagramas. Segundo as características desses diagramas, é correto afirmar que um diagrama de classe

- a) mostra a interação de um caso de uso organizada em torno de objetos e classes e seus vínculos mútuos, evidenciando a sequência de mensagens.
- b) denota a estrutura estática de um sistema.
- c) descreve a funcionalidade do sistema.
- d) descreve a interação de sequência de tempo dos objetos e classes percebida por atores externos.
- e) mostra as sequências de estados que uma classe e objetos assumem em sua vida em resposta a estímulos recebidos, juntamente com suas respostas e ações.

**025.** (ESAF/AFRF/2005) Analise as seguintes afirmações relacionadas à análise e ao projeto orientados a objetos:

I – O principal propósito do diagrama entidade relacionamento (E-R) é representar os objetos e suas relações.

II – As tabelas de objetos de dados podem ser “normalizadas”, aplicandose um conjunto de regras de normalização, resultando em um “modelo relacional” para os dados. Uma dessas regras especifica que: determinada instância de um objeto tem um e somente um valor para cada atributo.

III – Um objeto em potencial não poderá ser utilizado ou considerado durante a análise se a informação sobre ele precisar ser lembrada para que o sistema possa funcionar.

IV – Devido à característica da reusabilidade da orientação a objetos, a prototipação é um modelo de desenvolvimento de software que não pode ser considerado nem utilizado na análise orientada a objetos.

Indique a opção que contenha todas as afirmações verdadeiras.

- a) I e III
- b) II e III
- c) III e IV
- d) I e II
- e) II e IV

**026.** (FCC/SABESP/TECNÓLOGO/2014) A engenharia de software apresenta um conjunto de princípios que podem ser usados quando um projeto de desenvolvimento de software for realizado, como os descritos abaixo:

I – Decomposição – o software é um produto complexo construído a partir de partes mais simples. A decomposição funcional é uma maneira de conceber o software como um conjunto

de funções de alto nível (requisitos) que são decompostas em partes cada vez mais simples até chegar a comandos individuais de linguagem de programação.

II – Abstração - muitas vezes é necessário descrever um elemento em uma linguagem de nível mais alto do que o necessário para sua construção. A abstração ajuda os interessados no processo de desenvolvimento a entenderem estruturas grandes e complexas por meio de descrições mais abstratas.

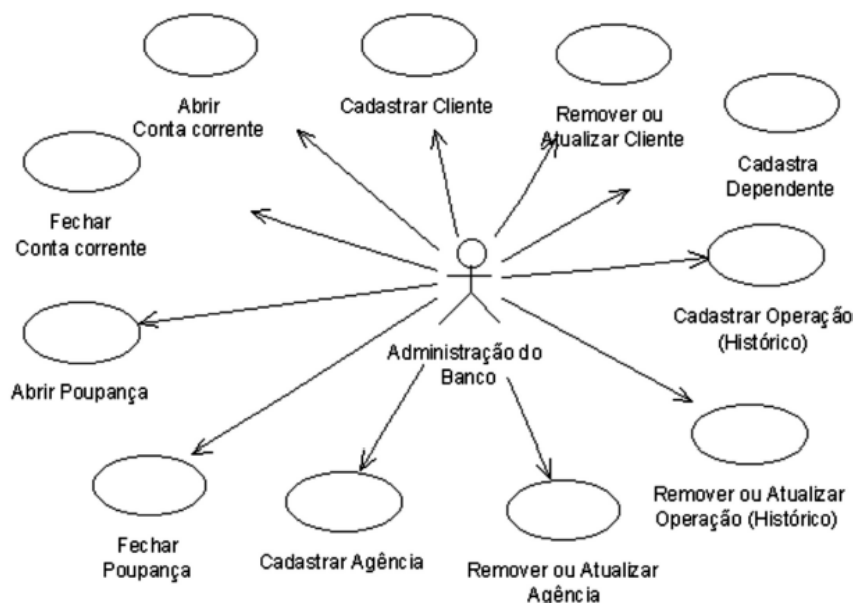
III – Composição - a composição deu origem à orientação a objetos, em que um objeto pode ser classificado simultaneamente em mais de uma classe. Por exemplo, um cão, além de ser um mamífero, é animal e vertebrado.

IV – Padronização - a criação de padrões (*patterns*) de programação, design e análise ajuda a elaborar produtos com qualidade mais previsível. São importantes para a captação de experiências e evitam a repetição de erros que já têm solução conhecida.

Apresentam princípio e descrição corretos o que se afirma APENAS em

- a) I, II e III.
- b) IV.
- c) II e III.
- d) III e IV.
- e) I, II e IV.

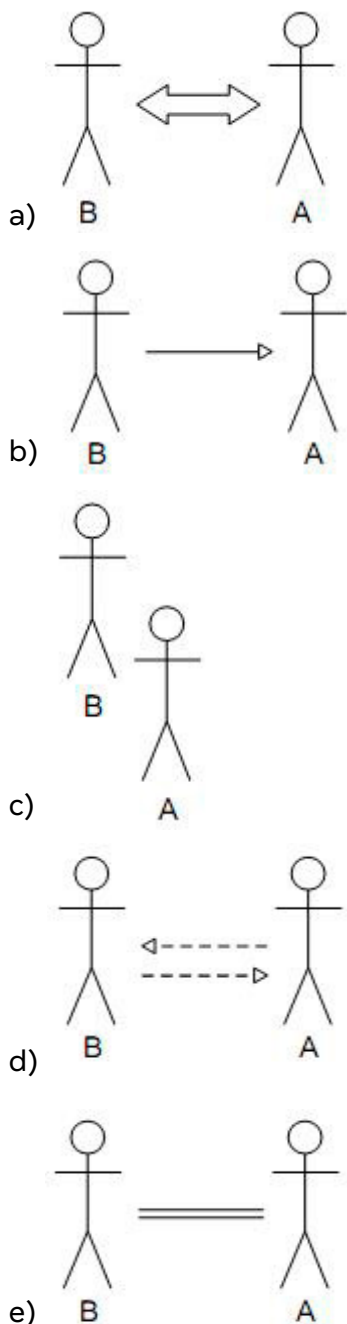
**027.** (FUNCAB/MP—RO/ANALISTA/2012) A figura abaixo representa o diagrama UML denominado:



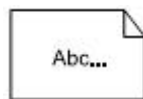
- a) Diagrama de Classes.
- b) Diagrama de Atividade.

- c) Diagrama de Caso de Uso.
- d) Diagrama de Sequência.
- e) Diagrama de Componentes.

**028.** (VUNESP/CÂMARA MUNICIPAL DE SÃO JOSÉ DOS CAMPOS-SP/ANALISTA LEGISLATIVO/2014) Deseja-se representar, em um diagrama de casos de uso da UML, a seguinte situação: o ator B herda as propriedades do ator A. A forma correta de representação é



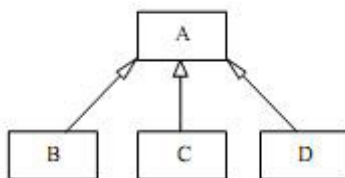
**029.** (VUNESP/SP-URBANISMO/ANALISTA/2014) Considere o seguinte símbolo de um diagrama de classes da orientação a objetos:



Tal símbolo representa

- a) um port.
- b) um subtipo.
- c) um comentário.
- d) uma subclasse.
- e) uma especialidade.

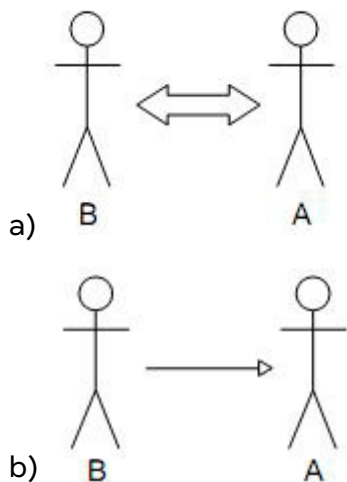
**030.** (VUNESP/DESENVOLVESP/ANALISTA DE SISTEMAS/2014) Considere o seguinte diagrama de classes da UML 2.0:



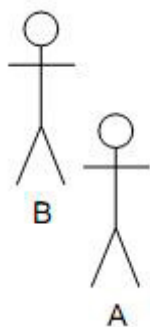
Com base nesse diagrama, é correto afirmar que

- a) a classe A é um tipo especial das classes B, C e D.
- b) as classes A, B, C e D são abstratas.
- c) as classes B, C e D são subclasses da classe A.
- d) está representada uma agregação.
- e) está representada uma composição.

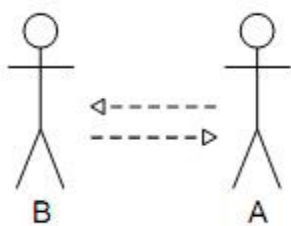
**031.** (VUNESP/CÂMARA MUNICIPAL DE SÃO JOSÉ DOS CAMPOS-SP/ANALISTA LEGISLATIVO/2014) Deseja-se representar, em um diagrama de casos de uso da UML, a seguinte situação: o ator B herda as propriedades do ator A. A forma correta de representação é



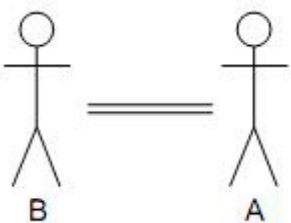




c)

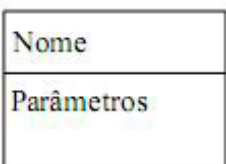


d)

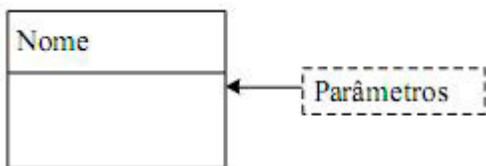


e)

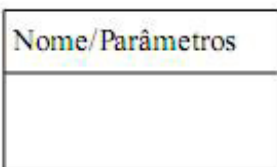
**032.** (VUNESP/EMPLASA/ANALISTA ADMINISTRATIVO/2014) Na UML 2.0, a notação utilizada para representar um template de classe é:



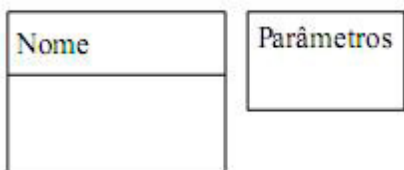
a)



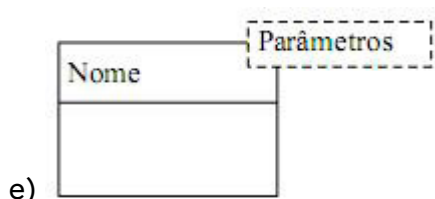
b)



c)



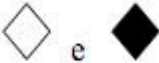
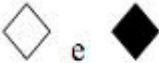
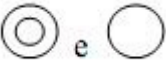
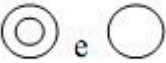
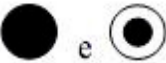
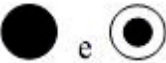
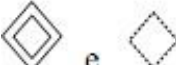
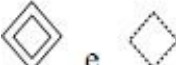
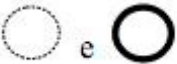
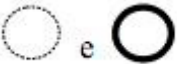
d)



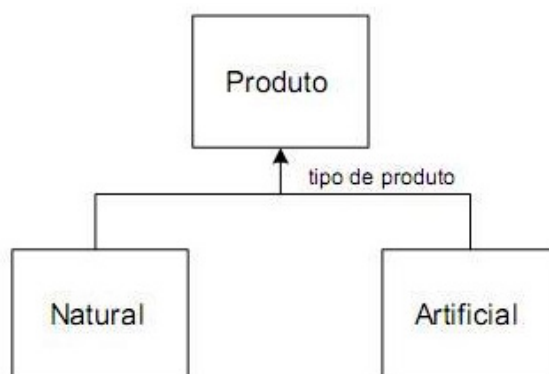
**033.** (VUNESP/EMPLASA/ANALISTA ADMINISTRATIVO/2014) No diagrama de objetos da UML 2.0, o nome de um objeto deve respeitar a seguinte notação:

- a) nome-classe >> nome-objeto
- b) nome-classe (nome-objeto)
- c) nome-objeto < > nome-classe
- d) nome-objeto (nome-classe)
- e) nome-objeto: nome-classe

**034.** (VUNESP/EMPLASA/ANALISTA ADMINISTRATIVO/2014) Em um diagrama de máquinas de estado da UML 2.0, os estados inicial e final são representados, respectivamente, pelos símbolos:

- a)  e 
- b)  e 
- c)  e 
- d)  e 
- e)  e 

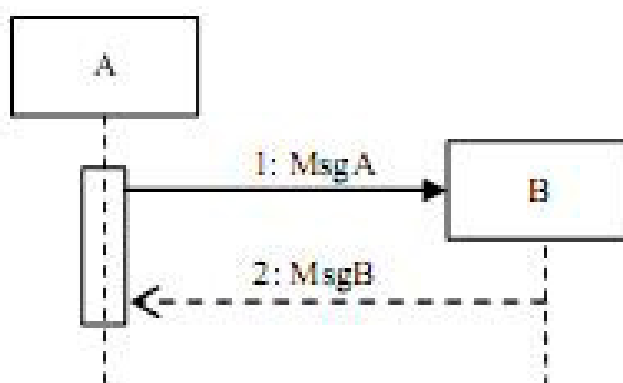
**035.** (VUNESP/DCTA/TECNOLOGISTA PLENO/2013) Considere o seguinte diagrama de classes da UML 2.0.



O texto “tipo de produto” é chamado de um(a)

- a) estado.
- b) interface.
- c) acoplamento.
- d) discriminador.
- e) compartimento.

**036.** (VUNESP/DESENVOLVESP/ANALISTA DE SISTEMAS/2014) Considere o seguinte diagrama de mensagens (ou de sequência), representado por meio da UML 2.0:



Com base nesse diagrama, é correto afirmar que

- a) ambos os objetos (A e B) já existiam antes da geração da mensagem MsgA.
- b) o objeto A é criado em consequência da mensagem MsgB.
- c) o objeto B é armazenado em um banco de dados.
- d) o objeto B é criado em consequência da mensagem MsgA.
- e) o objeto B é destruído após a geração da mensagem MsgB.

## GABARITO

<b>1.</b> E	<b>13.</b> d	<b>25.</b> d
<b>2.</b> C	<b>14.</b> E	<b>26.</b> e
<b>3.</b> e	<b>15.</b> C	<b>27.</b> c
<b>4.</b> c	<b>16.</b> e	<b>28.</b> b
<b>5.</b> a	<b>17.</b> C	<b>29.</b> c
<b>6.</b> b	<b>18.</b> E	<b>30.</b> c
<b>7.</b> C	<b>19.</b> C	<b>31.</b> b
<b>8.</b> C	<b>20.</b> E	<b>32.</b> e
<b>9.</b> C	<b>21.</b> e	<b>33.</b> e
<b>10.</b> C	<b>22.</b> b	<b>34.</b> c
<b>11.</b> C	<b>23.</b> E	<b>35.</b> d
<b>12.</b> C	<b>24.</b> b	<b>36.</b> d

## GABARITO COMENTADO

**013.** (IBFC/EBSERH/ANALISTA TECNOLOGIA DA INFORMAÇÃO/2022) Assinale, das alternativas abaixo, a única que identifica incorretamente sobre os conceitos básicos de programação orientada a objetos.

- a) Encapsulamento consiste na separação de aspectos internos e externos de um objeto.
- b) Interface é um contrato entre a classe e o mundo externo.
- c) Polimorfismo permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam.
- d) Herança é a sobrecarga pelo qual uma classe pode estender outro objeto ou ser estendida por outro método.
- e) Classe representa um conjunto de objetos com características afins.



a) Certa. O **encapsulamento** é a **habilidade de esconder** de outros objetos, as características intrínsecas de um **dado objeto**.

Toda a comunicação inter objetos precisa ser realizada via **métodos**. Um objeto **não** deve ser capaz de acessar e alterar atributos de outro diretamente.

b) Certa. As **interfaces não** são **classes**. As **interfaces** são como um **contrato** ou **padrão que descreve O QUE as classes devem fazer**, **sem** especificar **COMO** devem fazer.

c) Certa. Uma **classe abstrata**:

- Pode ou não incluir métodos abstratos;
- **Não podem ser instanciadas**;
  - Mas elas podem ter subclasses; e
- Pode ter:
  - os Atributos estáticos; e
  - os Métodos estáticos;

Na linguagem Java, é uma classe declarada com o **modificador abstract**.

Uma **classe descendente concreta** é uma classe:

- que descende da classe abstrata; e
- que implementa todos os métodos abstratos da classe herdada.

O **polimorfismo** permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam.

d) Errada. A herança não é sobrecarga, mas sim uma **propriedade** dos objetos **que permite a criação de uma hierarquia entre eles**, em que os descendentes herdam o acesso ao código e estruturas de dados dos seus ancestrais. A POO permite que **classes herdem o estado** e o **comportamento** comumente utilizados de outras classes.

e) Certa. As **classes** são **definições** de **atributos** e **funções** de um **tipo de objetos**. Elas são **coleções de objetos** que:

- podem ser descritos por um **conjunto básico de atributos**;
- possuem **operações semelhantes**;
- estão em uma **mesma semântica**.

**Letra d.**

**014.** (CESPE/MEC/ANALISTA DE TESTE E QUALIDADE/2015) Com relação à orientação a objetos, julgue o item subsecutivo. O foco da orientação a objetos está nos procedimentos a serem contemplados pelo sistema e nas informações que este manipulará e(ou) armazenará.



O paradigma orientado a objetos **não** possui foco nos procedimentos a serem contemplados pelo sistema e nas informações que este manipulará e(ou) armazenará como nos paradigmas procedurais.

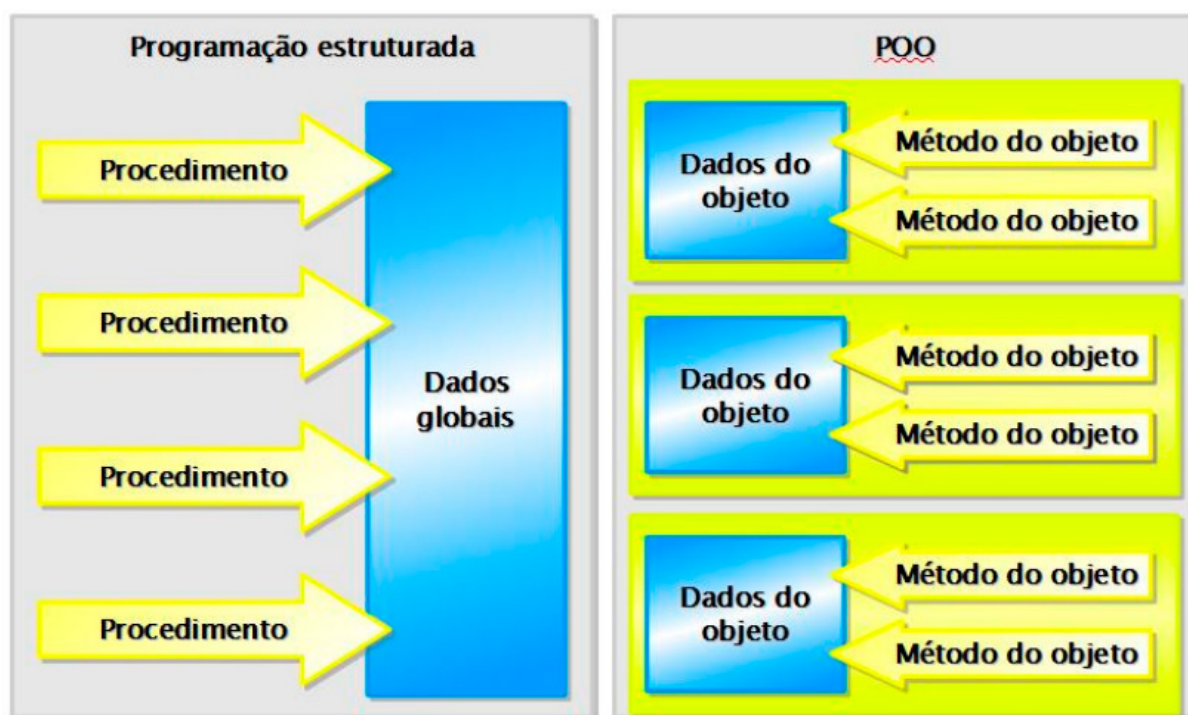


Figura. Programação estrutura x POO (ARAÚJO, R., 2018)

A abordagem embasada em objetos preocupa-se primeiro em identificar os objetos contidos no domínio da aplicação e, em seguida, em estabelecer os procedimentos relativos a eles.

**Errado.**

**015.** (CESPE/SERPRO/ANALISTA DESENVOLVIMENTO DE SISTEMAS/2013) Julgue os itens que se seguem, a respeito das tecnologias JSE, JME e JEE.

A herança — um princípio de orientação a objetos que permite que classes compartilhem atributos e métodos — é utilizada para reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.



Este princípio permite representar membros comuns, serviços e atributos uma só vez, assim como especializar estes membros em casos específicos. **Herança** é a propriedade dos objetos que permite a criação de uma hierarquia entre eles, onde os descendentes herdam o acesso ao código e estruturas de dados dos seus ancestrais.

Observe que a herança torna possível que uma classe compartilhe os atributos e as operações de outra classe. Mas a classe “herdeira” tem suas operações e atributos próprios. Na herança simples, uma classe pode herdar os atributos e as operações de uma única classe. Na chamada herança múltipla, uma classe pode herdar os atributos e as operações de mais de uma classe-pai. A vantagem da herança múltipla é maior capacidade de especificação de classes e a maior oportunidade de reutilização; e a desvantagem é a perda da simplicidade conceitual e de implementação.

**Certo.**

**016.** (FCC/AL-RN/ANALISTA LEGISLATIVO/2013) Um dos conceitos básicos de orientação a objetos é o fato de um objeto, ao tentar acessar as propriedades de outro objeto, deve sempre fazê-lo por uso de métodos do objeto ao qual se deseja atribuir ou requisitar uma informação, mantendo ambos os objetos isolados. A essa propriedade da orientação a objetos se dá o nome de

- a) herança.
- b) abstração.
- c) polimorfismo.
- d) mensagem.
- e) encapsulamento.



Este é o conceito de **encapsulamento**, também conhecido como **ocultamento de informações**, **que consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, dos detalhes internos da implementação daquele objeto**. O encapsulamento impede que um programa se torne tão dependente que uma pequena modificação possa causar grandes efeitos de propagação.

É importante ressaltar que ao observar o encapsulamento, o desenvolvedor terá maior facilidade em realizar as futuras alterações no software, conceito conhecido como Manutenibilidade.

**Letra e.**

---

**017.** (CESPE/BANCO DA AMAZÔNIA/TECNOLOGIA DA INFORMAÇÃO/2010) Julgue os itens seguintes de acordo com os conceitos do paradigma orientado a objetos. Objetos têm identidade própria. Isso garante que, mesmo tendo os mesmos valores de variáveis e pertencendo à mesma classe, dois objetos sejam considerados diferentes.



Na orientação a objetos, o conceito de **identidade** define que um objeto é uma instância única de uma classe, ocupando uma posição de memória específica. Então, podemos ter dois objetos não-idênticos (ocupam posições de memória distintas), mas iguais (são da mesma classe e possuem os mesmos valores para os atributos).

Além disso, cada objeto ao ser criado aloca espaço de memória para si e possui seus dados armazenados em estrutura própria. Não há confusão entre os objetos, especialmente quanto à identidade. Para simplificar o entendimento, podemos pensar em cada objeto como uma variável estruturada contendo os atributos.

**Certo.**

---

**018.** (CESPE/TRE-MT/TÉCNICO JUDICIÁRIO/2010) A estrutura interna de um objeto possui dois componentes básicos: atributos, que descrevem o estado do objeto; e métodos, que são responsáveis pela comunicação entre objetos.



Em orientação a objeto, um **método** é uma subrotina que é executada por um objeto ao receber uma mensagem. Os métodos determinam o comportamento dos objetos de uma classe e são análogos às funções ou procedimentos da programação estruturada. A comunicação entre os objetos é realizada por mensagens que um objeto envia a outro.

**Errado.**

---

**019.** (CESPE/BANCO DA AMAZÔNIA/TECNOLOGIA DA INFORMAÇÃO/2010) Na modelagem de classes, a hierarquia entre elas é representada por meio de um relacionamento chamado generalização.





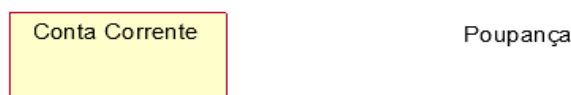
Explicando um pouco mais a **generalização**, ela é um **relacionamento entre um elemento geral e um outro mais específico**. O elemento mais específico possui todas as características do elemento geral e contém ainda mais particularidades. Um objeto mais específico pode ser usado como uma instância do elemento mais geral.

A **generalização**, também chamada de **herança**, permite a criação de elementos especializados em outros.

Na generalização normal a classe mais específica, chamada de **subclasse**, herda tudo que for público ou protegido da classe mais geral, chamada de **superclasse**. Os atributos, operações e todas as associações são herdados pela subclasse.

A generalização normal é representada por uma linha entre as duas classes que fazem o relacionamento, sendo que se coloca uma seta no lado da linha onde se encontra a superclasse indicando a generalização.

A figura a seguir apresenta um exemplo de generalização normal.



**Certo.**

**020.** (CESPE/CEHAP PB/ANALISTA DE SISTEMAS/2009) O projeto orientado a objetos encapsula atributos e serviços ou operações (comportamentos) em objetos. Os objetos comunicam-se entre si por meio de associações e os relacionamentos entre classes se dão por meio de interfaces.



Os objetos comunicam-se por meio de **mensagens (interface)** e os **relacionamentos** acontecem por meio de **associações**.

**Errado.**

**021.** (ESAF/MPOG/2008) Em Modelagem Orientada a Objetos, é correto afirmar que:

- a) uma classe deve representar uma abstração poliforme das propriedades dos objetos individuais que pertencem a um atributo.
- b) uma invariante de objeto é uma condição que toda classe desse objeto deve satisfazer para condições opcionais (quando o objeto está em equilíbrio).
- c) por meio do escopo de polimorfismo, uma classe é derivada diretamente de mais de uma subclasse.

d) uma classe com coesão de escopo tem algumas características que são indefinidas para algumas classes dos objetos.

e) por meio da herança múltipla, uma classe é derivada diretamente de mais de uma superclasse.



Os objetos são criados de acordo com uma definição de **classe** de objeto. Uma definição de classe de objeto funciona tanto como uma especificação quanto como um *template* para criação de objetos. Essa definição inclui declarações de todos os atributos e operações que devem ser associados a um objeto de classe. Por exemplo, a classe Aluno define alguns atributos que mantêm informações sobre os alunos como nome, idade, número de matrícula etc. As operações associadas ao objeto podem ser definidas quando um aluno é matriculado, ou uma chamada quando alguma informação do aluno precisa ser modificada.

Uma **invariante de objeto** significa que para utilizar determinado objeto será preciso que este esteja instanciado e seus atributos sejam informados.

O escopo de **polimorfismo** permite a manipulação de instâncias de classes que herdaram de uma mesma classe ancestral de forma unificada, por exemplo, duas classes filhas possuem os mesmos métodos da classe ancestral, porém esses métodos executam operações diferentes.

**Uma classe com coesão de escopo** permite a inter-relação dos recursos (atributos e operações) localizada na interface externa de uma classe (SANTOS, 2003).

Com isso podemos concluir que as letras A, B, C e D são falsas.

**Letra e.**

---

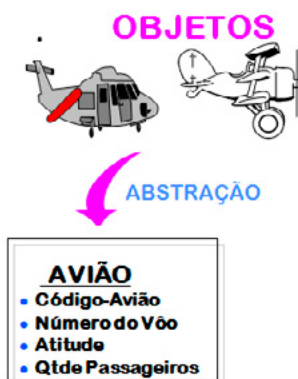
**022.** (ESAF/SEFAZ-CE/2007) A representação de classes em diagramas UML contempla os seguintes tipos básicos de informação:

- a) o nome da instância da classe e os seus objetos.
- b) o nome da classe, os seus atributos e os seus métodos.
- c) o nome da instância da classe e os seus relacionamentos.
- d) o nome da classe, os seus atributos e suas exceções.
- e) o nome da classe e suas visibilidades.



A **UML (Unified Modeling Language - Linguagem Unificada para Modelagem)** normalmente é definida como uma linguagem de modelagem e não um método propriamente dito. A UML apresenta uma série de diagramas para a modelagem de sistemas orientados a objetos. De todos os diagramas da UML, é o **diagrama de classes** o mais comumente utilizado pelas empresas. Esse diagrama, de forma simplificada, descreve os “tipos” de objetos do software e os vários tipos de relacionamentos estáticos que existem entre eles. Uma proposta de

processo de desenvolvimento que pode ser utilizada em conjunto é o RUP (*Rational Unified Process*), definido por Booch, Jacobson e Rumbaugh.

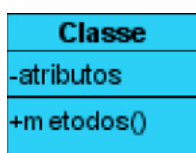


## Objetos

- Representam elementos do mundo real.
- É uma abstração de conjunto de coisas do mundo real.
- Possuem:
- **Atributos (estado)**
- **Operações (comportamento).**
- O único acesso aos dados desse objeto é por meio de suas **operações**.

## Classes

- Permitem que sejam representados no mundo computacional elementos do mundo real, ou seja, do problema para o qual o software está sendo desenvolvido.



- Elas permitem descrever um **conjunto de objetos** que compartilhem os mesmos atributos, operações, relacionamentos e semântica, e representam o principal bloco de construção de um software orientado a objetos.
- Representam os tipos de objetos existentes no modelo, e são descritas a partir de seus **atributos, métodos e restrições**.
- A figura seguinte apresenta a simbologia para uma **CLASSE** chamada ContaBancaria, utilizando a UML.



Figura. Classe ContaBancaria

Com as classes definidas, precisam-se especificar quais são seus **ATRIBUTOS (propriedades que caracterizam um objeto)**. Por exemplo, uma entidade conta bancária possui como atributos o número e o saldo. É bastante simples identificar os atributos de cada classe, basta identificar as **características que descrevam sua classe no domínio do problema em questão**. Cabe destacar que os atributos identificados devem estar alinhados com as necessidades do usuário para o problema.

A figura seguinte apresenta a classe ContaBancaria com alguns de seus atributos.

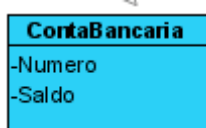


Figura. Classe ContaBancaria com alguns atributos

Identificadas as classes e seus atributos, o próximo passo é a identificação das **OPERAÇÕES** de cada classe, também chamadas de **métodos ou serviços**.

Fazendo um paralelo com objetos do mundo real, operações são **ações que o objeto é capaz de efetuar**. Dessa forma, ao procurar por operações, devem-se identificar ações que o objeto de uma classe é responsável por desempenhar dentro do escopo do sistema que será desenvolvido.

A figura seguinte apresenta algumas operações da classe ContaBancaria. Ao contrário dos atributos, normalmente operações são públicas, permitindo sua utilização por outras classes e objetos.

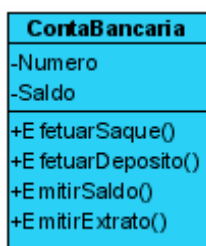


Figura. Classe ContaBancaria com seus atributos e operações

Finalizando, a UML (*Unified Modeling Language* – Linguagem Unificada para Modelagem) nos permite representar graficamente as classes, conforme nos mostrou a figura anterior, dando ênfase às **partes mais importantes de uma abstração: seu nome, atributos e operações**.  
**Letra b.**

**023.** (ESAF/TRF/2006) Um mesmo nome de objeto pode ser usado para identificar diferentes objetos em uma mesma classe ou diferentes objetos em classes diferentes, evitando assim,

que seja necessário usar nomes diferentes para objetos diferentes que realizam a mesma operação. A esta característica da Orientação a Objetos dá-se o nome de Polimorfismo.



O **polimorfismo** representa a **capacidade de poder existir métodos com assinaturas iguais, mas executando ações diferentes.**

**Errado.**

---

**024.** (ESAF/RECEITA FEDERAL/2005) O modo para descrever os vários aspectos de modelagem pela UML é por meio do uso da notação definida pelos seus vários tipos de diagramas. Segundo as características desses diagramas, é correto afirmar que um diagrama de classe

- a) mostra a interação de um caso de uso organizada em torno de objetos e classes e seus vínculos mútuos, evidenciando a sequência de mensagens.
- b) denota a estrutura estática de um sistema.
- c) descreve a funcionalidade do sistema.
- d) descreve a interação de sequência de tempo dos objetos e classes percebida por atores externos.
- e) mostra as sequências de estados que uma classe e objetos assumem em sua vida em resposta a estímulos recebidos, juntamente com suas respostas e ações.



- a) Errada. O **Diagrama de Classe** mostra as diferentes classes que fazem um sistema e como elas se relacionam. São chamados diagramas “estáticos” porque mostram as classes, com seus métodos e atributos bem como os relacionamentos estáticos entre elas.
- b) Certa. Como foi mencionada na letra anterior, os **Diagramas de Classe** são chamados diagramas “estáticos” porque mostram as classes, com seus métodos e atributos bem como os relacionamentos estáticos entre elas.
- c) Errada. O **diagrama de caso de uso** descreve a funcionalidade proposta para um novo sistema, que será projetado, ou seja, não está relacionado com o diagrama de classes.
- d) Errada. A interação de sequência de tempo dos objetos e classes está relacionada ao **diagrama de sequência**. É representando pela sequência de processos (mais especificamente, de mensagens passadas entre objetos). O diagrama de sequência descreve a maneira como os grupos de objetos colaboram em algum comportamento ao longo do tempo.
- e) Errada. O **diagrama de estados** mostra como um único objeto se comporta por meio de muitos casos de uso; mostra sequências de estados que um objeto ou uma interação assume em sua vida em resposta a eventos, juntamente com suas respostas e ações; é um complemento de uma classe relacionando os possíveis estados que objetos da classe

podem ter e quais eventos podem causar a mudança de estado (transição). Este então não está relacionado ao diagrama de classes.

**Letra b.**  
-----

**025.** (ESAF/AFRF/2005) Analise as seguintes afirmações relacionadas à análise e ao projeto orientados a objetos:

I – O principal propósito do diagrama entidade relacionamento (E-R) é representar os objetos e suas relações.

II – As tabelas de objetos de dados podem ser “normalizadas”, aplicandose um conjunto de regras de normalização, resultando em um “modelo relacional” para os dados. Uma dessas regras especifica que: determinada instância de um objeto tem um e somente um valor para cada atributo.

III – Um objeto em potencial não poderá ser utilizado ou considerado durante a análise se a informação sobre ele precisar ser lembrada para que o sistema possa funcionar.

IV – Devido à característica da reusabilidade da orientação a objetos, a prototipação é um modelo de desenvolvimento de software que não pode ser considerado nem utilizado na análise orientada a objetos.

Indique a opção que contenha todas as afirmações verdadeiras.

- a) I e III
- b) II e III
- c) III e IV
- d) I e II
- e) II e IV



I – Certo. O **diagrama de entidades** descreve o modelo de dados de um sistema com alto nível de abstração. Ele é a principal representação do modelo de entidades e relacionamentos. É usado para representar o modelo conceitual do negócio.

II – Certo. As tabelas de objetos podem ser “normalizadas” aplicandose um conjunto de regras de normalização, resultando num modelo relacional de dados. As regras de normalização, quando aplicadas às tabelas de objetos de dados, resultam em mínima redundância – ou seja, a quantidade de informações que precisamos manter para satisfazer um problema em particular é minimizada.

III – Errado. A identificação de objetos inicia-se ao examinar a declaração do problema ou ao executar uma “análise gramatical” da narrativa de processamento do sistema a ser construído. Objetos são determinados sublinhando-se cada nome ou cláusula nominal e colocandose numa tabela. O objeto em potencial será útil durante a análise somente se

a informação sobre ele precisar ser lembrada de forma que o sistema possa funcionar. Este deve ter um conjunto de operações identificáveis que podem mudar o valor de seus atributos de alguma forma. Esse conceito foi cunhado por Yourdon de informação retida, necessária para identificação de objetos, tornando-a falsa.

IV – Errado. **Prototipação** é uma abordagem baseada numa visão evolutiva do desenvolvimento de software, afetando o processo como um todo. Esta abordagem envolve a produção de versões iniciais, ou seja, protótipos (análogo a maquetes para a arquitetura) de um sistema futuro com o qual se pode realizar verificações e experimentações para se avaliar algumas de suas qualidades antes que o sistema venha realmente a ser construído.

**Letra d.**

---

**026.** (FCC/SABESP/TECNÓLOGO/2014) A engenharia de software apresenta um conjunto de princípios que podem ser usados quando um projeto de desenvolvimento de software for realizado, como os descritos abaixo:

I – Decomposição – o software é um produto complexo construído a partir de partes mais simples. A decomposição funcional é uma maneira de conceber o software como um conjunto de funções de alto nível (requisitos) que são decompostas em partes cada vez mais simples até chegar a comandos individuais de linguagem de programação.

II – Abstração – muitas vezes é necessário descrever um elemento em uma linguagem de nível mais alto do que o necessário para sua construção. A abstração ajuda os interessados no processo de desenvolvimento a entenderem estruturas grandes e complexas por meio de descrições mais abstratas.

III – Composição – a composição deu origem à orientação a objetos, em que um objeto pode ser classificado simultaneamente em mais de uma classe. Por exemplo, um cão, além de ser um mamífero, é animal e vertebrado.

IV – Padronização – a criação de padrões (*patterns*) de programação, design e análise ajuda a elaborar produtos com qualidade mais previsível. São importantes para a captação de experiências e evitam a repetição de erros que já têm solução conhecida.

Apresentam princípio e descrição corretos o que se afirma APENAS em

- a) I, II e III.
- b) IV.
- c) II e III.
- d) III e IV.
- e) I, II e IV.



A única afirmação incorreta é a III. Na **composição**, um objeto pode ser formado (composto) por outros objetos, como por exemplo, um carro composto por pneus, motor e chassi.

Um objeto NÃO pode ser de mais de uma classe ao mesmo tempo, pois assim perderia a sua identidade.

**Uma composição é um caso particular da agregação.** A agregação indica que uma das classes do relacionamento é uma parte, ou está contida em outra classe. As palavras chaves usadas para identificar uma agregação são: “consiste em”, “contém”, “é parte de”.

Existem dois tipos de agregação: Agregação Compartilhada e Agregação de Composição.

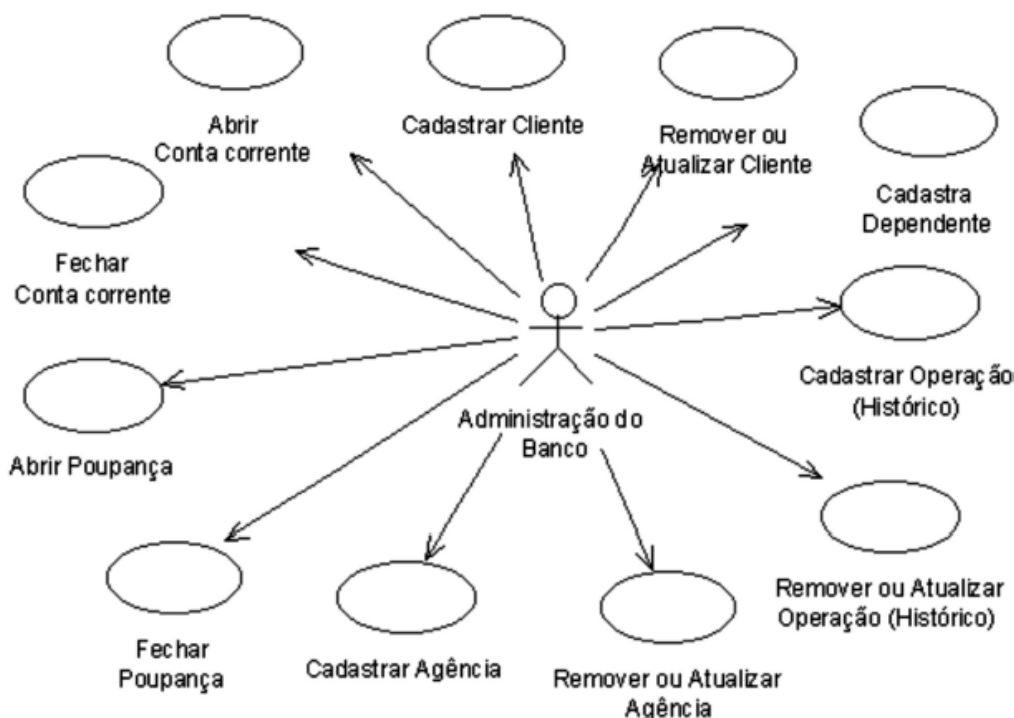
Uma **agregação é dita Compartilhada** quando uma das classes é uma parte, ou está contida na outra, mas esta parte pode estar contida na outra em um mesmo momento. A figura a seguir apresenta um exemplo deste tipo de agregação.



Uma Agregação de Composição é aquela classe que está contida na outra “vive” e constitui a outra. Se o objeto da classe que contém for destruído, as classes da agregação de composição serão destruídas juntamente já que as mesmas fazem parte da outra.

**Letra e.**

**027.** (FUNCAB/MP—RO/ANALISTA/2012) A figura abaixo representa o diagrama UML denominado:





- a) Diagrama de Classes.
- b) Diagrama de Atividade.
- c) Diagrama de Caso de Uso.
- d) Diagrama de Sequência.
- e) Diagrama de Componentes.



Questão fácil. Nela é apresentado um Diagrama de Casos de Uso (Use Case Diagram). Os Diagramas de Caso de Uso representam um conjunto de sequências de ações que um sistema desempenha para produzir um resultado observável de valor para um ator específico. Os casos de uso descrevem a funcionalidade do sistema percebida por **atores externos**. Um ator interage como o sistema, podendo ser um usuário, dispositivo ou outro sistema. O Diagrama de Casos de Uso é empregado visando:

- Definir Escopo: Um conjunto de Casos de Uso define o escopo do sistema de uma maneira simples.
- Organizar e dividir o trabalho: O Caso de Uso é uma importante unidade de organização do trabalho dentro do projeto. A unidade do Caso de Uso divide o trabalho da equipe entre as pessoas, fora isso, é comum dizer que o Caso de Uso está em Análise, em Programação ou em Teste. Casos de Uso também são entregues separadamente aos usuários em conjuntos divididos em fases ou iterações no projeto. Então, dizemos que a primeira iteração (ou entrega) terá os seguintes Casos de Uso e na segunda iteração terá os outros.
- Estimar o tamanho do projeto: O Caso de Uso fornece métricas para definir o tempo de desenvolvimento.
- Direcionar os testes: Os testes do sistema (essencialmente os funcionais que são os mais importantes) são derivados do Caso de Uso. A partir dos Casos de Uso, Casos de Teste são criados para validar o funcionamento do software.

Uma das questões em aberto sobre os Casos de Uso é a confusão que fazem com o diagrama e a narrativa (texto) do Caso de Uso. Isso porque a UML define somente como deve ser o Diagrama de Casos de Uso, e não a narrativa. Desse modo, não há um consenso geral sobre como descrever a narrativa, existem muitas técnicas e é difícil julgar que uma técnica é certa e a outra errada, depende muito do projeto, dos seus processos e ferramentas que você tem à disposição.

Componentes do Diagrama de Casos de Uso:

- Atores;
- Casos de Uso; e
- Associações (Inclusão, Extensão e Generalização).

## Atores

Um **Ator** é um modelo dos possíveis usuários, representando o seu papel. Modela uma categoria de usuários do sistema (usuário é uma instância de ator).

Para se identificar os Atores de um cenário de um sistema deve-se responder as seguintes perguntas:

- Quem utilizará a principal funcionalidade do sistema (atores principais)?
- Quem provê serviço externo ao sistema (atores secundários)?
- Quem proverá suporte ao sistema em seu processamento diário?
- Quem ou o quê tem interesse nos resultados produzidos pelo sistema?
- Com quais outros sistemas o sistema irá interagir?

## Casos de Uso

Um **Caso de uso** é uma situação específica do uso do sistema, que constitui um curso completo de eventos iniciado por algum ator e/ou especifica a interação entre o ator e o sistema. Características principais dos casos de uso:

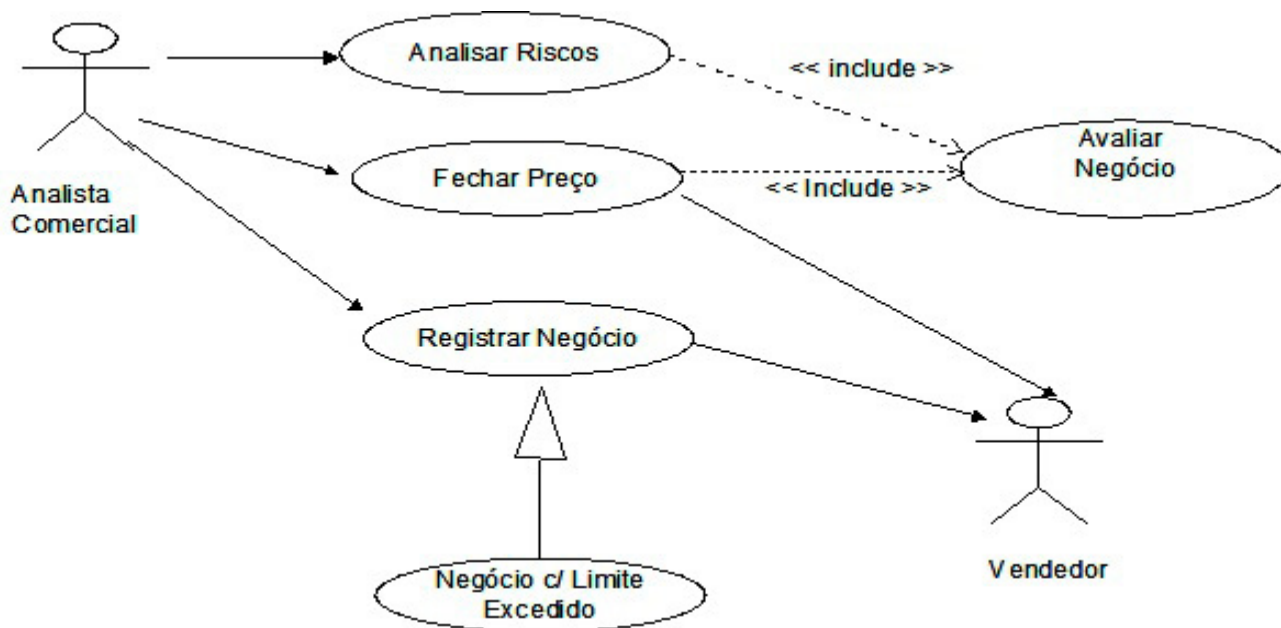
- Um caso de uso é sempre iniciado por um ator – Um caso de uso é sempre realizado em nome de um ator que, por sua vez, deve pedir direta ou indiretamente ao sistema tal realização;
- Um caso de uso é completo – Um caso de uso deve ser uma descrição completa, portanto, não estará completo até que o valor final seja produzido mesmo se várias comunicações ocorrerem durante uma interação; e
- Um caso de uso provê valor a um ator – Um caso de uso deve prover um valor tangível a um ator em resposta à sua solicitação.

Para se identificar os Casos de Uso de um cenário de um sistema devem-se analisar as ações do sistema neste cenário, tais como:

- O ator precisa ler, criar, destruir, modificar ou armazenar algum tipo de informação no sistema?
- O trabalho cotidiano do ator pode ser simplificado ou tornado mais eficiente por meio de novas funções no sistema?
- O ator tem de ser notificado sobre eventos no sistema ou ainda notificar o sistema em si? Quais são as funções que o ator necessita do sistema?
- O que o ator necessita fazer?
- Quais são os principais problemas com a implementação atual do sistema?
- Quais são as entradas e as saídas, juntamente com a sua origem e destino, que o sistema requer?

## Associações

Existem as seguintes **associações** entre casos de uso:



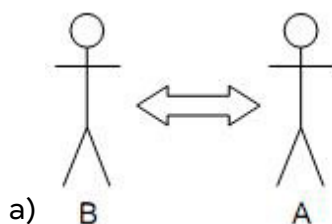
**Inclusão:** um caso de uso inclui outro, se um passo do mesmo “chama” o outro caso de uso. Pode-se representar também que uma parte do comportamento é semelhante em mais de um caso de uso. No exemplo anterior, tanto “Analisar Risco” quanto “Fechar Preço” requerem a função “Avaliar Negócio”.

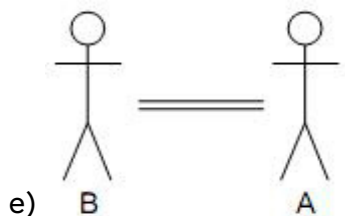
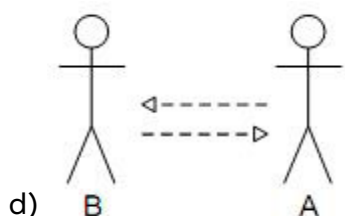
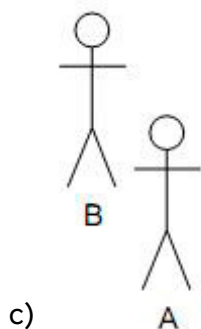
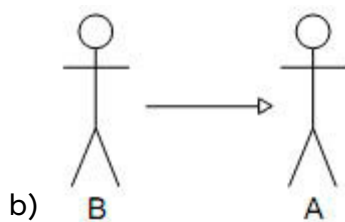
**Generalização:** Um caso de uso é semelhante a outro, mas faz um pouco a mais, apresentando pequenas diferenças. No exemplo, “Negócio com Limites Excedidos” realiza uma função a mais que “Registrar Negócio”.

**Extensão:** descreve situações opcionais, que somente ocorrerão se determinada condição for satisfeita, a qual interrompe a execução do caso de uso. Expressa uma variação do comportamento normal, mas aplicada de forma mais controlada. Pode-se também declarar os “pontos de extensão”. Pode-se também representar serviços assíncronos que o usuário pode ativar para interromper o caso de uso base.

**Letra c.**

**028.** (VUNESP/CÂMARA MUNICIPAL DE SÃO JOSÉ DOS CAMPOS-SP/ANALISTA LEGISLATIVO/2014)  
Deseja-se representar, em um diagrama de casos de uso da UML, a seguinte situação: o ator B herda as propriedades do ator A. A forma correta de representação é

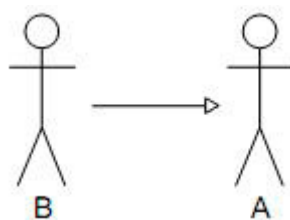




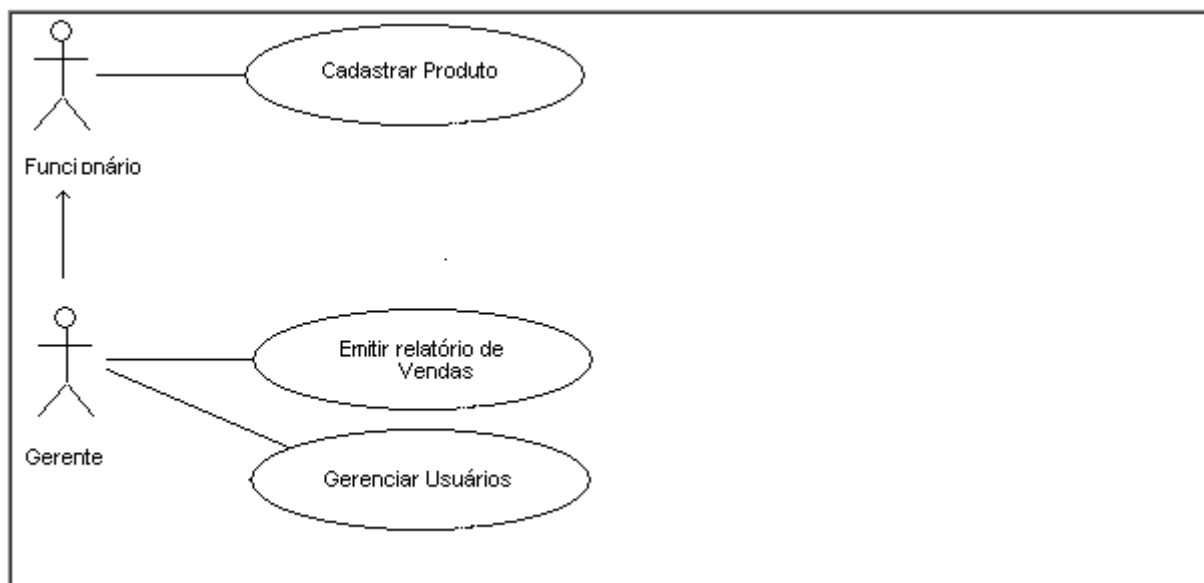
Características dos diagramas de Casos de Uso:

- mostram um conjunto de casos de uso, atores e seus relacionamentos;
- indicam como o sistema interage com as entidades externas (atores);
- fornecem uma representação contextual do sistema.

A forma correta para representar que o ator B herda as propriedades do ator A é a que está listada em:



A seguir, tem-se um exemplo em que o Ator gerente acessa os casos de uso do ator funcionário.



**Letra b.**

**029.** (VUNESP/SP-URBANISMO/ANALISTA/2014) Considere o seguinte símbolo de um diagrama de classes da orientação a objetos:



Tal símbolo representa

- a) um port.
- b) um subtipo.
- c) um comentário.
- d) uma subclasse.
- e) uma especialidade.



A figura apresenta um **comentário** no padrão UML. **Comentários** são incluídos nos diagramas para descrever, esclarecer ou fazer observações sobre outros elementos dos modelos. Além dos comentários, a UML permite também a inclusão de **Mecanismos de Extensibilidade** para incrementar as informações de um modelo.

**Os Mecanismos de Extensibilidade da UML permitem que os elementos do modelo possam ser modificados e estendidos adicionando-se uma nova semântica aos diagramas.**

Os **Mecanismos de Extensibilidade** são:

- **Estereótipos:** permitem a inclusão de informações adicionais sobre os elementos dos diagramas. Um estereótipo é apresentado entre os símbolos << >>.

#### EXEMPLO

<<abstract>> para definir uma classe abstrata. A UML apresenta um conjunto de estereótipos já definidos, podendo-se definir outros adicionais.

- **Valores atribuídos (Tagged Values):** permitem atribuir um valor particular a uma propriedade, no formato {valor atribuído = valor}.

#### EXEMPLO

{estoque = 0}, exemplo de uma pré-condição para a execução de um método.

- **Restrições:** definem condições ou proposições que devem ser mantidas verdadeiras. Uma restrição é mostrada como uma expressão entre chaves {}.

#### EXEMPLO

{ordenado}, para indicar que uma lista deverá ser mantida ordenada.

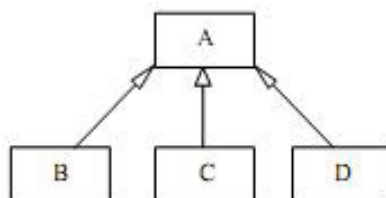
**Um conjunto de mecanismo específico pode definir um Perfil.** Os Perfis permitem a adição de construções específicas de um domínio, plataforma ou método em particular.

#### EXEMPLO

Perfil para modelagem de dados geográficos.

**Letra c.**

**030.** (VUNESP/DESENVOLVESP/ANALISTA DE SISTEMAS/2014) Considere o seguinte diagrama de classes da UML 2.0:



Com base nesse diagrama, é correto afirmar que

- a) a classe A é um tipo especial das classes B, C e D.
- b) as classes A, B, C e D são abstratas.
- c) as classes B, C e D são subclasses da classe A.
- d) está representada uma agregação.
- e) está representada uma composição.

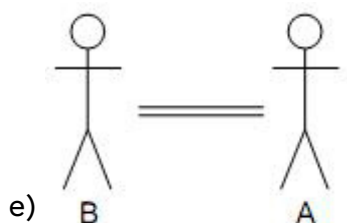
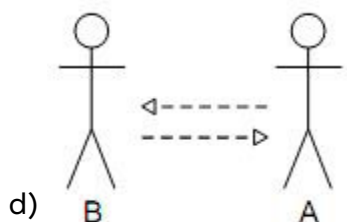
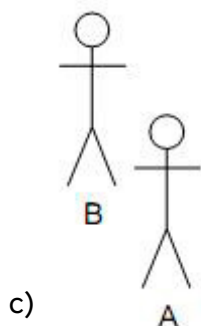
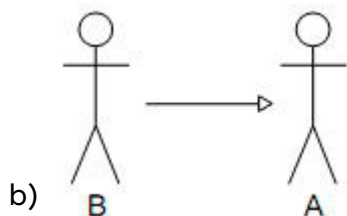
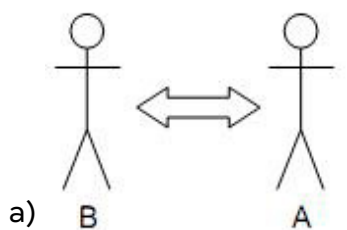


Com base nesse diagrama, cabe destacar que as classes B, C e D são subclasses da classe A.

**Letra c.**

**031.** (VUNESP/CÂMARA MUNICIPAL DE SÃO JOSÉ DOS CAMPOS-SP/ANALISTA LEGISLATIVO/2014)

Deseja-se representar, em um diagrama de casos de uso da UML, a seguinte situação: o ator B herda as propriedades do ator A. A forma correta de representação é

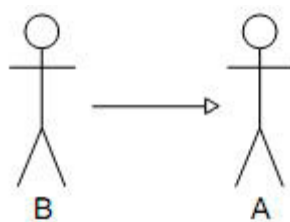




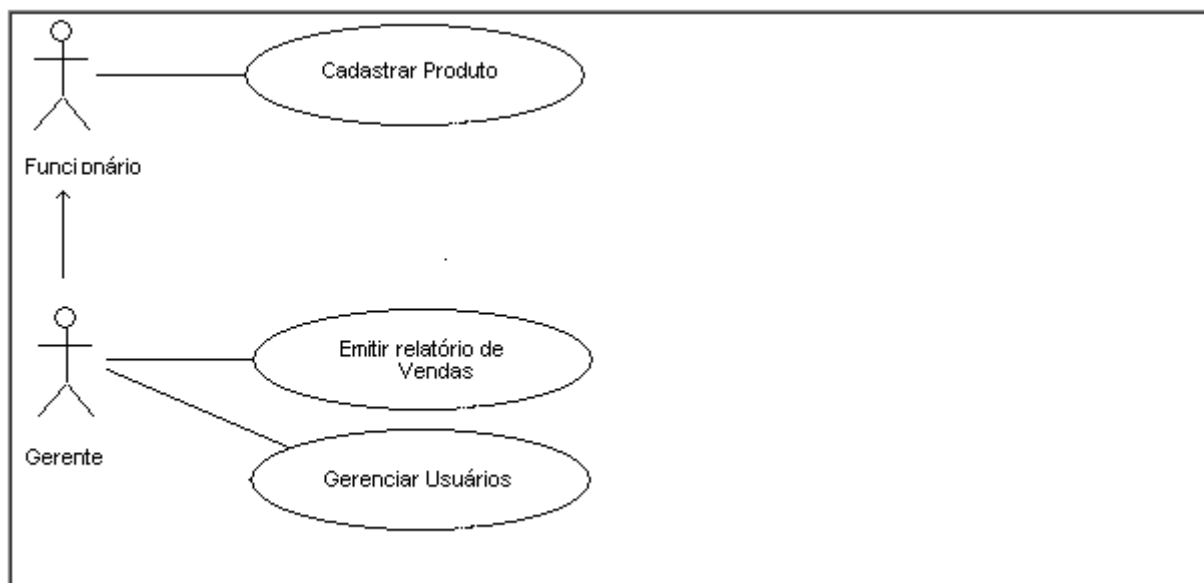
### Características dos diagramas de Casos de Uso:

- mostram um conjunto de casos de uso, atores e seus relacionamentos;
- indicam como o sistema interage com as entidades externas (atores);
- fornecem uma representação contextual do sistema.

A forma correta para representar que o ator B herda as propriedades do ator A é a que está listada em:

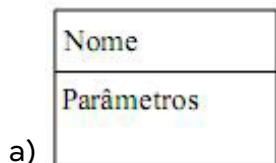


A seguir, tem-se um exemplo em que o Ator gerente acessa os casos de uso do ator funcionário.

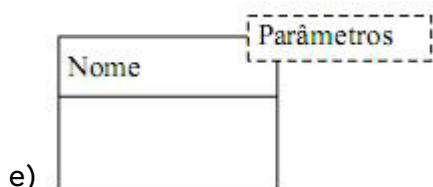
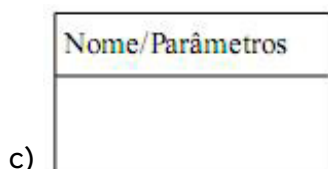
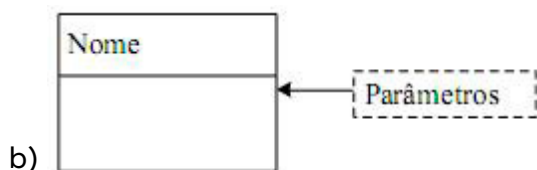


### Letra b.

**032.** (VUNESP/EMPLASA/ANALISTA ADMINISTRATIVO/2014) Na UML 2.0, a notação utilizada para representar um template de classe é:







De acordo com o Guia do Usuário UML, uma classe template pode ser representada como qualquer outra classe, **mas com uma caixa tracejada adicional no canto superior direito do ícone da classe, listando os parâmetros para o template.**

Veja mais:

<http://www.uml-diagrams.org/template.html>

<http://oengenheirodesoftware.blogspot.com.br/2010/11/como-representar-class-templates-ou.html>

**Letra e.**

**033.** (VUNESP/EMPLASA/ANALISTA ADMINISTRATIVO/2014) No diagrama de objetos da UML 2.0, o nome de um objeto deve respeitar a seguinte notação:

- a) nome-classe >> nome-objeto
- b) nome-classe (nome-objeto)
- c) nome-objeto < > nome-classe
- d) nome-objeto (nome-classe)
- e) nome-objeto: nome-classe



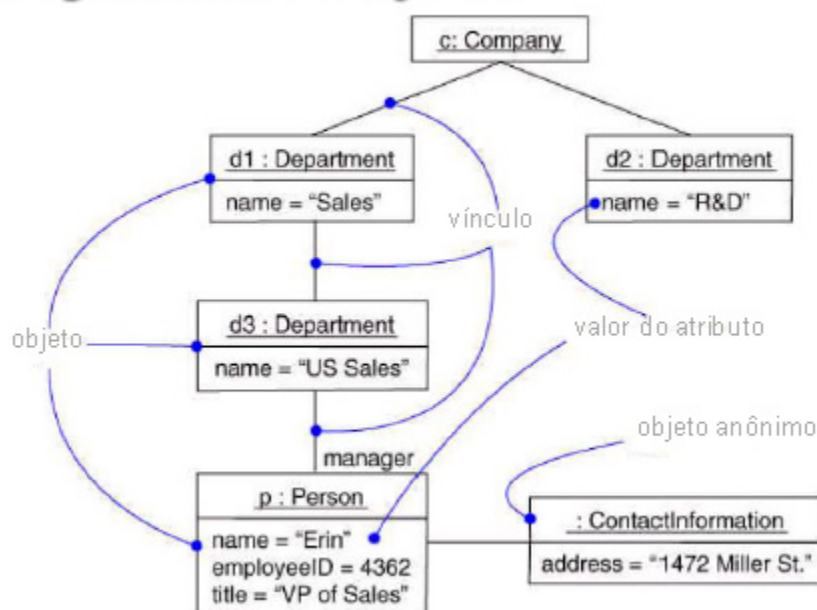
No diagrama de objetos da UML 2.0, o nome de um objeto deve respeitar a seguinte notação **nome-objeto: nome-classe.**

**Nota:**

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











Veja um exemplo:

## Diagrama de Objetos



**Letra e.**

**034.** (VUNESP/EMPLASA/ANALISTA ADMINISTRATIVO/2014) Em um diagrama de máquinas de estado da UML 2.0, os estados inicial e final são representados, respectivamente, pelos símbolos:

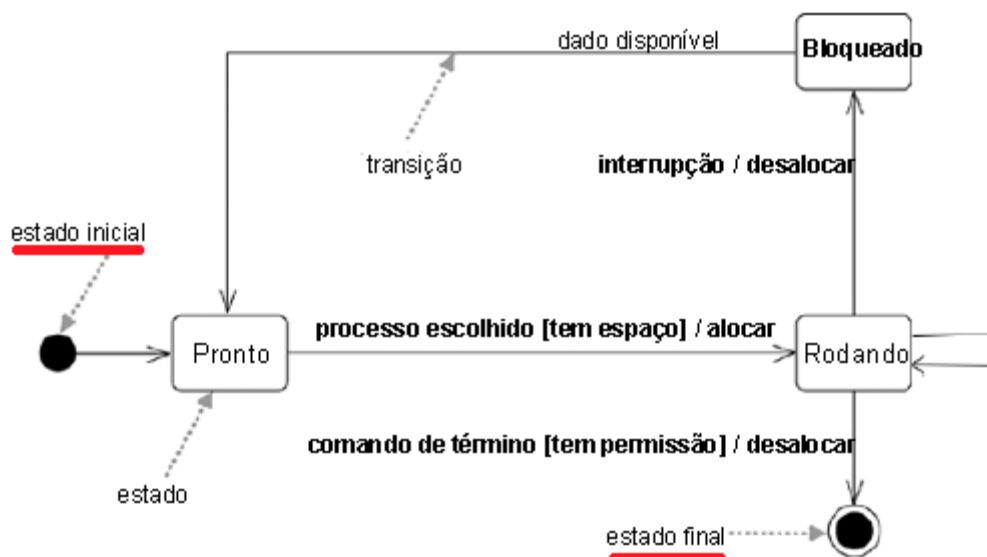
- a)  e 
- b)  e 
- c)  e 
- d)  e 
- e)  e 



O **Diagrama de Máquina de Estados** mostra os vários estados possíveis pelos quais um objeto pode passar. São elementos desse diagrama:

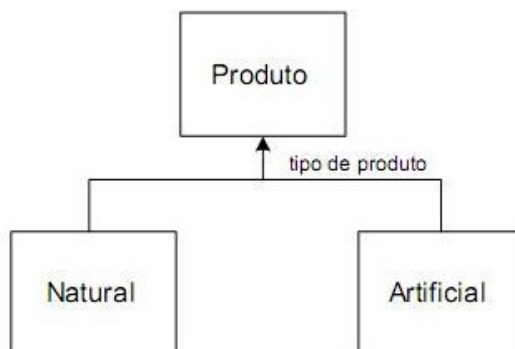
<b>Estados</b>	<ul style="list-style-type: none"> <li>Situações na vida de um objeto na qual ele satisfaz uma condição ou realiza alguma atividade.</li> </ul>
<b>Transições</b>	<ul style="list-style-type: none"> <li>Estados são associados por meio de transições.</li> <li>Transições têm eventos associados.</li> </ul> <p>Sintaxe: <b>evento [condição]/ação</b></p>
<b>Ações</b>	<ul style="list-style-type: none"> <li>Ao passar de um estado para o outro o objeto pode realizar ações.</li> </ul>
<b>Atividades</b>	<ul style="list-style-type: none"> <li>Executadas durante um estado.</li> </ul>

Exemplo de Diagrama de Máquina de Estados para Escalonamento de Processos:



Letra c.

**035.** (VUNESP/DCTA/TECNOLOGISTA PLENO/2013) Considere o seguinte diagrama de classes da UML 2.0.



O texto “tipo de produto” é chamado de um(a)

- a) estado.
- b) interface.
- c) acoplamento.
- d) discriminador.
- e) compartimento.

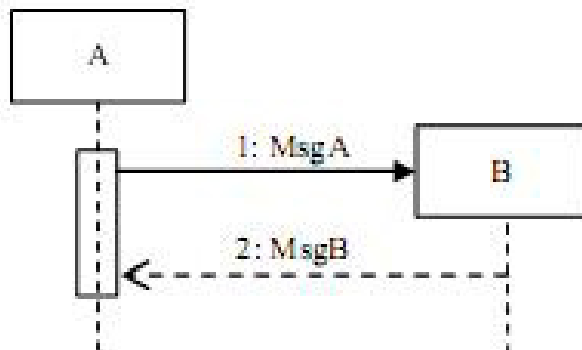


O texto “tipo de produto” é chamado de um **discriminador**.

Veja uma excelente referência para o estudo da UML: <https://books.google.com.br/books?id=xxoXcuh0°SOC&pg=PA86&lpg=PA86&dq=UML+2+%2B+DIAGRAMA+DE+classes+%2B+discriminador&source=bl&ots=us0H3FIJ6u&sig=O65Y-rQtvfTlR2trPVEi2VgijY&hl=pt-BR&sa=X&ei=kOylVOSFI4OgNsaEhKgD&ved=OCCMQ6AEwAQ#v=onepage&q=UML%202%20%2B%20DIAGRAMA%20DE%20classes%20%2B%20discriminador&f=false>

**Letra d.**

**036.** (VUNESP/DESENVOLVESP/ANALISTA DE SISTEMAS/2014) Considere o seguinte diagrama de mensagens (ou de sequência), representado por meio da UML 2.0:



Com base nesse diagrama, é correto afirmar que

- a) ambos os objetos (A e B) já existiam antes da geração da mensagem MsgA.
- b) o objeto A é criado em consequência da mensagem MsgB.
- c) o objeto B é armazenado em um banco de dados.
- d) o objeto B é criado em consequência da mensagem MsgA.
- e) o objeto B é destruído após a geração da mensagem MsgB.



Ao final da mensagem originada no objeto A (MsgA), o objeto B é iniciado. Logo, existe a criação deste objeto. Neste tipo de situação pode-se acrescentar o estereótipo <<create>> à mensagem para indicar essa criação.

**Letra d.**

## REFERÊNCIAS

BARNES, D. *Programação Orientada a Objetos com Java*. 4 ed. São Paulo: Prentice Hall, 2009.

CAELUM. *Java e Orientação a Objetos*. Disponível em: <<http://www.alura.com.br/apostila-java-orientacao-objetos/>> Acesso em: 05 mar. 2021.

CÉSAR, B. *Modelos de desenvolvimento de software: a Catedral e o Bazar*. Disponível em: <<http://www.brod.com.br/node/536>>. Acesso em: 11 jul. 2021.

CUKIER, D. *DDD – Introdução a Domain Driven Design*. 2010. Disponível em: <http://www.agileandart.com/2010/07/16/ddd-introducao-a-domain-driven-design/>>. Acesso em: jun. 2021.

DEITEL, Paul. *Java Como Programar*. 8ª Edição. São Paulo: Prentice Hall, 2010.

DEVMEDIA. *Principais conceitos da Programação Orientada a Objetos*. 2015. Disponível em: <<https://www.devmedia.com.br/principais-conceitos-da-programacao-orientada-a-objetos/3228>>. Acesso em: 01 abr. 2020.

\_\_\_\_\_. *Conceitos e Exemplos – Polimorfismo: Programação Orientada a Objetos*. 2020. Disponível em: <<https://www.devmedia.com.br/conceitos-e-exemplos-polimorfismo-programacao-orientada-a-objetos/18701>>. Acesso em: 21 ago. 2022.

FREEMAN, E.; ROBSON, E.; BATES, B.; SIERRA, K. *Head First Design Patterns: A Brain-Friendly Guide*. O'Reilly, 2004.

GUERRA, E. *Design Patterns com Java: Projeto Orientado a Objetos guiado por Padrões. Caso do Código*, 2014.

IBM. Disponível em: <[https://www.ibm.com/developerworks/community/blogs/ctaurion/entry/evolucao\\_do\\_open\\_source?lang=en](https://www.ibm.com/developerworks/community/blogs/ctaurion/entry/evolucao_do_open_source?lang=en)>. Acesso em: 05 dez. 2021.

KOSCIANSKI, A.; SOARES, M. S. *Qualidade de Software*. São Paulo: Editora Novatec, 2007.

KRUCHTEN, P. *Introdução ao RUP Rational Unified Process*. 2. ed., Rio de Janeiro: Ciência Moderna, 2003.

LARMAN, Craig. *Utilizando UML e Padrões*. Bookman Editora, 2002.

ORACLE. *Object-Oriented Programming Concepts*. Disponível em: <<https://docs.oracle.com/javase/tutorial/java/concepts/index.html>>. Acesso em: 11 abr. 2020.

PRESSMAN, R. S. *Engenharia de Software: Uma Abordagem Profissional*, 7. ed. Porto Alegre: Editora Mc GrawHill, 2011.

PFLEEGER, S. L. *Engenharia de Software: Teoria e Prática*. 2.ed., São Paulo: Prentice Hall, 2004.

QUINTÃO, P. L. *Notas de aula da disciplina "Tecnologia da Informação"*. 2023.

SANTOS, R. *Introdução à programação orientada a objetos usando Java*. Campus, 2003.

SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo: Pearson Addison - Wesley, 2007.

\_\_\_\_\_. *Engenharia de Software*. 9. ed. São Paulo: Pearson Addison - Wesley, 2011.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. 2020. Disponível em: <<https://engsoftmoderna.info/>>. Acesso em: 20 de abril de 2023.

WEST, D.; POLLICE, G. *Use a Cabeça! Análise e Projeto Orientado ao Objeto*. Alta Books, 2007.

Abra



caminhos



crie

futuros

gran.com.br

