PROGRAMAÇÃO ORIENTADA A OBJETOS

1. INTRODUÇÃO

1.1 UM BREVE HISTÓRICO DE LINGUAGENS DE PROGRAMAÇÃO

Os caminhos da programação têm mudado dramaticamente desde a invenção do computador. A primeira razão para as mudanças é acomodar o aumento da complexidade dos programas. Por exemplo, quando os computadores foram inventados, a programação era feita por chaveamentos em instruções binárias de máquina, usando-se o painel frontal. Enquanto os programas continham somente algumas centenas de instruções, esse método funcionava. Quando cresceram, a linguagem assembly foi inventada para permitir a um programador manipular complexidades. A primeira linguagem de alto nível difundida foi, obviamente, FORTRAN. Ainda que o FORTRAN tenha dado um primeiro passo bastante considerável, é uma linguagem que somente torna os programas mais claros e fáceis de entender do que o assembly, sem introduzir mudanças consideráveis no estilo de programação.

Nos anos 60 nasceu a programação estruturada. Esse é o método estimulado por linguagens como C e Pascal. Usando-se linguagens estruturadas, foi possível, pela primeira vez, escrever programas moderadamente complexos de maneira razoavelmente fácil. Entretanto, com programação estruturada, quando um projeto atinge um certo tamanho, torna-se extremamente difícil e muito custoso efetuar sua manutenção e fazer qualquer modificação. A cada marco no desenvolvimento da programação, métodos foram criados para permitir ao programador tratar com complexidades incrivelmente grandes. Cada passo combinava os melhores elementos dos métodos anteriores com elementos mais avançados. Atualmente, muitos projetos estão próximos ou no ponto em que o tratamento estruturado não mais funciona. Para resolver esse problema, a programação orientada a objetos foi criada.

A programação orientada a objetos aproveitou as melhores idéias da programação estruturada e combinou-as com novos conceitos poderosos que levam a ver na tarefa de programação uma nova luz, permitindo que um problema seja mais facilmente decomposto em subgrupos relacionados. Então, usando-se a linguagem, pode-se traduzir esses subgrupos em objetos.

A primeira linguagem a incorporar facilidades para definir classes de objetos genéricos na forma de uma hierarquia de classes e subclasses foi a linguagem Simula, que foi idealizada em 1966, na Noruega, como uma extensão da linguagem ALGOL 60.

Uma classe em Simula, é um módulo englobando a definição da estrutura e do comportamento comuns a todas as suas instâncias (objetos). Como o nome indica, é uma linguagem adequada à programação de simulações de sistemas que podem ser modelados pela interação de um grande número de objetos distintos.

As idéias de Simula serviram de base para as propostas de utilização de Tipos Abstratos de Dados, e também para Smalltalk. Smalltalk foi desenvolvida no Centro de Pesquisas da Xerox durante a década de 70, e incorporou, além das idéias de Simula, um outro conceito importante, devido a Alan Kay, um de seus idealizadores: o princípio de objetos ativos, prontos a "reagir" a "mensagens" que ativam "comportamentos" específicos do objeto. Ou seja, os objetos em Smalltalk deixam de ser meros "dados" manipulados por "programas", e passam a ser encarados como "processadores idealizados" individuais e independentes, aos quais podem ser transmitidos comandos em forma de "mensagens".

Outras linguagens orientadas para objetos tem sido desenvolvidas. Notadamente C++, uma extensão de C, Objetive-C, outra extensão de C, menos popular que a anterior, Pascal orientado a objetos, Eiffel e mais recentemente, no Brasil, TOOL.

Além da Xerox, que criou a ParcPlace Systems especialmente para comercializar Smalltalk-80 e seus sucedâneos (objectWorks), a Digitalk lançou em 1986 uma versão de Smalltalk para ambiente DOS, e mais recentemente a versão para Windows, o que contribuiu para uma maior difusão da linguagem [Digitalk].

Smalltalk, uma das mais populares linguagens orientadas a objetos, assim como outras linguagens orientadas para objetos, tem sido usada em aplicações variadas onde a ênfase está na simulação de modelos de sistemas, como automação de escritórios, animação gráfica, informática educativa, instrumentos virtuais, editores de texto e bancos de dados em geral, entre outras. Tais aplicações diferem substancialmente daquelas em que a ênfase está na resolução de problemas através de algoritmos, tais como problemas de busca, otimização e resolução numérica de equações. Para essas aplicações, é mais adequado o uso de linguagens algorítmicas convencionais, como Pascal, Algol e Fortran.

1.2 PROGRAMAÇÃO ORIENTADA A OBJETOS

Uma das atividades mais interessantes em Informática é certamente a busca constante de melhorias nas linguagens e técnicas para o desenvolvimento de software. Desta busca decorrem as transformações e evoluções das linguagens de programação, surgindo novas linguagens e novos paradigmas.

A Programação Orientada a Objetos utiliza os conceitos que aprendemos no jardim de infância: objetos e atributos, todos e partes, classes e membros. É difícil explicar por que demoramos tanto a aplicar estes conceitos à análise e especificação de sistemas de informações - talvez porque estivéssemos ocupados demais "seguindo a boiada" durante o auge da análise estruturada para imaginar que havia alternativas.

A Enciclopédia Britânica afirma:

Na compreensão do mundo real, as pessoas empregam constantemente três métodos de organização, sempre presentes em todos os seus pensamentos:

(1) Diferenciação, baseado na experiência de cada um, de objetos particulares e seus atributos - quando distinguem uma árvore, e seu tamanho ou relações espaciais, dos outros objetos,

(2) Distinção entre objetos como um todo e entre suas partes componentes - por exemplo, quando separam uma árvore dos seus galhos, e

(3) Formação de, e distinção entre, as diferentes classes de objetos - por exemplo, quando formam uma classe de todas as árvores, uma outra classe de todas as rochas e distinguem-nas.

A Programação Orientada a Objetos se apóia nestes três métodos usuais de organização.

Programação Orientada a Objetos é a programação implementada pelo envio de mensagens a objetos. Cada objeto irá responder às mensagens conhecidas por este, e cada objeto poderá enviar mensagens a outros, para que sejam atendidas, de maneira que ao final do programa, todas as mensagens enviadas foram respondidas, atingindo-se o objetivo do programa. Programação Orientada a Objetos, técnicas e artefatos ditos "orientados a objetos" incluem linguagens, sistemas, interfaces, ambientes de desenvolvimento, bases de dados, etc.

No entanto, cabe ressaltar que o conceito de Orientação Objeto depende mais da mentalidade do programador do que da linguagem de programação que está sendo utilizada. Pode-se conseguir programas razoavelmente orientados a objeto em linguagens tipicamente estruturadas, assim como pode-se conseguir programas estruturados em linguagens voltadas para objetos. Tomemos como exemplo a frase:

"O navio atraca no porto e descarrega sua carga."

se analisássemos esta frase estruturadamente, pensaríamos logo em como o navio atraca no porto e como ele faz para descarregar sua carga, ou seja, pensaríamos na ação que está sendo feita (que na frase é representada pelos verbos) para transformá-la em procedimento. Em orientação objeto, o enfoque com que se encara a frase é diferente: primeiro pensaríamos no objeto navio, no objeto porto e no objeto carga, pensando como eles seriam e procurando definir seu comportamento. Após isto é que pensaremos em como o navio se relaciona com o porto e com a carga, e como o porto se relaciona com carga. De modo grosseiro, podemos dizer que ao analisarmos uma frase pensando estruturadamente, damos ênfase aos verbos, e pensando orientado a objetos, damos ênfase aos substantivos.

Pelo que foi visto acima, percebe-se que o programador experiente terá inicialmente grande dificuldade em migrar para a orientação a objeto, pois terá que esquecer os anos de prática analisando os problemas estruturadamente, para reaprender a analisá-los de forma voltada a objeto. Ninguém faria isto se não tivesse bons motivos. Abaixo são mostradas algumas das vantagens que motivam veteranos programadores a readaptar-se para o paradigma de orientação a objeto:

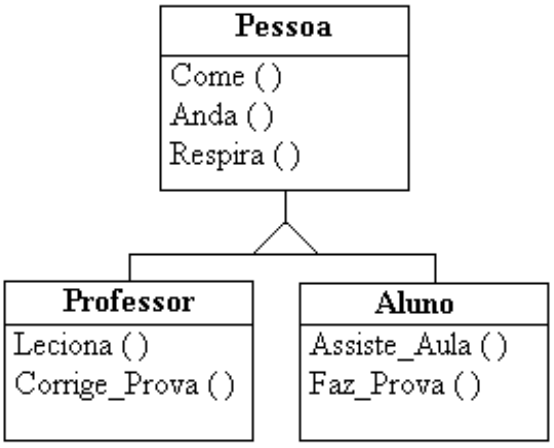
? ?Sensível redução no custo de manutenção do software
? ?Aumento na reutilização de código

?? **Redução no custo de manutenção:**

Na programação orientada a objetos, existem certas características (herança e encapsulamento) que permitem que, quando for necessária alguma alteração, modifique-se apenas o objeto que necessita desta alteração, e ela propagar-se-á automaticamente às demais partes do software que utilizam este objeto. Veremos mais detalhadamente o motivo desta propagação e os conceitos de herança e encapsulamento em capítulos adiante na apostila.

?? **Aumento na reutilização de código:**

Pode-se dizer, de modo simplório, que o conceito de orientação objeto fornece a possibilidade de um objeto acessar e usar como se fossem seus os métodos e a estrutura de outro objeto. Deste modo, quando, por exemplo, existirem dois objetos bastante semelhantes, com mínimas diferenças, pode-se escrever os métodos apenas uma vez e usá-los para os dois objetos. Apenas os métodos que realmente forem diferentes para os dois objetos é que precisam ser escritos individualmente. Observamos o exemplo abaixo:



Neste caso, tanto PROFESSOR quanto ALUNO comem, andam e respiram, e foi necessário escrever o código do método Come ( ), Anda ( ) e Respira ( ) somente uma vez. Apenas os métodos específicos de PROFESSOR (no caso, Leciona ( ) e Corrige_Prova ( )) e de ALUNO (Assiste_Aula ( ) e Faz_Prova ( )) é que precisam ser escritos

Atualmente existe uma grande discussão sobre o que é exatamente uma linguagem orientada a objetos. É importante lembrar que um claro conceito de objeto, não caracteriza um sistema como orientado a objetos. Ao invés de entrarmos na discussão do que seja uma linguagem orientada a objetos, vamos apresentar os principais mecanismos utilizados nestas linguagens, sem uma preocupação de explicitar quais deles seriam necessários para dar a uma linguagem o "Título" de Orientada a Objetos.

Este texto visa então, introduzir o leitor às principais idéias e conceitos de Programação Orientada a Objetos. Serão expostos exemplos de implementação em C++ e ilustrações que

permitirão um melhor entendimento do assunto. Nas páginas seguintes abordaremos os conceitos básicos do Paradigma de Orientação a Objetos: Objetos, Classes, Mensagens, Métodos, Herança, Generalização, Especialização, Abstração, Polimorfismo, Encapsulamento e Late-Binding. Conheceremos suas definições conceituais, como podem ser aplicados, quando devem ser utilizados e principalmente por que devemos usar estes conceitos e quais são suas vantagens e desvantagens sobre a programação estruturada.

2. CLASSE

Classe é o termo técnico utilizado em linguagens orientadas a objetos que descreve um conjunto de dados estruturados que são caracterizados por propriedades comuns. Também pode ser interpretado como uma estrututura modular completa que descreve as propriedades estáticas e dinâmicas dos elementos manipulados pelo programa.

Pode-se definir classes de objetos como a descrição de um grupo de objetos por meio de um conjunto uniforme de atributos e serviços. Uma classe é um conjunto de objetos que compartilham as mesmas operações.

Enquanto um objeto individual é uma entidade concreta que executa algum papel no sistema como um todo, uma classe captura a estrutura e o comportamento comum a todos os objetos que são relacionados. Um objeto possui uma identidade e suas características serão definidas para a classe.

Uma classe é definida por:

?? um nome da classe;
?? o nome da sua superclasse;
?? o nome de suas variáveis privadas;
?? os nomes e as definições de todas as operações associadas a esta classe;

Classe é um conceito estático: uma classe é um elemento reconhecido num texto de programa. por outro lado, um objeto é um conceito puramente dinâmico, o qual pertence não ao texto do programa, mas à memória do computador, local onde os objetos ocupam espaço durante a execução. (Conceitualmente, classes não são necessárias durante a execução, mas em linguagens interpretadas elas podem ser mantidas).

Exemplo de classe:

Uma classe é semelhante a uma *struct* e em C++ podemos definir a *classe fila* do seguinte modo:

```
class fila {
        int  f [100];
        int  primeiro, ultimo;
```

public:

```
        void   inicio (void);
        void   put (int valor);
        int      get (void)

};
```

Examinando a declaração anterior, vemos:
Uma classe pode contertanto partes públicas como privadas. Por exemplo, as variáveis *f* , *primeiro* e *ultimo* são privadas. Isto significa que não podem ser acessadas por qualquer função que nao seja membro dessa classe.

Para tornar públicas as partes de uma classe, ou seja, acessíveis o outras partes do programa, é preciso declará-las após a palavra **public.**

2.1 ATRIBUTOS

Um atributo é um dado para o qual cada objeto tem seu próprio valor.

Atributos são, basicamente, a estrutura de dados que vai representar a classe.

Exemplo de atributos, usando a *classe fila*:

```
int f [100] ;
int primeiro, ultimo;
```

2.2 MÉTODOS

Métodos são declarados dentro de uma classe para representar as operações que os objetos pertencentes a esta classe podem executar.

Um método é a implementação de uma rotina, ou seja, o código propriamente dito. Pode ser comparado a um procedimento ou função das linguagens imperativas.

Exemplo de métodos, utilizando a *classe fila*:

```
void iniciar (void)
{
        primeiro = 0;
        ultimo = 0;
};
```

```
void  put  ( int valor)
{
        f [ultimo] = valor;
        ultimo++;
};

int get (void)
{
        return  f [primeiro];
        primeiro++;
};
```

3. OBJETOS

O que caracteriza a programação orientada a objetos são os objetos. De um modo geral podemos encarar os objetos como sendo os objetos físicos do mundo real, tal como: carro, avião, cachorro, casa, telefone, computador, etc., por isso que às vezes é dito que orientação a objetos representa os problemas mais próximo ao mundo real, dando assim mais facilidade a programação como um todo, mais isso não é sempre verdade, porque às vezes temos problemas que são extremamente funcionais[1]. Nesses problemas funcionais é difícil representar a estrutura lógica em torno de objetos. Com isso, não são todos os problemas que giram em torno dos objetos facilmente visíveis.

De maneira simples, um objeto é uma entidade lógica que contém dados e código para manipular esses dados. Os dados são denominados como sendo atributos do objeto, ou seja, a estrutura que o objeto tem, e o código que o manipula denominamos método. Um método é uma função que manipula a estrutura de dados do objeto.

Um objeto é um ente independente, composto por:

?? **estado interno**, uma memória interna em que valores podem ser armazenados e modificados ao longo da vida do objeto.
?? **comportamento**, um conjunto de ações pré-definidas (métodos), através das quais o objeto responderá a demanda de processamento por parte de outros objetos.

Por exemplo:

Uma tela de computador pode ter os seguintes atributos e métodos:

atributos
modo de operação /* texto/gráfico */

[1]funcionais: gira em torno de processos

tamanho horizontal
tamanho vertical
paleta de cores
cor atual
métodos
modo texto ( )
modo gráfico ( )
fecha modo gráfico ( )
muda cor ( )
escreve caracter ( )
coloca pixel ( )
muda dimensões (x,y)
...

Um guarda-roupa:

estrutura
conjunto de roupas /* tipo, tamanho, cor, estilo, preço, etc. */
portas
número de portas
capacidade máxima

métodos
abre porta ( )
fecha porta ( )
escolhe roupa ( )
tira roupa ( )
coloca roupa ( )
estado do guarda-roupa ( ) /* portas abertas e fechadas, quantidade de roupas, etc. */
...

Uma lista:

estrutura
(nodo e um apontador para um próximo nodo) Primeiro e atual

métodos
cria lista ( ) /* cria célula cabeça e inicializa */
próximo ( ) /*vai para o próximo elemento da lista */
insere ( ) /* insere um elemento na posição atual */
deleta ( ) /* apaga posição atual */
volta ao começo ( ) /* atual = primeiro */
...

Podemos notar que um objeto é composto por estrutura e processos, onde esses processos giram em torno da estrutura, ao contrário das linguagens funcionais, nas quais a estrutura se adapta a função. Um objeto só pode ser manipulado por sua estrutura e seus métodos, nada mais do que isso.

Somente um objeto de um determinado tipo pode acessar seus métodos e estrutura, um outro tipo de objeto não tem nenhum acesso a estes. Por exemplo, em uma classe cachorro temos o método **fala**. Se por exemplo definirmos um objeto da classe gato, este objeto não tem acesso nenhum ao método **fala** de cachorro.

Dentro de um objeto, alguns métodos e/ou estrutura podem ser privados ao objeto, o que nos diz que são inacessíveis diretamente para qualquer elemento fora dele, o que impede que outros objetos tenham acesso direto às partes privadas do objeto referenciado. Para o objeto poder referenciar seus elementos privados ele deve passar pelos seus métodos, neste caso um método específico que faça a operação desejada, ou seja, ele pode acessar sua estrutura privada somente através de seus métodos, dando assim uma certa abstração de como é feita a manipulação da estrutura. Isso consiste no encapsulamento de dados que será explicado na seção referente a este tema. A princípio toda a estrutura deve ser privada, mas algumas linguagens como C++ permitem que a estrutura de um objeto possa ser acessada diretamente por outros objetos. Já em SmallTalk toda a estrutura é privada. Dessa maneira, um objeto evita significativamente que algumas outras partes não relacionadas de programa modifiquem ou usem incorretamente as partes privadas do objeto referenciado, dando assim maior confiabilidade na manipulação do objeto. Isso tudo nos mostra uma característica muito grande para construção de módulos independentes e abstração ao usuário.

Mais exatamente, cada objeto é uma instância de sua classe. É a classe que contém a descrição da representação interna e dos métodos comuns a todas as suas instâncias (objetos). Cada instância da classe, por sua vez, possui sua própria memória privativa (seu estado interno) onde ficam armazenados os valores de seus componentes, que representam suas características individuais. Associando com a linguagem C, uando você define uma estrutura como por exemplo:

```
struct aluno {
        char nome [30];
        char telefone [20];
        int número;
};
```

Quando você declara uma variável do tipo struct aluno você define uma instância da estrutura aluno.

```
main ( ) {
    struct aluno a;  /* a é uma variável do tipo da estrutura aluno */
    ...
}
```

Agora quando define uma classe aluno

```
class Aluno {
        char nome [30];
        char telefone [20];
        int número;
        lista_notas notas;
        ...
public:
        insereNota (matéria, nota);
        listaNotas ( );
        ...
};
```

Definiremos uma variável aluno

```
void main ( ) {
        Aluno João;
        ...
}
```

Neste caso João é uma variável (instância) do tipo aluno, o qual denominamos de **objeto**, porque a classe aluno descreve uma estrutura que o caracteriza e métodos que o manipulam.

Com isso podemos diferenciar claramente uma classe de objeto. A classe é apenas um tipo de dado, que somente representa características comuns a determinados objetos. Uma classe pode ser comparada com uma estrutura, com apenas uma forma definida, mas nenhuma variável que a manipula. Para manipulá-las é preciso definir uma variável. Esta variável do tipo da classe é que é chamada de objeto.

Os objetos de uma determinada classe não são iguais. Por exemplo, podemos ter os objetos João e Pedro da classe Aluno, cada um vai ter um nome, telefone, número, notas, e uma posição na memória. A sua similaridade consiste apenas no fato de que possuem propriedades idênticas.

Uma coisa importante de um objeto é seu ciclo de vida, que engloba o momento em que é declarado até sua eliminação. No instante em que um objeto é declarado, é alocado um espaço em memória para ele e automaticamente executado seu construtor. A partir deste momento o objeto está pronto para ser usado. Esta fase vai até a eliminação do objeto. A sua eliminação pode ser de duas formas: a primeira, que todas as linguagens utilizam, elimina o objeto no final do programa se ele for global, no final de um módulo se for local, e no final de um bloco se for declarado dentro deste. A segunda forma de eliminação é chamada de Garbage Collection. Esta maneira de eliminação não é implementada em todas as linguagens e

não é uma característica somente de orientação a objetos. Garbage Collection consiste em eliminação pelo compilador do objeto/variável depois de sua última utilização. A partir do momento em que ele não é mais referenciado, passa a não existir mais na memória. Por exemplo, Garbage Collection é implementado em Lisp e SmallTalk enquanto que em C++ e Pascal não. Quando o objeto é eliminado ele automaticamente executa seu destrutor.

Em programação orientada a objetos, primeiramente o que deve ser identificado são os objetos que o problema requer (até mesmo os tipos simples de dados são vistos como objetos, porque têm estrutura e operações (métodos) que o manipulam). Por exemplo um objeto inteiro é comparado com outro, recebe uma atribuição, tem operações aritméticas. Esta nova concepção não é tão fácil de ser encarada, principalmente para quem já é experiente em programação imperativa. As principais dificuldades a princípio são a identificação dos objetos, e o tratamento do problema somente através de objetos.

4. MENSAGENS

Mensagens são requisições para que um objeto execute uma de suas ações. Cada objeto somente pode responder às mensagens que constem do seu protocolo. O protocolo de um objeto são as mensagens correspondentes as suas operações, além do protocolo de sua superclasse.

Os objetos interagem através de mensagens.

O atendimento de uma mensagem envolve a execução de algum tipo de código, ou seja, os métodos, sobre um dado associado àquela operação, ou seja, sobre os atributos.

Quando um objeto é criado, o acesso a suas características é feito através de mensagens. Para cada mensagem recebida pelo objeto, existe um método associado para respondê-la. Quando a mensagem estiver se referenciando a um atributo, o valor deste deve ser devolvido, e no caso de uma operação, o procedimento desta é executado.

As operações podem ter parâmetros de entrada e saída com tipos determinados. Esta característica, juntamente com o seu nome, definem a assinatura de uma mensagem.

Exemplo de mensagem, utilizando a *classe fila*:

seja a declaração de um objeto:

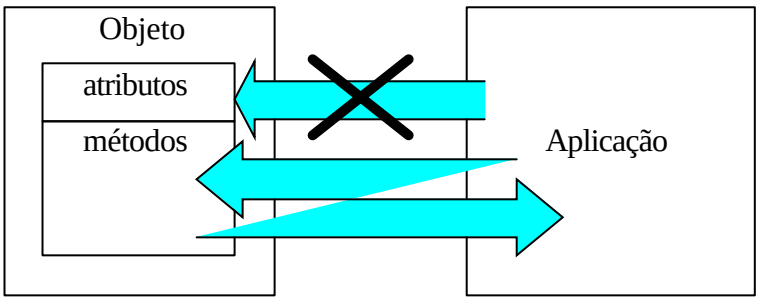
```
fila fila_atual;
.
.
.
```

A mensagem a este objeto seria:

fila_atual.get( );

5. ENCAPSULAMENTO

O conceito de encapsulamento é decorrente do fato de se combinar os dados (*atributos*) e o código que manipula estes dados (*métodos*) em um único *Objeto*. Ele garante que a única forma de acesso aos dados é através dos métodos disponíveis ao usuário (chamados *públicos*). Os demais métodos e os atributos da classe ficam sendo *privados*, ou seja, apenas funções-membro da classe têm acesso direto aos mesmos.



O conceito de encapsulamento não é exclusivo do paradigma de Orientação a Objetos. Ele já era utilizado na definição de Tipos Abstratos de Dados, para dizer que os dados só deveriam ser manipulados pelas funções que compunham o TAD.

Trocando em miúdos, o encapsulamento diz respeito à definição de uma estrutura que contenha os dados, defina quais os métodos de acesso público a esses dados e possua meios de proteger os demais métodos e os dados contra acesso direto.

Mas... por que impedir este acesso?

Responderemos esta pergunta com um exemplo prático.

Imaginemos um objeto *Polinômio*. Vamos supor que a estrutura de dados utilizada para representar um polinômio dentro do objeto seja um vetor. Este objeto, quando criado, recebe um string contendo o polinômio que ele representará (por exemplo, "x^2 + 2x +10") e possui, entre outros, um método para, dado um *x*, calcular *f(x)*.

```
typedef struct{
        float coeficiente;
        int expoente;
} termo;

class polinômio{
        termo    Vetor[30];

    public:
        polinomio(char *string);
        ~polinomio ( ) { };
        void insere (float coef, int exp);
        ...
        float calcula_fx (float x);
}
```

Digamos que o encapsulamento não existe e eu não tenho este método, mas eu estou fazendo uma aplicação que usa a classe *polinômio*, e preciso calcular *f(x)*. Eu posso então acessar diretamente o vetor dentro do objeto, obter o coeficiente e expoente de cada termo do polinômio e calcular *f(x)* sem mais problemas, não posso? Pode.

Mas um belo dia você chega à brilhante conclusão que um vetor não é a maneira mais econômica (em se tratando de memória) de se representar um polinômio, e resolve alterar a estrutura de dados para uma lista.

E agora, o que aconteceu com o código para calcular *f(x)*? Bem, você terá que implementar praticamente tudo de novo, só que agora com lista. E assim seria se a estrutura de dados fosse mudada de novo, e de novo, e de novo...

Agora vejamos o que acontece no mesmo objeto *polinômio*, mas agora **com** encapsulamento: A minha aplicação chama o método da classe *calcula_fx* e obtém o resultado desejado. Se eu mudar a estrutura de dados do objeto, precisarei fazer mudanças neste método, para que ele opere corretamente. **MAS**... a minha aplicação não mudou em nada!!! Isto quer dizer que se eu modificar o objeto sem alterar a interface dos meus métodos de acesso, não precisarei mudar uma linha na minha aplicação para que ela funcione corretamente. Isso só será possível se eu sempre acessar os dados através dos métodos da classe, e nunca diretamente.

É claro que não é sempre que eu consigo preservar a interface de um objeto quando da mudança da estrutura de dados, e às vezes a minha aplicação também precisará ser modificada. Entretanto, o Encapsulamento facilita estas mudanças, que não precisarão ser tão drásticas quanto no exemplo sem a aplicação deste conceito.

Além desse aspecto, ainda há o da proteção dos dados encapsulados. Restringir o acesso dos atributos aos métodos da classe garante que nenhum dado será alterado *por engano* ou de forma descontrolada. Por exemplo, se a lista do meu objeto polinômio fosse acessível à minha aplicação, nada impediria que eu, por engano, mudasse um apontador da

mesma e perdesse algum dado. O encapsulamento *protege* os dados, e faz o uso do objeto ser mais seguro.

Em outras palavras, o *Encapsulamento* garante que a minha classe seja uma *caixinha preta* para o usuário: ele não sabe o que há dentro do objeto, sabe apenas para que ele serve e quais os métodos disponíveis para a manipulação deste.

Note-se que este conceito possui efeito contrário para um objeto mal definido. Se, ao projetarmos uma classe, não fornecemos métodos de acesso adequados, teremos dificuldades em criar aplicações eficientes com objetos instanciados da mesma. Por exemplo, a nossa classe *polinômio* deveria possuir um método para informar se um termo de ordem *n* está presente no objeto. Caso contrário, fica difícil, por exemplo, implementar uma aplicação que compare dois polinômios e diga se possuem todos os termos de ordens iguais (ex.: $x^2 + 2$ e $3x^2 + 5$).

Logo, a tarefa de projetar uma classe envolve, entre outras atividades, definir da melhor maneira possível quais métodos de acesso ao objeto serão disponibilizados ao usuário. Um bom projeto inicial evita a necessidade de se redefinir uma classe já implementada.

6. HERANÇA

Herança é a propriedade dos objetos que permite a criação de uma hierarquia entre eles, onde os descendentes herdam o acesso ao código e estruturas de dados dos seus ancestrais.

Coad-Yourdon define Herança como:

Mecanismo para expressar a similaridade entre classes, simplificando a definição de Classes similares a outras que já foram definidas. Ela representa generalização e especialização, tornando atributos e serviços comuns em uma hierarquia de Classe.

Para entender o significado dessa definição, usaremos um exemplo simples. Digamos que eu tenha uma classe *Animal*. Todos os objetos da classe Animal possuem características (atributos) comuns, como peso, altura, idade, estados como ter fome, etc. Também fazem determinadas tarefas (serviços ou métodos) como Comer, Procriar, Nascer, Morrer, se movimentar, etc. Eu posso abstrair esse objeto Animal para reduzi-lo à seguinte representação:

```
class Animal {
        int             Peso, Altura, Idade;
        boolean         EstaComFome;
public:
        Animal( ) { ... }       /* Construtor */
        ~Animal( ) { ... }      /* Destrutor */
        boolean ComFome( );
        int Peso( );
        int Altura( );
```

...
}

Muito bem, então eu defini minha classe Animal e a usei no meu aplicativo. Digamos que a classe foi projetada e implementada de uma maneira ótima, que possui todos os atributos e classes necessárias para a representação de um animal qualquer.

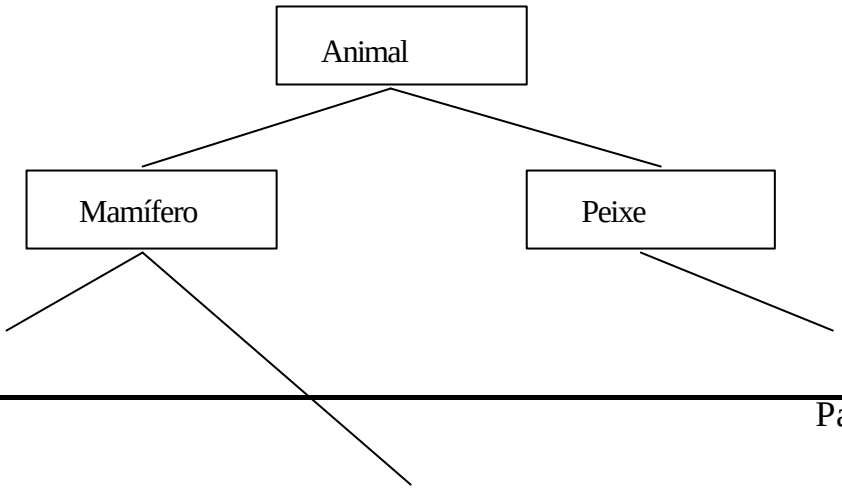
Digamos que eu tenha mais um aplicativo para fazer, que precise de uma Classe Mamífero. A representação de um mamífero é muito parecida com a de um Animal, mas ele possui atributos e métodos que não são comuns a todos os animais: mamíferos (mamar, emitir sons etc.), além de fazerem algumas atividades de forma diferente dos demais Animais (Nem todos os animais comem ou procriam como os mamíferos, por exemplo).

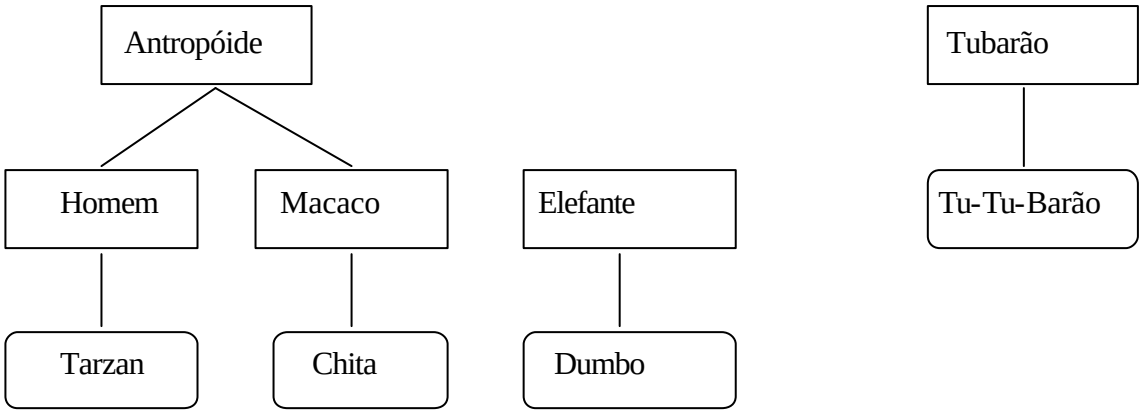
Numa linguagem de programação sem herança, a implementação da classe Mamífero provavelmente implicaria na replicação do código de Animal, com as modificações necessárias para caracterizar a classe. Se fosse necessário implementar uma classe Macaco, eu replicaria o código de Mamífero na nova classe. Uma classe Chimpanzé teria o código de Macaco replicado em si, e assim sucessivamente.

Isso não está bom. Se fosse necessário implementar todas as classes existentes entre o Ser Vivo e o Mosquito, teríamos centenas de classes, todas com replicação do código das classes anteriores. Além disso, se houvesse a necessidade de modificar algum método de Animal que fosse comum a todas estas centenas de classes, esta modificação teria que ser feita em todas elas, uma a uma. Será que não existe uma maneira melhor?

Na verdade, é fácil perceber que Mamífero é na verdade uma *especialização de Animal*, um *herdeiro* dessa classe. Assim, se eu dispuser de um mecanismo que me permita declarar Mamífero como tal, e assim *herdar* todos os métodos e atributos da classe ancestral, não precisarei replicar código, apenas incluirei os métodos e atributos específicos na classe Mamífero, e redefinirei os métodos que achar necessário. O mecanismo que me permite isso é a Herança.

A Herança vai produzir uma ordem de hierarquia entre as diversas Classes-Objetos que estiverem relacionadas desta forma. Um objeto herdeiro é em geral uma *especialização* do seu ancestral, que por conseqüência será uma *generalização* de seu sucessor. Eu posso montar uma estrutura de hierarquias entre Classes-Objetos baseada na relação generalização-especialização, resultando que os objetos mais ancestrais são mais genéricos ou abrangentes, e os seus sucessores são cada vez mais específicos, à medida que nos aprofundamos na estrutura. Observe como eu posso montar uma estrutura desse tipo baseado na classe Animal:

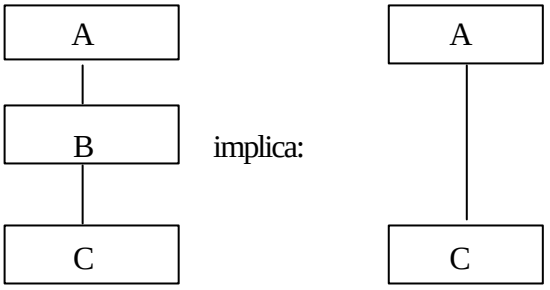




Nessa estrutura, podemos observar alguns níveis de especialização da classe mamífero. Mamíferos e Anfíbios são especializações de Animais: possuem as características comuns a todos os animais, mas comem e procriam de suas próprias maneiras, por exemplo. Antropóide é ancestral comum a Homem e Macaco, e reúne as características comuns a seus descendentes. Já Elefante é descendente (herdeiro) direto de Mamífero, pois não possui as características de um Antropóide. Homem e Macaco são herdeiros diretos de Antropóide e herdeiros indiretos de Mamífero e Animal, já que todo o antropóide é necessariamente um mamífero, que por sua vez é um animal. Peixe não é mamífero, logo não herda as características desta classe. Tubarão é herdeiro de Peixe, e herdeiro indireto de Animal. Nesta estrutura, as instâncias de classe foram representadas por retângulos com os cantos arredondados, e as classes por retângulos normais. As arestas representam as relações de herança entre as classes.

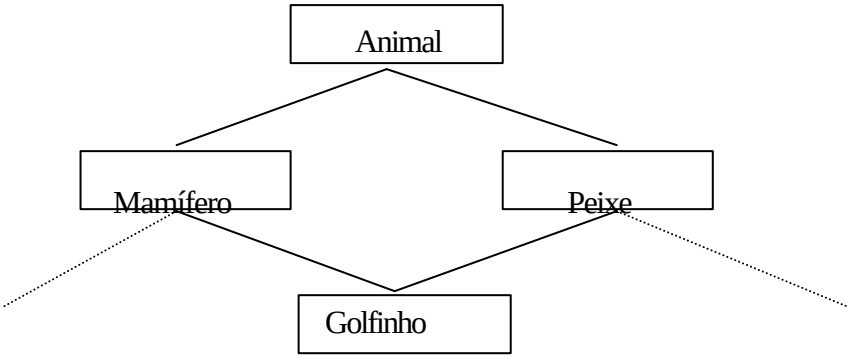
É importante ressaltar que seria possível ter instâncias de qualquer das classes desta estrutura, e não apenas das classes "folha". A princípio, qualquer destas classes é "instanciável".

Uma propriedade que fica clara aqui é a transitividade. Se a classe B herda de A, e a classe C herda de B, então C herda de A:



Por exemplo, Um Mamífero é um Animal, e um Antropóide é um mamífero. Logo, um Antropóide é um Animal, e possui as características de um animal.

Uma classe pode ser herdeira direta de mais de uma classe ao mesmo tempo. Isto é chamado de Herança Múltipla. A herança múltipla se aplica em situações como a do golfinho. Ele é um mamífero, mas possui características de Peixe. Assim, fazemos a classe Golfinho herdar de Peixe e Mamífero, e redefinimos os métodos específicos para Golfinho.



É importante ressaltar que normalmente não é preciso redefinir os métodos, e atributos herdados não precisam ser redeclarados ou redefinidos, a não ser que haja necessidade de refiná-los. Por exemplo, Antropóide e Homem respiram do mesmo jeito, logo eu vou utilizar o método Respira( ) de Antropóide para fazer Homem respirar. Entretanto Os mamíferos andam de formas diferentes, logo um método Anda( ) em Mamífero provavelmente terá que ser redefinido para Antropóide e Elefante, por exemplo.

A principal vantagem da herança em linguagens Orientadas a Objeto está no reaproveitamento de código que ela proporciona, pela declaração de classes herdeiras mais específicas ao meu problema. Do ponto de vista da Análise Orientada a Objetos, a herança permite uma melhor organização dos elementos que envolvem o Domínio do Problema, facilitando a compreensão do mesmo e a procura de soluções adequadas para o Sistema em questão.

A estrutura dos exemplos acima poderia ser implementada em C++ desta forma:

```
class Mamifero:public Animal {

        boolean querMamar;

        public:
                Mamífero( ):Animal( ) {}
                ~Mamífero( );
                void Mamar( );
                void Respirar( );
                ...
}

class Elefante:public Mamifero {
        public:
                Elefante( ):Mamífero( ) {}
```

```
                ~Elefante( );
                void LevantarTromba( );
                void Andar( );
                ...
}

class Antropoide:public Mamifero {
        public:
                Antropoide( ):Mamifero( ) {};
                ~Antropoide( );
                void Andar( );
                void Procriar( );
                ...
}


class Homem: public Antropoide {
        public:
                Homem( ):Antropoide( ) {};
                ~Homem( );
                 void Fala( );
                ...
}

class Macaco:public Antropoide {
        public:
                Macaco( ):Antropoide( ) {};
                ~Macaco( );
                void SobeNaArvore( );
                ...
}

class Peixe: public Animal {
        int número_de_escamas;
        public:
                Peixe( ):Animal( ) {};
                ~Peixe( );
                void Nadar( );
                void Comer( );
                ...
}

class Tubarao:public Peixe {
        public:
                Tubarao( ):Peixe( ) {};
```

```
                    ~Tubarao( );
                    void Comer( );
                    ...
        }

        class Golfinho:public Mamifero, Peixe {
            public:
                    Golfinho( ):Mamifero( ),Peixe( ) {};
                    void Comer( );
                    void Respirar( );
                    ...
        }

Homem Tarzan;
Macaco Chita;
Elefante Dumbo;
Tubarao Tu_Tu_Barao;
Golfinho Flipper;
void main(void)
{
        Tarzan.Comer( );
        Chita.Comer( );
        Tarzan.Respirar( );
        Chita.Respirar( );
        Dumbo.Respirar( );
        Dumbo.Andar( );
        Tanzan.Andar( );
        Chita.Mamar( );
        Flipper.Mamar( );
        Tarzan.Mamar( );
        Chita.SobeNaArvore( );
        Tu_Tu_Barao.Comer( );
        Flipper.Comer( );
}
```

Observe a declaração do método Comer( ): ele é declarado em Antropóide, e redefinido para Homem e para Macaco, já que estes comem de formas diferentes. No momento de enviarmos uma mensagem para o objeto Tarzan e para o objeto Chita, fazemos da mesma forma, já que os métodos possuem nomes idênticos, mas a mensagem a Tarzan resultará na execução do método Comer( ) de Homem, e a mensagem a Chita, na do método Comer( ) de Macaco. Já na chamada a Respirar( ), o método chamado tanto para Tarzan como para Chita e Dumbo é o mesmo, herdado de Mamífero.

É importante entender que cada objeto vai ter o seu próprio grupo de atributos. A chamada à Chita.Mamar( ) vai afetar as variáveis de Chita unicamente, e não de algum objeto Mamífero declarado. Homem e Macaco são mamíferos diferentes, objetos diferentes. Cada um

deles possui um atributo quer_mamar, herdado de Mamifero. Tarzan.Mamar( ) afeta quer_mamar de Tarzan, e não de algum objeto ancestral.

Também é importante não confundir Herança com Composição de Classes. Composição de classes é quando temos uma classe que possui como atributo um objeto de outra classe. Por exemplo, podemos dizer que o objeto carro possui pneus, e podemos colocar na sua lista de atributos quatro objetos Pneu:

```
class Carro{
        Pneu pneus[4];
        ...
        }
```

Neste caso, o objeto Carro *possui* objetos Pneu, mas não *herda* nada de Pneu. Se Carro *fosse* um Pneu, então poderíamos definir uma relação de herança, e fazer Carro herdar desta classe:

```
class Carro:public Pneu{
        ...
        }
```

É obvio que Carro *não é* um Pneu, logo esta definição é um absurdo, não faz sentido. Neste caso, nós precisamos compor a classe com objetos Pneu (a classe precisa de atributos Pneu), como mostrado no primeiro exemplo.

Assim, a pergunta-chave para distinguir herança de composição é: *A minha classe possui a classe X, ou ela é X?* Se ela possui, temos composição, se ela é, temos herança.

7. POLIMORFISMO

7.1 Definição:

Polimorfismo é a propriedade de uma ou mais classes responderem a mesma mensagem, cada uma de uma forma diferente. Numa linguagem orientada a objeto, uma referência polimórfica é tal que, no decorrer do desenvolvimento do software, refere-se a mais de uma classe. Desta forma é possível explorar similaridades entre diferentes classes de objetos. Este conceito é útil para distinguir mensagens de um método. Um objeto emissor envia uma mensagem, se o objeto receptor implementa um método com a mesma assinatura, ele poderá respondê-la. Diferentes respostas serão possíveis, dependendo de como os métodos dos receptores estão implementados.

No C++ o polimorfismo é implementado pelo uso de sobrecarga de funções. Em C++, duas ou mais funções podem compartilhar o mesmo nome, contanto que as suas declarações de parâmetros sejam diferentes. Nessa situação, as funções que compartilham o mesmo nome

são conhecidas como sobrecarregadas e o processo é chamado de sobrecarga de funções. Por exemplo, considere este programa:

```
#include <iostream.h>

// a função quadrado é sobrecarregada três vezes
int quadrado (int i);
double quadrado (double d);
long quadrado (long l);
main (void)
{
    cout << quadrado (10) << "\n";
    cout << quadrado (11.0) << "\n";
    cout << quadrado (9L) << "\n" ;
    return 0;
}
int quadrado ( int i)
{
    cout << "Dentro da função quadrado ( ) que usa ";
    cout << "um argumento inteiro.\n";
    return i*i;
}

double quadrado (double d)
{
    cout << "Dentro da função quadrado ( ) que usa ";
    cout << "um argumento double.\n";
    return d*d;
}

long quadrado (long l)
{
    cout << "Dentro da função quadrado ( ) que usa ";
    cout << "um argumento long.\n";
    return l*l;
}
```

Esse programa cria três funções similares, porém diferentes, chamadas de **quadrado.** Cada uma delas retorna o quadrado do seu argumento. Como o programa ilustra, o compilador sabe qual função usar em cada situação por causa do tipo de argumento. O mérito da sobrecarga de funções é permitir que conjuntos relacionados de funções sejam acessados usando-se somente um nome. Nesse sentido, a sobrecarga de funções deixa que se crie um nome genérico para algumas operações, com o compilador resolvendo exatamente qual função é necessária no momento para realizar a operação.

O que torna a sobrecarga de funções importante é o fato de ela poder ajudar à simplificar o entendimento do software. Para entender como, considere este exemplo. Muitos compiladores de linguagem C contêm funções como **atoi( ), atof ( )** e **atol( )** nas suas bibliotecas-padrões. Coletivamente, essas funções convertem uma string de dígitos em formatos internos de **inteiros, double** e **long**, respectivamente. Embora essas funções realizem ações quase idênticas, três nomes completamente diferentes devem ser usados em C para representar essas tarefas, o que torna a situação, conceitualmente, mais complexa do que o é na realidade. Ainda que o conceito fundamental da cada função seja o mesmo, o programador tem três coisas para se lembrar, e não somente uma. Entretanto, em C++, é possível usar o mesmo nome, **atonum ( )**, por exemplo, para todas as três funções. Assim, o nome **atonum( )** representa a ação geral que está sendo realizada. É de responsabilidade do compilador selecionar a versão específica para uma circunstância particular. Assim, o programador só precisa lembrar da ação geral que é realizada. Portanto, aplicando-se o polimorfismo, três coisas a serem lembradas foram reduzidas a uma. Ainda que esse exemplo seja bastante trivial, se você expandir o conceito, verá como o polimorfismo pode ajudar na compreensão de programas muito complexos.

Um exemplo mais prático de sobrecarga de funções é ilustrado pelo programa seguinte. Como você sabe, a linguagem C (e o C++) não contém nenhuma função de biblioteca que solicite ao usuário uma entrada, esperando, então, uma resposta. Este programa cria três funções, chamadas **solicitação( )**, que realizam essa tarefa para dados dos tipos **int, double** e **long:**

```
# include <iostream.h>

void solicitação ( char *str, int *i);
void solicitação ( char *str, double *d);
void solicitação ( char *str, long *l);
main (void)
{
    int i;
    double d;
    long l;

    solicitação (“Informe um inteiro: “, &i);
    solicitação (“Informe um double: “, &d);
    solicitação (“Informe um long: “, &l);
    cout << i << “ ” << d << “ “ << l;
    return 0;
}
void solicitação (char *str, int *i)
{
    cout << str;
    cin >> *i;
}
```

```
void solicitação (char *str, double *d)
{
    cout << str;
    cin >> *d;
}
void solicitação (char *str, long *l)
{
    cout << str;
    cin >> *l;
}
```

CUIDADO Você pode usar o mesmo nome para sobrecarregar funções não relacionadas, mas não deve fazê-lo. Por exemplo, você pode usar o nome **quadrado ( )** para criar funções que retornam o quadrado de um **int** e a raiz quadrada de um **double**. Entretanto, essas duas operações são fundamentalmente diferentes e a aplicação de sobrecarga de função, dessa maneira, desvirtua inteiramente o seu propósito principal. Na prática, você somente deve usar sobrecarga em operações intimamente relacionadas.

7.2 Tipos Clássicos de Polimorfismo:

1) De operadores (distinção pelo tipo do elemento): já vem implementado em todas as linguagens. Por exemplo, se você deseja somar dois números inteiros
2) Redefinição de operadores: utilizada quando necessita-se de operações que normalmente não são disponíveis na linguagem de programação. Por exemplo, soma de matrizes.
3) Dois métodos iguais na mesma classe: distinguem-se pelo número e/ou tipo de parâmetros ou objetos referenciados.

8. LATE BINDING

8.1 Definição

Late Binbing é um técnica que não esta ligada somente as linguagens Orientadas a Objetos, muitas outras linguagem usam esta técnica, principalmente as linguagens não tipadas. Cada linguagem que a implementa faz algum tipo de Binding, que não são necessariamente iguais. As linguagens orientadas a objetos interpretadas é que fazem com maior freqüência o Late Binding.

Não existe muita bibliografia que fale sobre essa técnica, pelo menos para linguagens orientadas a objetos. Geralmente é citada no decorrer da explicação dos conceitos de orientação objeto e/ou da linguagem, não ressaltando a técnica especificamente. No entanto, é

a técnica que permite a existência de muitos conceitos do paradigma orientado a objetos. As raras bibliografias que falam especificamente sobre o Late Binding geralmente ressaltam esta importância, que é enfocada porque um programa orientado a objeto não age de forma seqüencial como linguagens imperativas compiladas/interpretadas. Esta seqüência se refere mais ou menos a uma ordem de compilação.

Na programação orientada a objeto ao definir uma classe ela pode usar objetos que não foram definidos ainda e ser compilada normalmente. Esta característica está em linguagens orientadas a objetos porque um programa orientado a objetos é basicamente um programa com descrições de formas e ações de objetos e a interação entre eles, e essa interação não é necessariamente seqüencial.

Fazendo-se uma analogia, pode-se comparar estes tipos de declaração como protótipos de funções que a linguagem C oferece, ou em compilação, às DLL's (Dynamic Link Library) do Windows. Um programa que usa DLL é compilado independente da existência desta DLL. Somente em tempo de execução é verificado se a DLL esta presente ou não. Se esta não estiver presente, ocorrerá um erro.

Mas afinal qual é o conceito de Ligação Tardia? Não existe apenas uma definição deste tema, principalmente porque existem várias formas de Ligação Tardia e diferentes em cada linguagem, portanto não existe nenhum padrão para Ligação Tardia. Em algumas linguagens uma ferramenta é feita com a técnica de Ligação Tardia e em outras esta mesma ferramenta não utiliza esta técnica (ex. Polimorfismo em Smalltalk é feito em tempo de execução, já em C++ em tempo de compilação, uma ligação bem menos dinâmica). Estas definições cobrem praticamente todas as ligações que linguagens orientadas a objetos fazem.

?? É a técnica de adaptar objetos a determinadas situações;
?? É usar objetos definidos em módulos completamente independentes, de modo que não é necessário conhecer o estado do módulo, apenas a sua interface;
?? Escolha de uma propriedade num conjunto delas. Durante a processamento (interpretação/compilação) são definidos conjuntos de possibilidades.

8.2 Tipos

Existem os seguintes tipos de Binding:

?? **Execution Time (tempo de execução)**: o mais comum é em tipos de variáveis.
  ?? Entrada em um subprograma: o binding ocorre apenas na entrada de um subprograma, por exemplo em Pascal, onde as variáveis são tipadas no início do procedimento (aloca espaço para as variáveis).
  ?? Pontos arbitrários durante a execução: pode ocorrer em qualquer ponto da execução, em exemplo são linguagens onde não existem declarações de tipos.
?? **Translation Time (tempo de compilação)**: existem em todas as linguagens.

?? Ligação escolhida pelo programador: decisões tomada pelo programador que podem ser decididas durante o processamento (tipos de variáveis).
?? Ligação escolhida pelo processador: são ligações feitas pelo processador, em FORTRAN ligações de determinadas variáveis para determinada lugares são feitas em tempo de carregamento. (atribuições, default).

?? **Tempo de implementação da linguagem**: alguns aspectos da linguagem que usam ligação são definidas no desenvolvimento da linguagem. Exemplo: representação de números (10 10.0).

?? **Tempo de definição de linguagem**: muitas estruturas de linguagens são fixadas na sua definição. Por exemplo a possibilidade de alternativas formas, dados, estrutura e tipos são definidas pela linguagem. (o operador + em C é definido para inteiros e reais).

8.3 Ligação Precoce e Tardia (O. O.)

Há dois termos que são normalmente usados quando se discute linguagens de programação orientada a objeto. Ligação precoce e ligação tardia, relativos a linguagens compiladas, eles referem-se aos eventos que ocorrem no momento da compilação, e no momento da execução respectivamente,

Em ralação à orientação a objetos, ligação precoce significa que um objeto é ligado a sua chamada de função no momento da compilação, isto é , todos as informações necessárias para determinar qual função será chamada são conhecidas quando o programa é compilado. A principal vantagem da ligação precoce é a eficiência - é mais rápido e requer menos memória.

A ligação tardia significa que um é ligado a sua chamada de função na momento da execução. Isso determina no decorre da execução do programa qual função se relacionará com qual objeto. A vantagem é a flexibilidade.

8.3.1 Dynamic Typing E Dynamic Binding - O.O.

Dynamic Typing e Dynamic binding estão diretamente associados a estes termos, onde:

?? **Dynamic Typing (tipos dinâmicos)**: alocação de variáveis - o software pode usar um objeto e seus processos sem ele ter sido definido.
?? **Dynamic Binding (ligação dinâmica)**: a operação é associada com um objeto particular, é selecionada em tempo de execução.

A programação orientada para objetos esta associada com um estilo de programação constituída pela criação dinâmica de objetos e ligação dinâmica de operações a objetos. Este estilo surgiu com as linguagens orientadas a objetos interpretadas como Smalltalk e Flavors. Estes sistemas apresentam grande flexibilidade durante a execução, são ferramentas bem sucedidas para prototipagem e solução imediata de problemas e situações.

É possível escrever programas orientados para objetos em C++, onde nos oferece criação dinâmica de instâncias de objetos e ligação dinâmica de operações com objetos com funções virtuais. As chamadas de funções virtuais são vinculadas dinamicamente em tempo de execução, enquanto que o tipo de instância do objeto em tempo de compilação.

Exemplos

Agora passaremos alguns exemplos para deixar mais claro o sua definição e propósito.

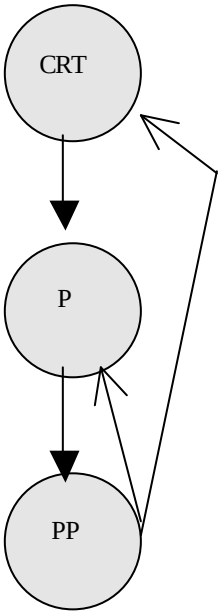
Ex1 - Uma comparação entre linguagens imperativas e O.O.

Programa imperativo

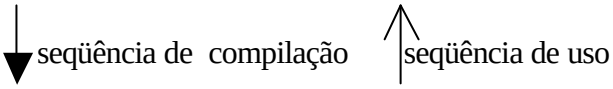
```
Program imperativo;
uses crt;
var i : integer;

Procedure P;
Begin
     {faz coisas}
end;

     { Programa Principal PP}
Begin
     clrscr;
     ...
     writeln(i); P ;
end.
```



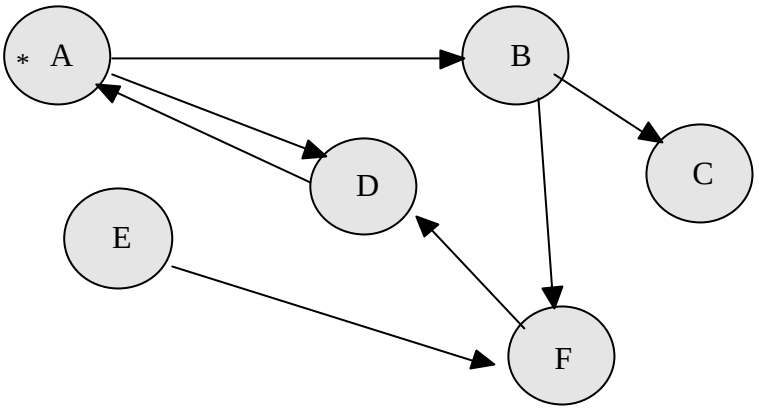
A compilação deve ser seqüencial



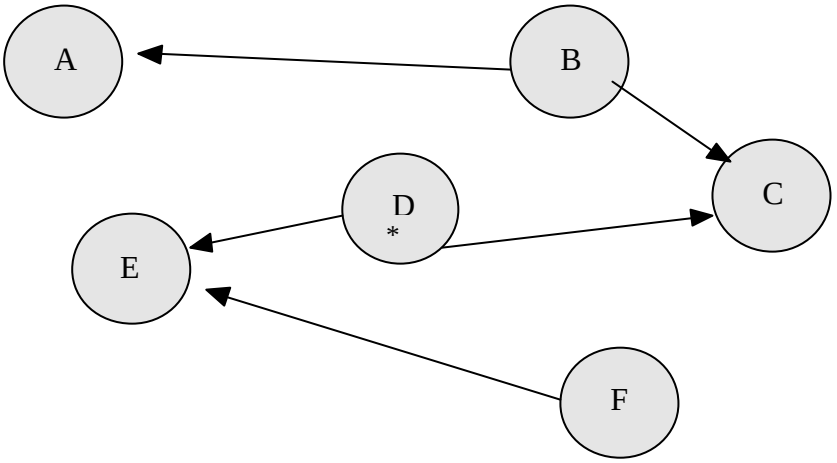
Programa O.O.

O desenvolvimento não é seqüencial, cada módulo é construído separadamente. A seguir mostraremos como funciona um programa orientado a objetos, onde as setas indicam a troca de mensagens e o * onde começa o processo.

Ora uma seqüência pode se aplicado como no exemplo seguinte:



Ora como no exemplo a seguir, e assim sucessivamente:



8.4 Conclusões

O programa que usa ligação tardia não depende de sua aplicação. Na prática, muitos programas grandes usarão uma combinação das duas ligações (precoce e tardia). Ligação tardia é uma das ferramentas mais poderosas de orientação a objetos. Porém, o preço que se paga por essa potencialidade é a lentidão do programa. Portanto deve-se usar ligação tardia somente quando acrescentar significativamente estruturação e flexibilidade ao programa. A perda de performance é muito pequena, da maneira que , quando a situação pedir ligação tardia, deve-se definitivamente usa-la.

Varia de linguagem para linguagem. As principais vantagens desta técnica é que nos permite facilidade de programação, uma maior reutilização de código e independência no desenvolvimento de módulos.

9. BIBLIOGRAFIA

Análise e Projeto Orientado a Objeto - Jaelson Freire Brelaz de Castro - [94]
Programação Orientada a Objetos - Tadao Takahashi e Hans K. E. Liesenberg - [90]
Smalltalk 80 - A. Goldberg & Robson
Smalltalk V - Digitalk
Introdução à Programação Orientada a Objeto - Miguel Jonathan [94]
Análise Baseada em Objetos - Peter Coad & Edward Yourdon [92]
Programação em C++ - Uma Abordagem Orientada a Objetos - Ben Ezzel [91]
Projeto de Algoritmos - Nívio Ziviani [93]
O Paradigma de Objetos: Introdução e Tendências - Tadao Takahashi [89]
Programando em C++ - Steven Holzner & The Peter Norton Computing Group [93]
Programming language: concepts and paradigms - David A. Watt
Object Oriented Software Construction - Bertrand Meyer