Programação Orientada ao Objeto: uma abordagem didática

Object-Oriented Programming: a didatic presentation

Mário Leite[1]
Nelson Abu Sanra Rahal Júnior[2]

Resumo

Este artigo focaliza a metodologia de escrita de código denominada Programação Orientada ao Objeto (OOP), cada vez mais empregada no desenvolvimento de sistemas, tornando-se o paradigma mais atual em termos de criação de *softwares*. Em particular, é mostrado um exemplo prático do uso de dois conceitos básicos dessa metodologia: herança e polimorfismo. O exemplo apresentado mostra a criação da superclasse *Mamífero* e de três subclasses: *Homem*, *Cão* e *Gato*. Usando o conceito de herança, o método Comunicar (originário da classe *Mamífero*) é herdado pelas subclasses, e através do conceito de polimorfismo esse método é redefinido.

Palavras chave: abstração, classe, herança, objeto, OOP, polimorfismo

Abstract

This article shows the methodology of writing code named **O**bject-**O**riented **P**rogramming (OOP), more and more used in the development of systems, becoming itself the most current paradigm in terms of creation of softwares. Specially, it has been shown a practical example of two basic concepts of the use of this methodology: Inheritance and polimorphism. The presented example shows the creation of the "Mamífero" superclass and three subclasses: "Homem", "Cão", and "Gato". Using the concept of Inheritance, the method Comunicar (originally from Mamifero class) is inherited by the subclasses, and by the concept of polymorphism this method is redefined.

Key-words: abstraction, class, inheritance, object, OOP, polymorphism.

1 - Introdução

A Programação Orientada ao Objeto (**O**bject-**O**riented **P**rogramming) pode ser considerada como uma extensão quase natural da Programação Modular; entretanto a sigla OOP tem causado um certo "frisson" entre a comunidade de Informática, nos últimos anos. Na verdade, isto não deveria acontecer, uma vez que a OOP foi concebida há muito tempo atrás (no inicio da década de 70). A sua

[1] Professor do CESUFOZ (Foz do Iguaçu), graduado em Engenharia (PUC/RJ), especialista em Engenharia (PUC/RJ) e em Análise de Sistemas (CESUMAR – Maringá/PR) e mestre em Engenharia de Produção (UFSC).
[2] Mestre em Ciência da Computação (UFSC), Doutorando em Ciência da Computação (UFSC)

origem vem da linguagem Simula (Simula Language), concebida na Noruega no início da década de 60, e como o nome indica, foi criada para fazer simulações; entretanto, seu uso alavancou um conceito que até então passava "despercebido" pela maioria dos projetistas: a similaridade com o mundo real. A primeira linguagem de programação a implementar sistematicamente os conceitos de OOP foi a linguagem SIMULA-68; em seguida surgiu a linguagem Smalltalk -criada pela Xerox -, que pode ser considerada a linguagem que popularizou e incentivou o emprego da OOP. Atualmente podemos encontrar versões de Smalltalk para microcomputadores, o que facilitou enormemente o seu uso, tirando-a dos ambientes privativos das Universidades. O resultado foi uma linguagem de pura linhagem OO, poderosíssima, que implementa todos os conceitos de OO, o que não acontece com as chamadas linguagens OO híbridas que implementam apenas alguns conceitos de orientação ao objeto. Com o aparecimento da famosa "crise do *software*", o emprego da OOP foi a saída protagonizada pelos desenvolvedores para minimizar os custos dos sistemas, em particular os custos relativos às manutenções corretivas, uma vez que cerca de 75% dos custos dos programas referem-se ao indesejável expediente de alterar e/ou remendar códigos dos sistemas já implantados e em operação. Basicamente, a OOP utiliza os mesmos princípios da engenharia de *hardware* que projeta novos equipamentos usando os mesmos componentes básicos como transistores, resistores, fusíveis, diodos, chips, etc. Os "objetos" já existentes são utilizados para produzir novos "objetos", tornando essa metodologia mais poderosa que as metodologias tradicionais de desenvolvimento de sistemas. Se consideramos a Orientação ao Objeto como um novo paradigma de desenho de *software*, devemos considerar, também, uma nova maneira de pensar, porque apesar de a escrita do código continuar sendo procedural, alguns conceitos mudam radicalmente: a estruturação e o modelo computacional. Fundamentalmente o que se deseja com esta metodologia são basicamente duas características: reutilização de código e modularidade de escrita; e nisto a OOP é imbatível quando comparada com as metodologias antigas. Em termos de modelo computacional podemos dizer que enquanto as metodologias tradicionais utilizam o conceito de um processador, uma memória e dispositivos de I/O para processar, armazenar e exibir as informações, a OOP emprega um conceito mais real, mais concreto, que é o de Objeto.

Uma definição para objeto seria a de um "ente" ativo dotado de certas características que o tornam "inteligente", a ponto de tomar certas decisões quando devidamente solicitado. Outra definição mais formal para objeto poderia ser: *uma unidade dinâmica, composta por um estado interno privativo (estrutura de dados) e um comportamento (conjunto de operações)*. E neste caso, segundo PRICE [12], um objeto em particular é como um processador com memória própria e independente de outros objetos. Em termos de implementação, objeto é um bloco de dados privados envolvidos por código, de maneira que o acesso a ele só pode ser feito sob condições especiais. Todo o comportamento desse "ente" encapsulado é descrito através de rotinas que manipulam seus dados, sendo que o seu estado corrente está em seus próprios dados; em outras palavras, cada objeto tem suas próprias características, moldadas a partir de uma matriz. Formalmente, para ser considerada uma linguagem OO, esta precisa implementar quatro conceitos básicos: abstração, encapsulamento, herança e polimorfismo.

Abstração é considerada como a habilidade de modelar características do mundo real do problema que o programador esteja tentando resolver. Por exemplo, se o programador estiver interessado em controlar dados dos clientes de uma empresa, é muito mais fácil lidar com uma linguagem que ofereça recursos em que ele possa criar algo chamado "Cliente" ao invés de recorrer à estruturas de dados tipo *array* ou *record*. Nesse contexto a abstração refere-se à capacidade de modelar o mundo real, e por outro lado, podemos considerá-la como um mecanismo pelo qual restringimos o nosso universo de análise e as variáveis e constantes que compõem esse universo, desprezando os dados que não nos interessa na análise. Podemos demostrar o uso de abstração facilmente, quando fechamos os olhos e pensamos em uma mesa; esta mesa imaginária provavelmente não vai ser igual à uma outra imaginada por outras pessoas, mas o que importa é que todos as pessoas que imaginaram uma mesa, colocaram nessa as informações que para elas são necessárias para a sua função (de ser uma mesa). Não

importa se a mesa é de três pés ou quatro, ou se o tampão é de vidro, madeira ou mármore; o que importa é que a imagem que idealizamos em nossa cabeça é de uma mesa e tenha as informações necessárias para cumprir sua função.

Encapsulamento é a base de toda a abordagem da Programação Orientada ao Objeto; isto porque contribui fundamentalmente para diminuir os malefícios causados pela interferência externa sobre os dados. Partindo desse princípio, toda e qualquer transação feita com esses dados só pode ser feita através de procedimentos colocados "dentro" desse objeto, pelo envio de mensagens. Desta maneira, dizemos que um dado está encapsulado quando envolvido por código de forma que só é visível na rotina onde foi criado; o mesmo acontece com uma rotina, que sendo encapsulada, suas operações internas são invisíveis às outras rotinas. E até mesmo em linguagens consideradas não OO, como no caso do Clipper 5.xx, segundo FERREIRA & JARABECK [1], pode-se observar um certo encapsulamento nas rotinas em que as variáveis são declaradas como LOCAL. Nesses casos tais variáveis são visíveis somente dentro dessas rotinas aonde foram declaradas, o que permite ao programador uma certa segurança quanto aos acessos indevidos por parte de outras rotinas, o que não acontece com variáveis PRIVATE ou PUBLIC, no contexto dessa linguagem. No encapsulamento podemos visualizar a sua utilidade pensando em um vídeo cassete, onde temos os botões de liga-desliga, para frente, para traz, etc. Estes botões executam uma série de operações existentes no aparelho, onde são executadas pelos componentes existentes dentro do aparelho (transistores, cabos, motores, etc.) Não interessa ao operador saber como é o funcionamento interno do equipamento; esta informação só é relevante para os projetistas do aparelho. As informações pertinentes ao usuário do equipamento são as existentes no meio externo (botões, controle remoto) que ativam as operações internas do equipamento. Desta maneira o aparelho de vídeo cassete pode evoluir com os avanços tecnológicos, e as pessoas que o utilizam continuam sabendo utilizar o equipamento, sem a necessidade de um novo treinamento. Na área de *software* acontece o mesmo: as classes podem continuar evoluindo, com aumento de tecnologia, e os programas que utilizam essas classe continuam compatíveis. Isto ocorre porque a esses programas não interessa saber como é o funcionamento interno da classe e sim sua função, para que ele possa executar, conforme ela evolui, novas funções colocadas à sua disposição.

Herança é um mecanismo que, se for bem empregado, permite altos graus de reutilização de código. Do ponto de vista prático, pode ser entendido como sendo um conjunto de instâncias criadas a partir de um outro conjunto de instâncias com características semelhantes, e os elementos desse subconjunto herdam todas as características do conjunto original. A idéia é fornecer um mecanismo simples (mas muito poderoso) para que se defina novas classes a partir de uma já existente. Assim sendo, dizemos que essas novas classes herdam todos os membros (propriedades+métodos) da classe-mãe; isto torna o mecanismo de herança uma técnica muito eficiente para construir, organizar e reutilizar código. Por isso, nas linguagens que não suportam esse mecanismo, as classes são criadas como unidades independentes: cada uma com seus membros concebidos do zero (sem vínculo direto com outras classes), o que torna o processo mais demorado e com códigos, às vezes, redundantes. Um exemplo de linguagem que não implementa herança na sua forma clássica é o Visual Basic® (até a atual versão *desktop* 6.0); neste caso o desenvolvedor tem que simular esse mecanismo, usando a criatividade. A herança possibilita a criação de uma nova classe de modo que essa classe (denominada subclasse, classe-filha ou classe derivada) herda TODAS as características da classe-mãe (denominada superclasse, classe base ou classe primitiva); podendo ainda, a classe-filha, possuir propriedades e métodos próprios. No processo de herança podemos imaginar um ser humano, que nasce com todas as características de um ser humano sadio; agora, coloquemos nele uma roupa e um relógio. A roupa e o relógio não faz parte do ser humano, mas quando "pegamos" este ser, vestido e com um relógio, e realizamos o processo de herança é gerada uma copia idêntica da matriz. Se colocarmos um sapato preto no ser humano original a sua copia também ficará calçada, e se trocarmos a camisa do ser humano original a sua cópia também vai receber a nova camisa; isto demostra que a copia continua vinculada à

matriz de origem. Podemos tirar quantas cópias que desejarmos da matriz original e todas estas cópias manterão o seu vínculo. Podemos, até, tirar cópias das cópias, mas o processo de modificarmos a matriz original implicará numa mudança em todas as outras que estão abaixo dela. Nunca uma modificação feita nas copias altera a matriz de origem, e nunca podemos remover um item que tenha sido recebido por intermédio da herança, isto que dizer que nenhuma das cópias (humanas) poderá se dar ao luxo de não ter o relógio.

O termo polimorfismo, etimologicamente, quer dizer "várias formas"; todavia, na Informática, e em particular no universo da OOP, é definido como sendo um código que possui "vários comportamentos" ou que produz "vários comportamentos"; em outras palavras, é um código que pode ser aplicado à várias classes de objetos. De maneira prática isto quer dizer que a operação em questão mantém seu comportamento transparente para quaisquer tipos de argumentos; isto é, a mesma mensagem é enviada a objetos de classes distintas e eles poderão reagir de maneiras diferentes. Um método polimórfico é aquele que pode ser aplicado à várias classes de objetos sem que haja qualquer inconveniente. É o caso por exemplo, do método *Clear* em Delphi®, que pode ser aplicado tanto à classe *TEdit* como à classe *TListBox*; nas duas situações o conteúdo desse objetos são limpos, mesmo pertencendo, ambos, à classes distintas, LEITE [6]. Segundo FERREIRA & JARABECK [2], um exemplo bem didático para o polimorfismo é dado por um simples moedor de carne. Esse equipamento tem a função de moer carne, produzindo carne moída para fazer bolinhos. Desse modo, não importa o tipo (classe) de carne alimentada; o resultado será sempre carne moída, não importa se de boi, de frango ou de qualquer outro tipo. As restrições impostas pelo processo estão no próprio objeto, definidas pelo seu fabricante e não pelo usuário do produto.

1.1 - Justificativa

Na programação Orientada ao Objeto temos a necessidade de buscar as classes de domínio, onde através da qual um indivíduo observa a realidade (domínio) e procura capturar sua estrutura (abstrair entidades, ações, relacionamento, etc) com elementos que forem considerados relevantes para a descrição desse domínio. Com as classes de domínio identificadas, podemos utilizar as técnicas de abstração para a composição das classes abstratas, construindo desta maneira um sistema facilmente reutilizável nas suas estruturas internas. Analisando o objetivo de desenvolvimento do nosso projeto, temos identificadas as classes Homem, Cão e Gato, onde estas classes possuem atributos e métodos semelhantes, juntamente com aqueles exclusivos de cada uma. Criando um grupo de dados em comum com estas classes, podemos criar uma classe chamada Mamífero, onde esta terá a característica de ser a superclasse que dará origem às demais classes do sistema. As classes geradas desta superclasse (Homem, Cão e Gato) têm em sua estrutura os atributos e métodos que dizem respeito apenas à elas. Desta maneira, a classe Homem não possui o atributo "cauda" que existe nas demais classes, mas todas as classes geradas a partir da superclasse possuem todos os atributos e métodos nela declarados.

Através do recurso de abstração podemos melhorar o sistema sem ter que realizar um grande esforço de manutenção. As correções necessárias às classes específicas são realizadas nelas mesmo, mas as correções necessárias à todas as classes devem ser feitas uma vez só: na superclasse, pois as demais classes geradas a partir dela já sofreram essas correções. Quando criamos as classes de domínio estamos criando *softwares* no paradigma da orientação ao objeto com características de camadas, onde identificamos várias camadas no mesmo projeto. Nosso aplicativo tem apenas duas camadas, a camada de domínio do problema (Homem, Cão e Gato) e a camada de interface (janela do aplicativo) em

conjunto com essas classes (figura 2).

As técnicas de abstração podem ser aplicadas na criação de janelas (*frames*), centralizando os recursos de operacionalidade em uma superclasse e deixando a especialização para as classes específicas (subclasses) criadas a partir daquela. A programação em camadas se faz necessária para se ter um reaproveitamento de código com as classes abstratas. Na ausência dessa técnica o reaproveitamento fica comprometido por estarmos trabalhando com o domínio e a janela juntos, como se fosse uma única peça. É verdade que o tempo necessário para se criar um software usando o paradigma da orientação a objetos é maior no início do projeto pela necessidade de identificarmos as classes abstratas. Todavia um bom *software* com técnicas de OOP é gerado a partir do tempo gasto nessa etapa; quanto mais abstratas forem as suas classes de apoio, maior será a facilidade de implementação das classes específicas. Embora um sistema de pequeno porte possa ser implementado muito mais rapidamente com técnicas de programação estruturada do que nas técnicas de OOP, terá sua manutenção comprometida na relação custo-tempo, onde a manutenção em um *software* codificado com a técnica da programação estruturada é muito mais difícil de ser realizada. Já no *software* orientado ao objeto basta alterar na classe primitiva para que todas as classes derivadas percebam essa correção. A manutenção de um software criado com a tecnologia OO é muito mais fácil e mais rápida do que numa programação estruturada pela facilidade que se tem em alterar a superclasse e todos as classes geradas a partir dela. Num programa com as características de estruturado isto já não acontece, tendo as suas correções de serem realizadas em todos os módulos que se façam necessárias.

1.2 - Objetivo

O objetivo deste trabalho é mostrar um exemplo prático e didático do uso de dois conceitos básicos da metodologia OOP: herança e polimorfismo. Estes dois conceitos quando usados em conjunto permite ganhos extraordinários na codificação de rotinas, evitando replicações desnecessárias de estruturas do tipo *If..Then*, *Select* ou *Cases*, e aumentando sensivelmente a produtividade do desenvolvedor. Este exemplo mostra a criação da superclasse Mamífero e de três subclasses: Homem, Cão e Gato. Usando o conceito de herança, o método Comunicar (originário da classe Mamífero) é herdado pelas subclasses; e através do conceito de polimorfismo, esse método é redefinido facilmente para as subclasses, (falar para o homens, latir para os cães e miar para os gatos) mostrando o quanto se ganha em produtividade na criação de sistemas com a reutilização de código (vide Figura 1). Como mostra o exemplo da figura 2, quando executamos o método **Comunicar** (originário da superclasse Mamífero) ele é redefinido para **latir**, de modo que a instância Totó o execute adequadamente, evidenciando a importância do polimorfismo

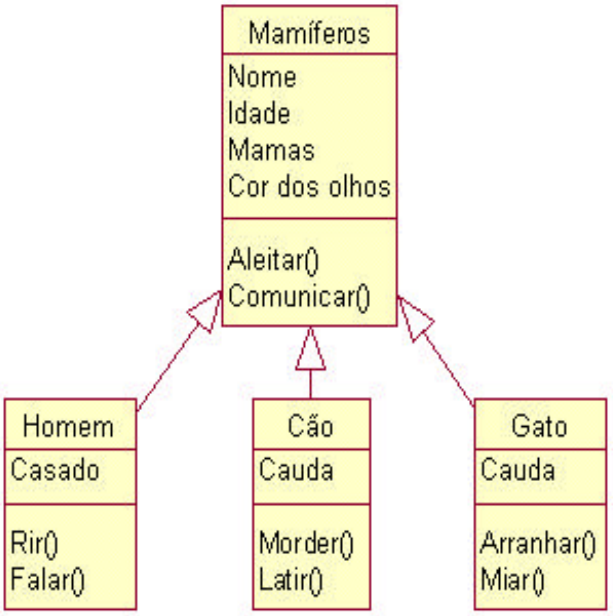

Figura 1 - Esquema da hierarquia das classes

Figura 2 - Interface da aplicação

2 - Metodologia

O exemplo criado foi idealizado com a necessidade de se projetar um conjunto de classes de fácil entendimento e que permitisse o uso dos conceitos de herança e polimorfismo, sendo este exemplo tirado do mundo real, originando a classe Mamífero. Primeiramente buscou-se identificar elementos em comum às classes Homem, Cão e Gato, para que estes fossem adicionados à classe Mamífero, que passou a ser a superclasse do exemplo. A partir dos elementos pertinentes à todas as demais classes, foi feito o estudo da especialização existente à cada classe, tendo a necessidade de uma redefinição de comportamento do método Comunicar (figura 2), que foi customizado conforme a classe em que ele se identificava melhor (polimorfismo). Toda a implementação do programa de exemplo e seus testes foram realizado dentro da própria estrutura da instituição de ensino superior CESUMAR (Centro de Ensino Superior de Maringá, localizada no norte do Paraná na cidade de Maringá), e para a criação do exemplo foi utilizado a linguagem de programação JAVA em ambiente operacional Windows 98.

3 - Resultados

Este trabalho foi apresentado aos alunos de graduação como objetivo de validação do conteúdo, sendo executado como complemento da disciplina "Tópicos Avançados em Programação", onde se demostrou sua facilidade e clareza dos termos técnicos e assimilação desse conteúdo por parte dos alunos. Com a sua aplicabilidade em sala de aula, os alunos experimentaram a produção de uma aplicação cem por cento orientada ao objeto numa linguagem portável em várias plataformas. Desse modo o docente tem a oportunidade de trabalhar com a área de portabilidade de *software*, *software* para Web e reaproveitamento de código. O resultado imediato foi um sensível aumento de interesse nessa metodologia por parte dos alunos e o aparecimento de algumas boas idéias de implementação de programas usando a OOP.

4 - Conclusões e Recomendações

O paradigma da programação orientada ao objeto possibilita a criação de sistemas que parecem teias de aranhas, onde cada nó dessa teia é uma classe, mas cada classe pode ser aproveitada para a composição de outros nós da teia, bem como a reutilização desses nós em outros sistemas, diminuindo drasticamente o tempo de codificação e depuração de códigos.

Na idealização de um sistema orientado ao objeto passamos a analisar a possibilidade de abstração que as classes podem sofrer, pois quanto maior for a capacidade de abstração, maior será a projeção de reutilização de classes no sistema. Nessa abstração surgem as superclasses que tem elementos em comum e que serão utilizados nas subclasses; estes elementos sofrem, conforme a necessidade, o recurso do polimorfismo. Projetar um sistema em OOP identificando superclasses e elementos que terão o recurso do polimorfismo, só se adquire com experiência em vários projetos que utilizam essa tecnologia. O exemplo apresentado neste trabalho tem um papel importante na identificação dos processos de herança e polimorfismo, pela sua simplicidade, facilidade de identificação de suas característica e esboço das mesmas. Dessa maneira, este material passa a ser um instrumento útil de apoio à transmissão das metodologias nele tratadas. Por outro lado, devido à sua importância na criação de *softwares* com qualidade - de acordo com PALADINI [10], para adequação

ao uso do cliente - recomendamos que a metodologia de Orientação ao Objeto seja assunto de discussão corrente nos cursos ligados à Informática, dessa e de outras instituições de ensino superior, não apenas como complemento de alguma disciplina. E as disciplinas que contemplarem esse assunto deverão tratá-lo de maneira didática e com exemplos bem esclarecedores para não cair nas mesmas armadilhas que existem na literatura, as quais não raramente exibem sofisticações desnecessárias e exemplos impossíveis de serem bem entendidos. Essa literatura "superior" faz com que os programadores considerem a metodologia de Orientação ao Objeto algo distante e só acessível para alguns "iniciados", o que não é verdade. Portanto, é fundamental que o ensino de OOP nas escolas seja direcionado para um aprendizado de maneira suave, empregando uma boa didática, de modo a fazer com que o aluno entenda perfeitamente os objetivos desta técnica, porém, sem mitificá-la, para que possa ser assimilada naturalmente e aplicada na prática.

Bibliografia e Referências Bibliográficas

[1] FERREIRA, Marcelo; JARABECK, Flávio. CA Visual Objects-O Livro, SP, Express, 1995.

[2] FERREIRA, Marcelo; JARABECK, Flávio. Programação Orientada ao Objeto com Clipper 5.0, São Paulo, Makron Books, 1991.

[3] FURLAN, José David. Modelagem de Objetos através da UML: análise e desenho orientados a objeto. São Paulo, Makron Books, 1998.

[4] JONES, Capers. Produtividade no desenvolvimento de software. SP, Makron Books, 1991.

[5] LEITE, Mário. Programação Orientada ao Objeto - Uma Abordagem Didática - Monografia para Especialização em Análise de Sistemas. Maringá, CESUMAR, 1998.

[6] LEITE, Mário. Curso Básico de Delphi. Maringá, SYC, 1998.

[7] MARTIN, James. Princípios de Análise e Projetos baseados em objetos. Rio, Campus, 1994.

[8] MAURO, R.C; MATOSO M. L. Aspetos de Implementação de Servidores de Banco de Dados OO", In: XXIII Conf. Latino Americana de Informática - CLEI'97 pp. 29-38 Chile, 1997.

[9] ODEL, James J; MARTIN, James. Análise e Projetos Orientados ao Objeto. SP, M Books, 1996.

]10] PALADINI, Edson Pacheco. Gestão da Qualidade: Teoria e Prática. São Paulo, Atlas, 2000.

[11] PRESSMAN, Roger S. Engenharia de software. São Paulo, Makron Books, 1995

[12] PRICE, Tom. Programa de Especialização: opção para o Mestrado. Porto Alegre,UFRGS, 1997

[13] RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William, EDY, Frederick; LORENSEN, William. Modelagem e Projetos Baseados em Objetos. SP, Campus, 1994.