

**Disciplina: Análise e Projeto Orientado a Objetos: UML**

# **Apostila de Análise e Projeto Orientado a Objetos: UML**

## SUMÁRIO

1.1 Modelagem de sistemas de software .....	5
1.2 O paradigma da orientação a objetos .....	6
1.3 Evolução histórica da modelagem de sistemas.....	13
1.4 A Linguagem de modelagem unificada .....	13
2 Processo de Desenvolvimento de Software .....	16
2.10 Atividades típicas de um PDS. 2.2 O componente humano em um PDS.....	16
2.3 Modelos de ciclo de vida.....	17
2.4 Utilização da UML no modelo iterativo e incremental .....	19
2.5 Prototipagem.....	19
2.6 Ferramentas de suporte .....	20
3 Mecanismos Gerais.....	20
3.1 Estereótipos .....	21
3.2 Notas explicativas .....	21
3.3 Etiquetas (Tags) .....	21
3.4 Restrições.....	22
3.5 Pacotes .....	22
3.6 OCL .....	23
4 Modelagem De Casos De Uso .....	24
4.2 Diagrama de casos de uso .....	28
4.3 Identificação dos elementos do MCU .....	31
4.4 Construção do MCU.....	32
4.5 Documentação suplementar ao MCU .....	34
4.6 O MCU em um processo de desenvolvimento iterativo e incremental .....	35
5 Modelagem de Classes de Análise .....	36
5.2 Diagrama de classes.....	38
5.3 Diagrama de objetos.....	47
5.4 Técnicas para identificação de classes .....	48
5.5 Construção do modelo de classes .....	58
5.6 Modelo de classes no processo de desenvolvimento.....	59
6 Passando da análise ao projeto.....	59
7 Modelagem de Interações .....	60
7.2 Diagrama de seqüência .....	65
7.3 Diagrama de comunicação.....	67
7.4 Modularização de interações .....	68
7.5 Construção do modelo de interações .....	70
7.6 Modelo de interações em um processo iterativo .....	75
8 Modelagem de classes de projeto .....	76
8.1 Transformação de classes de análise em classes de projeto.....	76
8.2 Especificações de atributos .....	79
8.3 Especificações de operações.....	79
8.4 .....	Especificação de associações
.....	81
8.5 .....	Herança
.....	85
8.6 .....	Padrões de projeto
.....	91

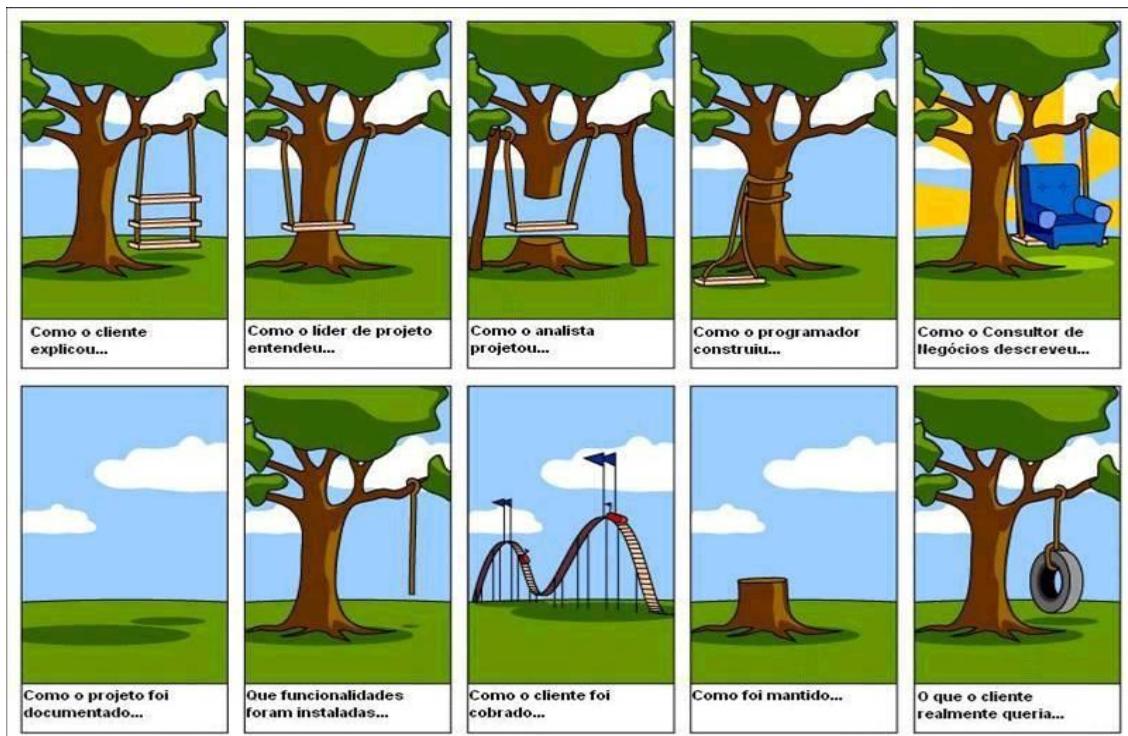
<b>9 Modelagem de estados .....</b>	<b>93</b>
<b>9.1 Diagramas de transição de estados .....</b>	<b>93</b>
<b>9.2 Identificação dos elementos de um diagrama de estados .....</b>	<b>99</b>
<b>9.3 Construção de diagramas de transição de estados .....</b>	<b>100</b>
<b>9.4 Modelagem de estados no processo de desenvolvimento .....</b>	<b>100</b>
<b>10 Modelagem de atividades .....</b>	<b>101</b>
<b>Diagrama de atividade no processo de desenvolvimento iterativo.....</b>	<b>103</b>
<b>11 Arquitetura do sistema.....</b>	<b>105</b>
<b>11.1 Arquitetura lógica .....</b>	<b>106</b>
<b>11.2 Implantação física.....</b>	<b>109</b>
<b>11.3 Projeto da arquitetura no processo de desenvolvimento .....</b>	<b>113</b>
<b>12 Mapeamento de objetos para o modelo relacional .....</b>	<b>114</b>
<b>12.1 Projeto de banco de dados .....</b>	<b>115</b>
<b>12.2 Construção da camada de persistência.....</b>	<b>121</b>

Livro-Texto  
**Princípios de Análise e Projeto de Sistemas com UML**  
2<sup>a</sup> edição  
Eduardo Bezerra  
Editora Campus/Elsevier

Visão Geral

*“Coisas simples devem ser simples e coisas complexas devem ser possíveis.”. -Alan Kay*

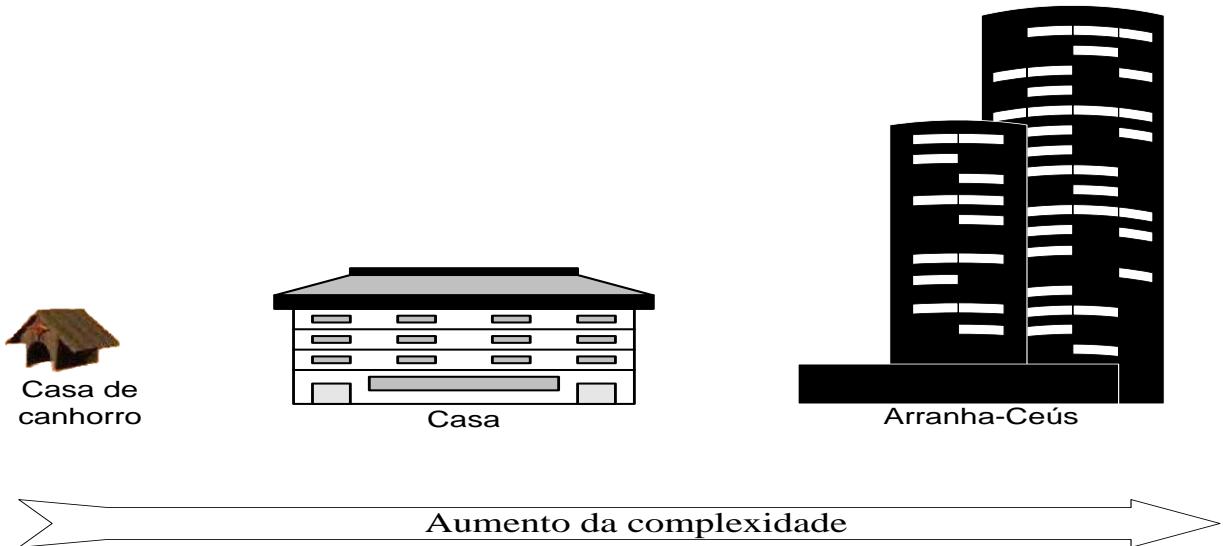
*Várias visões de um mesmo problema*



### Sistemas de Informações

- A necessidade é a mãe das invenções
  - Em consequência do crescimento da importância da informação, surgiu a necessidade de gerenciar informações de uma forma adequada e eficiente e, desta necessidade, surgiram os denominados *sistemas de informações*.
- Um SI é uma combinação de pessoas, dados, processos, interfaces, redes de comunicação e tecnologia que interagem com o objetivo de dar suporte e melhorar o processo de negócio de uma organização com relação às informações.
  - Vantagens do ponto de vista competitivo.

- O Principal objetivo da construção de um SI é a *adição de valor à organização*.
- Um dos componentes de um SI é denominado **sistema de software**.
- Compreende os módulos funcionais computadorizados que interagem entre si para proporcionar a automatização de diversas tarefas.
- Característica intrínseca do desenvolvimento de sistemas de software: **complexidade**.



### 1.1 Modelagem de sistemas de software

#### Modelos de Software

- Na construção de sistemas de software, assim como na construção de sistemas habitacionais, também há uma graduação de complexidade.
  - A construção desses sistemas necessita de um planejamento inicial.
- Um modelo pode ser visto como uma representação idealizada de um sistema que se pretende construir.
- Maquetes de edifícios e de aviões e plantas de circuitos eletrônicos são apenas alguns exemplos de modelos.

#### Razões para construção de modelos

- A princípio, podemos ver a construção de modelos como uma atividade que atrasa o desenvolvimento do software propriamente dito.
- Mas essa atividade propicia...
  - O **gerenciamento da complexidade** inerente ao desenvolvimento de software.
  - A **comunicação** entre as pessoas envolvidas.
  - A **redução dos custos** no desenvolvimento.
  - A **predição do comportamento** futuro do sistema.
- Entretanto, note o fator **complexidade** como condicionante dessas vantagens.

#### Diagramas e Documentação

- No contexto de desenvolvimento de software, correspondem a desenhos gráficos que seguem algum padrão lógico.
- Podemos também dizer que um diagrama é uma apresentação de uma coleção de elementos gráficos que possuem um significado predefinido.
- Diagramas normalmente são construídos de acordo com regras de notação bem definidas.
  - Ou seja, cada forma gráfica utilizada em um diagrama de modelagem tem um significado específico.

Diagramas permitem a construção de uma representação concisa de um sistema a ser construído .

“uma figura vale por mil palavras”



No entanto, modelos também são compostos de informações textuais.

Dado um modelo de uma das perspectivas de um sistema, diz- se que o seu diagrama, juntamente com a informação textual associada, formam a *documentação* deste modelo.

### Modelagem de Software

A modelagem de sistemas de software consiste na utilização de notações gráficas e textuais com o objetivo de construir modelos que representam as partes essenciais de um sistema, considerando-se diversas perspectivas diferentes e complementares.

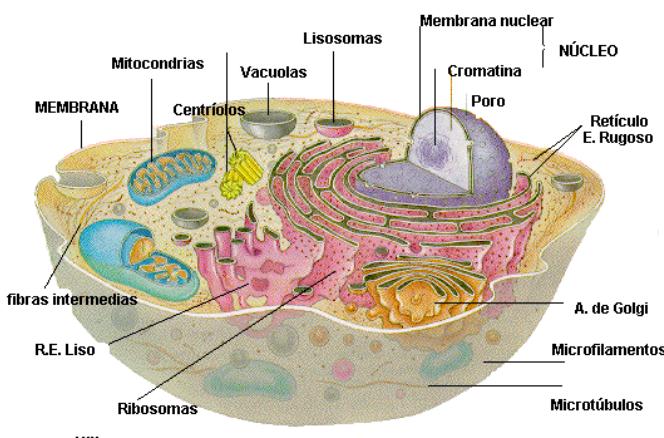
## 1.2 O paradigma da orientação a objetos

- *Um paradigma é uma forma de abordar um problema.*
- No contexto da modelagem de um sistema de software, um paradigma tem a ver com a forma pela qual esse sistema é entendido e construído.
- A primeira abordagem usada para modelagem de sistemas de software foi o ***paradigma estruturado***.
  - Uso da técnica de *decomposição funcional*
  - “divida sucessivamente um problema complexo em subproblemas”
- Hoje em dia, praticamente suplantou o paradigma anterior, o ***paradigma da orientação a objetos...***
- O paradigma da OO surgiu no fim dos anos 60.
- Alan Kay, um dos pais desse paradigma, formulou a chamada ***analogia biológica***.
- “*Como seria um sistema de software que funcionasse como um ser vivo?*”



## Analogia Biológica

- Cada “célula” interagiria com outras células através do envio de mensagens para realizar um objetivo comum.
- Adicionalmente, cada célula se comportaria como uma unidade autônoma.
- De uma forma mais geral, *Kay* pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si.



## Fundamentos da Orientação a Objetos

- Através de sua analogia biológica, *Alan Kay* definiu os fundamentos da orientação a objetos.
  1. Qualquer coisa é um objeto.
  2. Objetos realizam tarefas através da requisição de serviços a outros objetos.
  3. Cada objeto pertence a uma determinada *classe*. Uma classe agrupa objetos similares.
  4. A classe é um repositório para comportamento associado ao objeto.
  5. Classes são organizadas em hierarquias.

- Conceitos
  - Classe
  - Objeto
  - Mensagem
- Princípios
  - Encapsulamento
  - Polimorfismo
  - Generalização (Herança)
  - Composição
- O paradigma da orientação a objetos visualiza um sistema de software como uma coleção de agentes interconectados chamados objetos. Cada objeto é responsável por realizar tarefas específicas. / através da interação entre objetos que uma tarefa computacional é realizada.
- Um sistema de software orientado a objetos consiste de objetos em colaboração com o objetivo de realizar as funcionalidades deste sistema. Cada objeto é responsável por tarefas específicas. / através da cooperação entre objetos que a computação do sistema se desenvolve.

## Conceitos e Princípios da OO

- Conceitos
  - Classe
  - Objeto
  - Mensagem
- Princípios
  - Encapsulamento
  - Polimorfismo
  - Generalização (Herança)
  - Composição

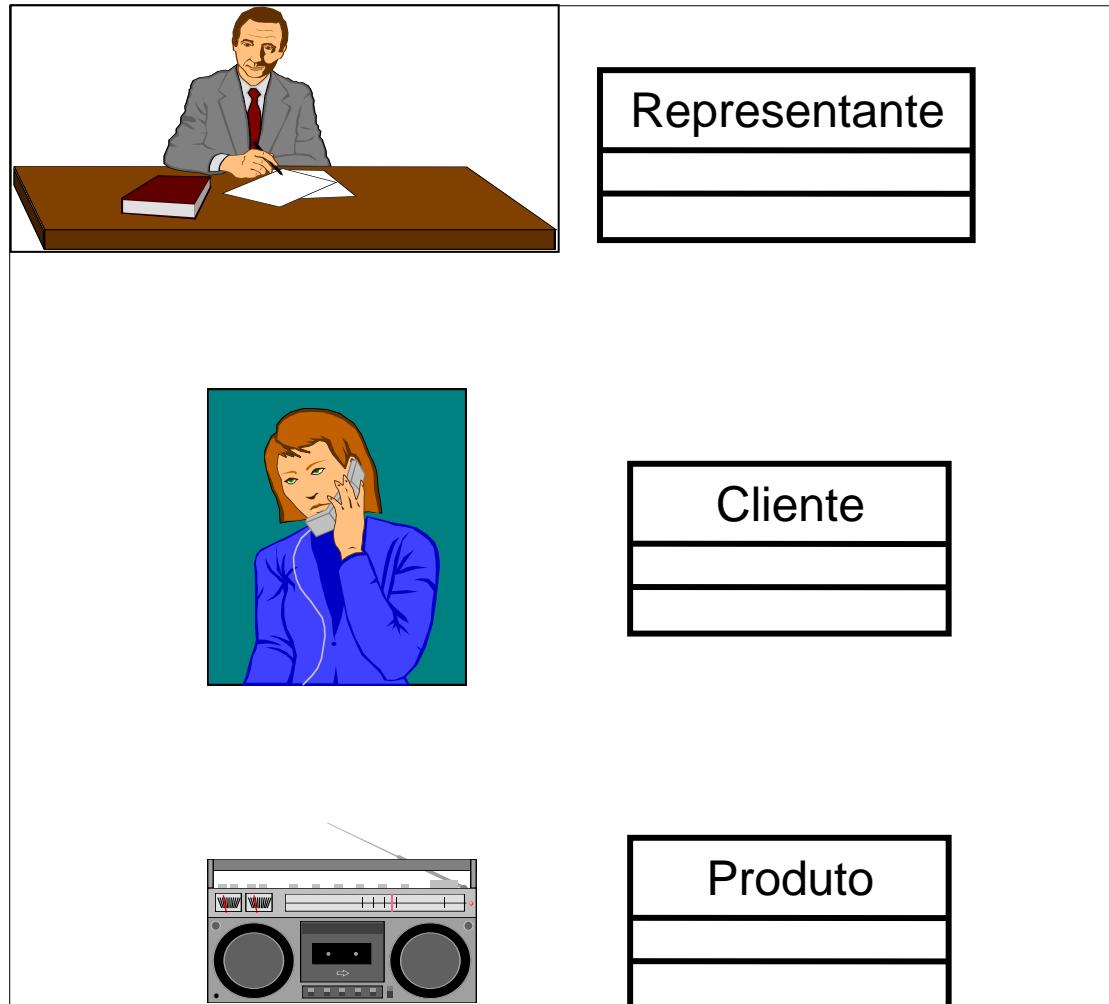
## Classes, objetos e mensagens

- O mundo real é formado de coisas.
- Na terminologia de orientação a objetos, estas coisas do mundo real são denominadas *objetos*.
- Seres humanos costumam agrupar os objetos para entendê-los.
- A descrição de um grupo de objetos é denominada ***classe de objetos***, ou simplesmente de ***classe***.

### O que é uma classe?

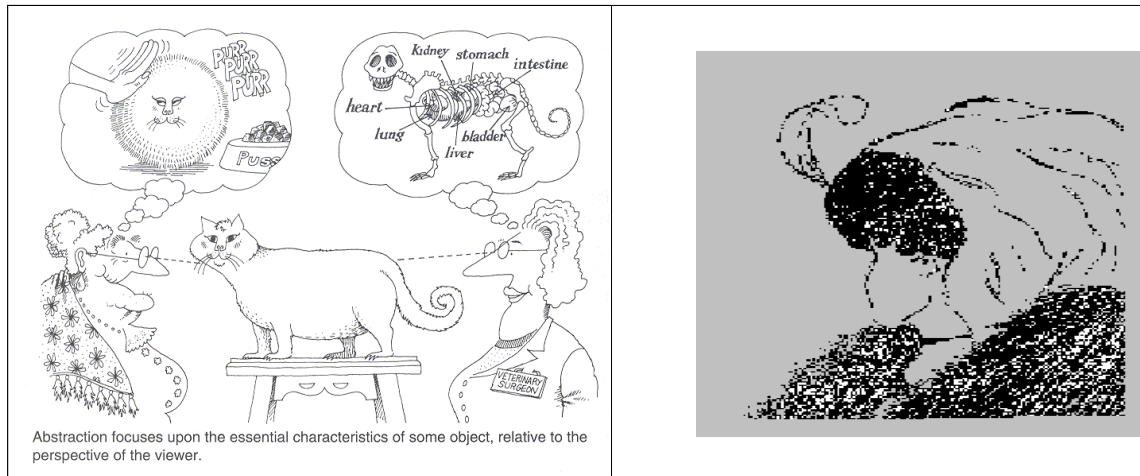
- Uma classe é um moldé para objetos. Diz-se que um objeto é uma instância de uma classe.
- Uma classe é uma *abstração* das características *relevantes* de um grupo de coisas do mundo real.

- Na maioria das vezes, um grupo de objetos do mundo real é muito complexo para que *todas* as suas características e comportamento sejam representados em uma classe.



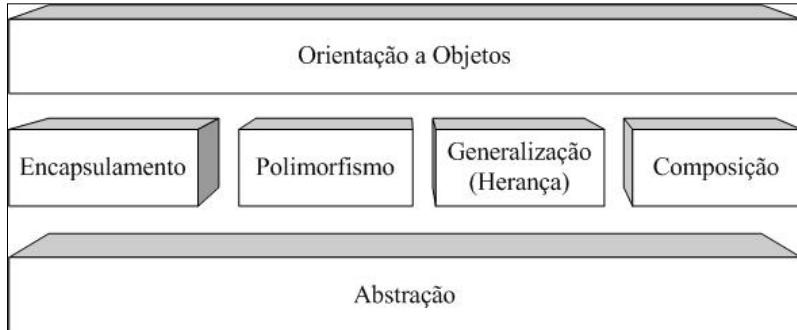
## Abstração

- Uma abstração é qualquer modelo que inclui os aspectos relevantes de alguma coisa, ao mesmo tempo em que ignora os menos importantes. *Abstração depende do observador.*



### Abstração na orientação a objetos

- A orientação a objetos faz uso intenso de abstrações.
  - Os princípios da OO podem ser vistos como aplicações da abstração.
- **Princípios da OO:** encapsulamento, polimorfismo, herança e composição.



### Objetos como abstrações

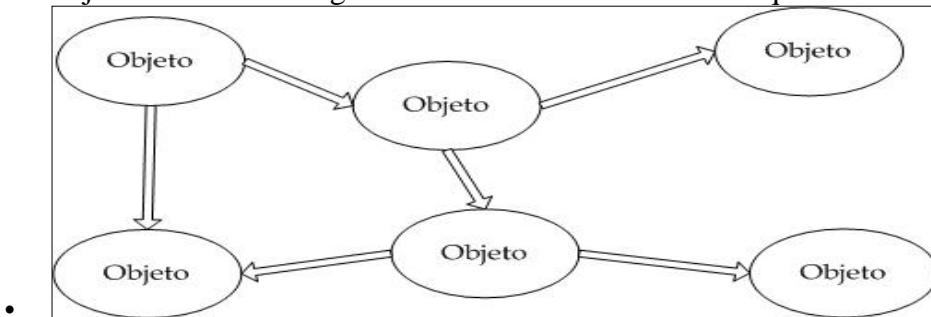
- Uma abstração é uma representação das características e do comportamento relevantes de um conceito do mundo real para um determinado problema.
- Dependendo do contexto, um mesmo conceito do mundo real pode ser representado por diferentes abstrações.
  - Carro (para uma transportadora de cargas)
  - Carro (para uma fábrica de automóveis)
  - Carro (para um colecionador)
  - Carro (para uma empresa de kart)
  - Carro (para um mecânico)

### Classe X Objeto

- Objetos são abstrações de entidades que existem no mundo real.
- Classes são definições estáticas, que possibilitam o entendimento de um grupo de objetos.
- CUIDADO: estes dois termos muitas vezes são usados indistintamente em textos sobre orientação a objetos.

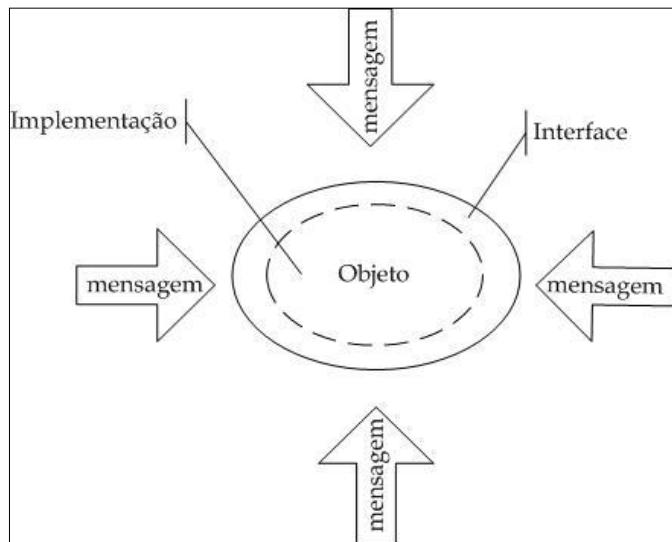
## Mensagens

- Para que um objeto realize alguma tarefa, deve haver um estímulo enviado a este objeto.
- Pense em um objeto como uma entidade ativa que representa uma abstração de algo do mundo real
  - Então faz sentido dizer que tal objeto pode responder a estímulos a ele enviados
  - Assim como faz sentido dizer que seres vivos reagem a estímulos que eles recebem.
- Independentemente da origem do estímulo, quando ele ocorre, diz-se que o objeto em questão está recebendo uma **mensagem**.
- Uma mensagem é uma requisição enviada de um objeto a outro para que este último realize alguma operação.
- *Objetos de um sistema trocam mensagens*
- isto significa que estes objetos estão enviando mensagens uns aos outros com o objetivo de realizar alguma tarefa dentro do sistema no qual eles estão inseridos.

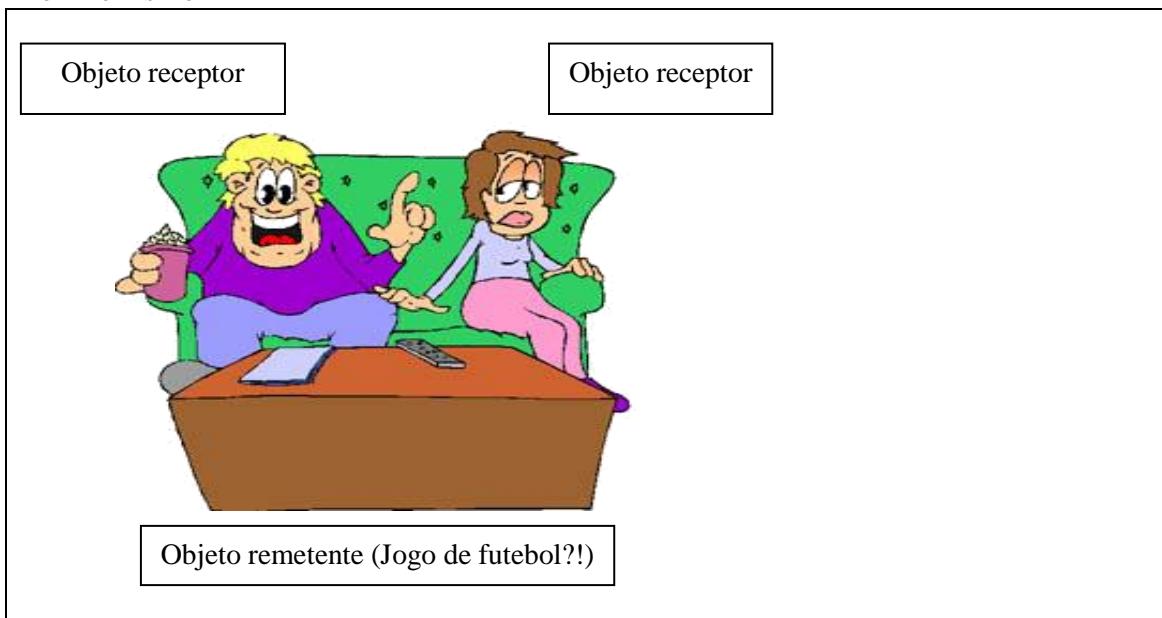


## Encapsulamento

- Objetos possuem **comportamento**.
  - O termo comportamento diz respeito a que operações são realizadas por um objeto e também de que modo estas operações são executadas.
- De acordo com o encapsulamento, objetos devem “esconder” a sua complexidade...
- Esse princípio aumenta qualidade do SSOO, em termos de:
  - Legibilidade
  - Clareza
  - Reuso
- Uma interface pode ter várias formas de **implementação**.
- Mas, pelo princípio do encapsulamento, a implementação utilizada por um objeto receptor de uma mensagem não importa para um objeto remetente da mesma.



## Polimorfismo

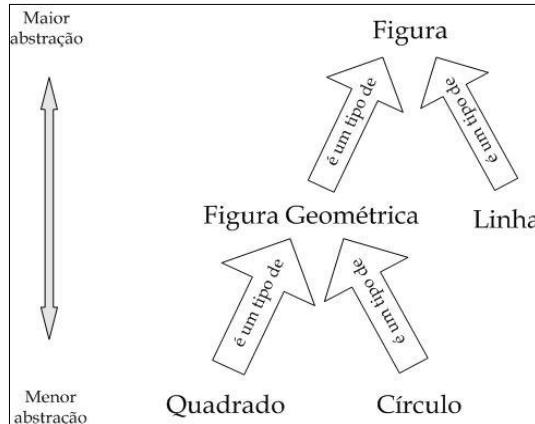


- É a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras.
- Em uma linguagem orientada a objetos:  
`for(i = 0; i < poligonos.tamanho(); i++)  
 poligonos[i].desenhar();`

## Generalização (Herança)

- A herança pode ser vista como um nível de abstração acima da encontrada entre classes e objetos.
- Na herança, classes semelhantes são agrupadas em hierarquias.
  - Cada nível de uma hierarquia pode ser visto como um nível de abstração.
  - Cada classe em um nível da hierarquia herda as características das classes nos níveis acima.

- A herança facilita o compartilhamento de comportamento entre classes semelhantes.
- As diferenças ou variações de uma classe em particular podem ser organizadas de forma mais clara.



### 1.3 Evolução histórica da modelagem de sistemas

#### 1.4 A Linguagem de modelagem unificada

##### Evolução do Hardware

- A chamada ***Lei de Moore*** é bastante conhecida da comunidade de computação.
- Essa lei foi declarada em 1965 pelo engenheiro *Gordon Moore*, co-fundador da Intel.
- ***Lei de Moore:*** “A densidade de um transistor dobra em um período entre 18 e 24 meses”.

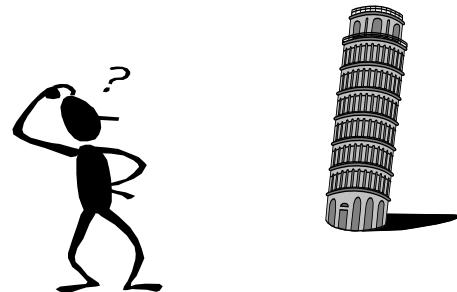
##### Evolução do Software

- O rápido crescimento da capacidade computacional das máquinas resultou na demanda por sistemas de software cada vez mais complexos.
- O surgimento de sistemas de software mais complexos resultou na necessidade de reavaliação da forma de se desenvolver sistemas.
- Conseqüentemente as técnicas utilizadas para a construção de sistemas computacionais têm evoluído de forma impressionante, notavelmente no que tange à modelagem de sistemas.
- Na primeira metade da década de 90 surgiram várias propostas de técnicas para modelagem de sistemas segundo o paradigma orientado a objetos.
- Houve uma grande proliferação de propostas para modelagem orientada a objetos.
- Diferentes notações gráficas para modelar uma mesma perspectiva de um sistema.
- Cada técnica tinha seus pontos fortes e fracos.

##### Necessidade de um Padrão

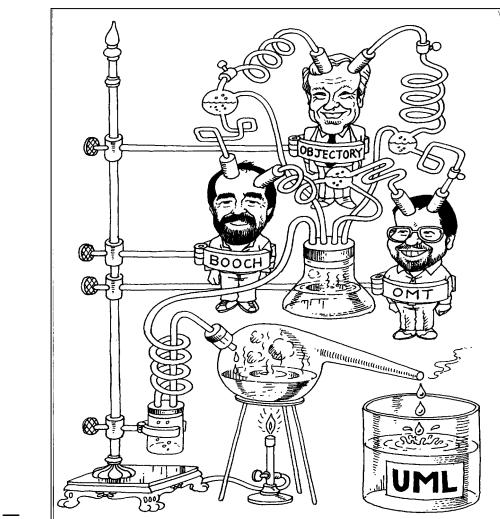
- Percebeu-se a necessidade de um padrão para a modelagem de sistemas, que fosse aceito e utilizado amplamente.

- Alguns esforços nesse sentido de padronização, o principal liderado pelo “três amigos”.
- Surge a UML (Unified Modeling Language) em 1996 como a melhor candidata para ser linguagem “unificadora”.
- Em 1997, a UML é aprovada como padrão pelo OMG.
- Desde então, a UML tem tido grande aceitação pela comunidade de desenvolvedores de sistemas.
- É uma linguagem ainda em desenvolvimento.
- Atualmente na versão 2.2.



UML (Linguagem de Modelagem Unificada)

- “A UML é a linguagem padrão para visualizar, especificar, construir e documentar os artefatos de software de um sistema.”
- Unificação de diversas notações anteriores.
- Mentores: Booch, Rumbaugh e Jacobson
  - “Três Amigos”
  - IBM Rational ([www.rational.com](http://www.rational.com))

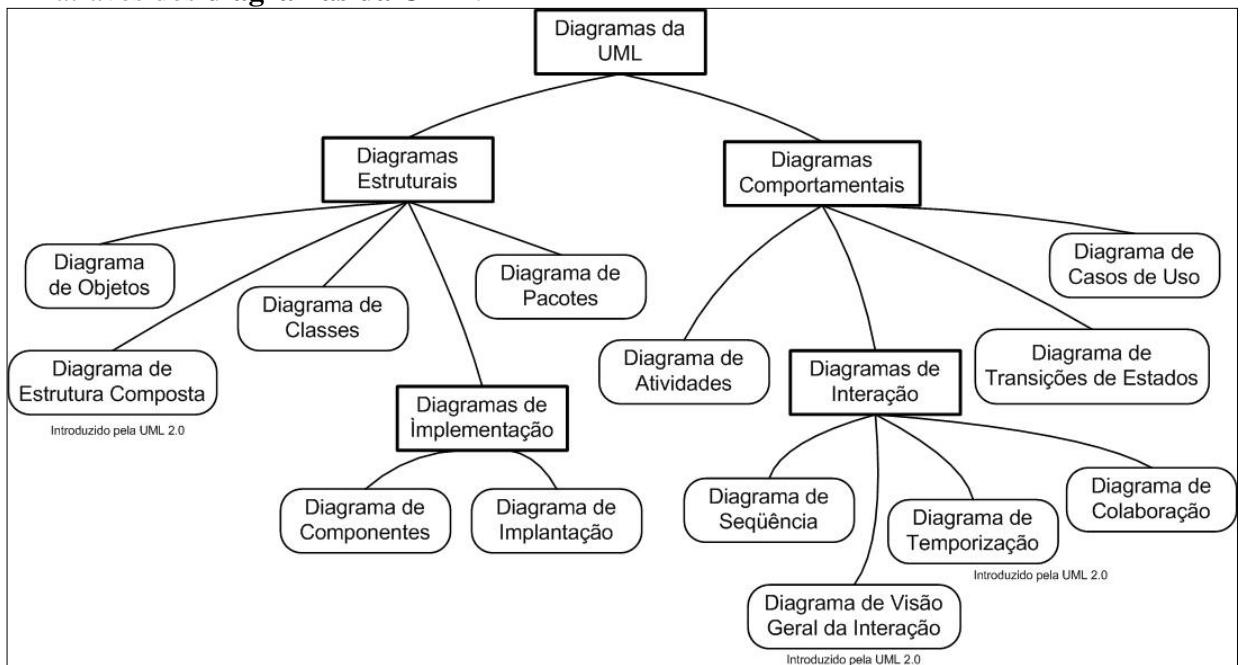


- UML é...
  - uma linguagem visual.
  - independente de linguagem de programação.
  - independente de processo de desenvolvimento.
- UML **não** é...
  - uma linguagem programação (mas possui versões!).
  - uma técnica de modelagem.
- Um processo de desenvolvimento que utilize a UML como linguagem de modelagem envolve a criação de diversos documentos.
  - Estes documentos, denominados **artefatos de software**, podem ser textuais ou gráficos.

- Os artefatos gráficos produzidos de um sistema OO são definidos através dos **diagramas da UML**.

## Diagramas da UML

- Um diagrama na UML é uma apresentação de uma coleção de **elementos gráficos** que possuem um significado predefinido.
  - No contexto de desenvolvimento de software, correspondem a desenhos gráficos que seguem algum padrão lógico.
- Um processo de desenvolvimento que utilize a UML como linguagem de modelagem envolve a criação de diversos documentos.
  - Estes documentos, denominados **artefatos de software**, podem ser textuais ou gráficos.
- Os artefatos gráficos produzidos no desenvolvimento de um SSOO são definidos através dos **diagramas da UML**.



## 2 Processo de Desenvolvimento de Software

*“Quanto mais livros você leu (ou escreveu), mais as aulas você assistiu (ou lecionou), mais linguagens de programação você aprendeu (ou projetou), mais software OO você examinou (ou produziu), mais documentos de requisitos você tentou decifrar (ou tornou decifrável), mais padrões de projeto você aprendeu (ou catalogou), mais reuniões você assistiu (ou conduziu), mais colegas de trabalho talentosos você teve (ou contratou), mais projetos você ajudou (ou gerenciou), tanto mais você estará equipado para lidar com um novo desenvolvimento.” - Bertrand Meyer*

“Software is hard...”

- Porcentagem de projetos que terminam dentro do prazo estimado: 10%
- Porcentagem de projetos que são descontinuados antes de chegarem ao fim: 25%
- Porcentagem de projetos acima do custo esperado: 60%
- Atraso médio nos projetos: um ano.

**Fonte: Chaos Report (1994)**

### Processo de desenvolvimento

- Tentativas de lidar com a complexidade e de minimizar os problemas envolvidos no desenvolvimento de software envolvem a definição de **processos de desenvolvimento de software**.
- Um processo de desenvolvimento de software (PDS) compreende todas as atividades necessárias para definir, desenvolver, testar e manter um produto de software.
- Exemplos de processos de desenvolvimento existentes:
  - ICONIX
  - RUP
  - EUP
  - XP
  - OPEN
- Alguns objetivos de um processo de desenvolvimento são:
  - Definir *quais* as atividades a serem executadas ao longo do projeto;
  - Definir *quando, como* e por *quem* tais atividades serão executadas;
  - Prover pontos de controle para verificar o andamento do desenvolvimento;
  - Padronizar a forma de desenvolver software em uma organização.

### 2.10 Atividades típicas de um PDS.

### 2.2 O componente humano em um PDS

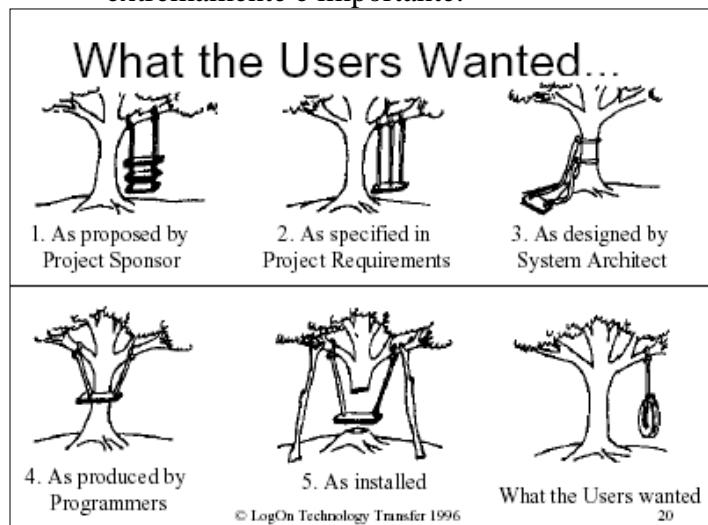
- Análise de requisitos
- Projeto
- Implementação
- Testes
- Implantação

### Participantes do processo

- Gerentes de projeto
- Analistas
- Projetistas
- Arquitetos de software
- Programadores
- Clientes
- Avaliadores de qualidade

### Participação do usuário

- A participação do usuário durante o desenvolvimento de um sistema extremamente é importante.

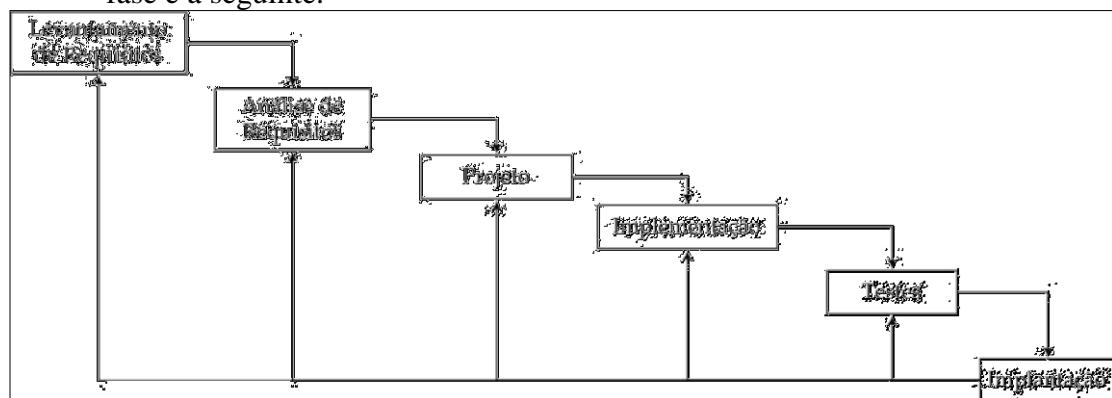


### 2.3 Modelos de ciclo de vida

- Um ciclo de vida corresponde a um encadeamento específico das fases para construção de um sistema.
- Dois modelos de ciclo de vida:
  - *modelo em cascata*
  - *modelo iterativo e incremental*.

#### Modelo em cascata

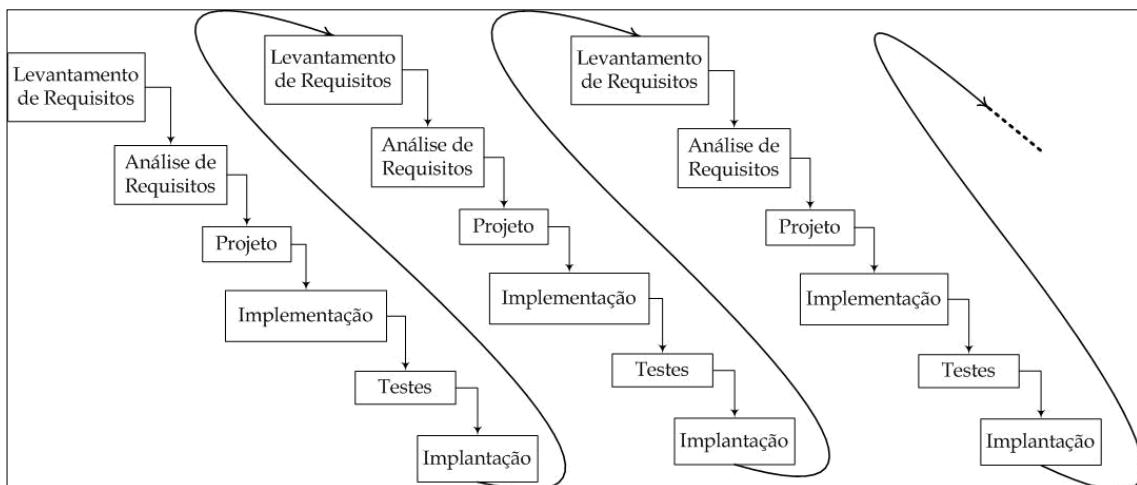
- Esse modelo apresenta uma tendência para a progressão seqüencial entre uma fase e a seguinte.



- Projetos reais raramente seguem um fluxo seqüencial.
- Assume que é possível declarar detalhadamente todos os requisitos antes do início das demais fases do desenvolvimento.
  - propagação de erros pelas as fases do processo.
- Uma versão de produção do sistema não estará pronta até que o ciclo do projeto de desenvolvimento chegue ao final.

### Modelo de iterativo e incremental

- Divide o desenvolvimento de um produto de software em **ciclos**.
- Em cada ciclo de desenvolvimento, podem ser identificadas as fases de análise, projeto, implementação e testes.
- Cada ciclo considera um subconjunto de requisitos.
- Esta característica contrasta com a abordagem clássica, na qual as fases são realizadas uma única vez.
- Desenvolvimento em “mini-cascatas”.



- **Iterativo:** o sistema de software é desenvolvido em vários passos similares.
- **Incremental:** Em cada passo, o sistema é estendido com mais funcionalidades.

### Modelo iterativo e incremental – vantagens e desvantagens

- ⌚ Incentiva a participação do usuário.
- ⌚ Riscos do desenvolvimento podem ser mais bem gerenciados.
  - Um **risco de projeto** é a possibilidade de ocorrência de algum evento que cause prejuízo ao processo de desenvolvimento, juntamente com as consequências desse prejuízo.
  - Influências: custos do projeto, cronograma, qualidade do produto, satisfação do cliente, etc.
- ⌚ Mais difícil de gerenciar

### Ataque os riscos

- “Se você não atacar os riscos [do projeto] ativamente, então estes irão ativamente atacar você.” (Tom Gilb).
  - A maioria dos PDS que seguem o modelo iterativo e incremental aconselha que as partes mais arriscadas sejam consideradas inicialmente.



## 2.4 Utilização da UML no modelo iterativo e incremental

UML no modelo iterativo e incremental

- A UML é independente do processo de desenvolvimento.
  - Vários processos podem utilizar a UML para modelagem de um sistema OO.
- Os artefatos de software construídos através da UML evoluem à medida que as iterações são realizadas.
  - A cada iteração, novos detalhes são adicionados a esses artefatos.
  - Além disso, a construção de um artefato fornece informações para adicionar detalhes a outros.

## 2.5 Prototipagem

- A *prototipagem* é uma técnica aplicada quando:
  - Há dificuldades no entendimento dos requisitos do sistema
  - Há requisitos que precisam ser mais bem entendidos.
- A construção de *protótipos* utiliza ambientes com facilidades para a construção da interface gráfica.
- Procedimento geral da prototipagem:
  - Após o LR, um protótipo é construído para ser usado na *validação*.
  - Usuários fazem críticas...
  - O protótipo é então corrigido ou refinado
  - O processo de revisão e refinamento continua até que o protótipo seja aceito.
  - Após a aceitação, o protótipo é descartado ou utilizado como uma versão inicial do sistema.
- Note que a prototipagem NÃO é um substituto à construção de modelos do sistema.
  - A prototipagem é uma técnica complementar à construção dos modelos do sistema.
  - Mesmo com o uso de protótipos, os modelos do sistema devem ser construídos.
  - Os erros detectados na validação do protótipo devem ser utilizados para modificar e refinar os modelos do sistema.

## 2.6 Ferramentas de suporte

- O desenvolvimento de um software pode ser facilitado através do uso de ferramentas que auxiliam:
  - na construção de modelos,
  - na integração do trabalho de cada membro da equipe,
  - no gerenciamento do andamento do desenvolvimento, etc.



- Há diversos sistemas de software que são utilizados para dar suporte ao desenvolvimento de outros sistemas.
- Um tipo bastante conhecido de ferramenta de suporte são as **ferramentas CASE**.
  - CASE: *Computer Aided Software Engineering*
- Além das ferramentas CASE, outras ferramentas importantes são as que fornecem suporte ao **gerenciamento**.
  - desenvolver cronogramas de tarefas,
  - definir alocações de verbas,
  - monitorar o progresso e os gastos,
  - gerar relatórios de gerenciamento, etc.
- Criação e manutenção da consistência entre estes diagramas
- *Round-trip engineering*
- Depuração de código fonte
- Relatórios de testes
- Testes automáticos
- Gerenciamento de versões
- Verificação de desempenho
- Verificação de erros em tempo de execução
- Gerenciamento de mudanças nos requisitos
- Prototipagem

## 3 Mecanismos Gerais

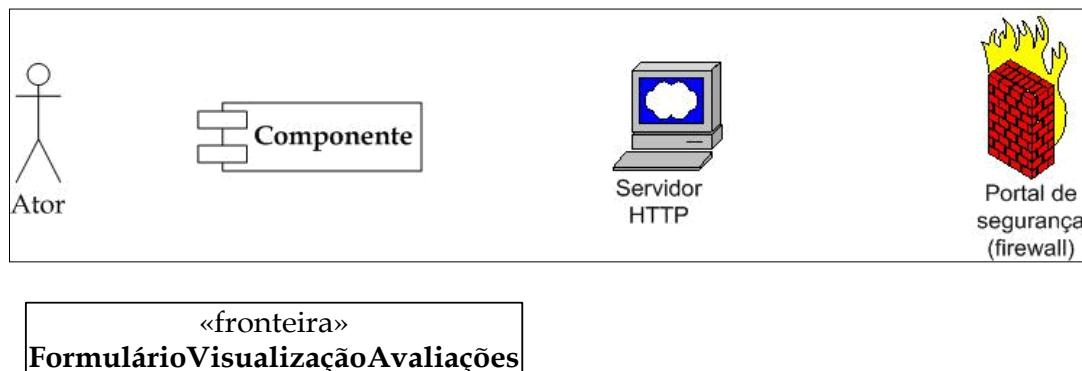
"Podemos apenas ver uma curta distância à frente, mas podemos ver que há muito lá a ser feito." -Alan Turing

### Tópicos

- Estereótipos
- Notas explicativas
- Etiquetas
- Restrições
- Pacotes
- OCL

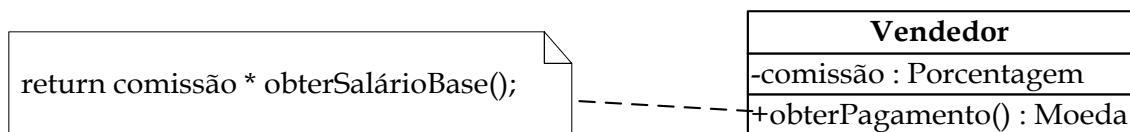
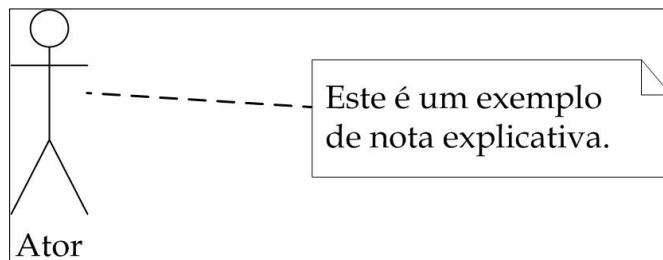
### 3.1 Estereótipos

- Utilizado para *estender* (enriquecer) o significado de um determinado elemento em um diagrama.
- A UML predefine diversos estereótipos.
- É possível também definir estereótipos específicos.
- Estereótipos podem ser classificados em dois tipos:
  - *estereótipo gráfico*: um ícone que lembre o significado do conceito a ele associado.
  - *estereótipos de rótulo*: um nome delimitado pelos símbolos << e >>.



### 3.2 Notas explicativas

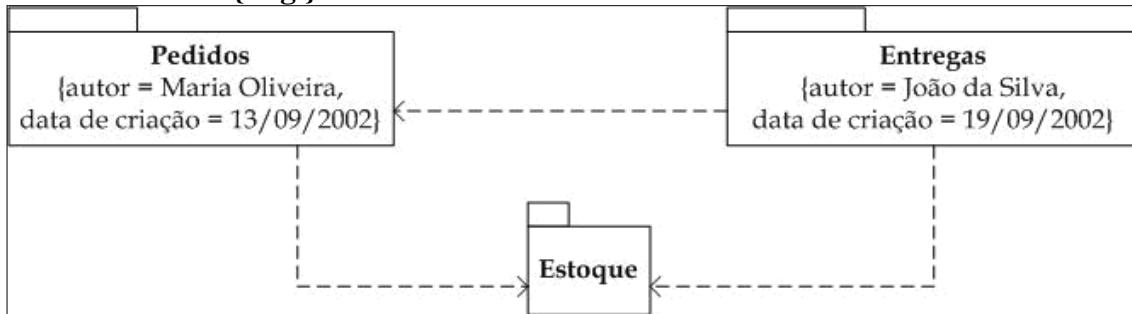
- Utilizadas para comentar ou esclarecer alguma parte de um diagrama.
- Podem ser descritas em texto livre; também podem corresponder a uma expressão formal utilizando OCL (adiante).



### 3.3 Etiquetas (Tags)

- Os elementos gráficos de um diagrama da UML possuem propriedades predefinidas.
- Propriedades adicionais para elementos gráficos de um diagrama podem ser definidas através do uso de *etiquetas*.
- Alternativas de notação para definição de etiquetas na UML:

{ tag = valor }  
 { tag1 = valor1 , tag2 = valor2 ... }  
 { tag }

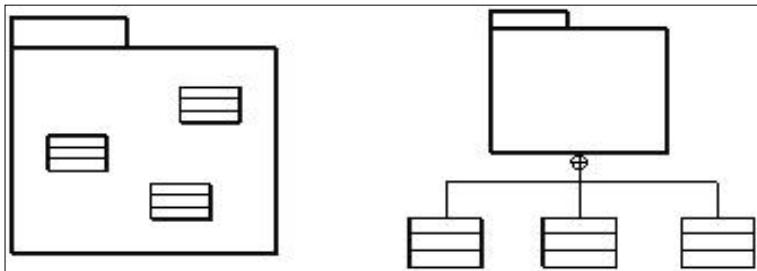


### 3.4 Restrições

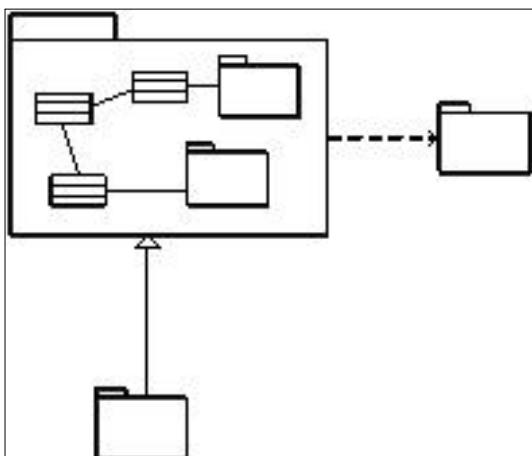
- A todo elemento da UML está associada alguma *semântica*.
  - Cada elemento gráfico possui um significado bem definido que, uma vez entendido, fica implícito na utilização do elemento em algum diagrama.
- As *restrições* permitem estender ou alterar a semântica natural de um elemento gráfico.
- Este mecanismo geral especifica restrições sobre um ou mais valores de um ou mais elementos de um modelo.
- A UML define uma linguagem formal que pode ser utilizada para especificar restrições sobre diversos elementos de um modelo.
- Esta linguagem se chama **OCL**, a *Linguagem de Restrição de Objetos*.
- A OCL pode ser utilizada para definir expressões de navegação entre objetos expressões lógicas, consulta, etc.

### 3.5 Pacotes

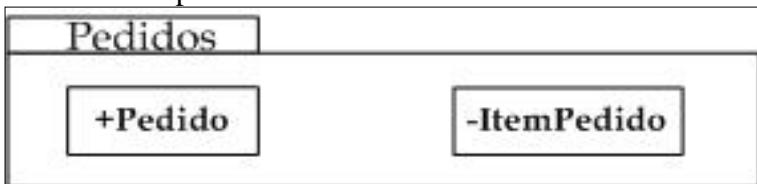
- Um mecanismo de agrupamento geral que pode ser utilizado para agrupar vários artefatos de um modelo.
- Notação: uma pasta com uma aba.
- Conteúdo, duas maneiras de representar graficamente:
  - 1) exibir o conteúdo dentro do pacote
  - 2) “pendurar” os elementos agrupados no ícone do pacote.



- Pacotes podem ser agrupados dentro de outros pacotes, formando uma hierarquia de contenção.



- Cada elemento de um pacote pode ter visibilidade pública, protegida ou privativa.
- Exemplo:



- Pode haver relacionamentos de dependência entre pacotes.
  - Assim, pode-se construir um *diagrama de pacotes* que representa dependências entre pacotes.
  - Um pacote P1 é dependente de outro, P2, se houver qualquer dependência entre quaisquer dois elementos de P1 e P2.

### 3.6 OCL

- A UML define uma linguagem formal que pode ser utilizada para especificar restrições sobre diversos elementos de um modelo, a *OCL*.
  - OCL: *Object Constraint Language* (Linguagem de Restrição de Objetos).
- A OCL pode ser utilizada para definir expressões de navegação, expressões lógicas, pré-condições, pós-condições, etc.
- A maioria das declarações em OCL consiste dos seguintes elementos estruturais: *contexto*, *propriedade* e *operação*.

- Um contexto define o domínio no qual a declaração em OCL se aplica.
  - Por exemplo, uma classe ou uma instância de uma classe.
- Uma propriedade corresponde a algum componente do contexto.
  - Por exemplo, o nome de um atributo em uma classe, ou uma associação entre dois objetos.
- Finalmente a operação define o que deve ser aplicado sobre a propriedade.
- Uma expressão em OCL pode envolver diversos operadores:
  - Operadores aritméticos, operadores de conjunto e operadores de tipo.
  - Outros operadores: and, or, implies, if, then, else, not, in.
- A OCL pode ser utilizada em qualquer diagrama da UML.
  - durante as descrições dos diagramas da UML em outros capítulos, são fornecidos alguns exemplos de expressões em OCL.

## 4 Modelagem De Casos De Uso

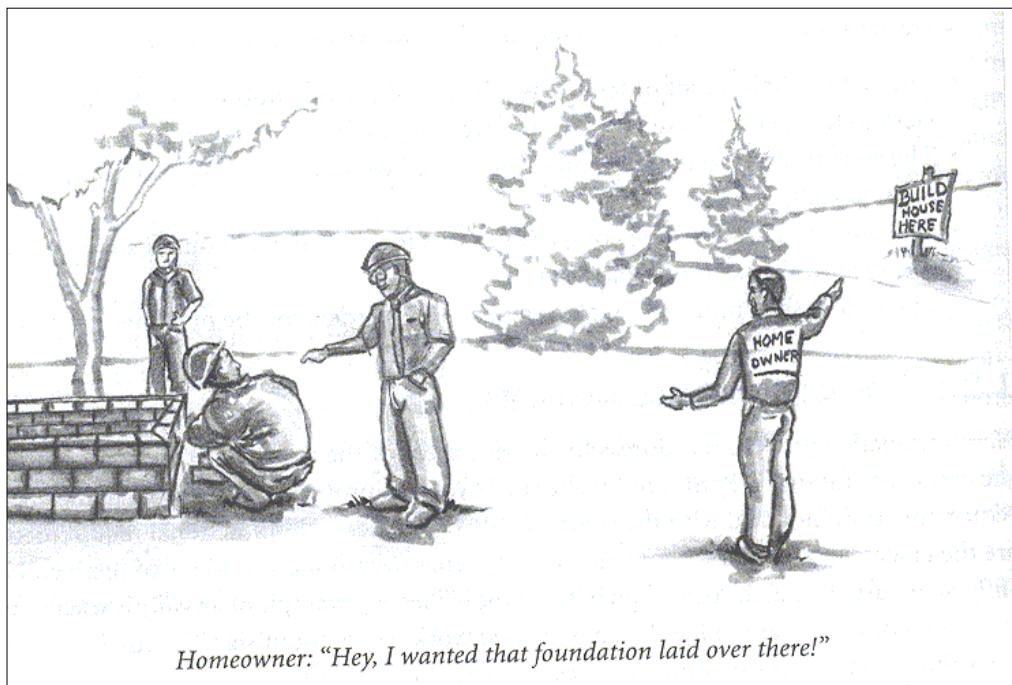
- Introdução
- Diagrama de casos de uso
- Identificação dos elementos do MCU
- Construção do MCU
- Documentação suplementar ao MCU
- O MCU em um processo de desenvolvimento iterativo e incremental

### Introdução

- O **modelo de casos de uso** é uma representação das *funcionalidades* externamente observáveis do sistema e dos *elementos externos* ao sistema que interagem com o mesmo.
- Esse modelo representa os **requisitos funcionais** do sistema.
- Também direciona diversas das atividades posteriores do ciclo de vida do sistema de software.
- Além disso, força os desenvolvedores a moldar o sistema de acordo com as **necessidades** do usuário.

### Utilidade dos Casos de Uso

- Equipe de clientes (**validação**)
  - aprovam o que o sistema deverá fazer
  - entendem o que o sistema deverá fazer
- Equipe de desenvolvedores
  - Ponto de partida para refinar requisitos de software.
  - Podem seguir um desenvolvimento dirigido a casos de uso.
  - Designer (projetista): encontrar classes
  - Testadores: usam como base para **casos de teste**



### Composição do MCU

- O modelo de casos de uso de um sistema é composto de duas partes, uma **textual**, e outra **gráfica**.
- O diagrama da UML utilizado na modelagem de gráfica é o **diagrama de casos de uso**.
  - Este diagrama permite dar uma visão global e de alto nível do sistema.
  - É também chamado de diagrama de contexto.
- Componentes: casos de uso, atores, relacionamentos entre os elementos anteriores.

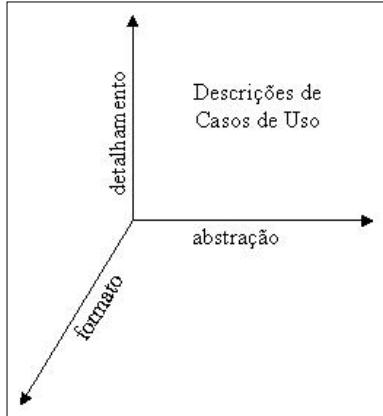
### Casos de uso

- Um caso de uso é a especificação de uma seqüência de interações entre um sistema e os agentes externos.
- Define parte da funcionalidade de um sistema, *sem revelar a estrutura e o comportamento internos deste sistema*.
- Um modelo de casos de uso típico é formado de vários casos de uso.
- Cada caso de uso é definido através da **descrição textual** das interações que ocorrem entre o(s) elemento(s) externo(s) e o sistema.
- Há várias “dimensões de estilo” para descrição de casos de uso: Grau de abstração; Formato; Grau de detalhamento.

### Dimensões para Descrições Textuais

- Um caso de uso é definido através da descrição textual das interações entre o(s) elemento(s) externo(s) e o sistema.
- Entretanto, a UML não define nada acerca de como essa descrição textual deve ser construída.
- Por conta disso, há várias dimensões independentes sobre as quais a descrição textual de um caso de uso pode variar:
  - Grau de abstração (essencial ou real)
  - Formato (contínua, tabular, numerado)

– Grau de detalhamento (sucinta ou expandida)



Formato

- Exemplo de descrição contínua

Este caso de uso inicia quanto o Cliente chega ao caixa eletrônico e insere seu cartão. O Sistema requisita a senha do Cliente. Após o Cliente fornecer sua senha e esta ser validada, o Sistema exibe as opções de operações possíveis. O Cliente opta por realizar um saque. Então o Sistema requisita o total a ser sacado. O Cliente fornece o valor da quantidade que deseja sacar. O Sistema fornece a quantia desejada e imprime o recibo para o Cliente. O Cliente retira a quantia e o recibo, e o caso de uso termina.

- Exemplo de descrição numerada

- 1) Cliente insere seu cartão no caixa eletrônico.
- 2) Sistema apresenta solicitação de senha.
- 3) Cliente digita senha.
- 4) Sistema valida a senha e exibe menu de operações disponíveis.
- 5) Cliente indica que deseja realizar um saque.
- 6) Sistema requisita o valor da quantia a ser sacada.
- 7) Cliente fornece o valor da quantia que deseja sacar.
- 8) Sistema fornece a quantia desejada e imprime o recibo para o Cliente
- 9) Cliente retira a quantia e o recibo, e o caso de uso termina.

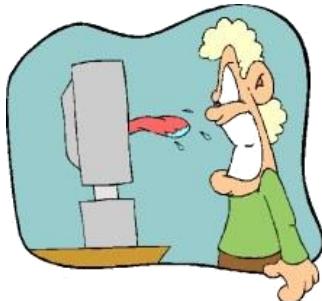
- Exemplo de descrição tabular

Cliente	Sistema
Insere seu cartão no caixa eletrônico. <input type="checkbox"/> Digita senha. <input type="checkbox"/>  Solicita realização de saque. <input type="checkbox"/>  Fornece o valor da quantia que deseja sacar. <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>  <input type="checkbox"/> Retira a quantia e o recibo.	<input type="checkbox"/> Apresenta solicitação de senha. <input type="checkbox"/>  Valida senha e exibe menu de operações disponíveis. <input type="checkbox"/>  Requisita quantia a ser sacada. <input type="checkbox"/>  Fornece a quantia desejada e imprime o recibo para o Cliente

## Grau de Abstração

- Exemplo de descrição essencial (e numerada):
  - 1) Cliente fornece sua identificação.
  - 2) Sistema identifica o usuário.
  - 3) Sistema fornece opções disponíveis para movimentação da conta.
  - 4) Cliente solicita o saque de uma determinada quantia.
  - 5) Sistema requisita o valor da quantia a ser sacada.
  - 6) Cliente fornece o valor da quantia que deseja sacar.
  - 7) Sistema fornece a quantia desejada.
  - 8) Cliente retira dinheiro e recibo e o caso de uso termina.

## Atores



- Elemento externo que interage com o sistema.
  - “externo”: atores não fazem parte do sistema.
  - “interage”: um ator troca informações com o sistema.
- Casos de uso representam uma seqüência de interações entre o sistema e o ator.
  - no sentido de troca de informações entre eles.
- Normalmente um agente externo inicia a seqüência de interações com o sistema.



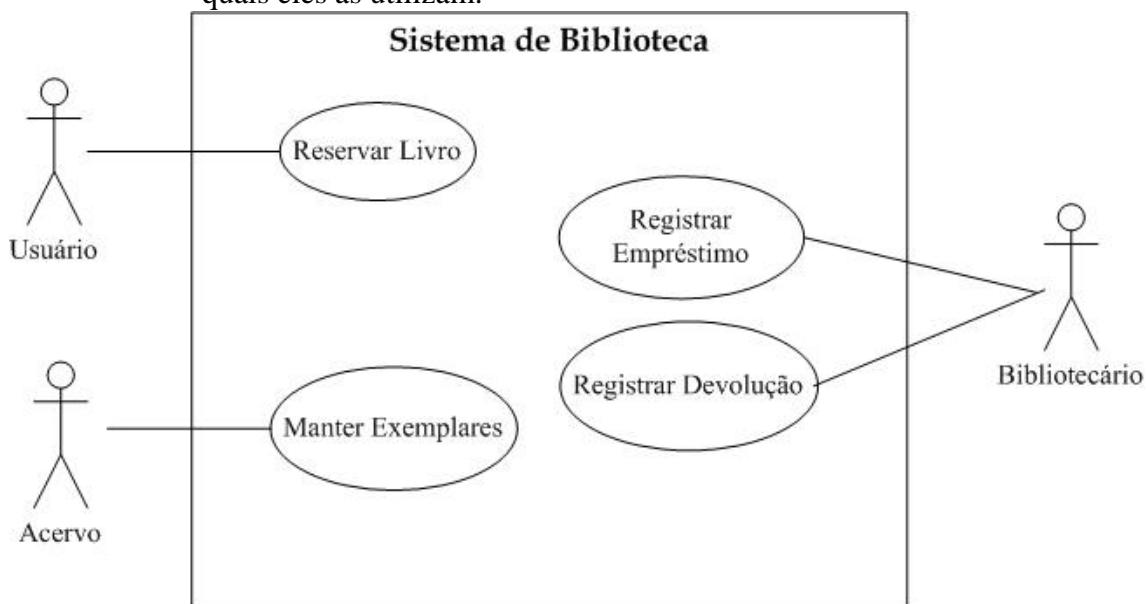
- Categorias de atores:
  - **cargos** (Empregado, Cliente, Gerente, Almoxarife, Vendedor, etc);
  - **organizações** (Empresa Fornecedor, Agência de Impostos, Administradora de Cartões, etc);
  - **outros sistemas** (Sistema de Cobrança, Sistema de Estoque de Produtos, etc).
  - **equipamentos** (Leitora de Código de Barras, Sensor, etc.)
- Essa categorização indica para nós que o conceito de ator depende do **escopo** do sistema.
- Um ator corresponde a um **papel** representado em relação ao sistema.
  - O mesmo indivíduo pode ser o **Cliente** que compra mercadorias e o **Vendedor** que processa vendas.
  - Uma pessoa pode representar o papel de **Funcionário** de uma instituição bancária que realiza a manutenção de um caixa eletrônico, mas também pode ser o **Cliente** do banco que realiza o saque de uma quantia.
- O nome dado a um ator deve lembrar o seu papel, em vez de lembrar quem o representa.
  - e.g.: João Fernandes versus Fornecedor

## Atores versus Casos de Uso

- Um **ator** representa um conjunto coerente de papéis que os usuários de casos desempenham quando interagem com o sistema
- Um **caso de uso** representa o que um ator quer que o sistema faça.
- Atores servem para definir o **ambiente do sistema**
- Atores representam um **papel** exercido por uma pessoa ou por um sistema externo que interage com o sistema.
- Comunicam-se enviando mensagens e/ou recebendo mensagens do sistema, conforme o caso de uso é executado
- Quando definimos o que os atores fazem e o que os casos de uso fazem, delimitamos, de forma clara, o **escopo do sistema**.

### 4.2 Diagrama de casos de uso

- Representa *graficamente* os atores, casos de uso e relacionamentos entre os elementos.
- Tem o objetivo de ilustrar em um nível alto de abstração quais elementos externos interagem com que funcionalidades do sistema.
- Uma espécie de “diagrama de contexto”.
  - Apresenta os elementos externos de um sistema e as maneiras segundo as quais eles as utilizam.

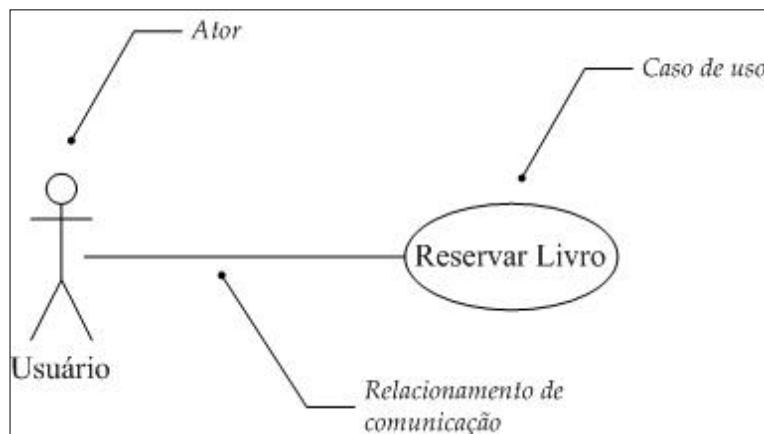


### Elementos de um MCU

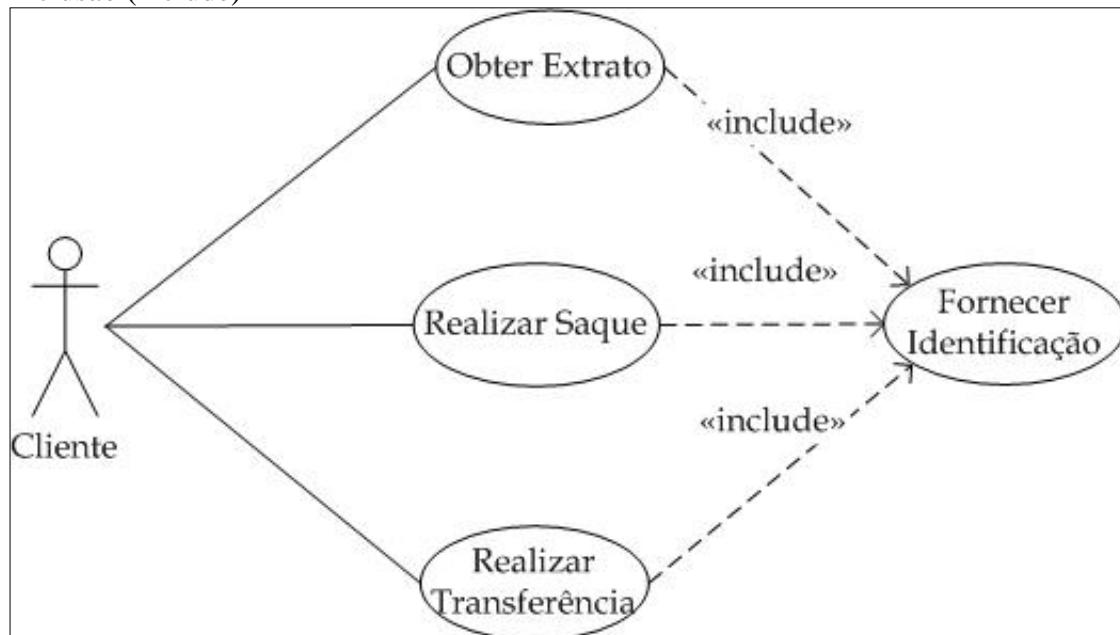
- Um MCU possui diversos elementos, e cada um deles pode ser representado graficamente. Os elementos mais comuns em um MCU são:
  - *Ator*
  - *Caso de uso*
- Além disso, a UML define diversos relacionamentos entre esses elementos para serem usados no modelo de casos de uso:
  - *Comunicação*
  - *Inclusão*
  - *Extensão*

- *Generalização*
- Para cada um desses elementos, a UML define uma notação gráfica e uma semântica específicas.

Autor, caso de uso, comunicação

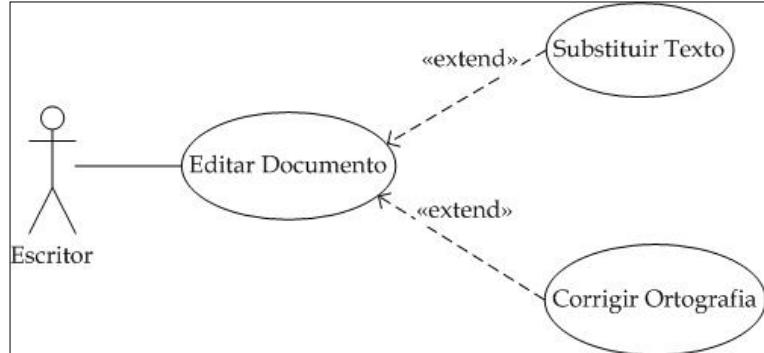


Inclusão (include)

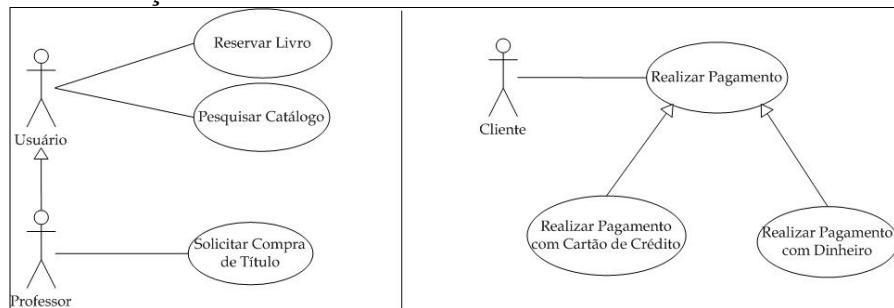


- Referência no texto do caso de uso inclusor:  
***Include(Fornecer Identificação)***

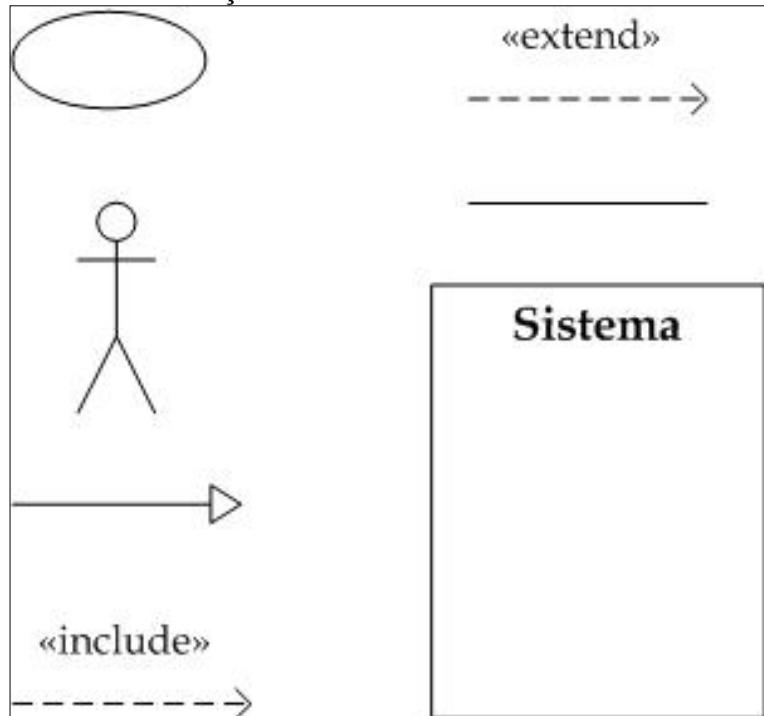
### Extensão (extend)



### Generalização



### Resumo da Notação



#### 4.3 Identificação dos elementos do MCU

- Atores e os casos de uso são identificados a partir de informações coletadas no levantamento de requisitos.
  - Durante esta fase, analistas devem identificar as atividades do negócio relevantes ao sistema a ser construído.
- Não há uma regra geral que indique quantos casos de uso e atores são necessários para descrever um sistema.
  - A quantidade de casos de uso e atores depende da complexidade do sistema.
- Note também que as identificações de atores e de casos de uso são atividades que se intercalam.
- Fontes e os destinos das informações a serem processadas são atores em potencial.
  - uma vez que, por definição, um ator é todo elemento externo que *interage* com o sistema.
- O analista deve identificar:
  - as áreas da empresa que serão afetadas ou utilizarão o sistema.
  - fontes de informações a serem processadas e os destinos das informações geradas pelo sistema.
- Há algumas perguntas úteis cujas respostas potencialmente identificam atores.
  - Que órgãos, empresas ou pessoas (cargos) irão utilizar o sistema?
  - Que outros sistemas irão se comunicar com o sistema?
  - Alguém deve ser informado de alguma ocorrência no sistema?
  - Quem está interessado em um certo requisito funcional do sistema?
- A partir da lista (inicial) de atores, deve-se passar à identificação dos casos de uso.
- Nessa identificação, pode-se distinguir entre dois tipos de casos de uso
  - Primário: representa os *objetivos* dos atores.
  - Secundário: aquele que não traz benefício direto para os atores, mas que é necessário para que sistema funcione adequadamente.

#### Casos de Uso Primários

- Perguntas úteis:
  - Quais são as necessidades e objetivos de cada ator em relação ao sistema?
  - Que informações o sistema deve produzir?
  - O sistema deve realizar alguma ação que ocorre regularmente no tempo?
  - Para cada requisito funcional, existe um (ou mais) caso(s) de uso para atendê-lo?
- Outras técnicas de identificação:
  - *Caso de uso “oposto”*
  - *Caso de uso que precede/sucede a outro caso de uso*
  - *Caso de uso temporal*
  - *Caso de uso relacionado a uma condição interna*

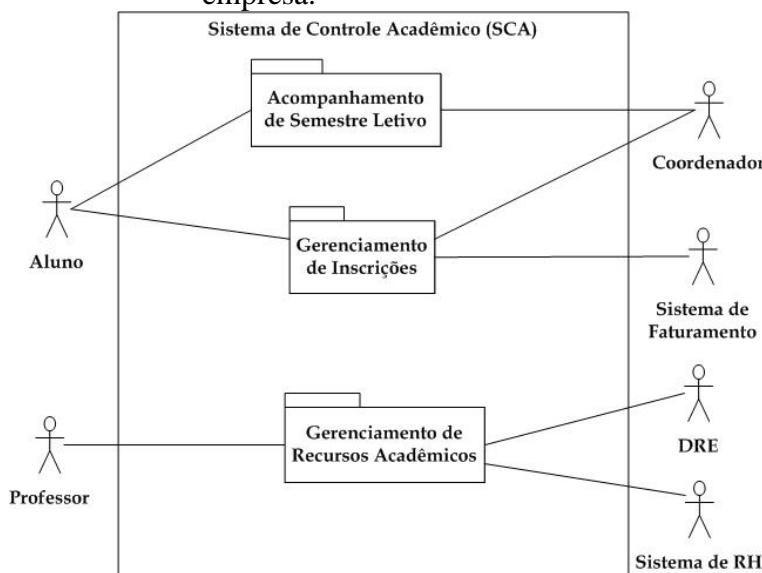
#### Casos de Uso Secundários

- Estes se encaixam nas seguintes categorias:
  - Manutenção de cadastros;

- Manutenção de usuários;
- Gerenciamento de acesso;
- Manutenção de informações provenientes de outros sistemas.
- Obs: casos de uso secundários, são menos importantes que os casos de uso primários.
  - O sistema de software não existe para cadastrar informações, nem tampouco para gerenciar os usuários.
  - O objetivo principal de um sistema é agregar valor ao ambiente no qual ele está implantado.

#### 4.4 Construção do MCU

- Os diagramas de casos de uso devem servir para dar suporte à parte textual do modelo, fornecendo uma visão de alto nível.
- Quanto mais fácil for a leitura do diagrama representando casos de uso, melhor.
- Se o sistema sendo modelado não for tão complexo, pode ser criado um único DCU.
- É útil e recomendada a utilização do retângulo de fronteira para delimitar e separar visualmente casos de uso e atores.
- Em sistemas complexos, representar todos os casos de uso do sistema em um único DCU talvez o torne um tanto ilegível.
- Alternativa: criar vários diagramas (de acordo com as necessidades de visualização) e agrupá-los em pacotes.
  - Todos os casos de uso para um ator;
  - Todos os casos de uso a serem implementados em um ciclo de desenvolvimento.
  - Todos os casos de uso de uma área (departamento, seção) específica da empresa.



#### Documentação dos atores

- Uma breve descrição para cada ator deve ser adicionada ao MCU.
- O nome de um ator deve lembrar o papel desempenhado pelo mesmo.
- Exemplo

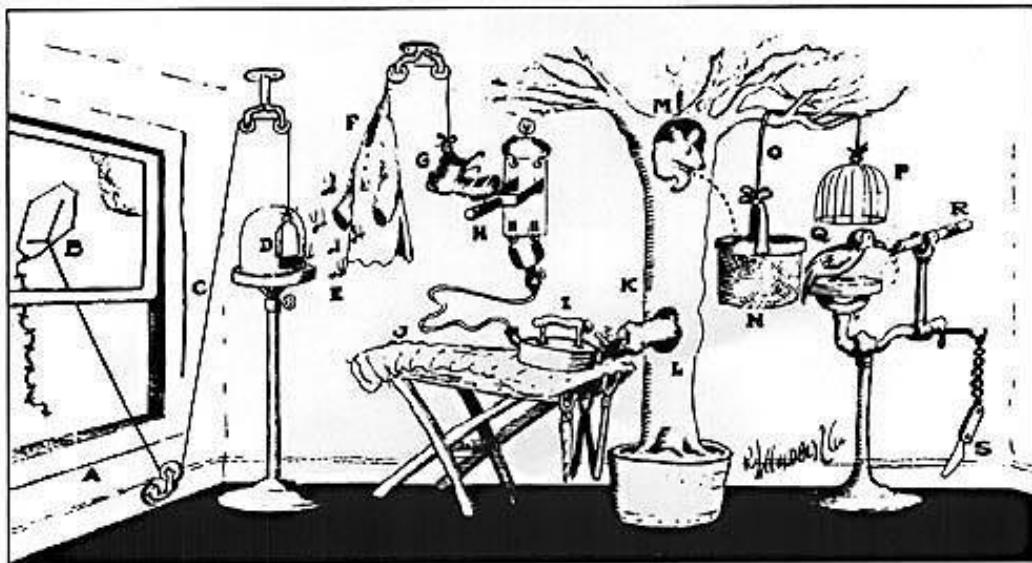
**“Aluno: representa pessoas que fazem um curso dentro da universidade.”**

## Documentação dos casos de uso

- Infelizmente, a UML não define um padrão para descrição textual dos casos de uso de um sistema.
- Por conta disso, há diversos estilos de descrição possíveis (numerada, livre, tabular, etc).
- É necessário, no entanto que a equipe de desenvolvimento padronize o seu estilo de descrição.
- Algumas seções normalmente encontradas:
  - Sumário
  - Atores
  - Fluxo principal
  - Fluxos alternativos
  - Referências cruzadas (para requisitos não funcionais)

<ul style="list-style-type: none"><li>• Nome</li><li>• Descrição</li><li>• Identificador</li><li>• Importância</li><li>• Sumário</li><li>• Ator Primário</li><li>• Atores Secundários</li><li>• Pré-condições</li></ul>	<ul style="list-style-type: none"><li>• Fluxo Principal</li><li>• Fluxos Alternativos</li><li>• Fluxos de Exceção</li><li>• Pós-condições</li><li>• Regras do Negócio</li><li>• Histórico</li><li>• Notas de Implementação</li></ul>
---	--

- Algumas boas práticas na documentação de casos de uso.
  - Comece o nome do caso de uso com um verbo no infinitivo (para indicar um processo ou ação).
  - Tente descrever os passos de caso de uso sempre na forma sujeito + predicado. Ou seja, deixe explícito quem é o agente da ação.
  - Não descreva **como** o sistema realiza internamente um passo de um caso de uso.
    - "You apply use cases to capture the intended behavior of the system [...], without having to specify how that behavior is implemented. (Booch)
  - Tente dar nomes a casos de uso seguindo perspectiva do ator primário. Foque no **objetivo** desse ator. Exemplos: Registrar Pedido, Abrir Ordem de Produção, Manter Referência, Alugar Filme, etc.
  - Tente manter a descrição de cada caso de uso no nível mais simples possível...
- ...repetindo: tente manter a descrição de cada caso de uso no nível mais simples possível!



#### 4.5 Documentação suplementar ao MCU

##### Documentação Associada

- O modelo de casos de uso força o desenvolvedor a pensar em como os agentes externos interagem com o sistema.
- No entanto, este modelo corresponde somente aos requisitos funcionais.
- Outros tipos de requisitos (desempenho, interface, segurança, regras do negócio, etc.) também devem ser identificados e modelados.
- Esses outros requisitos fazem parte da documentação associada ao MCU.
- Dois itens importantes dessa documentação associada são o **modelo de regras do negócio** e os **requisitos de desempenho**.

##### Regras do Negócio

- São políticas, condições ou restrições que devem ser consideradas na execução dos processos de uma organização.
  - Descrevem a maneira pela qual a organização funciona.
- Estas regras são identificadas e documentadas no chamado **modelo de regras do negócio** (MRN).
  - A descrição do modelo de regras do negócio pode ser feita utilizando-se texto informal, ou através de alguma forma de estruturação.
- Regras do negócio normalmente influenciam o comportamento de determinados casos de uso.
  - Quando isso ocorre, os identificadores das regras do negócio devem ser adicionados à descrição dos casos de uso em questão.
  - Uso da seção “regras do negócio” da descrição do caso de uso.

##### Exemplos de Regras do Negócio

- O valor total de um pedido é igual à soma dos totais dos itens do pedido acrescido de 10% de taxa de entrega.
- Um professor só pode estar lecionando disciplinas para as quais esteja habilitado.

- Um cliente de uma das agências do banco não pode retirar mais do que R\$ 1.000 por dia de sua conta. Após as 18h00minh, esse limite cai para R\$ 100,00.
- Os pedidos para um cliente não especial devem ser pagos antecipadamente.
- Possível formato para documentação de uma regra de negócio no MRN.

Nome	Quantidade de inscrições possíveis (RN01)
Descrição	Um aluno não pode ser inscrever em mais de seis disciplinas por semestre letivo.
Fonte	Coordenador da escola de informática
Histórico	Data de identificação: 12/07/2002

#### Requisitos de desempenho

- Conexão de casos de uso a requisitos de desempenho.

Identificador do caso de uso	Freqüência da utilização	Tempo máximo esperado	...
CSU01	5/mês	Interativo	...
CSU02	15/dia	1 segundo	...
CSU03	60/dia	Interativo	...
CSU04	180/dia	3 segundos	...
CSU05	600/mês	10 segundos	...
CSU07	500/dia durante 10 dias seguidos.	10 segundos	...

## 4.6 O MCU em um processo de desenvolvimento iterativo e incremental

### Casos de uso e outras atividades

- Validação
  - Clientes e usuários devem entender o modelo (validação) e usá-lo para comunicar suas necessidades de forma consistente e não redundante.
- Planejamento e gerenciamento do projeto
  - Uma ferramenta fundamental para o gerente de um projeto no planejamento e controle de um processo de desenvolvimento incremental e iterativo
- Testes do sistema
  - Os casos de uso e seus cenários oferecem *casos de teste*.
- Documentação do sistema para os usuários
  - Manuais e guias do usuário podem ser construídos com base nos casos de uso.
- Realização de uma iteração

- Os casos de uso podem se **alocados** entre os membros de equipe de desenvolvimento
- Essa estratégia de utilizar o MCU como ponto de partida para outras atividades é denominada **Desenvolvimento Dirigido por Casos de Uso**
  - ***Use Case Driven Development***

MCU no processo de desenvolvimento

- Casos de uso formam uma base natural através da qual podem-se realizar as iterações do desenvolvimento.
- Um grupo de casos é alocado a cada iteração.
- Em cada iteração, o grupo de casos de uso é detalhado e desenvolvido.
- O processo continua até que todos os casos de uso tenham sido desenvolvidos e o sistema esteja completamente construído.
- A descrição expandida de um caso de uso pode ser deixada para a iteração na qual este deve ser implementado.
  - Evita perda de tempo inicial no detalhamento.
  - Estratégia mais adaptável aos requisitos voláteis.
- Cantor propõe uma classificação em função do risco de desenvolvimento e das prioridades estabelecidas pelo usuário.
  - 1) Risco alto e prioridade alta
  - 2) Risco alto e prioridade baixa
  - 3) Risco baixo e prioridade alta
  - 4) Risco baixo e prioridade baixa
  - Considerando-se essa categorização, devemos considerar os casos de uso mais importantes e mais arriscados primeiramente.
    - Atacar o risco maior mais cedo...

## 5 Modelagem de Classes de Análise

*“O engenheiro de software amador está sempre à procura da mágica, de algum método sensacional ou ferramenta cuja aplicação promete tornar trivial o desenvolvimento de software. É uma característica do engenheiro de software profissional saber que tal panacéia não existe” -Grady Booch*

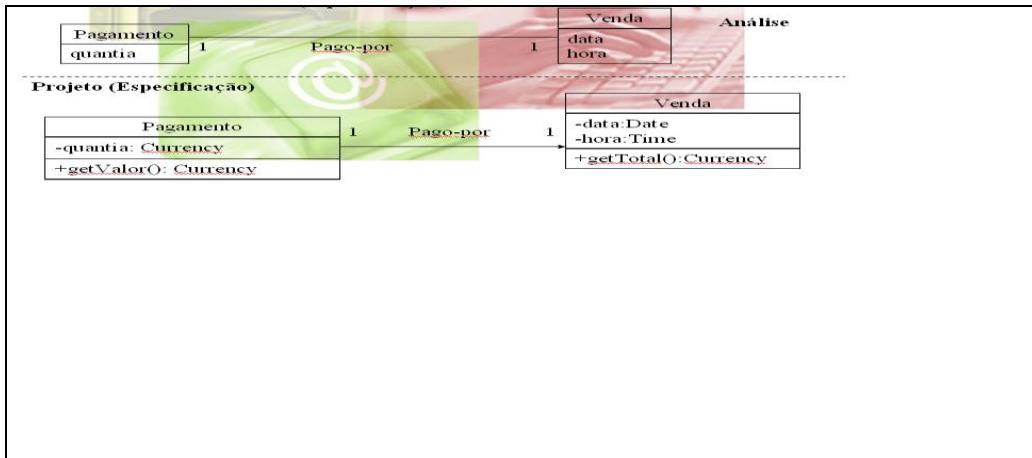
### Tópicos

- Introdução
- Diagrama de classes
- Diagrama de objetos
- Técnicas para identificação de classes
- Construção do modelo de classes
- Modelo de classes no processo de desenvolvimento
- As funcionalidades de um SSOO são realizadas internamente através de **colaborações** entre objetos.
  - Externamente, os atores visualizam resultados de cálculos, relatórios produzidos, confirmações de requisições realizadas, etc.
  - Internamente, os objetos colaboram uns com os outros para produzir os resultados.

- Essa colaboração pode ser vista sob o *aspecto dinâmico* e sob o *aspecto estrutural estático*.
- O modelo de objetos representa o aspecto estrutural e estático dos objetos que compõem um SSOO.
- Dois diagramas da UML são usados na construção do modelo de objetos:
  - Diagrama de classes
  - Diagrama de objetos
- Na prática o diagrama de classes é bem mais utilizado que o diagrama de objetos.
  - Tanto que o modelo de objetos é também conhecido como modelo de classes.
- Esse modelo *evolui* durante o desenvolvimento do SSOO.
  - À medida que o SSOO é desenvolvido, o modelo de objetos é incrementado com novos detalhes.
- Há três níveis sucessivos de detalhamento:
  - *Análise → Especificação (Projeto) → Implementação.*
- O objetivo da modelagem de classes de análise é prover respostas para as seguintes perguntas:
  - Por definição um sistema OO é composto de objetos...em um nível alto de abstração, que objetos constituem o sistema em questão?
  - Quais são as classes candidatas?
  - Como as classes do sistema estão relacionadas entre si?
  - Quais são as responsabilidades de cada classe?

#### Modelo de Classes de Análise

- Representa termos do domínio do negócio.
  - Idéias, coisas, e conceitos no mundo real.
- Objetivo: descrever o *problema* representado pelo sistema a ser desenvolvido, sem considerar características da *solução* a ser utilizada.
- É um dicionário “visual” de conceitos e informações relevantes ao sistema sendo desenvolvido.
- Duas etapas:
  - Modelo conceitual (modelo de domínio).
  - Modelo da aplicação.
- Elementos de notação do diagrama de classes normalmente usados na construção do modelo de análise:
  - Classes e atributos; associações, composições e agregações (com seus adornos); classes de associação; generalizações (herança).
- O modelo de análise não representa detalhes da solução do problema.
  - Embora este sirva de ponto de partida para uma posterior definição das classes de software (especificação).



## 5.2 Diagrama de classes

- Uma classe descreve esses objetos através de **atributos** e **operações**.
  - Atributos correspondem às informações que um objeto armazena.
  - Operações correspondem às ações que um objeto sabe realizar.
- Notação na UML: “caixa” com no máximo três compartimentos exibidos.
  - Detalhamento utilizado depende do estágio de desenvolvimento e do nível de abstração desejado.



Exemplo (classe ContaBancária)

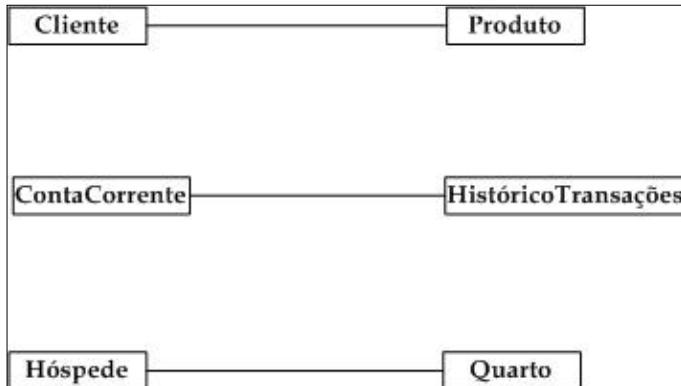
ContaBancária	ContaBancária	ContaBancária	ContaBancária	ContaBancária
	número saldo dataAbertura	criar() bloquear() desbloquear() creditar() debitar()	número saldo dataAbertura	-número : String -saldo : Quantia -dataAbertura : Date  +criar() +bloquear() +desbloquear() +creditar(in valor : Quantia) +debitar(in valor : Quantia)

## Associações

- Para representar o fato de que objetos podem se relacionar uns com os outros, utilizamos associações.
- Uma associação representa relacionamentos (ligações) que são formados entre objetos durante a execução do sistema.
- Note que, embora as associações sejam representadas entre classes do diagrama, tais associações representam ligações possíveis entre os objetos das classes envolvidas.

## Notação para Associações

- Na UML associações são representadas por uma linha que liga as classes cujos objetos se relacionam.
- Exemplos:



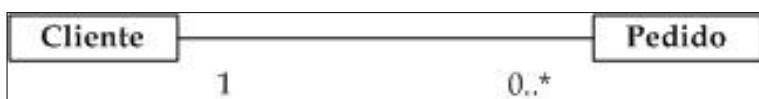
## Multiplicidades

- Representam a informação dos limites inferior e superior da quantidade de objetos aos quais outro objeto pode se associar.
- Cada associação em um diagrama de classes possui duas multiplicidades, uma em cada extremo da linha de associação.

Nome	Simbologia na UML
Apenas Um	1..1 (ou 1)
Zero ou Muitos	0..* (ou *)
Um ou Muitos	1..*
Zero ou Um	0..1
Intervalo Específico	li..ls

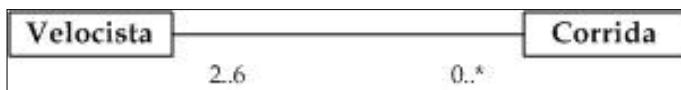
- Exemplo

- Pode haver um cliente que esteja associado a vários pedidos.
- Pode haver um cliente que não esteja associado a pedido algum.
- Um pedido está associado a um, e somente um, cliente.



- Exemplo

- Uma corrida está associada a, no mínimo, dois velocistas
- Uma corrida está associada a, no máximo, seis velocistas.
- Um velocista pode estar associado a várias corridas.

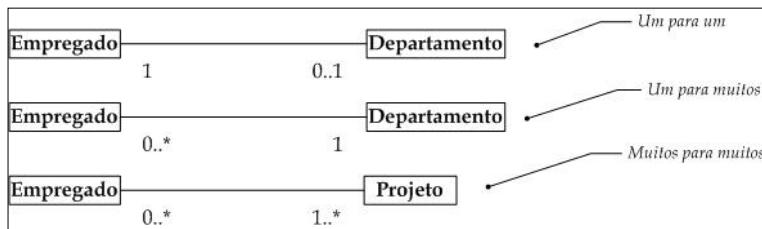


- A **conectividade** corresponde ao tipo de associação entre duas classes: “*muitos para muitos*”, “*um para muitos*” e “*um para um*”.

- A conectividade da associação entre duas classes depende dos símbolos de multiplicidade que são utilizados na associação.

Coneectividade	Em um extremo	No outro extremo
<b>Um para um</b>	<b>0..1</b> <b>1</b>	<b>0..1</b> <b>1</b>
<b>Um para muitos</b>	<b>0..1</b> <b>1</b>	* <b>1..*</b> <b>0..*</b>
<b>Muitos para muitos</b>	* <b>1..*</b> <b>0..*</b>	* <b>1..*</b> <b>0..*</b>

### Exemplo (coneectividade)

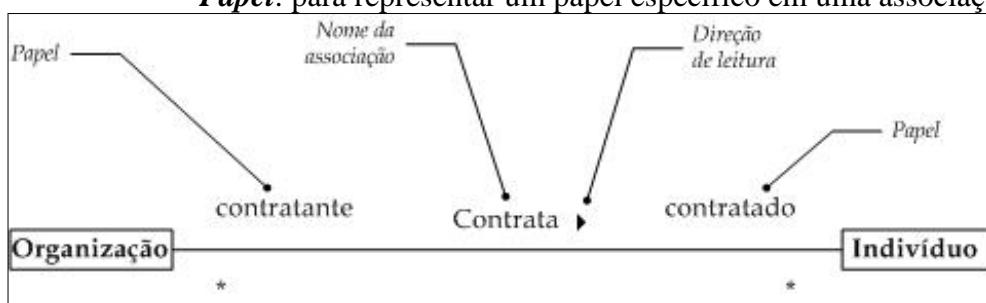


### Participação

- Uma característica de uma associação que indica a necessidade (ou não) da existência desta associação entre objetos.
- A participação pode ser *obrigatória* ou *opcional*.
  - Se o valor mínimo da multiplicidade de uma associação é igual a 1 (um), significa que a participação é *obrigatória*
  - Caso contrário, a participação é *opcional*.

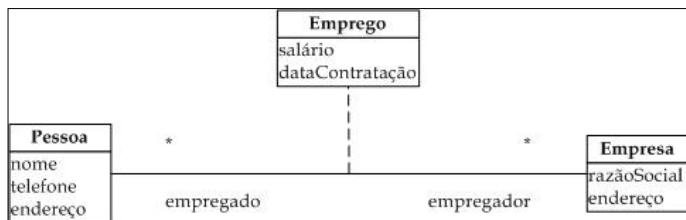
### Acessórios para Associações

- Para melhor esclarecer o significado de uma associação no diagrama de classes, a UML define três recursos de notação:
  - Nome da associação*: fornece algum significado semântico a mesma.
  - Direção de leitura*: indica como a associação deve ser lida
  - Papel*: para representar um papel específico em uma associação.



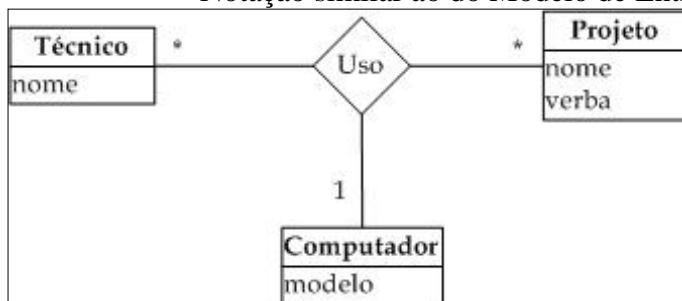
### Classe associativa

- É uma classe que está ligada a uma associação, em vez de estar ligada a outras classes.
- É normalmente necessária quando duas ou mais classes estão associadas, e é necessário manter informações sobre esta associação.
- Uma classe associativa pode estar ligada a associações de qualquer tipo de conectividade.
- Sinônimo: **classe de associação**
- Notação é semelhante à utilizada para classes ordinárias. A diferença é que esta classe é ligada a uma associação por uma linha tracejada.
- Exemplo: para cada par de objetos [pessoa, empresa], há duas informações associadas: salário e data de contratação.



### Associações n-árias

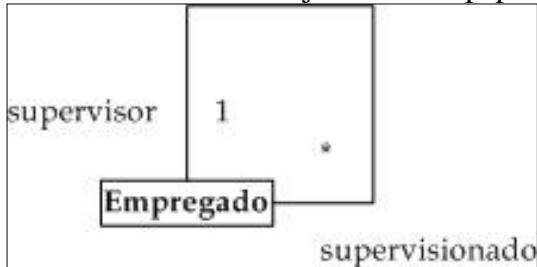
- Define-se o **grau** de uma associação como a quantidade de classes envolvidas na mesma.
- Na notação da UML, as linhas de uma **associação n-ária** se interceptam em um losango.
- Na grande maioria dos casos práticos de modelagem, as associações normalmente são **binárias**.
- Quando o grau de uma associação é igual a três, dizemos que a mesma é **ternária**.
  - Uma associação ternária são uma caso mais comum (menos raro) de associação n-ária ( $n = 3$ ).
- Na notação da UML, as linhas de uma associação n-ária se interceptam em um losango nomeado.
  - Notação similar ao do Modelo de Entidades e Relacionamentos



### Associações reflexivas

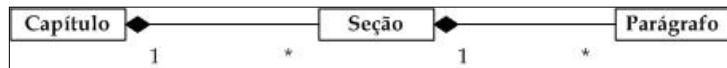
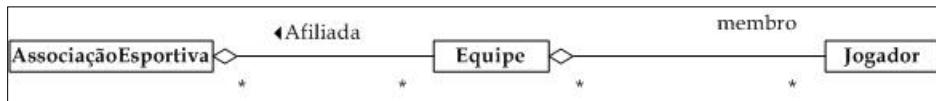
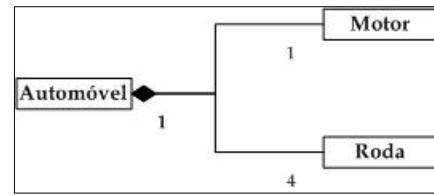
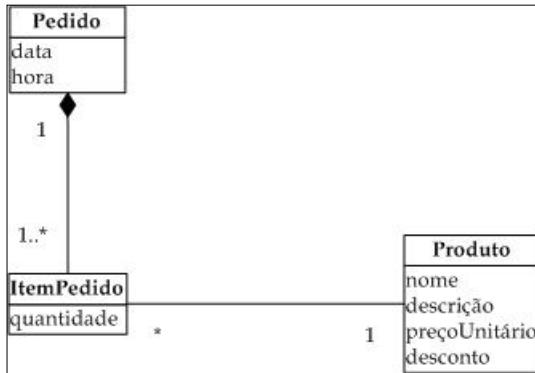
- Tipo especial de associação que representa ligações entre objetos que pertencem a uma mesma classe.
  - Não indica que um objeto se associa a ele próprio.
- Quando se usa associações reflexivas, a definição de papéis é importante para evitar ambigüidades na leitura da associação.

- Cada objeto tem um papel distinto na associação.



### Agregações e Composições

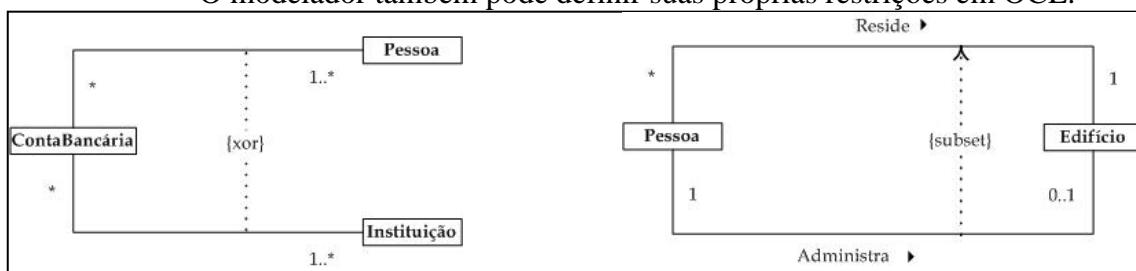
- A **semântica** de uma associação corresponde ao seu significado, ou seja, à natureza conceitual da relação que existe entre os objetos que participam daquela associação.
- De todos os significados diferentes que uma associação pode ter, há uma categoria especial de significados, que representa **relações todo-partes**.
- Uma relação todo-partes entre dois objetos indica que um dos objetos está contido no outro. Podemos também dizer que um objeto contém o outro.
- A UML define dois tipos de relacionamentos todo-partes, a **agregação** e a **composição**.
- Algumas particularidades das agregações/composições:
  - são assimétricas, no sentido de que, se um objeto A é parte de um objeto B, o objeto B não pode ser parte do objeto A.
  - propagam comportamento, no sentido de que um comportamento que se aplica a um todo automaticamente se aplica às suas partes.
  - as partes são normalmente criadas e destruídas pelo todo. Na classe do objeto todo, são definidas operações para adicionar e remover as partes.
- Se uma das perguntas a seguir for respondida com um sim, provavelmente há uma agregação onde X é todo e Y é parte.
  - *X tem um ou mais Y?*
  - *Y é parte de X?*

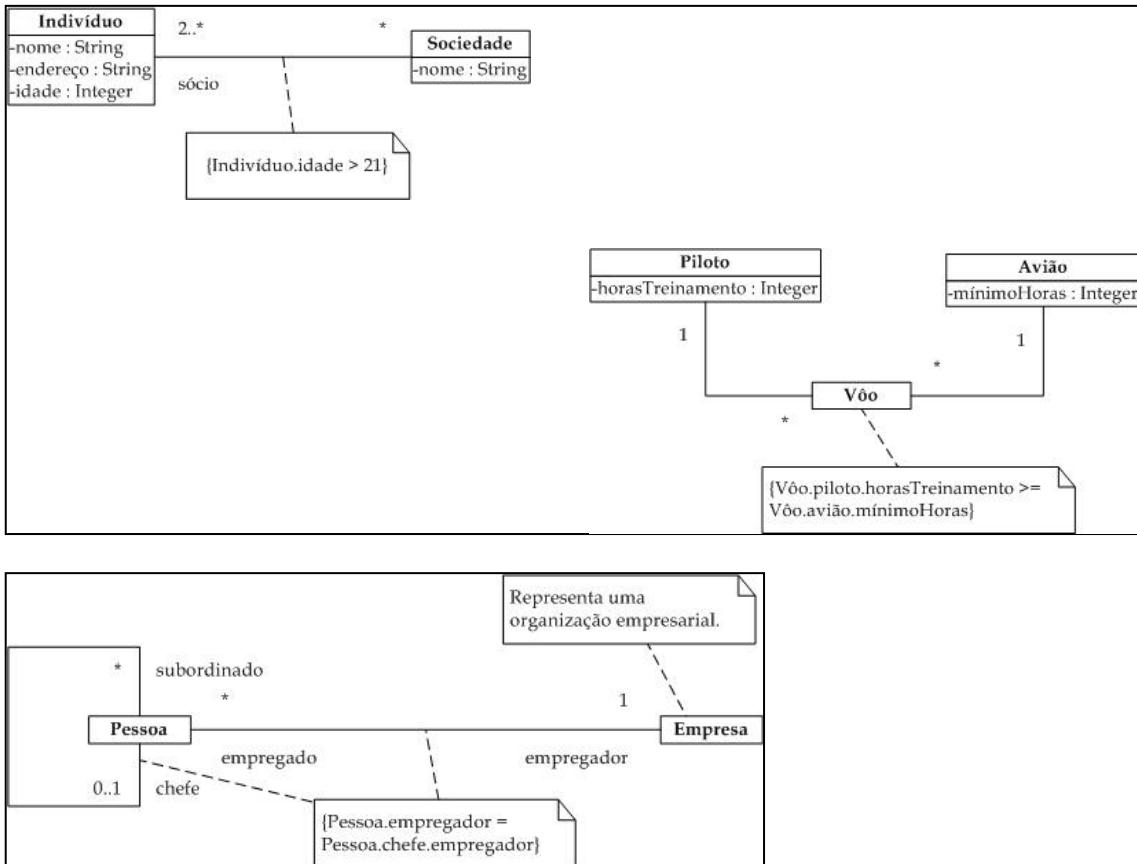


- As diferenças entre a agregação e composição não são bem definidas. A seguir, as diferenças mais marcantes entre elas.
- Destruuição de objetos
  - Na agregação, a destruição de um objeto todo não implica necessariamente na destruição do objeto parte.
- Pertinência
  - Na composição, os objetos parte pertencem a um único todo.
    - Por essa razão, a composição é também denominada agregação não-compartilhada.
  - Em uma agregação, pode ser que um mesmo objeto participe como componente de vários outros objetos.
    - Por essa razão, a agregação é também denominada agregação compartilhada.

#### Restrições sobre associações

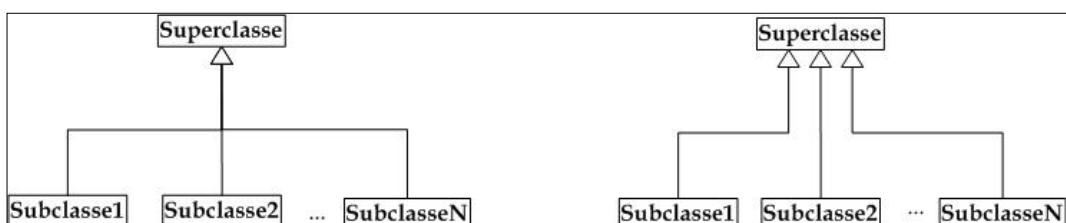
- Restrições OCL podem ser adicionadas sobre uma associação para adicionar a ela mais semântica.
  - Duas das restrições sobre associações predefinidas pela UML são **subset** e **xor**.
  - O modelador também pode definir suas próprias restrições em OCL.





## Generalizações e Especializações

- O modelador também pode representar relacionamentos entre classes.
  - Esses denotam relações de generalidade ou especificidade entre as classes envolvidas.
  - Exemplo: o conceito *mamífero* é mais genérico que o conceito *ser humano*.
  - Exemplo: o conceito *carro* é mais específico que o conceito *veículo*.
- Esse é o chamado **relacionamento de herança**.
  - relacionamento de generalização/especialização
  - relacionamento de gen/espec
- Terminologia
  - *subclasse X superclasse*.
  - *supertipo X subtipo*.
  - *classe base X classe herdeira*.
  - classe de *especialização* X classe de *generalização*.
  - *ancestral* e *descendente* (herança em vários níveis)
- Notação definida pela UML

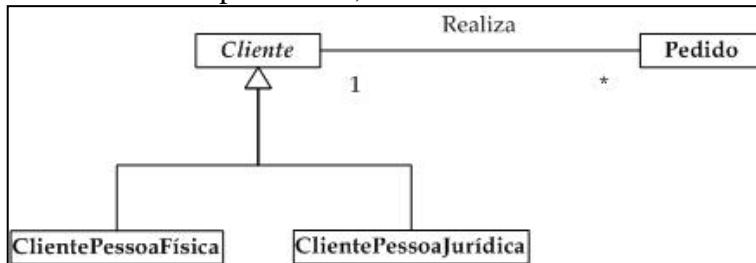


## Semântica da Herança

- Subclasses herdam as características de sua superclasse
  - É como se as características da superclasse estivessem definidas também nas suas subclasses
  - Além disso, essa herança é transitiva e anti-simétrica
- Note a diferença semântica entre a herança e a associação.
  - A primeira trata de um relacionamento entre classes, enquanto que a segunda representa relacionamentos entre instâncias de classes.
  - Na associação, objetos específicos de uma classe se associam entre si ou com objetos específicos de outras classes.
  - Exemplo:
    - Herança: “Gerentes são tipos especiais de funcionários”.
    - Associação: “Gerentes chefiam departamentos”.

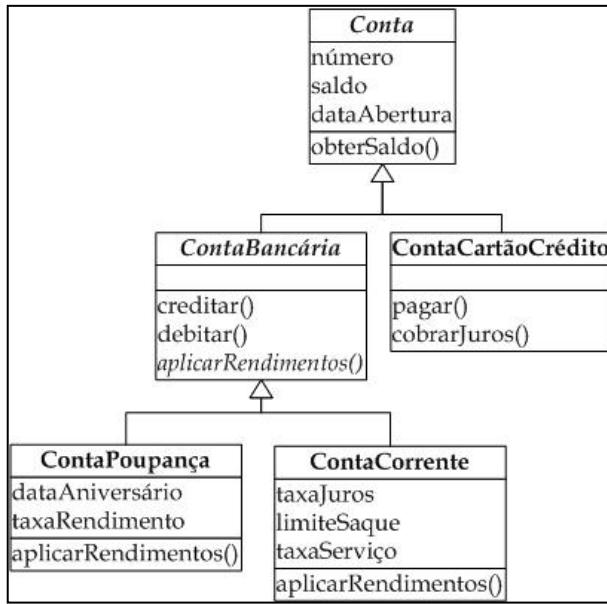
## Herança de Associações

- Não somente atributos e operações, mas também associações são herdadas pelas subclasses.
- No exemplo abaixo, cada subclass está associada a Pedido, por herança.



## Propriedades da Herança

- **Transitividade:** uma classe em uma hierarquia herda propriedades e relacionamentos de todos os seus ancestrais.
  - Ou seja, a herança pode ser aplicada em vários níveis, dando origem a *hierarquia de generalização*.
  - uma classe que herda propriedades de uma outra classe pode ela própria servir como superclasse.
- **Assimetria:** dadas duas classes A e B, se A for uma generalização de B, então B não pode ser uma generalização de A.
  - Ou seja, *não* pode haver ciclos em uma hierarquia de generalização.

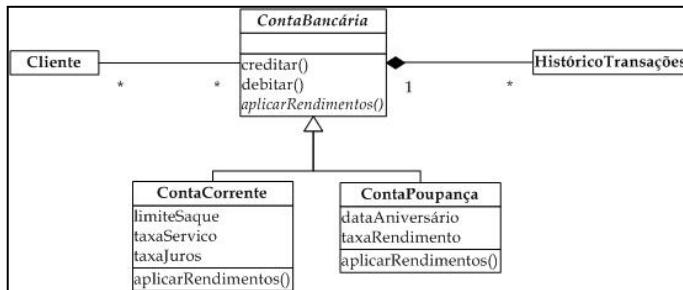


### Classes Abstratas

- Usualmente, a existência de uma classe se justifica pelo fato de haver a possibilidade de gerar instâncias da mesma
  - Essas são as **classes concretas**.
- No entanto, podem existir classes que não geram instâncias diretas.
  - Essas são as **classes abstratas**.
- Classes abstratas são utilizadas para organizar e simplificar uma hierarquia de generalização.
  - Propriedades comuns a diversas classes podem ser organizadas e definidas em uma classe abstrata a partir da qual as primeiras herdam.
- Subclasses de uma classe abstrata também podem ser abstratas, mas a hierarquia deve terminar em uma ou mais classes concretas.

### Notação para classes abstratas

- Na UML, uma classe abstrata é representada com o seu nome em *italico*.
- No exemplo a seguir, ContaBancária é uma classe abstrata.



### Refinamento do Modelo com Herança

- Critérios a avaliar na criação de subclasses:
  - A subclass tem atributos adicionais.
  - A subclass tem associações.
  - A subclass é manipulada (ou reage) de forma diferente da superclasse.

- Se algum “subconceito” (subconjunto de objetos) atenda a tem algum(ns) das critérios acima, a criação de uma subclasses deve ser considerada.
- Sempre se assegure de que se trata de um relacionamento do tipo “é-um”: X é um tipo de Y?
- A regra “é-um” é mais formalmente conhecida como regra da substituição ou princípio de Liskov.

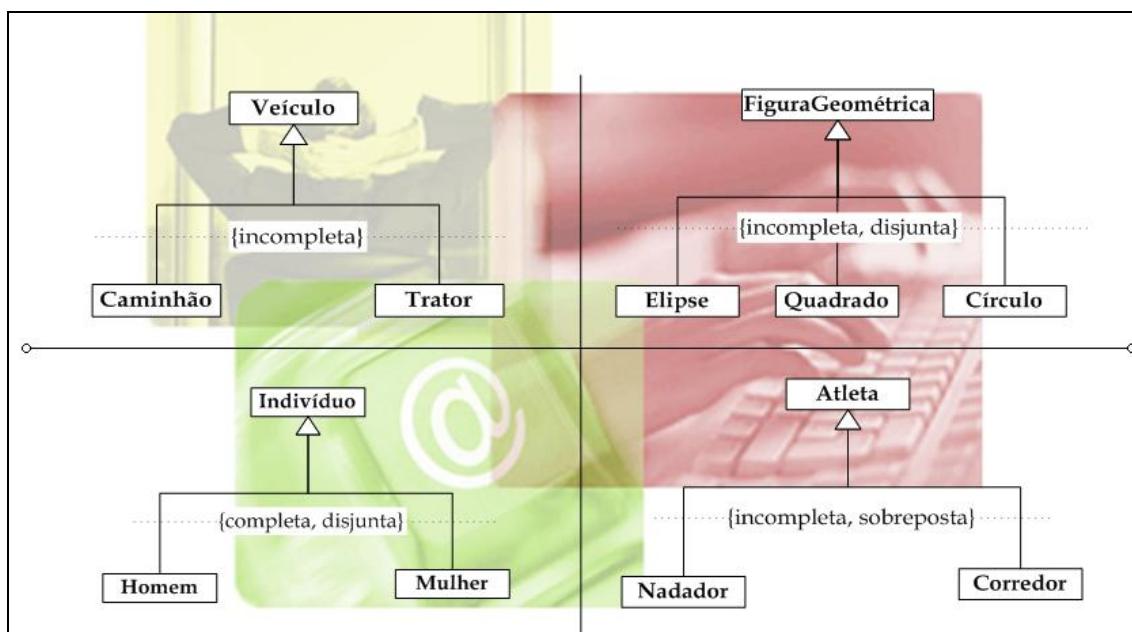


Barbara Liskov (<http://www.pmg.csail.mit.edu/~liskov/>)

**Regra da Substituição:** sejam duas classes A e B, onde A é uma generalização de B. Não pode haver diferenças entre utilizar instâncias de B ou de A, do ponto de vista dos clientes de A.

Restrições sobre gen/espec

- Restrições OCL sobre relacionamentos de herança podem ser representadas no diagrama de classes, também com o objetivo de esclarecer seu significado.
- Restrições predefinidas pela UML:
  - Sobreposta X Disjunta
  - Completa X Incompleta

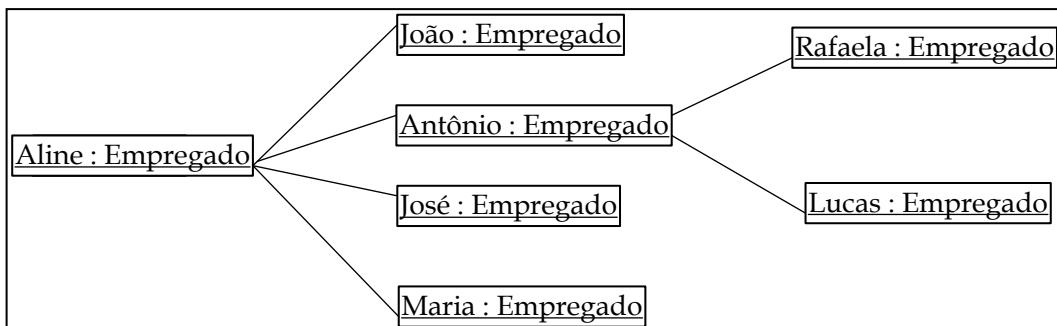
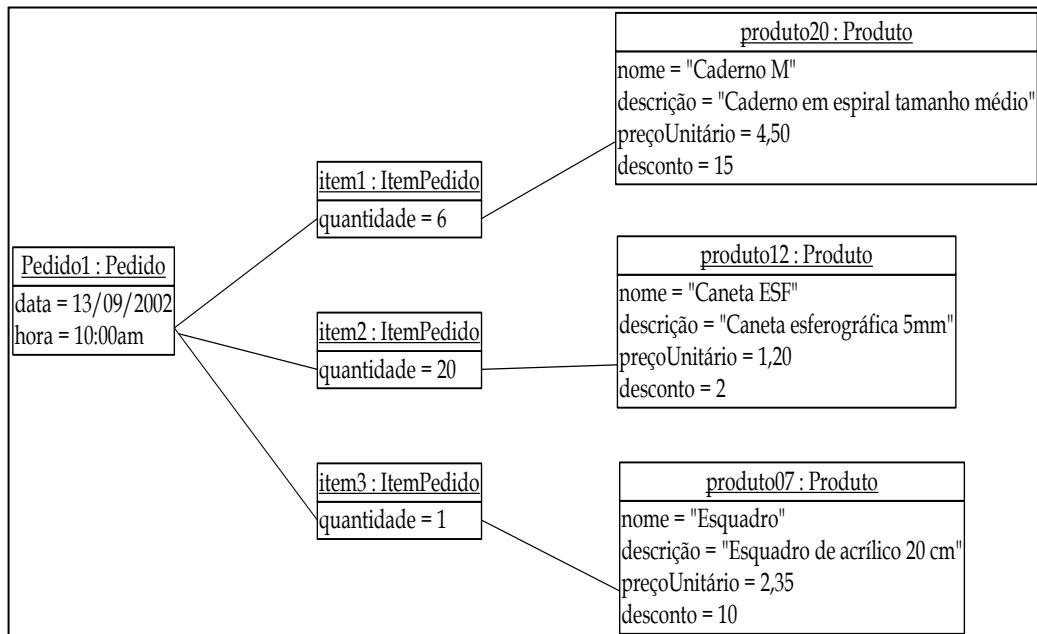


### 5.3 Diagrama de objetos

- Além do diagrama de classes, A UML define um segundo tipo de diagrama estrutural, o diagrama de objetos.
- Pode ser visto com uma instância de diagramas de classes
- Representa uma “fotografia” do sistema em um certo momento.
  - exibe as ligações formadas entre objetos conforme estes interagem e os valores dos seus atributos.

Exemplos de Diagrama de objetos:

Formato	Exemplo
<u>nomeClasse</u>	<u>Pedido</u>
<u>nomeObjeto: NomeClasse</u>	<u>umPedido: Pedido</u>



#### 5.4 Técnicas para identificação de classes

Apesar de todas as vantagens que a OO pode trazer ao desenvolvimento de software, um problema fundamental ainda persiste: identificar corretamente e completamente objetos (classes), atributos e operações.

#### Técnicas de Identificação

Várias técnicas (de uso não exclusivo) são usadas para identificar classes:

Categorias de Conceitos

Análise Textual de Abbott (*Abbot Textual Analysis*)

Análise de Casos de Uso

Categorização BCE

Padrões de Análise (Analisis Patterns)

## Identificação Dirigida a Responsabilidades

### Categorias de Conceitos

Estratégia: usar uma lista de conceitos comuns.

**Conceitos concretos.** Por exemplo, edifícios, carros, salas de aula, etc.

**Papéis** desempenhados por seres humanos. Por exemplo, professores, alunos, empregados, clientes, etc.

**Eventos**, ou seja, ocorrências em uma data e em uma hora particulares. Por exemplo, reuniões, pedidos, aterrissagens, aulas, etc.

**Lugares**: áreas reservadas para pessoas ou coisas. Por exemplo: escritórios, filiais, locais de pouso, salas de aula, etc.

**Organizações**: coleções de pessoas ou de recursos. Por exemplo: departamentos, projetos, campanhas, turmas, etc.

**Conceitos abstratos**: princípios ou idéias não tangíveis. Por exemplo: reservas, vendas, inscrições, etc.

### Análise Textual de Abbott

Estratégia: identificar termos da narrativa de casos de uso e documento de requisitos que podem sugerir classes, atributos, operações.

Nesta técnica, são utilizadas diversas fontes de informação sobre o sistema: documento e requisitos, modelos do negócio, glossários, conhecimento sobre o domínio, etc.

Para cada um desses documentos, os nomes (substantivos e adjetivos) que aparecem no mesmo são destacados. (São também consideradas locuções equivalentes a substantivos.)

Após isso, os sinônimos são removidos (permanecem os nomes mais significativos para o domínio do negócio em questão).

Cada termo remanescente se encaixa em uma das situações a seguir:

O termo se torna uma classe (ou seja, são classes candidatas);

O termo se torna um atributo;

O termo não tem relevância alguma com ao SSOO.

Abbott também preconiza o uso de sua técnica na identificação de operações e de associações.

Para isso, ele sugere que destaquemos os verbos no texto.

Verbos de ação (e.g., calcular, confirmar, cancelar, comprar, fechar, estimar, depositar, sacar, etc.) são operações em potencial.

Verbos com sentido de “ter” são potenciais agregações ou composições.

Verbos com sentido de “ser” são generalizações em potencial.

Demais verbos são associações em potencial.

A ATA é de aplicação bastante simples.

No entanto, uma desvantagem é que seu resultado (as classes candidatas identificadas) depende de os documentos utilizados como fonte serem completos.

Dependendo do estilo que foi utilizado para escrever esse documento, essa técnica pode levar à identificação de diversas classes candidatas que não gerarão classes.

A análise do texto de um documento pode não deixar explícita uma classe importante para o sistema.

Em linguagem natural, as variações lingüísticas e as formas de expressar uma mesma idéia são bastante numerosas.

## Análise de Casos de Uso

Essa técnica é também chamada de identificação dirigida por casos de uso, e é um caso particular da ATA.

Técnica preconizada pelo Processo Unificado.

Nesta técnica, o MCU é utilizado como ponto de partida.

Premissa: um caso de uso corresponde a um comportamento específico do SSOO. Esse comportamento somente pode ser produzido por objetos que compõem o sistema.

Em outras palavras, a realização de um caso de uso é responsabilidade de um conjunto de objetos que devem colaborar para produzir o resultado daquele caso de uso.

Com base nisso, o modelador aplica a técnica de análise dos casos de uso para identificar as classes necessárias à produção do comportamento que está documentado na descrição do caso de uso.

Procedimento de aplicação:

O modelador estuda a descrição textual de cada caso de uso para identificar classes candidatas.

Para cada caso de uso, se texto (fluxos principal, alternativos e de exceção, pós-condições e pré-condições, etc.) é analisado.

Na análise de certo caso de uso, o modelador tenta identificar classes que possam fornecer o comportamento do mesmo.

Na medida em que os casos de uso são analisados um a um, as classes do SSOO são identificadas.

Quando todos os casos de uso tiverem sido analisados, todas as classes (ou pelo menos a grande maioria delas) terão sido identificadas.

Na aplicação deste procedimento, podemos utilizar as **categorização BCE**...

### Categorização BCE

Na categorização BCE, os objetos de um SSOO são agrupados de acordo com o tipo de responsabilidade a eles atribuída.

- Objetos de entidade: usualmente objetos do domínio do problema
- Objetos de fronteira: atores interagem com esses objetos
- Objetos de controle: servem como intermediários entre objetos de fronteira e de entidade, definindo o comportamento de um caso de uso específico.

Categorização proposta por Ivar Jacobson em 1992.

Possui correspondência (mas não equivalência!) com o framework *model-view-controller* (MVC)

Ligações entre análise (o que; problema) e projeto (como; solução)

Estereótipos na UML: «boundary», «entity», «control»



Objetos de Entidade



Repositório para **informações** e as **regras de negócio** manipuladas pelo sistema.

Representam conceitos do domínio do negócio.

Características

Normalmente armazenam informações persistentes.

Várias instâncias da mesma entidade existindo no sistema.

Participam de vários casos de uso e têm ciclo de vida longo.

Exemplo:

Um objeto *Pedido* participa dos casos de uso *Realizar Pedido* e *Atualizar Estoque*. Este objeto pode existir por diversos anos ou mesmo tanto quanto o próprio sistema.



Objetos de Fronteira



Realizam a comunicação do sistema com os atores.

Traduzem os eventos gerados por um ator em eventos relevantes ao sistema → eventos de sistema.

Também são responsáveis por apresentar os resultados de uma interação dos objetos em algo inteligível pelo ator.

Existem para que o sistema se comunique com o mundo exterior.

Por consequência, são altamente dependentes do ambiente.

Há dois tipos principais de objetos de fronteira:

Os que se comunicam com o usuário (atores humanos): relatórios, páginas HTML, interfaces gráficas desktop, etc.

Os que se comunicam com atores não-humanos (outros sistemas ou dispositivos): protocolos de comunicação.



Objetos de Controle



É a “ponte de comunicação” entre objetos de fronteira e objetos de entidade.

Responsáveis por controlar a lógica de execução correspondente a um caso de uso.

Decidem o que o sistema deve fazer quando um evento de sistema ocorre.

Eles realizam o controle do processamento

Agem como **gerentes** (coordenadores, controladores) dos outros objetos para a realização de um caso de uso.

Traduzem **eventos de sistema** em operações que devem ser realizadas pelos demais objetos.

#### Importância da Categorização BCE

A categorização BCE parte do princípio de que cada objeto em um SSOO é especialista em realizar um de três tipos de tarefa, a saber:

- Se comunicar com atores (**fronteira**),
- Manter as informações (**entidade**) ou
- Coordenar a realização de um caso de uso (**controle**).

A categorização BCE é uma “receita de bolo” para identificar objetos participantes da realização de um caso de uso.

A importância dessa categorização está relacionada à capacidade de adaptação a eventuais mudanças.

- Se cada objeto tem atribuições específicas dentro do sistema, mudanças podem ser menos complexas e mais localizadas.
- Uma modificação em uma parte do sistema tem menos possibilidades de resultar em mudanças em outras partes.

## Visões de Classes Participantes

Uma Visão de Classes Participantes (VCP) é um diagrama das classes cujos objetos participam da realização de determinado caso de uso.

- É uma recomendação do UP (Unified Process). UP: “definir uma VCP por caso de uso”

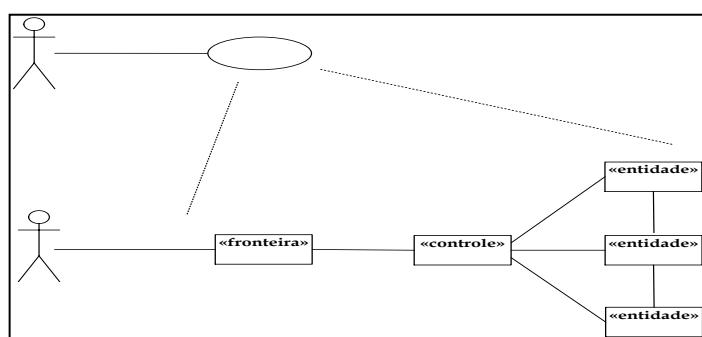
Termo original: *View Of Participating Classes* (VOPC).

Em uma VCP, são representados objetos de fronteira, de entidade e de controle para um caso de uso particular.

Uma VCP é definida através da utilização da categorização BCE previamente descrita...

### Estrutura de uma VCP

- Uma VCP representa a estrutura das classes que participam da realização de um caso de uso em particular.



### Construção de uma VCP

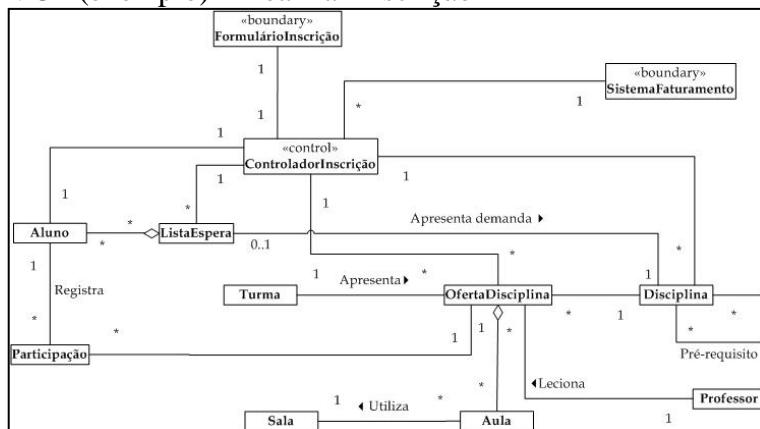
Para cada caso de uso:

- Adicione uma fronteira para cada elemento de interface gráfica principal, tais com uma tela (formulário) ou relatório.
- Adicione uma fronteira para cada ator não-humano (por exemplo, outro sistema).
- Adicione um ou mais controladores para gerenciar o processo de realização do caso de uso.
- Adicione uma entidade para cada conceito do negócio.

Esses objetos são originários do modelo conceitual.

Os estereótipos gráficos definidos pela UML podem ser utilizados.

### VCP (exemplo) – Realizar Inscrição



### Regras Estruturais em uma VCP

Durante a fase de análise, use as regras a seguir para definir a VCP para um caso de uso.

Atores somente podem interagir com objetos de fronteira.

Objetos de fronteira somente podem interagir com controladores e atores.

Objetos de entidade somente podem interagir (receber requisições) com controladores.

Controladores somente podem interagir com objetos de fronteira e objetos de entidade, e com (eventuais) outros controladores.

### Padrões de Análise

Após produzir diversos modelos para um mesmo domínio, é natural que um modelador comece a identificar características comuns entre esses modelos.

Em particular, um mesmo conjunto de classes ou colaborações entre objetos costuma recorrer, com algumas pequenas diferenças, em todos os sistemas desenvolvidos para o domínio em questão.

Quantos modelos de classes já foram construídos que possuem os conceitos Cliente, Produto, Fornecedor, Departamento, etc?

Quantos outros já foram construídos que possuíam os conceitos Ordem de Compra, Ordem de Venda, Orçamento, Contrato, etc?

A identificação de características comuns acontece em consequência do ganho de experiência do modelador em determinado domínio de problema.

Ao reconhecer processos e estruturas comuns em um domínio, o modelador pode descrever (catalogar) a essência dos mesmos, dando origem a **padrões de software**.

Quando o problema correspondente ao descrito em um padrão de software acontecer novamente, um modelador pode utilizar a solução descrita no catálogo.

A aplicação desse processo permite que o desenvolvimento de determinado aspecto de um SSOO seja feito de forma mais rápida e menos suscetível a erros.

- Um padrão de software pode então ser definido como uma descrição essencial de um problema recorrente no desenvolvimento de software.

*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*

<http://www.patternlanguage.com/leveltwo/ca.htm>

**“Cada padrão descreve um problema que ocorre freqüentemente no nosso ambiente, e então descreve o núcleo de uma solução para tal problema. Esse núcleo pode ser utilizado um milhão de vezes, sem que haja duas formas de utilização iguais.”**

*Christopher Alexander*

Padrões de software podem ser utilizados em diversas atividades e existem em diversos níveis de abstração.

**Padrões de Análise** (Analysis Patterns)

**Padrões de Projeto** (Design Patterns)

**Padrões Arquiteturais** (Architectural Patterns)

**Idiomas** (Idioms)

**Anti-padrões** (Anti-patterns)

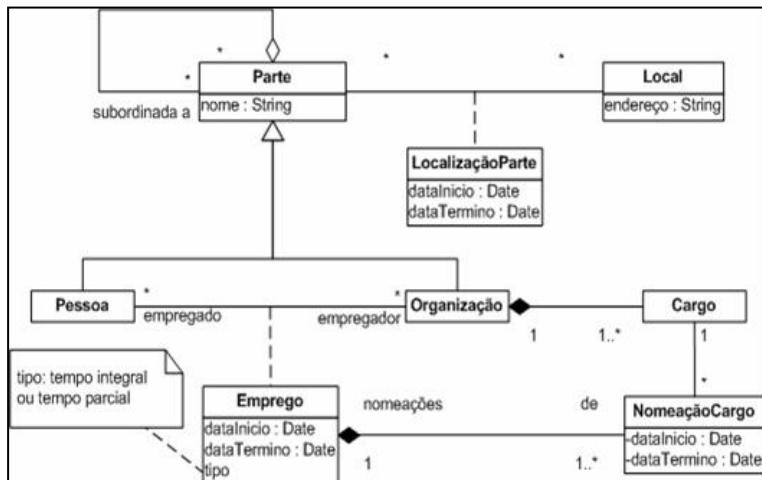
Um padrão de software pode então ser definido como uma descrição essencial de um problema recorrente no desenvolvimento de software.

Padrões de software vêm sendo estudados por anos e são atualmente utilizados em diversas fases do desenvolvimento.

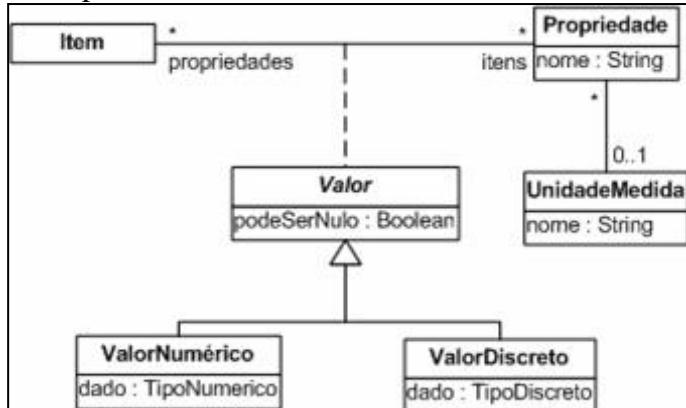
Padrões de software utilizados na fase de análise de um SSOO são chamados **padrões de análise**.

Um padrão de análise normalmente é composto, dentre outras partes, de um fragmento de diagrama de classes que pode ser customizado para uma situação de modelagem em particular.

Exemplo 1 – Padrão Party



Exemplo 2 – Padrão Metamodel



### Identificação Dirigida a Responsabilidades

Nesta técnica, a ênfase está na identificação de classes a partir de seus comportamentos relevantes para o sistema.

O esforço do modelador recai sobre a identificação das *responsabilidades* que cada classe deve ter dentro do sistema.

Método foi proposto por Rebecca Wirfs-Brock et al (*RDD*).

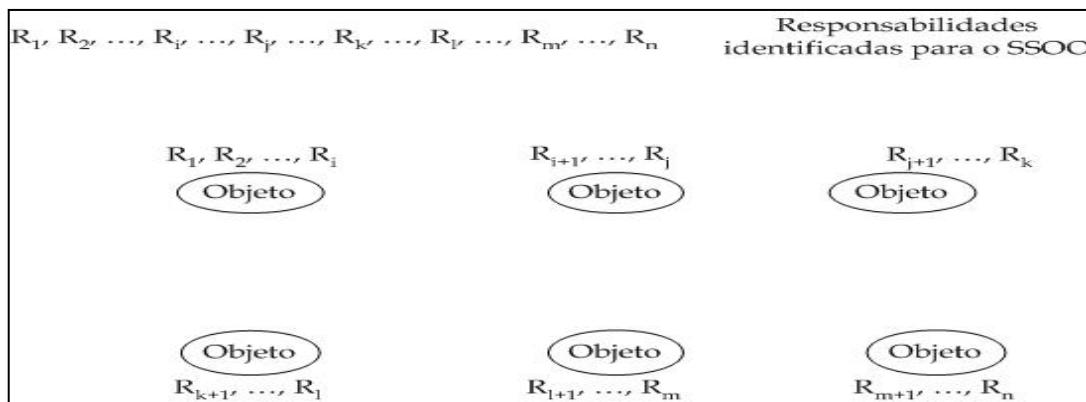
*“O método dirigido a responsabilidades enfatiza o encapsulamento da estrutura e do comportamento dos objetos.”*

Essa técnica enfatiza o **princípio do encapsulamento**:

A ênfase está na identificação das responsabilidades de uma classe que são úteis externamente à mesma.

Os detalhes internos à classe (*como* ela faz para cumprir com suas responsabilidades) devem ser abstraídos.

As responsabilidades de um SSOO devem ser alocadas aos objetos (classes) componentes do mesmo.



### Responsabilidades de uma Classe

Em um SSOO, objetos encapsulam comportamento.

O comportamento de um objeto é definido de tal forma que ele possa cumprir com suas **responsabilidades**.

Uma responsabilidade é uma obrigação que um objeto tem para com o sistema no qual ele está inserido.

Através delas, um objeto colabora (ajuda) com outros para que os objetivos do sistema sejam alcançados.

Na prática, uma responsabilidade é alguma coisa que um objeto conhece ou sabe fazer (sozinho ou “pedindo ajuda”).

Se um objeto tem uma responsabilidade com a qual não pode cumprir sozinho, ele deve requisitar colaborações de outros objetos.

### Responsabilidades e Colaboradores

Exemplo: considere clientes e seus pedidos:

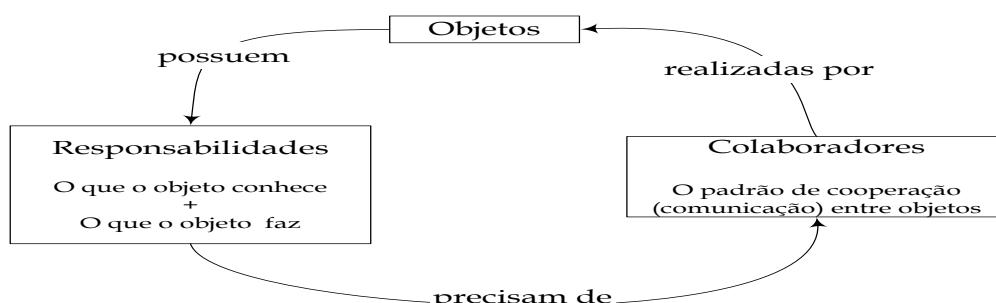
Um objeto Cliente *conhece* seu nome, seu endereço, seu telefone, etc.

Um objeto Pedido *conhece* sua data de realização, *conhece* o seu cliente, *conhece* os seus itens componentes e *sabe fazer* o cálculo do seu total.

Exemplo: quando a impressão de uma fatura é requisitada em um sistema de vendas, vários objetos precisam colaborar:

- Um objeto Pedido pode ter a responsabilidade de fornecer o seu valor total
- Um objeto Cliente fornece seu nome
- Cada itempedido informa a quantidade correspondente e o valor de seu subtotal
- Os objetos Produto também colaboraram fornecendo seu nome e preço unitário.

Um objeto cumpre com suas responsabilidades através das informações que ele possui ou das informações que ele pode derivar a partir de colaborações com outros objetos.



*Pense em um SSOO como uma sociedade onde os cidadãos (colaboradores) são objetos.*

### Modelagem CRC

A identificação dirigida a responsabilidades normalmente utiliza uma técnica de modelagem que facilita a participação de especialistas do domínio e analistas.

Essa técnica é denominada *modelagem CRC* (CRC modeling).

CRC: Class, Responsibility, Collaboration

A modelagem CRC foi proposta em 1989 por Kent Beck e Ward Cunningham.

A princípio, essa técnica de modelagem foi proposta como uma forma de ensinar o paradigma OO a iniciantes.

Contudo, sua simplicidade de notação a tornou particularmente interessante para ser utilizada na identificação de classes.

### Sessão CRC

Para aplicar essa técnica, esses profissionais se reúnem em uma sala, onde tem início uma **sessão CRC**.

Uma sessão CRC é uma reunião que envolve cerca de meia dúzia de pessoas.

Entre os participantes estão especialistas de domínio, projetistas, analistas e o moderador da sessão.

A cada pessoa é entregue um cartão de papel que mede aproximadamente 10cm x 15cm. Uma vez distribuídos os cartões pelos participantes, um conjunto de cenários de determinado caso de uso é selecionado.

Então, para cada cenário desse conjunto, uma sessão CRC é iniciada.

(Se o caso de uso não for tão complexo, ele pode ser analisado em uma única sessão.)

Um cartão CRC é dividido em várias partes.

Na parte superior do cartão, aparece o nome de uma classe.

A parte inferior do cartão é dividida em duas colunas.

Na coluna da esquerda, o indivíduo ao qual foi entregue o cartão deve listar as responsabilidades da classe.

Na coluna direita, o indivíduo deve listar os nomes de outras classes que colaboram com a classe em questão para que ela cumpra com suas responsabilidades.

### Modelagem CRC

Normalmente já existem algumas classes candidatas para determinado cenário.

Identificadas através de outras técnicas.

A sessão CRC começa com algum dos participantes simulando o ator primário que dispara a realização do caso de uso.

Na medida em que esse participante simula a interação do ator com o sistema, os demais participantes encenam a colaboração entre objetos que ocorre internamente ao sistema.

Através dessa encenação dos participantes da sessão CRC, as classes, responsabilidades e colaborações são identificadas.

## Estrutura de um Cartão CRC

Nome da classe	
Responsabilidades	Colaboradores
1 <sup>a</sup> responsabilidade	1º colaborador
2 <sup>a</sup> responsabilidade	2º colaborador
...	...
n-ésima responsabilidade	m-ésimo colaborador

Disciplina	
Responsabilidades	Colaboradores
<ul style="list-style-type: none"> <li>1. Conhecer seus pré-requisitos</li> <li>2. Conhecer seu código</li> <li>3. Conhecer seu nome</li> <li>4. Conhecer sua quantidade de créditos</li> </ul>	Disciplina

Aluno	
Responsabilidades	Colaboradores
<ul style="list-style-type: none"> <li>1. Conhecer seu número de registro</li> <li>2. Conhecer seu nome</li> <li>3. Conhecer as disciplinas que já cursou</li> </ul>	Participação

## Criação de Novos Cartões

Durante uma sessão CRC, para cada responsabilidade atribuída a uma classe, o seu proprietário deve questionar se tal classe é capaz de cumprir com a responsabilidade sozinha.

Se ela precisar de ajuda, essa ajuda é dada por um colaborador.

Os participantes da sessão, então, decidem que outra classe pode fornecer tal ajuda.

Se essa classe existir, ela recebe uma nova responsabilidade necessária para que ela forneça ajuda.

Caso contrário, um novo cartão (ou seja, uma nova classe) é criado para cumprir com tal responsabilidade de ajuda.

## 5.5 Construção do modelo de classes

Após a identificação de classes, o modelador deve verificar a consistência entre as classes para eliminar incoerências e redundâncias.

Como dica, o modelador deve estar apto a declarar as razões de existência de cada classe identificada.

Depois disso, os analistas devem começar a definir o mapeamento das responsabilidades e colaboradores de cada classe para os elementos do diagrama de classes.

Esse mapeamento resulta em um diagrama de classes que apresenta uma estrutura estática relativa a todas as classes identificadas como participantes da realização de um ou mais casos de uso.

### Definição de propriedades

Uma responsabilidade de conhecer é mapeada para um ou mais atributos.

Operações de uma classe são um modo mais detalhado de explicitar as responsabilidades de fazer.

Uma operação pode ser vista como uma contribuição da classe para uma tarefa mais complexa representada por um caso de uso.

Uma definição mais completa das operações de uma classe só pode ser feita após a construção dos **diagramas de interação**.

### Definição de associações

O fato de uma classe possuir colaboradores indica que devem existir relacionamentos entre estes últimos e a classe.

Isto porque um objeto precisa conhecer o outro para poder lhe fazer requisições.

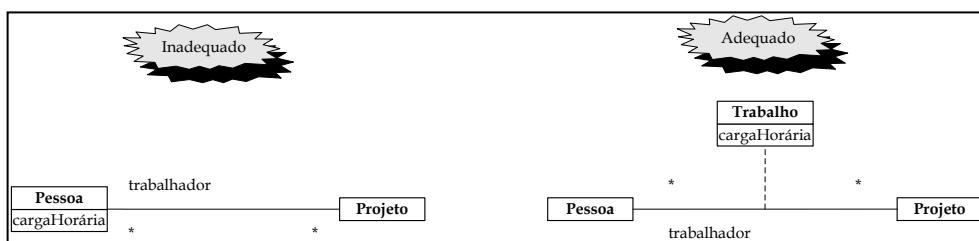
Portanto, para criar associações, verifique os colaboradores de uma classe.

O raciocínio para definir associações reflexivas, ternárias e agregações é o mesmo.

### Definição de classes associativas

Surge a partir de responsabilidades de conhecer que o modelador não conseguiu atribuir a alguma classe.

(mais raramente, de responsabilidades *de fazer*)



### Organização da documentação

As responsabilidades e colaboradores mapeados para elementos do modelo de classes devem ser organizados em um diagrama de classes e documentados, resultando no **modelo de classes de domínio**.

Podem ser associados estereótipos predefinidos às classes identificadas:

<<fronteira>>

<<entidade>>

<<controle>>

A construção de um único diagrama de classes para todo o sistema pode resultar em um diagrama bastante complexo. Um alternativa é criar uma visão de classes participantes (VCP) para cada caso de uso.

Em uma VCP, são exibidos os objetos que participam de um caso de uso.

As VCPs podem ser reunidas para formar um único diagrama de classes para o sistema como um todo.

## 5.6 Modelo de classes no processo de desenvolvimento

Em um desenvolvimento dirigido a casos de uso, após a descrição dos casos de uso, é possível iniciar a identificação de classes.

As classes identificadas são refinadas para retirar inconsistências e redundâncias.

As classes são documentadas e o diagrama de classes inicial é construído, resultando no modelo de classes de domínio.

Inconsistências nos modelos devem ser verificadas e corrigidas.

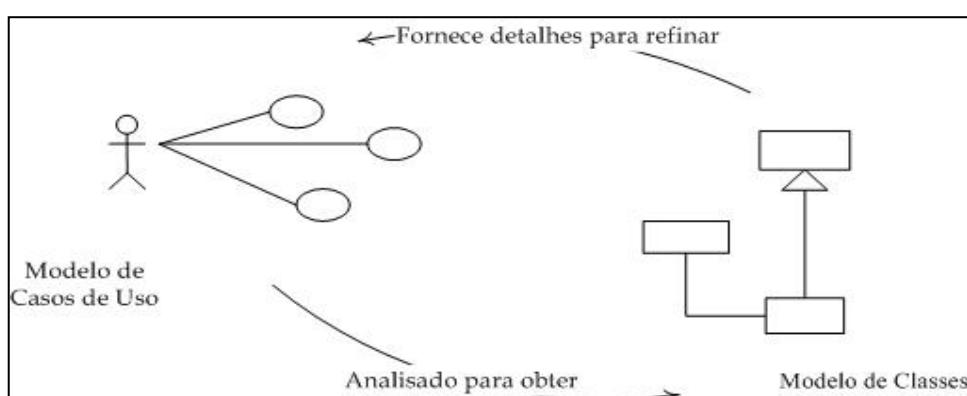
As construções do modelo de casos de uso e do modelo de classes são retroativas uma sobre a outra.

Durante a aplicação de alguma técnica de identificação, novos casos de uso podem ser identificados

Pode-se identificar a necessidade de modificação de casos de uso preexistentes.

Depois que a primeira versão do modelo de classes de análise está completa, o modelador deve retornar ao modelo de casos de uso e verificar a consistência entre os dois modelos.

Interdependência entre o modelo de casos de uso e o modelo de classes.



## 6 Passando da análise ao projeto

*“Há duas maneiras de fazer o projeto de um sistema de software. Uma delas é fazê-lo tão simples que obviamente não há deficiências. E a outra é fazê-lo tão complexo que não há deficiências óbvias. O primeiro método é de longe o mais difícil.”*

-C.A.R. Hoare

Da análise ao projeto

- No desenvolvimento de um SSOO, a mesma representação para as classes é utilizada durante a análise e o projeto desse sistema.
  - Vantagem: há uma uniformidade na modelagem do sistema.
  - Desvantagem: torna menos nítida a separação entre o que é feito na análise e o que é feito no projeto.

- Na fase de análise, estamos interessados em identificar as funcionalidades e classes do SSOO.
  - Modelo de casos de uso
  - Modelo de classes de análise
  - Modelo de interações de análise
- O modelo de classes de análise e o modelo de casos de uso esclarecem o problema a ser resolvido.
- O modelo de interações também deve começar na fase de análise para representar os aspectos dinâmicos do sistema.
- No entanto, esses modelos são insuficientes para se ter uma visão completa do SSOO para que a implementação comece.
  - Antes disso, diversos aspectos referentes à *solução* a ser utilizada devem ser definidos.
  - É na fase de *projeto* que essas definições são realmente feitas.
- Na fase de *projeto*, o interesse recai sobre refinar os modelos de análise.
  - Objetivo: encontrar alternativas para que o SSOO atenda aos requisitos funcionais, ao mesmo tempo em que respeite as restrições definidas pelos requisitos não-funcionais.
- Portanto, o foco da fase de *projeto* é definir a *solução* do problema relativo ao desenvolvimento do SSOO.
- Além disso, essa fase deve aderir a certos *princípios de projeto* para alcançar uma qualidade desejável no produto de software final.
- Após a realização do projeto de um SSOO, os modelos que resultarem estarão em um nível de detalhamento grande e suficiente para que o sistema possa ser implementado.
- As principais atividades realizadas na fase de projeto são:
  1. Detalhamento dos aspectos dinâmicos do sistema.
  2. Refinamento dos aspectos estáticos e estruturais do sistema.
  3. Detalhamento da arquitetura do sistema.
  4. Definição das estratégias para armazenamento, gerenciamento e persistência dos dados manipulados pelo sistema.
  5. Realização do projeto da interface gráfica com o usuário.
  6. Definição dos algoritmos a serem utilizados na implementação.

## 7 Modelagem de Interações

*“Somente após a construção de diagramas de interação para os cenários de um caso de uso, pode-se ter certeza de que todas as responsabilidades que os objetos devem cumprir foram identificadas”*

*-Ivar Jacobson.*

### Tópicos

- Introdução
- Diagrama de seqüência
- Diagrama de comunicação
- Modularização de interações
- Construção do modelo de interações
- Modelo de interações em um processo iterativo

## Introdução

- O objetivo dos modelos vistos até agora é fornecer um entendimento do problema correspondente ao SSOO a ser desenvolvido.
- Entretanto, esses modelos deixam algumas perguntas sem respostas.
- No modelo de casos de uso:
  - Quais são as operações que devem ser executadas internamente ao sistema?
  - A que classes estas operações pertencem?
  - Quais objetos participam da realização deste caso de uso?
- No modelo de classes de análise:
  - De que forma os objetos colaboram para que um determinado caso de uso seja realizado?
  - Em que ordem as mensagens são enviadas durante esta realização?
  - Que informações precisam ser enviadas em uma mensagem de um objeto a outro?
  - Será que há responsabilidades ou mesmo classes que ainda não foram identificadas?
- Sessões CRC podem ajudar a identificar quais são as responsabilidades de cada objeto e com que outros objetos ele precisa colaborar.
  - Mas sessões CRC não fornecem um modo de documentar essas interações.
- Para responder às questões anteriores, o **modelo de interações** deve ser criado.
- Esse modelo representa mensagens trocadas entre objetos para a execução de cenários dos casos de uso do sistema.
- A construção dos **diagramas de interação** é uma consolidação do entendimento dos aspectos dinâmicos do sistema, iniciado nas sessões CRC.
- A modelagem de interações é uma parte da **modelagem dinâmica** de um SSOO.

Diagramas de interação representam como o sistema age internamente para que um ator atinja seu objetivo na realização de um caso de uso. A modelagem de um SSOO normalmente contém diversos diagramas de interação. O conjunto de todos os diagramas de interação de um sistema constitui o seu **modelo de interações**.

- Os objetivos da construção do modelo de interação são:
  1. Obter informações adicionais para completar e aprimorar outros modelos (principalmente o modelo de classes)
    - Quais as operações de uma classe?
    - Quais os objetos participantes da realização de um caso de uso (ou cenário deste)?
    - Para cada operação, qual a sua assinatura?
    - Uma classe precisa de mais atributos?
  2. Fornecer aos programadores uma visão detalhada dos objetos e mensagens envolvidos na realização dos casos de uso.

## Mensagem

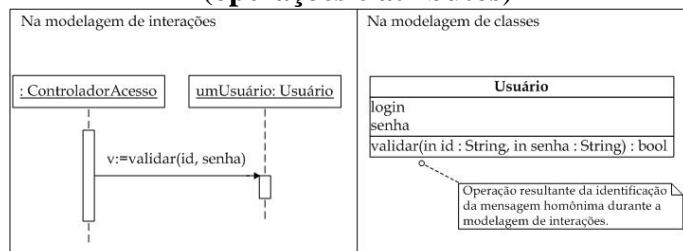
- O conceito básico da interação entre objetos é a **mensagem**.
- Um sistema OO é uma rede de objetos que trocam mensagens.
  - Funcionalidades são realizadas pelos objetos, que só podem interagir através de mensagens.

- Um objeto envia uma mensagem para outro objeto quando o primeiro deseja que o segundo realize alguma tarefa.
- O fato de um objeto “precisar de ajuda” indica a necessidade de este enviar mensagens.
- Na construção de diagramas de interação, mensagens de um objeto a outro implicam em operações que classes devem ter.

Uma mensagem representa a requisição de um objeto remetente a um objeto receptor para que este último execute alguma operação definida para sua classe. Essa mensagem deve conter informação suficiente para que a operação do objeto receptor possa ser executada.

### Mensagens *versus* responsabilidades

- Qual o objetivo da construção dos diagramas de interação?
  - Identificar **mensagens** e, em última análise, **responsabilidades (operações e atributos)**



Uma mensagem implica na existência de uma operação no objeto receptor. A resposta do objeto receptor ao recebimento de uma mensagem é a execução da operação correspondente.

### Sintaxe da UML para mensagens

- Na UML, o rótulo de uma mensagem deve seguir a seguinte sintaxe:

**[[expressão-sequência] controle:] [v :=] nome [(argumentos)]**

- Onde o termo **controle** pode ser uma condição ou um iteração:

<b>“*” ‘[’ cláusula iteração ‘]’</b>	<b>‘[’ cláusula condição ‘]’</b>
--------------------------------------	----------------------------------

- O único termo obrigatório corresponde ao **nome** da mensagem.

### Exemplos (sintaxe UML para mensagens)

- Mensagem simples, sem cláusula alguma.

1: adicionarItem(item)

- Mensagem com cláusula de condição.

3 [a > b]: trocar(a, b)

- Mensagem com cláusula de iteração e com limites indefinidos.

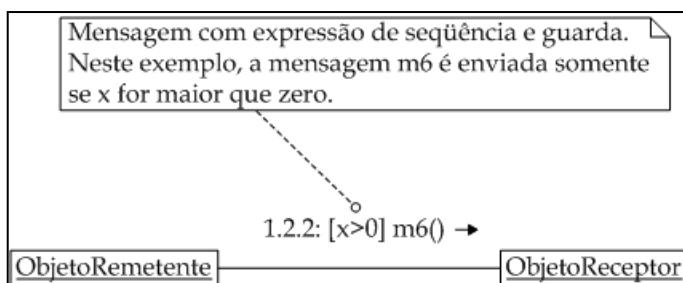
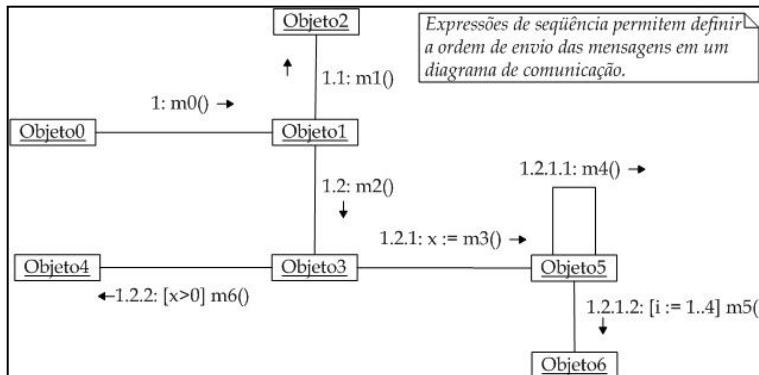
2 \*: desenhar( )

- Mensagem com cláusula de iteração e com limites definidos.

2 \*[i := 1..10]: figuras[i].desenhar( )

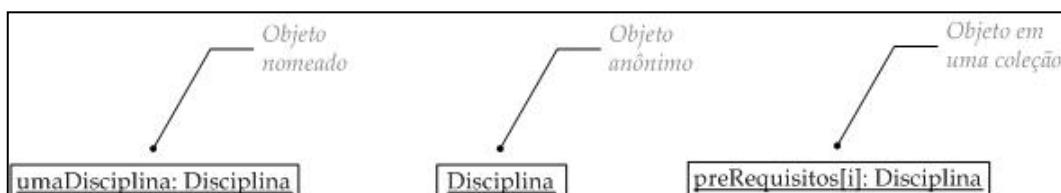
- Mensagem aninhada com retorno armazenado na variável x.

1.2.1: x := selecionar(e)



## Notação para objetos

- Objetos são representados em um diagrama de interação utilizando-se a mesma notação do diagrama de objetos.
- Pode-se representar objetos anônimos ou objetos nomeados, dependendo da situação.
- Elementos de uma coleção também podem ser representados.
- Classes também podem ser representadas.
  - Para o caso de mensagens enviadas para a classe.
  - Uma mensagem para uma classe dispara a execução de uma *operação estática*.
  - A representação de uma classe em um diagrama de seqüência é a mesma utilizada para objetos, porém o nome da classe não é sublinhado



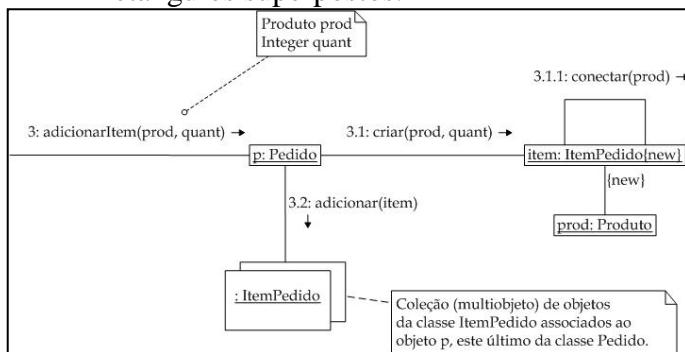
Multiobjetos

- Um **multiobjeto** é o nome que a UML dá para uma *coleção* de objetos de uma mesma classe. Pode ser utilizado para:

- representar o lado muitos de uma associação de conectividade um para muitos.
- representar uma lista (temporária ou não) de objetos sendo formada em uma colaboração.
- Um multiobjeto é representado na UML através de dois retângulos superpostos.
  - A superposição dos retângulos evita a confusão com a notação usada para objetos.
  - O nome do multiobjeto é apresentado no retângulo que fica por cima e segue a mesma nomenclatura utilizada para objetos.
  - Convenção: usar o nome da classe de seus elementos para nomear o multiobjeto.

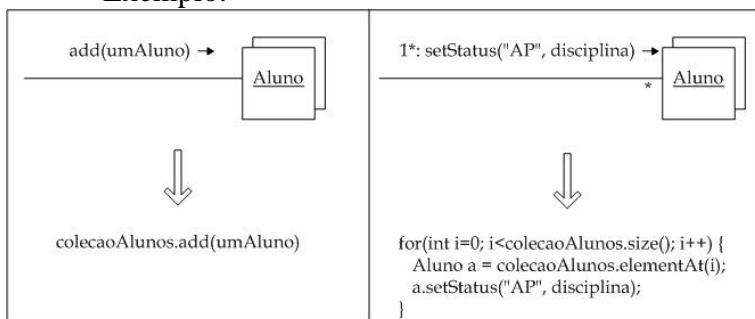
#### Notação para multiobjetos

- Uma multiobjeto é representado graficamente na UML através de dois retângulos superpostos.



#### Mensagens para Objetos/Coleção

- Uma mensagem pode ser enviada para um multiobjeto, ou pode ser enviada para um único objeto (elemento) do multiobjeto.
- Quando o símbolo de iteração não é usado, convenciona-se que a mensagem está sendo enviada para o próprio multiobjeto.
- Exemplo:



#### Implementação de multiobjetos

- Multiobjetos são normalmente implementados através de alguma estrutura de dados que manipule uma coleção de objetos.
- Portanto, algumas mensagens típicas que podemos esperar que um multiobjeto aceite são as seguintes:
  - Posicionar o cursor da coleção no primeiro elemento.
  - Retornar o i-ésimo objeto da coleção.
  - Retornar o próximo objeto da coleção.
  - Encontrar um objeto de acordo com um identificador único.
  - Adicionar um objeto na coleção.
  - Remover um objeto na coleção.

- Obter a quantidade de objetos na coleção.
- Retornar um valor lógico que indica se há mais objetos a serem considerados.
- A interface List da linguagem Java apresenta operações típicas de um multiobjeto.

```
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);
    boolean add(E element);
    void add(int index, E element);
    E remove(int index);
    abstract boolean addAll(int index, Collection<? extends E> c);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int from, int to);
}
```

#### Tipos de diagrama de interação

- Há três tipos de diagrama de interação na UML 2.0: **diagrama de seqüência**, **diagrama de comunicação** e **diagrama de visão geral da interação**.
  - O diagrama de seqüência e o diagrama de comunicação são equivalentes.

**Diagrama de seqüência:** foco nas mensagens enviadas no decorrer do tempo.

**Diagrama de comunicação:** foco nas mensagens enviadas entre objetos que estão relacionados.

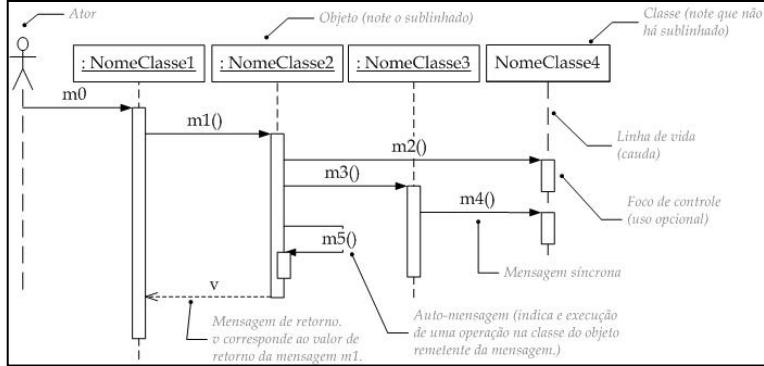
**Diagrama de visão geral de interação.** Pode ser utilizada para apresentar uma visão geral de diversas interações entre objetos, cada uma delas representada por um diagrama de interação. Diagrama é útil para **modularizar** a construção do diagramas de seqüência (ou de comunicação).

## 7.2 Diagrama de seqüência

- Os objetos participantes da interação são organizados na horizontal.
- Abaixo de cada objeto existe uma linha (linha de vida)
- Cada linha de vida possui o seu foco de controle.
  - Quando o objeto está fazendo algo.
- As mensagens entre objetos são representadas com linhas horizontais rotuladas partindo da linha de vida do objeto remetente e chegando a linha de vida do objeto receptor.
- A posição vertical das mensagens permite deduzir a ordem na qual elas são enviadas.
- Ordem de envio de mensagens em um diagrama de seqüência pode ser deduzida a partir das expressões de seqüência.
- Criação e destruição de objetos podem ser representadas.

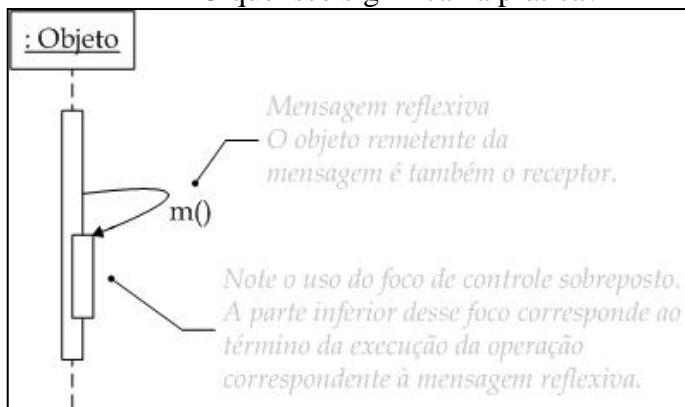
## Elementos gráficos de um DS

- Elementos básicos em um diagrama de seqüência:
  - Atores
  - Objetos, multiobjetos e classes
  - Mensagens
  - Linhas de vida e focos de controle
  - Criação e destruição de objetos
  - Iterações
  -

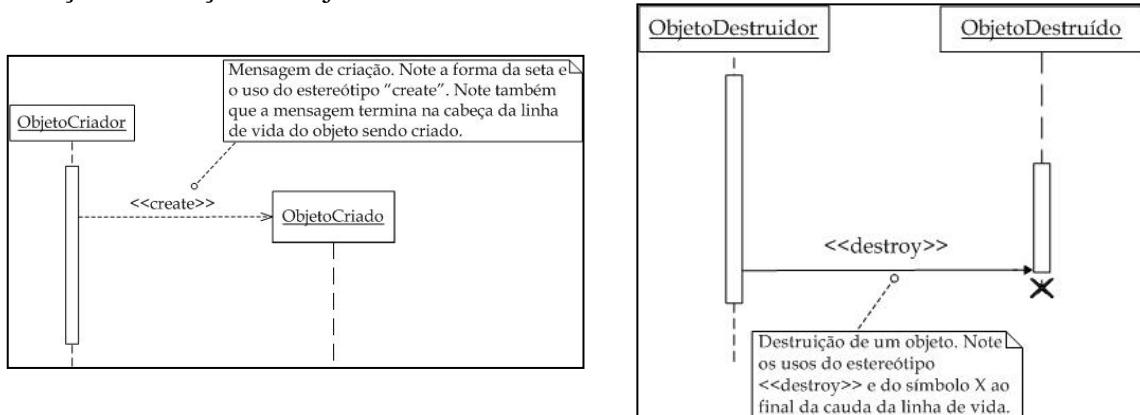


## Mensagens reflexivas em um DS

- Em uma mensagem reflexiva (ou auto-mensagem) o remetente é também o receptor.
  - Corresponde a uma mensagem para this (self).
  - O que isso significa na prática?



## Criação/destruição de objetos em um DS

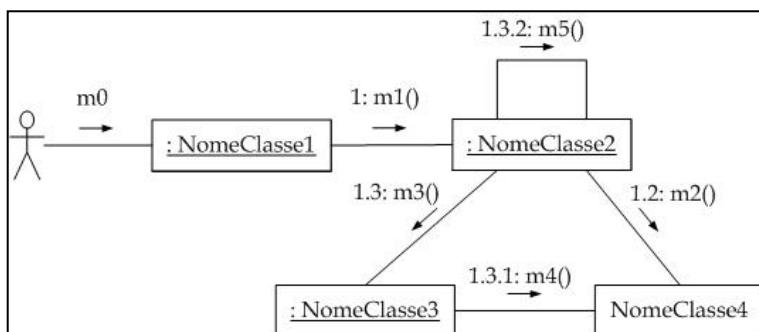


### 7.3 Diagrama de comunicação

- Chamado de **diagrama de colaboração** na UML 1.X.
- Estruturalmente, é bastante semelhante a um diagrama de objetos.
  - A diferença é que são adicionados setas e rótulos de mensagens nas ligações entre esses objetos.
- As ligações (linhas) entre objetos correspondem a relacionamentos existentes entre os objetos.
  - Deve haver consistência com o diagrama de classes...
- Os objetos estão distribuídos em duas dimensões
  - Vantagem: normalmente permite construir desenhos mais legíveis comparativamente aos diagramas de seqüência.
  - Desvantagem: não há como saber a ordem de envio das mensagens a não ser pelas expressões de seqüência.
- Direção de envio de mensagem é indicada por uma seta próxima ao rótulo da mensagem.

#### Elementos gráficos de um DC

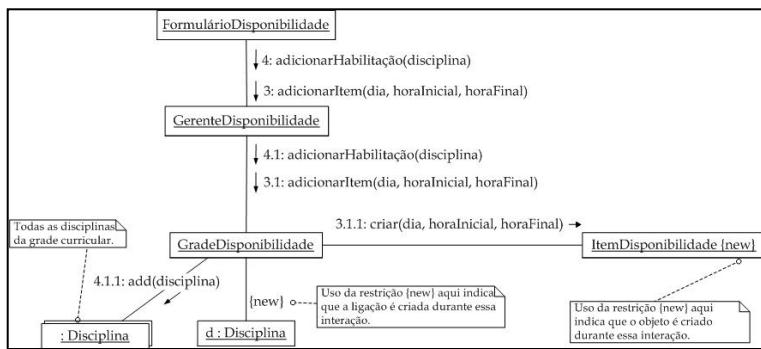
- Elementos básicos em um diagrama de comunicação:
  - Atores
  - Objetos, multiobjetos e classes
  - Mensagens
  - Ligações entre objetos
  - Criação e destruição de objetos
  - Iterações



#### Criação de objetos em um DC

- Durante a execução de um cenário de caso de uso, objetos podem ser criados e outros objetos podem ser destruídos.
- Alguns objetos podem sobreviver à execução do caso de uso (se conectando a outro objetos); outros podem nascer e morrer durante essa execução.
- A UML define etiquetas (tags) para criação e destruição de objetos (ou de ligações entre objetos) no diagrama de comunicação.
  - **{new}**: objetos ou ligações criados durante a interação.
  - **{destroyed}**: objetos ou ligações destruídos durante a interação.
  - **{transient}**: objetos ou ligações destruídos e criados durante a interação.

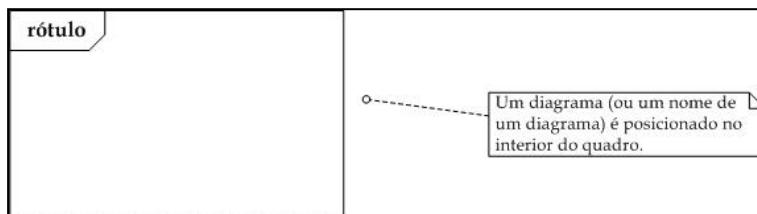




## 7.4 Modularização de interações

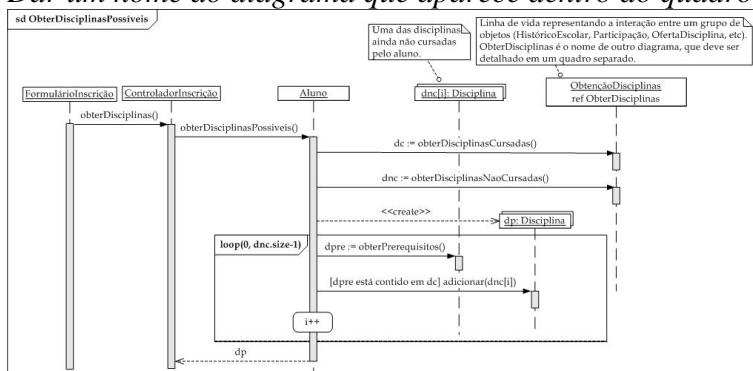
Quadros de interação

- Elemento gráfico, que serve para modularizar a construção de diagramas de seqüência (ou de comunicação).
- Objetivos específicos:
  - Dar um nome ao diagrama que aparece dentro do quadro;
  - Fazer referência a um diagrama definido separadamente;
  - Definir o fluxo de controle da interação.
- Notação:



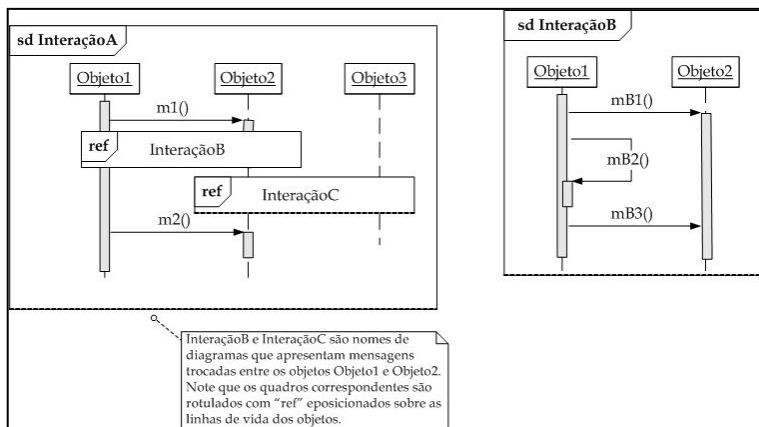
Diagramas nomeados

*Dar um nome ao diagrama que aparece dentro do quadro*

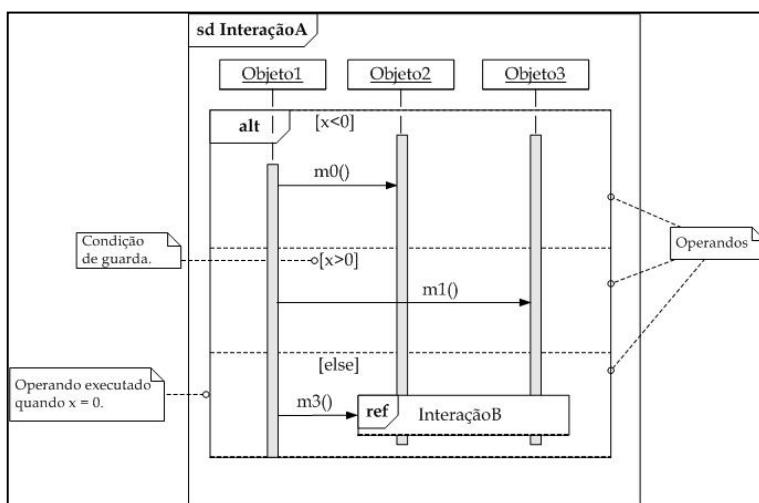


Diagramas referenciados

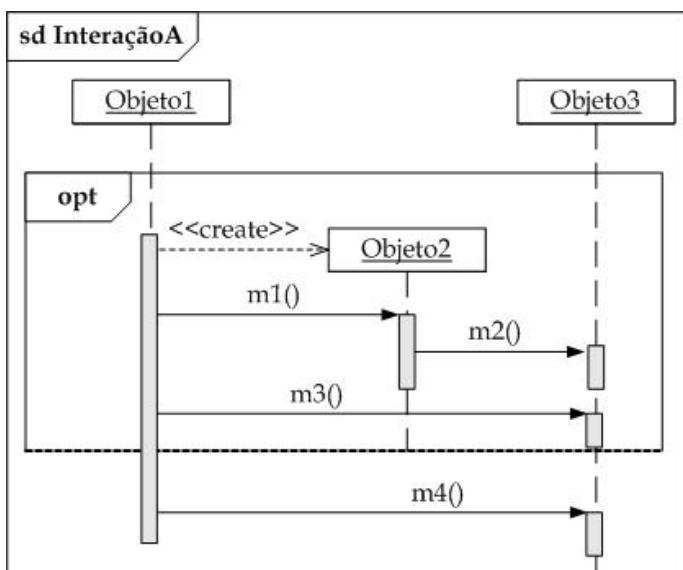
*Fazer referência a um diagrama definido separadamente.*



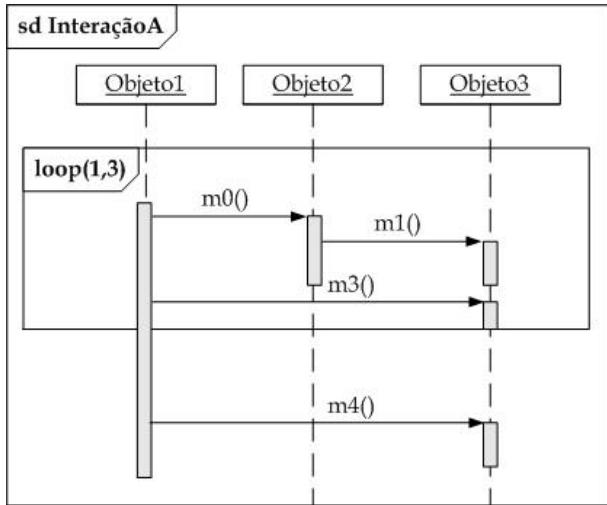
### Fluxo de controle: alternativas



### Fluxo de controle: opções



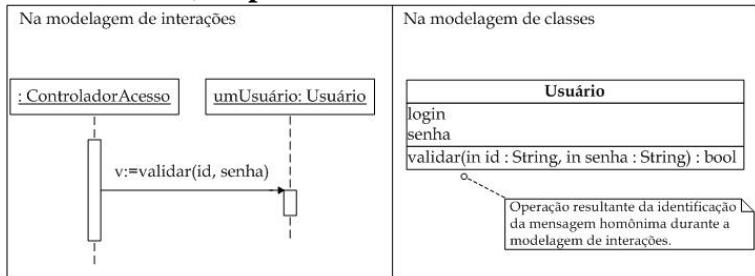
## Fluxo de controle: iterações



## 7.5 Construção do modelo de interações

Mensagens *versus* responsabilidades

- O objetivo da modelagem de interações é identificar **mensagens** e, em última análise, **responsabilidades**.



Uma mensagem implica na existência de uma operação no objeto receptor. A resposta do objeto receptor ao recebimento de uma mensagem é a execução da operação correspondente.

Alocação de responsabilidades

- Podemos então entender a modelagem de interações como um processo cujo objetivo final é decompor as responsabilidades do sistema e alocá-las a classes.
- Dado um conjunto de N responsabilidades, uma possibilidade é criar uma única classe no sistema para assumir com todas as N responsabilidades.
- Outra possibilidade é criar N classes no sistema, a cada um delas sendo atribuída uma das N responsabilidades.
- Certamente, as duas alternativas anteriores são absurdas do ponto de vista prático. Mas, entre as muitas maneiras possíveis de alocar responsabilidades, como podemos saber quais delas são melhores que outras?

Acoplamento e coesão

- A resposta à pergunta anterior não é nenhuma receita de bolo.
- De fato, para construirmos um bom modelo de interações, devemos lançar mão de diversos princípios de projeto:
- Dois dos principais princípios são o acoplamento e a coesão.

- A **coesão** é uma medida do quanto fortemente relacionadas e focalizadas são as responsabilidades de uma classe.
- É extremamente importante assegurar que as responsabilidades atribuídas a cada classe sejam altamente relacionadas.
  - Em outras palavras, o projetista deve definir classes de tal forma que cada uma delas tenha alta coesão.
- O **acoplamento** é uma medida de quanto fortemente uma classe está conectada a outras classes, tem conhecimento ou depende das mesmas.
- Uma classe com **acoplamento fraco** (baixo) não depende de muitas outras.
  - Por outro lado, uma classe com **acoplamento forte** é menos inteligível isoladamente e menos reutilizável.
- Além disso, uma classe com alto acoplamento é mais sensível a mudanças, quando é necessário modificar as classes da qual ela depende.
- Conclusão: criar modelos com **alta coesão** e **baixo acoplamento** deve ser um objetivo de qualquer projetista.

Dicas para construção do MI

- **Identifique as classes conceituais que participam em cada caso de uso.**
  - Estas são as entidades do mundo real que estariam envolvidas na tarefa do caso de uso se este fosse executada manualmente.
    - Exemplos são: Aluno, OfertaDisciplina, Venda, Pagamento, etc.
  - Note que classes de fronteira também podem ser classes conceituais.
    - Por exemplo, FormulárioInscrição é um objeto de fronteira (para o caso de uso Realizar Inscrição) que também corresponde a um conceito existente no domínio do problema.
- **Identifique quaisquer classes de software que ajudem a organizar as tarefas a serem executadas.**
  - Essas classes normalmente são necessárias para manter a **coesão** das demais classes em um nível alto.
  - Segundo Craig Larman, essas classes são **fabricações puras** (*pure fabrications*).
  - Aqui, se encaixam algumas classes de fronteira, classes de controle.
  - Também: classes de acesso ao mecanismo de armazenamento, classes de autenticação, etc.
- **Defina também que objetos criam (destróem) outros objetos.**
  - Na realização de um caso de uso, objetos de entidade podem ser criados pelo objeto de controle, que recebe os dados necessários à instanciação a partir de objetos de fronteira.
  - Objetos de entidade também podem ser criados (destruídos) por outros objetos de entidade.
  - De uma forma geral, em uma agregação (ou composição), o objeto todo tem prioridade para criar (destruir) suas partes.
  - Portanto, em uma agregação (ou composição) entre objetos de entidade, é mais adequado que o objeto todo crie (destrua) suas partes quando requisitado por outros objetos.
- **Verifique a consistência dos diagramas de interação em relação ao MCU e ao modelo de classes.**
  - Verifique que cada cenário relevante para cada caso de uso foi considerado na modelagem de interações.

- Se assegure de que as mensagens que um objeto recebe estão consistentes com as responsabilidades a ele atribuídas.
  - Alguns dos objetos necessários em uma interação já podem ter sido identificados durante a construção do modelo de classes de análise.
  - Durante a construção do diagrama de interação, o modelador pode identificar novas classes.
  - Atributos, associações e operações também surgem como subproduto da construção dos diagramas de interação.
- ***Se certifique de que o objeto de controle realiza apenas a coordenação da realização do caso de uso.***
  - Como o controlador tem a responsabilidade de coordenação, todas as ações do ator resultam em alguma atividade realizada por esse objeto de controle.
  - Isso pode levar ao alto acoplamento; no pior caso, o controlador tem conhecimento de todas as classes participantes do caso de uso.
  - Responsabilidades específicas no domínio devem ser atribuídas aos objetos de domínio (entidades).
  - Sempre que for adequado, segundo os princípios de coesão e de acoplamento, devemos fazer com que as classes de domínio enviem mensagens entre si, aliviando o objeto de controle.
- ***Faça o máximo para construir diagramas de interação o mais inteligíveis possível.***
  - Por exemplo, podemos utilizar notas explicativas para esclarecer algumas partes do diagrama de interação que esteja construindo.
    - Essas notas podem conter pseudocódigo ou mesmo texto livre.
  - Outra estratégia que ajuda a construir um modelo de interações mais inteligível é utilizar os recursos de modularização que a UML 2.0 acrescentou.
    - quadros de interação, referências entre diagramas, etc.
- ***Utilize o princípio de projeto conhecido como Lei de Demeter.***
  - Esse princípio está associado ao princípio do acoplamento e impõe restrições acerca de quais são os objetos para os quais devem ser enviadas mensagens na implementação de uma operação:
    - (a) ao próprio objeto da classe (ou self);
    - (b) a um objeto recebido como parâmetro do método;
    - (c) a um atributo da classe;
    - (d) a um objeto criado dentro do método;
    - (e) a um elemento de uma coleção que é atributo da classe.
  - A intenção é evitar acoplar excessivamente um objeto e também evitar que ele tenha conhecimento das associações entre outros objetos.
- Na modelagem de interações, quando definimos uma mensagem, estamos criando uma dependência entre os objetos envolvidos.
- Isso é mesmo que dizermos que estamos aumentando o acoplamento entre os objetos em questão.
- Portanto, é necessário que o modelador fique atento para apenas definir mensagens que são realmente necessárias.
  - Sempre que possível, devemos evitar o envio de mensagens que implique na criação de associações redundantes no modelo de classes.
  - Isso porque a adição de uma associação entre duas classes aumenta o acoplamento entre as mesmas.

## Procedimento de construção

- Vamos agora descrever um procedimento para construção do modelo de interações.
  - Note, primeiramente,
- Esse procedimento genérico serve tanto para diagramas de seqüência quanto para diagramas de comunicação, resguardando-se as diferenças de notação entre os dois.
- Durante a aplicação desse procedimento, é recomendável considerar todas as dicas descritas anteriormente.
- Antes de descrevermos esse procedimento, é necessário que definamos o conceito de **evento de sistema**...

## Eventos de sistema

- Eventos de sistema correspondem às ações do ator no cenário de determinado caso de uso.
- Sendo assim, é relativamente fácil identificar eventos de sistemas em uma descrição de caso de uso: devemos procurar nessa descrição os eventos que correspondem a **ações do ator**.
- *No caso particular em que o ator é um ser humano e existe uma interface gráfica para que o mesmo interaja com o sistema, os eventos do sistema são resultantes de ações desse ator sobre essa interface gráfica, que corresponde a objetos de fronteira.*
- Considere o formulário a seguir, para o caso de uso (do SCA) denominado "Fornecer Grade de Disponibilidades":

Dia Semana	Hora inicial	Hora final
Quarta-feira	08:30	11:30
Segunda-feira	07:00	10:40
Quarta-feira	08:30	11:30
Sexta-feira	14:30	18:00
Sexta-feira	07:00	10:40

- No formulário anterior, temos a seguinte lista de eventos de sistema:
  - Solicitação de validação de matrícula de professor;
  - Solicitação de adição de uma disciplina à grade;
  - Solicitação de adição de um item de disponibilidade (dia, hora final e hora final) à grade;
  - Solicitação de registro da grade.
- Importante: nem todo evento de sistema é originado em um objeto de fronteira correspondente a uma interface gráfica.
  - essa ocorrência pode ser gerada por um ator que não seja um ser humano (e.g., outro sistema ou um equipamento).
- Mas, por que os eventos de sistema são importantes para a modelagem de interações?
- Porque as interações entre objetos de um sistema acontecem por conta do acontecimento daqueles.

- Um evento de sistema é alguma ação tomada por um ator que resulta em uma sequencia de *mensagens* trocadas entre os objetos do sistema.
- Portanto, o ponto de partida para a modelagem de interações é a identificação dos eventos do sistema.
- Uma vez feita essa identificação, podemos desenhar diagramas de interação que modelam como os objetos colaboram entre si para produzir a resposta desejada a cada evento do sistema.

#### Procedimento de construção

- Para cada caso de uso, selecione um conjunto de cenários relevantes.
  - O cenário correspondente ao fluxo principal do caso de uso deve ser incluído.
  - Considere também fluxos alternativos e de exceção que tenham potencial em demandar responsabilidades de uma ou mais classes.
- Para cada cenário selecionado, identifique os eventos de sistema:
  - Posicione o(s) ator(es), objeto de fronteira e objeto de controle no diagrama.
  - Para cada passo do cenário selecionado, defina as mensagens a serem enviadas de um objeto a outro.
  - Defina as cláusulas de condição e de iteração, se existirem, para as mensagens.
  - Adicione multiobjetos e objetos de entidade à medida que a sua participação se faça necessária no cenário selecionado.

#### Observações sobre o procedimento

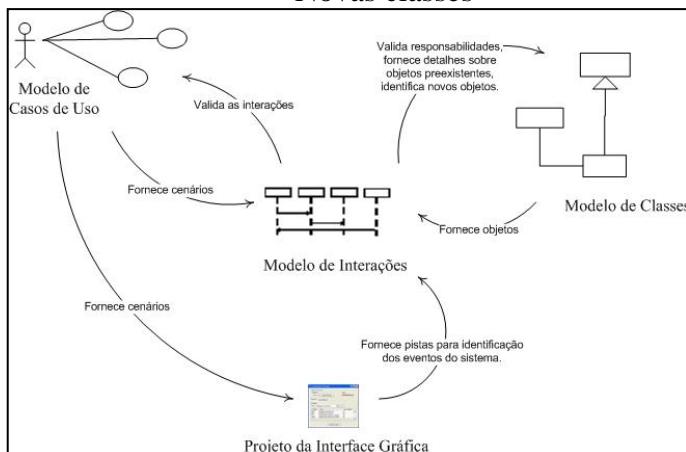
- A definição das mensagens deve ser feita com base nas responsabilidades de cada objeto envolvido:
  - O nome da mensagem
  - Os argumentos de cada mensagem, se existirem.
  - O valor de retorno da operação correspondente, se existir.
  - Cláusulas de condição e de repetição, se existirem.
- A maioria dos objetos já devem ter sido identificados durante a construção do modelo de classes.
- Verificar as consistências:
  - Cada cenário relevante para cada caso de uso foi considerado?
  - As mensagens que um objeto recebe estão consistentes com suas responsabilidades?
- As mensagens de um ator a um objeto de fronteira normalmente são rotuladas com a informação fornecida
  - Por exemplo, *item de pedido, id e senha*, etc.
- Mais de um controlador podem ser criados em um mesmo caso de uso, dependendo de sua complexidade.
  - O controlador pode mesmo ser suprimido, também em função da complexidade do caso de uso.
- Mensagens enviadas pelo objeto de fronteira por conta de um evento de sistema resultam na necessidade de definir ***operações de sistema*** no objeto controlador do caso de uso.
  - Por exemplo, no formulário de fornecimento de disponibilidades, o controlador deve possuir as seguintes operações de sistema:
    - validarProfessor(matrícula);

- adicionarDisciplina(nomeDisciplina);
- adicionarItemDisponibilidade(dia, horaInicial, horaFinal).
- registrarGrade()

## 7.6 Modelo de interações em um processo iterativo

MI em um processo iterativo

- São utilizados na fase de construção de um ciclo de vida incremental e iterativo.
  - São construídos para os casos de uso alocados para uma iteração desta fase.
- Há controvérsias sobre o momento de início da utilização desse modelo (se na análise ou se no projeto).
- Inicialmente (+análise), pode exibir apenas os objetos participantes e mensagens exibindo somente o nome da operação (ou nome da responsabilidade).
- Posteriormente (+projeto), pode ser refinado.
  - Criação e destruição de objetos, tipo e assinatura completa de cada mensagem, etc.
- Embora modelos de um SSOO representem visões distintas, eles são *interdependentes e complementares*.
  - O MCU fornece cenários a serem considerados pelo MI.
  - O modelo de classes de análise fornece objetos iniciais para o MI.
  - A construção do MI fornece informações úteis para transformar o modelo de classes de análise no modelo de classes de especificação. Em particular, MI fornece os seguintes itens para refinar o modelo de classes de análise:
    - Detalhamento de operações
    - Detalhamento de associações
    - Operações para classes
    - Novos atributos para classes
    - Novas classes



### Discussão

- Como informações são passadas de um objeto a outro em um sistema OO?
- Quando utilizar diagramas de interações (seqüência ou comunicação)?
  - Há alternativas para esse momento?
- Qual é a consequência da construção dos DI's sobre os demais artefatos do sistema.

- Há possibilidade de geração de código a partir de um diagrama de interações?

## 8 Modelagem de classes de projeto

*“A perfeição (no projeto) é alcançada, não quando não há nada mais para adicionar, mas quando não há nada mais para retirar.”*

*-Eric Raymond, The Cathedral and the Bazaar*

Tópicos

- Introdução
- Transformação de classes de análise em classes de projeto
- Especificação de atributos
- Especificação de operações
- Especificação de associações
- Herança
- Padrões de projeto

Introdução

- O **modelo de classes de projeto** é resultante de refinamentos no modelo de classes de análise.
- Esse modelo é construído em paralelo com o **modelo de interações**.
  - A construção do MI gera informações para a transformação do modelo de classes de análise no modelos de classes de projeto.
- O modelo de classes de projeto contém detalhes úteis para a **implementação** das classes nele contidas.
- Aspectos a serem considerados na fase de projeto para a modelagem de classes:
  - Estudo de novos elementos do diagrama de classes que são necessários à construção do modelo de projeto.
  - Descrever transformações pelas quais passam as classes e suas propriedades com o objetivo de transformar o modelo de classes análise no modelo de classes de projeto.
  - Adição de novas classes ao modelo
  - Especificação de atributos, operações e de associações
  - Descrever refinamentos e conceitos relacionados à herança, que surgem durante a modelagem de classes de projeto
    - Classes abstratas, interfaces, polimorfismo e padrões de projeto.
  - Utilização de padrões de projeto (*design patterns*)

### 8.1 Transformação de classes de análise em classes de projeto

Especificação de classes de fronteira

- Não devemos atribuir a essas classes responsabilidades relativas à lógica do negócio.
  - Classes de fronteira devem apenas servir como um ponto de captação de informações, ou de apresentação de informações que o sistema processou.
  - A única inteligência que essas classes devem ter é a que permite a elas realizarem a comunicação com o ambiente do sistema.
- Há diversas razões para isso:

- Em primeiro lugar, se o sistema tiver que ser implantado em outro ambiente, as modificações resultantes sobre seu funcionamento propriamente dito seriam mínimas.
  - Além disso, o sistema pode dar suporte a diversas formas de interação com seu ambiente (e.g., uma interface gráfica e uma interface de texto).
  - Finalmente, essa separação resulta em uma melhor ***coesão***.
- Durante a análise, considera-se que há uma única classe de fronteira para cada ator. No projeto, algumas dessas classes podem resultar em várias outras.
- Interface com seres humanos: ***projeto da interface gráfica*** produz o detalhamento das classes.
- Outros sistemas ou equipamentos: devemos definir uma ou mais classes para encapsular o protocolo de comunicação.
  - É usual a definição de um **subsistema** para representar a comunicação com outros sistemas de software ou com equipamentos.
  - É comum nesse caso o uso do padrão Façade (mais adiante)
- O projeto de objetos de fronteira é altamente dependente da natureza do ambiente...
- Clientes WEB clássicos
  - Classes de fronteira são representadas por páginas HTML que, muitas vezes, representam sites dinâmicos.
- Clientes móveis
  - Classes de fronteira implementam algum protocolo específico com o ambiente.
    - Um exemplo é a WML (Wireless Markup Language).
- Clientes stand-alone
  - Nesse caso, é recomendável que os desenvolvedores pesquisem os recursos fornecidos pelo ambiente de programação sendo utilizado.
    - Um exemplo disso é o Swing/JFC da linguagem Java.
- Serviços WEB (*WEB services*)
  - Um serviço WEB é uma forma de permitir que uma aplicação forneça seus serviços (funcionalidades) através da Internet.

#### Especificação de classes de entidade

- A maioria das classes de entidade normalmente permanece na passagem da análise ao projeto.
  - Na verdade, classes de entidade são normalmente as primeiras classes a serem identificadas, na análise de domínio.
- Durante o projeto, um aspecto importante a considerar sobre classes de entidade é identificar quais delas geram objetos que devem ser persistentes.
  - Para essas classes, o seu mapeamento para algum mecanismo de armazenamento persistente deve ser definido (Capítulo 12).
- Um aspecto importante é a forma de representar associações, agregações e composições entre objetos de entidade.
  - Essa representação é função da navegabilidade e da multiplicidade definidas para a associação, conforme visto mais adiante.
- Outro aspecto relevante para classes de entidade é o modo como podemos identificar cada um de seus objetos unicamente.
  - Isso porque, principalmente em sistemas de informação, objetos de entidade devem ser armazenados de modo persistentes.

- Por exemplo, um objeto da classe Aluno é unicamente identificado pelo valor de sua matrícula (um atributo do domínio).
- A manipulação dos diversos atributos identificadores possíveis em uma classes pode ser bastante trabalhosa.
- Para evitar isso, um *identificador de implementação* é criado, que não tem correspondente com atributo algum do domínio.
  - Possibilidade de manipular identificadores de maneira uniforme e eficiente.
  - Maior facilidade quando objetos devem ser mapeados para um SGBDR

#### Especificação de classes de controle

- Com relação às classes de controle, no projeto devemos identificar a real utilidade das mesmas.
  - Em casos de uso simples (e.g., manutenção de dados), classes de controle não são realmente necessárias. Neste caso, classes de fronteira podem repassar os dados fornecidos pelos atores diretamente para as classes de entidade correspondentes.
- Entretanto, é comum a situação em que uma classe de controle de análise ser transformada em duas ou mais classes no nível de especificação.
- No refinamento de qualquer classe proveniente da análise, é possível a aplicação de padrões de projeto (*design patterns*)
- Normalmente, cada classe de controle deve ser particionada em duas ou mais outras classes para controlar diversos aspectos da solução.
  - Objetivo: de evitar a criação de uma única classe com baixa coesão e alto acoplamento.
- Alguns exemplos dos aspectos de uma aplicação cuja coordenação é de responsabilidade das classes de controle:
  - Produção de valores para preenchimento de controles da interface gráfica,
  - Autenticação de usuários,
  - Controle de acesso a funcionalidades do sistema, etc.
- Um tipo comum de controlador é o *controlador de caso de uso*, responsável pela coordenação da realização de um caso de uso.
- As seguintes responsabilidades são esperadas de um controlador de caso de uso:
  - Coordenar a realização de um caso de uso do sistema.
  - Servir como canal de comunicação entre objetos de fronteira e objetos de entidade.
  - Se comunicar com outros controladores, quando necessário.
  - Mapear ações do usuário (ou atores de uma forma geral) para atualizações ou mensagens a serem enviadas a objetos de entidade.
  - Estar apto a manipular exceções provenientes das classes de entidades.
- Em aplicações WEB, é comum a prática de utilizar outro tipo de objeto controlador chamado *front controller* (FC).
- Um FC é um controlador responsável por receber todas as requisições de um cliente.
- O FC identifica qual o controlador (de caso de uso) adequado para processar a requisição, e a despacha para ele.
- Sendo assim, um FC é um ponto central de entrada para as funcionalidades do sistema.
  - Vantagem: mais fácil controlar a autenticação dos usuários.
- O FC é um dos *padrões de projeto* do catálogo J2EE

- <http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html>

### Especificação de outras classes

- Além do refinamento de classes preexistentes, diversas outros aspectos demanda a identificação de novas classe durante o projeto.
  - Persistência de objetos
  - Distribuição e comunicação (e.g., RMI, CORBA, DCOM, WEB)
  - Autenticação/Autorização
  - Logging
  - Configurações
  - Threads
  - Classes para testes (*Test Driven Development*)
  - Uso de **bibliotecas, componentes e frameworks**
- Conclusão: a tarefa de identificação (reuso?) de classes não termina na análise.

### 8.2 Especificações de atributos

#### 8.3 Especificações de operações

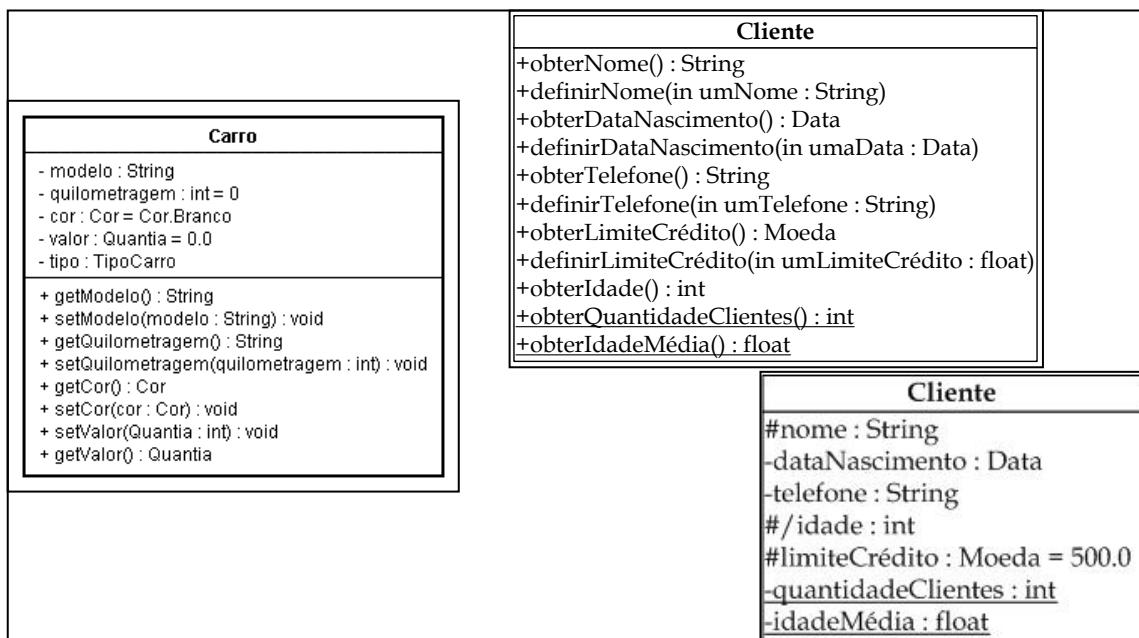
##### Refinamento de Atributos e Métodos

- Os atributos e métodos de uma classe a habilitam a cumprir com suas **responsabilidades**.
- **Atributos:** permitem que uma classe armazene informações necessárias à realização de suas tarefas.
- **Métodos:** são funções que manipulam os valores do atributos, com o objetivo de atender às **mensagens** que o objeto recebe.
- A especificação completa para atributos/operações deve seguir as sintaxes definidas pela UML:

<code>[/] [visibilidade] nome [multiplicidade] [: tipo] [= valor-inicial]</code>
--

<code>[visibilidade] nome [(parâmetros)] [: tipo-retorno] [{propriedades}]</code>
---

## Sintaxe para atributos e operações



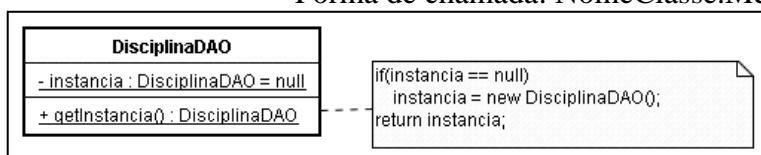
- Obs.: classes utilitárias podem ser utilizadas como tipos para atributos. (e.g., Moeda, Quantia, TipoCarro, Endereco)

### Visibilidade e Encapsulamento

- Os três qualificadores de visibilidade aplicáveis a atributos também podem ser aplicados a operações.
- + representa visibilidade pública
- # representa visibilidade protegida
- representa visibilidade privativa
  - O real significado desses qualificadores depende da linguagem de programação em questão.
  - Usualmente, o conjunto das operações públicas de uma classe é chamado de *interface* dessa classe.
    - Note que há diversos significados para o termo interface.

### Membros estáticos

- Membros estáticos são representados no diagrama de classes por declarações sublinhadas.
  - Atributos estáticos** (variáveis de classe) são aqueles cujos valores valem para a classe de objetos como um todo.
    - Diferentemente de atributos não-estáticos (ou variáveis de instância), cujos valores são particulares a cada objeto.
  - Métodos estáticos** são os que não precisam da existência de uma instância da classe a qual pertencem para serem executados.
    - Forma de chamada: NomeClasse.Método(argumentos)



## Projeto de métodos

- Métodos de construção (criação) e destruição de objetos
- Métodos de acesso (getX/setX) ou propriedades
- Métodos para manutenção de associações (conexões) entre objetos.
- Outros métodos:
  - Valores derivados, formatação, conversão, cópia e clonagem de objetos, etc.
- Alguns métodos devem ter uma operação inversa óbvia
  - e.g., habilitar e desabilitar; tornarVisível e tornarInvisível; adicionar e remover; depositar e sacar, etc.
- Operações para desfazer ações anteriores.
  - e.g., padrões de projeto GoF: Memento e Command

## Operações para manutenção de associações (exemplo)

```
public class Turma {  
    private Set<OfertaDisciplina> ofertasDisciplina = new  
    HashSet();  
  
    public Turma() {  
    }  
  
    public void adicionarOferta(OfertaDisciplina oferta) {  
        this.ofertasDisciplina.add(oferta);  
    }  
  
    public boolean removerOferta(OfertaDisciplina oferta) {  
        return this.ofertasDisciplina.remove(oferta);  
    }  
  
    public Set getOfertasDisciplina() {  
        return  
    Collections.unmodifiableSet(this.ofertasDisciplina);  
    }  
}
```

## Detalhamento de métodos

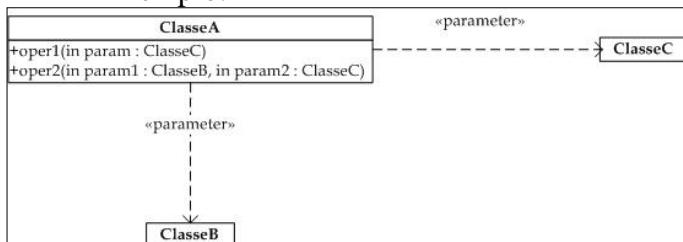
- Diagramas de interação fornecem um indicativo sobre como métodos devem ser implementados.
- Como complemento, notas explicativas também são úteis no esclarecimento de como um método deve ser implementado.
- O diagrama de atividades também pode ser usado para detalhar a lógica de funcionamento de métodos mais complexos.

## 8.4 Especificação de associações

### O conceito de dependência

- O *relacionamento de dependência* indica que uma classe depende dos serviços (operações) fornecidos por outra classe.
- Na análise, utilizamos apenas a *dependência por atributo* (ou estrutural), na qual a classe dependente possui um atributo que é uma referência para a outra classe.
- Entretanto, há também as *dependências não estruturais*:

- Na **dependência por variável global**, um objeto de escopo global é referenciado em algum método da classe dependente.
- Na **dependência por variável local**, um objeto recebe outro como retorno de um método, ou possui uma referência para o outro objeto como uma variável local em algum método.
- Na **dependência por parâmetro**, um objeto recebe outro como parâmetro em um método.
- Dependências não estruturais são representadas na UML por uma linha tracejada direcionada e ligando as classes envolvidas.
  - A direção é da classe dependente (*cliente*) para a classe da qual ela depende (*fornecedor*).
  - Estereótipos predefinidos: <<global>>, << local>>, << parameter>>.
- Exemplo:

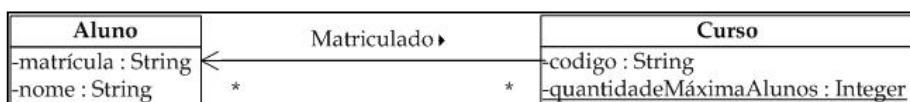


#### De associações para dependências

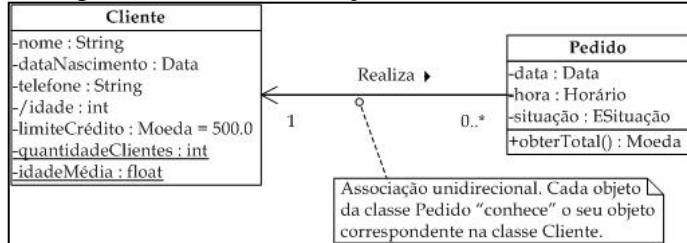
- Durante o projeto de classes, é necessário avaliar, para cada associação existente, se é possível transformá-la em uma dependência não estrutural.
- Objetivo: aumentar o encapsulamento de cada classe e diminuir o acoplamento entre as classes.
  - A dependência por atributo é a forma mais forte de dependência.
  - Quanto menos dependências por atributo houver no modelo de classes, maior é o encapsulamento e menor o acoplamento.

#### Navegabilidade de associações

- Associações podem ser **bidirecionais** ou **unidirecionais**.
  - Uma **associação bidirecional** indica que há um conhecimento mútuo entre os objetos associados.
  - Uma **associação unidirecional** indica que apenas um dos extremos da associação tem ciência da existência da mesma.
    - Representada através da adição de um sentido à seta da associação.
- A escolha da navegabilidade de uma associação pode ser feita através do estudo dos **diagramas de interação**.
  - O sentido de envio das mensagens entre objetos influencia na necessidade ou não de naveabilidade em cada um dos sentidos.



## Navegabilidade de associações



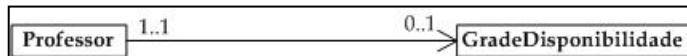
## Implementação de associações

- Há três casos, em função da conectividade: 1:1, 1:N e N:M
- Para uma associação 1:1 entre duas classes A e B:
  - Se a navegabilidade é unidirecional no sentido de A para B, é definido um atributo do tipo B na classe A.
  - Se a navegabilidade é bidirecional, podemos aplicar o procedimento acima para as duas classes.
- Para uma associação 1:N ou N:M entre duas classes A e B:
  - São utilizados atributos cujos tipos representam coleções de elementos.
  - É também comum o uso de classes parametrizadas.
    - Idéia básica: definir uma classe parametrizada cujo parâmetro é a classe correspondente ao lado *muitos* da associação.
    - O caso N:M é bastante semelhante ao refinamento das associações um para muitos.

## Classe Parametrizada

- Uma coleção pode ser representada em um diagrama de classes através uma **classe parametrizada**.
  - Def.: é uma classe utilizada para definir outras classes.
  - Possui operações ou atributos cuja definição é feita em função de um ou mais parâmetros.
- Uma coleção pode ser definida a partir de uma classe parametrizada, onde o parâmetro é o tipo do elemento da coleção.
  - Qual é a relação desse conceito com os **multiobjetos**?

## Conectividade 1:1

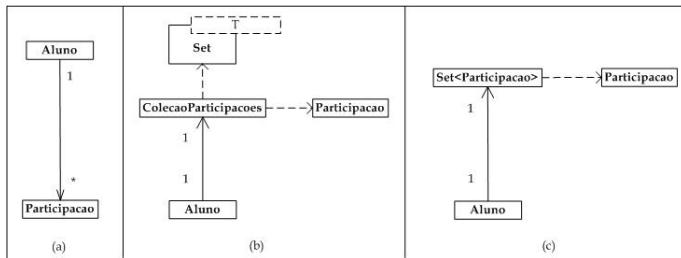


```

public class Professor {
    private GradeDisciplinas grade;
    ...
}
  
```

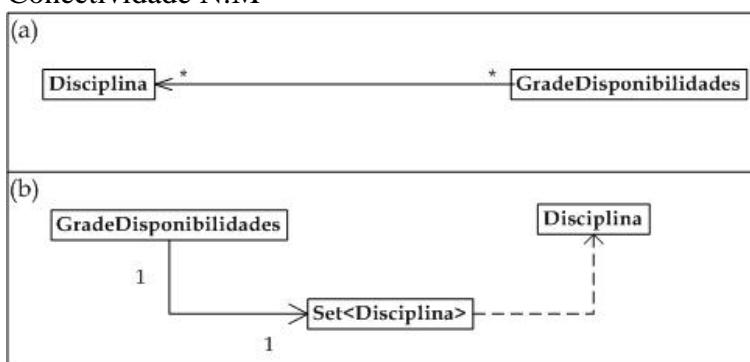
## Conectividade 1:N

- Formas alternativas para representação de uma associação cuja conectividade é 1:N.

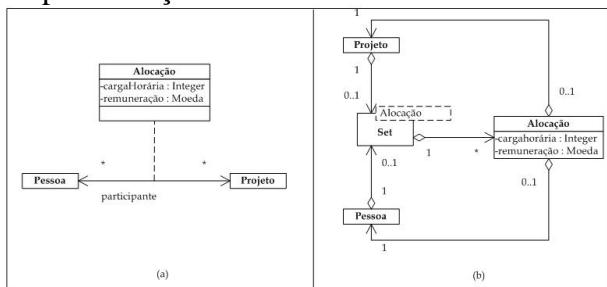


```
public class Aluno {
    private Set<Participacao> participacoes;
    ...
    public boolean adicionarParticipacao(Participacao p) {
        ...
    }
    public boolean removerParticipacao(Participacao p) {
        ...
    }
}
```

## Conectividade N:M



## Implementação de classes associativas



## 8.5 Herança

### Relacionamento de Herança

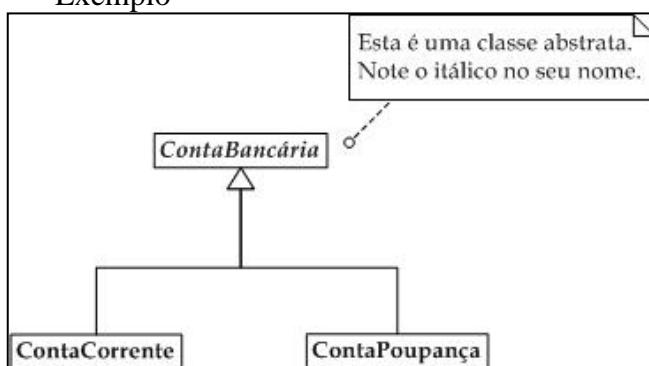
- Na modelagem de classes de projeto, há diversos aspectos relacionados ao de **relacionamento de herança**.
  - Tipos de herança
  - Classes abstratas
  - Operações abstratas
  - Operações polimórficas
  - Interfaces
  - Acoplamento concreto e abstrato
  - Reuso através de delegação e através de generalização
  - Classificação dinâmica

### Tipos de herança

- Com relação à quantidade de superclasses que certa classe pode ter.
  - **herança múltipla**
  - **herança simples**
- Com relação à forma de reutilização envolvida.
  - Na **herança de implementação**, uma classe reusa alguma implementação de um “ancestral”.
  - Na **herança de interface**, uma classe reusa a interface (conjunto das assinaturas de operações) de um “ancestral” e se compromete a implementar essa interface.

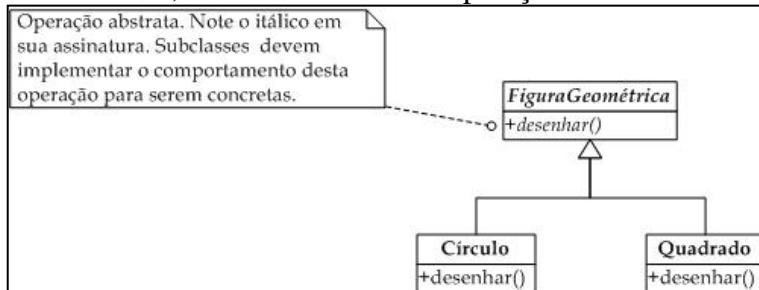
### Classes abstratas

- Usualmente, a existência de uma classe se justifica pelo fato de haver a possibilidade de gerar instâncias a partir da mesma.
  - Essas classes são chamadas de **classes concretas**.
- No entanto, podem existir classes que não geram instâncias “diretamente”.
  - Essas classes são chamadas de **classes abstratas**.
- Classes abstratas são usadas para organizar hierarquias gen/spec.
  - Propriedades comuns a diversas classes podem ser organizadas e definidas em uma classe abstrata a partir da qual as primeiras herdaram.
- Também propiciam a implementação do **princípio do polimorfismo**.
- Na UML, uma classe abstrata pode ser representada de duas maneiras alternativas:
  - Com o seu nome em *italico*.
  - Qualificando-a com a propriedade *{abstract}*
- Exemplo



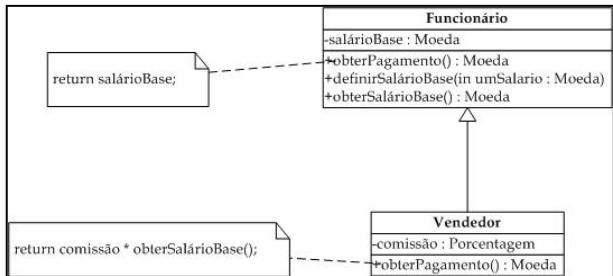
## Operações abstratas

- Uma classe abstrata possui ao menos uma **operação abstrata**, que corresponde à especificação de um serviço que a classe deve fornecer (sem método).
  - Uma classe qualquer pode possuir tanto operações abstratas, quanto operações concretas (ou seja, operações que possuem implementação).
  - Entretanto, uma classe que possui pelo menos uma operação abstrata é, por definição abstrata, abstrata.
- Uma operação abstrata definida com visibilidade pública em uma classe também é herdada por suas subclasses.
- Quando uma subclasse herda uma operação abstrata e não fornece uma implementação para a mesma, esta classe também é abstrata.
- Na UML, a assinatura de uma operação abstrata é definida em itálico.

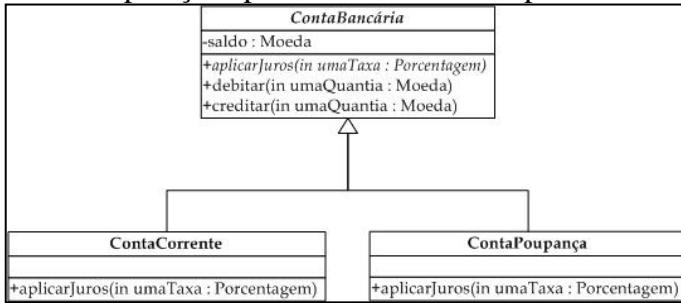


## Operações polimórficas

- Uma subclasse herda todas as propriedades de sua superclasse que tenham visibilidade pública ou protegida.
- Entretanto, pode ser que o comportamento de alguma operação herdada seja diferente para a subclasse.
- Nesse caso, a subclasse deve redefinir o comportamento da operação.
  - A assinatura da operação é reutilizada.
  - Mas, a implementação da operação (ou seja, seu **método**) é diferente.
- Operações polimórficas são aquelas que possuem mais de uma implementação
- Operações polimórficas possuem sua assinatura definida em diversos níveis de uma hierarquia gen/spec.
  - A assinatura é repetida na(s) subclasse(s) para enfatizar a redefinição de implementação.
  - O objetivo de manter a assinatura é garantir que as subclasses tenham uma interface em comum.
- Operações polimórficas facilitam a implementação.
  - Se duas ou mais subclasses implementam uma operação polimórfica, a mensagem para ativar essa operação é a mesma para todas essas classes.
  - No envio da mensagem, o remetente não precisa saber qual a verdadeira classe de cada objeto, pois eles aceitam a mesma mensagem.
  - A diferença é que os métodos da operação são diferentes em cada subclasse.
- A operação `obterPagamento` é polimórfica.



- Operações polimórficas também podem existir em classes abstratas.



- Operações polimórficas implementam o **princípio do polimorfismo**, no qual dois ou mais objetos respondem a mesma mensagem de formas diferentes.

```

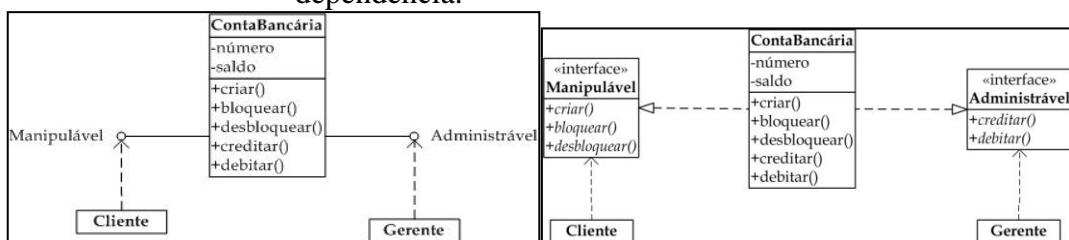
ContaCorrente cc;
ContaPoupanca cp;
...
List<ContaBancaria> contasBancarias;
...
contasBancarias.add(cc);
contasBancarias.add(cp);
...
for(ContaBancaria conta : contasBancarias) {
    conta.aplicarJuros();
}
...

```

## Interfaces

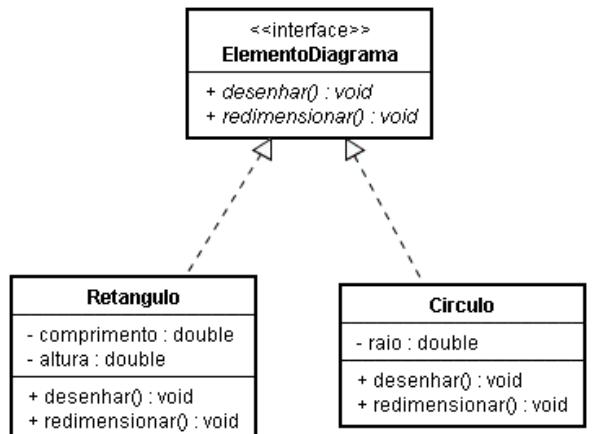
- Uma **interface** entre dois objetos compreende um conjunto de **assinaturas de operações** correspondentes aos serviços dos quais a classe do objeto cliente faz uso.
- Uma interface pode ser interpretada como um **contrato de comportamento** entre um objeto cliente e eventuais objetos fornecedores de um determinado serviço.
  - Contanto que um objeto fornecedor forneça implementação para a interface que o objeto cliente espera, este último não precisa conhecer a verdadeira classe do primeiro.

- Interfaces são utilizadas com os seguintes objetivos:
  - 1. Capturar semelhanças entre classes não relacionadas sem forçar relacionamentos entre elas.
  - 2. Declarar operações que uma ou mais classes devem implementar.
  - 3. Revelar as operações de um objeto, sem revelar a sua classe.
  - 4. Facilitar o desacoplamento entre elementos de um sistema.
- Nas LPOO modernas (Java, C#, etc.), interfaces são definidas de forma semelhante a classes.
  - Uma diferença é que todas as declarações em uma interface têm visibilidade pública.
  - Adicionalmente, uma interface não possui atributos, somente declarações de assinaturas de operações e (raramente) constantes.
- Notações para representar interfaces na UML:
  - A primeira notação é a mesma para classes. São exibidas as operações que a interface especifica. Deve ser usado o estereótipo <<interface>>.
  - A segunda notação usa um segmento de reta com um pequeno círculo em um dos extremos e ligado ao classificador.
    - Classes clientes são conectadas à interface através de um relacionamento de notação similar à do relacionamento de dependência.



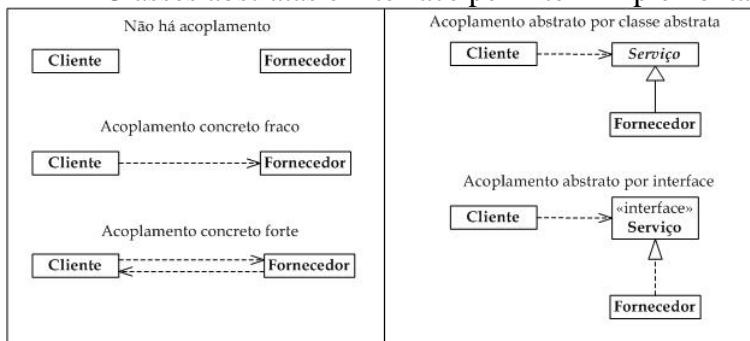
```

public interface ElementoDiagrama {
    double PI = 3.1425926; //static
and final constant.
    void desenhar();
    void redimensionar();
}
public class Circulo implements
ElementoDiagrama {
    ...
    public void desenhar() { /* draw a circle */ }
    public void redimensionar()
    { /* draw a circle */ }
}
public class Retangulo implements
ElementoDiagrama {
    ...
    public void desenhar() { /* draw a circle */ }
    public void redimensionar()
    { /* draw a circle */ }
}
  
```



## Acoplamento concreto e abstrato

- Usualmente, um objeto A faz referência a outro B através do conhecimento da classe de B.
  - Esse tipo de dependência corresponde ao que chamamos de **acoplamento concreto**.
- Entretanto, há outra forma de dependência que permite que um objeto remetente envie uma mensagem para um receptor sem ter conhecimento da verdadeira classe desse último.
  - Essa forma de dependência corresponde ao que chamamos de **acoplamento abstrato**.
  - A acoplamento abstrato é preferível ao acoplamento concreto.
- Classes abstratas e interface permitem implementar o acoplamento abstrato.



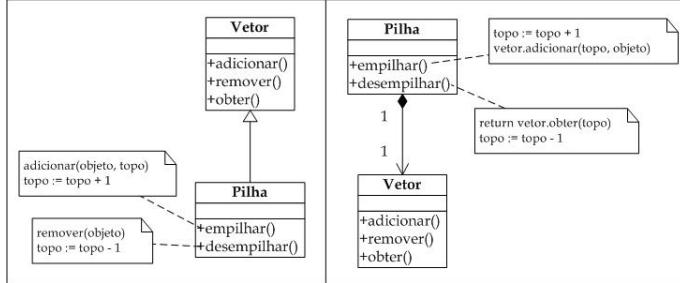
## Reuso através de generalização

- No reuso por generalização, subclasses que herdam comportamento da superclasse.
  - Exemplo: um objeto ContaCorrente não tem como atender à mensagem para executar a operação debitar só com os recursos de sua classe. Ele, então, utiliza a operação herdada da superclasse.
- Vantagem: fácil de implementar.
- Desvantagem:
  - Exposição dos detalhes da superclasse às subclasses (Violação do *princípio do encapsulamento*).
  - Possível violação do *Princípio de Liskov (regra da substituição)*.

## Reuso através de delegação

- A delegação é outra forma de realizar o reuso.
- “Sempre que um objeto não pode realizar uma operação por si próprio, ele delega uma parte dela para outro(s) objeto(s)”.
- A delegação é mais genérica que a generalização.
  - um objeto pode reutilizar o comportamento de outro sem que o primeiro precise ser uma subclasse do segundo.
- O compartilhamento de comportamento e o reuso podem ser realizados em tempo de execução.
- Desvantagens:
  - Desempenho (implica em cruzar a fronteira de um objeto a outro para enviar uma mensagem).
  - Não pode ser utilizada quando uma classe parcialmente abstrata está envolvida.
  -

## Generalização versus delegação

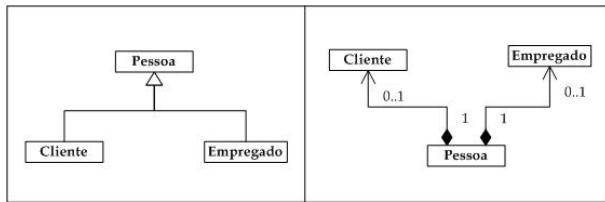


- Há vantagens e desvantagens tanto na generalização quanto na delegação.
- De forma geral, não é recomendado utilizar generalização nas seguintes situações:
  - Para representar papéis de uma superclasse.
  - Quando a subclasse herda propriedades que não se aplicam a ela.
  - Quando um objeto de uma subclasse pode se transformar em um objeto de outra subclasse.
    - Por exemplo, um objeto Cliente se transforma em um objeto Funcionário.

## Classificação dinâmica

- Problema na especificação e implementação de uma generalização.
  - Um mesmo objeto pode pertencer a múltiplas classes simultaneamente, ou passar de uma classe para outra.
- Considere uma empresa em que há empregados e clientes.
  - Pode ser que uma pessoa, em um determinado momento, seja apenas cliente;
  - Depois pode ser que ela passe a ser também um empregado da empresa.
  - A seguir essa pessoa é desligada da empresa, continuando a ser cliente.
- As principais LPOO (C++, Java, Smalltalk) não dão suporte direto à implementação da classificação dinâmica.
  - se um objeto é instanciado como sendo de uma classe, ele não pode pertencer posteriormente a uma outra classe.
- Solução parcial: definir todas as possíveis subclasses em uma determinada situação.
  - Exemplo (para a situação descrita há pouco): as classes Empregado, Cliente e EmpregadoCliente seriam criadas.
- Não resolve o problema todo:
  - Pode ser que um objeto mude de classe! (**metamorfose**)
  - A adição de novas classes à hierarquia torna o modelo ainda mais complexo.
- Uma melhor solução: utilizar a delegação.
  - Uma generalização entre cada subclasse e a superclasse é substituída por uma composição.
  - Exemplo no próximo slide.



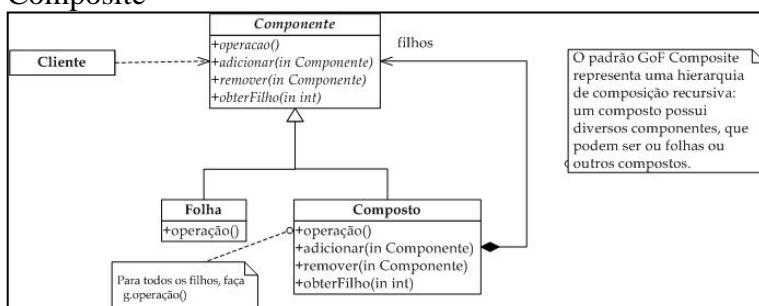


## 8.6 Padrões de projeto

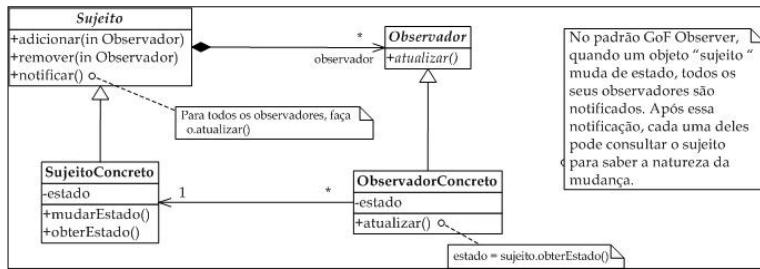
- É da natureza do desenvolvimento de software o fato de que os mesmos problemas tendem a acontecer diversas vezes.
- Um **padrão de projeto** corresponde a um esboço de uma solução reusável para um problema comumente encontrado em um contexto particular.
- Estudar esses padrões é uma maneira efetiva de aprender com a experiência de outros.
- O texto clássico sobre o assunto é o de Erich Gamma et al.
  - Esses autores são conhecidos *Gang of Four*.
  - Nesse livro, os autores catalogaram 23 padrões.
- Os padrões GoF foram divididos em três categorias:
  - **Criacionais**: procuram separar a operação de uma aplicação de como os seus objetos são criados.
  - **Estruturais**: provêem generalidade para que a estrutura da solução possa ser estendida no futuro.
  - **Comportamentais**: utilizam herança para distribuir o comportamento entre subclasses, ou agregação e composição para construir comportamento complexo a partir de componentes mais simples.

Criacionais	Estruturais	Comportamentais
Abstract Factory Builder <b>Factory Method</b> Prototype Singleton	Adapter Bridge <b>Composite</b> Decorator <b>Façade</b> Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator <b>Mediator</b> Memento <b>Observer</b> State <b>Strategy</b> Template Method Visitor

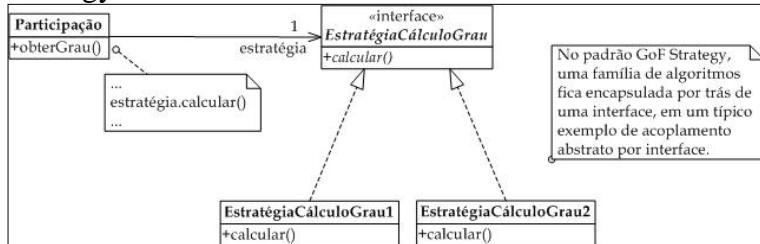
## Composite



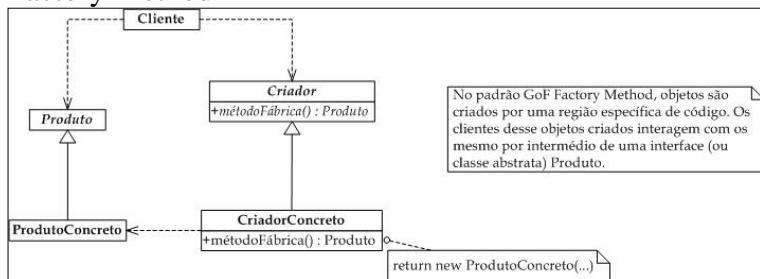
## Observer



## Strategy



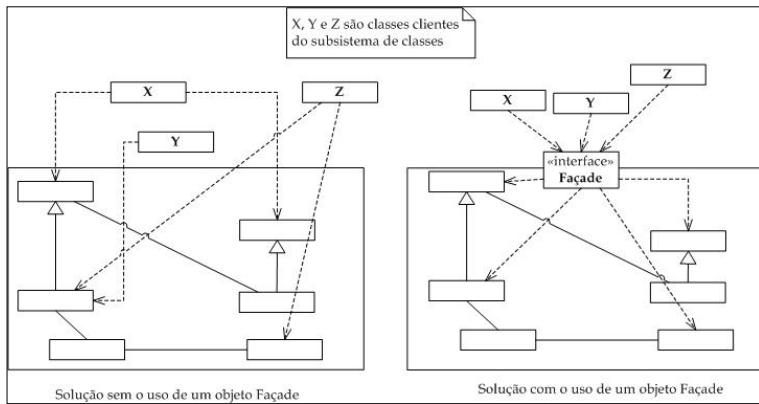
## Factory Method



## Mediator

- O padrão **Mediator** permite a um grupo de objetos interagirem, ao mesmo tempo em que mantém um acoplamento fraco entre os componentes desse grupo.
- A solução proposta pelo Mediator é definir um objeto, o *mediador*, para encapsular interações da seguinte forma: o resultado da interação de um subgrupo de objeto é passado a outro subgrupo pelo mediador.
- Dessa forma, os subgrupos não precisam ter conhecimento da existência um do outro e podem variar independentemente.
- *Objetos de controle* são exemplos de mediadores.

## Façade



## 9 Modelagem de estados

*Todos os adultos um dia foram crianças, mas poucos se lembram disso.*  
--O Pequeno Príncipe, Antoine de Saint-Exupéry

### Tópicos

- Introdução
- Diagramas de transição de estados
- Identificação dos elementos de um diagrama de estados
- Construção de diagramas de transição de estados
- Modelagem de estados no processo de desenvolvimento

### Introdução

- Objetos do mundo real se encontram em estados particulares a cada momento.
  - Uma jarra está cheia de líquido
  - Uma pessoa está cansada.
- Da mesma forma, cada objeto participante de um sistema de software orientado a objetos se encontra em um *estado* particular.
- Um objeto muda de estado quando acontece algum *evento* interno ou externo ao sistema.
- Durante a transição de um estado para outro, um objeto realiza determinadas *ações* dentro do sistema.
- Quando um objeto transita de um estado para outro, significa que o sistema no qual ele está inserido também está mudando de estado.

### 9.1 Diagramas de transição de estados

- Através da análise das *transições* entre *estados* dos objetos de um sistema de software, podem-se prever todas as possíveis *operações* realizadas, em função de *eventos* que possam ocorrer.
- O diagrama da UML que é utilizado para realizar esta análise é o *diagrama de transição de estado* (DTE).
- A UML tem um conjunto rico de notações para desenhar um DTE.
  - Estados
  - *Transições*
  - *Evento*

- *Ação*
- *Atividade*
- *Transições internas*
- *Estados aninhados*
- *Estados concorrentes*

### Estado

- Situação na vida de um objeto em que ele satisfaz a alguma condição ou realiza alguma atividade. É função dos *valores dos atributos* e (ou) das *ligações com outros objetos*.
  - O atributo *reservado* deste objeto livro tem valor *verdadeiro*.
  - Uma conta bancária passa para o *vermelho* quando o seu saldo fica *negativo*.
  - Um professor está *licenciado* quando não está ministrando curso algum durante o semestre.
  - Um tanque está *na reserva* quando nível de óleo está abaixo de 20%.
  - Um pedido está *atendido* quando todos os seus itens estão atendidos.
- Estados podem ser vistos como uma abstração dos atributos e associações de um objeto.

### Estado inicial e final

- O estado inicial indica o estado de um objeto quando ele é criado. Só pode haver um estado inicial em um DTE.
  - Essa restrição serve para definir a partir de que ponto um DTE deve começar a ser lido.
- O estado final é representado como um círculo “eclipsado” e indica o fim do ciclo de vida de um objeto.
  - é opcional e pode haver mais de um estado final em um DTE.
- Notação da UML para estados:



### Transições

- Os estados estão associados a outros pelas transições.
- Uma transição é mostrada como uma linha conectando estados, com uma seta apontando para um dos estados.
- Quando uma transição entre estados ocorre, diz-se que a transição foi disparada.
- Uma transição pode ser rotulada com uma expressão da seguinte forma:

evento (lista-parâmetros) [guarda] / ação

### Eventos

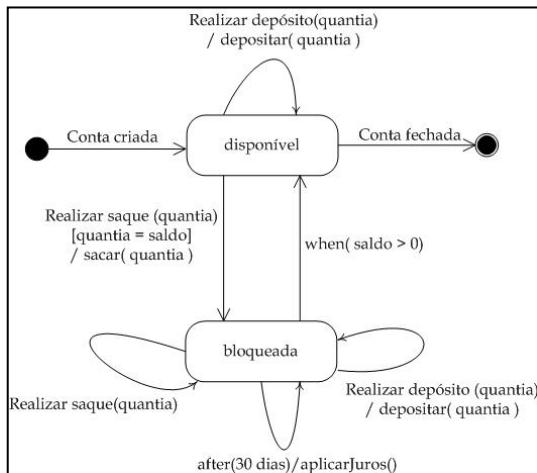
- Uma transição possui um evento associado.
- Um evento é algo que acontece em algum ponto no tempo e que pode modificar o estado de um objeto:
  - Pedido realizado
  - Fatura paga
  - Cheque devolvido

- Os eventos relevantes a um sistema de software podem ser classificados em nos seguintes tipos.
  - **Evento de chamada:** recebimento de uma mensagem de outro objeto.
  - **Evento de sinal:** recebimento de um sinal.
  - **Evento temporal:** passagem de um intervalo de tempo predefinido.
  - **Evento de mudança:** uma condição que se torna verdadeira.

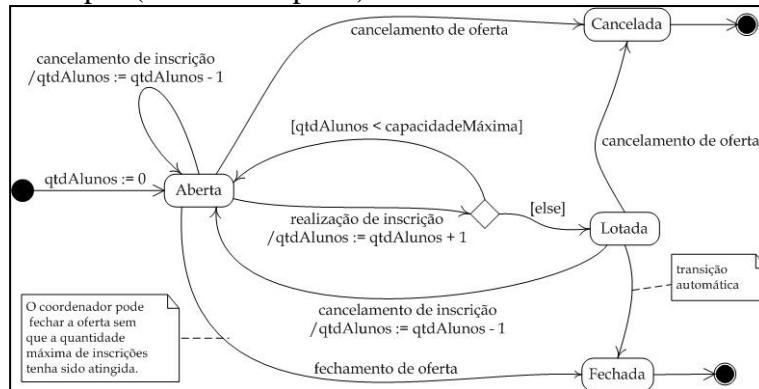
## Tipos de Evento

- Evento de chamada
  - Corresponde ao recebimento de uma mensagem de outro objeto.
  - Pode-se pensar neste tipo de evento como uma solicitação de serviço de um objeto a outro.
- Evento de sinal
  - Neste evento o objeto recebe um sinal de outro objeto que pode fazê-lo mudar de estado.
  - A diferença básica entre o evento de sinal e o evento de chamada é que neste último o objeto que envia a mensagem fica esperando a execução da mesma.
    - No evento de sinal, o objeto remetente continua o seu processamento após ter enviado o sinal.
- Evento de temporal
  - Corresponde à passagem de um intervalo de tempo predefinido.
    - O objeto pode interpretar a passagem de um certo intervalo de tempo como sendo um evento.
  - É especificado com a cláusula **after** seguida de um parâmetro que especifica um intervalo de tempo.
    - **after(30 segundos):** indica que a transição será disparada 30 segundos após o objeto ter entrado no estado atual.
- Evento de mudança
  - Corresponde a uma condição que se torna verdadeira.
  - É representado por uma expressão de valor lógico (verdadeiro ou falso) e é especificado utilizando-se a cláusula **when**.
    - **when(saldo > 0):** significa que a transição é disparada quando o valor do atributo saldo for positivo.
  - Eventos temporais também podem ser definidos utilizando-se a cláusula **when**.
    - **when(data = 13/07/2002)**
    - **when(horário = 00:00h)**

## Exemplo (ContaBancária)

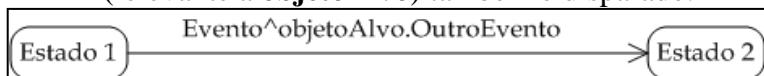


## Exemplo (OfertaDisciplina)



### Eventos resultando em eventos

- A ocorrência de um evento A relevante pode ocasionar a ocorrência de um evento B relevante para outro objeto.
- No exemplo a seguir, além da transição de estados, o evento **OutroEvento** (relevante a **objetoAlvo**) também é disparado.



### Condição de guarda

- É uma expressão de valor lógico que condiciona o disparo de uma transição.
- A transição correspondente é disparada se e somente se o evento associado ocorre e a condição de guarda é verdadeira.
  - Uma transição que não possui condição de guarda é sempre disparada quando o evento ocorre.
- A condição de guarda pode ser definida utilizando-se parâmetros passados no evento e também atributos e referências a ligações da classe em questão.

### Ações

- Ao transitar de um estado para outro, um objeto pode realizar uma ou mais **ações**.

- Uma ação é uma expressão definida em termo dos atributos, operações, associações da classe ou dos parâmetros do evento também podem ser utilizados.
- A ação associada a uma transição é executada se e somente se a transição for disparada.

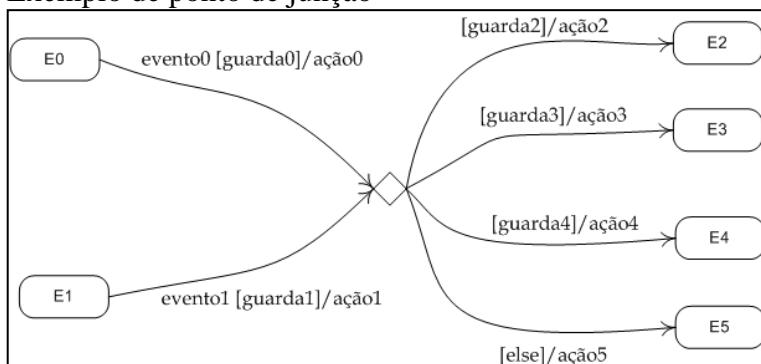
### Atividades

- Semelhantes a ações, atividades são algo que deve ser executado.
- No entanto, uma atividade pode ser *interrompida* (uma ação não pode).
  - Por exemplo, enquanto a atividade estiver em execução, pode acontecer um evento que a interrompa.
- Outra diferença: uma atividade sempre está associada a um estado (ao contrário, uma ação está associada a uma transição).

### Ponto de junção

- Pode ser que o próximo estado de um objeto varie de acordo com uma condição.
  - Se o valor da condição for verdadeiro, o objeto vai para um estado E1; se o valor for falso, o objeto vai para outro estado E2.
  - É como se a transição tivesse bifurcações, e cada transição de saída da bifurcação tivesse uma condição de guarda.
- Essa situação pode ser representada em um DTE através de um **ponto de junção**
- Pontos de junção permitem que duas ou mais transições compartilhem uma “trajetória de transições”.
- De uma forma geral, pode haver um número ilimitado de transições saindo de um ponto de junção.
- Pode haver também uma transição de saída que esteja rotulada com a cláusula **else**.
  - Se as outras condições forem falsas, a transição da cláusula **else** é disparada.

### Exemplo de ponto de junção

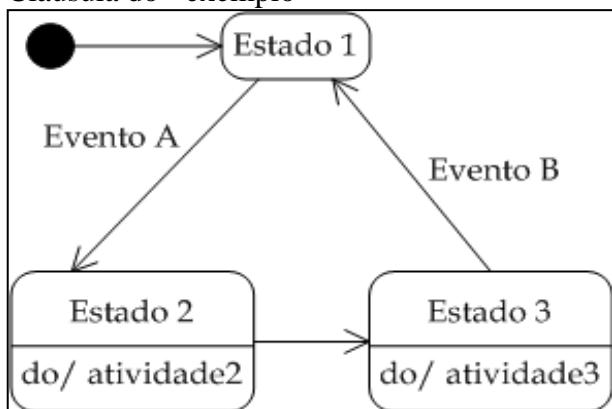


### Cláusulas

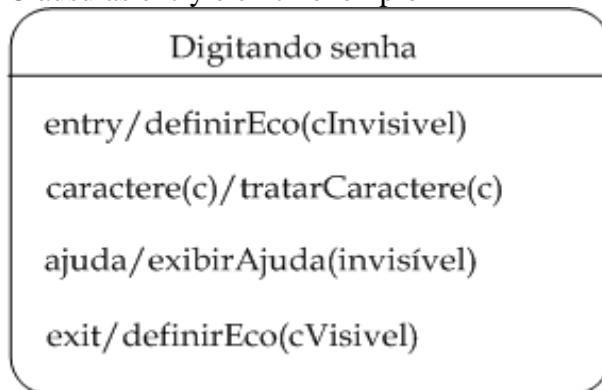
- No compartimento adicional de um retângulo de estado podem-se especificar ações ou atividades a serem executadas.
- Sintaxe geral: **evento / [ação | atividade]**
- Há três cláusulas predefinidas: *entry, exit, do*
- Cláusula **entry**
  - Pode ser usada para especificar uma ação a ser realizada no momento em que o objeto entra em um estado.

- A ação desta cláusula é sempre executada, independentemente do estado do qual o objeto veio.
  - É como se a ação especificada estivesse associada a todas as transições de entrada no estado.
- Cláusula **exit**
  - Serve para declarar ações que são executadas sempre que o objeto sai de um estado.
  - É sempre executada, independentemente do estado para o qual o objeto vai.
    - É como se a ação especificada estivesse associada a todas as transições de saída do estado.
- Cláusula **do**
  - Usada para definir alguma atividade a ser executada quando o objeto passa para um determinado estado.
  - Ao contrário da cláusula entry, serve para especificar uma atividade, em vez de uma ação.

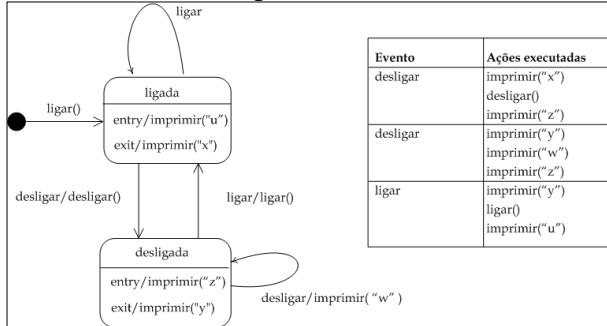
Cláusula do - exemplo



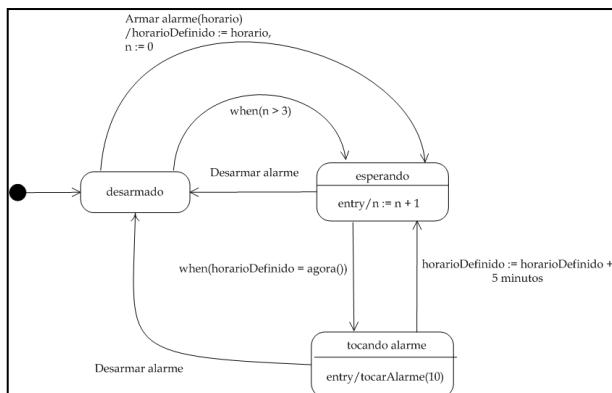
Cláusulas entry e exit - exemplo



## Cláusula do - exemplo



## Exemplo (Despertador)



## 9.2 Identificação dos elementos de um diagrama de estados

### Identificação de elementos do DTE

- Um bom ponto de partida para identificar estados é analisar os possíveis valores de seus atributos e as ligações que ele pode realizar com outros objetos.
- No entanto, a existência de atributos ou ligações não é suficiente para justificar a criação de um DTE.
  - O comportamento de objetos dessa classe deve depender de tais atributos ou ligações.
- Já que transições dependem de eventos para ocorrer, devem-se identificar estes eventos primeiramente.
- Além disso, deve-se examinar também se há algum fator que condicione o disparo da transição.
  - Se existir, este fator deve ser modelado como uma condição de guarda da transição.
- Um bom ponto de partida para identificar eventos é a descrição dos casos de uso.
- Os eventos encontrados na descrição dos casos de uso são externos ao sistema.
- Contudo, uma transição pode também ser disparada por um evento *interno* ao sistema.
- De uma forma geral, cada operação com visibilidade pública de uma classe pode ser vista como um evento em potencial.
- Uma outra fonte para identificação de eventos associados a transições é analisar as *regras de negócio*.
  - “Um cliente do banco não pode retirar mais de R\$ 1.000 por dia de sua conta”.

- “Os pedidos para um cliente não especial devem ser pagos antecipadamente”.
- “O número máximo de alunos por curso é igual a 30”.

### 9.3 Construção de diagramas de transição de estados

Um DTE para uma classe

- Os diagramas de estados são desenhados por classe.
  - Desvantagem: dificuldade na visualização do estado do sistema como um todo.
  - Essa desvantagem é parcialmente compensada pelos diagramas de interação.
- Nem todas as classes de um sistema precisam de um DTE.
  - Somente classes que exibem um comportamento dinâmico relevante.
  - Objetos cujo histórico precisa ser rastreado pelo sistema são típicos para se construir um diagrama de estados.

Procedimento para construção

1. Identifique os estados relevantes para a classe.
2. Identifique os eventos relevantes. Para cada evento, identifique qual a transição que ele ocasiona.
3. Para cada estado: identifique as transições possíveis quando um evento ocorre.
4. Para cada estado, identifique os eventos internos e ações correspondentes.
5. Para cada transição, verifique se há fatores que influenciam no seu disparo. (definição de condições de guarda e ações).
6. Para cada condição de guarda e para cada ação, identifique os atributos e ligações que estão envolvidos.
7. Defina o estado inicial e os eventuais estados finais.
8. Desenhe o DTE.

### 9.4 Modelagem de estados no processo de desenvolvimento

Modelagem de estados no PDS

- Os DTEs podem ser construídos com base nos diagramas de interação e nos diagramas de classes.
- Durante a construção do DTE para uma classe, novos atributos e operações podem surgir.
  - Essas novas propriedades devem ser adicionadas ao modelo de classes.
- A construção de um DTE freqüentemente leva à descoberta de novos atributos para uma classe
  - Principalmente atributos para servirem de abstrações para estados.
- Além disso, este processo de construção permite identificar novas operações na classe
  - Pois os objetos precisam reagir aos eventos que eles recebem.
- O comportamento de um objeto varia em função do estado no qual ele se encontra.
- Pode ser necessária a atualização de uma ou mais operações de uma classe para refletir o comportamento dos objetos em cada estado.
- Por exemplo, o comportamento da operação **sacar()** da classe **ContaBancária** varia em função do estado no qual esta classe se encontra

- Saques não podem ser realizados em uma conta que esteja no estado **bloqueado**.
- Os DTEs podem ser construídos com base nos diagramas de interação e nos diagramas de classes.
- Durante a construção do DTE para uma classe, novos atributos e operações podem surgir.
  - Essas novas propriedades devem ser adicionadas ao modelo de classes.
- A construção de um DTE freqüentemente leva à descoberta de novos atributos para uma classe
  - Principalmente atributos para servirem de abstrações para estados.
- Além disso, este processo de construção permite identificar novas operações na classe
  - Pois os objetos precisam reagir aos eventos que eles recebem.
- O comportamento de um objeto varia em função do estado no qual ele se encontra.
- Pode ser necessária a atualização de uma ou mais operações de uma classe para refletir o comportamento dos objetos em cada estado.
- Por exemplo, o comportamento da operação **sacar()** da classe **ContaBancária** varia em função do estado no qual esta classe se encontra
  - saques não podem ser realizados em uma conta que esteja no estado **bloqueada**.

## 10 Modelagem de atividades

*Qualquer um pode escrever código que um computador pode entender. Bons programadores escrevem código que seres humanos podem entender.*

-- Martin Fowler

### Tópicos

- Diagrama de atividade
- Diagrama de atividade no processo de desenvolvimento iterativo

#### Diagrama de atividade

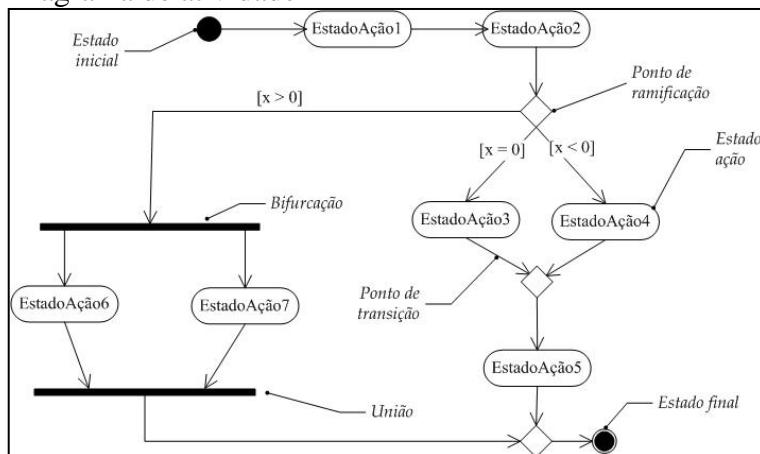
- Há diversos diagramas da UML que descrevem os aspectos dinâmicos de um sistema.
  - Diagramas de estados, diagramas de seqüência e de comunicação e diagrama de atividade
- O diagrama de atividade é um tipo especial de diagrama de estados, onde são representados os estados de uma atividade.
- Um diagrama de atividade exibe passos de uma computação.
  - Cada atividade é um passo da computação.
  - É orientado a fluxos de controle (ao contrário dos DTEs que são orientados a eventos).
- É um tipo de *fluxograma estendido*..., pois permitem representar ações concorrentes e sua sincronização.
- Elementos podem ser divididos em dois grupos: controle seqüencial e controle paralelo.
- Elementos utilizados em fluxos seqüenciais:
  - Estado ação

- Estado atividade
- Estado inicial e final, e condição de guarda
- Transição de término
- Pontos de ramificação e de união
- Elementos utilizados em fluxos paralelos:
  - Barras de sincronização
    - Barra de bifurcação (fork)
    - Barra de junção (join)

### Fluxos de controle seqüenciais

- Um estado em um diagrama de atividade pode ser:
  - Um *estado atividade* leva um certo tempo para ser finalizado.
  - Um *estado ação*: realizado instantaneamente.
- Deve haver um *estado inicial* e pode haver vários *estados finais* e *guardas* associadas a transições.
  - Pode não ter estado final, o que significa que o processo ou procedimento é cíclico.
- Uma *transição de término* significa o término de um passo e o consequente início do outro.
  - Em vez de ser disparada pela ocorrência de um evento, é disparada pelo término de um passo.
- Um *ponto de ramificação* possui uma única transição de entrada e várias transições de saída.
  - Para cada transição de saída, há uma condição de guarda associada.
  - Quando o fluxo de controle chega a um ponto de ramificação, uma e somente uma das condições de guarda deve ser verdadeira.
  - Pode haver uma transição com **[else]**.
- Um *ponto de união* reúne diversas transições que, direta ou indiretamente, têm um ponto de ramificação em comum.

### Diagrama de atividade

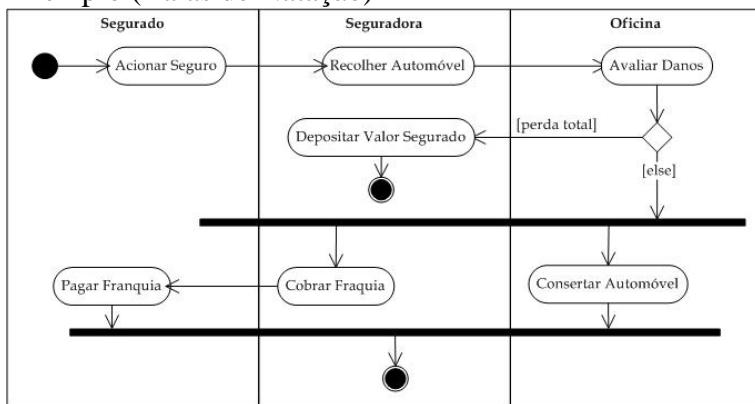


### Fluxos de controle paralelo

- Fluxos de controle paralelos: dois ou mais fluxos sendo executados simultaneamente.
- Uma **barra de bifurcação** recebe uma transição de entrada, e cria dois ou mais fluxos de controle paralelos.
  - Cada fluxo é executado independentemente e em paralelo com os demais.

- Uma **barra de junção** recebe duas ou mais transições de entrada e une os fluxos de controle em um único fluxo.
  - Objetivo: sincronizar fluxos paralelos.
  - A transição de saída da barra de junção somente é disparada quando todas as transições de entrada tiverem sido disparadas.
- Algumas vezes, as atividades de um processo podem ser distribuídas por vários agentes que o executarão.
  - Processos de negócio de uma organização.
- Isso pode ser representado através de **raias de natação** (swim lanes).
- As raias de natação dividem o diagrama de atividade em *compartimentos*.
- Cada compartimento contém atividades que são realizadas por uma entidade.

### Exemplo (Raias de Natação)



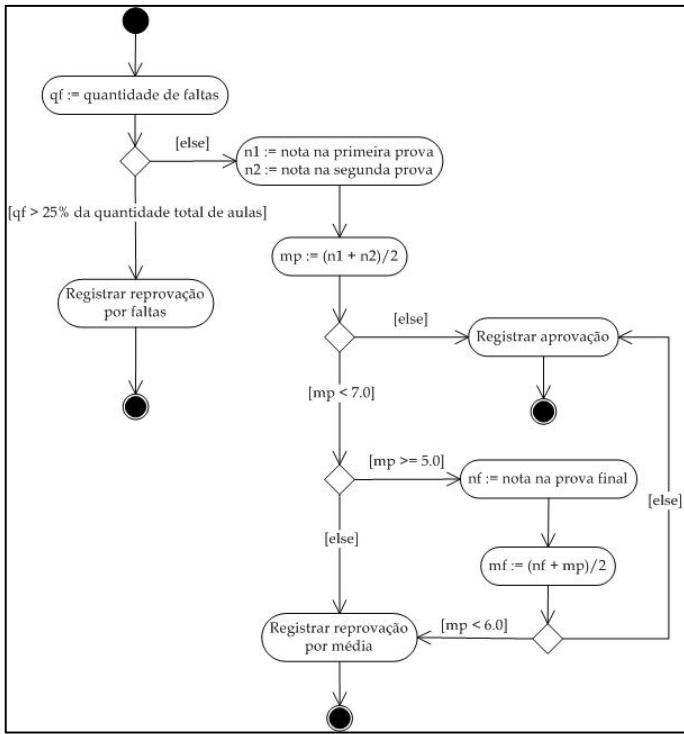
### Diagrama de atividade no processo de desenvolvimento iterativo

#### Usos de diagramas de atividades

- Não são freqüentemente utilizados na prática...
- Importante: na orientação a objetos o sistema é dividido em objetos, e não em módulos funcionais como na Análise Estruturada (Diagrama de Fluxos de Dados).

#### Modelar o processo do negócio

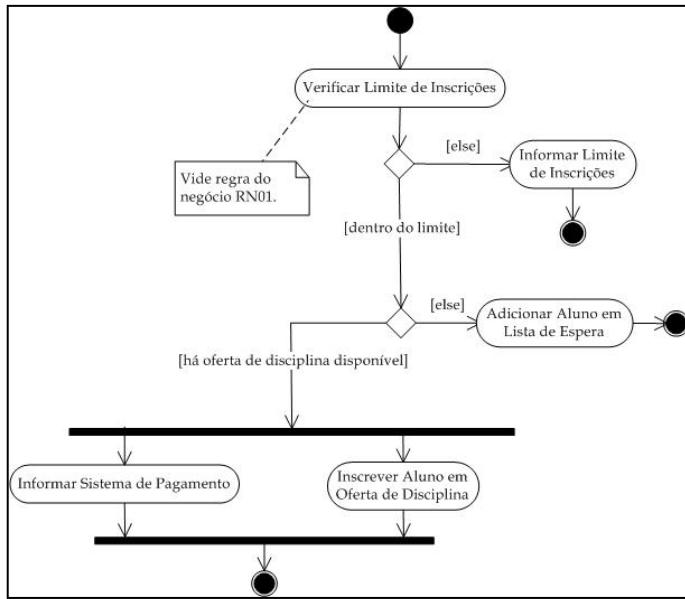
- *Modelagem* também é um processo de entendimento.
  - O desenvolvedor constrói modelos para entender melhor um problema.
- Neste caso, o enfoque está em entender o comportamento do sistema no decorrer de diversos casos de uso (*processos de negócio*).
  - Como determinados casos de uso do sistema se relacionam no decorrer do tempo.



### Modelar a lógica de um caso de uso

A realização de um caso de uso requer que alguma computação seja realizada.

- Esta computação pode ser dividida em atividades.
- “Passo P ocorre até que a C seja verdadeira”
- “Se ocorre C, vai para o passo P”.
- Nessas situações, é interessante complementar a descrição do caso de uso com um diagrama de atividade.
- Os fluxos principal, alternativos e de exceção podem ser representados em um único diagrama de atividade.
  - Complementar e não substituir a descrição.
- Identificação de atividades através do exame dos fluxos do caso de uso.
- Casos de uso são descritos na perspectiva dos atores, enquanto diagramas de atividade descrevem atividades internas ao sistema.



Modelar a lógica de uma operação

- Quando um sistema é adequadamente decomposto em seus objetos, as maioria das operações são bastante simples.
  - Estas não necessitam de modelagem gráfica.
- No entanto, pode haver a necessidade de descrever a lógica de uma operação mais complexa.
  - Implementação de regras de negócio.

## 11 Arquitetura do sistema

*Nada que é visto, é visto de uma vez e por completo.*  
--EUCLIDES

Tópicos

- Introdução
- Arquitetura lógica
  - Diz respeito a como um SSOO é decomposto em diversos subsistemas e como as suas classes são dispostas pelos diversos subsistemas.
- Alocação (arquitetura) física
  - Diz respeito a como os subsistemas devem ser dispostos fisicamente quando o sistema tiver de ser implantado.)
  - Dois aspectos importantes:
    - Alocação de camadas
    - Alocação de componentes
- Projeto da arquitetura no processo de desenvolvimento

Introdução

- Em um SSOO, os objetos interagem entre si através do envio de mensagens com o objetivo de executar suas tarefas.
  - Um SSOO também pode ser visto como um conjunto de *subsistemas* que o compõem.

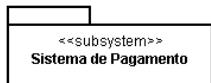
- A definição dos subsistemas de um SSOO é feita no *projeto da arquitetura* ou *projeto arquitetural*.
- Essa atividade define de que forma o sistema se divide em partes e quais são as interfaces entre essas partes.
- Vantagens de dividir um SSOO em subsistemas:
  - Produzir unidades menores de desenvolvimento;
  - Maximizar o reuso no nível de subsistemas componentes;
  - Ajuda a gerenciar a complexidade no desenvolvimento.
- Questões relacionadas à arquitetura de um sistema:
  - Como um sistema é decomposto em subsistemas, e como as suas classes são dispostas pelos diversos subsistemas?
  - Como subsistemas devem ser dispostos fisicamente quando o sistema tiver de ser implantado? (nós de processamento)
- De acordo com a especificação da UML: “Arquitetura de software é a estrutura organizacional do software. Uma arquitetura pode ser recursivamente decomposta em partes que interagem através de interfaces.”
- As decisões tomadas para a definição da arquitetura de software influenciam diretamente na forma como um SSOO irá atender a seus *requisitos não-funcionais*.

### 11.1 Arquitetura lógica

- Chamamos de arquitetura lógica à organização das classes de um SSOO em subsistemas, que correspondem a aglomerados de classes.
  - Um SSOO pode ser subdividido em diversos subsistemas
- Um subsistema provê serviços para outros através de sua interface.
- A interface de um subsistema corresponde ao conjunto de serviços que ele provê.
  - Cada subsistema provê ou utiliza serviços de outros subsistemas.

#### Diagrama de subsistemas

- Uma visão gráfica dos diversos componentes de um SSOO pode ser representada por um *diagrama de subsistemas*.
  - Um diagrama de susbistemas é um diagrama de pacotes, onde cada pacote representa um subsistema
  - Cada subsistema é rotulado com o estereótipo <<subsystem>>.
- Exemplo de susbsistema:



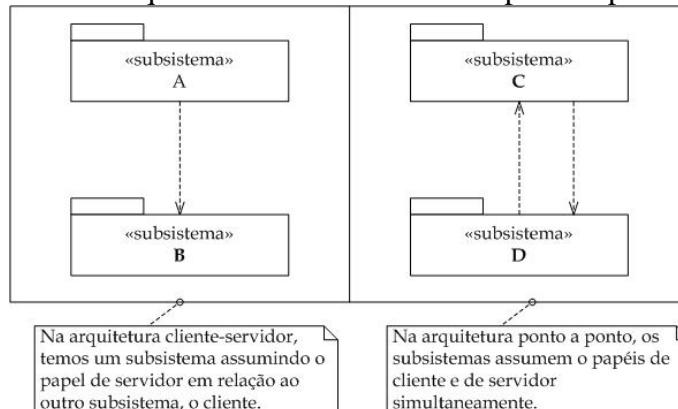
#### Alocação de classes a subsistemas

- Durante o desenvolvimento de um SSOO, seus subsistemas devem ser identificados, juntamente com as interfaces entre eles.
- Cada classe do sistema é, então, alocada aos subsistemas.
- Uma vez feito isso, esses subsistemas podem ser desenvolvidos quase que de forma independente uns dos outros.
- A seguir, são descritas algumas dicas que podem ser utilizadas para realizar a alocação de classes a subsistemas.
- Modelo de classes de domínio: este é o ponto de partida para a identificação dos subsistemas de um SSOO.

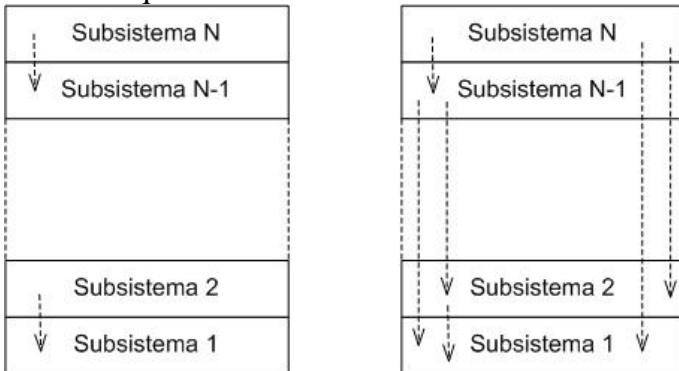
- Classes devem ser agrupadas segundo algum critério para formar subsistemas.
- Um critério de agrupamento possível: considerar as classes mais importantes do modelo de classes de domínio.
  - Para cada uma dessas classes, um subsistema é criado.
  - Outras classes menos importantes e relacionadas a uma classe considerada importante são posicionadas no subsistema desta última.
- Por outro lado, na fase de projeto:
  - Algumas das classes do modelo de domínio podem sofrer decomposições adicionais, o que resulta em novas classes.
  - Pode ser que uma classe de domínio seja ela própria um subsistema.
- Quando essas classes forem criadas, elas são adicionadas aos subsistemas pré-definidos.
- Subsistemas devem ser minimamente acoplados.
- Subsistemas devem ser maximamente coesivos.
- Dependências cíclicas entre subsistemas devem ser evitadas.
  - A alternativa para eliminar ciclos é quebrar um subsistema pertencente ao ciclo em dois ou mais. Uma outra solução é combinar dois ou mais subsistemas do ciclo em um único.
- Uma classe deve ser definida em um único subsistema (embora possa ser utilizada em vários).
  - O subsistema que define a classe deve mostrar todas as propriedades da mesma.
  - Outros subsistemas que fazem referência a essa classe podem utilizar a notação simplificada da mesma.

### Camadas de software

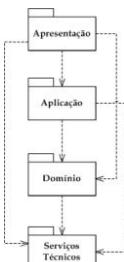
- Dizemos que dois subsistemas interagem quando um precisa dos serviços do outro.
- Há basicamente duas formas de interação entre subsistemas:
  - **ponto a ponto**: na arquitetura ponto a ponto, a comunicação pode acontecer em duas vias.
  - **cliente-servidor**: há a comunicação somente em uma via entre dois subsistemas, do cliente para o servidor. Nessa arquitetura, chamamos de camadas os subsistemas envolvidos.
- Essas formas de interação entre subsistemas influenciam o modo pelo qual os subsistemas são distribuídos fisicamente pelos nós de processamento.
- Arquiteturas cliente-servidor e ponto a ponto



- Um SSOO projetado em camadas pode ter uma *arquitetura aberta* ou uma *arquitetura fechada*.
  - Em uma arquitetura fechada, um componente de uma camada de certo nível somente pode utilizar os serviços de componentes da sua própria camada ou da imediatamente inferior.
  - Em uma arquitetura aberta, uma camada em certo nível pode utilizar os serviços de qualquer camada inferior.
- Na maioria dos casos práticos, encontramos sistemas construídos através do uso de uma arquitetura aberta.
- Arquiteturas abertas e fechadas



- Uma divisão tipicamente encontrada para as camadas lógicas de um SSOO é a que separa o sistema nas seguintes camadas:
  - *apresentação, aplicação, domínio e serviços técnicos*.
- Da esquerda para a direita, temos camadas cada vez mais genéricas.
- Também da esquerda para a direita, temos a ordem de dependência entre as camadas;
  - por exemplo a camada da apresentação depende (requisita serviços) da camada de aplicação, mas não o contrário.



- Princípio básico: camadas mais altas devem depender das camadas mais baixas, e não o contrário.
  - Essa disposição ajuda a gerenciar a complexidade através da divisão do sistema em partes menos complexas que o todo.
  - Também incentiva o reuso, porque as camadas inferiores são projetadas para serem independentes das camadas superiores.
  - O acoplamento entre camadas é mantido no nível mínimo possível.
  - Uma mudança em uma camada mais baixa que não afete a sua interface não implicará em mudanças nas camadas mais altas.

- E vice-versa, uma mudança em uma camada mais alta que não implica na criação de um novo serviço em uma camada mais baixa não irá afetar estas últimas.
- É importante notar que uma aplicação típica normalmente possui diversos subsistemas (ou pacotes) internamente a cada uma das camadas.
  - Por exemplo, em uma aplicação que forneça certo serviço que é acessível tanto por um usuário final, quanto por outra aplicação (através de um serviço WEB), há duas camadas de aplicação, possivelmente fazendo acesso a mesma camada da aplicação.
- Certa camada pode ser dividida verticalmente no que costumamos chamar de *partições*.
  - Por exemplo, em um sistema de vendas pela WEB, a camada de domínio pode ser decomposta nos seguintes subsistemas (partições): Clientes, Pedidos e Entregas.
- Durante a definição da arquitetura lógica de um SSOO, o uso de *padrões de projeto* é comum.
  - Para comunicação entre subsistemas, normalmente o padrão *Façade* é utilizado.
  - Para diminuir o acoplamento entre camadas (ou entre partições dentro de uma camada), o padrão *Factory Method* pode ser utilizado.
  - O padrão *Observer* também pode ser utilizado quando uma camada em certo nível precisa ser comunicar com uma camada de um nível superior.
    - O componente da camada inferior representa o sujeito, enquanto o componente da camada superior representa o observador. O sujeito não precisa ter conhecimento da classe específica do observador.

## 11.2 Implantação física

### Arquitetura de implantação

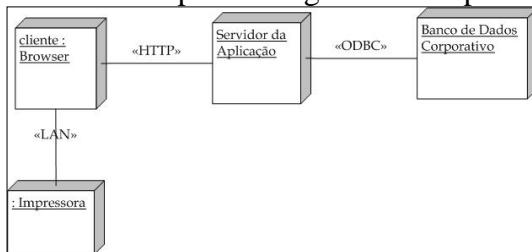
- Representa a disposição física do sistema de software pelo hardware disponível.
- A divisão de um sistema em camadas é independente da sua disposição física.
  - As camadas de software podem estar fisicamente localizadas em uma única máquina, ou podem estar distribuídas por diversos processadores.
  - Alternativamente, essas camadas podem estar distribuídas fisicamente em vários processadores. (Por exemplo, quando a camada da lógica do negócio é dividida em duas ou mais máquinas.)
- O modelo que representa a arquitetura física é denominado *modelo de implementação* ou *modelo da arquitetura física*.
- A arquitetura de implantação diz respeito à disposição dos subsistemas de um SSOO pelos nós de processamento disponíveis.
- Para sistemas simples, a arquitetura de implantação não tem tanta importância.
- No entanto, na modelagem de sistemas complexos, é fundamental conhecer quais são os componentes físicos do sistema, quais são as interdependências entre eles e de que forma as camadas lógicas do sistema são dispostas por esses componentes.

## Alocação de camadas

- Em um sistema construído segundo a arquitetura a cliente-servidor, é comum utilizar as definições das camadas lógicas como um guia para a alocação física dos subsistemas.
- Sendo assim, a cada nó de processamento são alocadas uma ou mais camadas lógicas.
- Note que o termo *camada* é normalmente utilizado com dois sentidos diferentes:
  - Para significar uma camada lógica (*layer*)
  - E para significar uma camada física, esta última normalmente associada a um nó de processamento (*tier*).
- Vantagens da alocação das camadas lógicas a diferentes nós de processamento:
  - A divisão dos objetos permite um maior grau de manutenção e reutilização, porque sistemas de software construídos em camadas podem ser mais facilmente estendidos.
  - Sistemas em camadas também são mais adaptáveis a uma quantidade maior de usuários.
    - Servidores mais potentes podem ser acrescentados para compensar um eventual crescimento no número de usuários do sistema.
- No entanto, a divisão do sistema em camadas apresenta a desvantagem de *potencialmente* diminuir o desempenho do mesmo.
  - a cada camada, as representações dos objetos sofrem modificações, e essas modificações levam tempo para serem realizadas.
- Um SSOO que divide a interação com o usuário e o acesso aos dados em dois subsistemas é denominado sistema cliente-servidor em duas camadas.
- A construção de sistemas em duas camadas é vantajosa quando o número de clientes não é tão grande (por exemplo, uma centena de clientes interagindo com o servidor através de uma rede local).
- No entanto, o surgimento da Internet causou problemas em relação à estratégia cliente-servidor em duas camadas.
  - Isso porque a idéia básica da Internet é permitir o acesso a variados recursos através de um programa navegador (browser).
- A solução encontrada para o problema da arquitetura em duas camadas foi simplesmente dividir os sistemas em mais camadas de software.
  - Sistemas construídos segundo essa estratégia são denominados sistemas cliente-servidor em três camadas ou sistemas cliente-servidor em quatro camadas.
- Entretanto, a idéia básica original permanece: dividir o processamento do sistema em diversos nós.
  - Em particular, uma disposição bastante popular, principalmente em aplicações para a WEB, é a arquitetura cliente-servidor em três camadas...



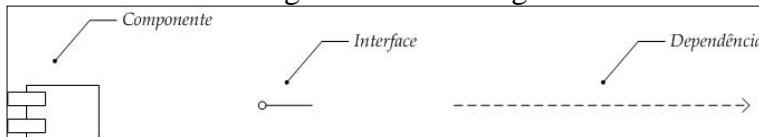
- Na arquitetura cliente-servidor em três camadas:
  - A camada lógica de apresentação fica em um nó de processamento (conhecido como *presentation tier*)
  - As camadas lógicas da aplicação e do domínio ficam juntas em outro nó (camada física denominada *middle tier*).
    - A camada física do meio corresponde ao *servidor da aplicação*.
    - A camada de apresentação requisita serviços ao servidor da aplicação.
    - É também possível haver mais de um servidor de aplicação, com o objetivo de aumentar a disponibilidade e o desempenho da aplicação.
  - A camada física do meio faz acesso a outra camada física, onde normalmente se encontra um SGBD.
    - Esta última camada física é chamada de *camada de dados (data tier)*.
- Uma vez definidas as alocações das camadas lógicas aos nós de processamento, podemos fazer a representação gráfica com suporte da UML, através do *diagrama de implantação*.
- Os elementos desse diagrama são os **nós** e as **conexões**.
- Um nó representa um recurso computacional e normalmente possui uma memória e alguma capacidade de processamento.
  - Exemplos: processadores, dispositivos, sensores, roteadores ou qualquer objeto físico de importância para o sistema de software.
- Os nós são ligados uns aos outros através de conexões.
  - As conexões representam mecanismos de comunicação: meios físicos (cabo coaxial, fibra ótica etc.) ou protocolos de comunicação (TCP/IP, HTTP etc.).
- Exemplo de diagrama de implantação



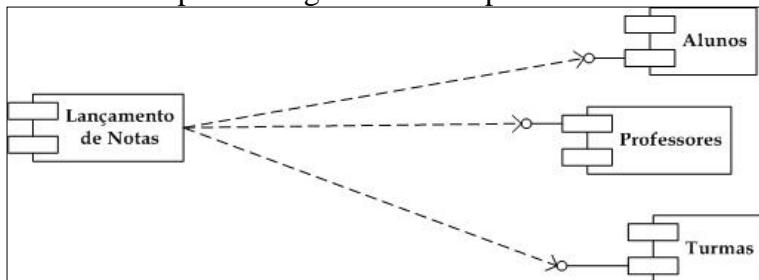
#### Alocação de Componentes

- Na arquitetura (alocação) física, devemos também definir quais os **componentes de software** de cada camada.
- Um componente de software é uma unidade que existe a tempo de execução, que pode ser utilizada na construção de vários sistemas e que pode ser substituída por outra unidade que tenha a mesma funcionalidade.

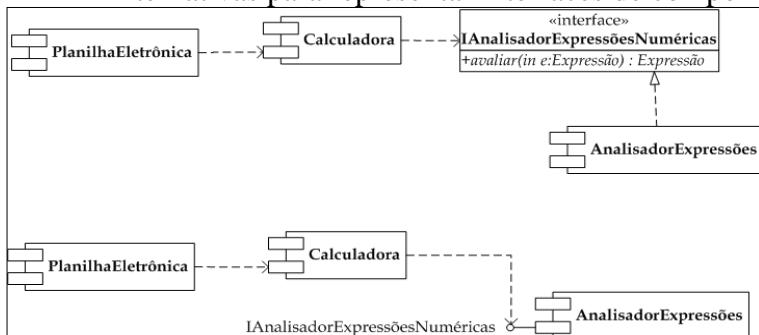
- As tecnologias COM (Microsoft), CORBA (OMG) e EJB (Sun) são exemplos de tecnologias baseadas em componentes.
- Um componente provê acesso aos seus serviços através de uma *interface*.
  - Quando construído segundo o paradigma OO, um componente é composto de diversos objetos.
  - Nesse caso, a interface do componente é constituída de um ou mais serviços que as classes dos referidos objetos implementam.
- A UML define uma forma gráfica para representar componentes visualmente, o diagrama de componentes.
- Esse diagrama mostra os vários componentes de software e suas dependências.
- Os elementos gráficos desse diagrama são ilustrados na figura abaixo.



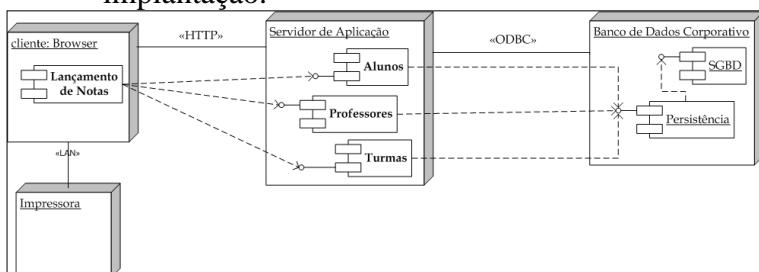
- Exemplo de diagrama de componentes.



- Alternativas para representar interfaces de componentes.



- Exemplo de diagrama de componentes embutido em um diagrama de implantação.



- A atividade de alocação de componentes aos nós físicos só tem sentido para sistemas distribuídos.
  - Para sistemas que utilizam um único processador, não há necessidade desta atividade.

- Um dos principais objetivos: distribuir a carga de processamento do sistema para aumentar o desempenho.
  - No entanto, nem sempre isso aumenta o desempenho.
  - Isso porque a sobrecarga de comunicação entre os nós pode anular os ganhos obtidos com a distribuição do processamento.
- Envio de mensagem versus “distância”
  - Dentro de um processo executando em um nó.
  - Entre processos no mesmo nó.
  - Ultrapassa as fronteiras de um nó para ser executada em outra máquina.
- Portanto, durante a alocação de componentes, o arquiteto de software deve considerar diversos fatores.
  - Muitos desses fatores se sobrepõem ou são incompatíveis.
- Envio de mensagem versus “distância”
  - Dentro de um processo executando em um nó.
  - Entre processos no mesmo nó.
  - Ultrapassa as fronteiras de um nó para ser executada em outra máquina.
- Portanto, durante a alocação de componentes, o arquiteto de software deve considerar diversos fatores.
  - Muitos desses fatores se sobrepõem ou são incompatíveis.
- Fatores relacionados ao desempenho:
  - Utilização de dispositivos
  - Carga computacional
  - Capacidade de processamento dos nós
  - Realização de tarefas
  - Tempo de resposta
- Outros fatores:
  - Outros requisitos não funcionais do sistema
  - Segurança
  - Diferenças de plataformas
  - Características dos usuários do sistema
  - Necessidade ou benefícios da distribuição das camadas lógicas do sistema
  - Redundância

### **11.3 Projeto da arquitetura no processo de desenvolvimento**

- A construção dos diagramas de componentes é iniciada na fase de elaboração (projeto da arquitetura) e refinada na fase de construção (projeto detalhado).
- A construção de diagramas de componentes se justifica para componentes de execução.
  - Permite visualizar as dependências entre os componentes e a utilização de interfaces a tempo de execução do sistema.
  - Sistema distribuído: representar seus componentes utilizando diagramas de implantação.
- Não é recomendável usar diagramas de componentes para representar dependências de compilação entre os elementos do código fonte do sistema.
  - A maioria dos ambientes de desenvolvimento tem capacidade de manter as dependências entre códigos fonte, códigos objeto, códigos executáveis e páginas de script.

- Só adiciona mais diagramas que terão que ser mantidos e que não terão uma real utilidade.
- Há também vários sistemas de gerenciamento de configurações e de versões de código (e.g., CVS).
- Em relação ao diagrama de implantação, sua construção tem início na fase de elaboração.
- Na fase de construção, os componentes são adicionados aos diversos nós.
  - Cada versão do sistema corresponde a uma versão do DI, que exibe os componentes utilizados na construção daquela versão.
  - O DI deve fazer parte dos manuais para instalação e operacionalização do sistema.
- Nem todo sistema necessita de diagramas de componentes ou diagramas de implantação.
  - Sistemas simples versus sistemas complexos.

## 12 Mapeamento de objetos para o modelo relacional

*“Na época, Nixon estava normalizando as relações com a China. Eu pensei que se ele podia normalizar relações, eu também podia.”* –E.F. Codd

### Tópicos

- Introdução
- Projeto de banco de dados
- Construção da camada de persistência

### Introdução

- Relevância do mapeamento de objetos para o modelo relacional:
  - A tecnologia OO como forma usual de desenvolver sistemas de software.
  - Sem dúvida os SGBDR dominam o mercado comercial.

Os princípios básicos do paradigma da orientação a objetos e do modelo relacional são bastante diferentes. No modelo de objetos, os elementos (objetos) correspondem a abstrações de comportamento. No modelo relacional, os elementos correspondem a dados no formato tabular.

- Os objetos de um sistema podem ser classificados em persistentes e transitentes.
- **Objetos transitentes:** existem somente na memória principal.
  - Objetos de controle e objetos de fronteira.
- **Objetos persistentes:** têm uma existência que perdura durante várias execuções do sistema.
  - Precisam ser armazenados quando uma execução termina, e restaurados quando outra execução é iniciada.
  - Tipicamente objetos de entidade.
- Para objetos persistentes, surge o problema de conciliar as informações representadas pelo estado de um objeto e pelos dados armazenados em registros de uma tabela.
- **O descasamento de informações (impedance mismatch)** é um termo utilizado para denotar o problema das diferenças entre as representações do modelo OO e do modelo relacional.

- Uma proporção significativa do esforço de desenvolvimento recai sobre a solução que o desenvolvedor deve dar a este problema.

### 12.1 Projeto de banco de dados

- Uma das primeiras atividades do projeto detalhado de um SSOO é o desenvolvimento do banco de dados a ser utilizado, se este não existir.
- Essa atividade corresponde ao **projeto do banco de dados**.
- As principais tarefas no projeto do banco de dados são:
  - Construção do esquema do banco de dados
  - Criação de índices
  - Armazenamento físico dos dados
  - Definição de visões sobre os dados armazenados.
  - Atribuição de direitos de acesso
  - Políticas de backup dos dados
- Restrição de escopo: apenas consideramos o aspecto de mapeamento de informações entre os modelos OO e relacional.
  - Ou seja, o mapeamento do modelo de classes para o modelo relacional.
  - Esse mapeamento possibilita a criação do **esquema do banco de dados**.
- Atualmente, há diversas ferramentas que automatiza grande parte desse mapeamento (engenharia direta e reversa).
  - Mas, nem sempre uma ferramenta está disponível.
  - Mesmo na existência de uma ferramenta, é importante para o desenvolvedor um conhecimento básico dos procedimentos do mapeamento.



#### Conceitos do modelo relacional

- O modelo relacional é fundamentado no conceito de **relação**.
- Cada coluna de uma relação pode conter apenas **valores atômicos**.
- Uma **chave primária**: colunas cujos valores podem ser utilizados para identificar unicamente cada linha de uma relação.
- Associações entre linhas: valores de uma coluna fazem referência a valores de outra coluna. (**chave estrangeira**).
  - Uma chave estrangeira também pode conter **valores nulos**, representados pela constante **NULL**.
- O **NULL** normalmente é usado para indicar que um valor não se aplica, ou é desconhecido, ou não existe.

The diagram illustrates the mapping of objects to a relational model. It shows four tables: Departamento, Alocacao, Projeto, and Empregado.

Departamento			
<u>id</u>	<u>sigla</u>	<u>nome</u>	<u>idGerente</u>
13	RH	Recursos Humanos	5
14	INF	Informática	2
15	RF	Recursos Financeiros	6

Alocacao		
<u>id</u>	<u>idProjeto</u>	<u>idEmpregado</u>
100	1	1
101	1	2
102	2	1
103	3	5
104	4	2

Projeto		
<u>id</u>	<u>nome</u>	<u>verba</u>
1	PNADO	R\$ 7.000
2	BMMO	R\$ 3.000
3	SGILM	R\$ 6.000
4	ACME	R\$ 8.000

Empregado						
<u>id</u>	<u>matrícula</u>	<u>CPF</u>	<u>nome</u>	<u>endereço</u>	<u>CEP</u>	<u>idDepartamento</u>
1	10223	038488847-89	Carlos	Rua 24 de Maio, 40	22740-002	13
2	10490	024488847-67	Marcelo	Rua do Bispo, 1000	22733-000	13
3	10377	NULL	Adelci	Av. Rio Branco, 09	NULL	NULL
4	11057	0345868378-20	Roberto	Av. Apiacás, 50	NULL	14
5	10922	NULL	Aline	R. Uruguaiana, 50	NULL	14
6	11345	0254647888-67	Marcelo	NULL	NULL	15

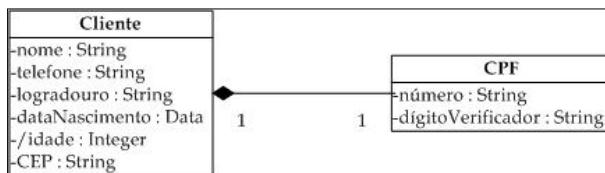
### Mapeamento de objetos para o modelo relacional

- Utilização de um SGBDR: necessidade do mapeamento dos valores de atributos de objetos persistentes para tabelas.
- É a partir do modelo de classes que o mapeamento de objetos para o modelo relacional é realizado.
  - Semelhante ao de mapeamento do MER.
  - Diferenças em virtude de o modelo de classes possuírem mais recursos de representação que o MER.
- Importante: o MER e o modelo de classes não são equivalentes.
  - Esses modelos são freqüentemente confundidos.
  - O MER é um modelo de dados; o modelo de classes representa objetos (dados e comportamento).
- Aqui, utilizamos a seguinte notação (simplificada):
  - Cada relação é representada através do seu nome e dos nomes de suas colunas entre parênteses.
  - Chaves primárias são sublinhadas
  - Chaves estrangeiras são tracejadas.
- Os exemplos dados a seguir utilizam sempre uma **coluna de implementação** como chave primária de cada relação.
  - Uma coluna de implementação é um identificador sem significado no domínio de negócio.
  - Essa abordagem é utilizada:
    - Para manter uma padronização nos exemplos

- É por ser uma das melhores maneiras de associar identificadores a objetos mapeados para tabelas.

#### Mapeamento: Classes e seus atributos

- Classes são mapeadas para relações.
  - Caso mais simples: mapear cada classe como uma relação, e cada atributo como uma coluna.
  - No entanto, pode não haver correspondência unívoca entre classes e relações..
- Para atributos o que vale de forma geral é que um atributo será mapeado para uma ou mais colunas.
- Nem todos os atributos de uma classe são persistentes (atributos derivados).



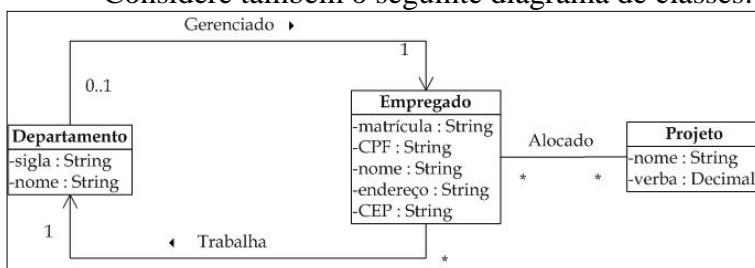
Cliente([id](#), CPF, nome, telefone, logradouro, dataNascimento, idCEP)

CPF([id](#), número, sufixo)

Cliente([id](#), nome, telefone, logradouro, dataNascimento, CPF, CEP)

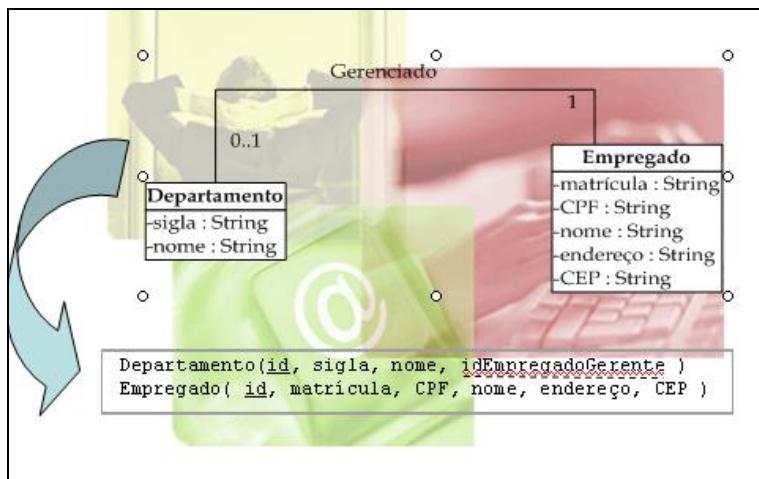
#### Mapeamento de associações

- O procedimento utiliza o conceito de **chave estrangeira**.
- Há três casos, cada um correspondente a um tipo de **conectividade**.
- Nos exemplos dados a seguir, considere, sem perda de generalidade, que:
  - há uma associação entre objetos de duas classes, Ca e Cb.
  - Ca e Cb foram mapeadas para duas relações separadas, Ta e Tb.
- Considere também o seguinte diagrama de classes:



#### Mapeamento de associações 1:1

- Deve-se adicionar uma chave estrangeira em uma das duas relações para referenciar a chave primária da outra relação.
- Escolha da relação na qual a chave estrangeira deve ser adicionada com base na **participação**.
- Há três possibilidades acerca da conectividade:
  - Obrigatória em ambos os extremos.
  - Opcional em ambos os extremos.
  - Obrigatória em um extremo e opcional no outro extremo.



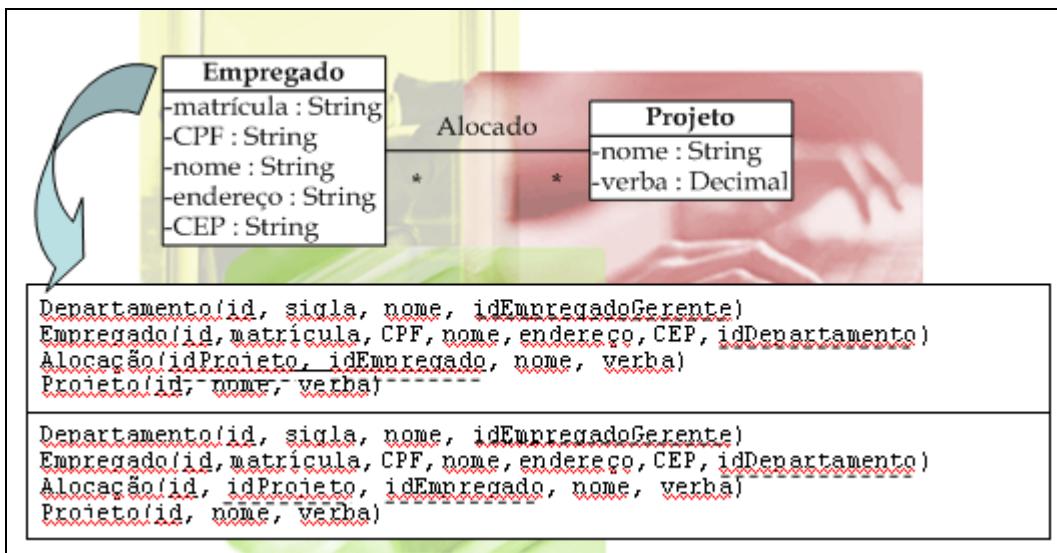
### Mapeamento de associações 1-muitos

- Seja Ca a classe na qual cada objeto se associa com muitos objetos da classe Cb.
- Sejam Ta e Tb as relações resultantes do mapeamento de Ca e Cb, respectivamente.
- Neste caso, deve-se adicionar uma chave estrangeira em Ta para referenciar a chave primária de Tb.



### Mapeamento de associações muitos-muitos

- Seja Ca a classe na qual cada objeto se associa com muitos objetos da classe Cb.
- Sejam Ta e Tb as relações resultantes do mapeamento de Ca e Cb, respectivamente.
- Uma **relação de associação** deve ser criada.
  - Uma relação de associação serve para representar a associação muitos para muitos entre duas ou mais relações.
- Equivalente à aplicação do mapeamento *um para muitos* duas vezes, considerando-se os pares (Ta, Tassoc) e (Tb, Tassoc).
- Alternativas para definir a chave primária de Tassoc.
  - Definir uma **chave primária composta**.
  - Criar uma coluna de implementação que sirva como chave primária simples da relação de associação.

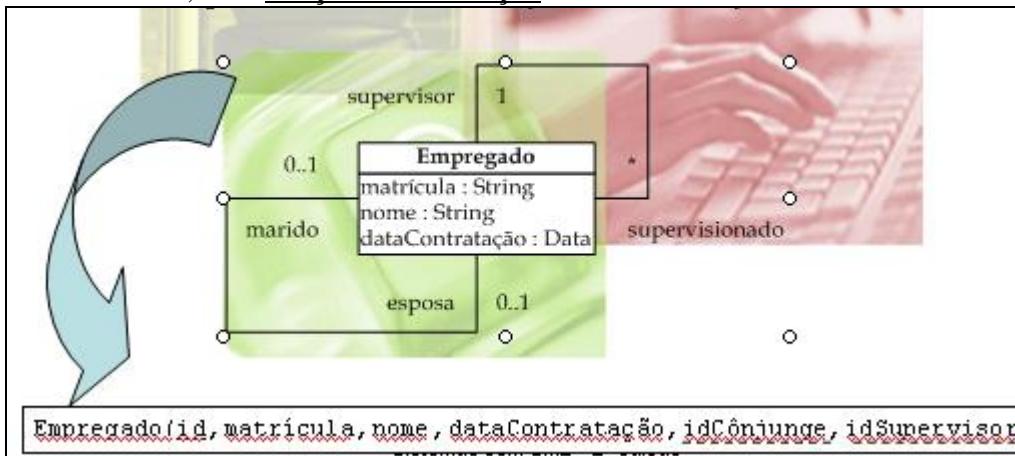


#### Mapeamento de agregações

- Forma especial de associação → *mesmo* procedimento para realizar o mapeamento de associações pode ser utilizado.
- No entanto, a diferença semântica influí na forma como o SGBDR deve agir quando um registro da relação correspondente ao *todo* deve ser excluído ou atualizado.
  - Remoção ou atualização em cascata.
  - Pode ser implementado como *gatilhos* e *procedimentos armazenados*.
- O padrão de acesso em agregações (composições) também é diferente do encontrado nas associações.
  - Quando um objeto todo deve ser restaurado, é natural restaurar também os objetos parte.
  - Em associações, isso nem sempre é o caso.
  - Definição de *índices* adequados é importante para acesso eficiente aos objetos *parte*.

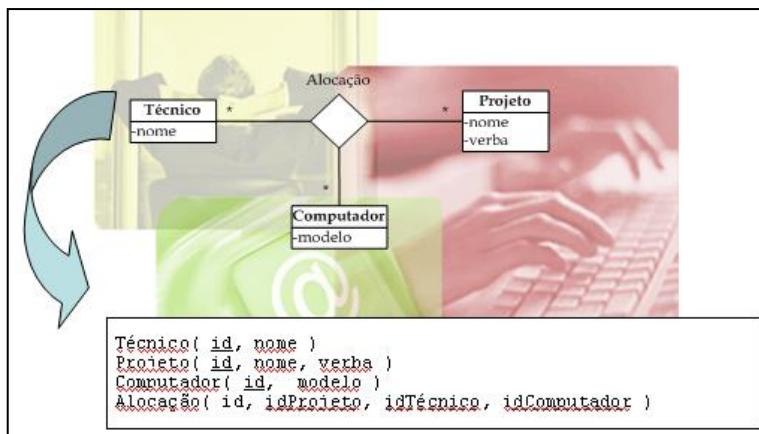
#### Mapeamento de associações reflexivas

- Forma especial de associação → *mesmo* procedimento para realizar o mapeamento de associações pode ser utilizado.
- Em particular, em uma associação reflexiva de conectividade *muitos para muitos*, uma *relação de associação* deve ser criada.



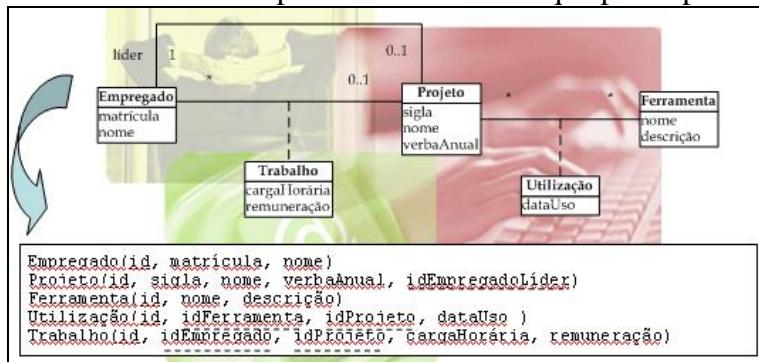
## Mapeamento de associações n-árias

- Associações n-árias ( $n \geq 3$ ): procedimento semelhante ao utilizado para associações binárias de conectividade *muitos para muitos*.
  - Uma relação para representar a associação é criada.
  - São adicionadas nesta relação chaves estrangeiras.
  - Se a associação n-ária possuir uma classe associativa, os atributos desta são mapeados como colunas da relação de associação.



## Mapeamento de classes associativas

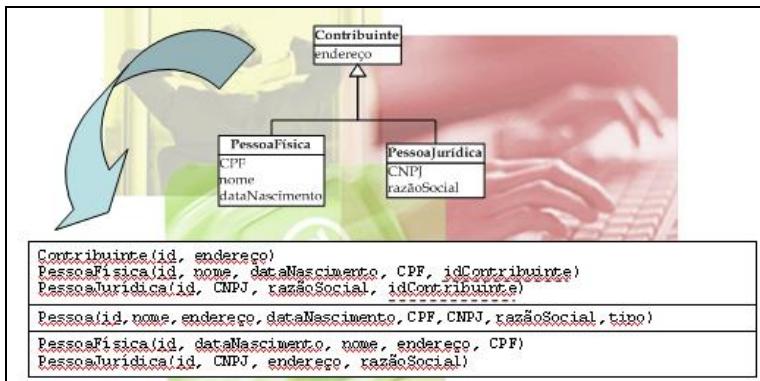
- Para cada um dos casos de mapeamento de associações, há uma variante onde uma classe associativa é utilizada.
- Mapeamento é feito através da criação de uma relação para representá-la.
  - Os atributos da classe associativa são mapeados para colunas dessa relação.
  - Essa relação deve conter chaves estrangeiras que referenciem as relações correspondentes às classes que participam da associação.



## Mapeamento de generalizações

- Três formas alternativas de mapeamento:
  - Uma relação para cada classe da hierarquia
  - Uma relação para toda a hierarquia
  - Uma relação para cada classe concreta da hierarquia
- Nenhuma** das alternativas de mapeamento de generalização pode ser considerada a melhor dentre todas.
  - Cada uma delas possui vantagens e desvantagens.
  - A escolha de uma delas depende das do sistema sendo desenvolvido.

- A equipe de desenvolvimento pode decidir implementar mais de uma alternativa.



- A 1<sup>a</sup> alternativa (uma relação para cada classe da hierarquia) é a que melhor reflete o modelo OO.
  - Classe é mapeada para uma relação
  - As colunas desta relação são correspondentes aos atributos específicos da classe.
  - Desvantagem: desempenho da manipulação das relações.
    - Inserções e remoções e *junções*.
- A 2<sup>a</sup> alternativa de implementação é bastante simples, além de facilitar situações em que objetos mudam de classe.
  - Desvantagem: alteração de esquema
    - Adição ou remoção de atributos.
    - Tem o potencial de desperdiçar bastante espaço de armazenamento:
      - Hierarquia com várias classes “irmãs”
      - Objetos pertencem a uma, e somente uma classe da hierarquia.
- A 3<sup>a</sup> alternativa apresenta a vantagem de agrupar os objetos de uma classe em uma única relação.
- Desvantagem: quando uma classe é modificada, cada uma das relações correspondentes às suas subclasses deve ser modificada.
  - Todas as relações correspondentes a subclasses devem ser modificadas quando a definição da superclasse é modificada.

## 12.2 Construção da camada de persistência

- Além da construção do esquema de banco de dados, outros aspectos importantes e relativos ao armazenamento de objetos em um SGBDR devem ser definidos.
- Alguns desses aspectos são enumerados a seguir.
  - **Materialização**: restaurar um objeto a partir do banco de dados, quando necessário.
  - **Atualização**: enviar modificações sobre um objeto para o banco de dados.
  - **Remoção**: remover um objeto do armazenamento persistente.
- Esses aspectos estão relacionados a funcionalidades que implementam o transporte de objetos da memória principal alocada ao SSOO para um SGBD e vice-versa.
- Para isolar os objetos do negócio de detalhes de comunicação com o SGBD, uma **camada de persistência** pode ser utilizada.

- O objetivo de uma camada de persistência é isolar os objetos do SSOO de mudanças no mecanismo de armazenamento.
  - Se um SGBD diferente tiver que ser utilizado pelo sistema , por exemplo, somente a camada de persistência é modificada;
  - Os objetos da camada de negócio permanecem intactos.
- A diminuição do acoplamento entre os objetos e a estrutura do banco de dados torna o SSOO mais flexível e mais portável.
- No entanto, as vantagens de uma camada de persistência não vêm de graça.
  - A intermediação feita por essa camada entre os objetos do domínio e o SGBD traz uma sobrecarga de processamento.
  - Outra desvantagem é que a camada de persistência pode aumentar a complexidade computacional da realização de certas operações, que seriam triviais com o uso direto de SQL.
- Entretanto, as vantagens adquiridas pela utilização de uma camada de software, principalmente em sistemas complexos, geralmente compensam a perda no desempenho e a dificuldade de implementação.

### Estratégias de persistência

- Há diversas estratégias que podem ser utilizadas para definir a camada de persistência de um SSOO:
  - Acesso direto ao banco de dados
  - Uso de um SGBDOO ou de um SGBDOR
  - Uso do padrão DAO (*Data Access Object*)
  - Uso de um framework ORM

### Acesso direto

- Uma estratégia simples para o mapeamento objeto-relacional é fazer com que cada objeto persistente possua comportamento que permita a sua restauração, atualização ou remoção.
  - Há código escrito em SQL para realizar a inserção, remoção, atualização e consulta das tabelas onde estão armazenados os objetos.
- Essa solução é de fácil implementação em Linguagens de quarta geração, como o Visual Basic, o PowerBuilder e o Delphi.
  - Uso de controles *data aware*.
- Essa estratégia de mapeamento objeto-relacional é justificável para sistemas simples.
- No entanto, a solução de acesso direto apresenta algumas desvantagens para sistemas mais complexos.
  - Classes relativas à lógica do negócio ficam muito acopladas às classes relativas à interface gráfica e ao acesso ao banco de dados.
    - Por vezes, essas classes sequer são criadas.
  - Mais complicado migrar o SSOO de um SGBD para outro.
  - A lógica da aplicação fica desprotegida de eventuais modificações na estrutura do banco de dados.

- A coesão das classes diminui, porque cada classe deve possuir responsabilidades relativas ao armazenamento e materialização de seus objetos, além de ter responsabilidades inerentes ao negócio.
- Dificuldades de manutenção e extensão do código fonte praticamente proíbe a utilização desta estratégia em sistemas complexos.

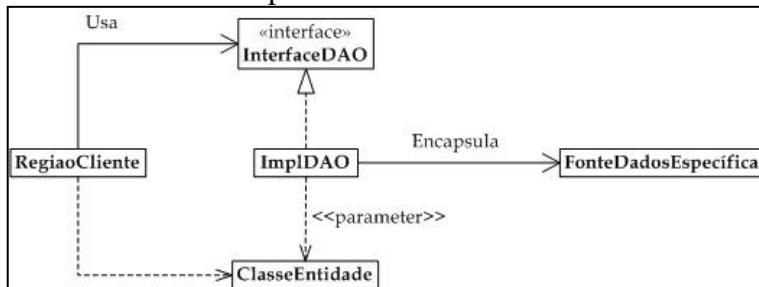
#### Uso de SGBDOO ou SGBDOR

- Na metade dos anos 1980, começou-se a falar em um novo modelo para SGBDs, o orientado a objetos.
- Nesse modelo, em vez de tabelas, os conceitos principais eram classes e objetos.
- No início da década de 1990, foram criados alguns produtos comerciais de **sistemas de gerência de bancos de dados orientados a objetos** (SGBDOO).
  - ORION (MOC), OPENOODB (Texas Instruments), Iris (HP), GEMSTONE (GEMSTONE Systems), ONTOS (Ontos), Objectivity (Objectivity Inc.), ARDENT (ARDENT software), POET (POET Software).
- Um SGBDOO permite a definição de estruturas de dados arbitrariamente complexas (classes) no SGBDOO.
- Nesse modelo, atributos de um objeto podem conter valores de tipos de dados estruturados, diferente do modelo relacional, onde as tabelas só podem armazenar itens atômicos.
- Também é possível definir hierarquias de herança entre classes.
- A linguagem de consulta para SGBDOO, OQL (Object Query Language), permite consultar e manipular objetos armazenados em um banco de dados.
  - Também possui extensões para identidade de objetos, objetos complexos, expressões de caminho, chamada de operações e herança.
- Algumas pessoas pensavam que a tecnologia de SGBDOO suplantaria a velha tecnologia relacional.
- No entanto, os principais SGBDR começaram a incorporar características de orientação a objetos.
- Esses SGBD passaram a adotar o **modelo de dados objeto-relacional**, que é uma extensão do modelo relacional, onde são adicionadas características da orientação a objetos.
- Hoje em dia os principais SGBD são **sistemas de gerência de bancos de dados objeto-relacionais** (SGBDOR).
  - Um SGBDOR é também conhecido por SGBD relacional-estendido.
- Exemplos: Oracle 9iTM e o DB2 Universal ServerTM.
- Os modelos de dados usados por SGBDOR e SGBDOO são mais adequados para realizar o mapeamento de objetos.
- Mas, o fato é que existe uma plataforma imensa de sistemas que usam o modelo relacional puro.
  - De fato, existe uma grande resistência em substituir esses sistemas.
- Isso leva a crer que o mapeamento de objetos para o modelo relacional ainda irá durar por muitos anos.

#### Uso do padrão DAO

- O padrão DAO é uma forma de desacoplar as classes do negócio dos aspectos relativos ao acesso ao armazenamento persistente.
  - DAO: *Data Access Object* (Objeto de Acesso a Dados).

- Nessa estratégia, um SSOO obtém acesso a objetos de negócios através de uma interface, a chamada ***interface DAO***.
  - Classes que implementam essa interface transformam informações provenientes do mecanismo de armazenamento em objetos de negócio, e vice-versa.
- O SSOO interage com o ***objeto DAO*** através de uma interface.
  - A implementação desse objeto simplesmente não faz diferença para a aplicação.
  - O objeto DAO isola completamente os seus clientes das particularidades do mecanismo de armazenamento (fonte de dados) sendo utilizado.
- Estrutura do padrão DAO



- Exemplo em linguagem Java de uma InterfaceDAO

```

public interface AlunoDAO
{
    public void inserir(Aluno aluno)
    throws AlunoDAOException;

    public void atualizar(Aluno aluno)
    throws AlunoDAOException;

    public void remover(Aluno aluno)
    throws AlunoDAOException;

    public List<Aluno> encontrarTodos() throws AlunoDAOException;

    public Aluno encontrarPorMatricula(String matricula) throws
    AlunoDAOException;

    public List<Aluno> encontrarPorTurma(int idTurma) throws AlunoDAOException;

    public HistoricoEscolar obterHistoricoEscolar() throws AlunoDAOException;
}
  
```

## Uso de um framework ORM

- Um framework ORM é um conjunto de classes que realiza o mapeamento objeto-relacional de forma transparente.
  - ORM: *Object-Relational Mapping* (mapeamento objeto-relacional).
- Os frameworks ORM tentam resolver o problema do mapeamento objeto relacional através de classes que o realizam de forma transparente.
- Normalmente, um framework ORM demanda a definição da correspondência entre a estrutura de objetos da aplicação e o esquema relacional do banco de dados.
  - Essa correspondência é fornecida através de um arquivo de configuração, denominado *arquivo de mapeamento*.

- De posse dessa correspondência, o framework está apto a mapear qualquer requisição por uma informação armazenada no SGBDR.