

Ciência da Computação  
Engenharia da Computação

# Projeto de Software Orientado a Objeto

UML ...

Prof. Dr. Fábio Fagundes Silveira

[fsilveira@unifesp.br](mailto:fsilveira@unifesp.br)

# Créditos

- Adaptação dos slides do livro “Engenharia de Software Moderna”, do Prof. Dr. Marco Túlio Valente, da UFMG.

# UML as Sketch

Prof. Marco Túlio Valente

[mtov@dcc.ufmg.br](mailto:mtov@dcc.ufmg.br)

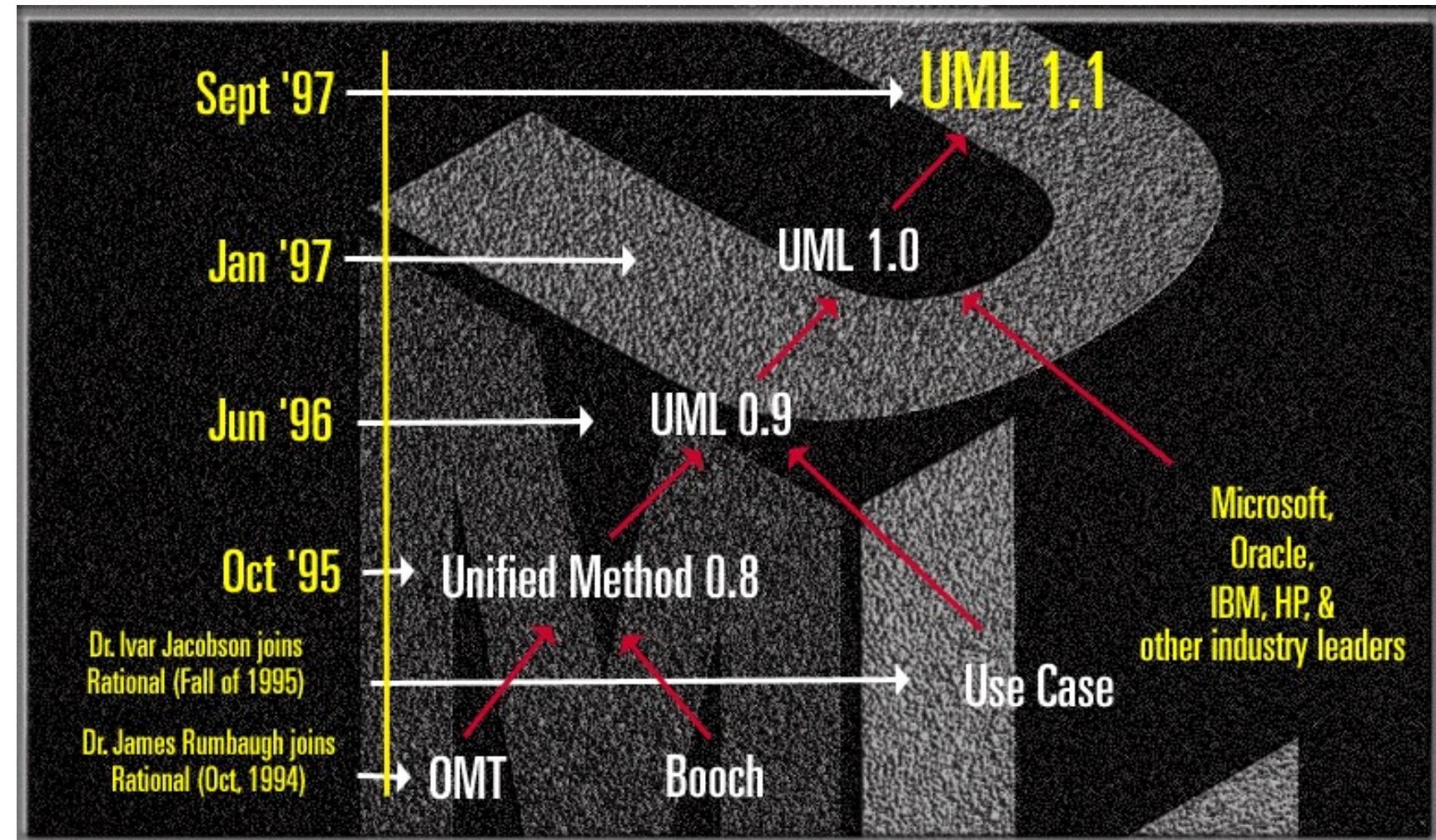
# Unified Modeling Language (UML)

- Linguagem ou notação gráfica (na prática, um conjunto de diagramas) para documentação e projeto de sistemas, notadamente sistemas OO
- Origem nas décadas de 80-90: auge dos princípios de OO; diversas linguagens de programação OO e diversas notações gráficas surgindo
- UML: esforço para unificar essas diversas notações
- Especificamente, aquelas propostas por Grady Booch (Rational, hoje IBM), Jim Rumbaugh e Ivar Jacobson
- Em seguida, passou a ser um padrão OMG (Object Management Group), que é uma organização de padronização, na área de software

# Versões de UML

- Diversas versões: UML 1.x, UML 2.x
- Padronização é importante para inter-operação entre ferramentas CASE
- Por exemplo; para permitir que um diagrama criado em uma ferramenta X possa ser aberto e editado em uma ferramenta Y, de uma empresa diferente
- Para ser mais preciso, a OMG padroniza um meta-modelo UML, isto é, um modelo que descreve as "regras" que definem cada um dos diversos modelos UML

# História e Padronização





# UML pode ser usada de 3 formas

- UML as Sketch
- UML as Blueprint
- UML as Programming Language

# UML as Sketch

- Neste modo, UML é usada para comunicar (i.e., para conversar sobre) alguma parte de um sistema
  - Discutir alguma parte do projeto
  - Discutir alternativas de projeto
- Isto é, UML como uma ferramenta para comunicação entre desenvolvedores
- Portanto, trata-se de um uso mais informal e "dinâmico" dos diagramas UML
- Esse diálogo pode acontecer para discutir:
  - A implementação de um novo código (forward-engineering)
  - A implementação de um código já existente (backward-engineering)

# UML as Blueprint

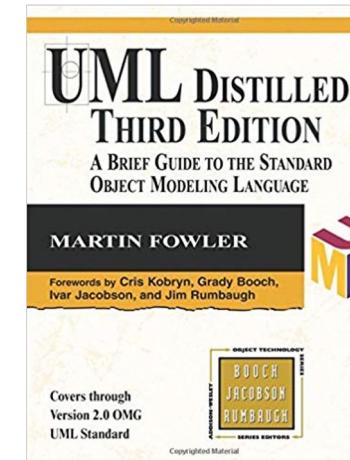
- Neste modo, UML é usada para documentar o projeto de um sistema
- Forward-engineering (comum em Waterfall, RUP e similares):
  - "Projetista" (ou analista de sistemas) documenta o projeto em UML, provavelmente usando uma ferramenta CASE
  - Documentação pode ser apenas no nível de interface
  - Programador implementa (isto é, de fato escreve o código)
- Reverse-engineering:
  - Objetivo é extrair um modelo UML completo, de um código existente
  - Por exemplo, para permitir o entendimento e manutenção desse código

# UML as Programming Language

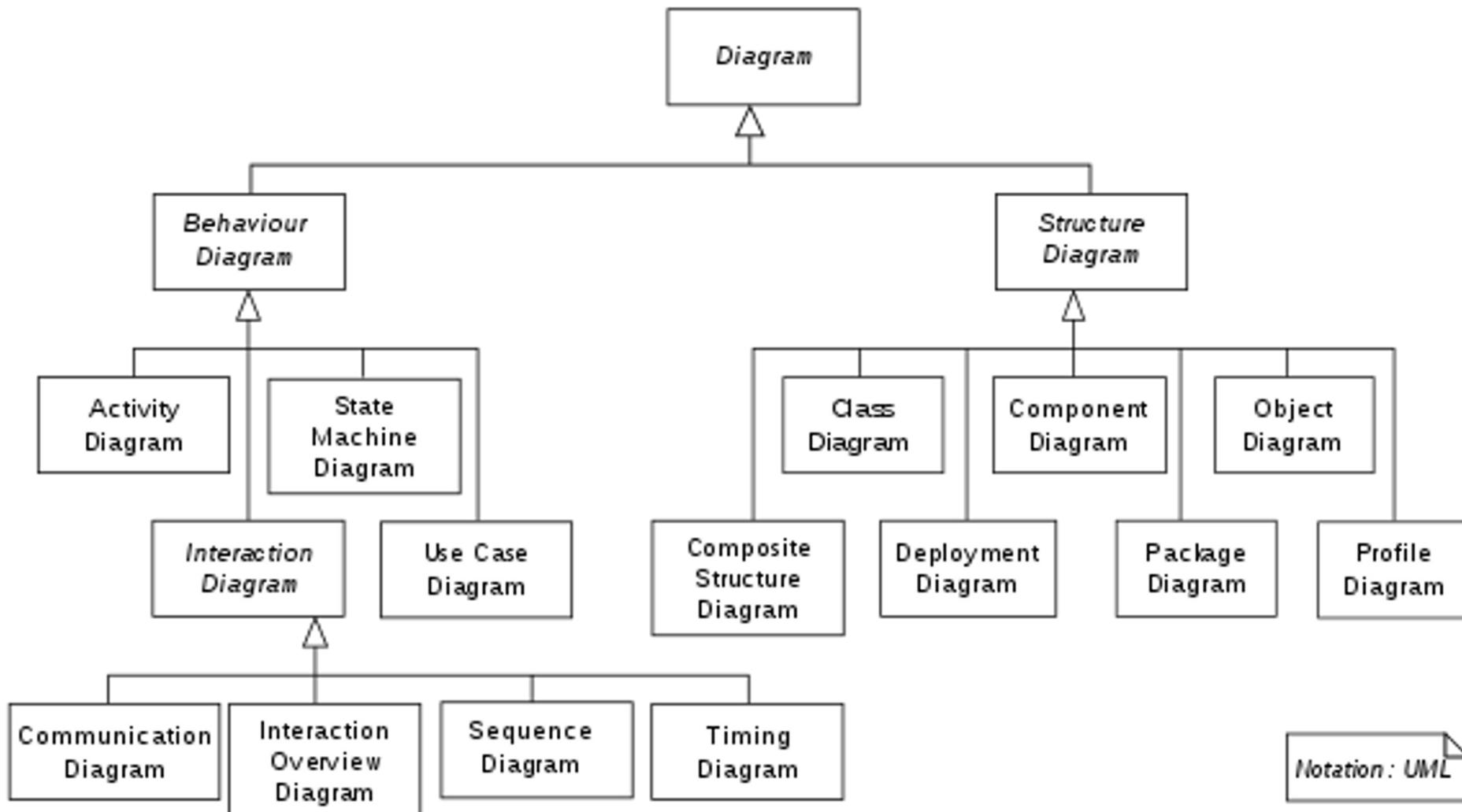
- UML como linguagem para -- de fato -- desenvolver sistemas
- Em outras palavras, geração de código automática a partir de modelos UML
  - Código completo e não apenas "esqueleto" das classes
- Conhecido como MDE/MDA (Model-Driven Engineering/Architecture)
- Não é uma "bala-de-prata", pois requer tomar um dos seguintes caminhos:
  - "Sofisticar" os modelos UML para aumentar seu poder de expressão (logo, modelos podem ficar mais complicados do que o próprio código)
  - Focar em um domínio bastante específico (exemplo: programação de um robô com tarefas bem definidas e limitadas)

# Neste curso: UML as Sketch

- Provavelmente, trata-se do uso mais comum de UML, atualmente
- Usando o seguinte livro:
  - UML Distilled: A Brief Guide to the Standard Object Modeling Language.  
Martin Fowler, 3rd Edition, 2003



# Diagramas UML (13)



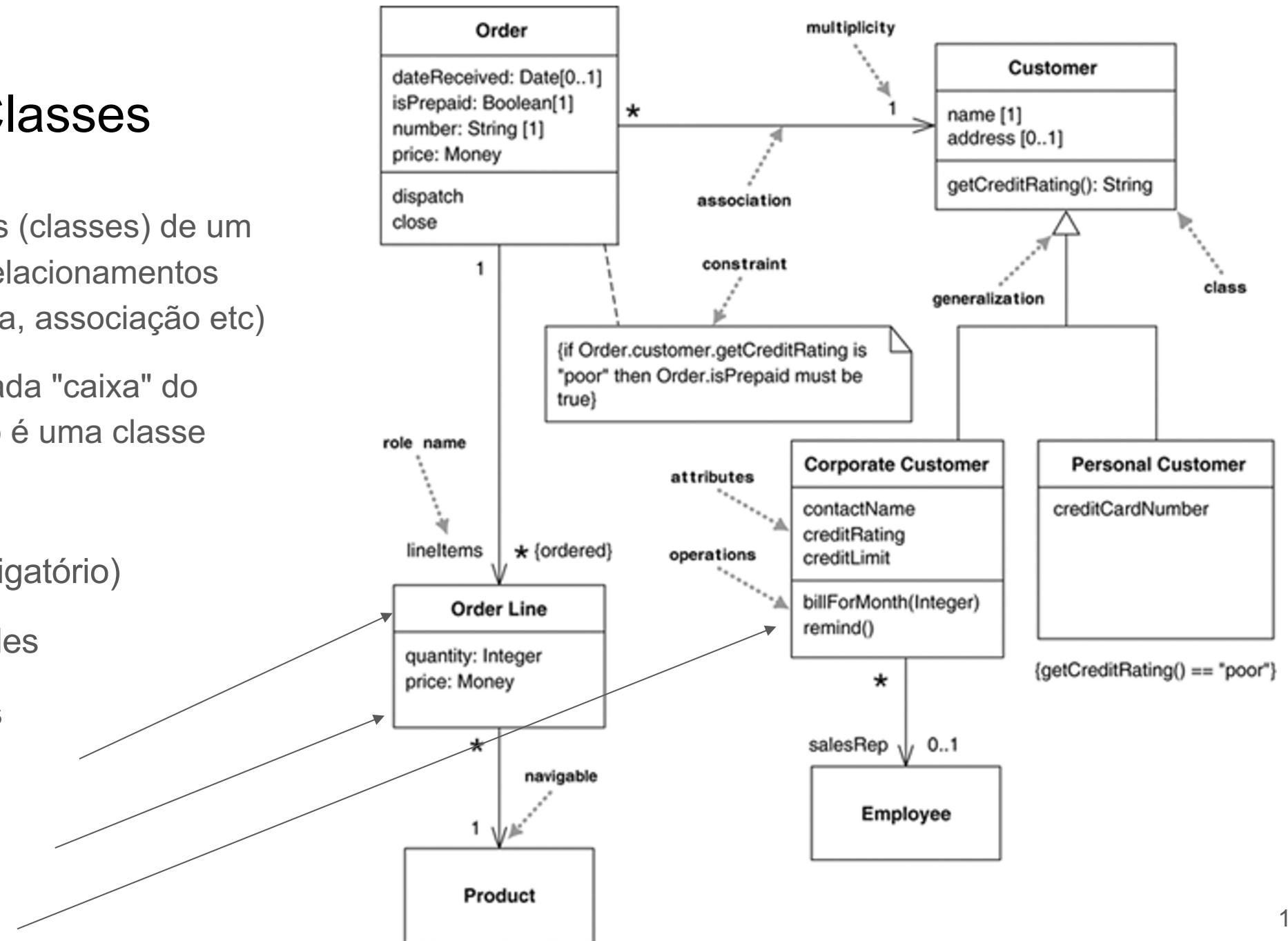
# Vamos Estudar os seguintes Diagramas

1. Classes (E)
2. Sequência (C)
3. Pacotes (E)
4. Deployment (E)
5. Casos de Uso (C)
6. Atividades (C)

# Diagrama de Classes

# Diagrama de Classes

- Descreve os tipos (classes) de um sistema e seus relacionamentos estáticos (herança, associação etc)
- Para começar: cada "caixa" do diagrama ao lado é uma classe
- Cada caixa tem:
  - Nome (obrigatório)
  - Propriedades
  - Operações



# Propriedades

- São os dados (ou os campos) de uma classe
- Podem ser:
  - Atributos
  - Associações

# Atributos

- Sintaxe e um exemplo:

```
visibility name: type multiplicity = default {property-string}
```

An example of this is:

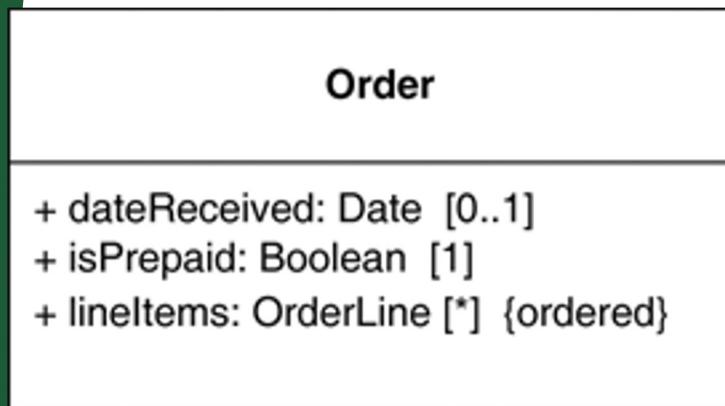
```
- name: String [1] = "Untitled" {readOnly}
```

# Sintaxe de Atributos

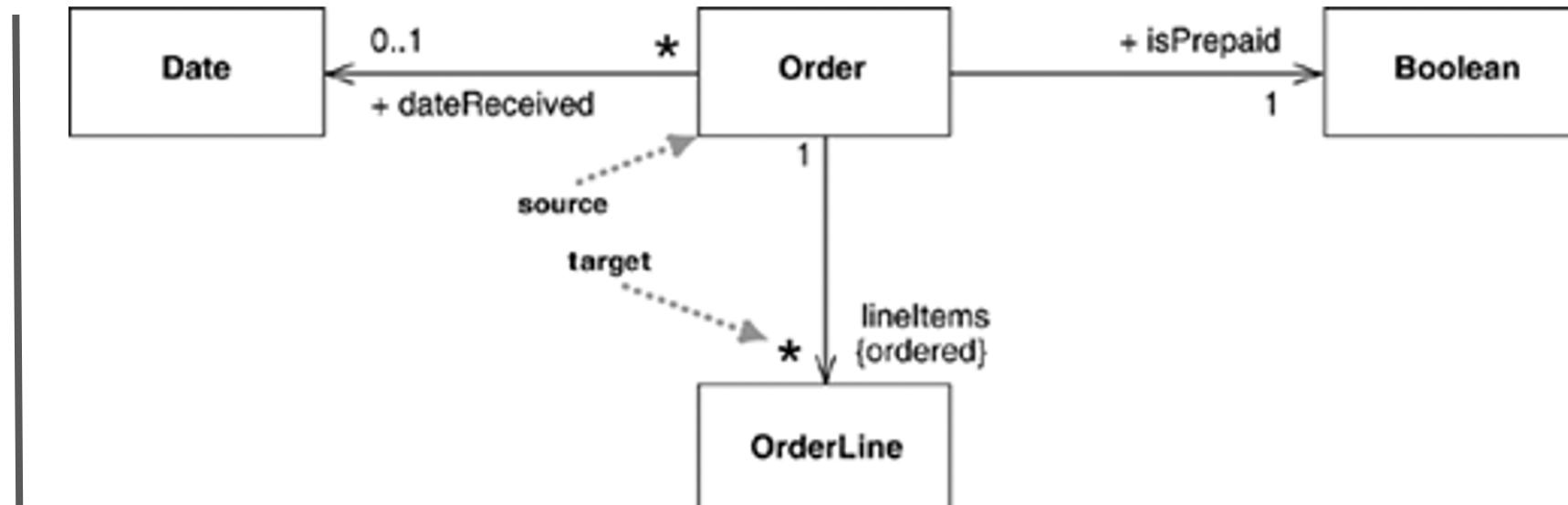
- Visibilidade: público (+) ou privado (-)
- Nome
- Tipo
- Multiplicidade (vamos explicar em breve)
- Default value
- Property-String: {read-only}, {unique}, {ordered} etc

# Associações

- Modo alternativo para definição de propriedades, que torna mais visível os relacionamentos com outras classes; as duas notações abaixo (para a classe Order) são equivalentes, mas a segunda é baseada em associações



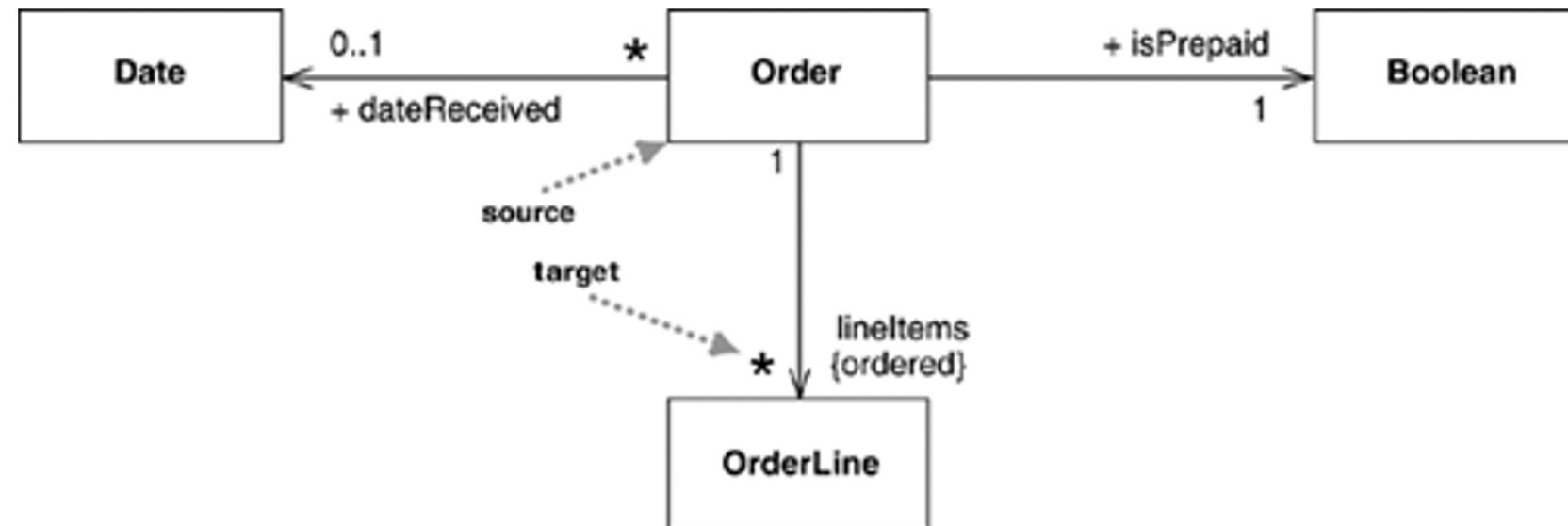
Order tem 3 propriedades, definidas por meio de atributos



Order tem as mesmas três propriedades (dateReceived, isPrepaid e lineItems); porém, definidas por meio de associações (setas que explicitam os relacionamentos de Order com outras classes do modelo)

# Associações: Sentido da Seta

- Seta (de uma associação) sai da classe que tem a propriedade
- E chega na classe que é o tipo da propriedade
- Nome da associação (propriedade) é escrito na "ponta" da seta

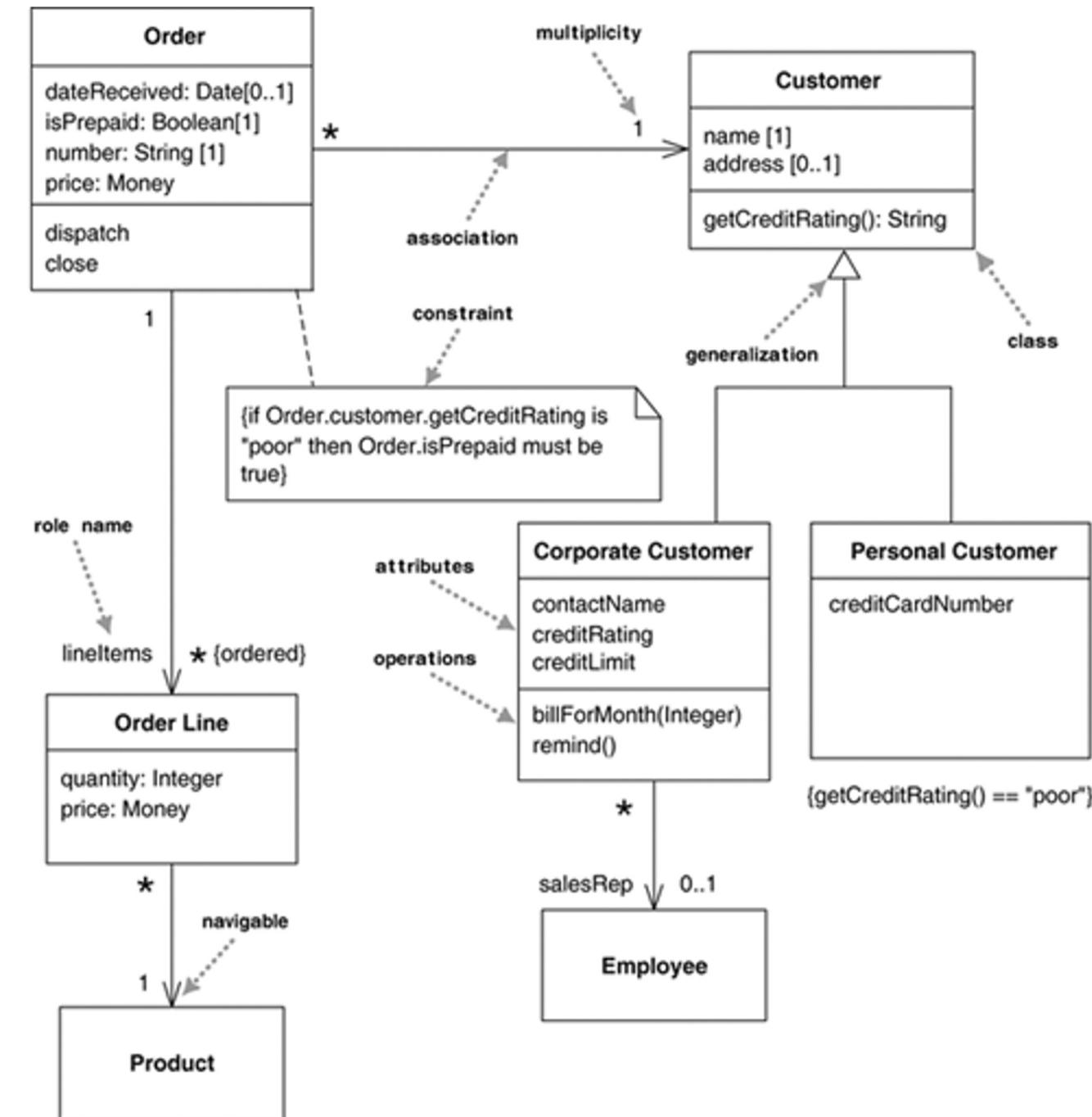


# Associações vs Atributos

- Quando usar associações? E quando usar atributos?
- Regra geral:
  - Associações são mais usadas para relacionamentos importantes; com outras classes do sistema
  - Atributos são usados para relacionamentos com tipos primitivos

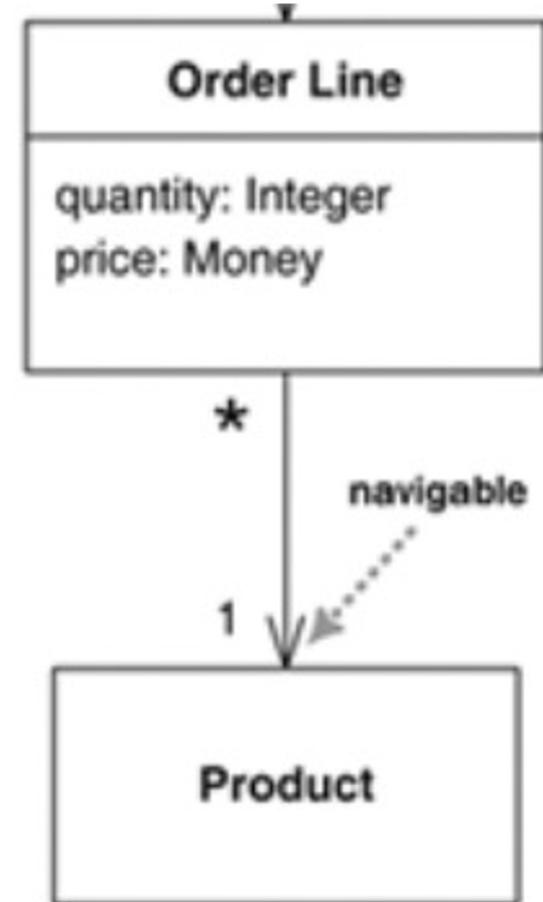
# Multiplicidade

- Pode ser: 1, 0..1, \*
- Exemplos:
  - Toda Order tem exatamente 1 Customer
  - Um CorporateCustomer pode ter 0 ou 1 salesRep (que é um Employee)
  - Uma Order pode ter vários (\*) linelItems, que são do tipo OrderLine
  - Alternativamente, pode-se usar algo assim:  
2..4

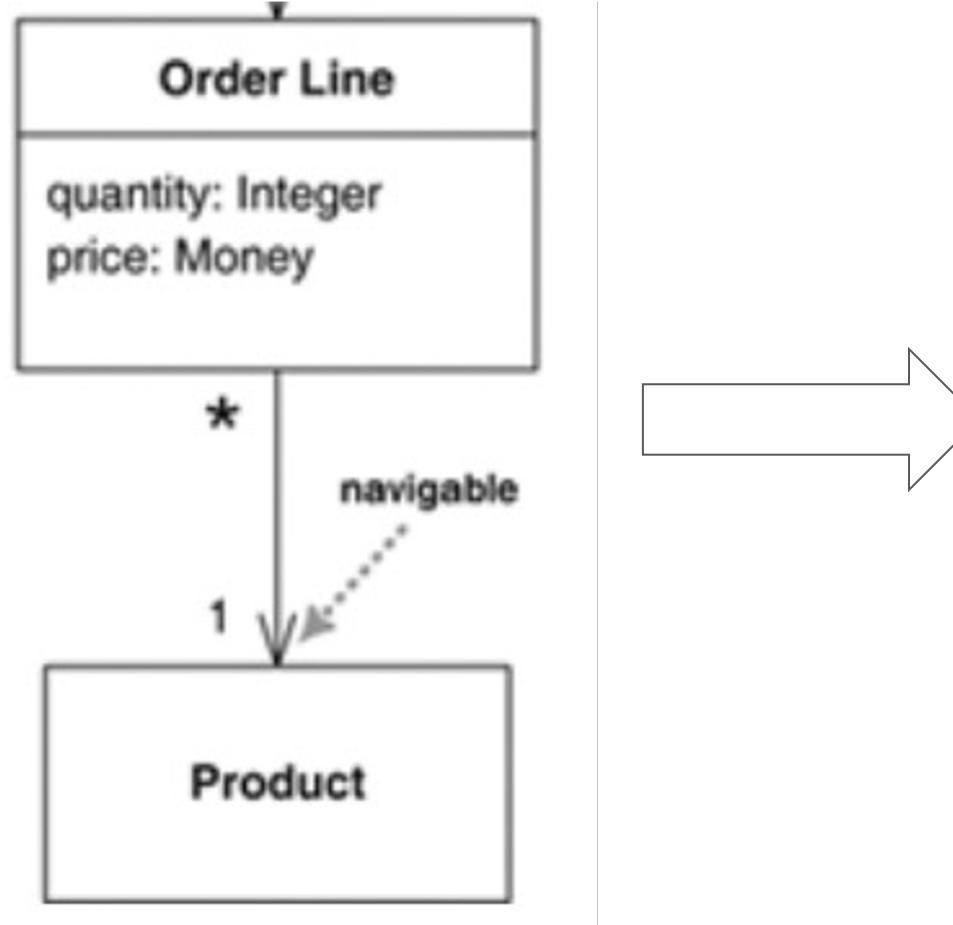


# Navegabilidade

- Neste exemplo, a classe OrderLine possui uma propriedade que permite acessar ("navegar para") o Produto referenciado nesta linha do pedido; além disso, essa propriedade referencia exatamente um Product.
- Por sua vez, um Product pode estar associado a vários objetos OrderLine; porém de Product não se consegue "navegar para" OrderLine. Isto é, um Product não possui uma propriedade que permita acessar as OrderLine onde ele aparece; ainda de outra forma, Product não conhece as OrderLine onde ele aparece.

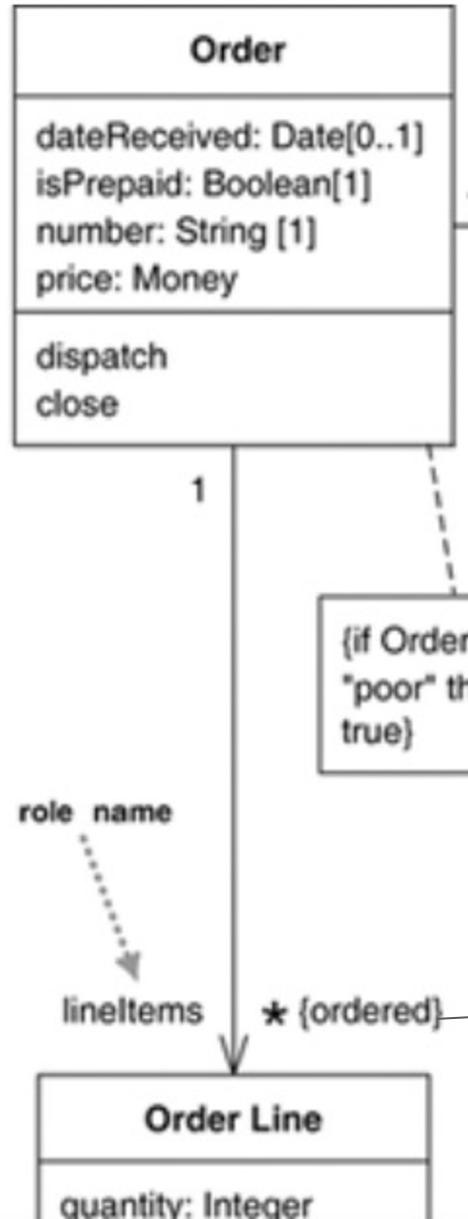


# Convertendo para Código (Java): OrderLine



```
public class Orderline {  
    int quantity;  
    Money price;  
    Product product;  
    .....  
}
```

# Convertendo para Código (Java)

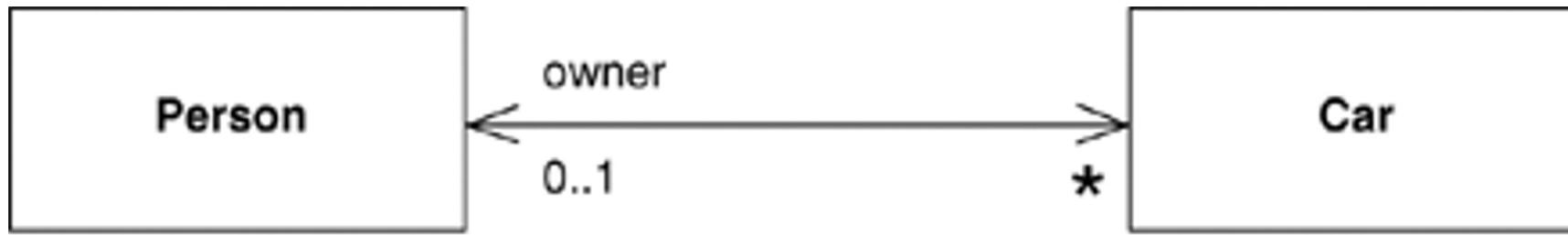


```
public class Order {  
    ....  
    ArrayList<OrderLine> lineltems;  
    ....  
}
```

Essa anotação (ordered) significa que a ordem dos elementos armazenados pela propriedade é relevante (isto é, deve-se usar, por exemplo, uma Lista (ArrayList) e não um Set).

# Associações Bidirecionais

- Sintaxe:



- Semântica:
  - Car tem no máximo 1 owner, que é uma Person
  - Uma Person pode ter vários carros (inclusive zero); nome da propriedade não é indicado no modelo (pois ele não importa, para fins da modelagem em questão ...)

# Operações (ou Métodos)

- Sintaxe:

```
visibility name (parameter-list) : return-type {property-string}
```

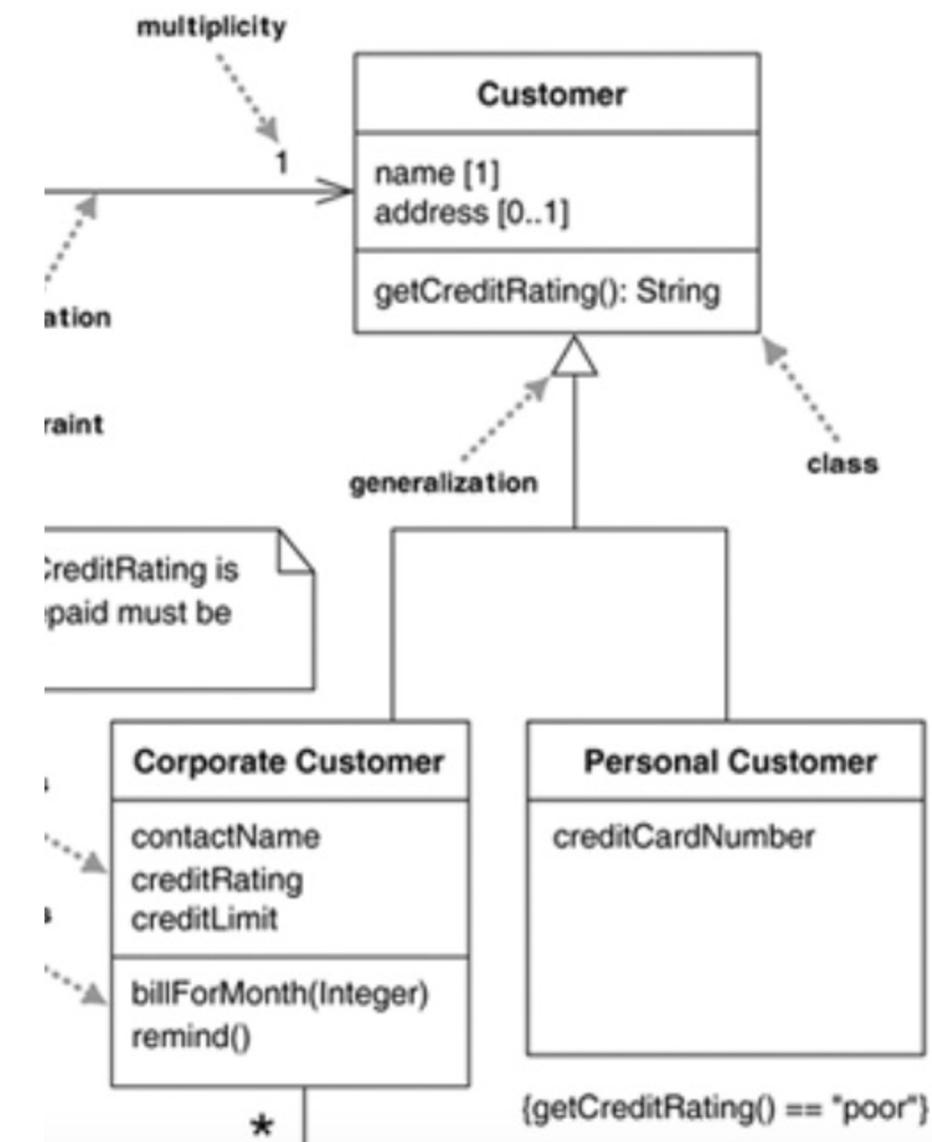
- Exemplo:

```
+ balanceOn (date: Date) : Money
```

- Exemplo de uma possível {property-string} para operações: {query} (indica que o método não altera o estado de um objeto; ele é read-only)

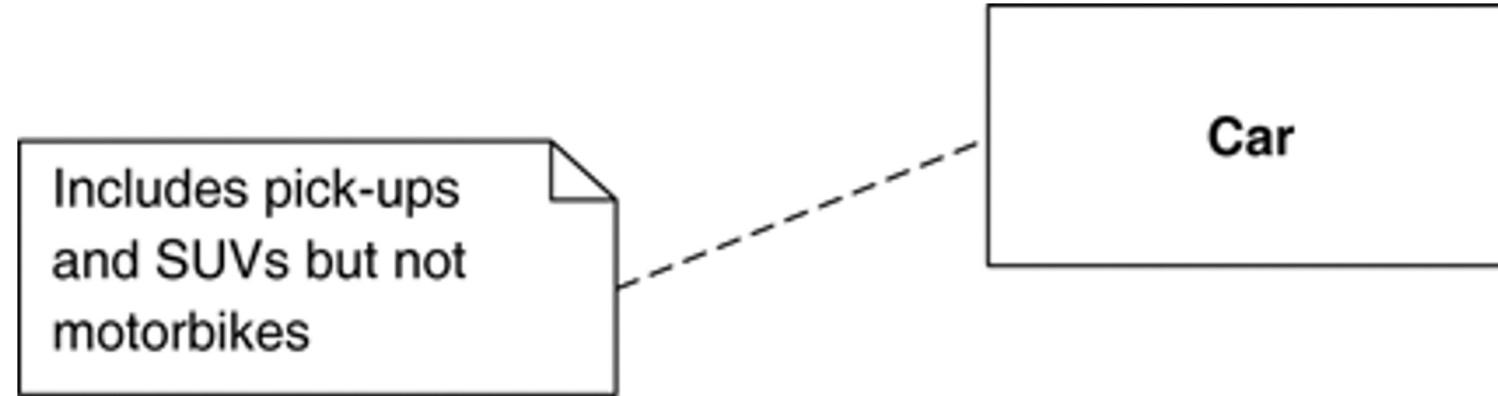
# Generalização (Herança)

- Corporate Customer e Personal Customer são subclasses de Customer



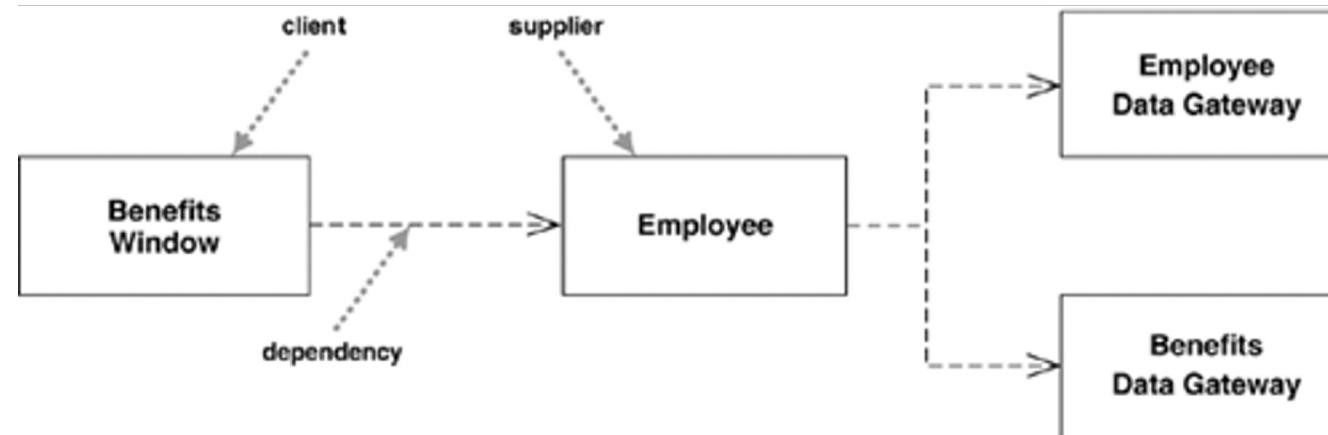
# Notas e Comentários

- Podem ser associados a qualquer parte de um diagrama
- São ligados a essa parte por uma linha tracejada



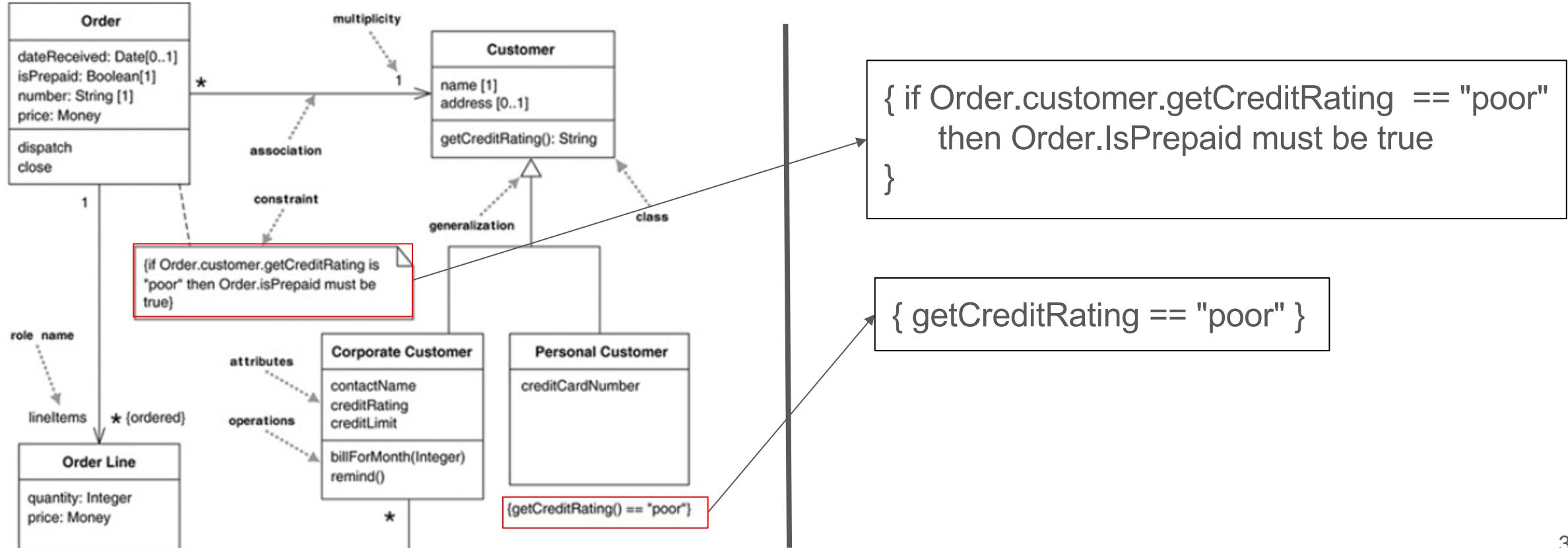
# Dependências

- Relacionamento usado para indicar que uma classe A usa uma classe B.
- Exemplos de uso: A chama um método de B; A declara um parâmetro ou variável local do tipo B; A instancia um objeto do tipo B; A implementa B
- São indicadas por uma seta tracejada (da classe A para a classe B)
- Observação: associação é um caso mais "forte" de dependência.



# Restrições

- Predicados que devem ser válidos em instâncias das classes de um Diagrama de Classes; normalmente, escritos em notação informal



# Restrições

- Restrições, portanto, acrescentam semântica a um Diagrama de Classes
- UML também define uma linguagem formal, chamada OCL (Object Constraint Language), baseada em lógica de predicados, para escrita de constraints
- No entanto, o estudo de OCL está fora do escopo deste curso
- Restrições OCL são mais importantes quando almeja-se usar UML como uma linguagem de programação; para geração de código automática etc.

# Como e Quando usar Diagramas de Classes

- (Sugestões retiradas do Livro do Fowler; isto é, referem-se ao uso de UML em projetos ágeis)
- Muitas vezes, um diagrama de classes pode ser útil para documentar o "vocabulário" principal de um sistema. Exemplos: Pedidos, Clientes, Empregados, Produtos etc
- Podem ser usados também para documentar as partes mais importantes e críticas de um sistema. Assim não corra o risco de gerar diagramas completos, para tudo, mas que rapidamente podem ficar obsoletos.

# Exercício: ENADE

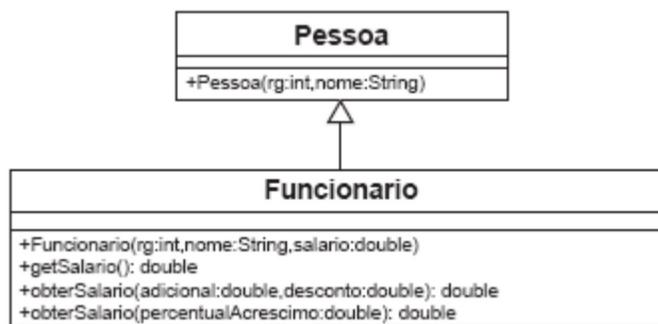
- Se um projeto não tem a documentação apropriada ou se está com a documentação desatualizada, uma opção é a engenharia reversa que possibilita mapear códigos para diagramas UML.
- A seguir, é apresentado um código na linguagem de programação JAVA (próximo slide).
- Utilizando a engenharia reversa nesse trecho de código, o diagrama UML de classes correspondente é (próximo slide).

```

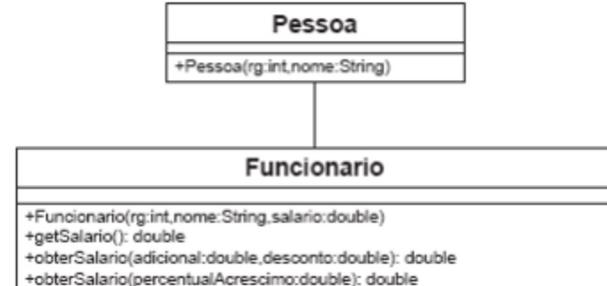
1 package default;
2
3 public class Funcionario extends Pessoa {
4     private double salario;
5
6     public Funcionario(int rg, String nome,
7                         double salario) {
8         super(rg, nome);
9         this.salario = salario;
10    }
11    public double getSalario() {
12        return salario;
13    }
14
15    public double obterSalario(double
16                                percentualAcrescimo) {
17        double salarioReajustado = salario +
18            salario * percentualAcrescimo / 100;
19        return salarioReajustado;
20    }
21
22    public double obterSalario(double
23                                adicional, double desconto){
24        return this.getSalario() + adicional - desconto;
25    }
26}
27

```

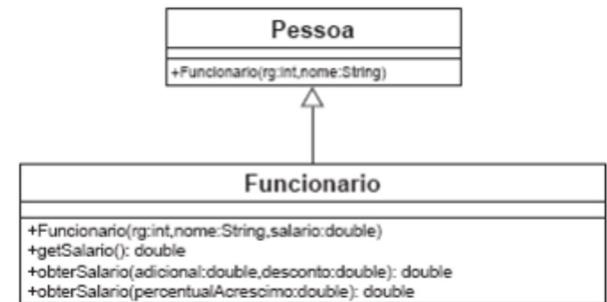
A



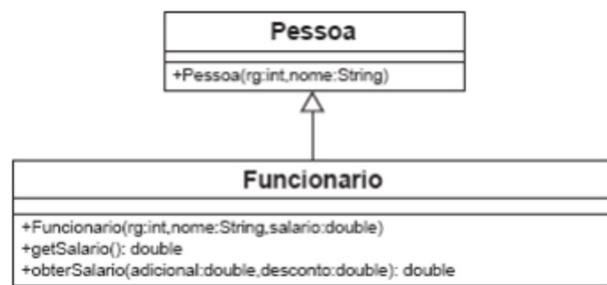
B



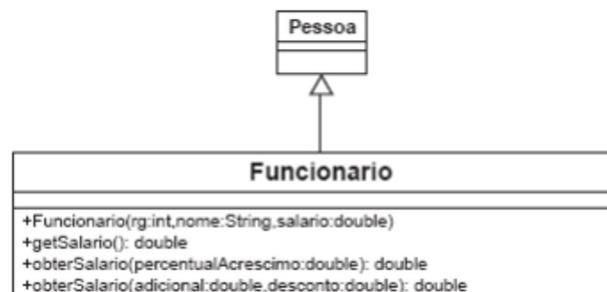
C



D



E

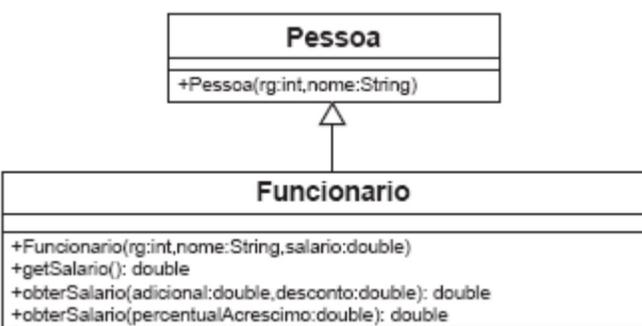


```

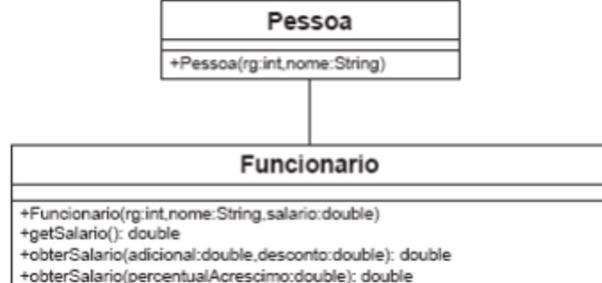
1 package default;
2
3 public class Funcionario extends Pessoa {
4     private double salario;
5
6     public Funcionario(int rg, String nome,
7                         double salario) {
8         super(rg, nome);
9         this.salario = salario;
10    }
11    public double getSalario() {
12        return salario;
13    }
14    public double obterSalario(double
15                                percentualAcrescimo) {
16        double salarioReajustado = salario +
17            salario * percentualAcrescimo / 100;
18        return salarioReajustado;
19    }
20    public double obterSalario(double
21                                adicional, double desconto){
22        return this.getSalario() + adicional - desconto;
23    }
24    }
25    }
26    }
27 }

```

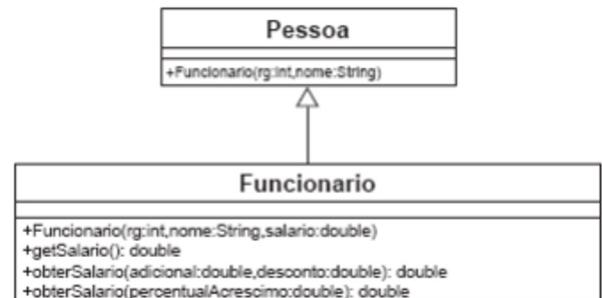
X



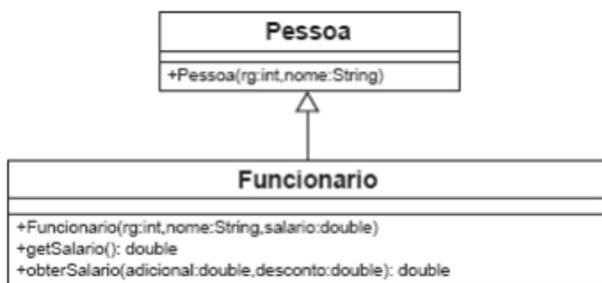
B



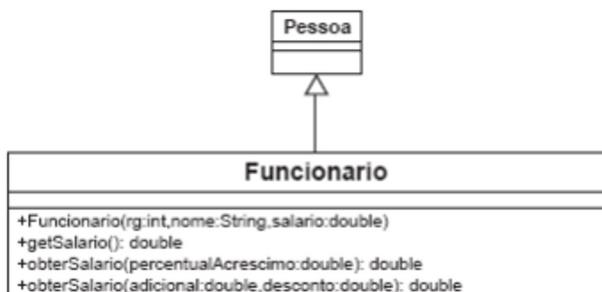
C



D



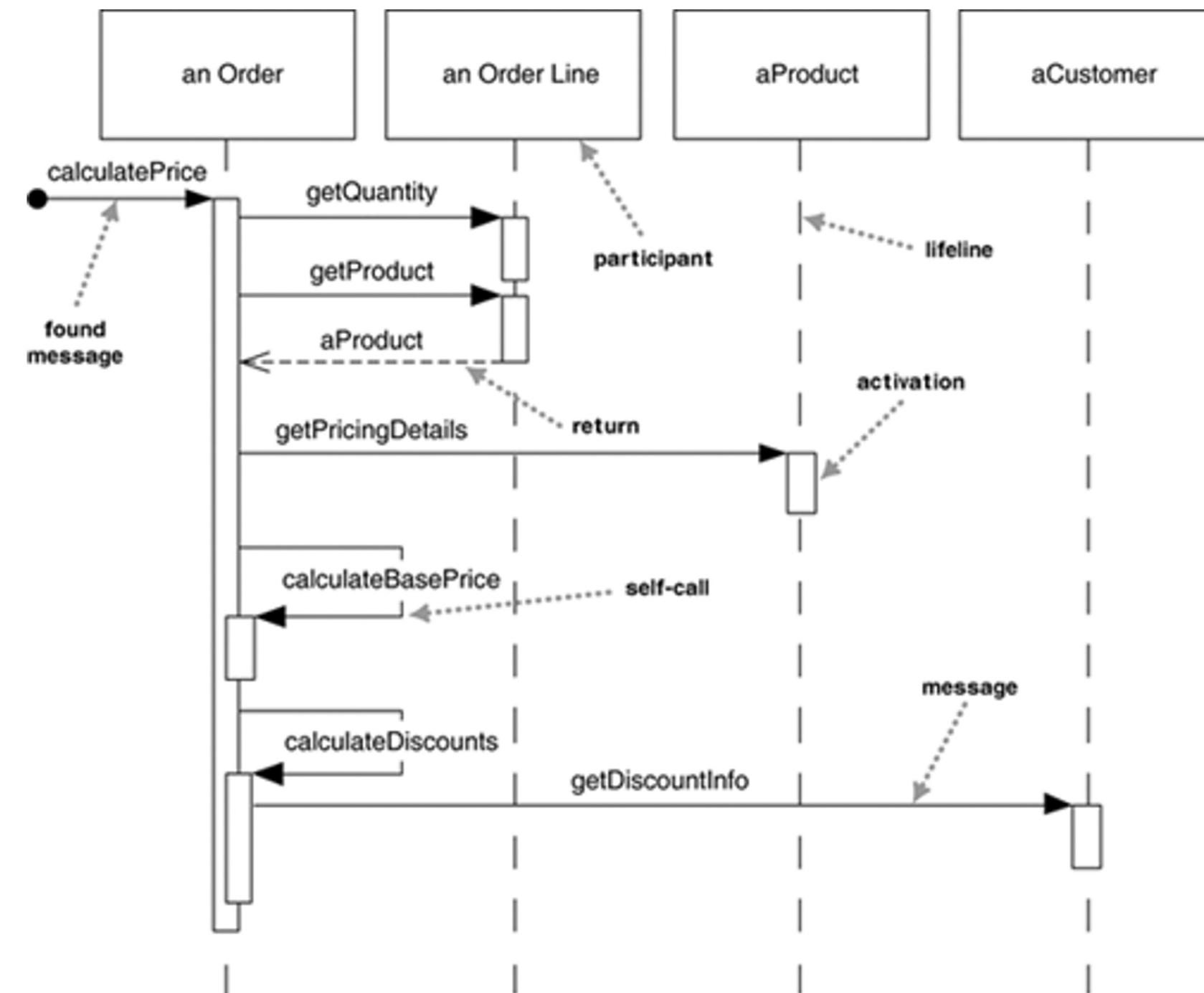
E



# **Diagramas de Sequência**

# Diagrama de Sequência

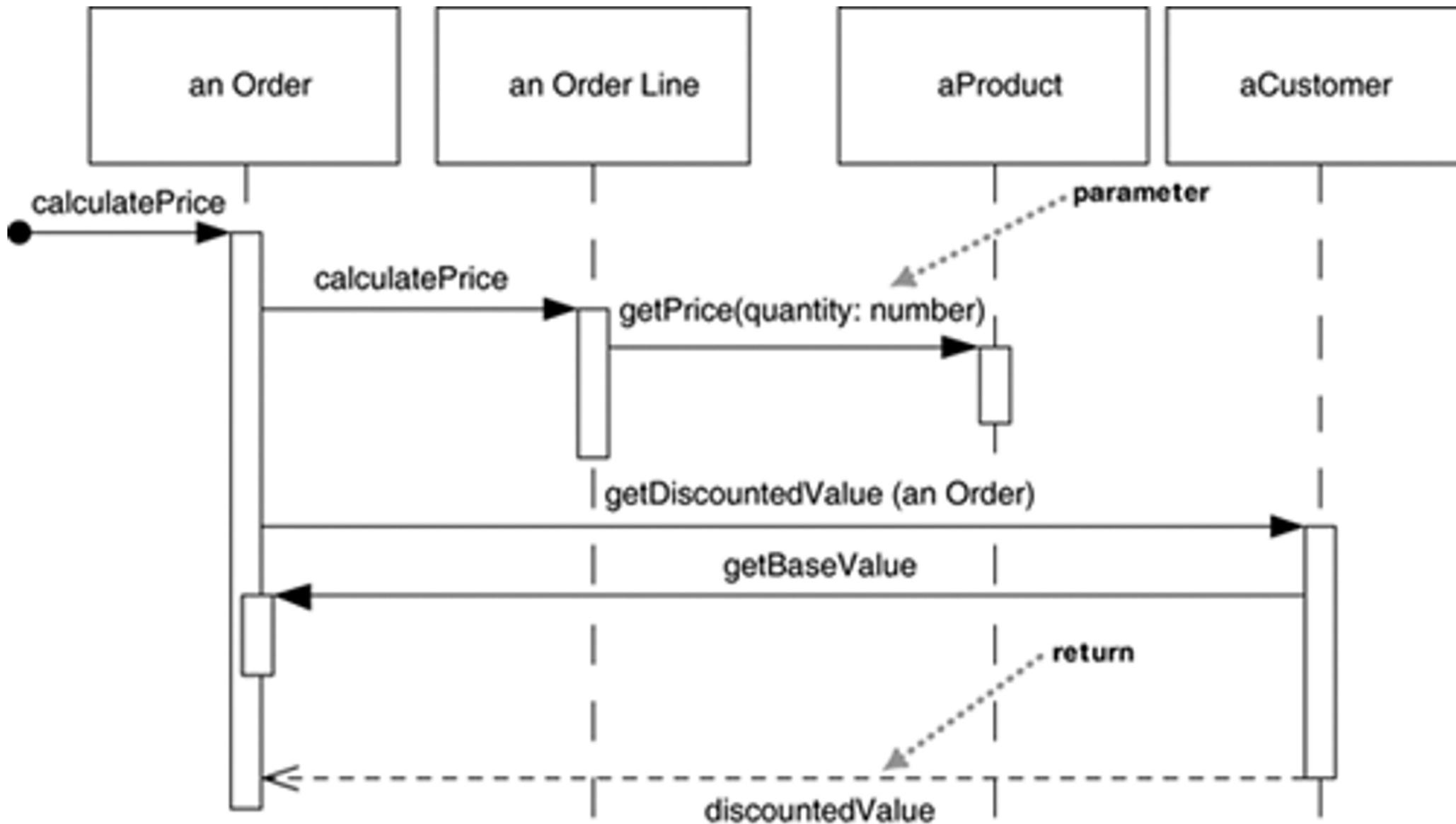
- Descreve como objetos colaboram para realizar uma tarefa
- Mostra alguns objetos e as mensagens que eles trocam (mensagens = chamadas de métodos)
- No exemplo ao lado, a tarefa é calcular o valor de uma Order



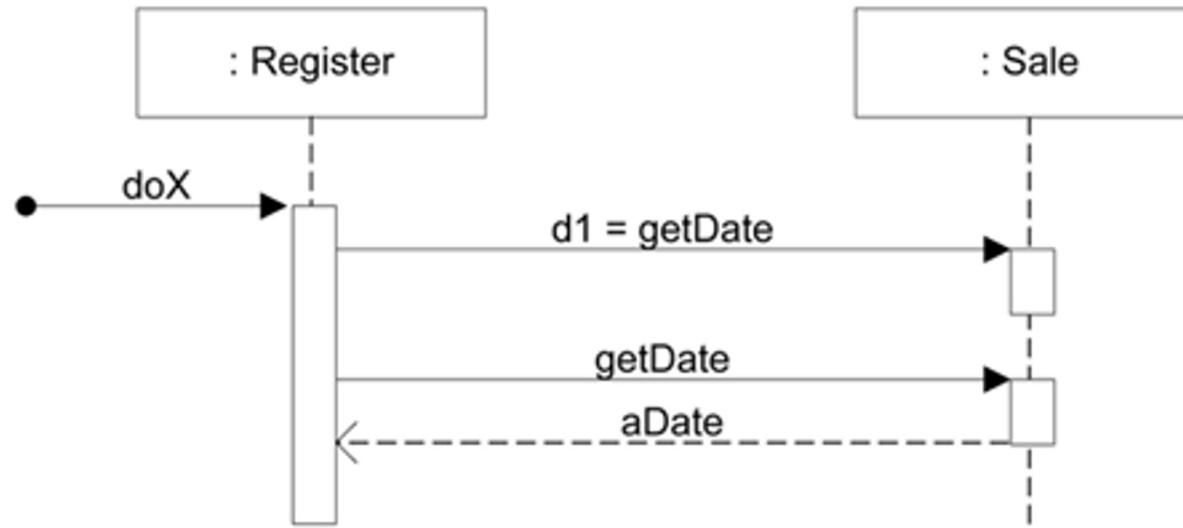
# Diagramas de Sequência

- Importante, linhas verticais o andamento do tempo.
- Na vertical, a barra mais "gorda" significa que o objeto está executando um método; ou que algum método está com sua execução ainda em andamento
- Na vertical, a barra tracejada significa que o objeto está "inativo"; não está processando nenhuma chamada de método, naquele momento
- Na horizontal, setas contínuas indicam o início de uma chamada de método; um objeto está chamando um método, de outro objeto
- Na horizontal, setas pontilhadas indicam retorno de métodos; às vezes, o valor retornado (a variável com o resultado) é indicada, em cima da seta
- Por outro lado, a seta de retorno às vezes é omitida, para não poluir o diagrama; já que o retorno de alguns métodos não é importante no modelo

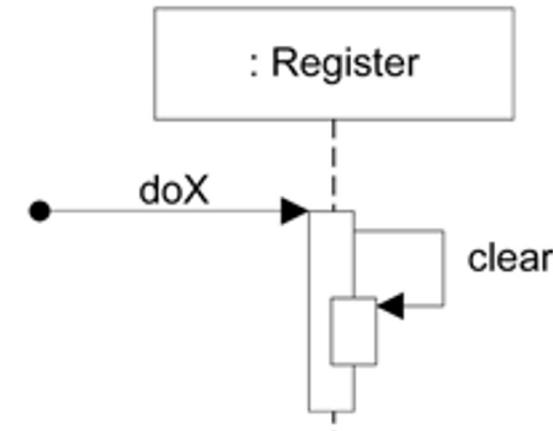
# Solução alternativa para mesma tarefa anterior



# Mais alguns exemplos: return e chamadas self



Mais dois exemplos de sintaxes usadas para retorno



Exemplo de chamadas usando "this"  
(ou "self"); "doX" chama "clear" (do mesmo objeto)

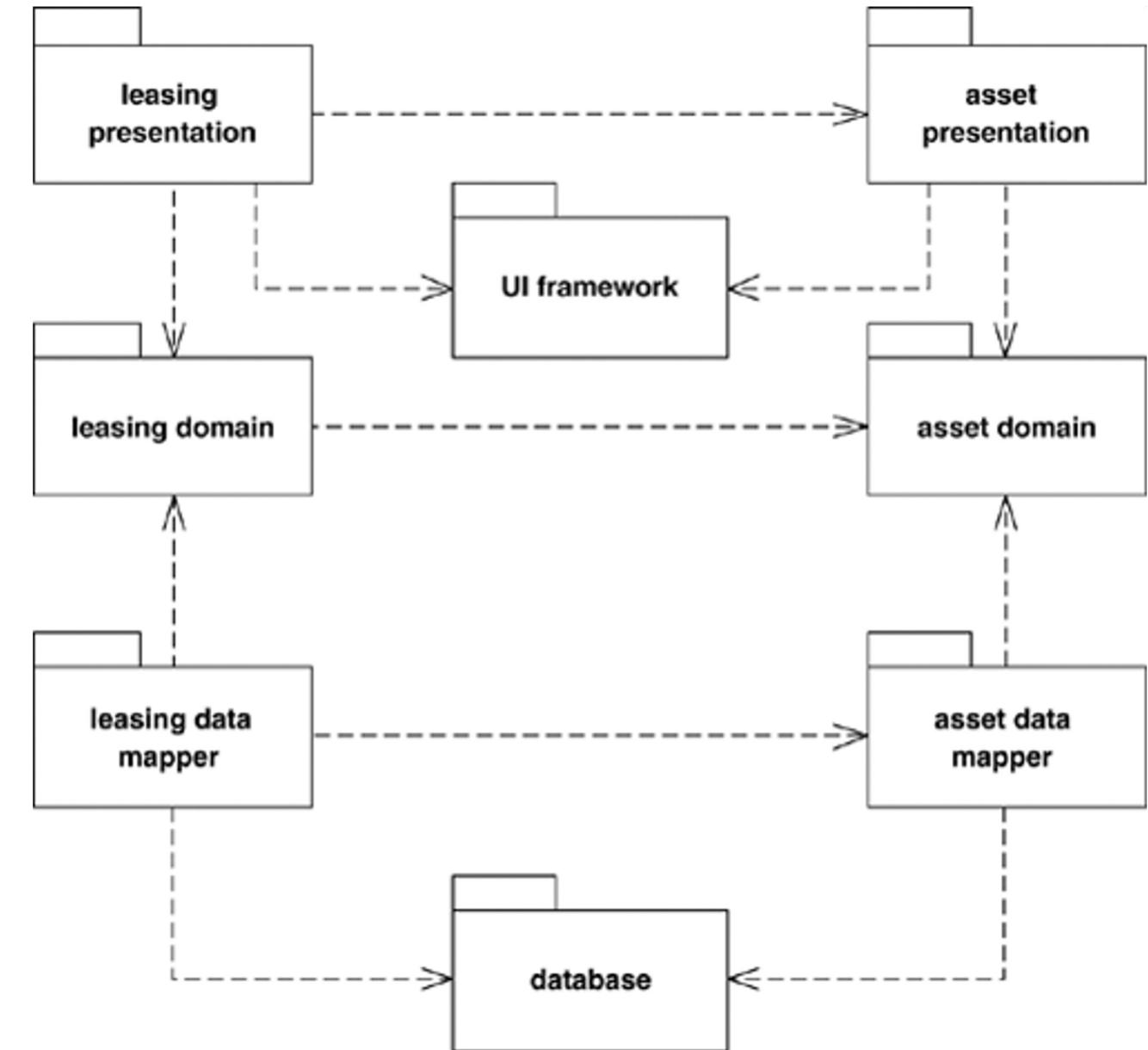
# Como e Quando usar Diagramas de Sequência?

- Quando se quer ter uma "primeira ideia" da implementação de uma determinada tarefa, envolvendo um grupo de objetos
- "Primeira ideia" pois diagrama de sequências não são bons para definir precisamente um determinado comportamento
- Alguns comportamentos são mais complexos, como loops, ifs etc. Embora existam alguns operadores para isso, eles são pouco usados na prática e não serão abordados neste curso

# Diagramas de Pacotes

# Diagrama de Pacotes

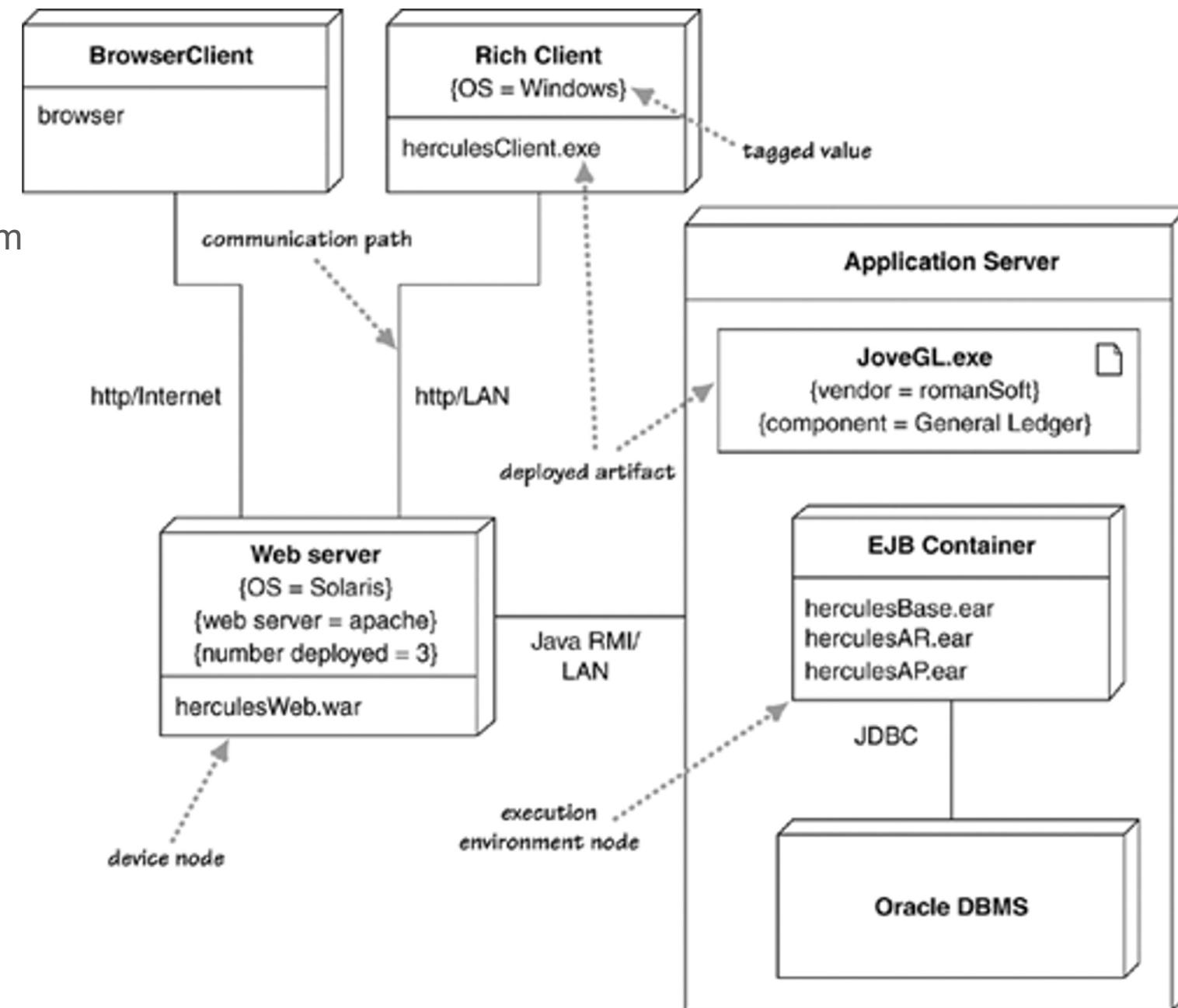
- Mostra os pacotes principais de um sistema
- Pacotes, ou módulos, ou subsistemas, ou componentes, etc (isto é, um grupo de classes)
- Dependências tal como definido para diagramas de classes



# Diagrams de Deployment (ou de Instalação)

# Diagrama de Deployment

- Representam a arquitetura física de um sistema; mostram "o que" (qual software) roda "aonde" (em qual máquina)
- Nodos são as máquinas que rodam algum software
- Linhas indicam como os nodos se comunicam
- Nodos podem ter propriedades.  
Exemplo: { OS = Solaris }



# **Diagramas de Casos de Uso (Use Cases)**

# Casos de Uso

- Antes de explicar o diagrama UML, vamos explicar o que é um **caso de uso**
- Caso de Uso: descrição de **cenários** de uso de um sistema, envolvendo casos de sucesso e de fracasso; esses cenários descrevem um **ator** usando um sistema, como meio para atingir um objetivo
- **Cenário:** uma sequência específica de ações entre atores e um sistema; é um documento que descreve de forma narrativa um ator usando um sistema
- **Autor:** usuário de um sistema (humano, outro sistema, uma máquina etc)
- Importante: caso de uso é um documento (para especificação de requisitos de um sistema) e não um diagrama; podem ser escritos pelos próprios usuários (ou com grande participação deles)

# Exemplo de Caso de Uso

## Buy a Product

Goal Level: Sea Level

Main Success Scenario:

1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming e-mail to customer

Extensions:

3a: Customer is regular customer

- .1: System displays current shipping, pricing, and billing information
- .2: Customer may accept or override these defaults, returns to MSS at step 6

6a: System fails to authorize credit purchase

- .1: Customer may reenter credit card information or may cancel

Casos de uso possuem um nome; que normalmente é um verbo.

Casos de uso documentam alguns cenários, incluindo um cenário principal. Cada cenário é documentado como uma lista itemizada de passos

Extensões podem detalhar o caso de sucesso principal (item 3a); bem como detalhar casos de fracasso (item 6a)

# Exemplo de Caso de Uso

Um caso de uso pode usar (ou fazer referência, ou **incluir**) um outro caso de uso; que aparece sublinhado neste exemplo

## Buy a Product

Goal Level: Sea Level

Main Success Scenario:

1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming e-mail to customer

Extensions:

- 3a: Customer is regular customer
- .1: System displays current shipping, pricing, and billing information
  - .2: Customer may accept or override these defaults, returns to MSS at step 6
- 6a: System fails to authorize credit purchase
- .1: Customer may reenter credit card information or may cancel

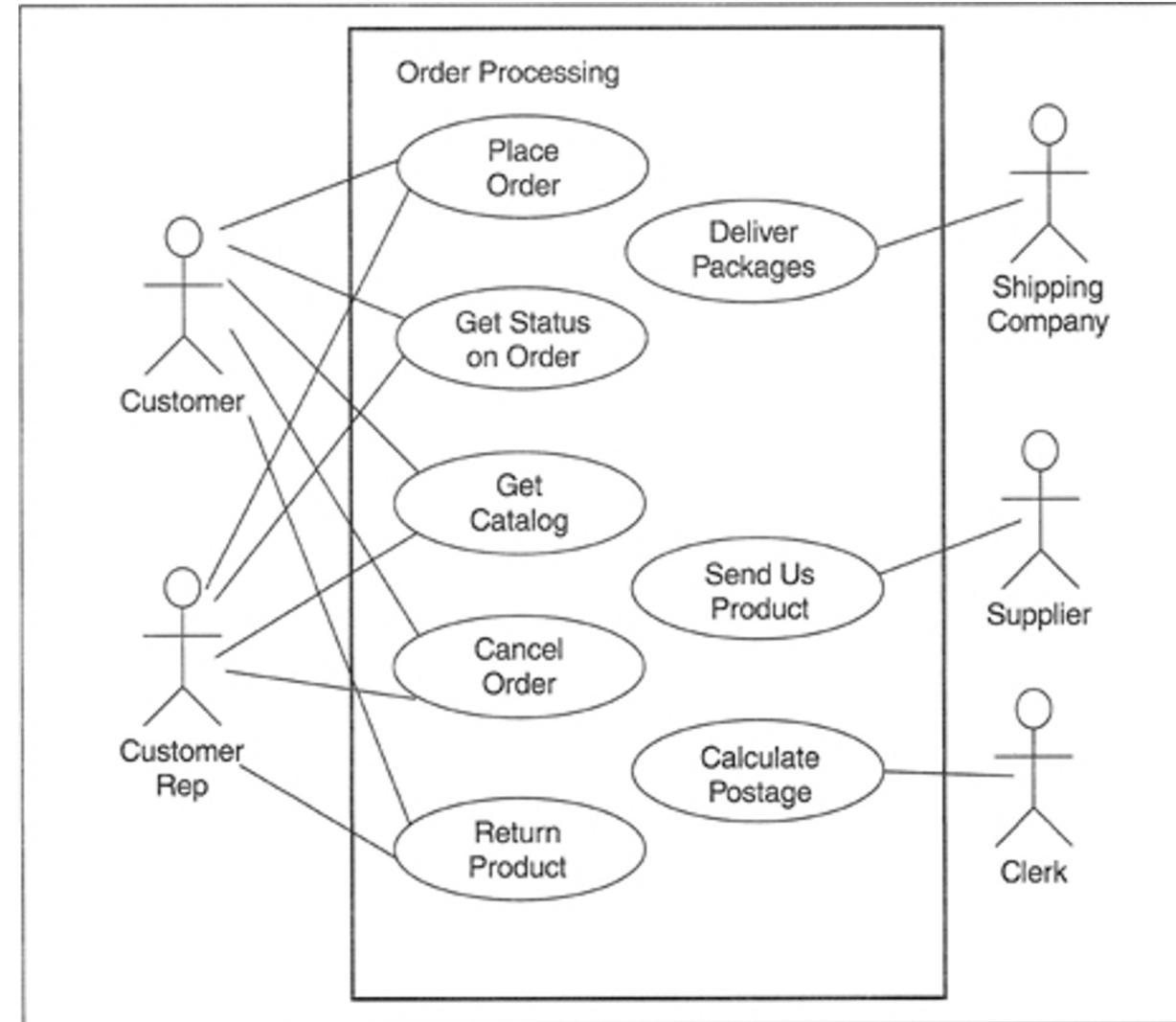
# Casos de Uso vs Estórias

- Ponto em comum: ambos são documentos textuais usado para especificar os requisitos (notadamente funcionais) de um sistema
- Porém, casos de uso são "narrativas", que descrevem alguns cenários de uso de um sistema, por um conjunto de atores
- Já estórias focam nas features de um sistema (o que o sistema deve oferecer); essas features tendem a ter uma granularidade mais fina, que casos de uso; normalmente, um caso de uso menciona diversas features de um sistema

# Diagrama de Caso de Usos

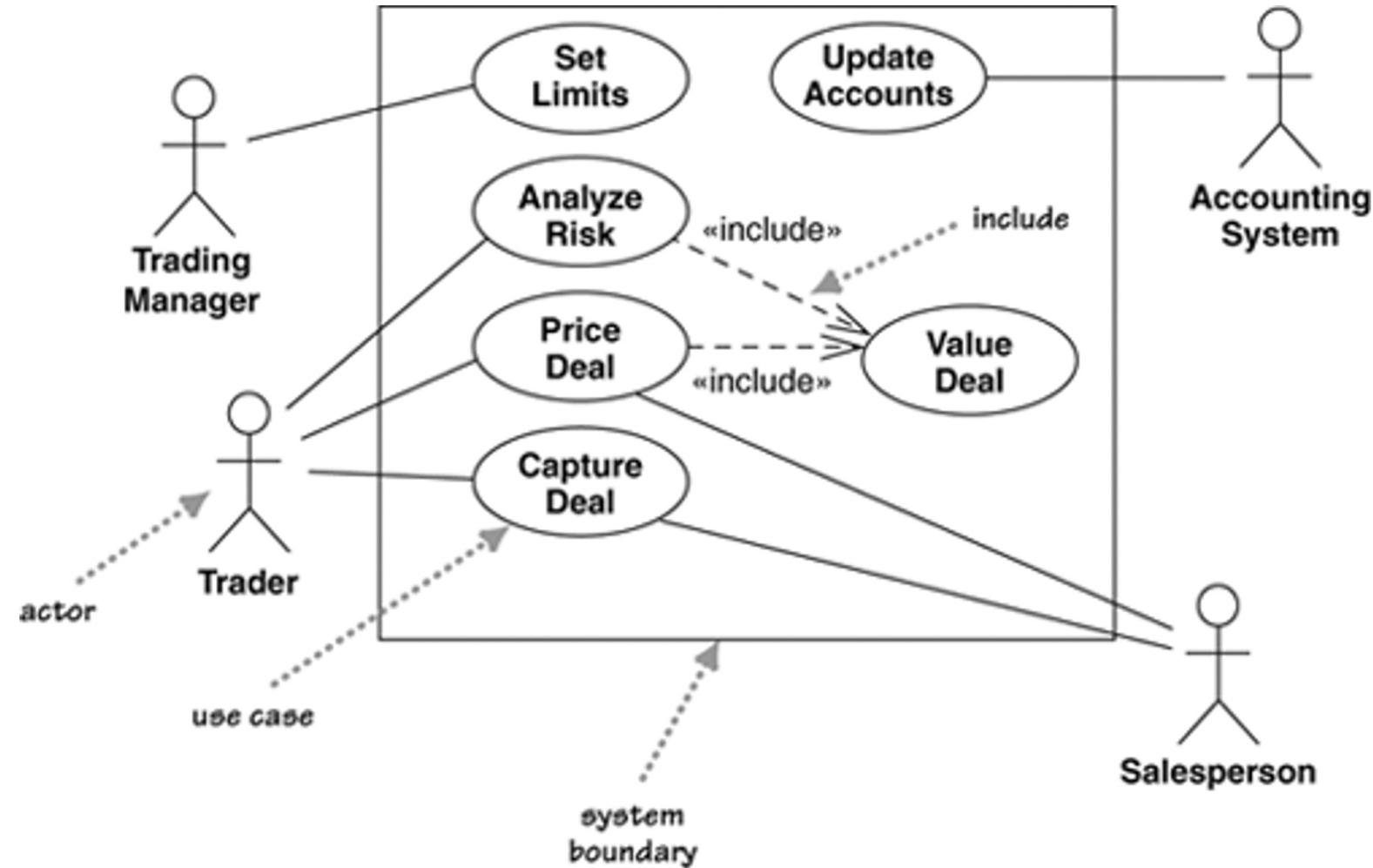
- Diagrama UML que mostra de forma gráfica:
  - Atores de um sistema (na forma de um "bonequinho")
  - Casos de uso de um sistema (só o nome deles, dentro de uma "elipse")
  - Relacionamento ator-caso de uso (ator X participa do caso de uso Y)
  - Relacionamento entre casos de uso (caso de uso X inclui caso de uso Y)
- Um diagrama de caso de uso é uma espécie de "índice" (table of contents) gráfico para os casos de uso de um sistema

# Diagrama de Caso de Uso: Exemplo 1



Fonte: Schneider & Winters; Applying Use Cases: A Practical Guide; Addison-Wesley, 2001.

# Diagrama de Caso de Uso: Exemplo 2



# Quando usar Diagramas de Casos de Uso?

- Nas fases iniciais de um sistema, quando os seus requisitos estiverem sendo especificados, pois diagramas de caso de uso são interessantes para entender os requisitos funcionais de um sistema
- É importante lembrar que casos de uso representam uma visão externa de um sistema (isto é, não é trivial estabelecer uma relação entre casos de uso e componentes internos de um sistema, como classes)

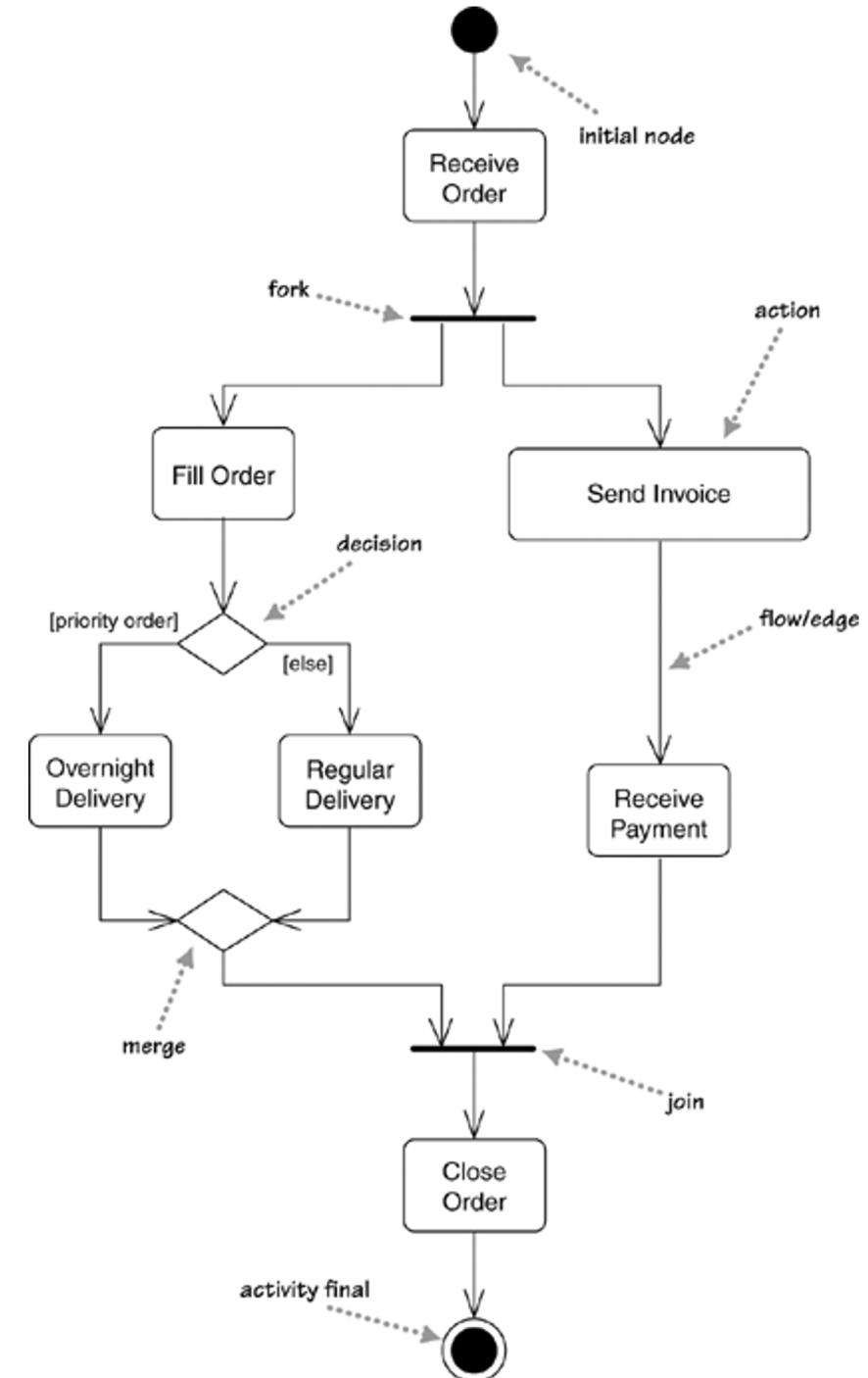
# **Diagrams de Atividades**

# Diagramas de Atividades

- Usados para representar "fluxos de execução" (workflow)
- Semelhantes a fluxogramas, porém supõem a execução paralela de ramos
- Quando usar? Para modelar processos de negócio. Logo, eles também não têm uma correspondência imediata com elementos de código fonte, como no caso de diagramas de casos de uso
- Observação: existem outras notações para modelagem de processos de negócio, como BPMN (Business Process Model and Notation)

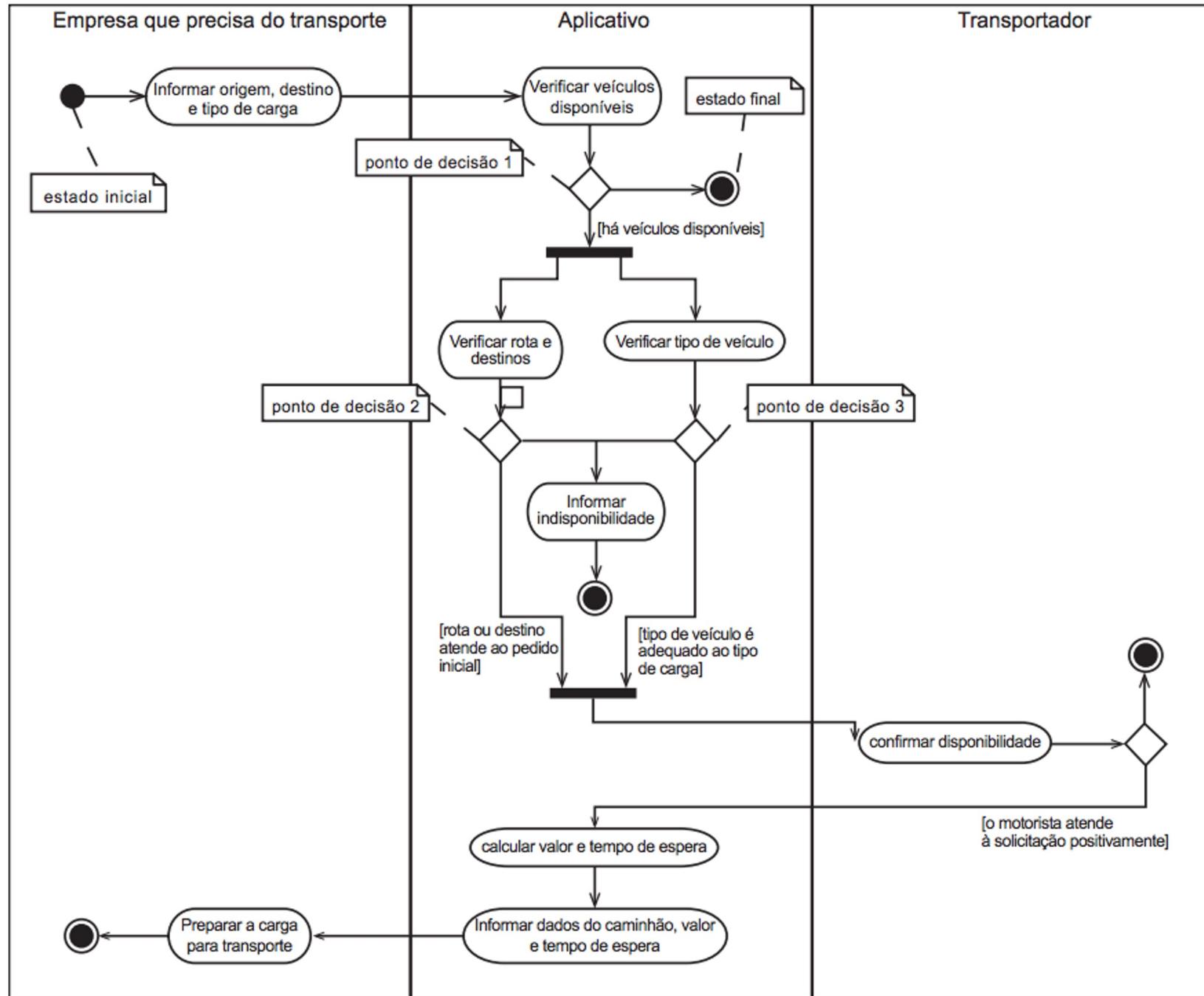
# Diagramas de Atividades: Exemplo

- **Nodos** são chamados de ações (e não de atividades); possuem 1 único fluxo de entrada e 1 único fluxo de saída
- **Forks**: possuem um fluxo de entrada e vários fluxos de saída (executados em paralelo)
- **Join**: possuem vários fluxos de entrada e um único fluxo de saída
- **Decisões**: são mutuamente exclusivas
- **Merge**: "une" fluxos derivados de uma decisão



## Exercício: ENADE 2017 (Sistemas de Informação)

Uma *startup* de tecnologia pretende desenvolver um aplicativo que aproxime empresas que precisam transportar mercadorias de condutores autônomos de veículos de carga (transportadores). O uso do aplicativo permitirá um aumento do número potencial de transportadores e de tipos de veículos, bem como o compartilhamento de frete, com redução do custo de transporte e do custo de armazenamento de mercadorias pelas empresas. Além disso, a visibilidade da disponibilidade dos transportadores pelo aplicativo pode aumentar suas possibilidades de negócios. Durante a concepção e o projeto do aplicativo, foram elaborados modelos UML (*Unified Modeling Language*), sendo um deles o diagrama de atividades, conforme apresentado na figura a seguir.



Com base no diagrama apresentado, avalie as afirmações a seguir.

- I. Os serviços oferecidos por outros aplicativos, como mapas para o cálculo de rotas, por exemplo, poderão ser usados no desenvolvimento do aplicativo, mesmo que não tenham sido representados no diagrama.
- II. Os fluxos paralelos indicados nos pontos de decisão 2 e 3 levam a erros no funcionamento do aplicativo, uma vez que permitem a continuidade do processo mesmo que não haja rota, destino ou tipo de veículo adequado.
- III. O diagrama apresenta atividades desempenhadas pela "Empresa que precisa de transporte" e pelo "Transportador", no entanto, as atividades que precisarão ser modeladas no sistema restringem-se às desempenhadas pelo "Aplicativo".
- IV. As ações "Informar indisponibilidade" e "Confirmar disponibilidade" correspondem a funcionalidades distintas no sistema, visto que a primeira se relaciona à existência de tipo de veículo, rota e destino e a segunda se relaciona à agenda do transportador.

É correto apenas o que se afirma em

- A** I e II.
- B** I e IV.
- C** II e III.
- D** I, III e IV.
- E** II, III e IV.

Com base no diagrama apresentado, avalie as afirmações a seguir.

- I. Os serviços oferecidos por outros aplicativos, como mapas para o cálculo de rotas, por exemplo, poderão ser usados no desenvolvimento do aplicativo, mesmo que não tenham sido representados no diagrama.
- II. Os fluxos paralelos indicados nos pontos de decisão 2 e 3 levam a erros no funcionamento do aplicativo, uma vez que permitem a continuidade do processo mesmo que não haja rota, destino ou tipo de veículo adequado.
- III. O diagrama apresenta atividades desempenhadas pela "Empresa que precisa de transporte" e pelo "Transportador", no entanto, as atividades que precisarão ser modeladas no sistema restringem-se às desempenhadas pelo "Aplicativo".
- IV. As ações "Informar indisponibilidade" e "Confirmar disponibilidade" correspondem a funcionalidades distintas no sistema, visto que a primeira se relaciona à existência de tipo de veículo, rota e destino e a segunda se relaciona à agenda do transportador.

É correto apenas o que se afirma em

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