3.4. Programação Orientada a Objetos

Os conceitos de objetos e classes, vistos nas seções anteriores, são fundamentais na programação orientada a objetos. Complementando a definição destes conceitos, nesta seção métodos, mensagens e polimorfismo são detalhadamente descritos. Na seção 3.4.3 são apresentadas algumas questões de projeto referentes às Linguagens Orientada à Objetos.

3.4.1. Método e Mensagem

Conforme descrito anteriormente, cada objeto inclui um conjunto de métodos, através dos quais seu estado local pode ser acessado e alterado. Como já comentado, métodos diferenciam-se de procedimentos e funções porque podem automaticamente acessar os atributos do objeto (ao contrário do "mundo externo") e pode-se considerar que eles incluem um parâmetro implícito representando o próprio objeto. A chamada de um método de um objeto é, geralmente, considerada equivalente ao envio de uma mensagem ao objeto [LOU 93].

Para exemplificar estes conceitos, a partir de agora será utilizada a sintaxe da linguagem *Smalltalk*. *Smalltalk* permite três tipos de mensagens, unária, binária e *keyword*, distinguidas através do número de parâmetros e da maneira pela qual são especificadas. Mensagens unárias não possuem parâmetros e são especificadas através de identificadores com a primeira letra maiúscula para uma mensagem de classe e minúscula para uma mensagem de instância. Por exemplo: "*6 factorial*" e "*12 negated*". Mensagens binárias são nomeadas através de um ou dois caracteres especiais e sempre requerem um único parâmetro que segue o símbolo que representa a mensagem. Um exemplo de mensagem binária é "*6 + 5*".

Mensagens *keyword* consistem em um ou mais pares de parâmetros formais. Tanto as *keywords* quanto os parâmetros são identificadores, e a *keyword* é separada do seu parâmetro formal correspondente através de dois pontos. As *keywords* para métodos são escolhidas para fazer a leitura de uma mensagem tão "explícita" da sua função quanto possível. O nome da mensagem é sempre dado como uma seqüência de *keywords*, cada uma seguida por dois pontos. O número de parâmetros aceitos por uma mensagem é sempre igual ao número de dois pontos. A passagem de parâmetros é implementada por cópia, tendo em mente que os parâmetros são limitados a ponteiros para objetos. Exemplos de chamadas de mensagens *keyword* são: "*6 gcd: 21*" e " *'test string' at:3 put:$x* ".

Um método, por sua vez, é definido como uma seqüência de mensagens separadas por pontos. Duas características especiais na definição de um método *Smalltalk* devem ser observadas. Primeiro, um método pode conter variáveis que são locais à execução do método. Estas variáveis são declaradas na primeira linha com os nomes separados através de espaços e a lista inteira inclusa em barras verticais. Não é necessário especificar as classes nesta declaração, uma vez que classes de variáveis são determinadas dinamicamente. A segunda característica consiste na habilidade de especificar um valor de retorno para o método. Isto é feito através da coloação de uma "seta para cima" antes da descrição de um objeto. O objeto que segue a seta é então retornado como resultado da mensagem. A descrição do método a seguir fornece um exemplo destas características.

```
01      longident
02              "This method computes the longest substring of
03              identical characters within the receiving string"
04              | n long lastchar consec |
05      n ← 1.
06      long ← 0.
07      lastchar ← self at: 1.
08      consec ← 1.
09      [n <= (self size)] whileTrue:
10          [(lastchar = (self at: n))
11              ifTrue:[consec ← consec + 1]
12              ifFalse:[lastchar ← self at: n.
13                      (consec > long)
14                          ifTrue: [long ← consec].
15                      consec ← 1].
16          n ← n + 1].
17      ↑long
```

Na linha 04, quatro variáveis locais são declaradas. A amarração de localização para estas variáveis é feita durante a execução do método (amarração dinâmica). Como todas as variáveis *Smalltalk*, elas somente são amarradas a uma classe quando um objeto é atribuído a elas. A variável "*n*" é um índice para o *string*, que guarda a posição do caracter corrente durante a varredura do *string*. A variável "*long*" contém o tamanho do maior *string* de caracteres idênticos encontrados, "*lastchar*" contém o último caracter varrido e "*consec*" mantém um contador do número de caracteres idênticos consecutivos em qualquer ponto. A linha 17 indica que o objeto amarrado a variável "*long*" é retornado. Qualquer descrição de objeto precedida por uma seta para cima indica que não só o objeto é retornado, como também que o método é imediatamente terminado naquele ponto [DER 90].

3.4.2. Polimorfismo

Esta característica das linguagens de programação orientadas a objetos, referenciada pela expressão "uma interface, diversos métodos", consiste num tipo de polimorfismo fornecido através da amarração dinâmica de mensagens a definições de métodos. Isto é suportado através da permissão da definição de variáveis "polimórficas" do tipo da superclasse que também são capazes de referenciar objetos de qualquer uma das subclasses desta classe. A superclasse pode definir um método que é sobrescrito por suas subclasses. As operações definidas através destes métodos são similares, mas devem ser adaptadas a cada classe da hierarquia. Quando tal método é chamado através da variável polimórfica, esta chamada é amarrada ao método da classe apropriada dinamicamente.

Um objetivo desta amarração dinâmica é permitir que os sistemas sejam mais facilmente estendidos durante o seu desenvolvimento e manutenção, de forma que objetos com estruturas internas muito diferentes possam compartilhar a mesma interface externa e possam ser usados da mesma maneira. Tais programas podem ser escritos para executar operações em objetos de classes genéricas. Estas operações são genéricas no sentido de que elas podem ser aplicadas a objetos de qualquer classe relacionados através da derivação da mesma superclasse [SEB 99].

Em outras palavras, todas as classes derivadas da mesma superclasse podem ser vistas como versões especializadas da superclasse, e as variáveis polimórficas permitem referenciar objetos de classes diferentes. Por exemplo, de acordo com o código apresentado a seguir na linguagem de programação *C++*, é definida a superclasse "*Figura*", usada para armazenar as dimensões de vários objetos bidimensionais e para calcular suas áreas, e três subclasses "*Triângulo*", "*Quadrado*" e "*Círculo*", cujos objetos representam as figuras geométricas correspondentes. Os objetos destas classes podem entender o método "*mostra_area()*" da superclasse, que faz com que um objeto exiba a área da figura geométrica correspondente na tela. Entretanto, a resposta para o método "*mostra_area()*" é claramente diferente para os objetos "*Triângulo*", "*Quadrado*" e "*Círculo*". Assim, é possível criar uma instância de "*Figura*" e através dela referenciar objetos "*Triângulo*", "*Quadrado*" ou "*Círculo*". A chamada do método "*mostra_area()*" será vinculada dinamicamente à versão correta de "*mostra_area()*", escolhida pelo tipo ao qual a variável polimórfica está então referenciando. Neste caso, pode-se usar a mesma interface para as subclasses, embora estas forneçam seus próprios métodos para calcular a área de seus objetos [SCH 97].

```
class Figura  {
        protected:
                double x, y;
        public:
                void DefineDim(double i, double j) { x = i;  y = j; }
                virtual void MostraArea() { cout << "Nenhuma area de calculo definida ";
                                            cout << "para esta classe.\n";  }
};
class Triangulo : public Figura {
        public:
                void MostraArea() { cout << "O triangulo com altura ";
                                    cout << x << " e base " << y;
                                    cout << " tem uma area de ";
                                    cout << x * 0.5 * y << ".\n";  }
};
```

```
class Quadrado : public Figura {
        public:
                void MostraArea() { cout << "O quadrado com dimensoes ";
                                    cout << x << "x" << y;
                                    cout << " tem uma area de ";
                                    cout << x *  y << ".\n";  }
};

main() {
        Figura *p;          // cria um ponteiro para a superclasse
        Triangulo t;        // instancia objetos das subclasses
        Quadrado s;         //
        p = &t;
        p->DefineDim(10.0, 5.0);
        p->MostraArea();
        p = &s;
        p->DefineDim(10.0, 5.0);
        p->MostraArea();
}
```

O polimorfismo ajuda a reduzir a complexidade, permitindo que a mesma interface seja usada para especificar uma classe geral de ações. É trabalho do compilador selecionar a ação específica (isto é, o método) que se aplica a cada situação. O programador não precisa fazer essa seleção manualmente, basta apenas se lembrar da interface geral e utilizá-la.

3.4.3. Considerações para o Projeto de Linguagens Orientada à Objetos

Nesta seção são discutidas algumas questões relacionadas com a introdução de características de orientação à objetos às linguagens de programação. Inicialmente, pode-se considerar que um projetista de LP que está totalmente comprometido com o modelo orientado à objetos, projeta um sistema que "absorve" todos os outros conceitos de tipo. Tudo, desde o menor inteiro até o sistema completo, é um objeto. As vantagens desta escolha são a elegante uniformidade da linguagem e de seu uso, e a facilidade de reutilização de código. Uma desvantagem é que até operações simples devem ser feitas através do processo de troca de mensagens, o que geralmente as torna mais lenta do que no paradigma imperativo, onde instruções de máquina simples e rápidas implementam tais operações.

No modelo puramente OO todos os tipos são classes. Não há distinção entre classes pré-definidas e definidas pelo usuário. Na verdade, todas as classes são tratadas da mesma maneira e todo processamento é realizado através da troca de mensagem. Uma alternativa para o uso exclusivo de objetos, que é comum no paradigma imperativo cujo suporte a programação OO foi adicionado, consiste em manter o modelo de tipos imperativo e simplesmente adicionar o modelo OO. Isto resulta numa linguagem "maior" cuja estrutura pode se tornar confusa [SEB 99].

A introdução de classes em uma linguagem como tipos de dados significa que as classes devem ser incorporadas de alguma maneira no sistema de tipos. Entre as várias possibilidades, destacam-se:

1. Classes podem ser especificamente excluídas da verificação de tipos. Objetos tornam-se, então, objetos sem tipo, cujos membros da classe podem ser mantidos separadamente do sistema de tipos - geralmente em tempo de execução usando um mecanismo de união. Este método de adicionar classes à uma linguagem existente é o mais simples e tem sido usado em poucas linguagens, tal como *Objective-C*.
2. Classes podem ser construtores de tipos, fazendo assim com que as declarações das classes tornem-se parte do sistema de tipos da linguagem. Esta é a solução adotada por *C++* e por outras LP que adicionaram facilidades de OO às linguagens existentes. Em *C++* uma classe é somente uma forma diferente de uma estrutura, por exemplo. Neste caso, a compatibilidade da atribuição de objetos de subclasses para objetos de superclasses deve ser incorporada nos algoritmos de verificação de tipos.

3. Classes podem também simplesmente tornarem-se o sistema de tipos. Isto é, todos os outros tipos estruturados são excluídos da linguagem. Esta é a abordagem de muitas linguagens projetadas desde o início como OO, tais como, *Smalltalk* e *Eiffel* [LOU 93].

Conforme já descrito, algumas linguagens fornecem mecanismos para o controle de acesso às características das classes tanto para os objetos como para subclasses. Por exemplo, em *C++* membros de uma classe podem ser declarados como públicos, protegidos ou privados. *Eiffel* possui um mecanismo similar. Normalmente, classes tornam-se disponíveis aos clientes através da sua localização numa biblioteca, e os objetos importam características da classe simplesmente utilizando-as [LOU 93].

Torna-se interessante comentar que se apenas a interface da superclasse está visível para a subclasse, a herança é chamada de **herança de interface**. Se os detalhes da implementação também estão visíveis, a herança é chamada de **herança de implementação**. Existem vantagens e desvantagens nestas duas abordagens que devem ser consideradas. A garantia de acesso das subclasses à parte "escondida" da superclasse torna a subclasse dependente destes detalhes. Qualquer alteração na implementação da superclasse irá requerer uma nova compilação da subclasse, e em muitos casos, a alteração irá requerer uma modificação da subclasse. Isto acaba com a vantagem de esconder a informação dos objetos das subclasses. Por outro lado, esconder a parte de implementação da superclasse das subclasses pode gerar uma ineficiência na execução das instâncias destas subclasses. Isto pode ser causado pela diferença de eficiência entre o acesso direto às estruturas de dados e a exigência de acesso por meio de operações definidas na superclasse. Por exemplo, considerando uma classe que define uma pilha e uma subclasse que deve incluir uma operação para retornar o segundo elemento do topo da pilha. Se a linguagem usa herança de implementação, esta operação pode ser definida para simplesmente retornar o elemento daquela posição da pilha. Entretanto, considerando herança de interface, o código necessário será algo semelhante a:

```
int second () {
        int temp = top();
        pop();
        int temp_result = top();
        push(temp);
        return temp_result;
}
```

Este processo é claramente mais lento do que o acesso direto ao segundo elemento do topo da pilha. Entretanto, se a implementação da pilha é alterada, este método provavelmente necessitará ser alterado.

Outra questão relacionada à herança, é se a linguagem permite herança múltipla ou não (seção 3.3). O objetivo da herança múltipla é permitir que uma nova classe herde características de duas ou mais subclasses. Por exemplo em *Java* é comum escrever *applets* que incluem animação. Tais animações geralmente executam concorrentemente com outras partes da *applet*. *Applets* são suportadas através da classe "*Applet*" e a concorrência é suportada através da classe "*Thread*". Neste caso, a herança destas duas classes será necessária. Entretanto, conforme descrito anteriormente, existem algumas desvantagens para não incluir a herança múltipla numa LP.

O polimorfismo consiste no uso de uma variável polimórfica ou referência para acessar um método cujo nome é sobrescrito na hierarquia de classes que define o objeto ao qual o ponteiro ou referência está apontando. A variável polimórfica é do tipo da superclasse, que define a interface do método que é sobrescrito através da subclasse, e pode referenciar objetos da superclasse e da subclasse. Assim, a classe do objeto para qual ela aponta não pode ser sempre determinado estaticamente. A amarração de mensagens a métodos que são enviadas através de variáveis polimórficas deve ser dinâmica. A questão aqui é quando a verificação de tipos desta amarração é feita.

Como pode-se observar, a amarração de mensagens a métodos em uma hierarquia é parte essencial da programação OO. A questão é saber quando todas as amarrações de mensagens a métodos são dinâmicas. Uma alternativa é permitir ao usuário especificar quando uma amarração específica deve ser dinâmica ou estática. A vantagem é que a amarração estática é mais rápida, então se uma amarração não necessita ser dinâmica, não há necessidade de usá-la [SEB 99].

3.4.5. Considerações de Implementação em Linguagens Orientada à Objetos

Nesta seção são discutidas algumas questões sobre implementação em linguagens OO. Inicialmente, considerando-se a implementação de objetos e métodos, pode-se dizer que objetos são implementados como registros em *C* ou *Pascal*, com variáveis de instância representando os campos de dados do registro. Um objeto de uma subclasse pode ser alocado como uma extensão do objeto de dado anterior, com espaço alocado para as novas variáveis de instância no final do registro.

Métodos também podem ser implementados como funções, porém com duas diferenças básicas. A primeira é que um método pode acessar diretamente os dados do objeto da classe. Neste caso, pode-se ver um método como uma função com um parâmetro extra implícito, que é um ponteiro setado para a instância corrente em cada chamada. Um problema mais significativo ocorre com a amarração dinâmica dos métodos durante a execução. Na implementação dos objetos descrita, é alocado espaço somente para variáveis de instância de cada objeto; alocação de métodos não é fornecida. Isto ocorre porque métodos são equivalentes às funções, e a localização de cada uma é conhecida em tempo de compilação.

Quando a amarração dinâmica é usada para métodos, ocorre um problema, pois o método preciso que deve ser chamado não é conhecido antes do tempo de execução. Uma solução é manter todos os métodos "virtuais" como campos extras nas estruturas alocadas para cada objeto. Porém, cada objeto deve manter uma lista de todas as funções virtuais disponíveis no momento que pode tornar-se muito grande. Uma alternativa, então, consiste em manter uma tabela de métodos virtuais para cada classe em uma localização central, tal como uma área de armazenamento global, que pode ser estaticamente carregada com um ponteiro para esta tabela armazenado em cada estrutura do objeto.

Linguagens OO também podem manter um ambiente de tempo de execução da mesma maneira que *Pascal* ou *C* possuem uma pilha/*heap*. Neste ambiente é possível alocar objetos tanto na pilha como no *heap*, mas desde que objetos são dinâmicos por natureza, é mais comum eles serem alocados no *heap* do que na pilha. Além disso, em algumas LP OO, objetos são vistos implicitamente como ponteiros, que devem ser alocados no *heap* e inicializados. Porém, em *C++*, por exemplo, um objeto pode ser alocado diretamente na pilha ou como um ponteiro (*heap*). No primeiro caso o compilador é responsável pela chamada automática do construtor e do destrutor, enquanto que no caso da alocação como ponteiro, chamadas explícita para *new* e *delete* fazem com que chamadas ao construtor e destrutor sejam listadas. *Eiffel* e *Simula*, por sua vez, não possuem rotinas de desalocação explícita, mas requerem o uso de um *garbage collector* [LOU 93].

Finalizando, pode-se dizer que a programação OO é um estilo de programação eficiente para muitas situações. Entretanto, o projeto de grandes sistemas com herança é uma atividade difícil. Características das linguagens de programação podem ajudar na implementação de bons projetos, mas não removem as deficiências de um mau projeto. Mais importante ainda, projetos não são necessariamente bons ou ruins. Na prática o desenvolvimento inicial de um projeto não é o maior problema. O projetista pode construir uma hierarquia que soluciona o problema, entretanto mais tarde podem surgir dificuldades durante a manutenção, quando novas classes necessitam ser definidas. Se uma hierarquia necessita ser alterada significativamente, o impacto no resto do sistema pode ser significativo também.