



TÉCNICO EM DESENVOLVIMENTO DE SISTEMAS

UML

Docente: Diego Rodrigues

O que é UML?

UML – Unified Modeling Language (Linguagem de Modelagem Unificada)

Trata-se de uma linguagem unificada que habilita profissionais de TI a modelar e documentar aplicações de software.

Em termos de design, a UML oferece um meio de visualizar a arquitetura de um sistema por meio de diagramas, incluindo atividades, componentes individuais do sistema, a interação desses componentes, interfaces, interação com o mundo externo, entre outras.

É importante notar que UML não é o método de desenvolvimento em si. E é independente de plataforma / linguagem.

Versão mais atual: 2.5

Termos

Rational Software: Empresa que fornecia ferramentas para prática de engenharia de software. Vendida para a IBM em 2003.

OMG: Consórcio internacional de padrões da indústria de computadores, sem fins lucrativos, com representantes do governo, indústria e academia. Fornece especificações para padrões (mas não implementações). Áreas de atuação da OMG incluem sistemas financeiros, de saúde, armazenamento de dados, blockchain, IoT, cibersegurança e astronomia.

Diagrama: Representação gráfica (geralmente parcial) do modelo de um sistema.

Histórico

UML – Unified Modeling Language (Linguagem de Modelagem Unificada)

É uma linguagem de modelagem de propósito geral, para desenvolvimento em engenharia de software, que permite visualizar de uma forma padronizada o projeto (design) de um sistema.

Foi desenvolvida por Grady Booch, Ivar Jacobson e James Rumbaugh quando trabalhava na Rational Software entre 1994 e 1995.

O Object Management Group (OMG) adotou a UML como linguagem padrão em 1997, e em 2005 foi publicada pela OMG como uma padrão aprovado.

Histórico

Décadas de 70 e 80

- Analise e projeto estruturado de sistema.
- Domínio das linguagens procedurais.

Final dos anos 80

- Analise e projeto de sistemas orientados a objetos
- Técnicas de modelagem variadas
- Sem padronização

1997

- OMG padroniza o UML 1.0
- Em novembro / 1997 é lançado o UML 1.1 pela Rational

O que um diagrama representa?

Os diagramas representam duas visões distintas de um modelo de sistema:

Estática (Estrutural) - estrutura estática por meio de objetos, operações, relações e atributos.

Dinâmica (comportamental) - comportamento dinâmico por meio de colaboração entre os objetos e mudanças de seus estados internos

Diagramas UML

Estruturais

- Classes
- Objetos
- Pacotes
- Componentes
- Implantação
- Estrutura composta
- Perfil

Comportamentais

- Caso de uso
- Sequência
- Comunicação
- Máquina de estados
- Atividade
- Visão geral de interação
- Temporização

Links

OMG: www.omg.org

Rational: www.ibm.com/products/rational-software-architect-designer

UML: www.uml.org



O que são casos de uso

O diagrama de Casos de Uso auxilia no levantamento dos requisitos funcionais do sistema, descrevendo um conjunto de funcionalidades do sistema e suas interações com elementos externos e entre si.

Cenários: quando falamos de casos de uso, temos que ter em mente o conceito de cenários, que seriam instâncias de casos de uso.

Um cenário pode ser compreendido como uma sequência de passos que descreve uma interação entre um usuário e o sistema.

O que são casos de uso

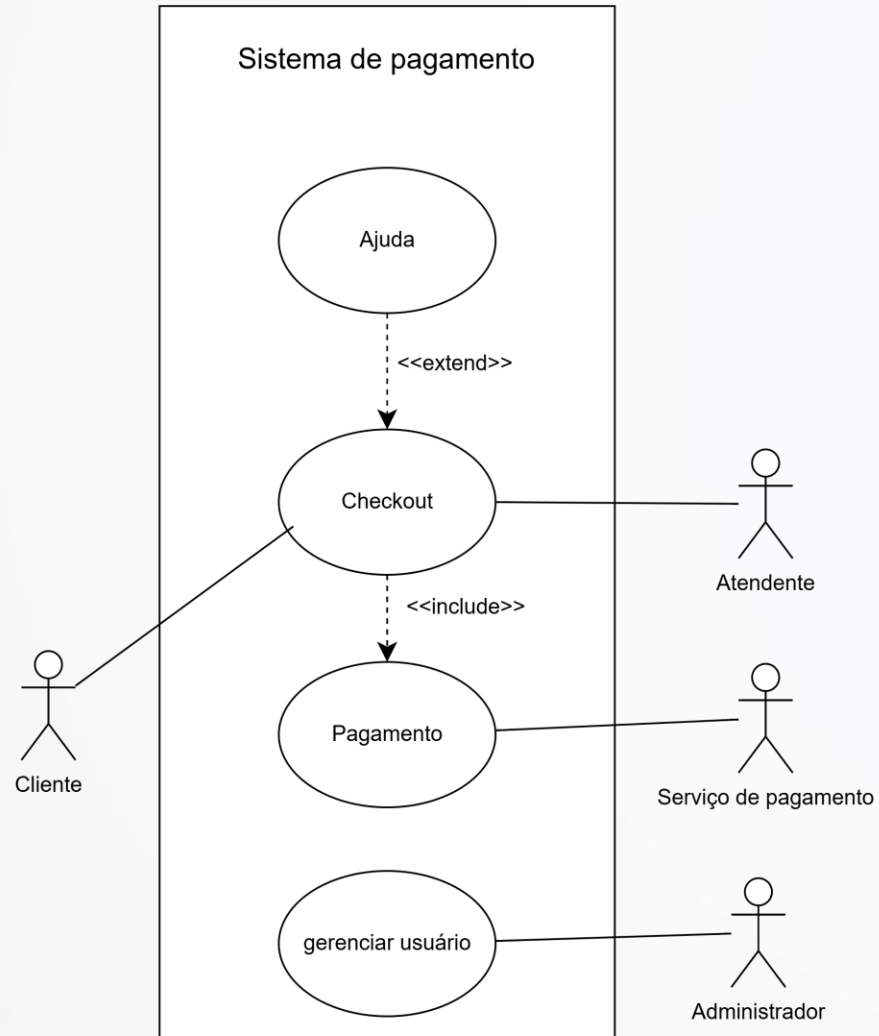
Sistemas não existem de forma isolada. Há a interação com humanos ou máquinas.

De acordo com Grady, Ivar e James:

"Um caso de uso específico o **comportamento** de um sistema (ou parte) é uma **descrição** de um conjunto de sequência de ações para produzir um resultado observável do valor de um **ator**"

Usando casos de uso para captar o comportamento pretendido de um sistema, sem especificar como esse comportamento é implementado.

O que são casos de uso



Aplicações dos casos de uso

No geral, empregamos casos de uso com duas finalidades

Definir escopo: Visualizar e entender as funcionalidades presentes no sistema

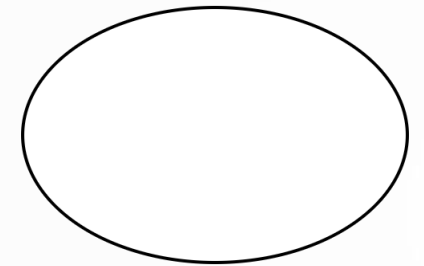
Identificar papéis: Identificar quem interage com o sistema e com quais funcionalidades essa interação ocorre.

A importância dos casos de uso

- Permite que os especialistas do domínio especifiquem sua visão externa de modo que desenvolvedores possam construir a visão interna
- Permitem que os desenvolvedores abordem um elemento e o compreendam como ele deve ser utilizado.
- Servem como base para testar cada elemento

Casos de uso

Elipse com rótulo que representa uma funcionalidade do sistema, sendo que esta pode estar estruturada em outra(s). Um caso de uso pode ser concreto, quando é iniciado diretamente por um ator, ou abstrato, quando é uma extensão de um outro caso de uso. Além disso há casos de uso primários e secundários. O primeiro representa os objetivos dos atores, já o segundo são funcionalidades do sistema que precisam existir para que este funcione corretamente.



Casos de uso: Nomes

Todo caso de uso possui um nome que o identifica e diferencia dos demais casos de uso do sistema.

O nome é uma sequência de caracteres de texto e deve ser único no pacote que o contém.

No geral, os nomes são expressões verbais ativas, que nomeiam um comportamento específico do sistema.

Exemplos de nomes de casos de uso:

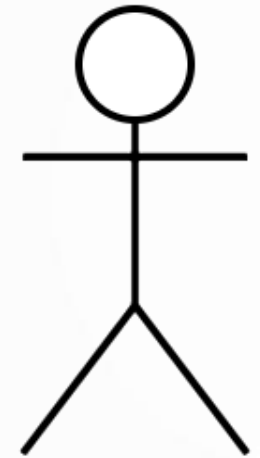
- Fazer pedido
- Pagar fatura
- Fazer login

O que são atores

Um caso de uso representa um requisito funcional do sistema, como um todo.

Envolve a interação de atores com sistema. Um ator apresenta um conjunto de papéis que os usuários dos casos de uso desempenham ao interagir com eles.

Atores podem ser humanos, organizações ou ainda sistemas automatizados (equipamentos, outros sistemas, etc.) e são elementos **externos** ao sistema.



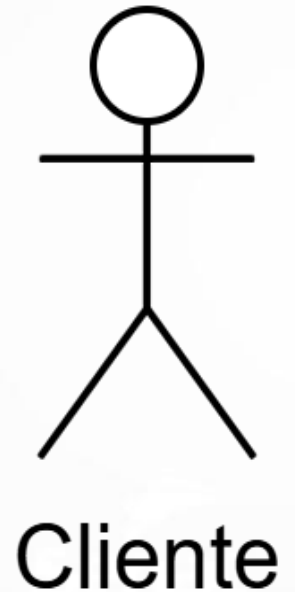
Ator

Atores e casos de uso

Atores se conectam a casos de uso por relação de associação, indicando que o ator e o caso de uso se comunicam entre si.

Por exemplo, para um caso de uso "Fazer pedido" pode haver um ator "Cliente" que se comunica com o caso de uso por uma associação.

O nome de um ator deve sempre informar seu papel, e não quem é representado.



Como identificar atores

Podemos identificar os atores que farão parte de um caso de uso fazendo perguntas como:

- Que organizações, pessoas ou entidades vão usar o sistema ou são importantes para realização de funções?
- Quais sistemas se comunicam com o sistema desenvolvido?
- Quem pode se interessar por algum requisito funcional do sistema?
- Quem deve receber informações sobre ocorrências no sistema.

Atores semelhantes devem ser organizados em uma hierarquia de generalização/ especialização

Como identificar casos de uso

Para identificar casos de uso, podemos fazer perguntas como:

- Quais as funcionalidades pretendidas para o sistema?
- Listar as necessidades e os objetivos de cada ator em relação ao sistema.
- Quais informações o sistema precisa retornar?
- O sistema precisa realizar ações que se repetem no tempo?
- Considerando os requisitos funcionais, determinar um ou mais casos de uso que os implementem.

Fluxo de eventos

Especificamos o comportamento de um caso de uso com **fluxo de eventos**. Nele incluiremos:

- Quando e como o caso de uso se inicia e termina
- Quando ele interage com os atores
- Fluxo básico e fluxos alternativos do comportamento.

Os fluxos de eventos podem ser especificados de várias maneiras, como:

- Texto informal (nosso exemplo)
- Tabela com pré e pós condições
- Máquina de estado
- Diagrama de atividades
- Pseudocódigo

Exemplo de fluxo de eventos: Validar usuário

Fluxo principal

- Sistema solicita ao cliente seu número de identificação (ID) e uma senha de acesso.
- Cliente digita seu ID e senha em um teclado numérico.
- Cliente confirma as entradas, pressionando Enter.
- Sistema verifica ID e senha fornecidos.
- Se ID for válido e senha corresponder, sistema reconhece usuário, permitindo sua entrada, finalizando o caso de uso

Exemplo de fluxo de eventos: Validar usuário

Fluxo alternativo

- Cliente fornece um ID ou senha inválidos.
- Neste caso, sistema reinicia o caso de uso.
- Se cliente fornecer credenciais erradas três vezes em sequência, sistema cancela transação e bloqueia acesso do usuário por uma hora.
- Sistema registra tentativa de acesso em log.

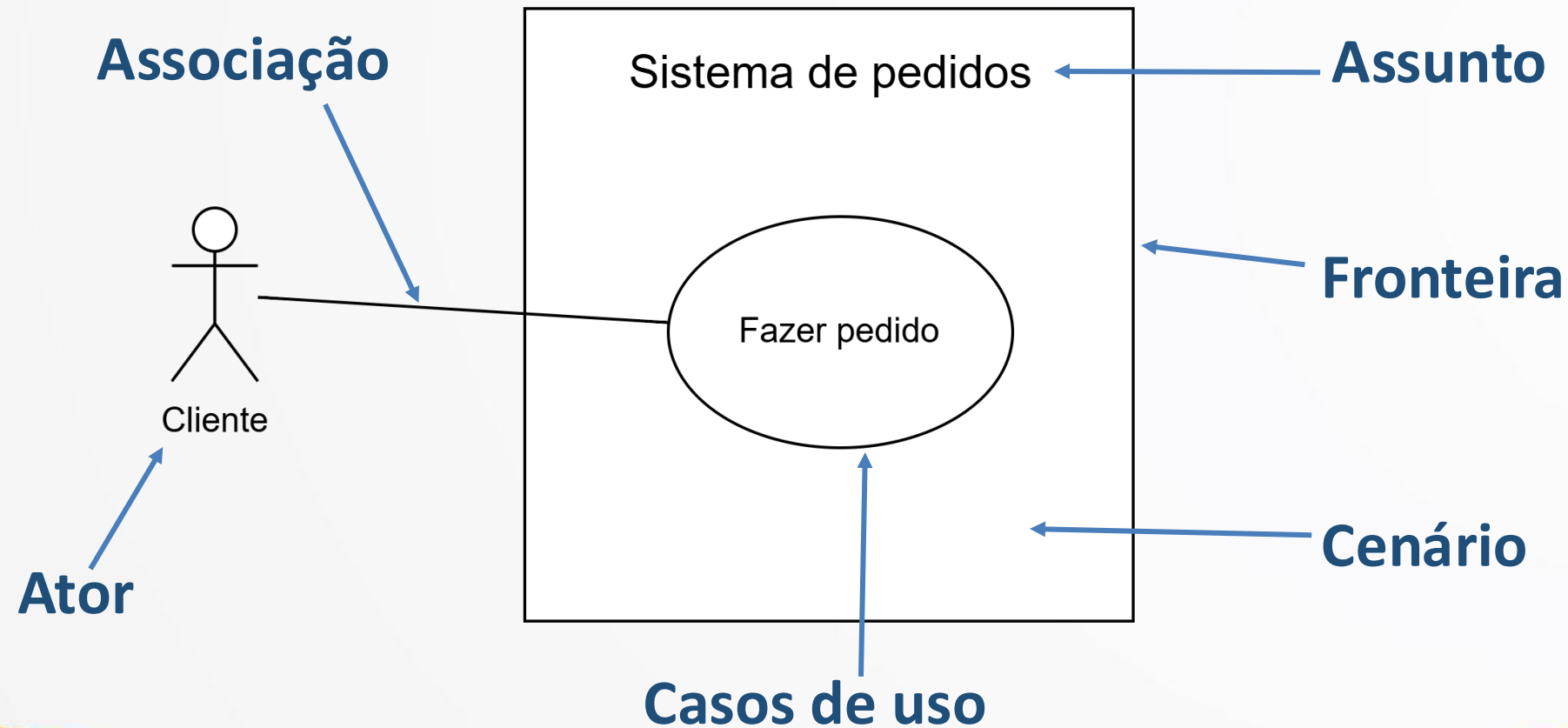
Exemplo de fluxo de eventos: Validar usuário

Fluxo alternativo

- Cliente cancela transação a qualquer momento, pressionando um botão Cancelar, exibido na tela.
 - Não são realizadas alterações na conta do cliente.
 - Caso de uso é finalizado.
-
- Também existe o fluxo de exceções, empregado para descrever possíveis restrições de sistema, como por exemplo não aceitar cartões como validade vencida.

Diagramas de casos de uso

Um diagrama de caso de uso contém



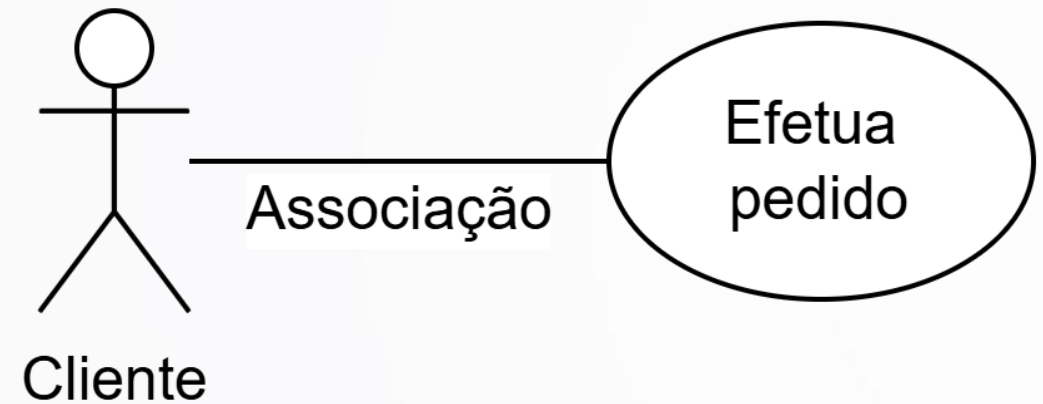
Relacionamentos

Entre casos de uso e atores:
Associação ou comunicação

Entre caso de uso:

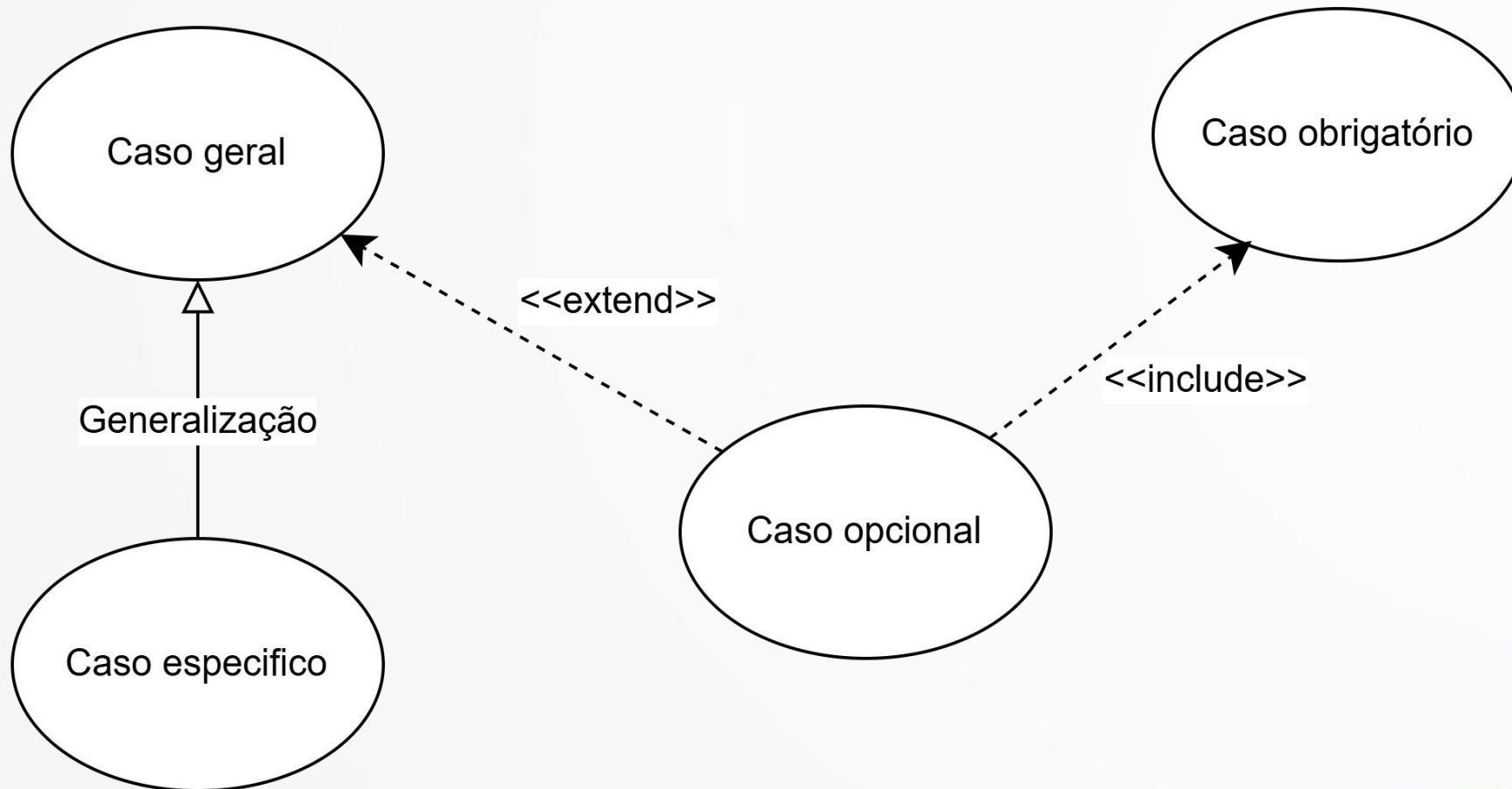
- Generalização / especialização
- Extensão / Estendido (extends)
- Inclusão (includes)

Entre os atores:
Generalização e especialização



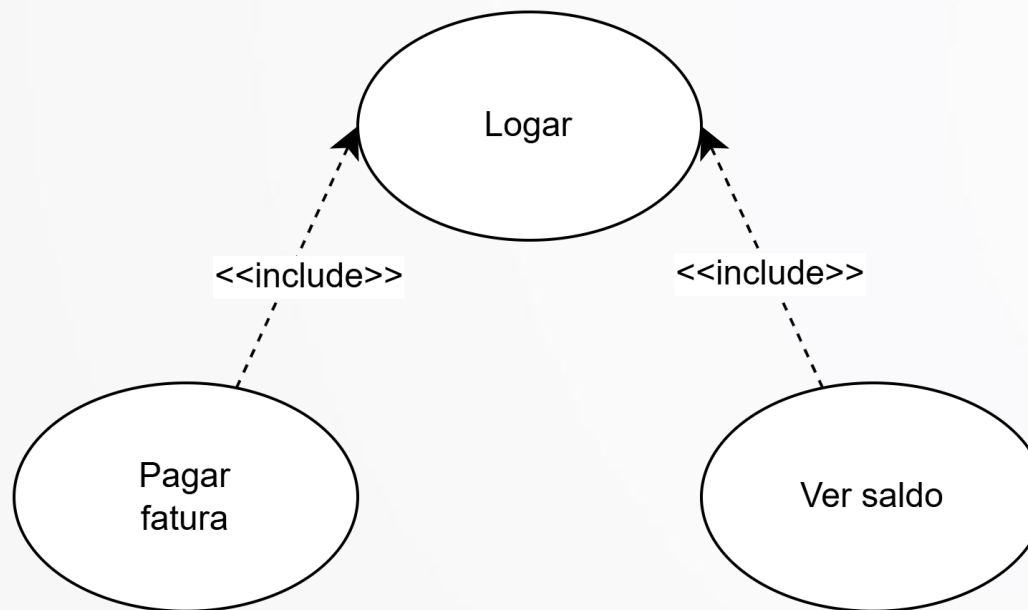
Relacionamento entre casos de uso

Abaixo temos as representações de relacionamentos entre casos de uso



Relacionamento de inclusão

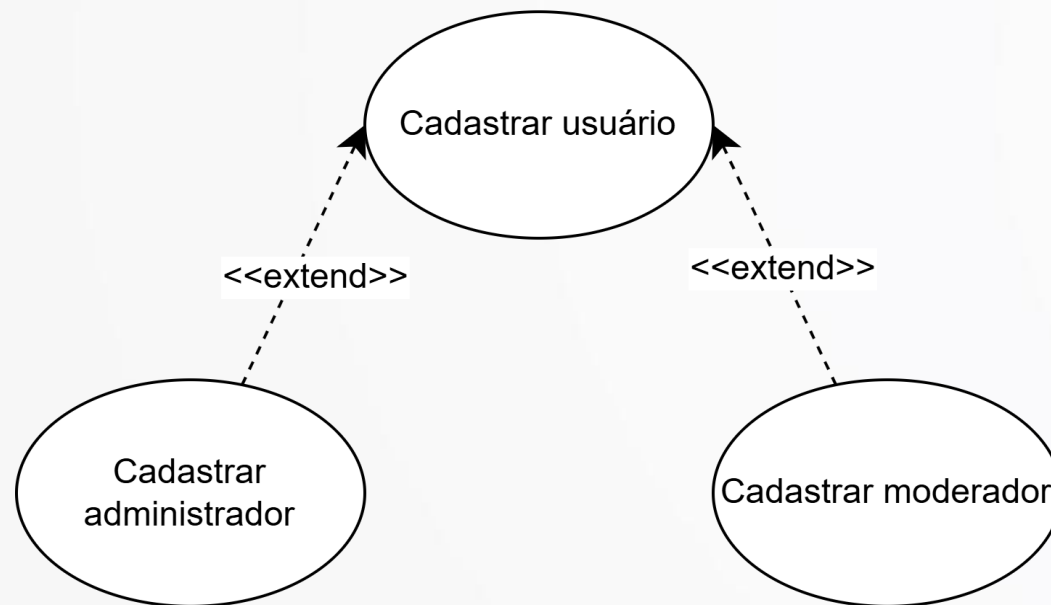
Utilizado quando um comportamento se repete em mais de um caso de uso. Por exemplo, num internet banking, um cliente que vai realizar um pagamento precisa se logar, assim como um cliente que vai visualizar o saldo também precisa se logar.



Logar é essencial para pagar fatura e para ver saldo. Ou Logar é parte de pagar fatura e também é parte de ver saldo

Relacionamento de Extensão

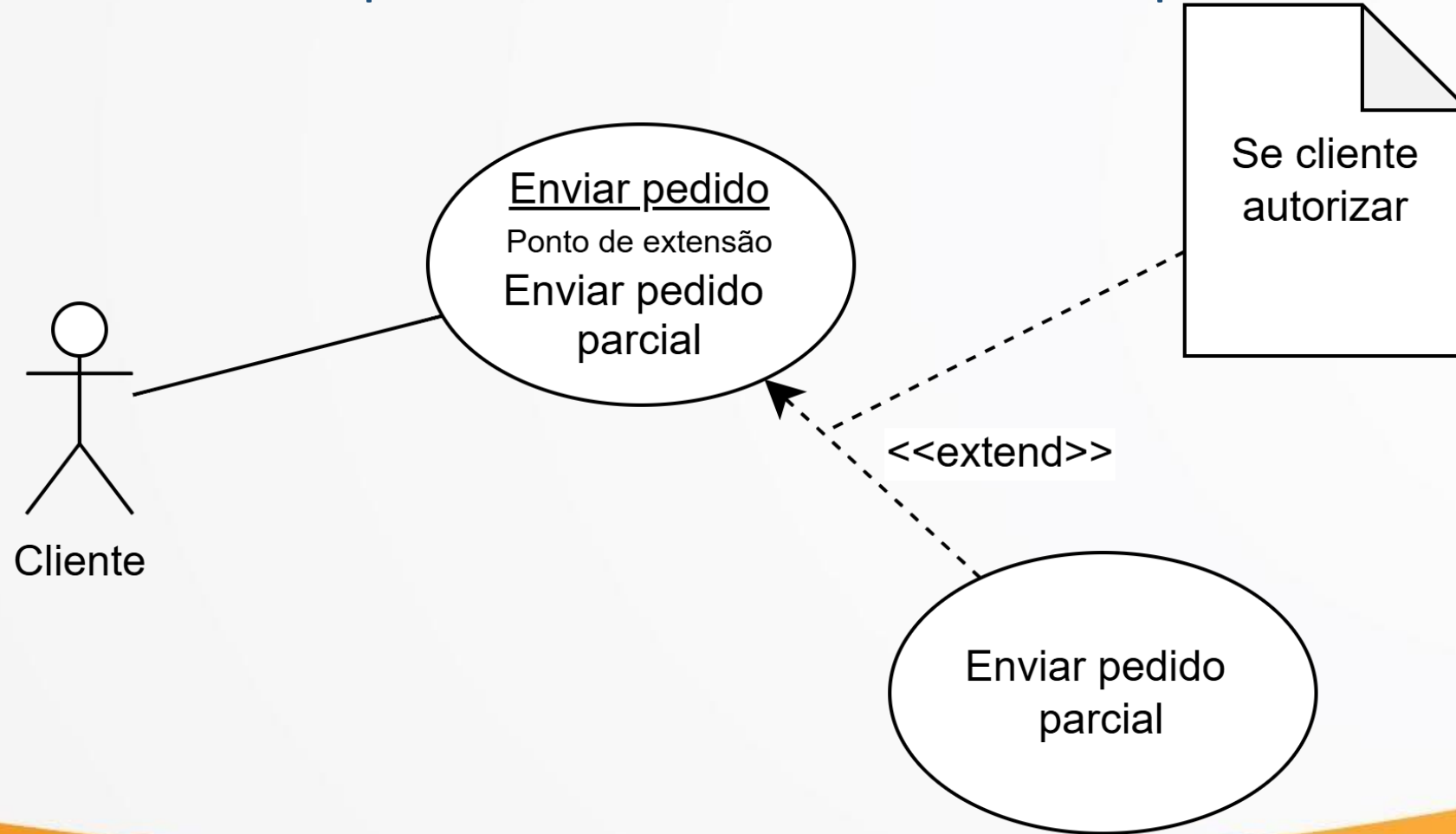
Utilizado quando se deseja modelar um relacionamento alternativo. Por exemplo, ao "cadastrar usuário" num sistema de forum, podemos "cadastrar um administrador" ou "cadastrar um moderador".



Cadastrar administrador e Cadastrar moderador são extensões de cadastrar usuário. Eles não são essenciais, só contém eventos adicionais sob certas condições.

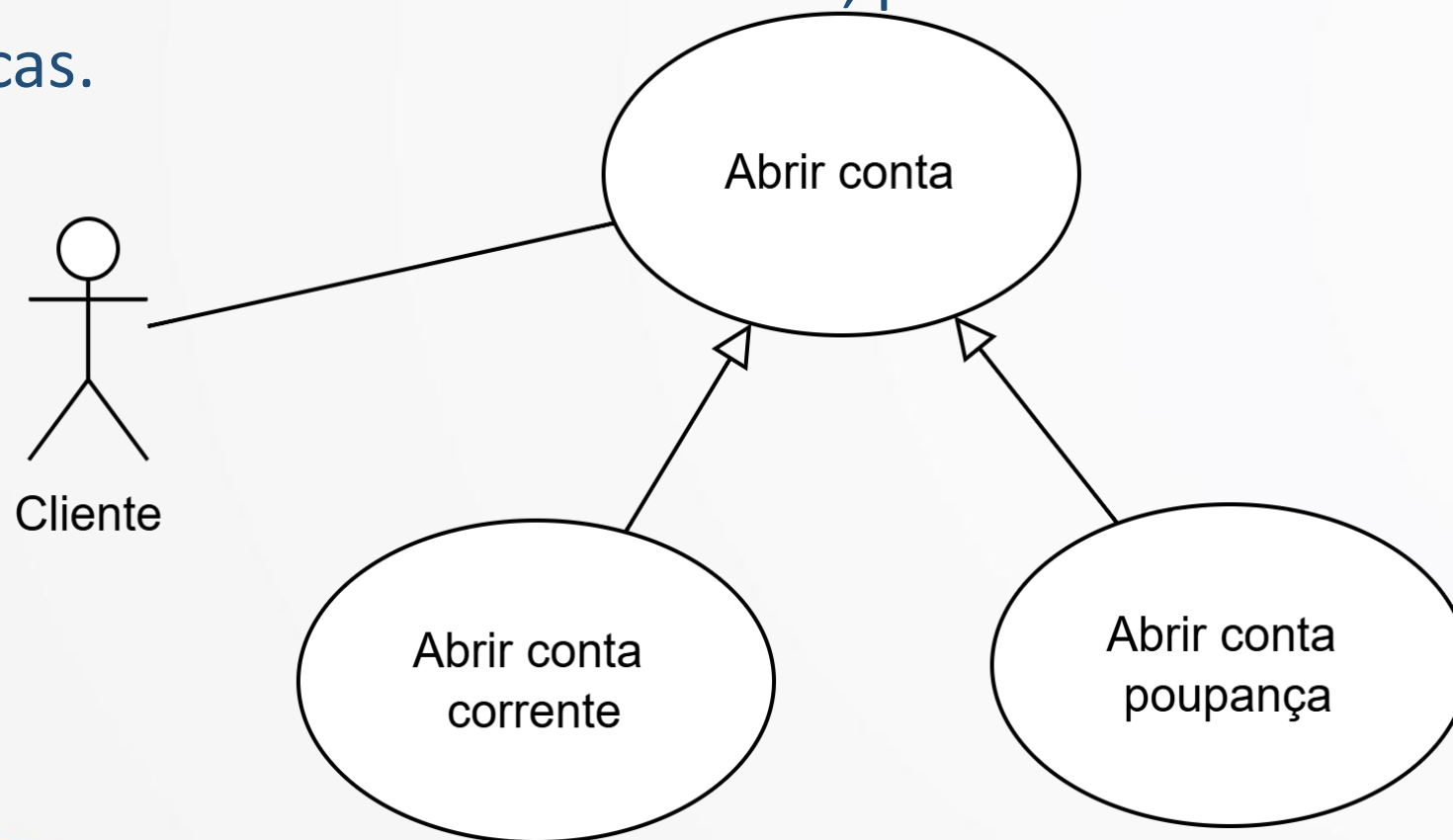
Relacionamento de extensão

Podemos também usar pontos de extensão e notas explicativas



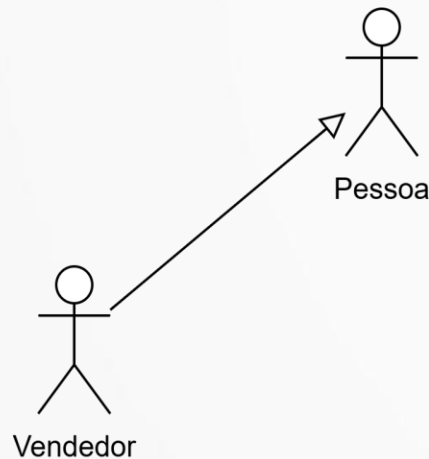
Relacionamento de generalização/ especialização

A generalização ocorre quando um ator ou caso de uso possui as mesmas características de um elemento base, porém com características específicas.



Relacionamento entre atores - generalização/especialização

Relacionamento entre atores, utilizado quando queremos representar uma especialização/generalização. Na figura a seguir, vendedor é especialização de pessoa (ou pessoa é generalização de vendedor), é representado por um alinhamento com um triângulo vazado.



Os casos de uso de pessoa são também casos de uso de vendedor mas vendedor tem seus próprios casos de uso.

Resumindo relacionamento

	Associação	Extensão	Inclusão	Generalização
Entre Casos de Uso		X	X	X
Entre Atores				X
Entre Casos de Uso e Atores	X			

Boas práticas ao criar diagramas de casos de uso

- Distribua os elementos no diagrama de forma a minimizar o cruzamento de linhas, que interfere na compreensão.
- Organize os elementos que têm comportamentos e papéis relacionados próximos entre si.
- Faça uso de notas e cores diferentes como indicações visuais para ressaltar características importantes
- Caso relacionamentos de inclusão e extensão fiquem muito complicados, os exiba em um outro diagrama complementar.

Exemplo de diagrama de casos de uso

Exemplo: Sistema bancário

Sistema permite a abertura e encerramento de contas.

Sistema permite consultar saldo e movimentações.

Sistema permite realizar saques, depósitos e transferências.

Contas do tipo corrente e poupança

Interagem com o sistema: Cliente pessoa física ou jurídica e funcionários como caixas e gerente do banco.

Identificar atores

Cliente

- Pessoa física
- Pessoa jurídica

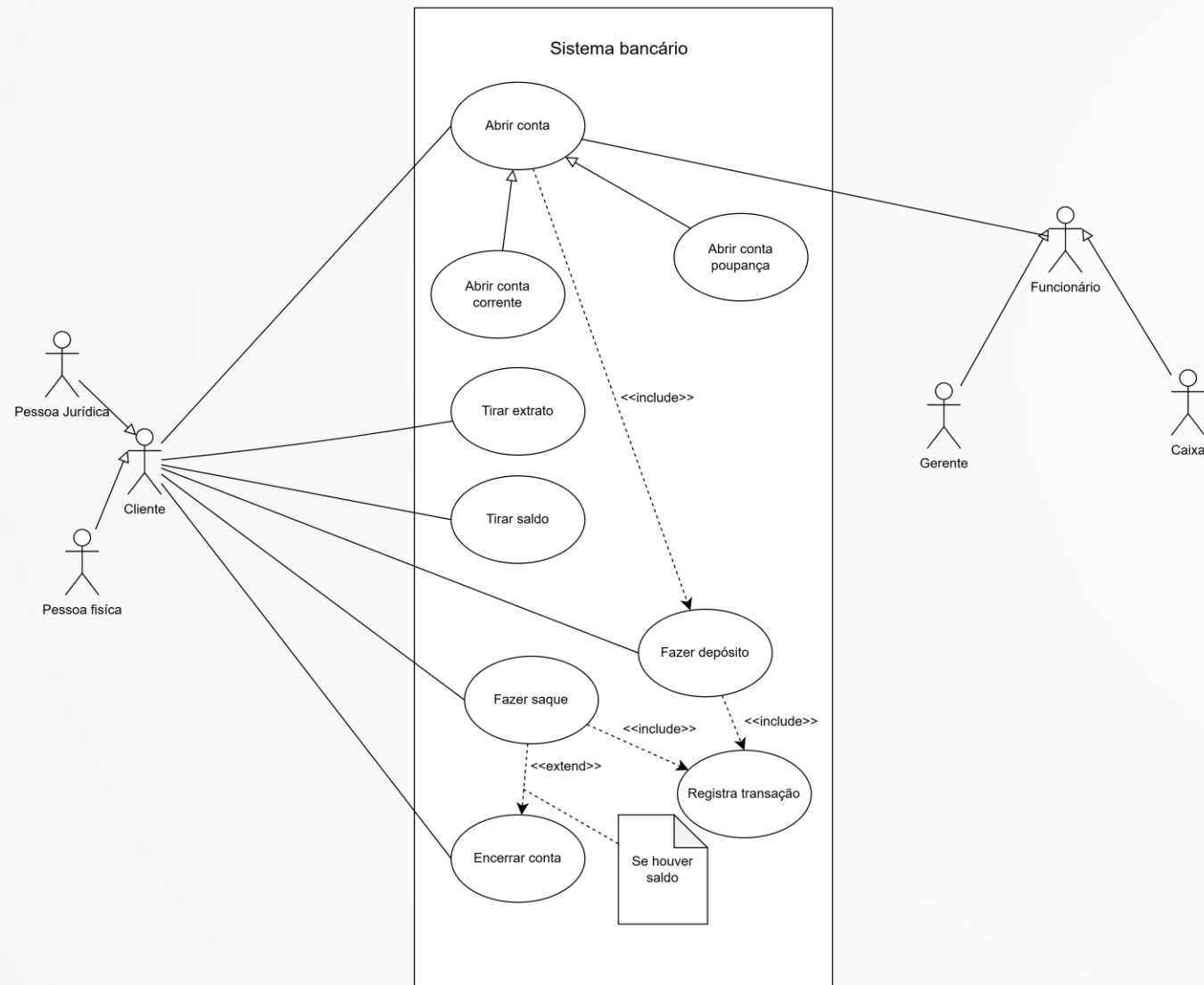
Funcionário

- Caixa
- Gerente

Identificar casos de uso

- Abrir conta corrente
- Abrir conta poupança
- Realizar saque
- Fazer depósito
- Encerrar conta
- Tirar extrato
- Tirar saldo
- Registrar transação

Identificar casos de uso



Conceito de objeto

Um objeto é uma representação de uma entidade do mundo real ou de algo abstrato, que possui características e pode realizar ações. Os objetos são instâncias de classes e representam algo único, como uma pessoa específica, um produto, um carro, etc.

Entidade do mundo real e abstrato

Uma entidade do mundo real ou algo abstrato, na programação orientada a objetos, refere-se a qualquer elemento que possua características e possa ser representado como um objeto. Esse termo é usado para definir objetos que queremos modelar, sejam eles tangíveis (como uma pessoa ou um carro) ou conceituais (como uma conta bancária ou uma transação financeira).

Entidade do mundo real (Tangível)

São objetos que representam coisas físicas que realmente existem no mundo ao nosso redor. Esses objetos possuem atributos e comportamentos que podem ser descritos de forma direta.

Exemplos:

- Um carro (com atributos como cor, modelo, ano, e métodos como acelerar e frear).
- Uma pessoa (com atributos como nome, idade e métodos como falar e andar).
- Um produto em uma loja (com atributos como nome, preço, categoria e métodos para aplicar desconto, calcular impostos).

Entidade abstrata (Conceitual)

São objetos que representam conceitos, ideias, ou elementos que não possuem uma forma física diretamente perceptível, mas que são importantes no contexto do sistema que estamos desenvolvendo.

Exemplos:

- Uma conta bancária (com atributos como saldo, número da conta e métodos para sacar, depositar).
- Uma transação financeira (com atributos como valor, data, tipo e métodos para validar ou processar a transação).
- Um pedido de compra (com atributos como número do pedido, data, lista de itens e métodos para calcular o total e verificar o status).

Componentes de um objeto

Atributos: Características ou propriedades de um objeto. Eles armazenam o estado de um objeto e são representados por variáveis dentro do código.

Exemplo: No objeto "Carro", os atributos poderiam incluir cor, modelo, ano, marca, etc.

Métodos: Comportamentos ou ações que o objeto pode realizar. Eles definem as operações que um objeto pode executar ou que podem alterar seu estado.

Exemplo: Para o objeto "Carro", métodos poderiam incluir acelerar(), frear(), ligar(), etc.

Exemplos práticos de objetos

Exemplo 1 - Pessoa:

Objeto: Uma pessoa chamada Diego, que tem 33 anos e é programador.

Atributos: Nome = Diego, Idade = 33, Profissão = "Programador".

Métodos: falar(), andar(), comer().

Exemplo 2 - Conta Bancária:

Objeto: Conta de um cliente no banco.

Atributos: saldo = R\$ 120.000,00, titular = Mauro, número da conta = 000000.

Métodos: depositar(), sacar(), consultarSaldo().

Conceito de classe

Uma classe é um "molde" ou "modelo" que define as características e comportamentos comuns de um conjunto de objetos.

Exemplo de classe

Objeto Pessoa1:

Objeto: Uma pessoa chamada Diego, que tem 33 anos e é programador.

Atributos: Nome = Diego, Idade = 33, Profissão = "Programador".

Métodos: falar(), andar(), comer().

Objeto Pessoa2:

Objeto: Uma pessoa chamada Jhenny, que tem 26 anos e é designer.

Atributos: Nome = Jhenny, Idade = 26, Profissão = "Designer".

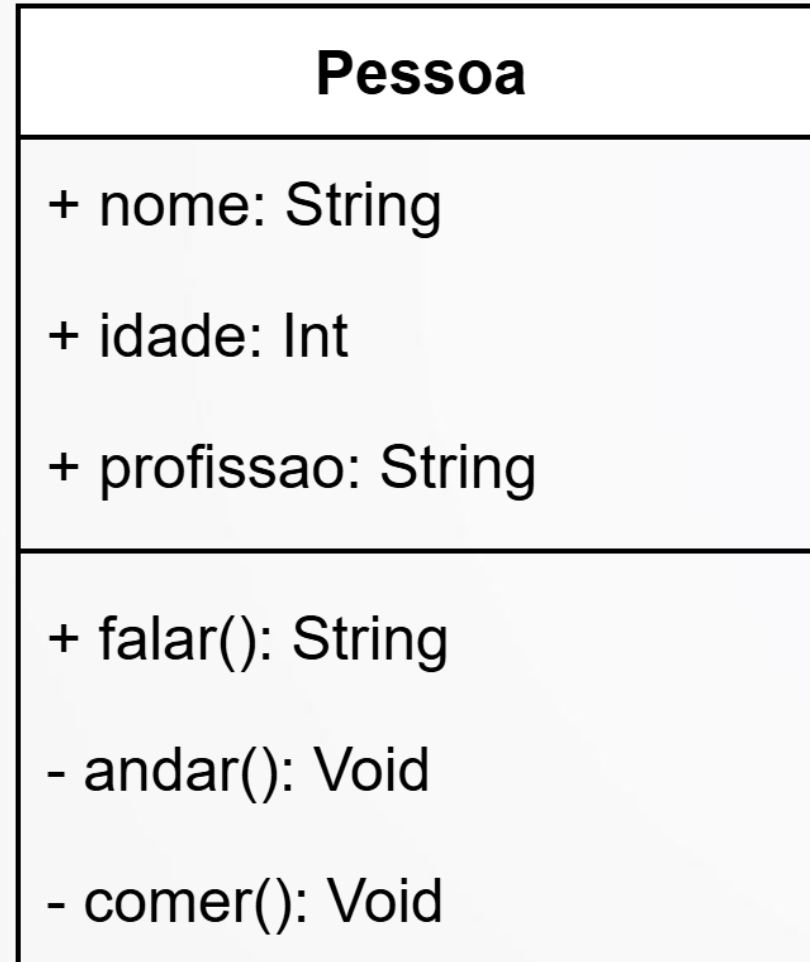
Métodos: falar(), andar(), comer().

Classe Pessoa

Atributos: Nome, Idade, Profissão.

Métodos: falar(), andar(), comer().

Exemplo de diagrama de classe



Diferença entre classe e objeto

A classe é uma definição genérica e abstrata (como um molde), enquanto o objeto é uma instância específica dessa classe.

Exemplo: A classe "Carro" define o que todos os carros têm em comum (atributos como cor, modelo, e métodos como acelerar()), mas cada objeto criado a partir dessa classe representa um carro específico, como um "Carro vermelho, modelo Sedan, ano 2022".

Senac

Diagrama de classe

Diagrama de classe é um tipo de diagrama da UML (Unified Modeling Language), usado para modelar a estrutura estática de um sistema orientado a objetos.

Propósito

- Representar graficamente as classes, seus atributos, métodos e os relacionamentos entre elas.
- Ajudar desenvolvedores e analistas a planejar a arquitetura do sistema antes do desenvolvimento.

Vantagens

- Facilita o entendimento da estrutura do sistema.
- Serve como documentação visual.
- Auxilia na comunicação entre a equipe, mostrando de forma clara os componentes do sistema e como eles interagem.

Componentes de um diagrama de classe

Representamos uma classe usando um diagrama dividido em três compartimentos:

Nome: Inclui o nome e o estereótipo da classe (informação sobre a classe)

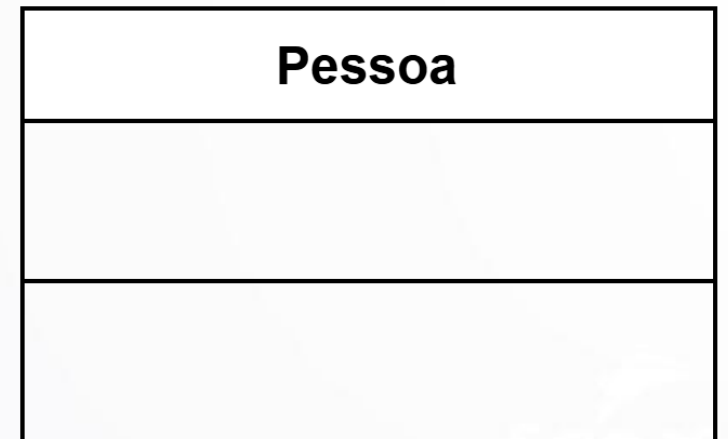
Atributo: Lista de atributos da classe no formato **nome:tipo** ou **nome:tipo=valor**

Operações: Lista de métodos da classe no formato **método(parametro):tipo_retorno**

Componentes de um diagrama de classe

Nome da Classe:

- Cada classe tem um nome que indica a entidade que ela representa (ex.: Pessoa, Produto, Pedido). Esse nome aparece no topo do retângulo que representa a classe no diagrama.
- Convenções de nomenclatura: Use nomes descritivos, no singular, que indiquem claramente o propósito da classe.



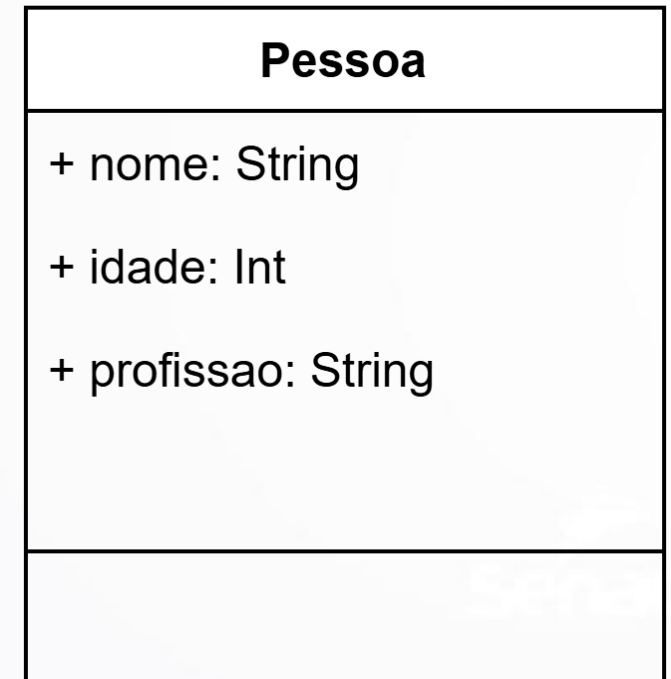
Componentes de um diagrama de classe

Atributos:

Os atributos são propriedades ou características de uma classe que armazenam dados e representam o estado de cada objeto dessa classe.

Notação: A lista de atributos aparece abaixo do nome da classe, com a seguinte estrutura:

- `modificador_de_acesso nome_do_atributo: tipo`
- Exemplo: `+ nome: String`, `- idade: int`



Modificadores de acesso (atributos / métodos)

Representamos a visibilidade dos atributos e das operações usando os modificadores de acesso a seguir:

Símbolo	Modificador	Descrição
+	Público	Acessível de qualquer lugar do programa.
-	Privado	Acessível apenas dentro da própria classe.
#	Protegido	Acessível dentro da própria classe e por suas subclasses.
~	Pacote	Acessível apenas por classes no mesmo pacote.
/	Derivado	Atributo ou método calculado a partir de outros, não armazenado diretamente.

+ Público (Public)

Descrição: Permite que o atributo ou método seja acessado de qualquer lugar, ou seja, por qualquer outra classe.

Uso: Geralmente usado para métodos que representam a interface pública da classe, ou seja, as funcionalidades que estão disponíveis para outras classes.

Exemplo: + nome: String — Este atributo nome pode ser acessado diretamente por qualquer outra classe.

- Privado (Private)

Descrição: Limita o acesso do atributo ou método apenas à própria classe onde ele é declarado. Não pode ser acessado diretamente por outras classes.

Uso: Usado para ocultar detalhes internos da classe, como dados sensíveis ou métodos auxiliares.

Exemplo: - saldo: float — Este atributo saldo só é acessível dentro da classe em que foi declarado.

Protegido (Protected)

Descrição: Permite que o atributo ou método seja acessado pela própria classe e por suas subclasses (classes que herdam dela), mas não por outras classes que não fazem parte da hierarquia.

Uso: Comum em sistemas de herança, onde queremos que apenas classes filhas possam acessar determinados elementos.

Exemplo: # calcularDesconto(): float — Esse método calcularDesconto() é acessível dentro da própria classe e em suas subclasses, mas não em classes externas.

~ Pacote (Package)

Descrição: Define o acesso ao atributo ou método apenas para classes que estão no mesmo pacote ou módulo.

Uso: Útil para organizar o acesso entre classes relacionadas que estão no mesmo contexto ou funcionalidade, mas que não precisam ser acessadas externamente.

Exemplo: ~ validaDados(): boolean — Esse método validaDados() só é acessível por outras classes dentro do mesmo pacote.

/ Derivado (Derived)

Descrição: Indica que o atributo ou método é derivado, ou seja, seu valor não é armazenado diretamente, mas calculado a partir de outros atributos ou métodos.

Uso: Usado quando um valor pode ser inferido a partir de outros dados, ajudando a evitar redundância.

Exemplo: /precoTotal: float — O atributo precoTotal é derivado de outros atributos e seu valor é calculado dinamicamente.

Tipo de dados

String: Textos ou sequências de caracteres.

Int: Números inteiros.

Long: Números inteiros grandes.

Double: Números de ponto flutuante com precisão dupla.

Float: Números de ponto flutuante com precisão simples.

Decimal: Números com precisão decimal (útil para valores financeiros).

Char: Um único caractere.

Boolean: Valores verdadeiro/falso (True/False).

Date: Data (apenas data, sem hora).

Time: Tempo (hora, minuto, segundo).

Timestamp: Combinação de data e hora.

Currency: Valores monetários, que podem incluir símbolo de moeda e precisão decimal.

Blob (Binary Large Object): Dados binários, como imagens e arquivos.

Clob (Character Large Object): Grandes quantidades de texto.

UUID (Universally Unique Identifier): Identificador único universal.

Tipo de dados

XML: Dados estruturados no formato XML.

JSON: Dados estruturados no formato JSON.

Enum: Tipo enumerado, com valores predefinidos (ex: dias da semana).

Array: Coleção de elementos do mesmo tipo.

Set/Collection: Conjunto de elementos únicos, sem duplicatas.

Map/Dictionary: Pares chave-valor para associações de dados.

Geolocation/Spatial: Dados de localização, como coordenadas geográficas.

Object/Reference: Referência a outra classe ou objeto.

Void: Tipo especial que indica ausência de valor de retorno em métodos.

List: Coleção de elementos.

Esses tipos cobrem as necessidades mais comuns e específicas para representação de dados e funcionalidades em diagramas de classe UML, permitindo flexibilidade e clareza na modelagem de sistemas.

Componentes de um diagrama de classe

Métodos:

Os métodos são as operações ou comportamentos que a classe pode realizar. Eles aparecem na terceira seção da classe, logo abaixo dos atributos.

Notação: modificador_de_acesso
nome_do_metodo(): tipo_de_retorno

Exemplo: + calcularTotal(): float, -
verificarEstoque(): bool

Pessoa
+ nome: String
+ idade: Int
+ profissao: String
+ falar(): String
- andar(): Void
- comer(): Void

Representação de uma classe

Exemplo: Representando uma classe Pessoa, que contém os atributos nome, sobrenome e dataNasc, além dos métodos calcularIdade e estudar:

Pessoa
+ nome: String + idade: Int + dataNascimento: Date
+ calcularIdade(dataNascimento): Int + estudar(): Void

Relacionamento entre classes

Um relacionamento é uma conexão entre itens. Existem vários tipos de relacionamento possíveis entre classes

Associação: Representa um vínculo entre duas classes, onde uma classe "usa" a outra.

Agregação: Relação onde uma classe é composta de outras, mas as partes podem existir independentemente (ex.: "Curso" e "Aluno").

Composição: Relação onde uma classe depende fortemente de outra (ex.: "Carro" e "Motor").

Herança: Relação onde uma classe herda atributos e métodos de outra (ex.: "Pessoa" e "Pessoa Física" ou "Pessoa Jurídica").

Associação

A associação é o relacionamento mais simples entre classes, representando que uma classe está ligada à outra de alguma forma. É uma conexão direta, mas sem uma hierarquia ou dependência rígida.

Notação: Uma linha simples entre as classes.

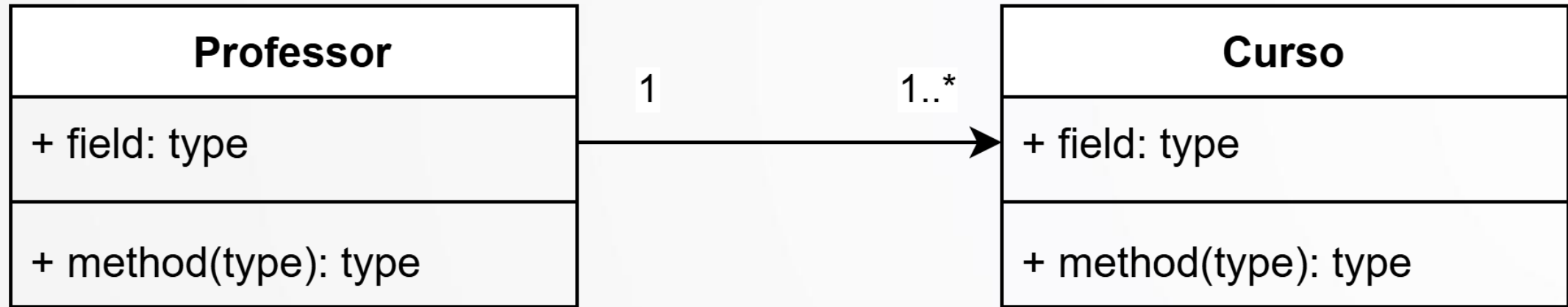
- Opcionalmente, pode-se especificar a multiplicidade (por exemplo, 1..*, 0..1), indicando quantas instâncias de uma classe podem estar associadas à outra.
- Também pode incluir uma seta, caso o relacionamento tenha uma direção, mas, geralmente, não usa setas.

Exemplo: Um relacionamento entre as classes Professor e Curso, onde um professor pode lecionar vários cursos e cada curso é ministrado por um professor.

Associação



Associação



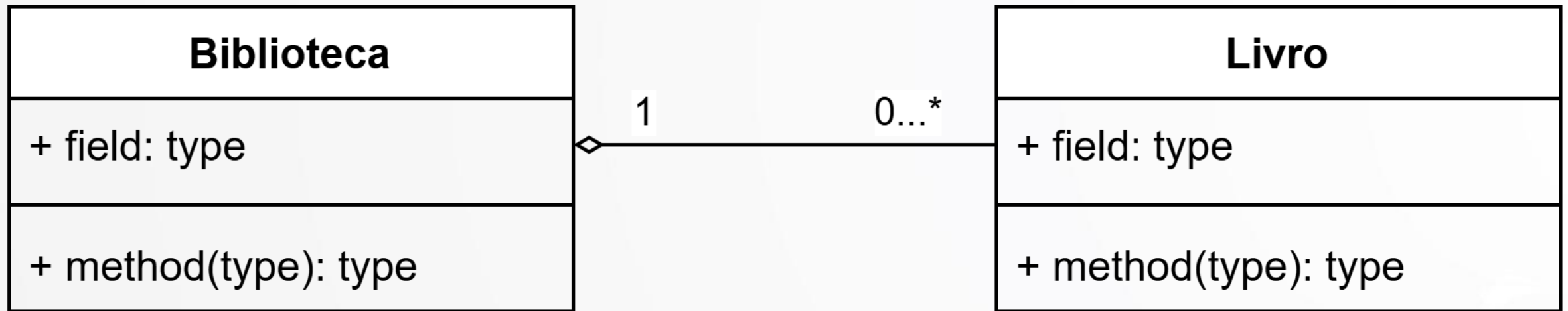
Agregação

A agregação é um tipo de associação mais específica, que representa uma relação de "todo-parte", onde uma classe é composta por outras, mas as partes podem existir independentemente do todo.

Notação: Uma linha com um losango aberto (ou vazio) na extremidade da classe "todo".

Exemplo: Uma Biblioteca é composta por Livros, mas os livros podem existir fora da biblioteca. Se a Biblioteca for destruída, os Livros não necessariamente serão destruídos.

Agregação



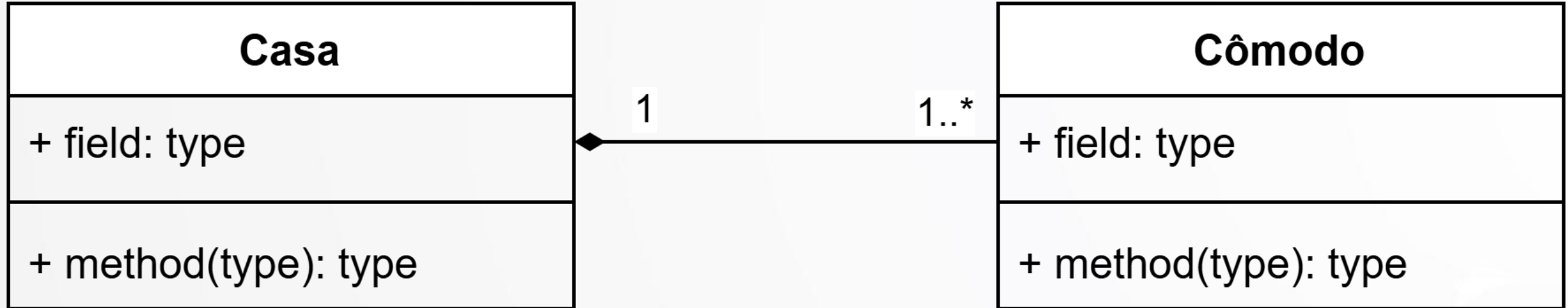
Composição

A composição é semelhante à agregação, mas representa uma relação mais forte de "todo-parte", onde as partes dependem do todo para existir. Se o "todo" é destruído, as "partes" também são.

Notação: Uma linha com um losango preenchido na extremidade da classe "todo".

Exemplo: Uma Casa é composta por Cômodos. Se a Casa for destruída, os Cômodos também serão.

Composição



Herança (ou Generalização)

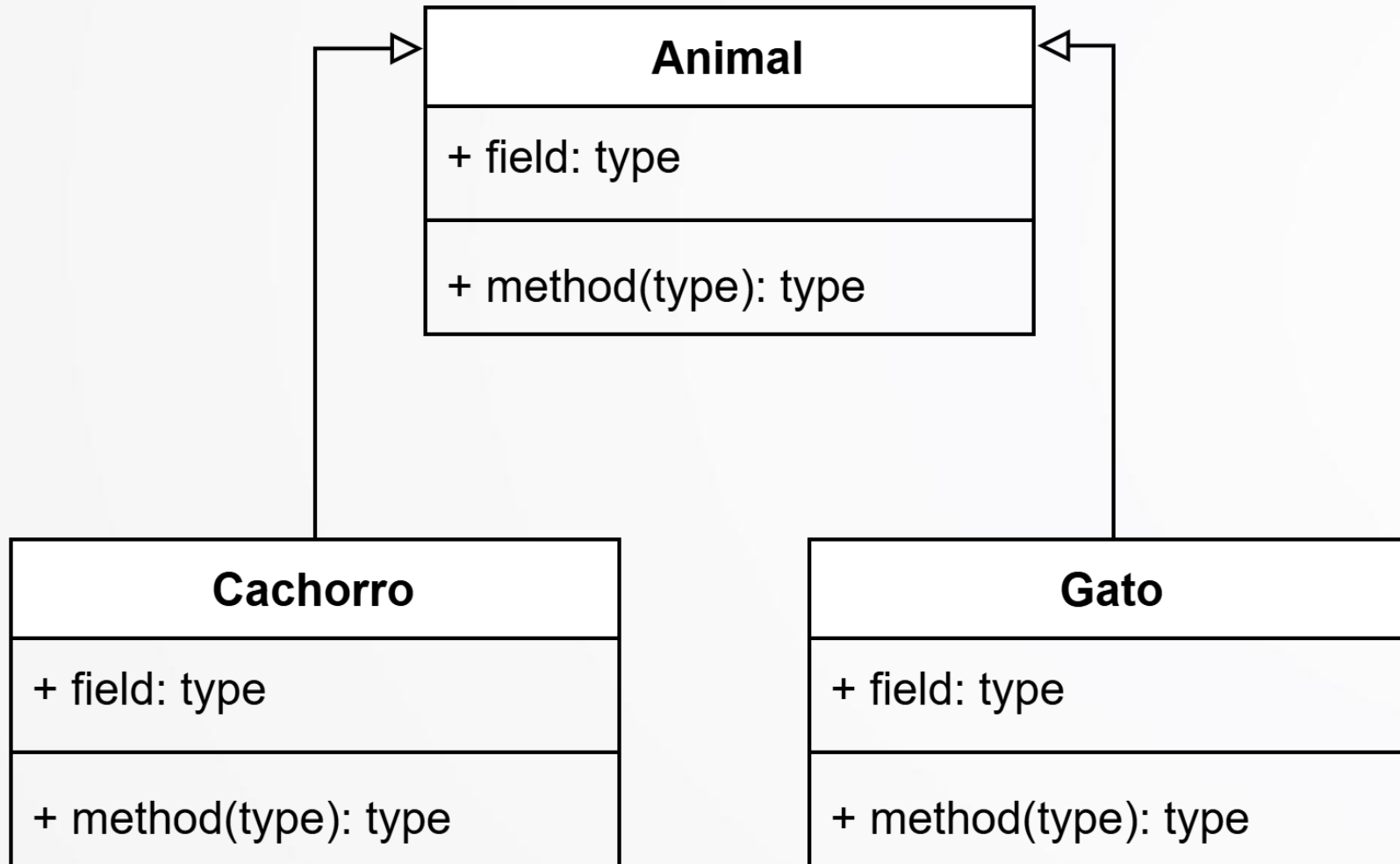
A herança é uma relação onde uma classe "filha" herda as características (atributos e métodos) de uma classe "pai". Essa relação é do tipo "é um" ou "é uma".

Notação: Uma linha com uma seta triangular preenchida ou vazado na direção da classe pai (superclasse).

Exemplo: A classe Animal pode ter subclasses como Cachorro e Gato, pois tanto Cachorro quanto Gato são tipos de Animal.

Senac

Herança (ou Generalização)



Dependência

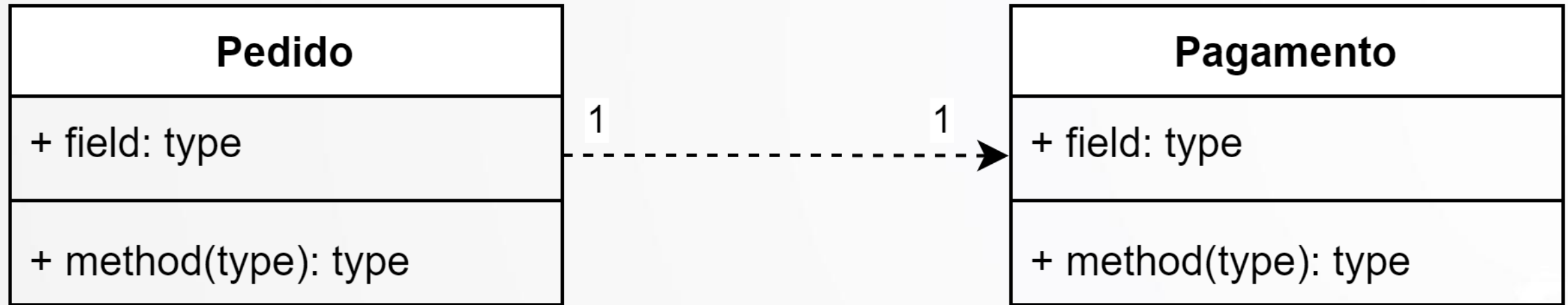
A dependência é um relacionamento mais fraco entre duas classes, onde uma classe usa outra temporariamente para realizar uma tarefa. Não é uma relação fixa, mas uma associação temporária que ocorre durante a execução de uma operação.

Notação: Uma linha tracejada com uma seta apontando para a classe da qual a outra depende.


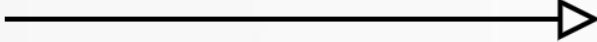


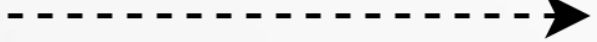
Exemplo: Uma classe Pedido pode depender de uma classe Pagamento, pois um Pedido usa Pagamento para processar uma transação.

Senac

Dependência



Símbolos

	Associação
	Herança
	Agregação
	Composição
	Dependência

Multiplicidade

A multiplicidade define o número de instâncias de uma classe que podem se relacionar com uma instância de outra classe em um diagrama de classes UML. Esse conceito é essencial para modelar a quantidade de elementos envolvidos em um relacionamento, como por exemplo, quantos funcionários uma empresa pode ter, ou quantos cursos um professor pode ministrar. A multiplicidade é expressa por números ou intervalos próximos ao relacionamento, indicando a quantidade mínima e máxima de instâncias permitidas para a associação.

Principais notações de multiplicidade

1 – Exatamente um:

Indica que a associação deve ter exatamente uma instância da classe relacionada.

Exemplo: Um Pessoa possui exatamente um CPF.

Notação: 1 ao lado do relacionamento.

0..1 – Zero ou um:

Representa que a associação é opcional e que pode haver no máximo uma instância da classe relacionada.

Exemplo: Um Cliente pode ter ou não um Endereço de Cobrança.

Notação: 0..1.

Principais notações de multiplicidade

*** – Muitos:**

Indica que pode haver qualquer número de instâncias (inclusive zero) da classe relacionada.

Exemplo: Um Autor pode ter escrito vários Livros, ou nenhum.

Notação: *.

1..* – Um ou mais:

Especifica que deve haver pelo menos uma instância da classe relacionada, mas que pode haver várias.

Exemplo: Uma Empresa deve ter pelo menos um Funcionário, mas pode ter muitos.

Notação: 1..*.

Principais notações de multiplicidade

n..m – Intervalo específico:

Define um intervalo específico para a multiplicidade, onde n é o mínimo e m o máximo de instâncias.

Exemplo: Uma Equipe pode ter de 2 a 5 Membros.

Notação: 2..5.

Exemplos de multiplicidade em relacionamentos

Relacionamento 1 para 1 (Exatamente um):

Uma Pessoa possui exatamente um CPF.

Notação: Pessoa 1 ----- 1 CPF

Significa que cada pessoa tem um único CPF e cada CPF pertence a uma única pessoa.

Relacionamento 1 para Muitos (Um ou mais):

Um Professor pode lecionar para um ou mais Alunos.

Notação: Professor 1 ----- * Aluno

Significa que cada professor pode ter vários alunos, mas cada aluno tem apenas um professor (nesse exemplo específico).

Exemplos de multiplicidade em relacionamentos

Relacionamento Muitos para Muitos:

Um Aluno pode estar matriculado em vários Cursos, e um Curso pode ter vários Alunos.

Notação: Aluno * ----- * Curso

Esse tipo de relacionamento é comum e muitas vezes modelado usando uma terceira classe (como Matricula) para representar a relação.

Relacionamento Zero ou Muitos (Opcional):

Uma Empresa pode ter nenhum ou vários Funcionários.

Notação: Empresa 0..* ----- 1 Funcionário

Significa que a empresa pode não ter funcionários ou pode ter vários, mas cada funcionário pertence a apenas uma empresa (nesse exemplo específico).

Exemplos de multiplicidade em relacionamentos

Relacionamento Intervalar:

Uma Equipe de projeto pode ter entre dois e cinco Membros.

Notação: Equipe 1 ----- 2..5 Membros

Define um intervalo exato de membros possíveis, permitindo entre 2 e 5 membros na equipe.

Importância da multiplicidade

A multiplicidade permite definir as restrições de quantidade em um relacionamento, contribuindo para uma modelagem mais precisa. Isso ajuda a:

- Definir a cardinalidade e limites de relacionamento.
- Especificar regras de negócio, como a necessidade de uma entidade ou a flexibilidade de existência.
- Modelar o comportamento esperado dos objetos na aplicação final.

Esses detalhes melhoram a compreensão do diagrama, ajudando os desenvolvedores e as equipes a implementarem o sistema de acordo com as expectativas e regras de negócio estabelecidas.