PROGRAMAÇÃO ORIENTADA A OBJETOS

AULA 4

Prof. Leonardo Gomes

CONVERSA INICIAL

Nesta aula, vamos abordar um princípio muito importante na orientação a objetos, a *herança*, princípio que permite que classes compartilhem atributos e métodos evitando retrabalho. Abordaremos também conceitos relacionados, como sobrecarga, e veremos maiores particularidades da linguagem Java no que diz respeito a essa relação.

Ao final desta aula, esperamos atingir os seguintes objetivos que serão avaliados ao longo da disciplina da forma indicada.

Quadro 1 – Objetivos da aula

Objetivos	Avaliação
1. Aplicar os conceitos de herança dentro da orientação a objetos	Questionário e questões dissertativas
2. Desenvolver algoritmos que fazem uso de múltiplas classes relacionadas por herança.	Questionário e questões dissertativas
3. Lidar com situações específicas como herança de construtores e sobrecarga de métodos.	Questionário e questões dissertativas

TEMA 1 – HERANÇA

Neste tema vamos debater o princípio da herança, que é um dos quatro pilares do paradigma orientado a objetos.

Não é raro nos depararmos com uma situação na qual desejamos representar classes que possuem semelhanças entre si e em que uma das classes pode ser entendida como baseada em outra. O conceito de herança atua nesse sentido, facilitando essa representação. Na vida real,

recebemos uma herança genética de nossos pais. Em parte, essa herança genética dita nossas características do que somos e do que conseguimos fazer, no entanto somos diferentes de nossos pais e possuímos características únicas. A herança na orientação a objetos se baseia nessa mesma ideia. É possível que uma classe (filha) herde comportamentos (métodos) e características (atributos) de outra classe (mãe), e nessa relação a classe filha pode ter suas características e comportamentos únicos também. De forma geral, a classe herdeira pode ampliar as funcionalidades da classe herdada e também modificar algumas das tais funcionalidades.

Com isso, podemos utilizar o conceito de *herança* para organizar uma classe de forma hierárquica como no exemplo a seguir.

Figura 1 – Representação hierárquica de classes

No alto da hierarquia, classes mais genéricas e abaixo, mais específicas. Com as classes específicas herdando características, comportamento e o próprio tipo da classe superior. Um celular é um tipo de telefone que, por sua vez, é um tipo de dispositivo eletrônico.

Essa relação hierárquica entre classes herdadas e herdeiras recebe diversos nomes na literatura, por exemplo:

Quadro 2 – Termos atribuídos à classe herdada e à classe herdeira

Classe herdada	Classe herdeira
Classe mãe	Classe filha
Superclasse	Subclasse
Classe base	Classe específica
Classe original	Classe derivada

Contextualmente esses termos podem ser utilizados dando uma contextualização diferente para cada abstração que seu código utiliza, mas representam a mesma relação hierárquica em termos práticos. No próximo tema vamos trabalhar com isso de forma mais prática e ver o conceito aplicado na linguagem Java. Vale reforçar que o conceito é o mesmo para qualquer linguagem moderna, pois mudam apenas alguns detalhes, por exemplo, certas linguagens permitem herança múltipla (uma classe com múltiplas superclasses) e outras não, mas há meios de se aplicar essa ideia de outras formas.

TEMA 2 – HERANÇA NA LINGUAGEM JAVA

Neste tema vamos analisar alguns exemplos de como a linguagem Java especificamente trata herança. Vamos iniciar analisando o código que consta na Figura 2. Suponha que desejamos implementar o registro de livros para um *site* de vendas. Para isso criamos a seguinte classe *Livro*.

Figura 2 – Exemplo de linguagem Java (1)

```
01.    package empresa.com;
02.    public class Livro {
03.
04.         public String autor;
05.         public float custoProducao;
06.         public float precoVenda;
07.         public String titulo;
08.         public int paginas;
09.
10.         //Cálculo do lucro
11.         public float lucro(){
12.              return (precoVenda - custoProducao);
13.         }
14.         //Cálculo do imposto de 20%
15.         public float imposto(){
16.              return (0.2f*this.lucro());
17.          }
18.         //Titulo do livro
19.         public void imprimirTitulo(){
20.              System.out.print("O titulo : "+ titulo);
21.         }
22.    }
```

A classe *Livro* possui atributos (linhas 4-8), um método de cálculo de lucro, baseado no custo de produção e valor da venda (linha 11) e também possui um método de cálculo de imposto, assumindo 20% sobre o lucro (linha 15) e por fim um método que imprime o título do livro (linha 19).

Porém suponha que nossa loja virtual passa a trabalhar com dois tipos de livros, os físicos e digitais. Ambos são livros, mas o livro digital possui certas características distintas que nossa classe *Livro* não possui. Na venda de livro digital existe *link* para *download* por exemplo que não faz sentido no livro físico. Então a implementação da classe LivroDigital fica como apresentada na Figura 3.

Figura 3 – Exemplo de linguagem Java (2)

```
01.    package empresa.com;
02.    public class LivroDigital {
03.         public String autor;
04.         public float custoProducao;
05.         public float precoVenda;
06.         public String titulo;
07.         public int paginas;
08.         public String linkDownload;
09.         public int tamanhoMB;
10.
11.         //Calculo do lucro
12.         public float lucro(){
13.              return (precoVenda - custoProducao);
14.         }
15.
16.         //Tamanho do arquivo por página
17.         public float tamanhoPorPagina(){
18.              return ((float)tamanhoMB/(float)paginas);
19.         }
20.
21.         //Titulo do livro
22.         public void imprimirTitulo(){
```

```
22.         public void imprimirTitulo(){
23.               System.out.print("O titulo : "+ titulo);
24.         }
25.
26.         //Calculo do imposto de 20% + R$ 2 por livro
27.         public float imposto(){
28.               return (0.2f*this.lucro() + 2);
29.         }
30.     }
```

A classe *LivroDigital* possui atributos (linhas 3-9) possui o mesmo método lucro (linha 12) e tem um método que calcula quantos Mb possui em média cada página do livro (linha 17). Outro método imprime o título do livro (linha 22) e, por fim, um método que faz o cálculo do imposto (linha 27).

Veja que, nesse exemplo, muito do código foi replicado, e se desejarmos efetuar uma alteração na representação dos nossos livros (por exemplo, em vez de simplesmente armazenar o nome do autor, criar um *array* para comportar múltiplos autores para um mesmo livro), essa alteração teria que ser feita duas vezes, gerando mais trabalho e duas vezes mais chance de erros por parte do programador. Na Figura 4, vemos como a classe *LivroDigital* seria representada utilizando herança da classe *Livro* na qual se baseia.

Figura 4 – Exemplo linguagem Java (3)

```
01.     package empresa.com;
02.     public class LivroDigital extends Livro {
03.           public String linkDownload;
04.           public int tamanhoMB;
05.
06.           //Tamanho do arquivo por página
07.           public float tamanhoPorPagina(){
08.                 return ((float)tamanhoMB/(float)paginas);
09.           }
10.           //Calculo do imposto de 20% + R$ 2 por livro
11.           public float imposto(){
12.                 return (0.2f*this.lucro() + 2);
13.           }
14.     }
```

Observe, na linha 2, a palavra reservada *extends* (significa *estender* no inglês). Essa palavra indica a herança na linguagem. A classe *LivroDigital* é filha, herdeira, e a classe *Livro* é a classe mãe, a classe herdada. Ou ainda, no linguajar proposto pelo Java, a classe *LivroDigital* é uma extensão da classe *Livro*. Todas as declarações de atributos e métodos que se repetem nas duas classes podem ser omitidas. Apenas o que é original da classe *LivroDigital* precisa ser declarado. Nessa situação, se desejarmos realizar a modificação de autor para um *array* de autores, bastaria uma única mudança que afetaria as duas classes, sem retrabalho.

Observe que existem duas situações ocorrendo na criação dos métodos:

1. O método *tamanhoPorPagina()* é um método exclusivo, uma adição. Basta declará-lo e utilizá-lo sem maiores problemas;

2. O método *imposto()*, por outro lado, já existia originalmente na classe Livro. O que acontece aqui é o que chamamos de uma *sobrescrita do método imposto()*. Para um objeto do tipo *LivroDigital*, o método sobrescrito será invocado em vez do método original.

Na Figura 5, vemos um código de exemplo utilizando as duas classes.

Figura 5 – Exemplo linguagem Java (4)

```
01.     package empresa.com;
02.     public class Teste {
03.          //Método principal, execução começa aqui.
04.          public static void main(String[] args){
05.               //Criação do livro e definição de
   atributos
06.               Livro livro1 = new Livro();
07.               livro1.autor = "Ignacio de Loyola";
08.               livro1.custoProducao= 9.5f;
09.               livro1.precoVenda= 19.99f;
10.               livro1.titulo ="O homem que odiava
   segunda-feira";
11.               livro1.paginas=100;
12.
13.               //Impressão na tela dos atributos
14.               System.out.println("Autor: " +
   livro1.autor +
    " Custo de producao: "+ livro1.custoProducao +" Preço: "
    + livro1.precoVenda +" Quantidade de páginas: "+
   livro1.paginas +" titulo: " + livro1.titulo +"\n");
15.               //Lucro e imposto

16.               System.out.println(" O livro " +
   livro1.titulo
    + " lucra por venda R$" + livro1.lucro() + " e paga em
    imposto $" + livro1.imposto());
17.               //Criação do Livro digital e atributos
18.               LivroDigital livro2 = new LivroDigital();
19.               livro2.autor = "Pierre Bayard";
20.               livro2.custoProducao= 15.0f;
21.               livro2.precoVenda= 34.99f;
22.               livro2.titulo ="Como falar dos livros que
   não
    lemos";
23.               livro2.paginas=200;
```

```
24.                 livro2.linkDownload="googleLivros";
25.                 livro2.tamanhoMB=4;
26.
27.                 //Impressão na tela dos atributos
28.                 System.out.println("Autor: " +
   livro2.autor +
     " Custo de producao: "+ livro2.custoProducao +" Preço:"
     + livro2.precoVenda +" Quantidade de páginas: "
     + livro2.paginas +" titulo: " + livro2.titulo +"\n");
29.                 //Lucro e imposto
30.                 System.out.println(" O livro " +
   livro1.titulo
    + " lucra por venda R$" + livro1.lucro() + " e paga em
    imposto $" + livro1.imposto());
31.                 //Tamanho da página
32.                 System.out.println("O livro " +
   livro2.titulo
    + " possui tamanho médio de página de " +
    livro2.tamanhoPorPagina() +" MB\n");
33.             }
34.     }
```

O código começa com a criação de um objeto *Livro* e o preenchimento dos seus atributos (linhas 6-11). Depois, os atributos são impressos na tela (linha 14). Em seguida, os métodos *lucro()* e *imposto()* são chamados (linha 16). Na sequência, a mesma coisa é feita para o *Livro Digital*: primeiro, ele é declarado e tem os atributos preenchidos (linhas 18-25); depois, os métodos *lucro(), imposto()* (linha 30) e *tamanhoPorPagina()* (linha 32) são chamados também. Observe que, no caso do método imposto(), foi sobreescrito pela classe filha LivroDigital. A versão do método da classe filha que será executada pois livro2 é uma instância de LivroDigital.

TEMA 3 – CONSTRUTORES E HERANÇA

Neste tema vamos discutir como trabalhar com os construtores e como estes se comportam com o uso de herança.

Os construtores funcionam de forma parecida com os métodos, pois são códigos executados no momento da instanciação. A aplicação mais comum dos construtores é a definição de valores para os

atributos no momento em que o objeto é instanciado. Relembrando o que já sabemos sobre o tema, para criar construtores, devemos fazê-lo semelhante a um método só que sem um parâmetro de retorno e com nome igual ao da classe.

Diferente dos métodos e atributos, os construtores não são herdados pelas classes filhas, mas podem ser invocados por elas. No código presente na Figura 6, vemos esse conceito do construtor sendo aplicado.

Figura 6 – Exemplo linguagem Java (5)

```
01.     class Base {
02.          int x;
03.          Base(){
04.               System.out.println("Construtor Base");
05.          }
06.          Base(int x) {
07.               this.x = x;
08.          }
09.     }
10.
11.     class Derivada extends Base {
12.          int y;
13.          Derivada(){
14.               System.out.println("Construtor
  Derivada");
15.          }
16.          Derivada(int x, int y) {
17.               super(x);
18.               this.y = y;
19.          }
20.          void exibir() {
21.               System.out.println("x = "+x+", y = "+y);
22.          }
23.     }
24.
25.     public class Teste {
```

```
26.
27.         public static void main(String[] args){
28.               Derivada obj1 = new Derivada();
29.               Derivada obj2 = new Derivada(10,50);
30.               obj2.exibir();
31.         }
32.     }
```

No código acima, colocamos três classes juntas para facilitar a visualização, porém, caso deseje implementar esse código para testes, lembre-se de colocar cada classe em um arquivo próprio ou fazer como foi feito no código acima, deixando apenas a classe com o método principal (Teste) como pública.

Observando o código, vemos que a classe Base (linha 1) implementa dois construtores, Base() (linha 3) que imprime uma mensagem na tela e Base (int) (linha 6) que inicializa o atributo x. Em seguida, temos a classe *Derivada* (linha 11), que é filha da classe Base, e que também possui dois construtores, Derivada() (linha 13) e apenas imprime uma mensagem, e Derivada(int,int) (linha 16), que inicializa os atributos x e y. Esta faz isso chamando o construtor da classe base, utilizando o comando *super* (linha 17). Veremos em detalhe o funcionamento dessa palavra reservada e seus usos na sequência.

Para ilustrar com outro exemplo, observe na Figura 7 o código para os construtores das classes *Livro* e *LivroDigital* que utilizamos anteriormente.

Figura 7 – Exemplo linguagem Java (6)

```
01.     public Livro(String autor, float custoProducao,
   float
   precoVenda, String titulo, int paginas) {
02.          this.autor = autor;
03.          this.custoProducao = custoProducao;
04.          this.precoVenda = precoVenda;
05.          this.titulo = titulo;
06.          this.paginas = paginas;
07.     }

08.     public LivroDigital(String autor, float
   0custoProducao, float precoVenda, String titulo, int
   paginas, String linkDownload, int tamanhoMB) {

09.
   super(autor,custoProducao,precoVenda,titulo,paginas);
10.      this.linkDownload = linkDownload;
11.      this.tamanhoMB = tamanhoMB;
12.     }
```

Na classe *Livro*, o construtor recebe todos os parâmetros e os inicializa normalmente. Já na classe *LivroDigital* o primeiro comando do construtor (linha 9) invoca o construtor da classe mãe e, na sequência, inicializa os atributos únicos da classe *LivroDigital*.

TEMA 4 – PALAVRAS RESERVADAS *SUPER* E *THIS*

Vamos agora debater dois comandos muito úteis para corrigir ambiguidades entre outras aplicações: o comando *super* e o *this*.

O comando *super* faz uma referência explícita à superclasse, à classe herdada, semelhante à palavra *this* que faz referência explícita à classe corrente. Observe, no código da Figura 6, que na classe *Base* declaramos o atributo x (linha 2) depois, no construtor (linha 6), recebemos um parâmetro x. Com o mesmo nome dentro do construtor quando escrevemos x, o compilador irá entender que se trata de uma referência para o parâmetro, mas se desejamos que a referência seja feita ao atributo, utilizamos *this*. O mesmo pode ser feito na classe filha, por exemplo: se de dentro da classe Derivada desejamos referenciar x de forma explícita, podemos fazer escrevendo *super.x*, mas observe que isso só é realmente necessário em caso de conflito de nomes entre variáveis, parâmetros e atributos. Se

esse conflito não existe, é possível referenciar diretamente apenas escrevendo *x*, por exemplo, sem utilizar *this* ou *super*.

Outro uso importante e bastante comum da palavra reservada *super* é para invocar o construtor da classe mãe, no código anterior (linha 17) (Figura 6) o construtor Derivada (int,int) chamada o construtor Base(int) passando x como parâmetro. Essa chamada de construtor só é possível exclusivamente nessa situação. Um construtor de classe filha invocando na primeira linha o construtor da classe mãe. Qualquer outro momento que esse código *super(int)* fosse invocado geraria um erro.

Ainda falando sobre o código anterior ao final temos uma classe Teste com o método principal, nele vemos sendo criado dois objetos da classe Derivada, *obj1* com construtor vazio (linha 28) e *obj2* com o construtor de dois inteiros (linha 29) (Figura 6). Ao executarmos temos o seguinte resultado.

>
Construtor classe base

Construtor classe derivada

x = 10, y = 50

Observe que o construtor da classe Base foi invocado. Isso acontece, pois quando não colocamos explicitamente a chamada de um construtor da classe mãe o construtor vazio da classe mãe é chamado de forma implícita na primeira linha do construtor da classe filha. Portanto, quando criamos o *obj1*, o Construtor *Derivada()* na primeira linha invoca *Base()* por isso vemos primeiro a mensagem do construtor base seguido do construtor derivada.

4.1 PALAVRA RESERVADA *INSTANCEOF*

Além de herdar atributos, a tipagem também é herdada. Ou seja, se temos uma classe *Base* e outra *Derivada* herdeira de *Base*, então objetos da classe Derivada também são do tipo Base. Em outras palavras, as classes filhas são consideradas do tipo da classe mãe também. Existe um comando chamando *instanceof* (*instância de*, em tradução do inglês), que é utilizado justamente para identificar se uma determinada instância pertence a determinada classe, ele retorna *true* ou *false* (verdadeiro ou falso) caso seja ou não uma instância. No código da Figura 8, por exemplo, será impresso *true* três vezes como resposta.

Figura 8 – Exemplo linguagem Java (7)

```
01.    class Animal {}
02.    class Mamifero extends Animal {}
03.    class Reptil extends Animal {}
04.    public class Cachorro extends Mamifero {
05.
06.       public static void main(String args[]) {
07.          Animal a = new Animal();
08.          Mamifero m = new Mamifero();
09.          Cachorro c = new Cachorro();
10.
11.          System.out.println(m instanceof Animal);
12.          System.out.println(c instanceof Mamifero);
13.          System.out.println(c instanceof Animal);
14.       }
15.    }
```

No código acima criamos quatro classes: *Animal*, que é a classe mãe; *Reptil* e *Mamifero*, filhas; e Cachorro, por sua vez, filha de *Mamifero* (linhas 1 até 4). Instanciamos objetos dessas classes e comparamos se a instância de *Mamifero* também é do tipo *Animal*, e se a instância de *Cachorro* também é do tipo *Mamifero* e *Animal*. Para as três verificações, a resposta é *verdadeiro* e, portanto, o código responde com *true*.

TEMA 5 – HERANÇA E UML

Vamos debater novamente o diagrama de classes UML. Aagora que definimos o conceito de herança, podemos debater as principais relações que são representadas por meio dessa modelagem.

Como podemos observar, as classes possuem relacionamentos entre si. Por vezes, elas compartilham informações, se comunicam e colaboram uma com as outras. Dentre os principais tipos de relacionamento, temos:

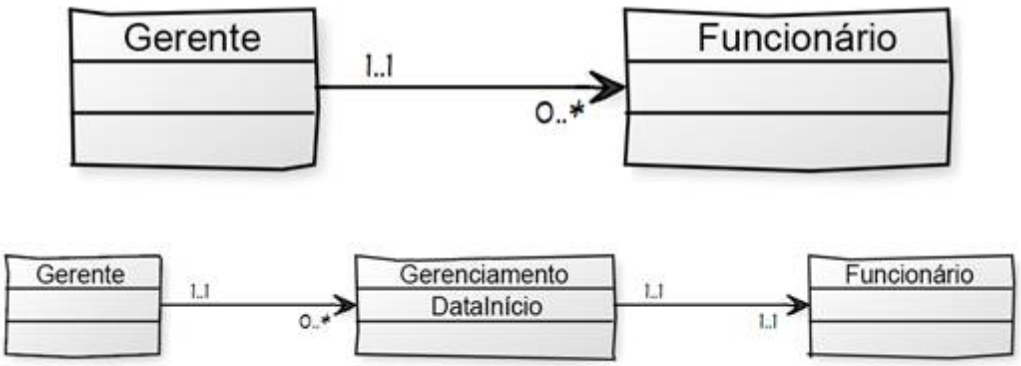
- Associação;
- Agregação;
- Composição;

- Herança;
- Dependência.

5.1 ASSOCIAÇÃO

Associação é um cenário bastante comum, em que existe um vínculo entre as classes. No exemplo da Figura 9 temos uma associação entre as classes *Gerente* e *Funcionário*, em que um gerente pode gerenciar diversos funcionários, e um funcionário só pode ser gerenciado por um único gerente. Pode ser pertinente nesse tipo de relação criar uma classe intermediária caso essa relação precise armazenar informações adicionais, por exemplo, seu sistema necessita saber desde que dia tal funcionário passou a ser responsabilidade de determinado gerente.

Figura 9 – Exemplo de associação



É possível representar também quantos elementos existem na relação. Chamamos isso de *multiplicidade da relação*.

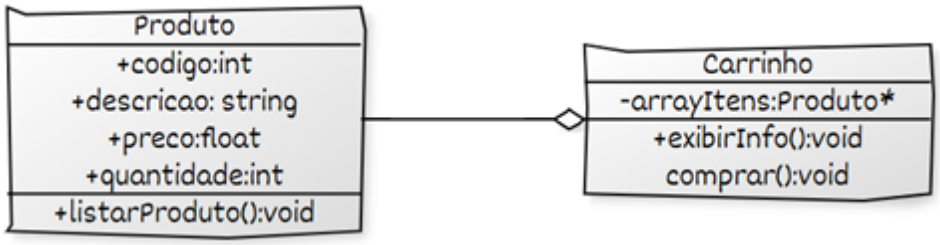
Quadro 3 – Multiplicidade da relação

Código	Descrição
0..1	0 ou 1 objeto, não é obrigatório e no máximo um único objeto da classe na relação.
1..1	Apenas 1 objeto, nunca mais ou menos.
0..*	Muitos objetos. De zero até um número qualquer de objetos na relação
1..*	Pelo menos um. Podem existir mais objetos na relação mais ao menos 1.
2..4	Na presença de valores específicos a relação está limitada aos valores apresentados.

5.2 AGREGAÇÃO

Agregação é um tipo especial de associação em que a temos uma classe que representa o todo e outra classe que representa a parte. Por exemplo, quando fazemos compras *online*, é comum os produtos escolhidos irem para um carrinho virtual e, no final da compra, fecharmos o pedido dos itens do carrinho. Ao implementar esse sistema, poderíamos ter a classe *Carrinho* (todo) e a classe *Produto* (parte). Na agregação, faz sentido existirem as partes mesmo sem o todo. Se a classe *Carrinho* não existisse, a classe *Produto* ainda faria sentido para ser utilizada em outros contextos, para, por exemplo, modificar o preço dos itens.

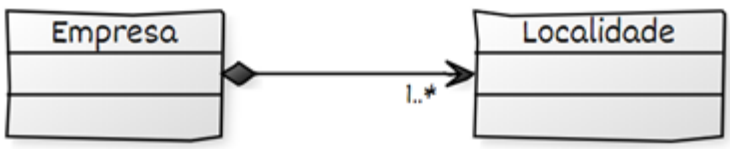
Quadro 4 – Exemplo de agregação



5.3 COMPOSIÇÃO

Composição pode ser entendida como uma variação da agregação, pois também representa uma relação de todo-parte, no entanto a relação aqui é mais próxima entre o todo e a parte, sendo que a parte não faz sentido sem o todo, pois o todo cria e destrói as partes. Por exemplo, uma classe que represente uma empresa (todo) pode ter uma classe chamada *Localidade* (parte). Na classe *Empresa*, as *Localidades* são criadas, modificadas e destruídas conforme convém e existem apenas para compor a classe *Empresa*. Assim, não faz sentido no sistema em questão olhar uma localidade separadamente da empresa.

Figura 10 – Exemplo de composição

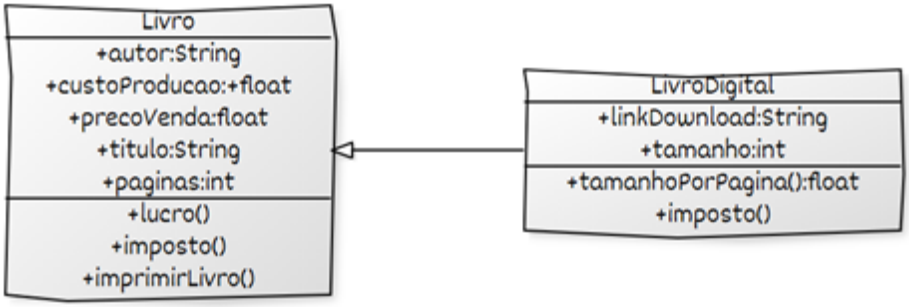


5.4 HERANÇA

A *herança* é outra relação representada pelo UML e serve para indicar qual é a classe geral e a classe especializada. É importante observar que os métodos da classe superclasse são acessados pelas

subclasses também. A relação é representada por um triângulo vazio. A imagem do Quadro 5 exemplifica essa relação com a classe *Livro* e *Livro Digital*, que temos utilizado como exemplo ao longo da aula.

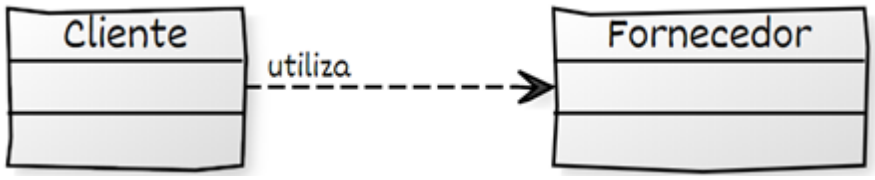
Quadro 5 – Exemplo de relação Herança



5.5 DEPENDÊNCIA

Dependência é outra relação importante que pode ser representada pelo UML. Essa relação indica simplesmente a dependência para compilar de uma classe pela outra. Por exemplo, uma classe A utiliza internamente objetos de outra classe B, e para um código assim compilar é necessário incluir a definição dessa outra classe B no código da primeira classe A. Assim, temos uma relação em que a classe A depende da classe B. Na Figura 11, temos um exemplo prático de uma classe *Cliente* que utiliza métodos da classe *Fornecedor* para realizar alguma tarefa, portanto *depende* da classe *Fornecedor*.

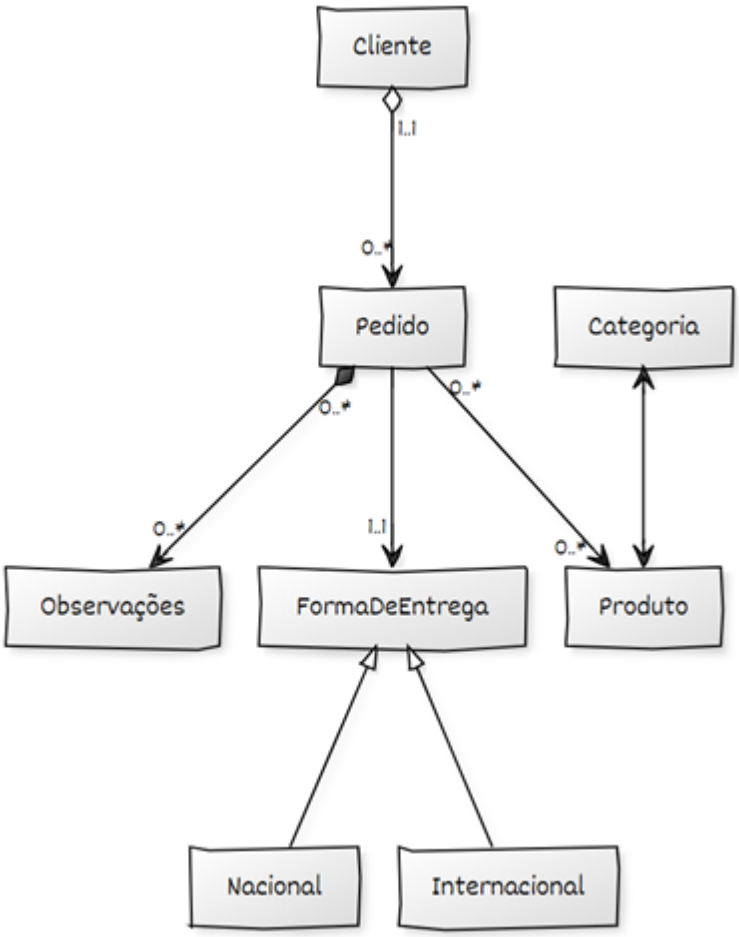
Figura 11 – Exemplo de dependência



Os diagramas UML são bastante versáteis e servem para auxiliar a comunicação em uma equipe de desenvolvimento, trazendo uma descrição visual do que é desejado de uma implementação. Não é necessário utilizar todos os recursos do UML simultaneamente, uma vez que ela é uma ferramenta que deve ser útil aos seus propósitos e só se faz necessário descrever com ela o que se deseja demonstrar. Existem diversas ferramentas que geram diagramas levando em conta as implementações e também o contrário: cria classes com atributos e o escopo dos métodos,

baseando-se em um modelo UML. Na Figura 12, temos mais um exemplo de uma modelagem UML envolvendo mais classes e combinando diversas relações distintas.

Figura 12 – Diagrama UML



FINALIZANDO

Nesta aula, introduzimos o conceito da *herança*, um dos pilares da programação a objetos, o que permite que diferentes classes compartilhem métodos e atributos evitando retrabalho de programarmos a mesma coisa para diversas classes que possuem escopo parecido. O conceito consiste de uma abstração em que temos uma superclasse com as principais definições e uma ou mais subclasses que se baseiam nela trazendo apenas as modificações particulares.

Iniciamos o texto discutindo o conceito de herança de forma geral, as diferentes terminologias. Na sequência, apresentamos como funciona sua aplicação na linguagem Java com alguns exemplos. Logo após, discutimos a utilização de construtores junto ao conceito de herança e debatemos também as palavras reservadas *super*, *this* e *instanceof*. Por fim, discutimos de forma mais completa os diagramas de classe UML e as várias relações que ele representa, incluindo herança.

REFERÊNCIAS

BARNES, D. J.; KÖLLING, M. **Programação orientada a objetos com Java**. 4. ed. São Paulo: Pearson Prentice Hall, 2009.

DEITEL, P.; DEITEL, H. **Java**: como programar. 10. ed. São Paulo: Pearson, 2017.

LARMAN, C. **Utilizando UML e padrões**: uma Introdução à análise e a projeto orientados a objetos e ao desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2007.

MEDEIROS, E. S. de. **Desenvolvendo *software* com UML 2.0**: definitivo. São Paulo: Pearson Makron Books, 2004.

PAGE-JONES, M. **Fundamentos do desenho orientado a objetos com UML**. São Paulo: Makron Book, 2001

PFLEEGER, S. L. Engenharia de *software*: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2007.

SINTES, T. **Aprenda programação orientada a objetos em 21 dias**. 5. reimp. São Paulo: Pearson Education do Brasil, 2014.

SOMMERVILLE, I. **Engenharia de *software***. 9. ed. São Paulo: Pearson, 2011.