METODOLOGIAS DE *DESIGN* DE CÓDIGO E ARMAZENAMENTO DE DADOS

Mário Leite[1]
Faculdades Integradas de Maringá
Centro de Ensino Superior de Maringá

RESUMO: Apesar da grande profusão de programas de computador para todos os segmentos, o *design* de *softwares* ainda é, na sua maioria, muito deficiente. Programadores e desenvolvedores de produtos do tipo comercial, científico, educacional ou mesmo de jogos, muitas vezes produzem sistemas que apesar de funcionais, não são corretos, necessariamente. Muitos produtos, embora atendam os clientes, não comportam uma estrutura adequada de programação para resistir a um mercado cada vez mais competitivo. Essa competitividade passa, necessariamente, por uma questão muito importante sobre um *software*: a manutenção. Outra questão, também muita séria, é o armazenamento dos dados em bancos de dados, uma vez que mais de 75% de todos os sistemas fazem uso intensivo desse tipo de arquivo para armazenamento. Por isso o desenho do *software* e o tipo de armazenamento e pesquisa dos dados são fundamentais para que um sistema seja considerado de ponta, com qualidade total para a satisfação plena do cliente.

Descritores: banco de dados; *design*; linguagem; manutenção; objeto; *software*.

METHODOLOGY DESIGN OF DATA CODE AND STORAGE

ABSTRACT: *In spite of the great profusion of computer programs in all segments, the majority of software is still very deficient. Programmers and developers of commercial, scientific and educational products or even games have been producing systems that are functional but not necessarily correct. Many products, although answering to clients needs, do not hold an adequate programmation structure to withstand a more and more competitive market. This competition involves a very serious issue in software: maintenance. Another very serious question is about the storage of data in data banks, since 75% of all systems use these types of devices. Therefore, the design of software and the type of storage and search of data are fundamental to a system to be considered top, with total quality for the fully satisfaction of the client.*

Index Terms*: databases; design; language; maintenance; object; software.*

Introdução

Este artigo enfoca as três principais metodologias de implementação e como elas podem influir no desempenho de um sistema comercial em termos de manutenibilidade e performance computacional. As metodologias *Design Top-Down*, *Design Bottom-Up* e ***O****bject-****O****riented* ***P****rogramming* são analisadas e comparadas quanto à maneira de desenhar um *software* e quanto ao problema da reutilização de código e manutenção do sistema. A questão da linguagem de programação e banco de dados também é analisada de forma a se ter uma visão de integração entre esses dois componentes num sistema, quando comparadas as três situações possíveis: linguagem estruturada com banco de dados relacional, linguagem orientada a objetos com banco de dados relacional e linguagem orientada a objetos com banco de dados orientado a objetos.

MAURO e MATOSO (1997) propõem o emprego da linguagem JAVA e servidor GOA^{++} com processamento distribuído para solucionar o problema de *impedance mismatch* e melhorar o problema do armazenamento dos dados de uma classe com o emprego de uma linguagem orientada a objetos (LOO) para implementação do código e um banco de dados orientado a objetos (BDOO).

É aventada a hipótese do uso de uma ferramenta rápida de desenvolvimento e de um banco de dados orientado a objetos, que possa resolver o problema do armazenamento dos dados e o descasamento de tipos entre a linguagem e sistema gerenciador de banco de dados.

A Questão Do Custo Do *Software*

Classicamente, a criação de um *software* envolve quatro etapas básicas: Análise, Projeto, Implementação e Implantação. Na maioria das vezes estas quatro etapas se confundem com a expressão "desenvolvimento", traduzindo o *design* de um *software* como algo único. Entretanto, uma questão ainda não é bem considerada pelos desenvolvedores: a manutenção. Dar manutenção num sistema é fundamental para manter sua funcionalidade ao gosto do cliente. A diminuição de manutenções corretivas também é muito importante na composição do custo final do produto para que ele possa ser competitivo no mercado globalizado.

[1] Docente das Faculdades Integradas de Maringá do Centro de Ensino Superior de Maringá. Especialista em Engenharia (Puc/RJ); Especialista em Análise de Sistemas (Faimar/Cesumar). Aluno regularmente matriculado no Curso de Mestrado em Engenharia da Produção convênio Faimar/Cesumar/Ufsc.
mario@cesumar.com

As curvas de evolução dos custos do *hardware* e do *software* versus tempos são exibidas na figura 1, simulando uma situação teórica, mas bem provável, caso não se atente para o problema da manutenção.

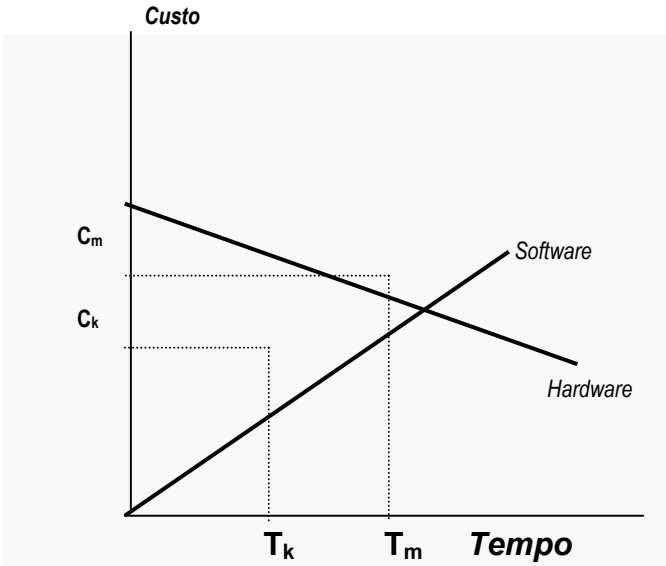


Gráfico 1: Evolução dos custos de *hardware* e *software*

A Figura 1 constata um fato que ocorria até bem pouco tempo: enquanto os custos do *hardware* decrescia com o tempo, os custos do *software* não seguia o mesmo processo. Num dado tempo T_k o custo do *software* era C_k , e num tempo T_m o custo correspondente a esse novo tempo era C_m, evidenciando um aumento nos custos.

Outro fato comprovado em laboratório é que os custos de desenvolvimento são praticamente fixos, não variando substancialmente a cada investimento num novo produto; isto quer dizer que o preço final do *software* depende mais dos custos de manutenção.

Desse, modo, ao se manter a evolução da curva, vê-se que o preço do produto aumenta(va) sempre, devido ao componente manutenção e não do desenvolvimento propriamente dito. Um dos componentes da solução para diminuir os custos com manutenções, é empregar uma boa metodologia de *design*, onde a reutilização de componentes deva ser prioritária, aliada a rotinas cujos códigos sejam bem legíveis e modulares, facilitando as possíveis manutenções futuras e evitando as corretivas.

Programação Estruturada

Programação Estruturada emprega a metodologia tipo “do todo para as partes”, permitindo ao programador um *design* de *software* mais limpo e bem mais legível do que os antigos códigos do tipo “spagetti”. Este tipo de programação é freqüentemente chamado de *Design Top-Down,* expressão que se refere ao fato de se poder "quebrar" - decompor - um grande sistema em partes cada vez menores, olhando "de cima para baixo". Esta metodologia permite que o programador atinja o ponto em que uma sub-rotina não precise mais ser decomposta para ser implementada como função ou *procedure* isolada.

A linguagem *Pascal* adota, fortemente, esse tipo de programação, a ponto de ser confundida com a própria técnica de desenvolvimento de algoritmos. A Programação Estruturada melhora, sensivelmente o *design* de *softwares*, fazendo com que o fluxo do programa siga um curso lógico e sem saltos incondicionais apresentados com o emprego do comando *GOTO*; comando esse que produz códigos quase ilegíveis, de manutenção difícil e bastante custosa, apesar de serem funcionais no contexto de alguns sistemas tradicionais.

O código-exemplo apresentado na Figura 2, LEITE (1996) mostra a codificação de um programa simples em *GWBASIC* (Basic antigo), onde são impressos os números primos menores que 100. O código é extremamente confuso e cheio de saltos com comandos *GOTO's,* criando um emaranhado de "idas e vindas" no programa, comprometendo seriamente a legibilidade e dificultando a manutenção. A solução para que esse código fique legível e de fácil manutenção é o emprego de estruturas bem definidas numa linguagem que as ofereça e interface gráfica para uma melhor interação com o usuário. Um exemplo simples de solução estruturada é mostrada na Figura 3, codificada em Visual Basic™ .

Todavia, apesar de ser um avanço muito grande na engenharia de *software,* como técnica de escrita de código, a Programação Estruturada não pode ser considerada um “divisor de águas” na programação, pois, mesmo sendo uma tecnologia muito importante -e ainda muito empregada- , ela tem sérios problemas a considerar. Um desses problemas, ainda persistente, é que, embora ofereça grandes níveis de decomposição - chegando a rotinas atômicas - não oferece nenhuma garantia de uma boa recomposição. Isto significa que mesmo que se consiga criar um grande sistema quebrando-o em módulos cada vez menores, é muito difícil reutilizar esses módulos em outros sistemas, uma vez que foram criados dentro de uma hierarquia bem específica. Por isso, mesmo sendo uma boa metodologia de codificação e definição das estruturas de dados, as manutenções corretivas ainda são problemáticas e a reutilização não muito satisfatória quando utilizamos o *Design Top-Down*.

```
10 REM  - Números primos menores que 100

15  CLS

20  LET  N = N + 1

30  IF  N=100   THEN   GOTO 120

40  LET  K = 1

50  LET  K = K + 1

60  LET  J = N/K

70  IF  INT(J)=J  AND  N<>2   THEN  GOTO 20

80  IF  k>=SQR(N)   THEN  GOTO 100

90  GOTO 50

100 PRINT  N

110 GOTO 20

20 END
```

Quadro 1: Código *não* estruturado para calcular primos

```
Private Sub cmdGeraPrimos_Click()
Dim n  As Integer, k  As Integer, j  As Single
'Números primos menores que 100
Cls
k = 2
n = 0
Do While (n < 100)
   n = n + 1
   j = n / k
   If  Int( j ) = j  And  n <> 2   Then
      k = 2
   Else
      If  k >= Sqr( n )  Then
         Print  n
         k = 2
      Else
         k = k + 1
         n = n - 1
      End If
   End If
Loop
End Sub
```

Quadro 2: Código estruturado para calcular primos

Programação Modular

Programação Modular, às vezes chamada de *Design Bottom-Up*, é uma metodologia em que o desenho do *software* é efetuado a partir de módulos independentes (ao invés de decomposição). Com esta metodologia são criados pequenos módulos que poderão ser agrupados para criar módulos cada vez maiores até compor novamente o sistema como um todo.

Como exemplo do mundo real, pode-se pensar na criação de um computador em que as peças são criadas independentemente uma das outras e encaixadas logicamente para compor o módulo principal. Isto mostra que dificilmente uma empresa constrói todas as peças que compõe um computador, pois implicaria em desenvolver chips, placas, fontes, BIOS, etc, o que inevitavelmente seria uma empreitada muito perigosa devido à incrível velocidade com que a Informática se renova a cada dia; esses componentes estariam obsoletos em pouco tempo. Ao invés disso, utiliza-se peças que se encaixam para produzirem novas peças, que podem ser produzidas até por terceiros.

Diz-se que a Programação Modular é componível, ao contrário da Programação Estruturada, uma vez que programar modularmente é criar pequenos módulos que se encaixam a outros com o objetivo de compor o programa. A essência desse tipo de programação baseia-se no fato de que a solução de um problema deve ser a mais genérica possível para que da próxima vez que ocorrer tal problema, uma solução (total ou parcial) já tenha sido encontrada anteriormente, diminuindo a introdução de código novo.

Essa técnica é chamada de reutilização, e muito embora leve mais tempo para criar programas modulares devido à procura de soluções genéricas, estaria sendo criado um código reutilizável; pois algumas partes do próximo programa já estariam prontas e depuradas. Assim, dentro de pouco tempo estaria disponível uma biblioteca de módulos genéricos que poderiam ser inseridos no próximo *software*, aumentando a produtividade e facilitando a manutenção.

Programação Orientada ao Objeto

Programação Orientada a Objetos (Object-Oriented Programming), ou simplesmente OOP, é uma extensão quase que natural da Programação Modular, aumentando a legibilidade do código e diminuindo em muito a necessidade de manutenção.

A origem da OOP vem da linguagem Simula -Simula Language-, concebida na Noruega no início da década de 60, e como o nome sugere, foi criada para fazer simulações. Todavia, seu uso aflorou um conceito que até então passava "despercebido" pela maioria dos projetistas: a similaridade com o mundo real. *SIMULA-68* implementou de maneira formal os conceitos de OOP, assim como *SmallTalk*, considerada a linguagem que popularizou e incentivou o emprego da OOP. Esta é considerada a melhor linguagem orientada ao objeto de fácil comunicação com os usuários devido ao fato de trazer dentro de si conceitos puramente do mundo real: objetos. O resultado foi uma linguagem muito poderosa que provê modelos exatos para transações comerciais, interações de sistemas, transferência de informações, etc.

A concepção da tecnologia de OOP é revolucionária, uma vez que é baseada em fatos do mundo real, permitindo que conceitos abstratos sejam tratados mais concretamente. *SmallTalk*, como outras linguagens orientadas a objeto, representa um avanço significativo na evolução da arte de desenhar *softwares*. A principal motivação para o uso dessa tecnologia é o fato dela se apoiar em exemplos do mundo real - não fictício. O uso dessa "nova-velha" técnica proporcionou ganhos significativos de produtividade, revertendo a tendência da curva *Custo do Software x Tempo*, que era crescente.

A engenharia de *hardware* está sempre projetando novos equipamentos, usando os mesmos componentes básicos: transistores, resistores, fusíveis, diodos, chips, etc. Isto é, estão sempre *reutilizando* os "objetos" já existentes para produzir novos "objetos". Por exemplo, um novo tipo de placa é composto de vários componentes que podem ser usados em outro equipamento; o mesmo acontece com um disco rígido ou um monitor de vídeo de alta definição.

Cada um desses componentes de *hardware* - transistor, resistência, diodo, etc - são como caixas pretas que uma vez fabricados podem ser usados para vários fins, de acordo com o seu projeto básico. O segredo de tudo isso é como as peças básicas são agrupadas para dar origem aos mais diferentes tipos de elementos.

A idéia é usar cada vez mais esse mesmo princípio na engenharia de *software*, com as *softhouses* criando vários códigos tipo "caixa preta" para serem adquiridos pelos desenvolvedores e "montados" tal como um quebra-cabeça na construção dos sistemas. Como conseqüência, a produtividade aumenta consideravelmente, uma vez que os códigos, objetos, adquiridos já estariam depurados e isentos de erros; basta que o desenvolvedor saiba encaixá-los logicamente no sistema. O resultado é a queda dos preços dos *softwares* com a diminuição drástica dos custos de manutenção.

Do ponto de vista da implementação, o paradigma da orientação ao objeto (OO) considera que, apesar da escrita do código continuar sendo procedural, alguns conceitos mudam de maneira bem radical; a estrutura e o modelo computacional são conceitualmente diferentes. De qualquer modo, duas características desejáveis no desenho de um *software* devem ser levadas em consideração:

Reutilização de código e Modularidade do sistema

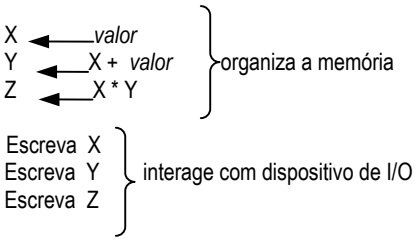
A primeira pode ser obtida através de reaproveitamento direto, especialização, estrutura de dados e bibliotecas de classes; a segunda com a definição de uma nova entidade: objeto.

Com essas duas características, consegue-se menores custos de desenvolvimento e de manutenção, respectivamente, tornando o preço final do produto bem mais competitivo.

Modelo Computacional *versus* **Modelo de Objetos**

Segundo PRICE (1997), existem vários modelos computacionais para dar suporte aos paradigmas de programação baseados no Modelo Computacional: concorrente, funcional, lógico e procedural. O mais conhecido entre eles é o da programação procedural, baseado em três componentes de *hardware*: p*rocessador, memória* e *dispositivo de I/O* . A programação procedural presume duas características básicas para desenho de *softwares*:

- Organização da memória;
- Interação com os dispositivos de I/O.



No Modelo de Objetos tudo é objeto, até os dispositivos de I/O, como por exemplo o objeto *TPrinter* do na linguagem Object Pascal. Não existe o conceito de "apenas um processador" e "uma memória" - num nível mais abstrato - e o mecanismo básico para a comunicação entre os objetos é a mensagem. A Figura 4 ilustra o mecanismo básico que rege esse modelo.

Figura 1: Interação entre objetos

O Objeto **A** solicita um determinado *serviço* ao Objeto **B** (através de uma mensagem) ; e se o objeto **B** puder atender à solicitação, o serviço será executado.

Por exemplo, em Delphi™; pode-se solicitar que o objeto FrmClientes (da classe *TForm*) seja exibido como uma janela modal, conforme o código da Figura 5.

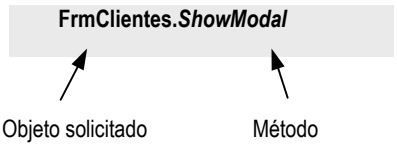


Figura 2: Solicitando serviço de um objeto

O objeto *FrmClientes* tem "dentro de si" todo o código necessário para responder às necessidades do programador. É como se ele fosse um "ente inteligente", necessitando apenas que solicitemos dele os serviços já pré-estabelecidos pela linguagem, tornando a programação mais modular e o código muito mais reutilizável.

A tecnologia de OOP baseia-se num termo: Objeto; e é em função desse conceito que toda a teoria desse tipo de programação se desenvolve. Objeto é uma unidade dinâmica, composta por um estado *interno privativo* (estrutura de dados) e um *comportamento* (conjunto de operações). Este conceito pode ser estendido para definir um "processador" com "memória" própria e independente de outros objetos (processadores). 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. Portanto, um objeto pode ser comparado a um pequeno universo, tendo no seu núcleo os dados internos e a porção que o envolve representa as rotinas que manipulam esses dados, de modo coerente a cada tipo de situação. O esquema da Figura 6 mostra o exemplo de um objeto.

Figura 3: Uma "visão" de um objeto

No exemplo da Figura 6, o objeto *Empregado*, tem no seu "núcleo" dados que poderiam representar valores das propriedades: Nome, Cargo, Salário e Setor. Em torno desse "núcleo" a expressão *Atualizar Vendas* é uma das ações que o objeto pode executar quando for solicitado esse serviço. A *mensagem* é a maneira como é feita a solicitação para que se possa modificar o "status" atual do objeto, e é composta de três partes:

1. Objeto receptor: uma *instância* da classe;
2. Membro: propriedade ou método do objeto
3. Parâmetros: valores especificados para esse *membro* quando necessários.

A *mensagem* enviada a um objeto pode ser tanto para que modifique um dado interno (*propriedade*) ou simplesmente para que execute alguma ação (*método*).

```
Objeto.Propriedade   valor  Empregado.Setor := "DEPVEN"

Objeto.Método[(parâmetros)]
Empregado.Pesquisar('12345-96')
```

Para os desenvolvedores que usam ferramentas RAD (*Rapid Application Development)* -Desenvolvimento Rápido de Aplicações-Visual Basic™ ou Delphi™- é bastante trivial o uso de instruções parecidas com as descritas acima. Tais ferramentas, além de proporcionar

ganhos expressivos de produtividade (com classes de objetos já prontas para uso), possuem o recurso da interface gráfica, o que melhora ainda mais a interatividade dos sistemas. Nas duas linhas de instrução em Delphi™ estamos modificando a propriedade *Setor* do *Empregado* para "DEPVEN" e solicitando o método *Pesquisar* para que seja pesquisado um empregado cuja matrícula é "12345-96".

Linguagens *versus* **Bancos de Dados**

Segundo MAURO (1997) e MATOSO (1997), três situações podem ser consideradas quando é levantada a questão: linguagem de programação e banco de dados.

- linguagem estruturada e banco de dados relacional;
- linguagem orientada a objetos e banco de dados relacional;
- linguagem orientada a objetos e banco de dados orientado a objetos.

No primeiro caso existe o problema do descasamento de tipos entre linguagem e banco de dados; é o caso de sistemas desenvolvidos em *COBOL* estruturado, acessando um banco de dados relacional. Esse problema acontece devido ao fato de o banco de dados agir sem o controle adequado do ambiente de programação.

No segundo caso -mais usual atualmente, com o uso de ferramentas *RAD* como os exemplos já citados Visual Basic™ e o Delphi™ - o problema do descasamento, entretanto, persiste. E, apesar da boa interação com o usuário através da interface gráfica, e como conseqüência disso forte apelo visual, cria-se um problema adicional: a abordagem lógica fica fortemente acoplada à essa interface, com o programador preocupado só com esse aspecto.

Essas duas ferramentas de desenvolvimento, em conjunto com o BDR, "indicado" como o mais adequado, Paradox™ para o Delphi™ e Access™ para o Visual Basic™, têm tido boa aceitação no mercado, com presença quase que obrigatória nos sistemas comerciais em que se necessita uma interação amigável com o usuário final. Entretanto, além do problema do descasamento de tipos, *impedance mismatch*, um outro problema sempre é deixado de lado: a questão do armazenamento dos dados.

O que se deseja é uma solução definitiva para o armazenamento num banco de dados que tenha um perfil ideal para isto. A solução é unir o útil ao agradável: implementar numa LOO e armazenar os dados num BDOO distribuído e atualizável instantaneamente em cada estação de um sistema em rede.

Uma linguagem orientada a objetos em conjunto com um banco de dados orientado a objetos é o mais indicado para o desenvolvimento de sistemas mais corretos, mais funcionais e com manutenções corretivas quase nulas. Para obter-se essa funcionalidade é fundamental que esse armazenamento seja feito em dispositivos condizentes com essa tecnologia. Os Sistemas Gerenciadores de Banco de Dados Orientados ao Objeto (SGBDOO), agora na atual geração, permitem que além de dados (*propriedades*) sejam também armazenadas as rotinas (*métodos*) das instâncias de uma classe.

A abordagem LOO com um BDOO agiliza em muito o processo de leitura e gravação dos membros de uma instância de uma classe e resolvendo o tradicional problema de inconsistência entre o armazenamento de dados e as rotinas. Com esse enfoque, tanto a questão do desenho do *software* quanto a do armazenamento dos dados poderão ser equacionados satisfatoriamente.

Considerações Finais

Como apresentado inicialmente neste artigo, o problema de estruturação e modularização de um sistema ou de uma rotina em particular, pode ser facilmente resolvida com as metodologias tradicionais apresentadas: *Design Top-Down*, *Design Bottom-Up* ou **O**bject-**O**riented **P**rogramming.

LOO com BDR não distribuído e não atualizável em tempo real, é a situação mais comum no contexto atual de desenvolvimento; mas apesar de ser esta a realidade atual, tal abordagem compromete a curto prazo a eficiência e confiabilidade do sistema, e a médio prazo sua manutenibilidade, aumentando o preço final do produto. Sistemas baseados em LOO e BDR com tecnologias de análise estruturada e/ou modular deverão, a médio prazo, ser substituídos pelos sistemas baseados em LOO (com interface gráfica) e BDOO para enfrentar os novos desafios da concorrência globalizada. Além disso, resolve em definitivo o problema de *impedance mismatch* no contexto da implementação. A segurança, a performance, a legibilidade do código e atualização em tempo real dos dados acessados pelo sistema, são fatores determinantes na escolha de um bom *software* que atenda às expectativas do cliente em potencial

Uma proposta de pesquisa é explorar melhor a tecnologia de OOP, empregando técnicas de análise OO (como por exemplo CRC (Colaboração e Responsabilidade de Classes) junto com uma metodologia OO - como por exemplo OMT, proposta por RUMBAUGH (1994)- , para produzir soluções de *software* em linguagem comercial e portátil, com armazenamento de dados em BDOO de modo a distribuir as informações em rede de computadores, como numa *Intranet*, garantindo segurança, performance e rapidez nas consultas.

A discussão pode levar ao emprego de uma ferramenta *RAD* com interface mais amigável (como por exemplo o Delphi™) em conjunto com um BDOO de última geração que permitisse tal integração via alguma API com uso de bibliotecas em C++

Referências

LEITE, Mário *Programação Orientada ao Objeto: Uma Abordagem Didática*. Maringá, 1988. Monografia (Especialização) pós-graduação *Lato-Sensu* em Análise de Sistemas. CESUMAR.

LEITE, Mário *Curso Básico de Visual Basic*. SYC, Maringá,1996

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

ODEL, James J.; MARTIN, James. *Análise e Projetos Orientados ao Objeto*. São Paulo: Makron Books, 1996.

PRICE, Tom.*Programa de especialização com opção para o Mestrado.* Porto Alegre*:* UFRGS, 1997.

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W. Premerlani et al. *Modelagem e Projetos Baseados em Objetos*. São Paulo: Campus, 1994.

SBBD'98. Anais do XIII Simpósio Brasileiro de Banco de Dados. Edição 1998.