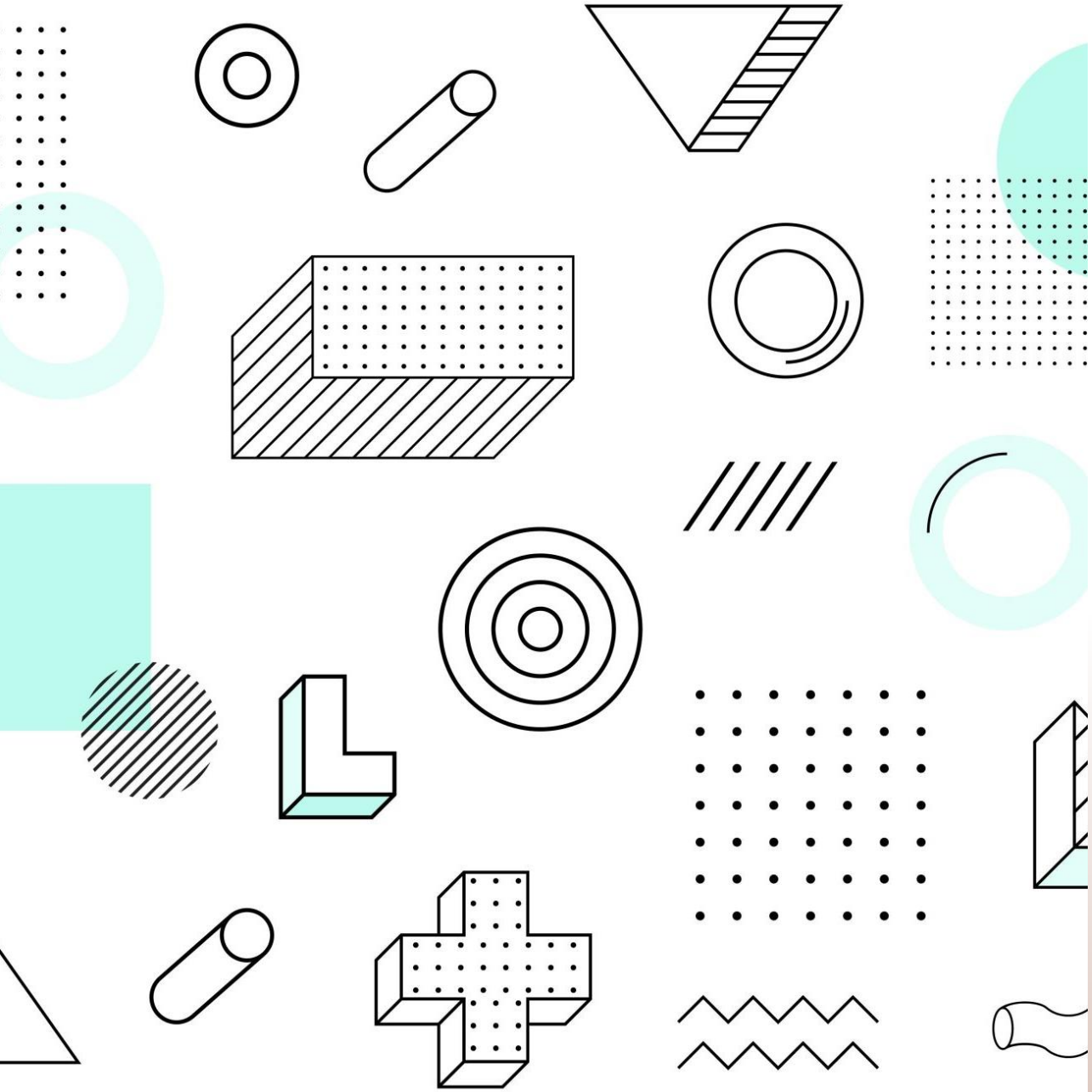


INTRODUÇÃO AOS PADRÕES DE PROJETO DE SOFTWARE

Design Patterns

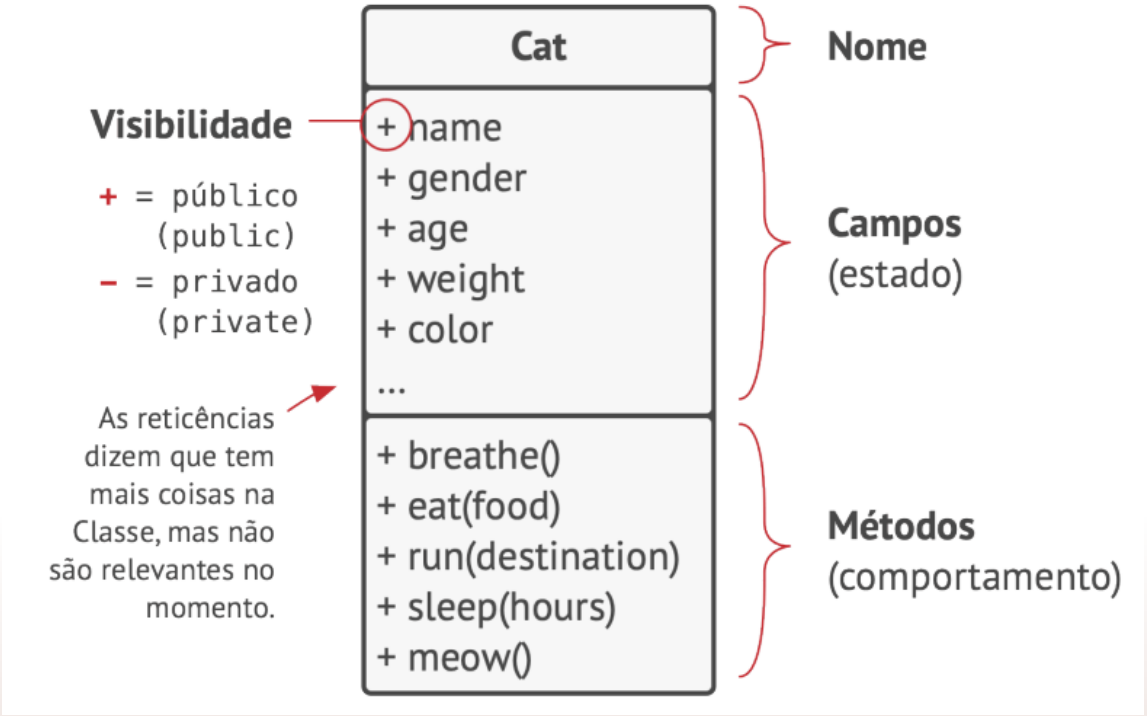
Prof. Me. Jefferson Passerini

BÁSICO DE POO

- A Programação Orientada à Objetos (POO) é um paradigma baseado no conceito de envolver pedaços de dados, e comportamentos relacionados aqueles dados, em uma coleção chamada objetos, que são construídos de um conjunto de “planos de construção”, definidos por um programador, chamados de classes.

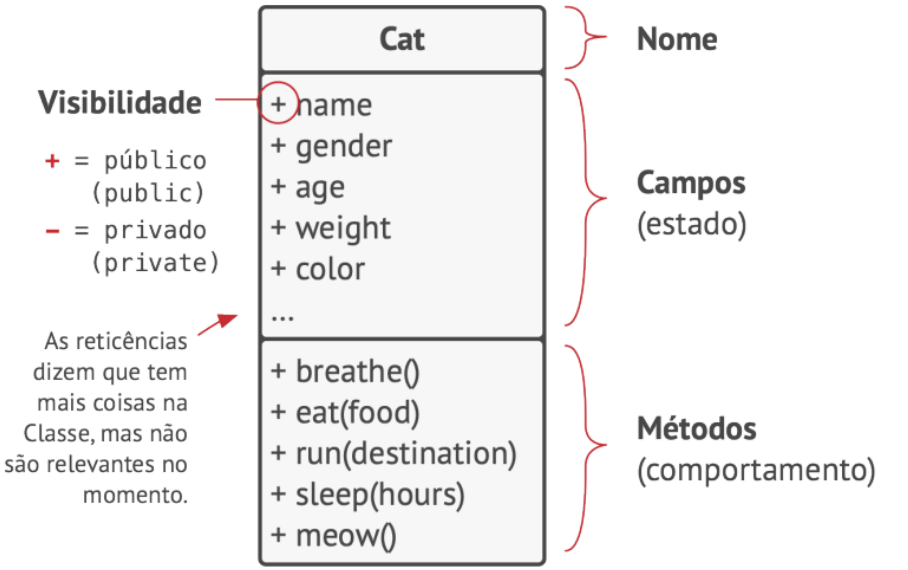
CONCEITOS

- Este é um diagrama UML da **classe Gato**.
- Pode-se identificar os atributos que definem as características de um gato;
- E os métodos que irão descrever os comportamentos de um gato.
- **É uma prática comum deixar os nomes dos membros e da classe nos diagramas em inglês, como faríamos em um código real. No entanto, comentários e notas também podem ser escritos em português.**



CONCEITOS

- Dados armazenados dentro dos campos do objeto são referenciados como estados, e todos os métodos de um objeto definem seu comportamento.
- A gata Nina é uma instância da classe Gato. Ela tem o mesmo conjunto de atributos que Tom. A diferença está nos valores destes seus atributos: seu gênero é fêmea, ela tem uma cor diferente, e pesa menos.
- Então uma classe é como uma planta de construção que define a estrutura para objetos, que são instâncias concretas daquela classe.



Tom: Cat	
name	"Tom"
sex	"macho"
age	3
weight	7
color	marrom
texture	listrada

Nina: Cat	
name	"Nina"
sex	"fêmea"
age	2
weight	5
color	cinza
texture	lisa

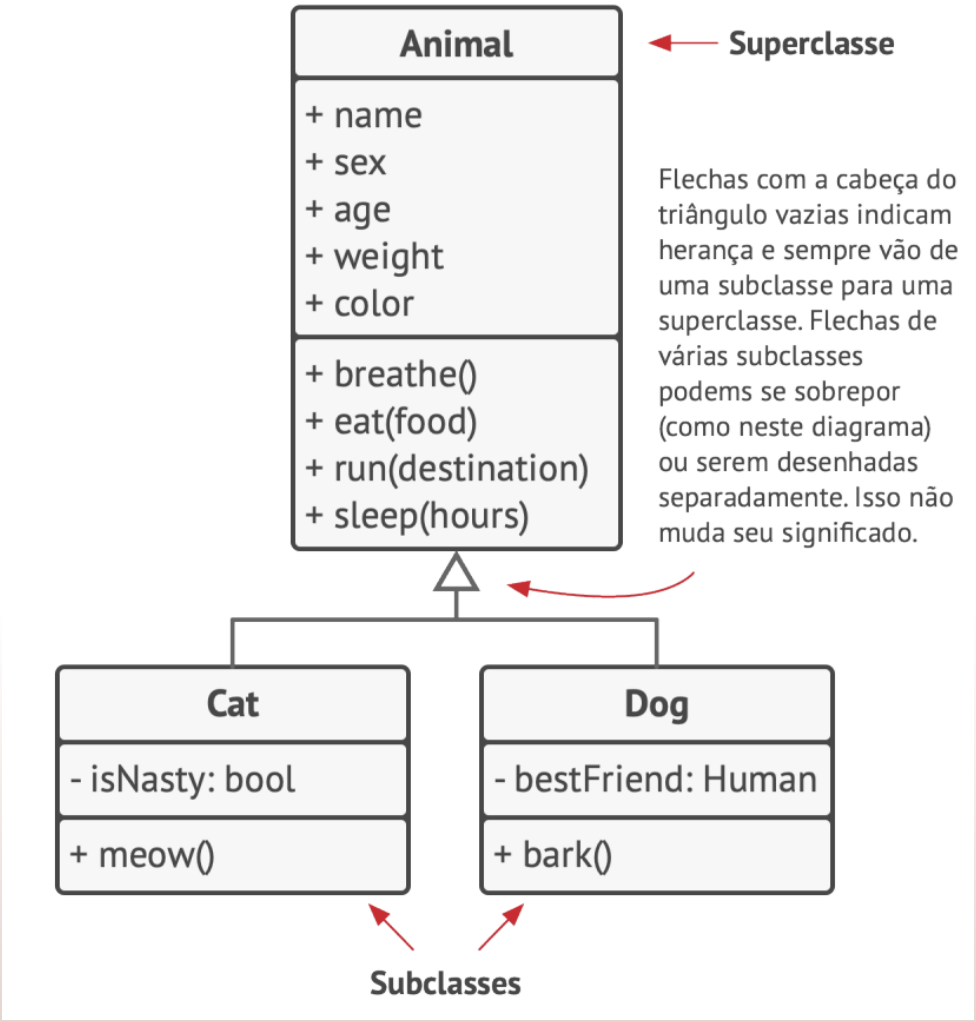
Objetos são instâncias de classes.

HIERARQUIA DE CLASSES

- Enquanto trabalhamos com apenas 1 classe está tudo bem!!!
- E se tivermos que incluir a descrição de cachorros ao nosso modelo anterior, esses animais tem atributos comuns com os gatos, possuem comportamentos semelhantes como correr, respirar e dormir.
- Mas também possuem características diferentes gatos miam e cachorros latem!!!
- Então parece que podemos definir a classe base Animal que listaria atributos e comportamentos em comum.

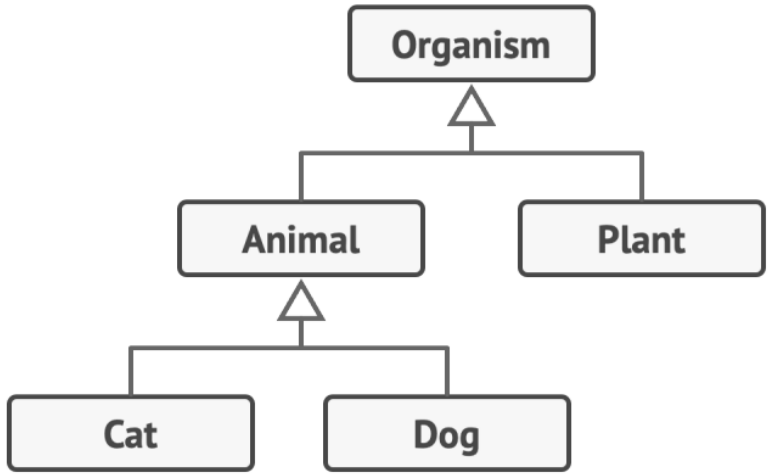
HIERARQUIA DE CLASSES

- Uma **classe mãe**, como a que recém definimos, é chamada de uma **superclasse**.
- Suas filhas são as subclasses. **Subclasses herdam estado e comportamento** de sua mãe, **definindo apenas atributos e comportamentos que diferem.**
- Portanto, a classe **Gato** teria o método **miado** e a classe **Cão** o método **latido**.



HIERARQUIA DE CLASSES

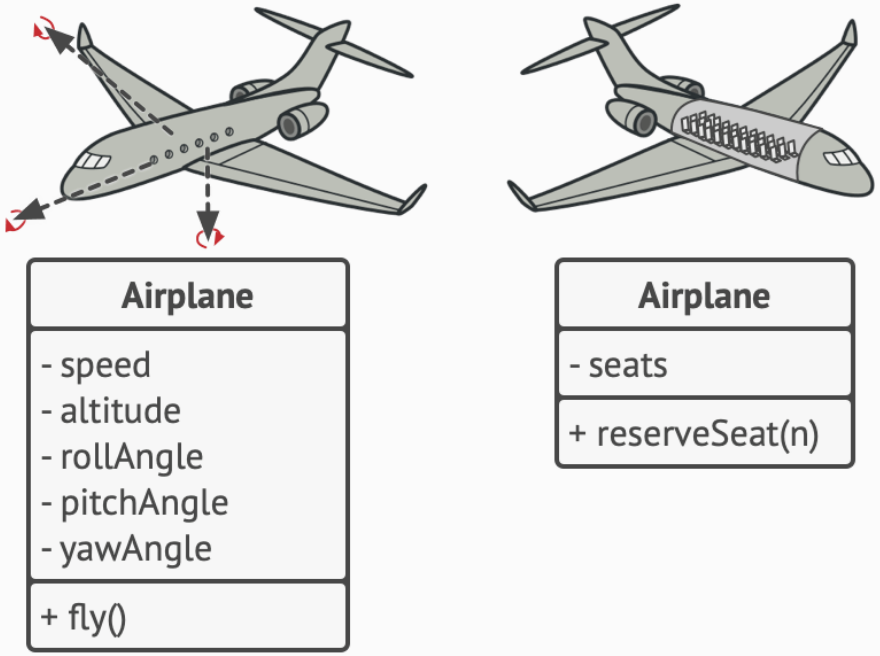
- Assumindo que temos um requisito de negócio parecido, podemos ir além e extrair uma classe ainda mais geral de todos os Organismos que será a superclasse para Animais e Plantas.
- Tal pirâmide de classes é uma hierarquia. Em tal hierarquia, a classe Gato herda tudo que veio das classes Animal e Organismos.
- Sub Classes podem sobrescrever o comportamento de métodos que herdaram de suas classes pais.
- Uma Sub Classe pode tanto substituir completamente o comportamento padrão ou apenas melhorá-lo com coisas adicionais.



Classes em um diagrama UML podem ser simplificadas se é mais importante mostrar suas relações que seus conteúdos.

PILARES DA POO

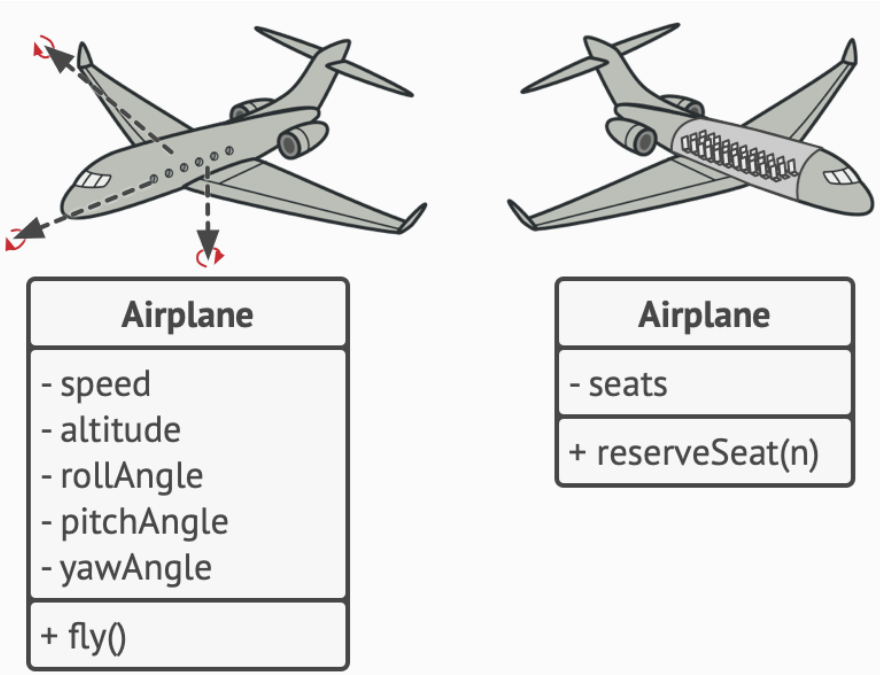
ABSTRAÇÃO



Diferentes modelos de um mesmo objeto real.

- Na maioria das vezes quando você está criando um programa com a POO, você molda os objetos do programa baseado em objetos do mundo real.
- Contudo, objetos do programa não representam os originais com 100% de precisão (e raramente é necessário que eles cumpram isso).
- Ao invés disso, seus objetos apenas modelam atributos e comportamentos de objetos reais em um contexto específico, ignorando o resto.

ABSTRAÇÃO



Diferentes modelos de um mesmo objeto real.

- Por exemplo, uma classe **Avião** poderia provavelmente existir em um simulador de voo e em uma aplicação de compra de passagens aéreas.
- Mas no primeiro caso, ele guardaria detalhes relacionados ao próprio voo, enquanto que no segundo caso você se importaria apenas com as poltronas disponíveis e os locais delas.
- A Abstração é um modelo de um objeto ou fenômeno do mundo real, limitado a um contexto específico, que representa todos os detalhes relevantes para este contexto com grande precisão e omite o resto.

ENCAPSULAMENTO

- Para ligar um motor de carro, você precisa apenas girar a chave ou apertar um botão. Você não precisa conectar os fios debaixo do capô, rotacionar o eixo das manivelas e cilindros, e iniciar o ciclo de força do motor.
- Estes detalhes estão embaixo do capô do carro. Você tem apenas uma **interface simples:** um interruptor de ignição, um volante, e alguns pedais. Isso ilustra como cada objeto tem um interface—uma parte pública do objeto, aberto a interações com outros objetos.
- **O Encapsulamento é a habilidade de um objeto de esconder parte de seu estado e comportamentos de outros objetos, expondo apenas uma interface limitada para o resto do programa.**

ENCAPSULAMENTO

- Encapsular alguma coisa significa torná-la privada, e portanto acessível apenas por dentro dos métodos da sua própria classe. Há um modo um pouco menos restritivo chamado protegido que torna um membro da classe disponível para subclasses também.
- Interfaces e classes/métodos abstratos da maioria das linguagens de programação são baseados em conceitos de abstração e encapsulamento.
- Em linguagens de programação modernas orientadas a objetos, o mecanismo de interface (geralmente declarado com a palavra-chave, interface ou protocolo) permite que você defina contratos de interação entre objetos.
- Esse é um dos motivos pelos quais as interfaces somente se importam com os comportamentos de objetos, e porque você não pode declarar um campo em uma interface.

ENCAPSULAMENTO

- O fato da palavra interface aparecer como a parte pública de um objeto enquanto que também temos o tipo interface na maioria das linguagens de programação pode ser bem confuso.
- Imagine que você tenha uma **interface TransporteAéreo** com um método voar(origem, destino, passageiros).
- Quando desenvolvendo um simulador de transporte aéreo você restringiu a **classe Aeroporto** para trabalhar apenas com objetos que implementam a i**nterface TransporteAéreo**.
- Após isso, você pode ter certeza que qualquer objeto passado para um objeto aeroporto, seja ela um **Avião**, um **Helicóptero**, ou um inesperado **Grifo Domesticado**, todos serão capazes de aterrissar ou decolar deste tipo de aeroporto.

ENCAPSULAMENTO

- Você pode mudar a implementação do método voar nessas classes de qualquer maneira que deseje.
- Desde que a assinatura do método permaneça a mesma que a declarada na interface, todas as instâncias da classe Aeroporto podem trabalhar com seus objetos voadores sem problemas.

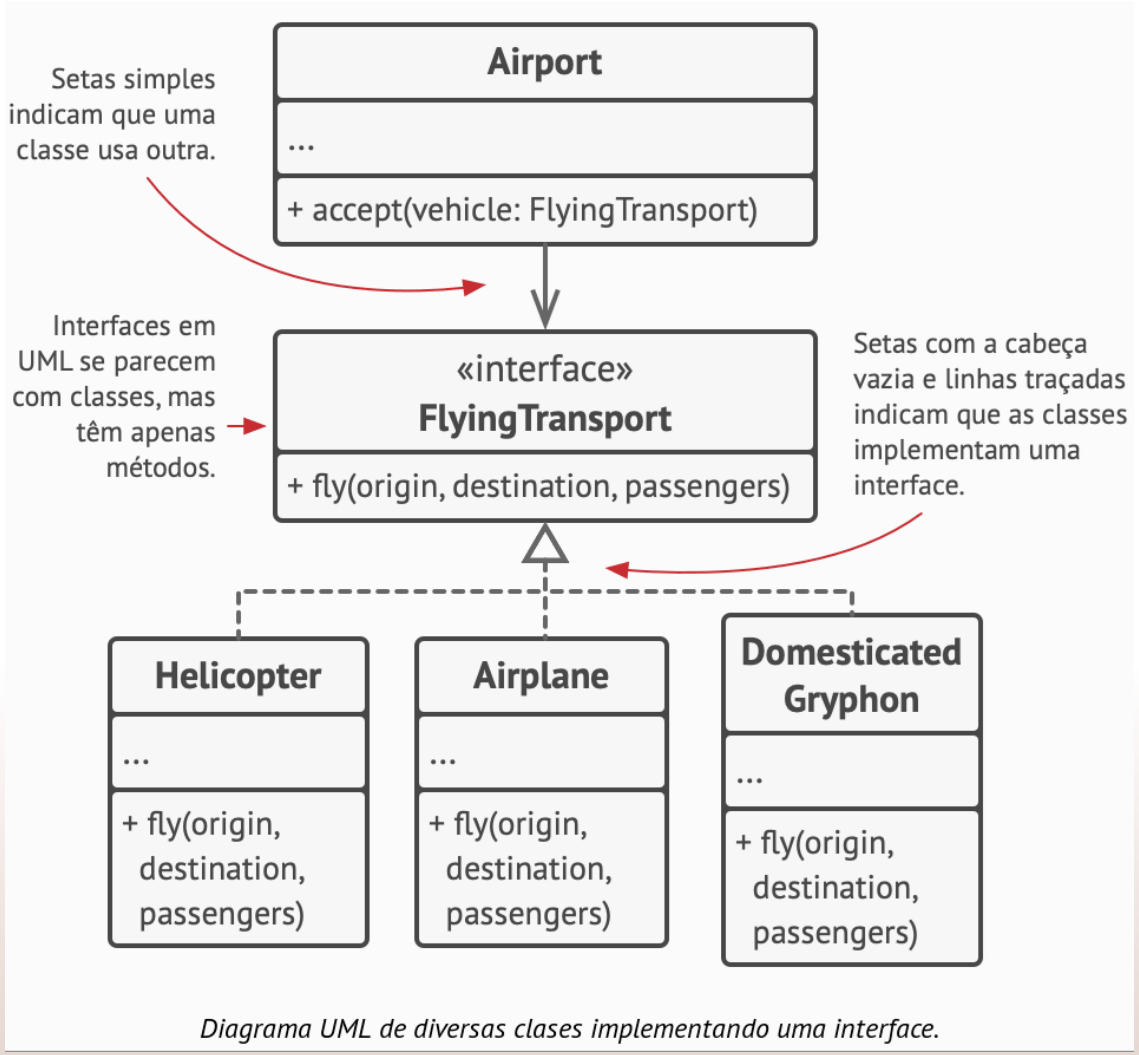


Diagrama UML de diversas clases implementando uma interface.

HERANÇA

- A Herança é a habilidade de construir novas classes em cima de classes já existentes.
- O maior benefício da herança é a reutilização de código.
- Se você quer criar uma classe que é apenas um pouco diferente de uma já existente, não há necessidade de duplicar o código.
- Ao invés disso, você estende a classe existente e coloca a funcionalidade adicional dentro de uma subclasse resultante, que herdará todos os campos de métodos da superclasse.

HERANÇA

- A consequência de usar a herança é que as subclasses têm a mesma interface que sua classe mãe.
- Você não pode esconder um método em uma subclasse se ele foi declarado na superclasse.
- Você deve também implementar todos os métodos abstratos, mesmo que eles não façam sentido em sua subclasse.
- Na maioria das linguagens de programação uma subclasse pode estender apenas uma superclasse (não permitem herança múltipla).
- Por outro lado, qualquer classe pode implementar várias interfaces ao mesmo tempo. Mas como mencionado anteriormente, se uma superclasse implementa uma interface, todas as suas subclasses também devem implementá-la.

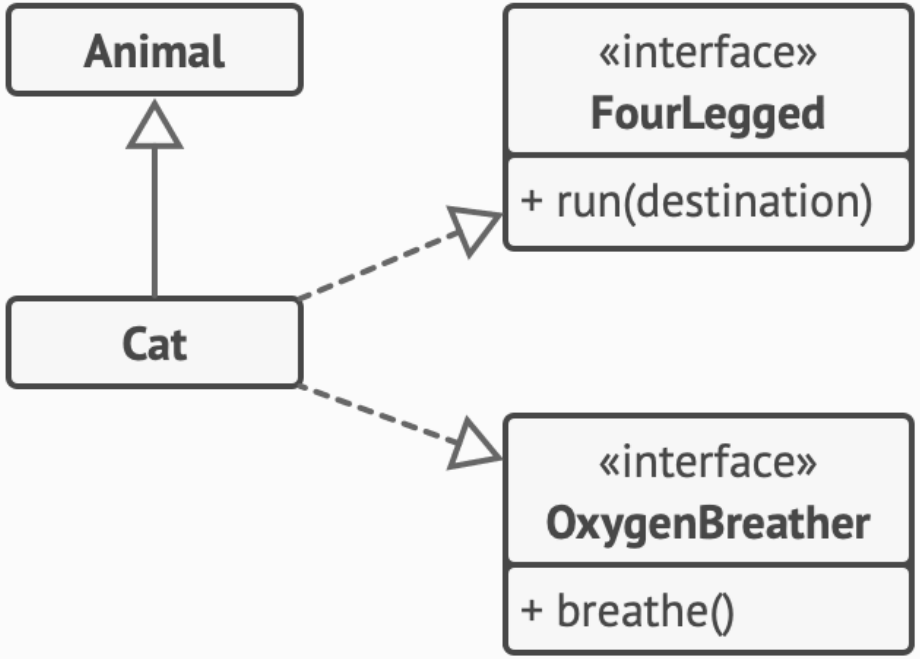
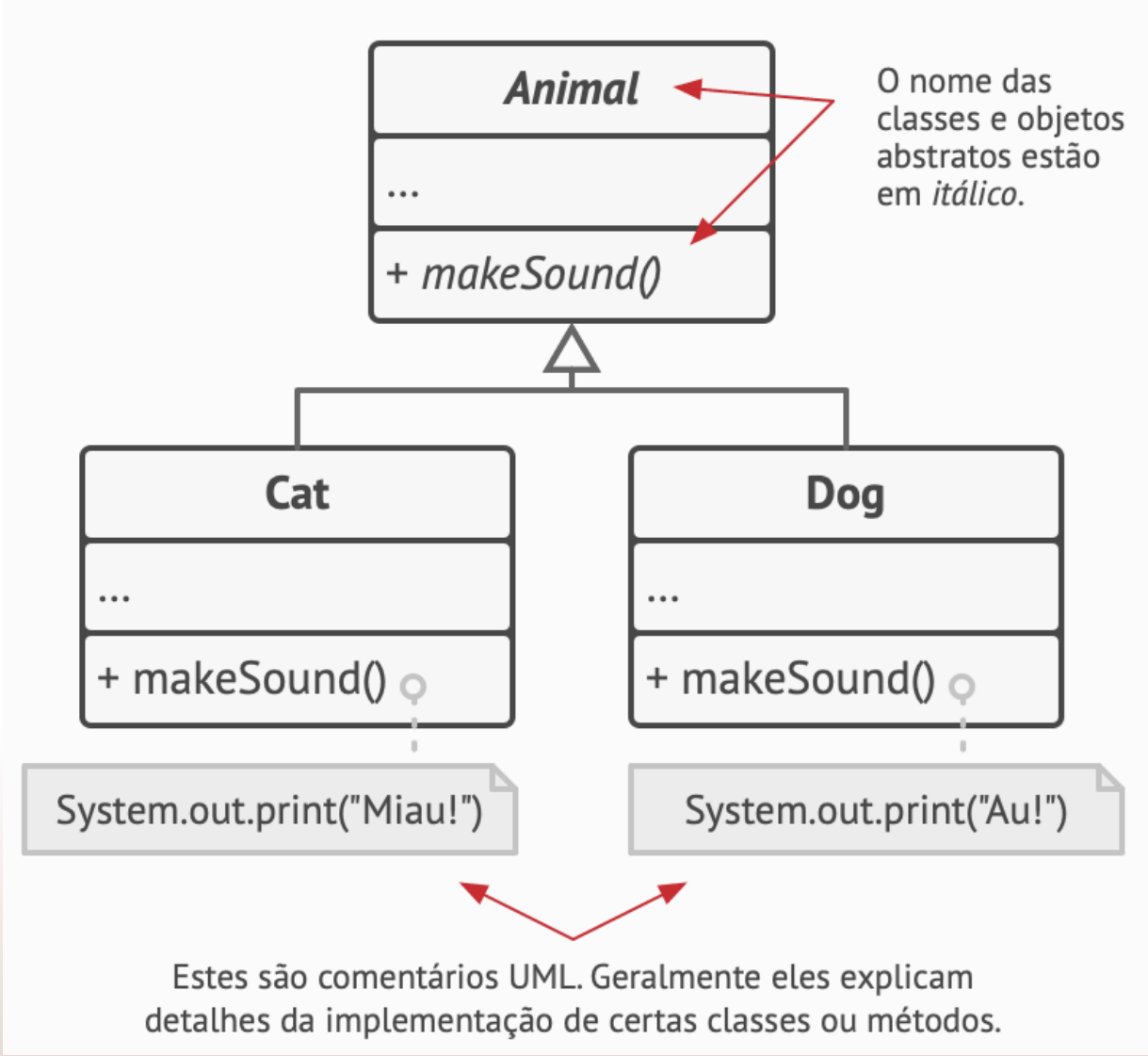


Diagrama UML de estender uma classe única versus implementar múltiplas interfaces ao mesmo tempo.

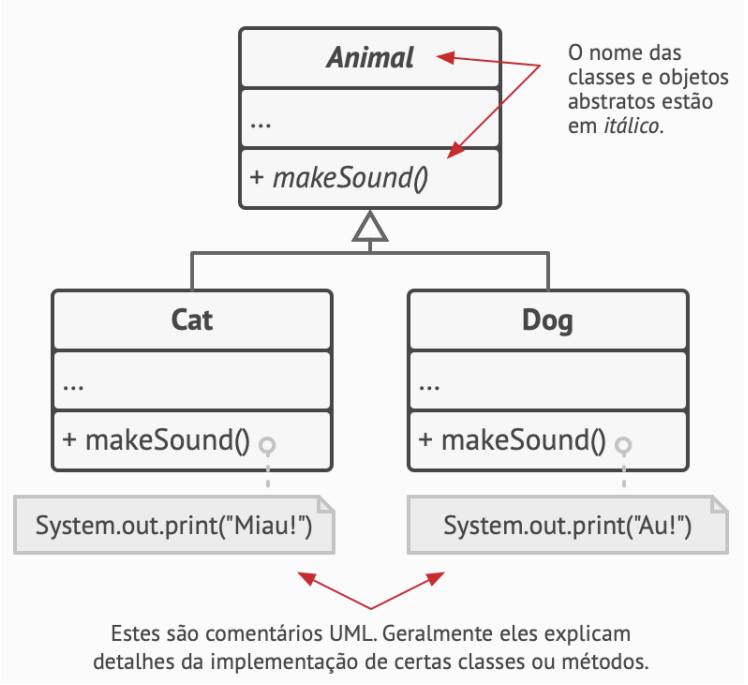
POLIMORFISMO

- Vamos ver alguns exemplos de animais. A maioria dos **Animais** podem produzir sons.
- Nós podemos antecipar que todas as sub-classes terão que sobrescrever o método base **produzirSom** para que cada subclasse possa emitir o som correto;
- portanto nós podemos declará-lo abstrato agora mesmo. Isso permite omitir qualquer implementação padrão do método na super-classe, mas força todas as subclasses a se virarem com o que têm.



POLIMORFISMO

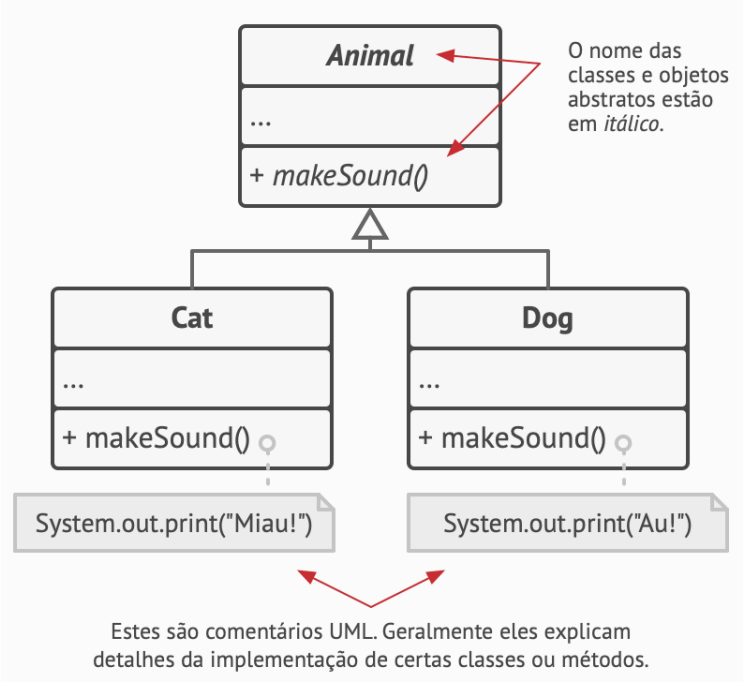
- Imagine que colocamos vários gatos e cães em uma bolsa grande.
- Então, com os olhos fechados, nós tiramos os animais um a um para fora da bolsa.
- Após tirarmos um animal da bolsa, nós não sabemos com certeza o que ele é.
- Contudo, se fizermos carícias no animal o suficiente, ele vai emitir um som de alegria específico, dependendo de sua classe concreta.




```
1  bolsa = [new Gato(), new Cão()];
2
3  foreach (Animal a : bolsa)
4    a.produzirSom()
5
6  // Miau!
7  // Au!
```

POLIMORFISMO

- O programa não sabe o tipo concreto do objeto contido dentro da variável a;
- Mas, graças ao mecanismo especial chamado polimorfismo, o programa pode rastrear a subclasse do objeto cujo método está sendo executado e executar o comportamento apropriado.
- O Polimorfismo é a habilidade de um programa detectar a classe real de um objeto e chamar sua implementação mesmo quando seu tipo real é desconhecido no contexto atual.
- Você também pode pensar no polimorfismo como a habilidade de um objeto “fingir” que é outra coisa, geralmente uma classe que ele estende ou uma interface que ele implementa. No nosso exemplo, os cães e gatos na bolsa estavam fingindo ser animais genéricos.




```
1  bolsa = [new Gato(), new Cão()];
2
3  foreach (Animal a : bolsa)
4    a.produzirSom()
5
6  // Miau!
7  // Au!
```

RELAÇÕES ENTRE OBJETOS

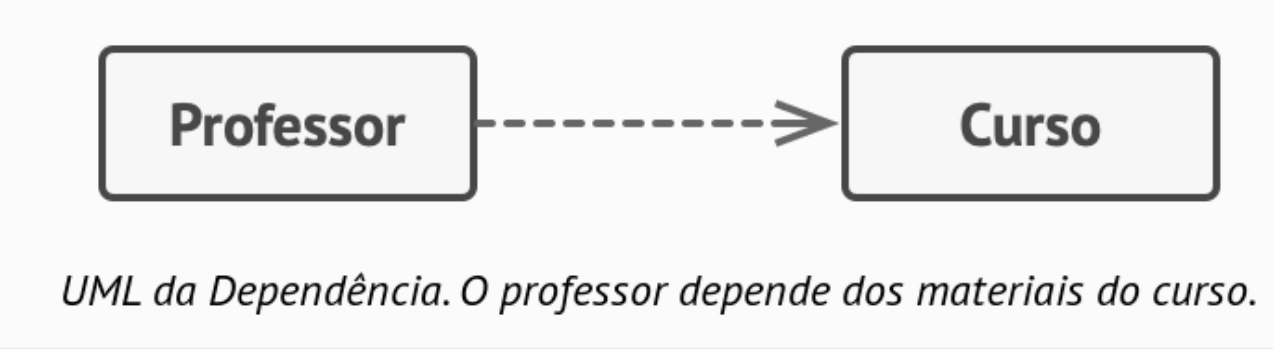
- Além da Herança temos outras relações entre objetos.

DEPENDÊNCIA

- A dependência é o mais básico e o mais fraco tipo de relações entre classes.
- Existe uma dependência entre duas classes se algumas mudanças na definição de uma das classes pode resultar em modificações em outra classe.
- A dependência tipicamente ocorre quando você usa nomes de classes concretas em seu código.
- Por exemplo, quando especificando tipos em assinaturas de métodos, quando instanciando objetos através de chamadas do construtor, etc. Você pode tornar a dependência mais fraca se você fazer seu código ser dependente de interfaces ou classes abstratas ao invés de classes concretas.

DEPENDÊNCIA

- Geralmente, um diagrama UML não mostra todas as dependências—há muitas delas em um código de verdade. Ao invés de poluir o diagrama com dependências, você pode ser seletivo e mostrar apenas aquelas que são importantes para o que quer que seja que você está comunicando.



UML da Dependência. O professor depende dos materiais do curso.

ASSOCIAÇÃO

- A associação é um relacionamento no qual um objeto usa ou interage com outro.
- Em diagramas UML, o relacionamento de associação é mostrado por uma seta simples desenhada de um objeto e apontada para outro que ele utiliza.
- A propósito, ter uma associação bidirecional é uma coisa completamente normal. Neste caso, a flecha precisa apontar para ambos.
- A associação pode ser vista como um tipo especializado de dependência, onde um objeto sempre tem acesso aos objetos os quais ele interage, enquanto que a dependência simples não estabelece uma ligação permanente entre os objetos.

UML da Associação. O professor se comunica com os alunos.

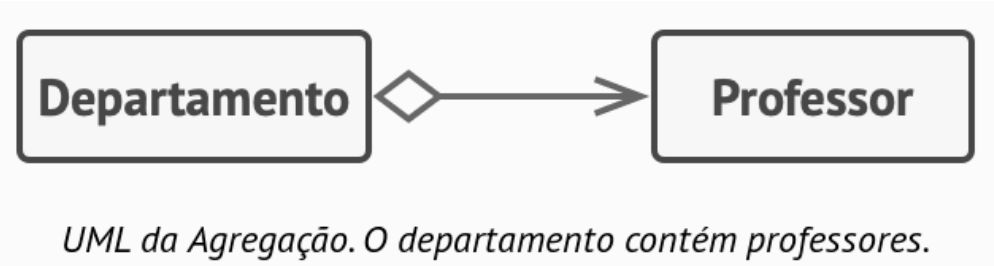
ASSOCIAÇÃO

- Para solidificar seu entendimento da diferença entre associação e dependência, vamos ver o exemplo combinado. Imagine que você tem uma classe Professor.
- ***Observe o método ensinar***. Ele precisa de um argumento da classe Curso, que então é usado no corpo do método. *Se alguém muda o método obterConhecimento* da classe Curso (altera seu nome, ou adiciona alguns parâmetros necessários, etc) nosso código irá quebrar. Isso é chamado de dependência.
- Agora olha para o campo aluno e como ele é usado no método ensinar. Nós podemos dizer com certeza que a classe Aluno é também uma dependência para a classe Professor: se o *método lembrar* mudar, o código da classe Professor irá quebrar.
- Contudo, uma vez que o campo aluno está sempre acessível para qualquer método do Professor, a classe Aluno não é apenas uma dependência, mas também uma associação.

```
1  class Professor is
2    field Student student
3    // ...
4    method teach(Course c) is
5      // ...
6      this.student.remember(c.getKnowledge())
```

AGREGAÇÃO

- A agregação é um tipo especializado de associação que representa relações individuais (one-to-many), múltiplas (many-to-many), e totais (whole-part) entre múltiplos objetos.
- *Geralmente, sob agregação, um objeto "tem" um conjunto de outros objetos e serve como um contêiner ou coleção.* O componente pode existir sem o contêiner e pode ser ligado através de vários contêineres ao mesmo tempo. No UML a relação de agregação é mostrada como uma linha e um diamante vazio na ponta do contêiner e uma flecha apontando para o componente.
- *Embora estejamos falando sobre relações entre objetos, lembre-se que o UML representa relações entre classes.* Isso significa que um objeto universidade pode consistir de múltiplos departamentos mesmo que você veja apenas um "bloco" para cada entidade no diagrama. A notação do UML pode representar quantidades em ambos os lados da relação, mas tudo bem omiti-las se as quantidades não participam do contexto.



UML da Agregação. O departamento contém professores.

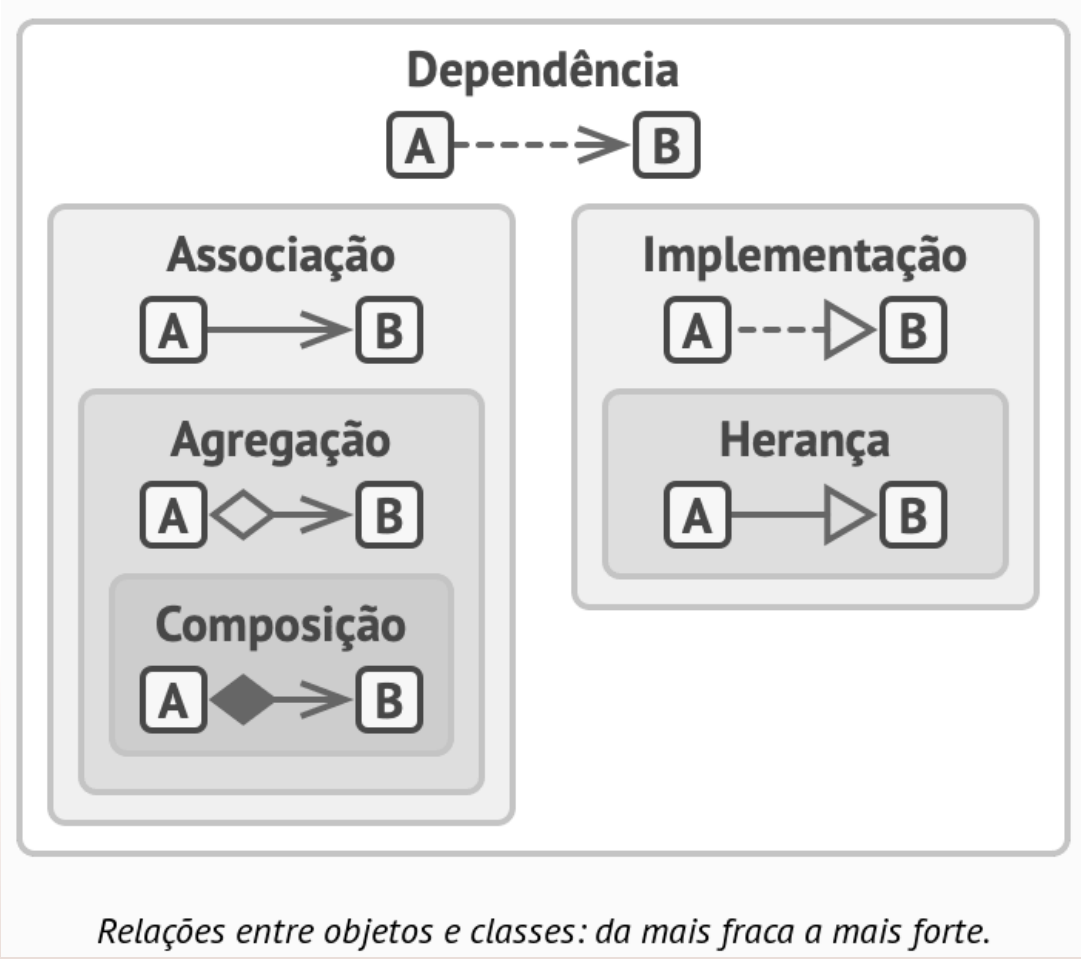
COMPOSIÇÃO

- A composição é um tipo específico de agregação, onde um objeto é composto de um ou mais instâncias de outro. ***A distinção entre esta relação e as outras é que o componente só pode existir como parte de um contêiner.*** No UML a relação de composição é desenhada do mesmo modo que para a agregação, mas com um diamante preenchido na base da flecha.
- Observe que, na língua Inglesa, muitas pessoas usam com frequência o termo “composition” (composição) quando realmente querem dizer tanto a agregação como a composição.
- O exemplo mais notório para isso é o famoso princípio ***“choose composition over inheritance” (escolha composição sobre herança).*** Não é porque as pessoas desconhecem a diferença, mas porque a palavra “composition” (por exemplo em “object composition”) soa mais natural para elas.

UML da Agregação. O departamento contém professores.

RESUMO

- ***Dependência***: Classe A pode ser afetada por mudanças na classe B.
- ***Associação***: Objeto A sabe sobre objeto B. Classe A depende de B.
- ***Agregação***: Objeto A sabe sobre objeto B, e consiste de B. Classe A depende de B.
- ***Composição***: Objeto A sabe sobre objeto B, consiste de B, e gerencia o ciclo de vida de B. Classe A depende de B.
- ***Implementação***: Classe A define métodos declarados na interface B. objetos de A podem ser tratados como B. Classe A depende de B.
- ***Herança***: Classe A herda a interface e implementação da classe B mas pode estendê-la. Objets de A podem ser tratados como B. Classe A depende de B.



Relações entre objetos e classes: da mais fraca a mais forte.