1.2 O paradigma da orientação a objetos

Indispensável ao desenvolvimento atual de sistemas de software é o *paradigma da orientação a objetos*. Esta seção descreve o que esse termo significa e justifica por que a orientação a objetos é importante para a modelagem de sistemas. Pode-se começar pela definição da palavra *paradigma*. Essa palavra possui diversos significados, mas o que mais se aproxima do sentido aqui utilizado encontra-se no dicionário Aurélio Século XXI: ***paradigma***. [Do gr. *parádeigma*, pelo lat. tard. *paradigma*.] S. m. Termo com o qual Thomas Kuhn designou as realizações científicas (p. ex., a dinâmica de Newton ou a química de Lavoisier) que geram modelos que, por período mais ou menos longo e de modo mais ou menos explícito, orientam o desenvolvimento posterior das pesquisas exclusivamente na busca da solução para os problemas por elas suscitados.

Para o leitor que ainda não se sentiu satisfeito com essa definição, temos aqui outra, mais concisa e apropriada ao contexto deste livro: *um paradigma é uma forma de abordar um problema*. Como exemplo, considere a famosa história da maçã caindo sobre a cabeça de Isaac Newton, citado na definição anterior.[1] Em vez de pensar que somente a maçã estava caindo sobre a Terra, Newton também considerou a hipótese de o próprio planeta também estar caindo sobre a maçã! Essa outra maneira de abordar o problema pode ser vista como um paradigma.

Pode-se dizer, então, que o termo "paradigma da orientação a objetos" é uma forma de abordar um problema. Há alguns anos, Alan Kay, um dos pais do paradigma da orientação a objetos, formulou a chamada "analogia biológica". Por meio dela, ele imaginou um sistema de software que funcionasse como um ser vivo. Nesse sistema, cada "célula" interagiria com outras células através do envio de mensagens com o objetivo realizar um objetivo comum. Além disso, cada célula se comportaria como uma unidade autônoma.

De uma forma mais geral, Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si. Ele estabeleceu então os seguintes princípios da orientação a objetos:

1. Qualquer coisa é um objeto.
2. Objetos realizam tarefas por meio da requisição de serviços a outros objetos.
3. Cada objeto pertence a uma determinada *classe*. Uma classe agrupa objetos similares.
4. A classe é um repositório para comportamento associado ao objeto.

5. Classes são organizadas em hierarquias.

Vamos ilustrar esses princípios com a seguinte história: suponha que alguém queira comprar uma pizza. Chame este alguém de João. Ele está muito ocupado em casa e resolve pedir sua pizza por telefone. João liga para a pizzaria e faz o pedido, informando ao atendente (digamos, José) seu nome, as características da pizza desejada e o seu endereço. José, que só tem a função de atendente, comunica a Maria, funcionária da pizzaria e responsável por preparar as pizzas, qual pizza deve ser feita. Quando Maria termina de fazer a pizza, José chama Antônio, o entregador. Finalmente, João recebe a pizza desejada das mãos de Antônio meia hora depois de tê-la pedido.

Pode-se observar que o objetivo de João foi atingido graças à colaboração de diversos agentes, os funcionário da pizzaria. Na terminologia do paradigma da orientação a objetos, esses objetos são denominados *objetos*. Há diversos objetos na história (1º princípio): João, Maria, José, Antônio. Todos colaboram com uma parte e o objetivo é alcançado quando todos trabalham juntos (2º princípio). Além disso, o comportamento esperado de Antônio é o mesmo esperado de qualquer entregador. Diz-se que Antônio é um objeto da classe Entregador (3º princípio). Um comportamento comum a todo entregador, não somente a Antônio, é o de entregar a mercadoria no endereço especificado (4º princípio). Finalmente, José, o atendente, é também um ser humano, mamífero, animal etc. (5º princípio).

Mas o que o paradigma da orientação a objetos tem a ver com a modelagem de sistemas? Antes da orientação a objetos, outro paradigma era utilizado na modelagem de sistemas: o paradigma estruturado. Nesse paradigma, os elementos são dados e processos. Os processos agem sobre os dados para que um objetivo seja alcançado. Por outro lado, no paradigma da orientação a objetos há um elemento, o objeto, uma unidade autônoma que contém seus próprios dados que são manipulados pelos processos definidos para o objeto e que interage com outros objetos para alcançar um objetivo. É o paradigma da orientação a objetos que os seres humanos utilizam no cotidiano para a resolução de problemas. Uma pessoa atende a mensagens (requisições) para realizar um serviço; essa mesma pessoa envia mensagens a outras para que elas realizem serviços. Por que não aplicar essa mesma forma de pensar à modelagem de sistemas?

O paradigma da orientação a objetos visualiza um sistema de software como uma coleção de agentes interconectados chamados *objetos*. Cada objeto é responsável por realizar tarefas específicas. Para cumprir com algumas das tarefas sob sua responsabilidade, um objeto pode ter que interagir com outros objetos. É pela interação entre objetos que uma tarefa computacional é realizada.

Pode-se concluir que a orientação a objetos, como técnica para modelagem de sistemas, diminui a diferença semântica entre a realidade sendo modelada e os modelos construídos. Este livro descreve o importante papel da orientação a objetos na modelagem de sistemas de software atualmente. Explícita ou implicitamente, as técnicas de modelagem de sistemas aqui descritas utilizam os princípios que Alan Kay estabeleceu há mais de trinta anos. As seções a seguir continuam descrevendo os conceitos principais da orientação a objetos.

Um sistema de software orientado a objetos consiste em objetos em colaboração com o objetivo de realizar as funcionalidades desse sistema. Cada objeto é responsável por tarefas específicas. É graças à cooperação entre objetos que a computação do sistema se desenvolve.

1.2.1 Classes e objetos

O mundo real é formado de coisas. Como exemplos dessas coisas, pode-se citar um cliente, uma loja, uma venda, um pedido de compra, um fornecedor, este livro etc. Na terminologia de orientação a objetos, essas coisas do mundo real são denominadas *objetos*.

Seres humanos costumam agrupar os objetos. Provavelmente realizam esse processo mental de agrupamento para tentar gerenciar a complexidade de entender as coisas do mundo real. De fato, é bem mais fácil entender a ideia *cavalo* do que entender todos os cavalos que existem. Na terminologia da orientação a objetos, cada ideia é denominada *classe de objetos*, ou simplesmente *classe*. Uma classe é uma descrição dos atributos e serviços comuns a um grupo de objetos. Sendo assim, pode-se entender uma classe como sendo um *molde* a partir do qual objetos são construídos. Ainda sobre terminologia, diz-se que um objeto é uma *instância* de uma classe.

5 destacadores

Por exemplo, quando se pensa em um cavalo, logo vem à mente um animal de quatro patas, cauda, crina etc. Pode ser que algum dia você veja dois cavalos, um mais baixo que o outro, um com cauda maior que o outro, ou mesmo, por um acaso infeliz, um cavalo com menos patas que o outro. No entanto, você ainda terá certeza de estar diante de dois cavalos. Isso porque a *ideia* (classe) cavalo está formada na mente dos seres humanos, independentemente das pequenas diferenças que possa haver entre os *exemplares* (objetos) da ideia cavalo.

É importante notar que uma classe é uma *abstração* das características de um grupo de coisas do mundo real. Na maioria das vezes, as coisas do mundo real são muito complexas para que *todas* as suas características sejam representadas em uma classe. Além disso, para fins de modelagem de um sistema, somente um subconjunto de características pode ser relevante. Portanto, uma classe representa uma abstração das características *relevantes* do mundo real.

Finalmente, é preciso atentar para o fato de que alguns textos sobre orientação a objetos (inclusive este livro!) utilizam os termos *classe* e *objeto* de maneira equivalente para denotar uma classe de objetos.

1.2.2 Operação, mensagem e estado

Dá-se o nome de ***operação*** a alguma ação que um objeto sabe realizar quando solicitado. De uma forma geral, um objeto possui diversas operações. Objetos não executam suas operações aleatoriamente. Para que uma operação em um objeto seja executada, deve haver um estímulo enviado a esse objeto. Se esse objeto for visto como uma entidade ativa que representa uma abstração de algo do mundo real, então faz sentido dizer que ele pode responder aos estímulos que lhe são enviados (assim como faz sentido dizer que seres vivos reagem aos estímulos que recebem). Seja qual for a origem do estímulo, quando ele ocorre diz-se que o objeto em questão está recebendo uma ***mensagem*** requisitando que ele realize alguma operação. Quando se diz na

terminologia de orientação a objetos que *objetos de um sistema estão trocando mensagens* significa que esses objetos estão enviando mensagens uns aos outros com o objetivo de realizar alguma tarefa dentro do sistema no qual eles estão inseridos.

Por definição, o ***estado*** de um objeto corresponde ao conjunto de valores de seus atributos em um dado momento. Uma mensagem enviada a um objeto tem o potencial de mudar o estado desse objeto. Por exemplo, o recebimento de uma mensagem em um objeto da classe `ContaBancaria` solicitando o saque de determinada quantia possivelmente irá alterar o valor do atributo desse objeto relativo ao saldo da conta, o que é o mesmo que dizer que esse objeto está mudando de estado. Por outro lado, uma mensagem enviada para solicitar o valor atual do saldo não altera o estado desse objeto.

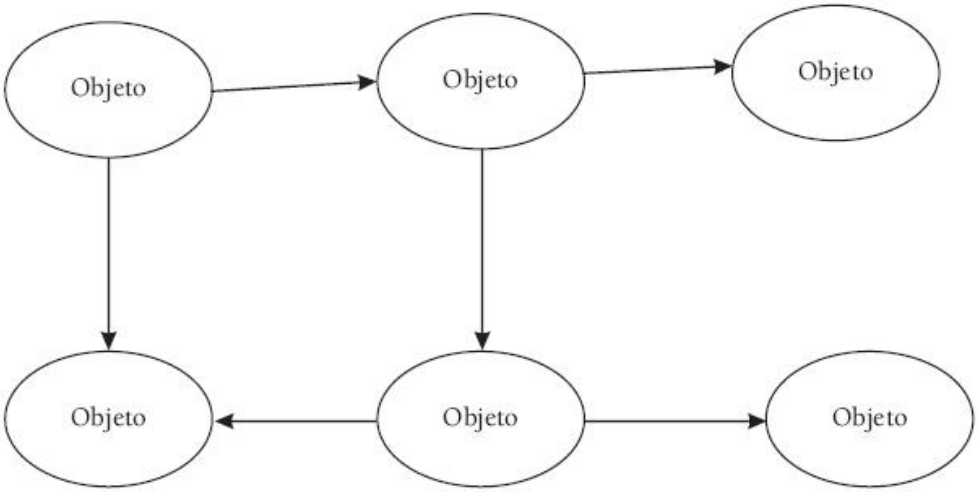

Figura 1-1: Objetos interagem através do envio de mensagens.

1.2.3 O papel da abstração na orientação a objetos

Nesta seção, apresentamos os principais conceitos do paradigma da orientação a objetos. Discutimos também o argumento de que todos esses conceitos são, na verdade, a aplicação de um único conceito mais básico, o *princípio da abstração*.

Primeiramente vamos descrever o conceito de abstração. A abstração é um processo mental pelo qual nós, seres humanos, nos atemos aos aspectos mais importantes (relevantes) de alguma coisa, ao mesmo tempo em que ignoramos os menos importantes. Esse processo mental nos permite gerenciar a complexidade de um objeto, enquanto concentramos nossa atenção nas características essenciais do mesmo. Note que uma abstração de algo é dependente

da perspectiva (contexto) sobre a qual uma coisa é analisada: o que é importante em um contexto pode não ser importante em outro. Nas próximas seções, descrevemos alguns conceitos fundamentais da orientação a objetos e estabelecemos sua correlação com o conceito de abstração.

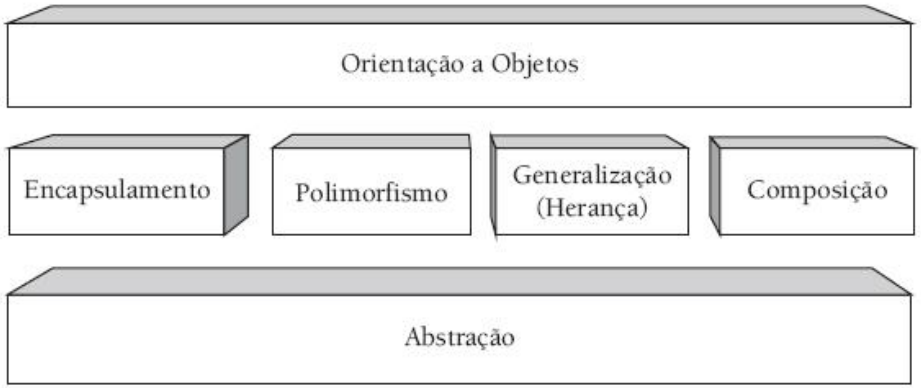

Figura 1-2: Princípios da orientação a objetos podem ser vistos como aplicações de um princípio mais básico, o da abstração.

1.2.3.1Encapsulamento

Objetos possuem *comportamento*. O termo comportamento diz respeito a operações realizadas por um objeto, a medida que ele recebe mensagens. O mecanismo de *encapsulamento* é uma forma de restringir o acesso ao comportamento interno de um objeto. Um objeto que precise da colaboração de outro objeto para realizar alguma operação simplesmente envia uma mensagem a este último. Segundo o mecanismo do encapsulamento, o *método* que o objeto requisitado usa para realizar a operação não é conhecido dos objetos requisitantes. Em outras palavras, o objeto remetente da mensagem não precisa conhecer a forma pela qual a operação requisitada é realizada; tudo o que importa a esse objeto remetente é obter a operação realizada, não importando como.

Certamente o remetente da mensagem precisa conhecer quais operações o receptor sabe realizar ou que informações este objeto receptor pode fornecer. Para tanto, a *classe* de um objeto descreve o seu comportamento. Na terminologia da orientação a objetos, diz-se que um objeto possui uma *interface* (ver Figura 1-3). Em termos bastante simples, a interface de um objeto corresponde ao que ele conhece e ao que sabe fazer, sem, no entanto, descrever *como* conhece ou faz. Se visualizarmos um objeto como um provedor de serviços, a interface de um objeto define os serviços que ele pode fornecer. Consequentemente, a interface de um objeto também define as mensagens que ele está apto a receber e a responder. Um serviço definido na interface de um objeto pode ter várias formas de *implementação*. Mas, pelo encapsulamento, a implementação de um serviço requisitado não importa ou não precisa ser

conhecida pelo objeto requisitante.

Porque existe o princípio do encapsulamento, a única coisa que um objeto precisa saber para pedir a colaboração de outro objeto é conhecer a interface deste último. Nada mais. Isso contribui para a autonomia dos objetos, pois cada um envia mensagens aos outros para realizar certas operações, sem se preocupar em *como* se realizaram as operações.

Note também que, de acordo com o encapsulamento, a implementação de uma operação pode ser trocada sem que o objeto requisitante da mesma precise ser alterado. Não posso deixar de enfatizar a importância desse aspecto no desenvolvimento de softwares. Se a implementação de uma operação pode ser substituída por outra sem alterações nas regiões do sistema que precisam dessa operação, há menos possibilidades de propagações de mudanças. No desenvolvimento de softwares modernos, nos quais os sistemas se tornam cada vez mais complexos, é fundamental manter as partes de um sistema tão independentes quanto possível. Daí a importância do mecanismo do encapsulamento no desenvolvimento de softwares orientados a objetos.

E qual a relação entre os conceitos de encapsulamento e abstração? Podemos dizer que o encapsulamento é uma aplicação do conceito de abstração. A aplicação da abstração, neste caso, está em esconder os detalhes de funcionamento interno de um objeto. Voltando ao contexto de desenvolvimento de softwares, note a consequência disso sobre a produtividade do desenvolvimento. Se pudermos utilizar os serviços de um objeto sem precisarmos entender seus detalhes de funcionamento, é claro que a produtividade do desenvolvimento aumenta.

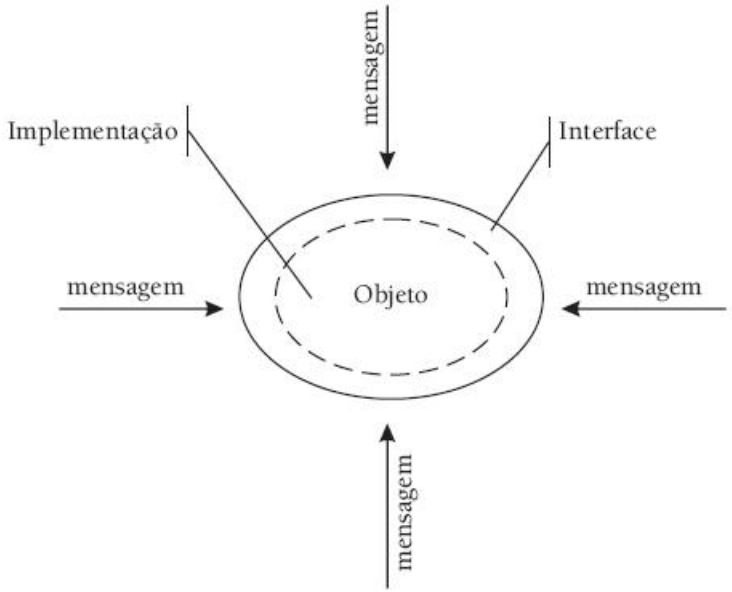

Figura 1-3: Princípio do encapsulamento: visto externamente, o objeto é a sua interface.

1.2.3.2 Polimorfismo

O polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface. Há algum tempo, o controle remoto de meu televisor quebrou. (Era realmente enfadonho ter de levantar para desligar o aparelho ou trocar de canal.) Um tempo depois, comprei um videocassete do mesmo fabricante de meu televisor. Para minha surpresa, o controle remoto do videocassete também funcionava para o televisor. Esse é um exemplo de aplicação do *princípio do polimorfismo* na vida real. Nesse caso, dois objetos do mundo real, o televisor e o aparelho de videocassete, respondem à mesma mensagem enviada.

E no que diz respeito à orientação a objetos, qual é a importância e quais são as consequências do polimorfismo? Nesse contexto, o polimorfismo se refere à capacidade de duas ou mais classes de objetos responderem à mesma mensagem, cada qual de seu próprio modo. O exemplo clássico do polimorfismo em desenvolvimento de software é o das formas geométricas. Pense em uma coleção de formas geométricas que contenha círculos, retângulos e outras formas específicas. Pelo princípio do polimorfismo, quando uma região de código precisa desenhar os elementos daquela

coleção, essa região não deve precisar conhecer os tipos específicos de figuras existentes; basta que cada elemento da coleção receba uma mensagem solicitando que desenhe a si próprio. Note que isso simplifica a região de código cliente (ou seja, a região de código que solicitou o desenho das figuras). Isso porque essa região de código não precisa conhecer o tipo de cada figura. Ao mesmo tempo, essa região de código não precisa ser alterada quando, por exemplo, uma classe correspondente a um novo tipo de forma geométrica (uma reta, por exemplo) tiver que ser adicionada. Esse novo tipo deve responder à mesma mensagem (solicitação) para desenhar a si próprio, muito embora implemente a operação a seu modo.

Note mais uma vez que, assim como no caso do encapsulamento, a abstração também é aplicada para obter o polimorfismo: um objeto pode enviar a *mesma* mensagem para objetos semelhantes, mas que implementam a sua interface de formas diferentes. O que está se abstraindo aqui são as diferentes maneiras pelas quais os objetos receptores respondem à mesma mensagem.

1.2.3.3 Generalização (Herança)

A generalização é outra forma de abstração utilizada na orientação a objetos. A Seção 1.2.1 declara que as características e o comportamento comuns a um conjunto de objetos podem ser abstraídos em uma classe. Dessa forma, uma classe descreve as características e o comportamento comuns de um grupo de objetos semelhantes. A generalização pode ser vista como um nível de abstração acima da encontrada entre classes e objetos. Na generalização, classes semelhantes são agrupadas em uma hierarquia (ver Figura 1-4). Cada nível dessa hierarquia pode ser visto como um nível de abstração. Cada classe em um nível da hierarquia herda as características e o comportamento das classes às quais está associada nos níveis acima dela. Além disso, essa classe pode definir características e comportamento particulares. Dessa forma, novas classes podem ser criadas a partir do reúso da definição de outras preexistentes.

O mecanismo de generalização facilita o compartilhamento de comportamento comum entre um conjunto de classes semelhantes. Além disso, as diferenças ou variações entre classes em relação àquilo que é comum entre elas podem ser organizadas de forma mais clara.

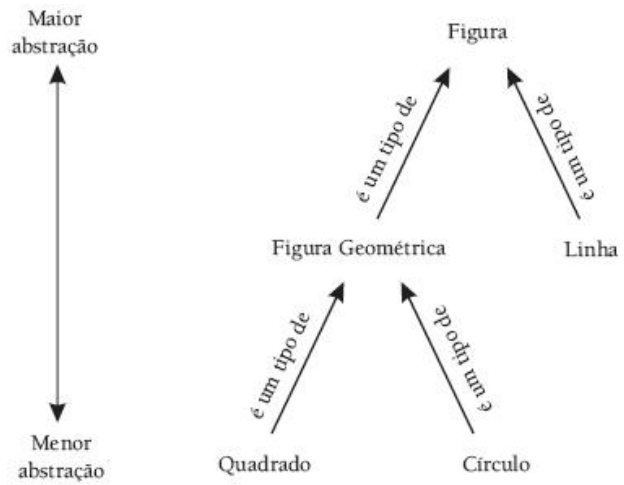

Figura 1-4: Princípio da generalização: classes podem ser organizadas em hierarquias.

1.2.3.4 Composição

É natural para nós, seres humanos, pensar em coisas do mundo real como objetos compostos de outros objetos. Por exemplo, um livro é composto de páginas; páginas possuem parágrafos; parágrafos possuem frases e assim por diante. Outro exemplo: uma televisão contém um painel de controle, uma tela. Da mesma forma que o livro e a televisão podem ser vistos como objetos por si próprios, seus respectivos componentes também podem ser vistos como objetos. De uma forma geral objetos podem ser *compostos* de outros objetos; esse é o princípio da composição. A composição permite que criemos objetos a partir da reunião de outros objetos.

Repare a semelhança entre os princípios de generalização e composição. Ambos propiciam formas de definir novos conceitos (i.e., classes) a partir de conceitos preexistentes. Entretanto, há situações adequadas para aplicação de um princípio ou outro, conforme descrito mais adiante neste livro. O correto entendimento da aplicação desses dois princípios é uma competência importante para o profissional que realiza atividades de modelagem e desenvolvimento de software orientados a objetos.

CAPÍTULO 2

Orientação a Objetos

A UML está totalmente inserida no paradigma de orientação a objetos. Por esse motivo, para compreendê-la corretamente, precisamos, antes, compreender os conceitos da orientação a objetos.

2.1 Classificação, Abstração e Instanciação

No início da infância, o ser humano aprende e pensa de maneira bastante semelhante à filosofia da orientação a objetos, representando seu conhecimento por meio de abstrações e classificações (na verdade, continuamos fazendo isso mesmo quando adultos, mas desenvolvemos outras técnicas que também utilizamos em paralelo). As crianças aprendem conceitos simples, como pessoa, carro e casa, por exemplo, e, ao fazerem isso, definem classes, ou seja, grupos de objetos, sendo cada um deles um exemplo de um determinado grupo, tendo as mesmas características e comportamentos de qualquer objeto do grupo em questão. A partir desse momento, qualquer coisa que tiver cabeça, tronco e membros torna-se uma pessoa, qualquer construção na qual as pessoas possam entrar passa a ser uma casa e qualquer peça de metal com quatro rodas que se locomova de um lugar para outro transportando pessoas recebe a denominação de carro.

Na verdade, esse processo por si só envolve um grande esforço de abstração, em razão, por exemplo, de os carros apresentarem diferentes formatos, cores e estilos. Uma criança deve se sentir um pouco confusa no começo ao descobrir que o objeto amarelo

e o objeto vermelho têm, ambos, a mesma classificação: carro. A partir desse momento, precisa abstrair o conceito de carro para chegar à conclusão de que "carro" é um termo geral que se refere a muitos objetos. No entanto, cada um dos objetos-carro tem características semelhantes entre si, por exemplo, todos têm quatro rodas e, no mínimo, duas portas, além de luzes de farol e freio, bem como vidros frontais e laterais. Além disso, os objetos-carro podem realizar determinadas tarefas, sendo a principal delas transportar pessoas de um lugar para outro.

No momento em que a criança compreende esse conceito, percebe que "carro" é a denominação de um grupo, ou seja, ela abstraiu uma classe: a classe carro. Assim, sempre que perceber a presença de um objeto com as características já determinadas, concluirá que aquele objeto é um exemplo do grupo carro, ou seja, uma instância da classe carro.

Assim, uma das formas de o ser humano aprender e representar conhecimento é, ao menos no início, orientada a objetos, ou seja, aprendemos por meio de classificações. Aprendemos a classificar praticamente tudo, criando grupos de objetos com características iguais, sendo cada um deles equivalente a uma classe. Sempre que precisamos compreender um conceito novo, criamos uma nova classe para esse conceito, muitas vezes derivando-a de classes mais simples (ou seja, conceitos mais simples), e determinamos que todo objeto com as características dessa classe é um exemplo, uma instância dela. Assim, instanciação constitui-se simplesmente em criar um exemplo de um tipo, um grupo, uma classe.

Quando instanciamos um objeto de uma classe, estamos criando um novo item do conjunto representado por essa classe, com as mesmas características e comportamentos de todos os outros objetos já instanciados. Além disso, depois de abstrair um conceito inicial, costumamos criar subdivisões dentro das classes, isto é, grupos dentro de grupos, o que já caracteriza um novo nível de abstração. Podemos criar subgrupos dentro da classe `Carro`, classificando-os por marca ou modelo, por exemplo, ou dentro da classe `Pessoa`, classificando os novos grupos por profissão, grau de parentesco ou status social.

No entanto, deve-se ter em mente que, apesar de terem os mesmos atributos, os objetos de uma classe não são exatamente iguais, pois cada objeto armazena valores diferentes em seus atributos. Por exemplo, todos os objetos da classe `Carro` possuem o atributo `placa`, mas cada um dos objetos possui um valor diferente para sua placa específica. Da mesma forma, todos os objetos da

classe `Pessoa` podem possuir o atributo `nome` ou o atributo `CPF`, mas os valores armazenados nesses atributos variam de pessoa para pessoa.

2.2 Classes de Objetos

Conforme foi dito, uma classe representa uma categoria e os objetos são os membros ou exemplos dessa categoria.

Na UML, uma classe é representada por um retângulo que pode ter até três divisões. A primeira divisão armazena o nome pelo qual a classe é identificada (e essa é a única divisão obrigatória), a segunda enuncia os possíveis atributos pertencentes à classe e a terceira lista as possíveis operações (métodos) que a classe contém. Em geral, uma classe tem atributos e métodos, mas é possível encontrar classes que contenham apenas uma dessas características ou mesmo nenhuma delas, como no caso de classes abstratas. A figura 2.1 apresenta um exemplo de classe.

Nesse caso, identificamos uma classe chamada `Pessoa`. Observe que essa classe tem apenas uma divisão que contém seu nome, pois não é obrigatório representar uma classe totalmente expandida, embora seja mais comum encontrar exemplos assim. Além disso, a classe pode simplesmente não conter atributos nem métodos quando se tratar de classes abstratas. As demais divisões de uma classe serão explicadas nas seções a seguir.

Figura 2.1 – Exemplo de uma classe.

2.3 Atributos ou Propriedades

Classes costumam definir atributos, também conhecidos como propriedades. Os atributos representam as características de uma classe, ou seja, as peculiaridades que costumam variar de um objeto para outro, como o nome, o CPF ou a idade em um objeto da classe `Pessoa` ou a placa ou a cor em um objeto da classe `Carro`, e que permitem diferenciar um objeto de outro da mesma classe em razão de tais variações.

Os atributos são apresentados na segunda divisão da classe e contêm, normalmente, duas informações: o nome que identifica o atributo e o tipo de dado que o atributo armazena, como `integer`, `float` ou `String`. Essa última informação não é obrigatória, mas muitas vezes é útil e recomendável.

Na realidade, não é exatamente a classe que contém os atributos, mas, sim, as instâncias, os objetos dessa classe. Não é realmente

possível trabalhar com uma classe apenas com suas instâncias. Por exemplo, a classe `Pessoa` não existe realmente: é uma abstração, uma forma de classificar e identificar um grupo de objetos semelhantes. Nunca poderemos trabalhar com a classe `Pessoa` propriamente dita, apenas com suas instâncias, como João, Pedro ou Paulo, que são nomes que identificam três objetos da classe `Pessoa`. Da mesma forma, ninguém pode morar na planta de uma casa. A planta exprime o conceito da classe e, com base nela, constroem-se quantas casas forem necessárias e nessas casas reais é que se pode morar, ou seja, a casa construída com base na planta é um objeto, enquanto a planta da casa é uma classe.

Assim, os objetos têm os atributos relativos à classe à qual pertencem. Tais atributos são as características do objeto, como placa, cor e número de portas, no caso de uma instância da classe carro, ou CPF, nome e idade em uma instância da classe `Pessoa`. Todas as instâncias de uma mesma classe têm exatamente os mesmos atributos, no entanto estes podem assumir valores diversos. Assim, o atributo nome do objeto `pessoa1` pode assumir o valor "João", enquanto no objeto `pessoa2` o valor do atributo `nome` pode ser "Paulo". A figura 2.2 demonstra um exemplo de classe com atributos.

O exemplo apresentado na figura 2.2 toma a mesma classe ilustrada na figura 2.1, mas, dessa vez, com duas divisões. A segunda divisão armazena os atributos da classe `Pessoa`, que, nesse caso, são cpf, nome e idade. No exemplo, não foram definidos os tipos dos atributos por ainda não serem necessários.

Figura 2.2 – Exemplo de classe com atributos.

2.4 Operações, Métodos ou Comportamentos

Classes costumam ter métodos, também conhecidos como operações ou comportamentos (a UML usa o termo operação). Um método representa uma atividade que um objeto de uma classe pode executar. Um método pode receber ou não parâmetros (valores utilizados durante a execução do método) e, em geral, tende a retornar valores. Esses valores podem representar o resultado da operação executada ou simplesmente indicar se o processo foi concluído com sucesso ou não.

Assim, um método representa um conjunto de instruções executadas quando o método é chamado. Por exemplo, um objeto da classe `Pessoa` pode executar a atividade de pensar. Grande parte da codificação propriamente dita dos sistemas de informação orientados a objeto está contida nos métodos definidos em suas classes. Os métodos são armazenados na terceira divisão de uma classe. A figura 2.3 apresenta um exemplo de uma classe com métodos.

Figura 2.3 – Exemplo de classe com métodos.

Novamente, tomamos a mesma classe `Pessoa`, ilustrada na figura 2.1, e acrescentamos uma terceira divisão para armazenar o método `pensar()`.

2.5 Visibilidade

A visibilidade é utilizada para indicar o nível de acessibilidade de um determinado atributo ou método, sendo representada à esquerda destes. Existem basicamente quatro modos de visibilidade: público, protegido, privado e pacote.

- A visibilidade privada é representada por um símbolo de menos (-) e significa que somente os objetos da classe detentora do atributo ou método poderão enxergá-lo.
- A visibilidade protegida é representada pelo símbolo de sustenido (#) e determina que, além dos objetos da classe detentora do atributo ou método, também os objetos de suas subclasses poderão ter acesso a este.

- A visibilidade pública é representada por um símbolo de mais (+) e determina que o atributo ou método pode ser utilizado por qualquer objeto.
- A visibilidade pacote é representada por um símbolo de til (~) e determina que o atributo ou método é visível por qualquer objeto dentro do pacote. Somente elementos que fazem parte de um pacote podem ter essa visibilidade. Nenhum elemento fora do pacote poderá ter acesso a um atributo ou método com essa visibilidade.

A figura 2.4 apresenta um exemplo de classe com atributos e métodos com sua visibilidade representada à esquerda de seus nomes.

Figura 2.4 – Exemplo de Visibilidade.

Como se pode verificar nessa figura, utilizamos novamente o mesmo exemplo de classe anterior, acrescentando visibilidade a seus atributos e métodos. No caso, é possível percebermos que os atributos “cpf”, “nome” e “idade” têm visibilidade privada, pois apresentam o símbolo de menos (-) à esquerda de sua descrição e, portanto, somente as instâncias da classe `Pessoa` propriamente

ditas podem enxergar esses atributos. Já o método `pensar()` tem visibilidade pública, como indica o símbolo de mais (+) acrescentado à sua descrição, o que permite que objetos de outras classes enxerguem o método.

É importante destacar que normalmente os atributos costumam ser privados ou protegidos, enquanto os métodos costumam ser públicos. A declaração de atributos como privados, ou eventualmente protegidos, é altamente recomendada, pois isto garante o encapsulamento dos atributos. Um atributo privado, além de só ser visível por objetos de sua classe, só poderá ser acessado por meio de métodos. Assim, objetos de outras classes não terão conhecimento sobre quais atributos estão contidos na classe em questão e nem poderão acessá-los. Eles saberão da existência dos métodos da classe (quando forem públicos), mas não como o método manipula seus atributos.

Nesse exemplo, pode-se observar que optamos por incluir também os tipos dos atributos. Assim, o atributo `cpf` recebeu o tipo `long`, enquanto `nome` e `idade` receberam os atributos `String` e `int`, respectivamente.

2.6 Herança

Como o polimorfismo, que será visto a seguir, a herança é uma das características mais poderosas e importantes da orientação a objetos. Isso se deve ao fato de permitir o reaproveitamento de atributos e métodos, otimizando o tempo de desenvolvimento, além de permitir a diminuição de linhas de código, bem como facilitar futuras manutenções.

A herança na orientação a objetos trabalha com os conceitos de superclasses e subclasses. Uma superclasse, também chamada de classe-mãe, contém classes derivadas dela, chamadas subclasses, também conhecidas como classes-filha. As subclasses, ao serem derivadas de uma superclasse, herdam suas características, ou seja, seus atributos e métodos.

A vantagem do uso da herança é óbvia: ao declararmos uma classe com atributos e métodos específicos e, depois disso, derivarmos uma subclasse da classe já criada, não precisamos redeclarar os atributos e métodos previamente definidos: a subclasse herda-os automaticamente, permitindo reutilização do código já pronto. Assim, só precisamos nos preocupar em declarar os atributos ou métodos exclusivos da subclasse, o que torna muito mais ágil o processo de desenvolvimento, além de facilitar igual-

mente futuras manutenções, sendo necessário apenas alterar o método da superclasse para que todas as subclasses estejam também atualizadas imediatamente.

Além disso, a herança permite trabalhar com especializações. Podemos criar classes gerais, com características compartilhadas por muitas classes, mas que tenham pequenas diferenças entre si. Assim, criamos uma classe geral com as características comuns a todas as classes e diversas subclasses a partir dela, detalhando apenas os atributos ou métodos exclusivos destas. Uma subclasse pode se tornar uma superclasse a qualquer momento, bastando para tanto que se derive uma subclasse dela. A figura 2.5 apresenta um exemplo de herança.

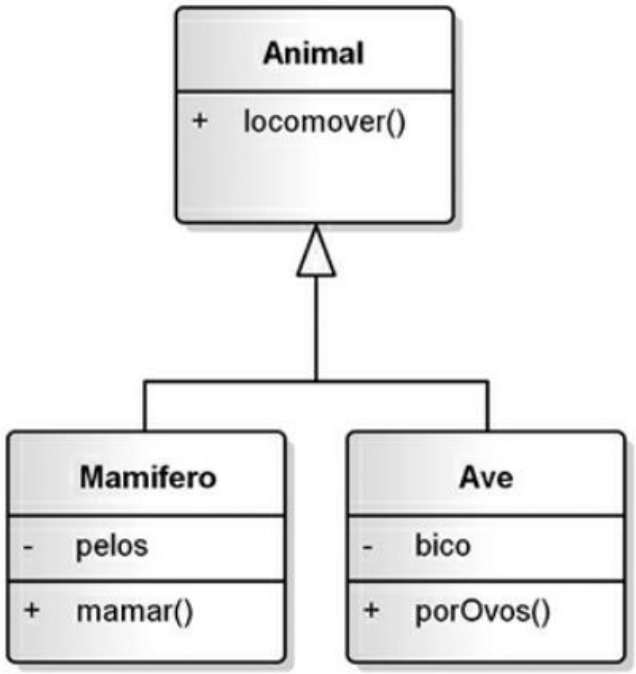

Figura 2.5 – Exemplo de Herança.

Ao observarmos a figura 2.5, percebemos a existência de uma classe geral chamada `Animal` que tem um método chamado `locomover()`. Assim, qualquer instância da classe `Animal` pode se locomover de um lugar para outro. A partir da classe `Animal`, derivamos duas subclasses, `Mamifero` e `Ave`. A classe `Mamifero` tem como atributo a existência de pelos e como método a capacidade de mamar, enquanto a classe `Ave` tem como atributo a existência de um bico e como método a capacidade de pôr ovos, mas ambas têm a capacidade de se locomover, herdada da superclasse `Animal`.

2.6.1 Herança Múltipla

Basicamente, a herança múltipla ocorre quando uma subclasse herda características de duas ou mais superclasses. No caso, uma subclasse pode herdar atributos e métodos de diversas superclasses. No entanto, não são todas as linguagens de programação orientadas a objeto que fornecem esse tipo de recurso, sendo este encontrado, por exemplo, na linguagem C++. Veja um exemplo de herança múltipla na figura 2.6.

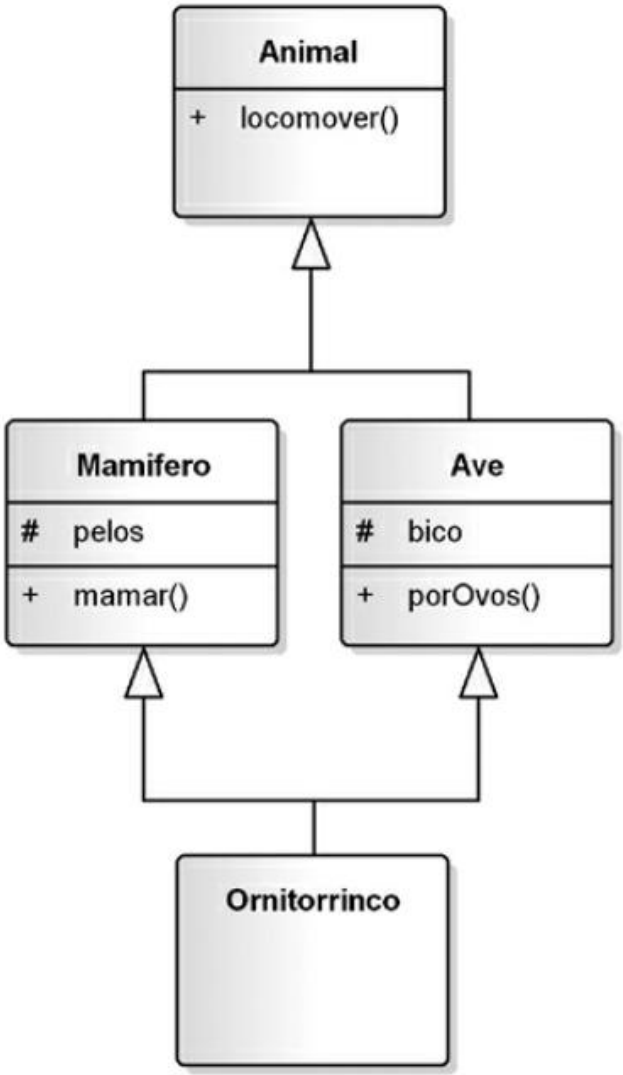
Animal
+ locomover()
Mamifero
pelos
+ mamar()
Ave
bico
+ porOvos()
Ornitorrinco

Figura 2.6 – Exemplo de Herança Múltipla.

No exemplo da figura 2.6, aproveitamos o exemplo da figura 2.5, acrescentando uma nova subclasse derivada das classes `Mamifero` e `Ave`, a classe `Ornitorrinco`. Assim, qualquer instância da classe `Ornitorrinco` terá pelos e poderá mamar, peculiaridades herdadas da classe `Mamifero`, e também terá bico e poderá pôr ovos, peculiaridades da classe `Ave`. Observe que a visibilidade dos atributos `pelos` e `bico` é protegida, de maneira que objetos da classe `Ornitorrinco` possam enxergá-los.

2.7 Polimorfismo

O conceito de polimorfismo está associado à herança. O polimorfismo trabalha com a redeclaração de métodos previamente herdados por uma classe. Esses métodos, embora semelhantes, diferem de alguma forma da implementação utilizada na superclasse, sendo necessário, portanto, reimplementá-los na subclasse.

Porém, para evitar ter de modificar o código-fonte, inserindo uma chamada a um método com um nome diferente, redeclara-se o método com o mesmo nome declarado na superclasse. Assim, podem existir dois ou mais métodos com a mesma nomenclatura, diferenciando-se na maneira como foram implementados, sendo o sistema responsável por verificar se a classe da instância em questão contém o método declarado nela própria ou se o herda de uma superclasse. A figura 2.7 ilustra um exemplo de polimorfismo.

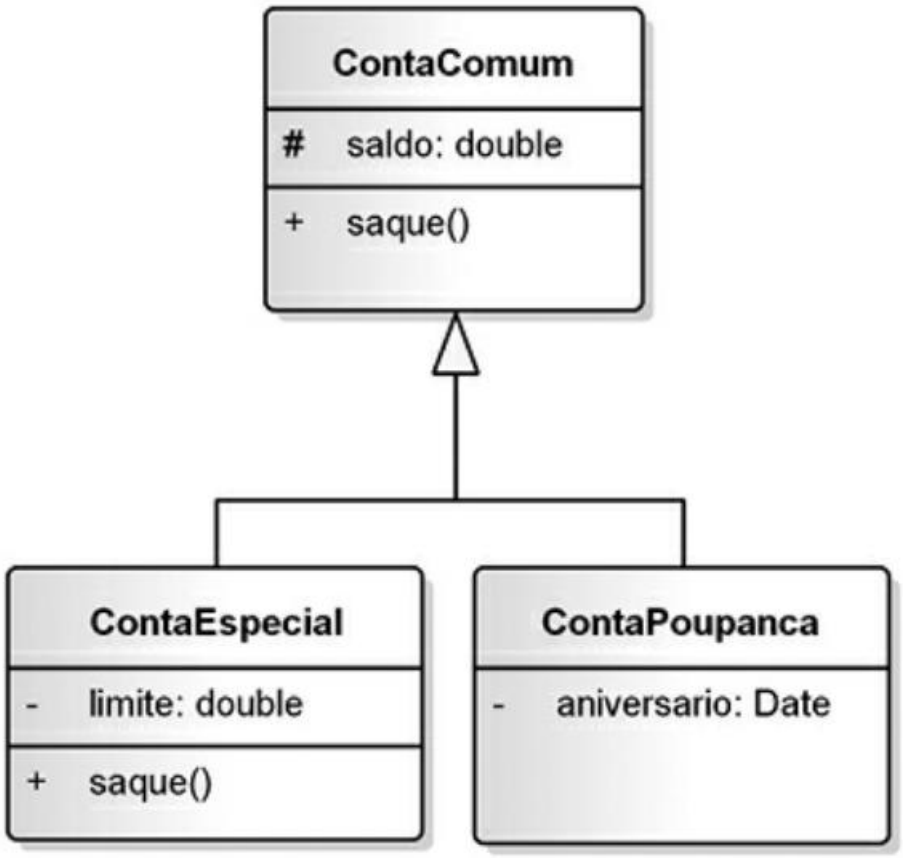

Figura 2.7 – Exemplo de Polimorfismo.

No exemplo apresentado na figura 2.7, utilizamos uma classe geral chamada `ContaComum`. Essa classe tem um atributo chamado "saldo" (com visibilidade protegida para que possa ser acessado pelas subclasses), o qual contém o valor total depositado em uma

determinada instância da classe, e um método chamado `saque`. O método `saque` da classe `ContaComum` simplesmente diminui o valor a ser debitado do saldo da conta e, se este não tiver o valor suficiente, a operação deverá ser recusada.

A partir da classe `ContaComum` derivamos uma nova classe chamada `ContaEspecial` que tem um atributo próprio, além dos herdados, chamado "limite". Esse atributo define o valor extra que pode ser sacado além do valor contido no saldo da conta. Por esse motivo, a classe `ContaEspecial` apresenta uma redefinição do método `saque`, porque o algoritmo do método `saque` da classe `ContaEspecial` não é idêntico ao do método `saque` declarado na classe `ContaComum`, já que é necessário incluir o limite da conta no teste para determinar se o cliente pode retirar ou não o valor solicitado. Porém, o nome do método permanece o mesmo: apenas no momento de executar o método o sistema deverá verificar se a instância que chamou o método pertence à classe `ContaEspecial` ou à `ContaComum`, executando o método definido na classe em questão.

Em uma situação em que existam diversas subclasses que herdem um método de uma superclasse, se uma delas redeclarar esse método, ele só se comportará de maneira diferente nos objetos da classe que o modificou, permanecendo igual à forma como foi implementado na superclasse para os objetos de todas as demais classes. Métodos que são redeclarados em subclasses e apresentam um comportamento diferente do método de mesmo nome contido na superclasse são chamados de métodos polimórficos.