Aula 1

Introdução à Orientação a Objetos

Por Que Orientação a Objetos?

A construção de sistemas de software é um processo intrinsicamente complexo e tem se tornado ainda mais complexo devido aos novos requisitos impostos às aplicações modernas: alta confiabilidade, alto desempenho, desenvolvimento rápido e barato, e tamanho e complexidade grandes. É possível, e muitas vezes até desejável, utilizar métodos e técnicas que nos permitem ignorar ou reduzir a complexidade de tal processo em algumas de suas etapas, mas isso não faz com que a complexidade desapareça, pois ela certamente aparecerá em uma outra etapa.

Os desenvolvedores de software já reconheceram há bastante tempo que a chave para o sucesso no desenvolvimento de softwares está no controle da sua complexidade. O surgimento de linguagens de programação de alto nível e os seus compiladores é um exemplo de um esforço antigo que propiciou uma redução da complexidade de desenvolvimento de softwares, uma vez que ninguém mais necessitou programar em linguagem de máquina, que consiste em uma tarefa bem menos produtiva do que aquela de programar em uma linguagem de alto nível.

Em 1968, na Conferência de Engenharia de Software da NATO, utilizou-se pela primeira vez o termo "crise do software" para designar o mal momento que atravessava a indústria de software, que não possuía métodos adequados para suprir suas demandas. Entre os vários problemas com o desenvolvimento de software, foram mencionados a baixa qualidade, o alto custo de manutenção e a duplicação de esforços no processo de construção. A partir daí surgiram muitos métodos para o desenvolvimento de software, todos eles com o intuito de superar a tal "crise do software". Neste contexto, surgiu o *paradigma da orientação a objetos*.

Muitas são as razões que fazem do paradigma da orientação a objeto um modelo promissor e mais confiável do que os demais para o gerenciamento da complexidade do desenvolvimento de software. Mas talvez a principal razão seja mesmo o simples fato de que todos os bons princípios de projeto de software conhecidos até o momento, tais como abstração de dados, modularização, encapsulamento, herança e reutilização de código, sejam inerentes ao modelo de objetos. Por ironia, o fato de muitos desses princípios existirem antes do surgimento do paradigma da orientação a objetos fez com que o surgimento do modelo de objetos seja visto mais como uma evolução do que como uma revolução.

Programação Orientada a Objetos

A programação orientada a objetos é freqüentemente tida como um novo *paradigma* de programação. Outros paradigmas são a programação procedimental (linguagens como C e Pascal), a programação lógica (Prolog) e a programação funcional (FP ou Haskell). Por *paradigma*, entendemos um conjunto de teorias, métodos e padrões que juntos representam uma forma de organizar o conhecimento, isto é, uma forma de ver o mundo. No contexto de programação, um paradigma é uma forma de conceitualizar o que entendemos por realizar uma computação e como tarefas a serem executadas em um computador deveriam ser estruturadas e organizadas.

Ao tentarmos entender exatamente o que significa o termo *programação orientada a objetos*, é interessante examinar a idéia a partir de várias perspectivas. Para tal, considere o seguinte exemplo:

> Suponha que eu deseje enviar flores para a minha esposa pelo nosso aniversário de casamento. Ela agora está na cidade de Natal e, portanto, a milhares de quilômetros de Campo Grande. Obviamente, eu pegar as flores e levá-las pessoalmente até lá está fora de questão. Entretanto, enviá-las para ela é uma tarefa razoavelmente fácil. Eu meramente vou a uma floricultura local, onde trabalha o florista Florisvaldo, informo-lhe o tipo e o número de flores que eu quero enviar e o endereço da minha esposa, e posso estar certo de que as flores serão segura e automaticamente entregues a tempo.

O mecanismo que eu usei para resolver o problema da entrega das flores foi encontrar um *agente* adequado (chamado Florisvaldo) e enviá-lo uma *mensagem* contendo a minha solicitação. É *responsabilidade* de Florisvaldo cumprir minha solicitação. Há algum *método* – algum algoritmo ou conjunto de operações – usado por Florisvaldo

para fazer o que eu pedi. Eu não preciso saber qual método ele usará para cumprir minha solicitação. Esta informação comumente é *ocultada* de mim. Entretanto, se eu investigar, descobrirei que Florisvaldo entrega uma mensagemn ligeiramente diferente para um outro florista em Natal. Este florista, por sua vez, toma as devidas providências e passa as flores, através de uma outra mensagem, para seu entregador e assim por diante. Desta forma, minha solicitação é finalmente cumprida por uma seqüência de solicitações de um agente para outro.

Nosso primeiro princípio de resolução de problemas de forma orientada objetos é o veículo pelo qual as atividades são iniciadas. Na programação orientada a objetos, a ação é iniciada através da transmissão de uma mensagem para um agente (um *objeto*) responsável pela ação. A mensagem codifica o pedido de realização da ação e é acompanhada de quaisquer informações adicionais (argumentos) necessários à execução do pedido. O *receptor* é o agente para quem a mensagem é enviada. Se o receptor aceita a mensagem, ele aceita a responsabilidade de cumprir a ação indicada. Em resposta à mensagem, o receptor executará algum método que satisfaça o pedido.

Embora tenha lidado com Florisvaldo apenas umas poucas vezes, eu tenho uma idéia geral de seu comportamento quando vou a sua loja e apresento a minha solicitação. Eu sou capaz de assumir certas verdades acerca do comportamento dele porque eu tenho alguma informação sobre floristas em geral, e eu espero que Florisvaldo, sendo um membro desta categoria, comporte-se como tal. Podemos usar o termo `Florista` para representar a categoria (ou *classe*) de todos os floristas. Vamos incorporar estas noções em nosso segundo princípio de programação orientada a objetos: todos os objetos são ocorrências de uma *classe*. O método invocado por um objeto em resposta a uma mensagem é determinado pela classe do receptor. Todos os objetos de uma dada classe utilizam o mesmo método em resposta a mensagens similares.

Eu também posso assumir mais verdades acerca de Florisvaldo não apenas porque ele é um florista mas também porque ele é um vendedor. Sei, por exemplo, que serei solicitado a pagar uma certa quantia de dinheiro como parte da transação e que ele me dará um recibo pelo pagamento que farei. Estas ações são também esperadas de açogueiros e caixas de supermercados. Desde que a categoria `Florista` é uma forma mais especializada da categoria `Vendedor`, qualquer conhecimento que eu tenha acerca de `Vendedores` poderá ser imaginado de `Floristas` e, portanto, de Florisvaldo.

Uma maneira de imaginar como eu organizei meu conhecimento acerca de Florisvaldo é em termos de uma hierarquia de categorias. Florisvaldo é um `Florista`, mas `Florista` é uma forma especializada de `Vendedor`. Mais ainda, um `Vendedor` é um `Humano` e, portanto, eu sei que Florisvaldo é provavelmente um bípede. Um `Humano` é um `Mamífero` e, portanto, amamenta seus filhos; e um `Mamífero` é um `Animal` e, por-

tanto, respira oxigênio; e um `Animal` é um `Objeto Material` e, portanto, ele possui massa e peso. Desta forma, há muito conhecimento acerca de Florisvaldo que não está diretamente associado a ele e nem mesmo à categoria `Florista`. O princípio segundo o qual conhecimento acerca de uma categoria mais geral pode também ser aplicado a uma categoria mais específica é denominado *herança*. Dizemos que a classe `Florista` herdará atributos da classe ou categoria `Vendedor`.

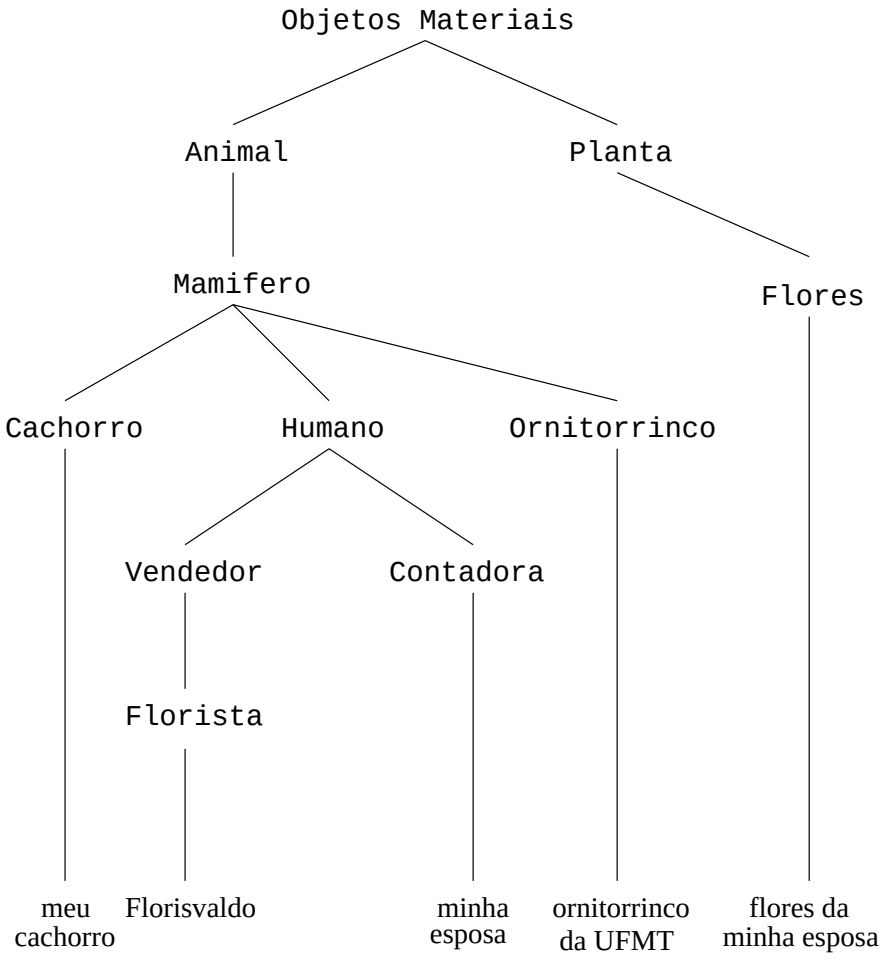

Figure 1.1: Exemplo de hierarquia de classes

A mesma hierarquia de categorias que utilizei para derivar conhecimento acerca de Florisvaldo também pode incluir a minha esposa, meu cachorro, o ornitorrinco que vive no zoológico da UFMT e as flores que enviei para minha esposa. Uma hierarquia deste tipo é comumente representada em forma de "árvore", como mostrado na Figura 1.1. A informação que possua acerca de Florisvaldo, que é um `Humano`, pode ser também aplicada a minha esposa, por exemplo. A informação que possuo acerca dela porque

ela é um `Mamífero` pode ser aplicada também ao meu cachorro. Informações acerca de todos os membros de `Objeto Material` pode ser igualmente aplicada a Florisvaldo e a suas flores. Capturamos isto na idéia de herança: classes podem ser organizadas em uma estrutura de herança hierárquica. Uma *classe filha* (ou *subclasse*) herdará atributos de uma *classe pai* mais alta na árvore. Uma *classe pai abstrata* é uma classe, tal como `Mamífero`, para a qual não há ocorrências diretas; ela é usada apenas para criar subclasses.

O ornitorrinco que vive no zoológico da UFMT apresenta um problema para a nossa organização hierárquica. Eu sei que mamíferos dão a luz a filhos vivos e o ornitorrinco, que certamente é um mamífero, põe ovos. Para acomodar este fato, necessitamos de uma técnica para codificar *exceções* a uma regra geral. Fazemos isso por decretar que informações contidas em uma subclasse podem *sobrescrever* informações herdadas de uma classe pai. Mais freqüentemente, implementações desta abordagem tomam a forma de um método em uma subclasse tendo o mesmo nome que um método na classe pai, combinado com uma regra segundo a qual uma busca por um método para ser acoplado a uma mensagem específica é conduzida.

A busca por um método a ser invocado em resposta a uma dada mensagem inicia na classe do *receptor*. Se nenhum método apropriado é encontrado, a busca é conduzida na *classe pai* desta classe. A busca segue acima, na cadeia da classe pai, até que um método seja encontrado ou o final da cadeia da classe pai tenha sido atingido. No primeiro caso, o método é executado; no segundo uma mensagem de erro é fornecida. Se métodos com o mesmo nome podem ser encontrados no alto de uma hierarquia o método executado é dito *sobrescrever* o comportamento herdado.

A *interpretação* de uma mensagem depende do receptor e pode variar para diferentes receptores. Eu posso enviar uma mensagem para minha esposa para ela entregar flores para minha mãe, por exemplo, e ela entenderá e produzirá um resultado desejado. Entretanto, o método que ela usará para cumprir minha solicitação poderá muito bem ser diferente daquele usado por Florisvaldo. Se eu pedir ao meu cachorro para fazer o mesmo, certamente ele não fará, pois não terá um método para resolver o problema. O fato de minha esposa e Florisvaldo responderem a minha mensagem usando métodos distintos é um exemplo de uma forma de *polimorfismo*. O fato de eu não saber que método eles usarão para cumprir a minha solicitação é um exemplo de *ocultamento de informação*.

Uma Nova Forma de Pensar

A visão de programação representada pelo exemplo de enviar flores para minha esposa é muito diferente da concepção tradicional de um computador. O modelo tradicional descrevendo o comportamento de um computador executando um programa é um modelo de *estado-processo*. De acordo com este modelo, o computador é um gerenciador de dados, seguindo algum padrão de instruções, caminhando ao longo da memória, removendo valores de várias entradas da memória, transformando-os de alguma forma e colocando-os de volta nas entradas da memória. Através da visualização dos valores na memória, podemos determinar o estado da máquina ou os resultados produzidos pela computação. Este modelo certamente não segue a forma como a maioria das pessoas resolvem seus problemas.

Por outro lado, na visão orientada a objetos do exemplo do envio de flores, não mencionamos endereços, variáveis, atribuições ou quaisquer termos da programação convencional. Ao invés disso, falamos sobre objetos, mensagens e responsabilidade por alguma ação. Na programação orientada a objetos, o desenvolvedor descreve quais são as várias entidades no "universo" do programa e como elas interagem uma com as outras para produzir o resultado desejado. Deste ponto de vista, programar orientado a objetos é muito mais do que simplesmente aprender uma linguagem orientada a objetos, tal como C++ ou Java. Na verdade, é uma nova forma de raciocinar sobre a solução de um problema.

É importante notar também que o uso de uma linguagem orientada a objetos pode *simplificar* o desenvolvimento de soluções orientadas a objetos, embora não *force* o desenvolvedor a se tornar um programador orientado a objetos. Para tal, ele deve se acostumar a "enxergar" o mundo sob a ótica dos princípios da orientação a objetos e aplicá-los através de uma linguagem de programação adequada. Para termos uma idéia da influência do uso de uma linguagem na forma de raciocínio de um programador, considere que o seguinte exemplo.

Há alguns anos, um estudante trabalhando em uma pesquisa genética se deparou com um problema de análise de uma seqüência de DNA, tal como

ACTCGGATCTTGCATTTCGCCAATTGGACCCTGACTTGGCCA ...,

na qual ele deveria descobrir se havia algum padrão de comprimento M, onde M era uma constante fixa e pequena (digamos cinco ou dez proteínas), sendo a seqüência de tamanho N, onde N é um valor muito grande (na ordem de dezenas de milhares de proteínas). O estudante então representou a seqüência de tamanho N por um vetor de caracteres de tamanho N e escreveu um simples código em FORTRAN que se assemelhava ao código abaixo:

```
DO 10 I = 1, N-M
DO 10 J = 1, N-M
ACHOU = .TRUE.
DO 20 K = 1, M
  IF X[I+K-1] .NE. X[J+K-1] THEN
    ACHOU = .FALSE.

  IF ACHOU THEN ...

10 CONTINUE
```

O estudante ficou desapontado quando executou seu programa e verificou que ele necessitava de algumas horas para terminar. Ele, então, resolveu discutir o problema com um colega que usava a linguagem de programação APL para codificar seus programas. O colega dele disse que ia tentar resolver o problema e, depois de alguns dias, veio com a seguinte solução em APL, que executava em alguns minutos, ao invés de horas. Ele reorganizou a seqüência em uma matriz com aproximadamente N linhas e M colunas:

A	C	T	C	G	G	posições de 1 a M
C	T	C	G	G	A	posições de 2 a $M+1$
T	C	G	G	A	T	posições de 3 a $M+2$
C	G	G	A	T	T	posições de 4 a $M+3$
G	G	A	T	T	C	posições de 5 a $M+4$
G	A	T	T	C	T	posições de 6 a $M+5$
	.	.	.			
T	G	G	A	C	C	
G	G	A	C	C	C	

E, em seguida, ordenou esta matriz de acordo com as linhas. Se houvesse algum padrão repetido, então duas linhas da matriz ordenada teriam valores idênticos. A razão pela qual o programa em APL era mais rápido do que o programa em FORTRAN nada tinha haver com a velocidade de APL versus FORTRAN, mas com a solução empregada. O programa em FORTRAN era $\mathcal{O}(M \times N^2)$, enquanto o programa em APL era $\mathcal{O}(M \times N \log N)$.

A moral da estória, entretanto, não é que APL é de alguma forma uma linguagem de programação melhor do que FORTRAN, mas que o programador de APL foi naturalmente levado a descobrir a melhor solução devido ao fato que laços são muito difíceis de escrever em APL, enquanto ordenação é trivial – esta operação é um operador embutido na linguagem. Logo, bons programadores em APL tendem a procurar novas

aplicações para a ordenação. A conclusão que chegamos aqui é que a linguagem de programação em que uma solução vai ser escrita direciona a mente do programador para "enxergar" o problema de uma certa maneira.