Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica

Uma ferramenta de apoio à análise orientada a objeto

autor: Sérgio Roberto Sigrist

orientador: Profª Beatriz Mascia Daltrini

Dissertação apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas, como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Elétrica.

Fevereiro de 1995

Para

Sérgio, Nilza, Nimó, Wagner, Marcos, Pri, Mário, Tiquinho e Luiza.

Agradecimentos

Aos professores do Departamento de Engenharia de Computação e Automação Industrial da FEE UNICAMP, especialmente àqueles com quem tive mais contato através das disciplinas.

À Profª Drª Beatriz Mascia Daltrini pela atenção e modo cordial como conduziu a orientação do trabalho.

Ao José Celso Carriel de Abreu pela valiosa cooperação na parte prática.

Ao Prof. Adriano Júlio de Barros Vicente de Azevedo Filho, pelas sugestões e colaboração em todos os momentos.

Ao Engº Nermano Franco Ferreira e Prof. Dr. José Vicente Caixeta Filho pela dispensa parcial das atividades profissionais.

Aos amigos do Centro de Informática na Agricultura e da USP-Piracicaba, que direta ou indiretamente auxiliaram na elaboração do trabalho.

Sumário

Resumo

Esta tese apresenta uma visão geral da orientação a objetos e oferece um protótipo de software para auxílio ao processo da análise.

Na revisão são destacados a motivação, problemas atuais, tendências, conceitos básicos, CASE, métodos de análise/design; também são mencionados os aspectos relevantes de linguagens de programação, interface de usuário, sistema operacional, base de dados e *framework*.

Esse estudo possibilitou colher subsídios para desenvolver uma aplicação de base de dados que permite o exercício da análise através da descrição dos objetos e suas propriedades. As características centrais são a manipulação de objetos a partir de dados previamente armazenados, verificação de erro e de relacionamento inverso, geração de relatório e mecanismo de herança.

São descritas as principais decisões de concepção do software, com uma exposição sobre a funcionalidade do mesmo. O protótipo é comparado a outras ferramentas, sendo apontadas as limitações atuais e as sugestões para evolução.

Abstract

This thesis presents an overview of object-orientation and offers a prototype software to assist the analysis process.

The motivation, recent problems, trends, basic concepts, CASE, and analysis/design methods are high-lighted in a review of the area; the relevant aspects of programming languages, user interface, operating system, data base, and framework are also pointed out.

This study has provided the means to develop a data base application allowing analysis task through the description of objects and their properties. The central features are object manipulation from previously stored data, error and inverse relationship checking, report generation and inheritance mechanism.

The major design decisions are described and the system's usabilility is demonstrated. The prototype is compared with others tools, its limitations and suggestions for improvement are pointed out.

1 - Introdução

O objetivo deste trabalho é oferecer uma ampla visão da situação atual da orientação a objetos e apresentar um protótipo de software para apoio ao processo da análise orientada a objeto.

Atualmente a orientação a objetos é reconhecida como uma importante ferramenta metodológica para o entendimento, representação e desenvolvimento de sistemas. Os benefícios freqüentemente declarados são: reuso; maior proximidade com o mundo real do que outras metodologias; obtenção de modelos mais estáveis (objetos são mais estáveis do que processos ou procedimentos); transição mais suave entre as fases de análise, design e programação; e construção de sistemas flexíveis, onde a manutenção pode ser feita de modo rápido e eficiente.

Embora a orientação a objeto seja uma área considerada ainda nova, um levantamento realizado na rede Internet em setembro de 1994 mostrou a existência de 42 produtos CASE e 20 métodos de análise/design. Isso sem contar as diversas linguagens de programação, os sistemas para construção de interface de usuário, sistemas operacionais e banco de dados que estão disponíveis, para citar algumas áreas de interesse.

Dentre essas áreas de interesse, encontram-se as ferramentas automatizadas de apoio à análise de sistemas. As ferramentas são importantes pois além de acelerar o processo de desenvolvimento, permitem que o profissional trabalhe sem se preocupar com certos detalhes, omissões, erros e, portanto, pode se concentrar mais na solução do problema.

O uso de ferramentas sempre foi enfatizado por outras metodologias, porém na orientação a objeto as ferramentas assumem um papel decisivo. Isso porque, em geral, um modelo gera uma rede muito grande de dependência entre os objetos participantes. Atividades comuns como a inserção ou a exclusão de objetos alteram todo o contexto

do modelo, portanto deve-se ter meios de garantir a integridade do mesmo. Boas ferramentas podem oferecer essa garantia!

Movido por esse interesse, buscou-se referências na literatura que pudessem fornecer subsídios, que serão apontados a seguir, para a concepção de uma ferramenta de apoio à análise.

Uma abordagem comum encontrada na metodologia consiste em descrever inicialmente a dimensão estática do problema, ou seja, desenvolver um modelo em termos de objetos e suas propriedades (atributos e serviços), e os relacionamentos com outros objetos. A dimensão é estática no sentido de que não há *tempo* envolvido. Segundo RUMBAUGH et al (1991), essa atividade permite identificar e entender melhor os objetos que irão compor o modelo.

Um ponto central na metodologia é *herança*, segundo a qual uma classe herda propriedades de outra(s) classe(s). Nota-se que a herança, já incorporada nas linguagens de programação, também deve se expressar na análise.

A herança está diretamente associada à classificação que é, em geral, arbitrária e de difícil tratamento. Não existe uma estrutura de classe perfeita. Em seu livro, BOOCH (1991) dedica um capítulo inteiro somente para discutir a classificação. Pode-se supor, a grosso modo, que as propriedades definidas nas classes, principalmente nas de mais alto nível, dificilmente sejam compartilhadas sem modificações pelas subclasses. É mais provável que à medida em que se caminha em direção às classes especializadas, haja alteração nas propriedades herdadas, inclusão de novas propriedades e exclusão das não desejadas.

Outra fonte de informação que auxiliou na elaboração do protótipo foram os produtos CASE. Duas ferramentas foram examinadas em detalhes: METAEDIT (1993) e OOTHER (1994). Estas ferramentas combinam a notação gráfica com a textual. A METAEDIT, por exemplo, permite trabalhar segundo a notação de diversos métodos, entre eles OMT (RUMBAUGH et al, 1991), OOD (BOOCH, 1991) e OOA (COAD & YOURDON, 1990). Ao selecionar um método, por exemplo OOA, tem-se acesso a um editor gráfico que permite traçar o desenho de acordo com a notação deste método. Durante a elaboração do desenho, pode-se optar por uma série de relatórios.

Com base na revisão da literatura, pôde-se desenvolver uma ferramenta dedicada a alguns pontos de interesse. A ferramenta, que pode ser classificada como uma aplicação de base de dados, permite manipular objetos (criar, eliminar, modificar, imprimir, armazenar, recuperar), especificar atributos e serviços, verificar automaticamente erros na entrada de dados e obter relatórios na forma de texto estruturado. Possibilita também o exercício da análise sem estar associado a algum método. O relatório é um bom ponto de partida para ser usado por qualquer método já conhecido.

Foi implementado ainda, em nível experimental, um mecanismo para tratar a herança (simples ou múltipla). A forma como a herança ocorre é a seguinte: quando se escolhe o relacionamento "generalização / especialização" entre duas classes, os atributos e os serviços (e as respectivas especificações) da classe "geral" são copiados para a classe "especializada", onde pode-se alterar livremente cada especificação no sentido de atender as características particulares de cada classe especializada.

Procurou-se através da implementação de herança, que não existe na METAEDIT e OOTHER, obter flexibilidade no que tange ao reuso de especificações.

Algumas limitações devem ser apontadas. Atualmente o protótipo permite somente a descrição textual, encontra-se em uma versão monousuário, roda sob o DOS e gera arquivos de dados no formato DBF.

A contribuição deste trabalho pode ser vista, portanto, a partir de dois aspectos: apresentar uma extensa revisão bibliográfica; e oferecer um protótipo de ferramenta para apoio ao processo da análise, através da descrição da dimensão estática do problema. A revisão bibliográfica extensa deve-se ao fato de que atualmente é importante conhecer e se posicionar sobre alguns pontos que demandam pesquisa.

O restante do trabalho é apresentado a seguir.

- Capítulo 2 - Revisão de literatura apresenta uma visão da situação atual da orientação a objetos. Inicialmente são examinados o contexto, a motivação, e o modelo abstrato de objetos adotado pela *Object Management Group* (OMG), uma organização que reúne empresas de hardware e software para estabelecer padrões industriais. Em seguida são resumidos alguns produtos que representam domínios específicos tais como linguagem de programação, interface de usuário, sistema

operacional, base de dados, *framework* e CASE. Em seguida é focalizado o processo da análise segundo a proposta de diversos métodos. Por último são feitas as considerações finais.

- Capítulo 3 - Detalhes de projeto apresenta os detalhes da ferramenta proposta. São declarados os requisitos do software, é apresentado o modelo de objetos que serviu de base para a construção da ferramenta, é mostrada a arquitetura geral do software e são prestadas as informações técnicas sobre a implementação.

- Capítulo 4 - Apresentação de resultados expõe a funcionalidade do software através de exemplos com alguns objetos. Os resultados obtidos são discutidos e o sofware é comparado a outros sistemas de apoio à análise orientada a objetos. Por último são feitas as considerações finais.

- Capítulo 5 - Conclusão e perspectivas apresenta a conclusão geral do trabalho e perspectivas de continuidade.

Nota

O termo *design* é empregado aqui sem tradução. O *design* (que poderia ser traduzido para *desenho* ou *plano*) representa, para o autor deste trabalho, a fase de projeto do ciclo de vida de um software. É o sumário da idéia básica que está sendo usada para tratar um problema específico; são as 5 - 10 decisões mais importantes para o sucesso da implementação.

As expressões "orientação a objeto" e "orientado a objeto" serão referidas por OO.

2 - Revisão da literatura

Este capítulo é dividido em cinco seções. A Seção 2.1 destaca a motivação e algumas tendências da OO. A Seção 2.2 apresenta os termos e os conceitos adotados pelo *Object Management Group* (OMG). A Seção 2.3 mostra a situação atual da OO em termos de alguns produtos em áreas de interesse como linguagem de programação, interface de usuário, sistema operacional, base de dados, *framework* e CASE. A Seção 2.4 focaliza o processo da análise/design OO de diversos métodos encontrados na literatura. Na Seção 2.5 são feitas as considerações finais.

2.1 Considerações iniciais

A solução de problemas em computador tem sido fortemente influenciada pela visão procedimental. Nesta visão "representar a solução de um problema envolve escrever uma série de ações (procedimentos) que, se executadas seqüencialmente, levam à sua solução. Várias razões poderiam ser usadas para explicar o crescimento da classe de linguagens procedimentais; devemos apontar o papel histórico do uso do computador em aplicações numéricas e o uso do computador dentro do próprio domínio da ciência da computação. O desenvolvimento de tais linguagens tem sido feita por especialistas em computação, para uso de especialistas em computação, dentro de seu próprio domínio, onde questões de eficiência e desempenho são fundamentais" (BARANAUSKAS, 1993).

Diversos outros paradigmas também têm sido exercitados. "LISP[1] e derivados ilustram o suporte concreto a um outro paradigma, o funcional, em que as tarefas são expressas através de composição de funções. Na década de 80, um novo paradigma, o

[1] O autor refere-se à formulação original. Atualmente a linguagem inclui recursos poderosos associados à OO com a CLOS (Common Lisp Object System).

lógico, foi reconhecido a partir da disseminação de PROLOG e de outras linguagens baseadas em lógica. Finalmente, já chegando aos anos recentes diversas adições foram sugeridas, tais como orientação a objetos, orientação por acessos, base em conhecimento e orientação por regras" (TAKAHASHI, 1989).

O interesse na OO, entretanto, tem crescido rapidamente nos últimos anos e está influenciando a comunidade de informática e de usuários de sistemas.

O paradigma de objetos, assim chamado por representar o resultado do raciocínio como um conjunto de classes de características, apresenta-se como uma importante ferramenta metodológica para o entendimento, representação e desenvolvimento de sistemas complexos, além de suportar de maneira mais efetiva o *raciocínio extrativo*. Este raciocínio revela novas características a partir de características latentes ou potenciais, sendo comum na manutenção de sistemas, onde novos requisitos (anteriormente ocultos) são incorporados (RINE & BHARGAVA, 1992).

A questão que procura-se destacar aqui situa-se no contexto dos sistemas atuais que desafiam os desenvolvedores de software, e das dificuldades de se construir tais sistemas baseado no enfoque procedimental.

Booch considera que o desafio atual consiste no desenvolvimento de software de esforço *industrial*, ou seja, "aplicações que exibem um conjunto rico de comportamentos como, por exemplo, sistemas reativos que controlam algum processo físico onde tempo e espaço são recursos escassos; aplicações que mantêm a integridade de milhares de registros de informação enquanto permite *queries* e atualizações concorrentes; sistemas que controlam entidades do mundo real, tais como a rota num tráfego aéreo ou terrestre; sistemas que duram um longo intervalo de tempo e tornam muitos usuários dependentes do seu funcionamento; programas que imitam certos aspectos da inteligência humana" (BOOCH, 1991).

Booch destaca que a decomposição de sistemas complexos empregada na OO é antagônica à decomposição no design estruturado, ou decomposição algorítmica, onde cada módulo denota uma etapa principal num processo. A decomposição por algoritmos enfatiza o ordenamento de eventos, enquanto a decomposição por objetos destaca os agentes autônomos que enviam e reagem às mensagens recebidas. A

segunda alternativa é preferível porque fornece unidades menores através do reuso de mecanismos comuns, é mais flexível às alterações e reduz o risco na construção de sistemas complexos, pois esses são projetados para evoluir incrementalmente a partir de sistemas menores, onde já se tem alguma confiança (BOOCH, 1991).

A atividade de manutenção de sistemas é uma das que mais tempo e recursos consome. Deve-se descobrir os locais de manutenção e considerar a introdução de novas linhas de código, podendo implicar em novas necessidades de análise, reformulação do design e teste.

Para Jacobson, um sistema projetado com um método OO apresenta duas qualidades: o *entendimento do sistema* é mais fácil na medida em que a distância semântica entre o sistema e a realidade é menor; e as *modificações* no modelo tendem a ser locais, na medida em que resultam de itens individuais representados por um único objeto (JACOBSON et al, 1992).

Adele Goldberg destaca que conceitualmente os objetos encorajam uma arquitetura modular de sistemas, flexível para acomodar mudanças e novas funcionalidades: "Tenho ouvido os clientes descreverem os seus sistemas (não baseados em objetos) como uma comida italiana. É a teoria do spagueti x ravioli! O spagueti é uma massa confusa e entrançada, que fica de lado porque é difícil de desembaraçar. O ravioli tira-se um por vez. Pode-se reprojetar e recolocar" (GOLDBERG, 1992). Em outras palavras, com objetos, pode-se facilmente discernir partes independentes e compreender a funcionalidade proporcionada por essas partes; pode-se alterar a implementação de uma parte, mantendo a sua interface intacta, sem romper com o resto do sistema.

Para Coad & Yourdon, a motivação em objetos está na obtenção de uma base mais estável para examinar o espaço do problema e realizar as atividades de análise, design e implementação utilizando a mesma representação básica (COAD & YOURDON, 1990).

Como motivação final, deve-se destacar o papel central que a OO enfatiza. Abaixo são reproduzidas as palavras que representam o ponto de vista fundamental:

"O enfoque estruturado não é mais antigo que o enfoque de objeto. São alternativas que vieram de duas perspectivas. O enfoque estruturado (função/dados) abstrai a forma do computador trabalhar. A separação entre função e dados força as pessoas a pensar em termos de uma máquina abstrata. O enfoque de objeto permite considerar o fenômeno no domínio do problema independentemente de como o computador irá processá-lo. A expressão no enfoque estruturado é orientada à máquina; em objeto, é orientada ao domínio" (JACOBSON, 1993).

"A orientação a objeto não é tecnologia, é mais uma metodologia de resolver problemas do que um meio técnico de usar o computador. Esta diferença implica antes de tudo em uma mudança de posicionamento da pessoa envolvida em relação ao mundo da informática e aos seus instrumentos. Tem que passar de uma cultura centrada na tecnologia e de servir à máquina, para uma cultura centrada em soluções, usuários, abstrações. A implementação passa a ser um meio e não um fim" (LEE, 1993).

Apesar dessas considerações, existem alguns estudos que indicam que para a OO se afirmar, deverá superar algumas barreiras.

Diversos mecanismos de partição de sistemas são propostos mas não existe ainda um consenso sobre qual tipo de agrupamento de objetos oferece uma visão de maior nível do que a visão proporcionada por uma classe de objetos (SIGRIST & DALTRINI, 1994).

MONARCHI & PUHR (1992) sugerem o desenvolvimento de modelos estimativos para medir a qualidade da análise/design OO. Eles acrescentam que os modelos atuais são prescritivos e que existe a necessidade de se medir a dimensão técnica, tal como a coesão e acoplamento entre objetos, e a dimensão semântica, isto é, a proximidade que o design se encontra do modelo mental que os analistas, usuários e desenvolvedores fazem da situação.

FICHMAN & KEMERER (1992 a), citados por KOZACZYNSKI & KUNTZMANN-COMBELLES (1993), compararam a OO a três outras tecnologias: base de dados relacional, linguagens de quarta geração e desenvolvimento de sistemas estruturados. Após estudarem semelhanças e diferenças, concluiram que a tecnologia de objetos vai ser considerada "experimental" por um longo tempo. Isso implica que,

num futuro próximo, os desenvolvedores tentarão evitar o seu uso em projetos grandes e críticos.

Para KOZACZYNSKI & KUNTZMANN-COMBELLES (1993) o paradigma será dominante se superar o legado do passado. Três problemas são destacados: existem sistemas antigos construídos sob a forma estruturada que não se misturam com objetos; as ferramentas atuais não suportam a análise e a construção OO; e pessoas cuja experiência e especialização estão em outras formas de pensar.

2.2 Modelo de objetos

Diversos autores têm se posicionado sobre a importância de existir um vocabulário comum para discutir os conceitos e a terminologia da OO.

Para BROWN (1991) há problemas decorrentes do fato de que o termo *orientado a objeto* tem sido empregado de diferentes formas em diferentes domínios, tais como linguagens de programação, interface de usuário, métodos de design, sistemas operacionais, arquitetura de hardware e banco de dados.

Com o objetivo de identificar os termos centrais presentes em diferentes domínios, SNYDER (1993) desenvolveu um modelo abstrato de objetos que atualmente é empregado pelo *Object Management Group.*

Snyder sugere inicialmente uma definição genérica: "objeto é qualquer entidade que desempenha um papel visível ao proporcionar serviços para clientes. A natureza exata de clientes e serviços depende da aplicação em particular. Um cliente pode ser uma pessoa ou programa, e o serviço pode ser qualquer atividade executada em benefício de um cliente".

Usa-se o termo *classe* como a representação de atributos e serviços comuns, e *instância* de classe como uma (ou mais) ocorrência(s) de uma classe. Em geral, *objeto* refere-se tanto a uma classe quanto a uma instância de classe.

Outros conceitos freqüentemente associados a objeto são:

- Objetos ativos: objetos ativos são processos concorrentes com atividades próprias (ou controle de *thread*), isto é, podem iniciar uma computação espontaneamente sem que os seus serviços sejam solicitados por um cliente. *Thread* é unidade de escalonamento, é um fluxo de execução que implementa um programa.

- Composição de objetos: refere-se a um agrupamento de objetos que podem ser manipulados como um todo.

- Notificação de evento: um cliente pode ser notificado quando ocorreu um evento ou uma condição associados a um objeto (por exemplo: uma mudança de estado).

- Relacionamento: um relacionamento é uma informação associada a vários objetos (por exemplo: o relacionamento *empregado* está associado a *empresa* e *pessoa*).

Snyder sugere também empregar o termo pedido (request) ao invés de mensagem (message), pois mensagem está associada ao modelo clássico, onde ela é transmitida e processada numa única locação. Num modelo mais geral, um pedido pode conter nenhum, um ou vários parâmetros, sendo que cada parâmetro pode identificar um objeto. Outros termos correlatos para *pedido* são: invocação de método, invocação de função-membro ou invocação de função genérica.

A seguir, são apresentados os conceitos centrais analisados por Snyder.

Os objetos expressam uma abstração. Num modelo de objetos, os objetos são abstrações com significado claro para os clientes e ocorrem em diferentes níveis. Por exemplo, um *arquivo* é uma abstração de armazenamento em disco de maior nível do que os dispositivos de armazenamento. Os arquivos, implicitamente, implementam um nível maior de abstração, tal como um programa-fonte ou um documento. Este alto nível de abstração é que é manipulada pelo cliente. Os clientes, portanto, não são responsáveis pela interpretação de dados. Num modelo diferente de objetos, a interpretação de dados é somente implícita, sendo proporcionada por programas que lêem ou escrevem dados. Diferentes programas podem resultar em diferentes interpretações de dados. Neste modelo, os clientes precisam saber como os dados devem ser interpretados. Um arquivo pode ser visto como um programa-fonte em C pelo compilador C ou uma longa seqüência de caracteres por um programa utilitário de localização de *string*.

Os objetos proporcionam serviços. As abstrações contidas nos objetos são caracterizadas por um conjunto de serviços que podem ser solicitados pelos clientes. Os serviços podem acessar e modificar os dados. O dado que pode ser modificado por um serviço é chamado estado. Os serviços são descritos independentemente da forma de dados ou dos algoritmos usados para implementá-los.

Os clientes distribuem pedidos. Ao invés de manipular diretamente os dados, os clientes enviam pedidos aos serviços associados aos objetos. Ao cliente não interessam os detalhes de como os serviços são executados. Os clientes não precisam se lembrar do código para executar um serviço. O único tipo de erro é pedir um serviço não disponível. Desta forma, o cliente fica isolado dos detalhes de implementação: muda-se a forma de implementar o objeto sem modificar o cliente.

Os objetos são encapsulados. Os clientes podem ter acesso aos objetos somente através de um pedido de serviço, para prevenir o acesso direto aos dados e satisfazer certas restrições de integridade. Outros termos correlatos para encapsulamento são: proteção, ocultamento da informação e controle de acesso.

Os pedidos podem identificar as operações. Um pedido indica qual é o serviço a ser executado. Isso ocorre através da identificação de uma operação. Operação é uma entidade que simboliza um serviço, portanto trata-se mais de uma definição do que uma característica. No C++, uma operação é identificada quando especifica-se uma função numa classe. Se duas classes apresentarem o mesmo nome de função e esta função não for uma função-membro virtual (não herdou propriedades de uma classe ancestral comum a ambas as classes), as funções são consideradas operações diferentes. Na CLOS, uma operação é identificada por um objeto particular denominado *função genérica.*

Os pedidos podem identificar os objetos. Um pedido pode apresentar parâmetros e um serviço pode gerar um ou mais resultados. Um parâmetro ou um resultado podem identificar um objeto. Na maioria dos sistemas, cada pedido possui um parâmetro distinto para identificar o objeto que vai executar o pedido. Um objeto pode ser identificado diretamente e de maneira segura. A identificação é direta no sentido de que um pedido chama um objeto pelo seu nome, não havendo necessidade de descrevê-lo. A identificação é segura no sentido de que a repetição de uma chamada

de objeto reporta-se ao mesmo objeto. Assim, os clientes podem depender de sua habilidade para se referir aos objetos e não se preocupar em acessar objetos errados. No sistema Dynamic Windows, por exemplo, quando o usuário lista um diretório, o sistema associa cada arquivo (objeto) a um nome na tela. Ao selecionar um arquivo, o sistema apresenta as ações que podem ser executadas sobre este arquivo, mesmo que o nome do arquivo tenha sido alterado. Como exemplo contrastante, tem-se a interface padrão do Unix. Nesta interface, um nome é mapeado ao arquivo numa estrutura hierárquica de diretório, porém este mapeamento é frágil. Não se tem certeza de que um arquivo já conhecido ainda se encontra disponível, pois o nome e a estrutura de diretório podem ser alterados durante uma sessão de trabalho por qualquer usuário em qualquer instante.

Novos objetos podem ser criados. Os clientes podem pedir a criação de novos objetos distintos dos já existentes, permitindo a interação de vários clientes com objetos cujos comportamentos variam no tempo. Esta habilidade assegura que os clientes não compartilhem o mesmo objeto inadvertidamente. Por exemplo, um cliente cria um objeto *dicionário* vazio; este objeto é diferente de outros já existentes e vazios; se um cliente muda o estado do dicionário entrando com um nome e um valor associado, outros objetos *dicionário* não serão afetados.

As operações podem ser genéricas. Um serviço pode ter diferentes códigos em diferentes objetos, podendo produzir diferentes comportamentos. Quando um cliente pede um serviço, uma implementação apropriada é selecionada. Não há limite para as implementações de um serviço. Uma operação com múltiplas implementações é chamada operação genérica. O termo *genérica* destaca o fato de que o serviço é comum a vários objetos. A seleção de código para executar um serviço é chamada ligação. Quando um objeto é identificado no ato do pedido, isto é, durante a execução, diz-se que a ligação é dinâmica. A seleção conhecida antes da execução (compilação ou linkagem) é chamada ligação estática. Outros termos correlatos para operações genéricas são: função-membro virtual (C++) e seletor de mensagem (Smalltalk). A capacidade de suportar operações genéricas é também chamada polimorfismo e função de sobrecarga.

Os objetos podem ser classificados em termos de seus serviços. Os serviços associados a um objeto podem ser descritos como a interface do objeto. Uma interface descreve como os clientes podem usar o objeto. Em alguns sistemas, os usuários

determinam visualmente o conjunto de operações que podem ser solicitadas; as que não podem, ficam com tonalidade mais clara na tela. O termo protocolo é comum em programação OO, significando um conjunto de mensagens que podem ser enviadas a um objeto. Em diversas linguagens OO, uma interface é descrita por uma classe abstrata ou classe virtual (são classes que omitem o código de programa e, portanto, não podem ser instanciadas).

Os objetos podem ter uma implementação comum. A implementação de serviços associados a um objeto especifica o formato dos dados e o código para executar os serviços. Os objetos com implementação comum têm formato de dados idêntico e compartilham o código executável, mas possuem a sua cópia própria de dados.

Os objetos podem compartilhar implementações parciais. O mecanismo de herança suporta a construção incremental de um objeto, estendendo ou refinando as implementações de outros objetos. A nova implementação pode ampliar com novos dados ou operações, ou apenas alterar esses elementos a partir de uma implementação já existente.

O trabalho de Snyder é importante pois existem situações onde termos distintos estão às vezes associados ao mesmo conceito, como poderá ser observado ao longo das próximas seções deste capítulo.

2.3 Resumo de produtos

2.3.1 Linguagens de programação

A OO tem origem histórica em duas áreas: inteligência artificial (IA) e linguagens de programação. Em IA, os objetos aproximam-se de *frame* proposto por Marvin Minsky em 1975, no contexto da representação do conhecimento, para representar unidades maiores do que constantes, termos e expressões oferecidos pela lógica. Em frames e objetos, descrição e comportamento estão ligadas entre si. Frames

porém vão além dos objetos, pois presumem suportar raciocínio, enquanto objetos são freqüentemente usados para facilitar a compreensão das operações do sistema (DE CHAMPEAUX et al, 1993).

Por outro lado, a OO surgiu da necessidade de descrever e simular fenômenos como, por exemplo, fluxo de tráfego. Em 1961, Kristen Nygaard apresentou as idéias de uma linguagem que pudesse servir a esses dois propósitos: descrição e simulação. Essa linguagem foi denominada Simula I. A primeira aplicação foi na simulação de circuitos lógicos, porém os usuários descobriram rapidamente que ela proporcionava outros recursos poderosos, tais como prototipação e desenvolvimento de aplicações e, portanto, poderia servir como uma linguagem de propósito geral (MARTIN, 1993).

Simula foi a responsável pelos conceitos de classe e herança. A abstração de dados passou a ser conhecida nos anos 70 a partir da linguagem Módula-2. Mais antiga ainda, LISP introduziu o conceito de ligação dinâmica. Dois grandes grupos surgiram a partir da última década de evolução: um representado pelas linguagens puras, tais como Eiffel e Smalltalk, e outro pelas linguagens híbridas (extensão de linguagem já existente), tais como C++ e CLOS (WINBLAD et al, 1993).

Algumas características de C++, Smalltalk, Eiffel e CLOS são citadas a seguir.

- C++

C++ (STROUSTRUP, 1991; BERRY, 1988) é uma linguagem construída em torno da linguagem C, compartilhando as mesmas características básicas de C tais como a estrutura de programa, o uso de funções com argumentos, operadores, declarações de controle e tipos de dados. C++ porém é mais expressiva pois além de suportar a programação procedimental e modular, que são os estilos essenciais em C, também suporta a programação OO, facultando ao desenvolvedor trabalhar em ambos os paradigmas.

As características principais em C++ são tipagem estática de dados, suporte a herança múltipla e o acesso a dados restrito a um conjunto específico de funções.

Uma classe em C++ pode ser declarada como pública, privada ou protegida. A parte pública fica totalmente disponível para qualquer função do programa; a parte

privada é restrita às funções pertencentes à classe; e a parte protegida, além de servir às funções da classe, fica disponível também para as subclasses da classe. Através dessas definições, um programa pode acessar uma classe somente através das funções disponíveis na parte pública, não tendo contato com os detalhes de implementação da classe, que ficam parcialmente ocultos. Um benefício resultante deste aspecto está na forma em que as classes são projetadas. Classes de propósito geral, que podem ser usadas em diferentes sistemas, são fáceis de escrever e podem fazer parte de uma biblioteca de rotinas, o que é bastante enfatizado no projeto estruturado.

O conjunto de serviços declarados numa classe é formado por funções denominadas *membro* e *amiga*. Objetos de uma classe são criados e iniciados por funções-membro chamadas *construtores*; objetos podem ser eliminados por funções-membro denominadas *destrutores*. Ambas as funções asseguram a manipulação apropriada do ciclo de vida das instâncias durante uma execução.

C++ oferece também um mecanismo para permitir que uma função não pertencente a uma classe possa acessar a parte privada desta classe. Declarando-se uma função como *amiga*, esta função pode obter os mesmos privilégios que um membro da classe. O enfoque de função *amiga* serve para estabelecer uma ligação entre duas classes independentes. Por exemplo, pode-se declarar a função *time_date*() como amiga das classes *time* e *date*. Esta função "super amiga" formata e exibe a hora e a data. Embora as classes operem independentemente, com os construtores e estratégias de armazenamento interno próprias, a função *amiga* atua temporariamente como uma ponte entre as classes para obter um processamento específico.

C++ é uma linguagem poderosa e flexível, cujo conceito chave é *classe*. Através de uma classe, C++ proporciona modularidade, ocultamento de dados, conversão de tipos, gerência de memória e operadores de sobrecarga (são funções de mesmo nome mas implementadas com código diferente), além de preservar a habilidade de C de lidar eficientemente com os objetos do hardware (bits, bytes, palavras, endereços etc.). Devido a essas razões, C++ tem atraido a atenção de um grande número de programadores.

- SMALLTALK

Smalltalk (GOLDBERG & ROBSON, 1983) é a mais pura de todas as linguagens OO. É totalmente baseada num vocabulário restrito a cinco palavras: objeto, mensagem, classe, instância e método. Neste ambiente todo objeto é instância de uma classe e toda classe, à exceção da classe de mais alto nível denominada *Object*, é uma subclasse de outra classe.

Smalltalk não é apenas uma linguagem mas um ambiente gráfico e interativo formado por vários componentes. Inclui objetos que proporcionam as funções geralmente atribuídas ao sistema operacional, tais como gerência de armazenamento e de arquivos, manipulação do display, edição de texto e imagens, dispositivos de apontamento e teclado, escalonamento de processador, e mecanismos de sincronização e exclusão mútua.

A maior contribuição da Smalltalk ao desenvolvimento de software é a gerência de complexidade, onde o mérito fica visível no design e implementação de grandes aplicações ou em modificações de sistemas já existentes.

Não há declaração nem verificação de tipos em Smalltalk; logo, pode-se tentar enviar uma mensagem para qualquer objeto e, na execução, a linguagem decide se o objeto pode entender e responder. Ainda que o sistema como um todo não esteja livre de erros (um erro comum é solicitar um serviço inexistente de um objeto), pode-se executar partes do sistema e, portanto, verificar a funcionalidade desta parte. Devido a esta capacidade, a Smalltalk é freqüentemente associada à prototipação de sistemas.

Smalltalk foi a primeira a usar o *framework* MVC (Model-View-Controller) para a construção de interfaces gráficas. O MVC é formado por três subclasses diretas de *Object: Model*, são os objetos da aplicação; *View*, é a exibição para o usuário; e *Controller*, para coordenar as entradas do usuário. Essas características encontram-se presentes em diversos sistemas de janela atuais.

- EIFFEL

Eiffel (MEYER 1990;1988) é uma linguagem projetada para apoiar a construção de software de alta qualidade, onde *qualidade* significa principalmente corretude e reusabilidade.

Eiffel é uma linguagem pura e pode ser vista também como um método de construção de software, pois é ao mesmo tempo uma linguagem de design e de programação. Segundo o autor, o design de um software não pode ser visto como uma atividade totalmente separada da implementação.

As principais características da Eiffel são classes, herança múltipla, polimorfismo, tipagem estática e ligação dinâmica, manipulação de exceção e um mecanismo denominado *deferred class* para análise e design. *Deferred class* é uma extensão do mecanismo de herança, é uma forma de descrever um conjunto de possíveis implementações para uma classe, ao invés de descrever uma única implementação.

O ambiente de desenvolvimento é formado por um editor que entende a sintaxe da linguagem, ferramentas gerenciadoras de configuração, uma biblioteca de classes e um *browser* gráfico de classes. *Browser* é um mecanismo que possibilita encontrar classes na biblioteca.

Uma das opções principais é a geração de um pacote *stand-alone* em C, que pode ser portado para qualquer ambiente que suporte o C. Isso cria algumas facilidades. Por exemplo, pode-se usar a Eiffel para desenhar e implementar o software, e entregar o produto ao cliente em C. Eiffel não necessita estar disponível no ambiente destino. A biblioteca de classes inclui estruturas de dados e algoritmos habituais da programação, permitindo aos programadores pensar e escrever em termos de listas, árvores, pilhas e tabelas ao invés de arrays, ponteiros e flags.

- CLOS

A Common Lisp Object System, CLOS (STEELE, 1990), é o padrão de linguagem de programação OO para a Common Lisp, que tem sido considerada uma boa plataforma para a prototipação de sistemas em inteligência artificial e outras áreas.

Aos tipos de dados manipulados pela Common Lisp tais como números, caracteres, símbolos, listas, arrays, tabelas hash, streams, funções etc., foram introduzidas novas categorias, cujos objetos fundamentais são classes, instâncias, funções genéricas e métodos. O objeto *classe* determina a estrutura e o comportamento das suas instâncias. Todo objeto Common Lisp é instância de uma classe. Um mapeamento do espaço de classes da CLOS ao espaço de tipos da Common Lisp é definido, de tal forma que diversos elementos da Common Lisp padrão encontram uma classe correspondente na CLOS, de mesmo nome e tipo.

Uma função genérica em CLOS está mais próxima do conceito de mensagem, para permitir a programação de eventos simultâneos. Um objeto *função genérica* contém um conjunto de métodos da aplicação, uma lista formal de parâmetros denominada *lista lambda* e uma combinação de quatro espécies de métodos: *primary*, *after*, *before* e *around*. Os métodos definem o comportamento específico de classe, em outras palavras, um método apresenta uma *lista lambda* cujos parâmetros podem ser "especializados" para uma classe particular. Os métodos são diferentes de funções, pois são encapsulados nas funções genéricas, não são invocados diretamente sobre os argumentos. Funções genéricas são funções que contêm, entre outras informações, um conjunto de métodos. Quando invocada, uma função genérica executa um subconjunto de seus métodos. O subconjunto depende das classes ou identidades dos argumentos aos quais é aplicado.

2.3.2 Interface de usuário

Um exemplo de ferramenta para construção de interface gráfica de usuário é InterViews (LINTON et al, 1989), um kit composto por uma biblioteca de classes C++. Três categorias de classes fundamentais, denominadas *interactor*, *graphic* e *text*,

estão disponíveis na biblioteca e, a partir dessas classes, são derivadas as subclasses de objetos primitivos tais como: botões e menus (objetos interativos); círculos e polígonos (objetos gráficos); palavras e espaçamentos (objetos textos).

Os objetos primitivos utilizam um protocolo comum que permite um tratamento uniforme para a interface. Por exemplo, o protocolo para objetos interativos inclui serviços para tornar visíveis os objetos que ficaram obscurecidos; ou notificar um objeto que a tela que ele ocupa mudou de tamanho (neste caso, o objeto toma então uma ação apropriada).

Pelo protocolo, os objetos primitivos são agrupados de diferentes maneiras, até formar a interface completa. Um agrupamento pode organizar "lado a lado" as janelas na tela, enquanto outro pode organizá-las "em cascata". Um conjunto rico de primitivas e a composição dessas primitivas proporcionam uma maneira sofisticada de especificar diferentes estilos de interface.

A interface criada com auxílio dos objetos do InterViews é *linkada* junto com o código da aplicação. O ambiente gráfico do sistema operacional é totalmente abstraído da aplicação; a interface do usuário é definida em termos dos objetos do InterViews, que se comunicam com o ambiente gráfico e o próprio sistema operacional.

O enfoque de objetos também é empregado na *User-Interface Design Environment* (FOLEY et al, 1989), um sistema para auxiliar o design e a implementação de interface de usuário. O design é representado internamente como instâncias de sete ART *frames*: objeto, ação, parâmetro, pré-condição, pós-condição, atributo e tipo de dados; e externamente por um texto estruturado denominado IDL (Interface Design Language).

2.3.3 Sistema operacional

Sistemas operacionais OO são sistemas que proporcionam recursos através de objetos. Caracterizam-se pela distribuição, isto é, os objetos podem ser passados livremente por entre as máquinas. São sistemas tipicamente baseados em "capacidades" uma vez que os objetos, e conseqüentemente os recursos do sistema,

somente podem ser acessados se as suas capacidades estiverem disponíveis aos programas.

Um ótimo exemplo é o NeXTstep, principal produto da NeXT, um sistema totalmente baseado em objetos. Os objetos fazem parte do próprio sistema operacional. Por exemplo, para adicionar a funcionalidade de recebimento e envio de fax dentro de um aplicativo, já há um objeto FAX, o que se deve fazer é estabelecer o diálogo do aplicativo com o objeto. Existe atualmente um mercado para objetos onde pessoas desenvolvem e vendem objetos que solucionam problemas interessantes como o exemplo do fax.

O NeXTstep é formado por vários níveis de software: um gerador de interfaces gráficas; um gerador de projetos (permite a concepção completa de uma aplicação); e um kit de base de dados. Essa tecnologia, segundo BEARD (1993), "está apontando o futuro dos sistemas operacionais, a prova disso está no fato de as três maiores empresas de informática - IBM, Apple e Microsoft - estarem a desenvolver os seus próprios sistemas operativos orientados para o objeto".

2.3.4 Base de dados

Bases de dados orientadas a objeto são bases de dados que suportam objetos e classes, permitindo todos os benefícios da OO, além de possuir uma forte equivalência com os programas OO.

Um exemplo é o GemStone (ROWAN, 1992), o mais antigo sistema comercial atualmente disponível. É particularmente indicado para multi-plataformas cliente/servidor complexas. Suporta acesso concorrente de linguagens externas como Smalltalk e C++, ou a partir de qualquer aplicação que aceite chamadas externas de funções C.

A definição interna de dados e a linguagem de manipulação de dados do GemStone são baseadas no Smalltalk, logo as classes do Smalltalk e do GemStone são similares em estrutura, permitindo um mapeamento natural dos objetos do programa em objetos de base de dados.

No topo do GemStone reside um ambiente para desenvolvimento de aplicações denominado GeODE (GemStone Object Development Environment), composto por quatro componentes principais: editor de formulários (ferramenta gráfica para criação de interface de usuário); construtor visual de programas (representa na tela os fluxos de dados e a seqüência de controle do programa); construtor de aplicações (armazena os formulários e os programas como uma aplicação única); e uma coleção de ferramentas tais como browser, debugger, profile, inspector etc..

2.3.5 Framework

Framework tem merecido destaque na literatura. Trata-se de uma categoria de ferramenta dirigida à redução do esforço de codificação. Embora não seja um conceito novo, somente agora passou a merecer atenção como ferramenta de desenvolvimento da plataforma PC. Em 1985, o MacApp[2] sistematizou as idéias de framework de aplicação como uma "coleção de classes estendidas que cooperam para suportar uma arquitetura completa de aplicação, oferecendo um suporte de desenvolvimento melhor do que as bibliotecas de classe" (VALDÉS, 1992a; 1992b).

É importante esclarecer a diferença entre framework e biblioteca de classes. Framework é um conjunto de classes especialmente projetadas para atender aplicações específicas, forçando um enfoque de arquitetura de sistema. Já em uma biblioteca de classes, as classes não possuem nenhuma ligação entre si, são classes de funções gerais como, por exemplo, funções de interface ou de verificação do hardware conectado à aplicação, pode-se escolher uma ou outra classe e incorporar na aplicação. Um exemplo de biblioteca é a OBJECT TOOLKIT (1991).

O MacApp foi o primeiro ambiente comercial para desenvolvimento, porém foi considerado uma versão primitiva. Os programadores tinham garantida a interface com o MacIntosh mas, para criar uma interface ou aplicação, era necessário aprender tanto o MacApp quanto uma linguagem de programação como o Pascal, que também estava disponível na época. Depois de definida a visão da interface do usuário, era possível

[2] Framework de aplicação da Apple projetada para a criação de interfaces e desenvolvimento de aplicações no Macintosh.

abandonar o ambiente da MacApp e concentrar-se no Pascal para elaborar os eventos da aplicação (WINBLAD et al,1993).

Framework é tema de pesquisa, onde três áreas são destacadas: design, uso e descrição. O *design* procura descobrir e projetar as características que um framework deve apresentar. O *uso* relaciona-se à atividade de juntar ou configurar objetos para atender os requisitos específicos de uma aplicação. A *descrição* dedica-se ao desenvolvimento de uma notação especial para framework, diferente da usada na análise/design, pois framework significa algo mais do que as classes nela contida (WIRFS-BROCK & JOHNSON, 1990).

Esta última área, descrição, é também referida por O'BRIEN (1993). Por contar com toda sorte de suposições e refletir um grande número de compensações e ajustes, framework não pode ser documentado com sucesso com diagrama hierárquico de classes e referências a métodos e variáveis de instância. Ainda permanece a tarefa crucial de explicar a filosofia de design embutida. No entanto, O'Brien considera que framework se tornará indispensável na maioria das aplicações.

2.3.6 CASE

As ferramentas de auxílio à engenharia de software têm sido bem difundidas. CASE (Computer Aided Software Engineering) são programas que auxiliam os profissionais nos processos de análise/design, sendo que algumas ferramentas vão ainda mais longe, pois geram automaticamente o software a partir das especificações (FISHER, 1990).

CASE pode ser classificado em dois grandes grupos: CASE específico, para atividade bem determinada (o termo original), e I-CASE, que dá suporte a mais fases do ciclo de vida do software, procurando integrar os diversos CASEs (MARTIN, 1993).

Um interessante levantamento sobre as características das ferramentas CASE que mais influenciaram o critério de seleção no mercado brasileiro foi realizado por LUCCA (1992). Ele aferiu, pela ordem, que as características mais importantes são:

compatibilidade com a metodologia usada na empresa; facilidades de uso; dicionário de dados; acompanhamento de todas as fases do ciclo de vida; documentação; verificação de integridade; integração com outros produtos; segurança; recursos gráficos; continuidade com o mesmo fornecedor; portabilidade; prototipação; multiusuário; adaptação para uma metodologia particular; "help on-line"; suporte a várias versões do sistema; etc..

Deve-se ressaltar que o trabalho de Lucca, cuja síntese é de 1992, colheu dados sobre ferramentas de suporte à metodologia estruturada.

Já o paradigma de objetos, segundo KORSON & MCGREGOR (1990), apóia-se mais pesadamente no uso de ferramentas e ambientes do que as técnicas procedimentais, pois o processo de design gera um grande número de componentes de software interconectados numa rede de relacionamentos.

COAD & YOURDON (1990) sugerem que um suporte de CASE para a análise OO deve ter os seguintes elementos:

- Uma notação para as camadas da análise OO.
- Uma seleção do tipo ON/OFF para cada camada.
- Um modelo de verificação para produzir antecipadamente mensagens de erro ou advertências como, por exemplo, verificar se os objetos, atributos e serviços têm nomes únicos, se os atributos cancelados em um objeto aparecem como atributos regulares nos seus descendentes, se os atributos e os serviços dos objetos estão todos especificados etc..

BOOCH (1991) aponta que o design OO, por destacar as abstrações e mecanismos-chave do problema, necessita de ferramentas que possam focalizar em semântica mais rica. O autor identifica seis tipos mais importantes de ferramenta:

- Um sistema gráfico para suportar a notação.
- Browser para examinar a estrutura de classe ou navegar pelas superclasses e subclasses.
- Compilador incremental, diferente da compilação em batch, para compilar declarações isoladas.
- Debugger conhecedor da semântica das classes e objetos.
- Ferramentas para gerência de configuração e controle de versões.
- Biblioteca de classe que permita localizar as classes segundo diversos critérios, e acrescentar novas classes à medida que elas forem sendo desenvolvidas.

WINBLAD et al (1993) citam os *browsers*, os *inspectors*, os *profiles* e os depuradores como as principais ferramentas. *Inspector* é ferramenta para programação. Serve para examinar o valor das instâncias de um objeto. *Profile* é ferramenta para identificar as partes do sistema que requerem refinamento de desempenho. Por exemplo, o *profile* da linguagem Actor lista o número de vezes que cada método foi executado durante a sessão de um programa. Os métodos com maior número de chamadas são identificados como candidatos à otimização do desempenho.

JACOBSON (1993) discute ferramentas CASE num contexto diferente. Para ele, a tecnologia de objetos deve ser oferecida na forma de um processo industrial, o que requer elevar os métodos OO a um nível em que hoje a maioria ainda não se encontra. Os métodos devem estar ligados a técnicas industriais de gerência de projetos, para produzir produtos comercialmente viáveis, e serem adaptáveis a novas empresas, novas aplicações e novas tecnologias. Necessitam-se ainda de técnicas para "reengenheirar" os sistemas antigos, construídos com outros métodos, e que continuam a serem desenvolvidos.

Para suportar um processo efetivo de desenvolvimento, Jacobson sustenta que o modo de expressão, isto é, as linguagens de programação, de especificação e de modelagem, devem melhorar em direção às seguintes tendências:

- O foco do processo de desenvolvimento deve voltar-se mais para a análise que para a implementação. As linguagens atuais de análise/design OO têm notação gráfica, são formalizadas, mas a semântica é incompleta. Todos os participantes dos projetos (usuários, líderes de projeto, gerentes e clientes), não aculturados na interpretação dessas linguagens, devem compreendê-las. O desafio é "penetrar no que a notação gráfica e a semântica estão tentando dizer, não apenas o que elas dizem" (JACOBSON, 1993).
- Deve-se colocar mais ênfase na aplicação e menos na técnica. Os desenvolvedores devem ser peritos na aplicação, não peritos em C++ ou Unix, pois irão trabalhar com objetos gráficos apresentados em diferentes dimensões e não simplesmente com textos. As linguagens atuais devem ser manuseadas como linguagem de máquina, que é invisível ao desenvolvedor.
- Usar a OO para a modelagem comercial, para que os objetos possam ser implementados como recursos humanos (*peopleware*), combinados com o software e o hardware.

Jacobson conclui que, após inventarem um processo de desenvolvimento OO, pode-se especificar as ferramentas CASE e um manual para os desenvolvedores.

A produção de CASE no mercado comercial, no entanto, cresce continuamente e diversas ferramentas estão disponíveis.

Um panorama de CASE é mostrado a seguir (Tabela 2.1). Estas ferramentas são classificadas em 4 categorias: (a) somente gráfico; (b) CASE com alguma geração de código; (c) meta CASE (CASE que constrói CASE); e (d) CASE de execução total.

A Tabela 2.1 mostra uma relação parcial de CASE. Outras ferramentas podem ser encontradas na rede Internet, algumas na forma de produtos *shareware*.

Nome (categoria)	Descrição	Ambiente
BridgePoint (d)	Notação de Shlaer/Mellor; gera código a partir de pseudo-código; ferramenta de simulação	Unix
EasyCASE (a)	Parte da notação de Shlaer/Mellor e outras notações não-OO	Dos, Windows
EiffelCase (b)	Gera classes Eiffel	Unix, Windows NT
Envision (c)	Independente de metodologia, definida pelo usuário.	Windows, NT, OS/2, (Chicago), Network Servers
Excelerator II (c)	Notação de Martin/Odell, Rumbaugh, Jacobson e Wirfs-Brock. Pode customizar e misturar partes entre os enfoques.	OS/2, Windows NT
HOMSuite (b)	RDD - Wirfs-Brock; gera C++ e Smalltal/V	Windows
I-CASE OMT (b)	Notação de Rumbaugh; gera SQL e C++	Windows
LOV/Object Editor (b)	Notação de Rumbaugh; gera C++	Unix, OSF/Motif
MetaEdit (c)	Notação de Booch, Coad/Yourdon, Rumbaugh	Windows 3.1
ObjecTime (b)	Notação do método ROOM (tempo-real); gera Smalltalk, C++	Unix
ObjectMaker (c)	Várias notações diagramáticas; customiza métodos e faz verificação com regras externas	Macintosh, VAX, Windows 3.1
ObjectModeler (b)	Notação de Rumbaugh, Coad/Yourdon, Jacobson e Booch; gera SQL, C++ e Smalltalk	Macintosh, Unix
ObjecTool (antiga OOA/OOD tool) (b)	OOA de Coad/Yourdon;sob Windows proporciona integração entre design e código C++	Windows, OS/2, HP/Sun Unix
Objectory (b)	Notação de Jacobson; gera C++	Windows, Unix
ObjectTeam (b)	Notação de Shlaer/Mellor, Rumbaugh; gera C++, Smalltalk	VAX/VMX, Unix, OS/2, PC-DOS
OEW (b)	Diagramas de Martin/Odell; gera C++	Sun OS, Windows 3.x
OMW (d)	Desenha e executa diagramas de Martin/Odell	Unix
OOther (b)	Notação de Coad/Yourdon, use case e diagrama de estado.	Windows 3.1
OMTool 2.0 (b)	Gera modelo gráfico e lógico para a OMT de Rumbaugh	Windows 3.1
Paradigm Plus (c)	Suporte a Booch e Coad/Yourdon; meta-case configurável	Windows, Unix
Prosa/om (b)	Notação de Coad/Yourdon; gera C++ SQL	Windows, Os/2, Motif
Ptech (d)	Notação de Martin/Odell; gera C++	Unix
Rose (b)	Notação de Booch; gera C++	Unix
System Architect (b)	Booch, Coad/Yourdon, Shlaer-Mellor; partes de design específico para Smalltalk, ADA e C++	Windows

Tabela 2.1 - Resumo de CASEs (Fonte: FAQ - Frequently Asked Questions, newsgroups da Usenet, setembro de 1994).

2.4 Métodos de análise/design

Diversas contribuições foram dadas à metodologia de análise/design, sendo que algumas estão mais próximas da OO.

Larry Constantine, ao introduzir os diagramas de estrutura de módulos e de fluxo de dados (DFD), é considerado o pioneiro no emprego de notação e ferramentas de modelagem (JACOBSON et al, 1992).

Michael Jackson estabeleceu as bases de como progredir de forma metódica na análise a partir da estrutura de dados (DEMARCO, 1978). Outra contribuição ao enfoque de dados é o diagrama hierárquico de Warnier, cuja notação para as construções de seqüência-seleção-repetição permite derivar a estrutura do software a partir da estrutura de dados (PRESSMAN, 1992).

As idéias de Constantine motivaram o interesse pelo design estruturado, cujo enfoque principal é o desenvolvimento da arquitetura do sistema segundo os princípios de modularidade, acoplamento e coesão entre módulos (YOURDON & CONSTANTINE, 1978).

Visando tornar mais claros os requisitos do sistema ao nível dos usuários, DEMARCO (1978), GANE & SARSON (1979), entre outros, apresentaram idéias e notações de uma disciplina que passou a ser conhecida por *análise estruturada*. Como alternativa aos pesados volumes de especificação até então existentes, a análise estruturada estimulou o uso do DFD, dicionário de dados, árvore e tabela de decisão, pseudo-código etc., e o método *top-down*, introduzido por WIRTH (1971). Este método cria inicialmente a lógica de mais alto nível e todas as interfaces importantes do sistema, antes que grande parte do código seja escrito, em oposição ao desenvolvimento considerado normal, o *bottom-up*, que divide o sistema em programas, implementa cada programa separadamente, e depois integra-os aos programas de controle.

FLAVIN (1981), citado por BELLIN & SUCHMAN (1993), acrescentou o componente de modelagem de informações, popularizada por vários autores, entre eles MARTIN & MACLURE (1991), sob a denominação de engenharia da informação, e

CHEN (1990), que contribuiu para o desenvolvimento do diagrama entidade-relacionamento. A orientação a dados, posteriormente aperfeiçoada com a hierarquia de tipos, é o enfoque que apresenta maior proximidade com a OO.

Conceitos e ferramentas para visualizar e verificar a operação de sistemas de tempo-real foram introduzidos por WARD & MELLOR (1985), através de extensões ao DFD. A extensão é obtida primariamente com o emprego do diagrama de transição de estados (HAREL, 1988).

Outra contribuição importante é devida a MCMENAMIN & PALMER (1984). Eles rompem com a abordagem clássica baseada em modelos físicos e lógicos, e propõem a criação de um modelo único denominado *modelo essencial*. Rompem também com a rigidez do desenvolvimento top-down, propondo o *particionamento de eventos*. O modelo essencial representa os aspectos do sistema que transcendem a todas as limitações tecnológicas possíveis, distingüindo entre os verdadeiros requisitos de um sistema e uma forma particular de implementação. O particionamento por eventos é um modelo de DFD onde os processos representam atividades essenciais, ou seja, atividades que seriam realizadas como resposta a um e apenas um evento se o sistema utilizasse uma tecnologia perfeita.

A modelagem essencial teve impacto no desenvolvimento posterior da metodologia estruturada. YOURDON (1989) enfatiza o modelo essencial, o particionamento de eventos, a integração dos modelos dados-eventos-função e encoraja a prototipação.

Paralelamente ao desenvolvimento de métodos e ferramentas, foram desenvolvidas também as etapas para combiná-los e orientá-los no processo de desenvolvimento de software, etapas essas conhecidas por *paradigmas de engenharia de software*. Quatro paradigmas são mais conhecidos: ciclo de vida clássico ou cascata, prototipação, modelo espiral e técnicas de quarta geração, sendo que o primeiro (clássico) é o mais antigo e utilizado (PRESSMAN, 1992).

As ferramentas da análise estruturada são comuns na OO, porém o ciclo de vida clássico é muito criticado, principalmente por separar de maneira rígida as fases de análise e design, como discutido em MONARCHI & PUHR (1992). Um ciclo alternativo denominado *fontain* (HENDERSON-SELLERS & EDWARDS, 1990) pode

ser mais apropriado ao desenvolvimento OO. O ciclo *fontain*, fundamentalmente, mostra uma representação mais clara da interação entre as fases de desenvolvimento do software, enfatiza análise e especificação de requisitos e mescla o desenvolvimento *top-down* em nível de sistema com o desenvolvimento *botton-up* de classes.

As contribuições à metodologia de análise/design OO são recentes. As primeiras publicações vieram a público a partir de 1988, porém o trabalho nessa área evolui rapidamente. Um levantamento atual indica a existência de 20 métodos[3].

Além dos métodos, outras idéias também são encontradas na literatura, tais como o *Real-Time Object-Oriented Design* (YAMAZAKI et al, 1993), um método específico para sistemas de telecomunicações; o *framework* de PITTMAN (1993), onde o autor abstrai idéias de diferentes métodos e sugere um ponto de apoio para o desenvolvimento OO; e as propostas de KORSON & MCGREGOR (1990) para o design de classes.

A seguir são resumidos alguns métodos citados com freqüência. Procurou-se destacar as características e os conceitos principais do processo da análise/design.

• OBJECT-ORIENTED REQUIREMENTS SPECIFICATION (OOS - Bailin)

A ***OOS*** (BAILIN, 1989) enfatiza a passagem da análise estruturada ao OOD. O autor pressupõe que já existe uma análise baseada em requisitos funcionais. Os processos introduzidos para transformar *input* em *output* são trocados por funções que operam sobre os mesmos dados. Como o critério para o agrupamento de funções é diferente em ambos os métodos, isto requer traçar um novo DFD. O novo DFD, denominado EDFD (diagrama de fluxo de dados de entidade) apóia-se em *entidades*, ao invés de processo, e *função*. Uma entidade possui um estado básico, enquanto uma função existe somente para transformar input em output.

[3] Fonte: FAQ - Frequently Asked Questions, newsgroups da Usenet, setembro/1994.

Os nódulos (círculos que representam os processos) do EDFD são de dois tipos: entidades[4] e funções, onde cada função deve ocorrer no contexto de uma entidade, isto é, deve ser chamada por ou atuar sobre uma entidade.

As entidades podem ser ativas ou passivas. As ativas são representadas por nódulos, enquanto as passivas aparecem como fluxos de dados ou depósitos de dados.

As etapas da OOS são as seguintes:

1. Identificar as entidades do domínio do problema. As fontes para descobrir essas entidades são: nomes de processos no DFD; os substantivos na especificação da base de dados e especificação de requisitos; e diagrama entidade-relacionamento.
2. Distinguir as entidades ativas e passivas.
3. Criar um EDFD onde os processos são as entidades ativas do diagrama entidade-relacionamento, e os fluxos/depósitos de dados são as entidades passivas.
4. Decompor as entidades em sub-entidades. Para cada sub-entidade identificada, criar um novo EDFD e continuar o processo de decomposição.
5. Verificar a existência de novas entidades, reorganizando o EDFD se necessário.
6. Agrupar as funções sob as novas entidades, reorganizando o EDFD se necessário.
7. Associar cada entidade introduzida na etapa 5 a algum domínio de aplicação e criar um conjunto de diagramas entidade-relacionamento, um para cada domínio, evitando que o diagrama original, que representa a aplicação, fique muito grande.

A ênfase do método concentra-se nas etapas de 4 a 6. O método amolda-se aos princípios da OO sem usar esta terminologia. O diagrama entidade-relacionamento atua como intermediário da classificação e herança, e o mapeamento de funções em entidades aproxima-se do conceito de encapsulamento de serviços.

Mesclar a análise estruturada com o design OO é uma estratégia adotada por vários autores. O centro da questão está em não trocar um método estruturado bem compreendido por um método OO menos familiar. WARD (1989) dedica-se a essa questão, além de oferecer diretrizes para realizar a transição.

[4] A OOS usa "entidade" na análise e "objeto" no design.

- OBJECT-ORIENTED STRUCTURED DESIGN (OOSD - Wasserman)

A ***OOSD*** (WASSERMAN et al, 1989; citados por FICHMAN & KEMERER, 1992 b), oferece uma notação híbrida detalhada para descrever um design de alto nível, denominado *design de arquitetura.* A meta é proporcionar uma notação padrão que possa suportar tanto o enfoque convencional quanto o OO.

A OOSD utiliza os símbolos de módulo, os parâmetros de dados e os parâmetros de controle do diagrama de estrutura convencional, e adiciona novos elementos para representar classes, objetos, métodos, instanciação, herança simples e múltipla, passagem de mensagem, polimorfismo, ligação dinâmica e monitor da programação concorrente.

Embora destinada ao design arquitetônico, a OOSD proporciona uma base para representar as decisões associadas ao design físico e recomenda que as anotações levem em conta a linguagem de implementação. A OOSD oferece uma notação opcional para pacotes ADA.

A ferramenta primária da OOSD é o diagrama de estrutura (Figura 2.1).

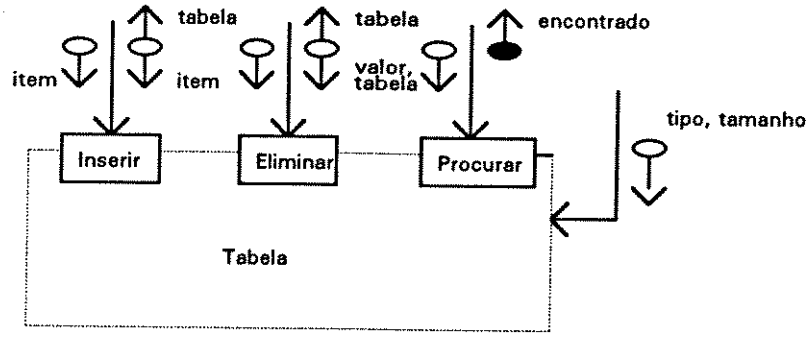


Figura 2.1 - Ilustração de uma classe em OOSD (WINBLAD et al, 1993).

O diagrama acima é uma versão atualizada do diagrama de estrutura clássico. O retângulo simboliza a classe Tabela. A linha pontilhada indica *classe genérica.* O

método faz distinção entre classe genérica (serve somente para suportar subclasse) e classe ordinária (serve para gerar objetos). Os retângulos menores representam os métodos Inserir, Eliminar e Procurar um item na classe Tabela. O círculo vazio e a seta indicam a passagem de dados entre blocos de processos. O círculo cheio mostra informação de controle. O diagrama mostra os elementos que parametrizam os atributos e as operações da classe. Para especializar uma subclasse de Tabela, basta passar os valores que indiquem o tamanho e o tipo de registro da subclasse, como mostra a seta apontando para a classe Tabela.

• OBJECT-ORIENTED ROLE ANALYSIS, SYNTHESIS AND STRUCTURING (OORASS - Reenskaug)

A ***OORASS*** (REENSKAUG & NORDHAGEN, 1989; citados por WIRFS-BROCK & JOHNSON, 1990), enfatiza a modelagem inicial de pequenos subproblemas para depois combiná-los num modelo maior de maneira controlada .

A utilização do termo OORASS deve-se aos três conceitos considerados fundamentais: herança, encapsulamento e ligação dinâmica. A análise descreve os subproblemas através do encapsulamento do comportamento dos objetos num modelo chamado *Role Model*. A síntese define os objetos que herdam o comportamento de objetos mais simples. A estruturação descreve como os objetos podem ser ligados como instâncias de um sistema.

O objetivo é tentar criar uma estrutura de objetos cooperantes e representar esta estrutura em computador. Os objetos são vistos a partir do conjunto de abstrações *how-what-why*, significando respectivamente: implementação, comportamento externo e a função do objeto.

A OORASS consiste de cinco etapas principais:

1. *Role Model*. Separa o domínio do problema por áreas de interesse. Cada área é modelada como uma estrutura de objetos que interagem. Cada objeto é identificado de acordo com a sua finalidade.

2. Especificação de objeto. Integra os modelos individuais deixando uma única estrutura de objetos desempenhar diferentes funções em diferentes modelos. Exemplo: uma pessoa pode ter a função de "entregador de material" num modelo de Manufatura e "comprador" num modelo de Compra.
3. Implementação de classe. Esta etapa proporciona os programas para os objetos. Na OORASS, classe é um programa que implementa uma especificação de objeto. É o único local onde a estrutura interna de um objeto é vista. Uma classe pode implementar objetos com várias funções e diferentes classes podem implementar objetos com uma função particular.
4. Especificação de estrutura. Proporciona uma gramática ou meta modelo para descrever como os objetos podem ser configurados para cooperarem entre si.
5. Instanciação de objetos. Cria e conecta os objetos de acordo com a descrição do meta modelo. Este é um processo tipicamente dinâmico onde novas instâncias são criadas e conectadas durante a execução da aplicação, e as não usadas viram "lixo".

O método não se refere ao uso de uma ferramenta em particular. Na primeira etapa várias ferramentas existentes podem ser utilizadas pois os diagramas são simples.

• OBJECT-ORIENTED ANALYSIS (OOA - Shlaer & Mellor)

A *OOA* (SHLAER & MELLOR, 1990), cujo título original "Object -Oriented Systems Analysis: Modeling the World in Data" foi publicado em 1988, é considerado o primeiro método de análise OO, com um forte componente derivado da modelagem de informação.

A OOA evoluiu e a partir de outra publicação (SHLAER & MELLOR, 1992; citado por FICHMAN & KEMERER, 1992 b) os autores propõem realizar a análise de acordo com as seis etapas seguintes:

1. Desenvolver um modelo de informações consistindo de objetos, atributos e relacionamentos. Os objetos podem ser: coisas tangíveis (são os objetos propriamente ditos, tais como avião, disjuntor de circuito, livro, reator etc.); funções exercidas por pessoas ou organizações (cliente, supervisor, proprietário, departamento etc.); ocorrências ou eventos (acidente, falha do sistema etc.); contrato ou transação[5].

[5] Objetos *contrato* ou transação normalmente referem-se a dois ou mais objetos. Por exemplo, "Compra" refere-se aos objetos Comprador, Vendedor e Item Adquirido.

2. Definir o ciclo de vida dos objetos do problema, da criação à destruição, formalizando o ciclo numa coleção de estados, eventos e regras de transição. A ferramenta usada nesta etapa é o diagrama de transição de estados para modelar o estado das entidades, em contraste com o diagrama de transição tradicional, que modela o sistema.
3. Definir a dinâmica do relacionamento entre as instâncias dos objetos caso haja concorrência para usar os recursos das instâncias de outros objetos.
4. Definir a dinâmica do sistema. Este passo gera dois modelos: de comunicação, que mostra o controle assíncrono entre o modelo de estados e as entidades externas; e de acesso ao objeto, que mostra o controle síncrono entre o modelo de estados e o sistema global.
5. Desenvolver o modelo dos processos. Para cada ação, desenvolver um DFD-ação para modelar ações elementares.
6. Definir domínio e subsistemas. Para grandes domínios, decompor em quatro domínios distintos: aplicação, serviço, arquitetura e implementação.

Na OOA há suporte implícito para classificação, herança e encapsulamento, porém deve haver um mapeamento para que esses elementos apareçam no design. O fato de *ação* (e DFD-ação associado) estar associado a exatamente um objeto preserva o encapsulamento dessas operações.

• OBJECT-ORIENTED ANALYSIS (OOA - Coad & Yourdon)

A *OOA* (COAD & YOURDON, 1990) é um método construído sobre os conceitos de modelagem de informação e linguagem de programação OO. Propõe o desenvolvimento de um modelo em cinco camadas (ou atividades), e o emprego de diagramas complementares tais como DFD, diagrama de transição de estados, diagrama de tempo e diagrama de ciclo de vida de objeto para modelar os serviços dos objetos.

As cinco camadas são:

1. Definir objetos e classes. Os objetos podem ser: funções exercidas por pessoas; sistemas ou dispositivos com os quais a aplicação interage; localizações; e eventos.
2. Definir as estruturas de generalização/especialização e todo/partes.

3. Definir a camada de *subject* (assunto), um mecanismo para examinar objetos de mais alto nível. As estruturas da etapa 2 fornecem o ponto de partida para a definição desta camada.
4. Definir os atributos e os relacionamentos entre os objetos.
5. Definir os serviços e os pedidos de serviço entre os objetos.

Um exemplo é dado a seguir (Figura 2.2).

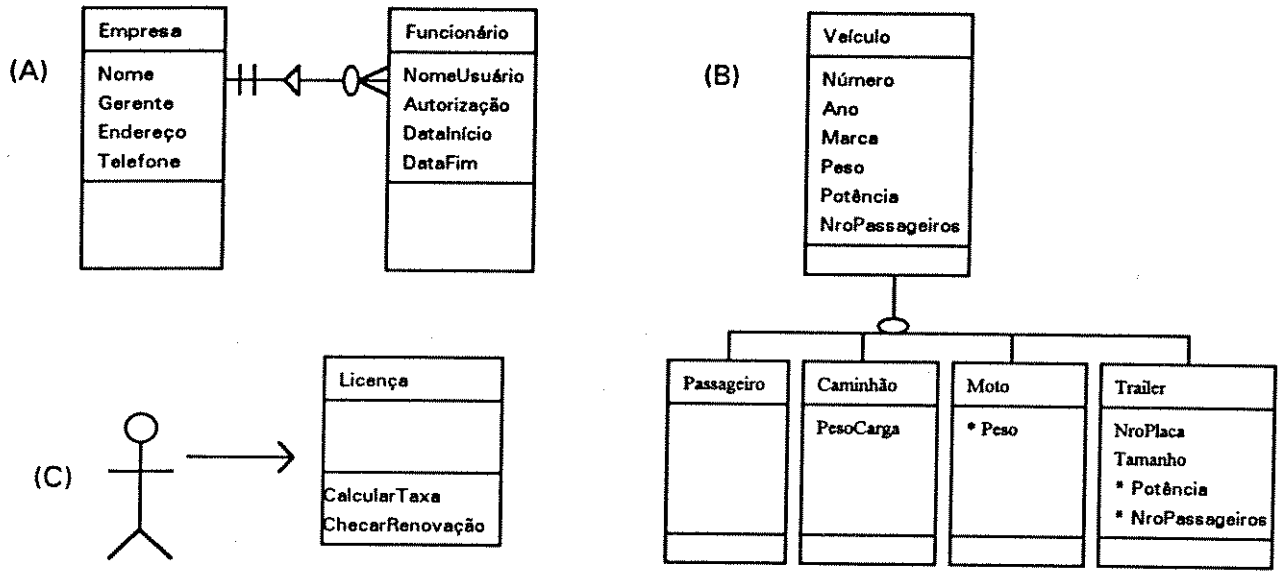


Figura 2.2 - Principais elementos da OOA (COAD & YOURDON, 1990).

Um objeto é representado por um retângulo dividido em três. Na parte superior destaca-se o nome do objeto e nas partes central e inferior são anotados respectivamente os seus atributos e serviços. A OOA fornece diretrizes para identificar e alocar os atributos e os serviços aos objetos.

A Figura 2.2 (A) representa a estrutura *todo/partes*, onde o triângulo indica que o objeto Empresa (todo) é formada por uma coleção de objetos Funcionários (partes).

A Figura 2.2 (B) representa a estrutura *generalização/especialização*. O objeto Veículo é uma classe geral e os objetos Passageiro, Caminhão, Moto e Trailer são

classes especializadas. Os atributos de Veículo são transferidos implicitamente para as classes especializadas. Os objetos Caminhão e Trailer adicionam, respectivamente, os atributos PesoCarga, Nroplaca e Tamanho. Os objetos Moto e Trailer cancelam os atributos não herdados da classe geral (indicado pelo asterisco "*").

A Figura 2.2 (C) mostra os serviços definidos para um objeto e a interação humana.

O diagrama pode ser expandido para realizar o design detalhado. Os autores sugerem, para uma arquitetura de hardware distribuída, anotar no próprio diagrama quais objetos devem ser copiados ou distribuídos. Em termos de arquitetura de software, se houver processos concorrentes, utilizar um diagrama à parte para mostrar as tarefas, prioridades, instanciações múltiplas, comunicação e sincronização (via rendezvous, semáforos ou outros mecanismos). Os autores sugerem ainda a normalização de dados.

• RESPONSIBILITY-DRIVEN DESIGN (RDD - Wirfs-Brock)

O ***RDD*** (WIRFS-BROCK & JOHNSON, 1990) baseia-se nas responsabilidades e colaborações dos objetos através do modelo cliente-servidor, o conceito de *contrato,* cartões CRC e cenário.

O método enfatiza a exploração de alternativas nas etapas iniciais do design, oferecendo meios para registrar e alterar as decisões de design. É orientado a responsabilidade pois força a simulação do sistema através das ações que devem ocorrer e dos objetos que devem realizá-las.

Segundo SOUZA (1993), a técnica empregada em RDD é bastante informal, semelhante à análise de cenários, onde os cenários são compostos por pessoas que são "estimuladas a serem classes". Com os cartões CRC, a ênfase em antropomorfismo[6] e aplicando-se vários cenários, o modelo vai sendo aperfeiçoado, culminando com a alocação de classes mais primitivas.

[6] Aplicação a algum domínio da realidade (social, biológico, físico, etc.) de linguagem ou de conceitos próprios ao homem ou ao seu comportamento.

O cartão CRC (Class-Responsibility-Collaboration), idealizado por Ward Cunningham (BECK & CUNNINGHAM, 1989), é um cartão de índice 4" x 6" onde anotam-se as três funções de um objeto no design: o nome da classe, as suas responsabilidades e as classes com as quais colabora (Figura 2.3). As responsabilidades identificam os problemas a serem resolvidos (soluções potenciais) e as colaborações são os demais objetos para os quais envia ou recebe mensagem.

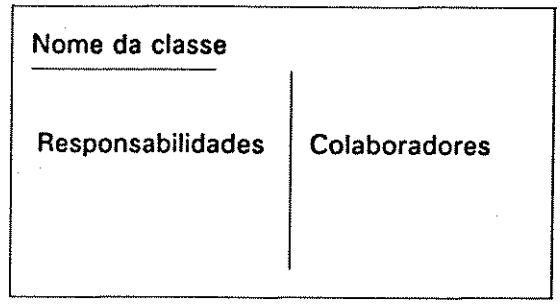


Figura 2.3 - Cartão de índice CRC (BECK & CUNNINGHAM, 1989)

O RDD propõe dois estágios (Figura 2.4):

1. Estágio exploratório. Usando os cartões CRC, descobrir as classes e modelar o comportamento do sistema associando as responsabilidades e os colaboradores.
2. Estágio de análise. As atividades primárias são: fatorar as responsabilidades comuns; modelar as colaborações entre os objetos e identificar como subsistema; determinar os protocolos; completar a especificação das classes e o contrato cliente-servidor.

O contrato cliente-servidor e o grafo de colaboração (Figura 2.5) são outras ferramentas do RDD. O contrato descreve a forma pela qual um cliente pode interagir com o servidor, ou seja, o cliente apresenta os seus pedidos e o servidor responde apropriadamente aos pedidos. Uma classe pode suportar um ou mais contratos, dependendo se seus serviços forem usados por um ou mais clientes. O grafo de colaboração ajuda a analisar os caminhos de comunicação e identificar os subsistemas. Subsistemas são conjuntos de classes que colaboram para satisfazer um conjunto comum de responsabilidades.

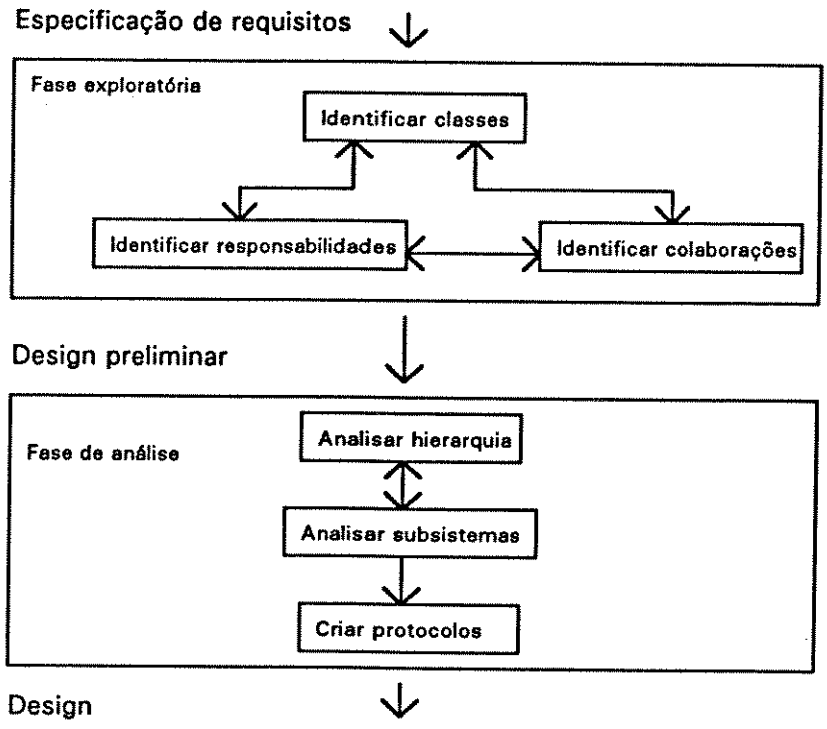


Figura 2.4 - Fases de exploração e análise (WIRFS-BROCK & JOHNSON, 1990).

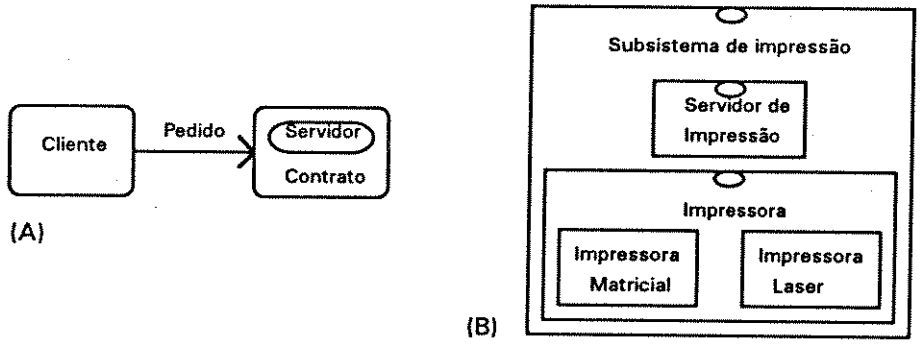


Figura 2.5 - (A) Contrato cliente-servidor; (B) Grafo de colaboração (WIRFS-BROCK & JOHNSON, 1990).

A seta na Figura 2.5 (A) indica o servidor que suporta o contrato. A Figura 2.5 (B) é um exemplo de grafo de colaboração de um subsistema de impressão, onde os retângulos representam as classes Servidor de Impressão e Impressora, o aninhamento de retângulos representa o relacionamento superclasse/subclasse e as elipses representam contratos. Subsistemas são entidades conceituais que não existem durante

a execução. Um subsistema não faz contratos, ele somente transfere os contratos às classes nele contida.

O RDD propõe um guia básico para simplificar o padrão de colaboração entre as classes num grafo de colaboração, sempre tendo em vista que os grafos são desenhados repetidas vezes para testar alternativas de design. Oferece também métodos específicos tais como *base*, *abstract* e *template*, para implementar uma classe. Métodos *base* implementam o comportamento que pode ser herdado pelas subclasses; métodos *abstract* proporcionam o comportamento *default* que as classes ocultam; e métodos *template* definem os algoritmos para os serviços das subclasses.

• OBJECT-ORIENTED DESIGN (OOD - Booch)

O ***OOD*** (BOOCH, 1991) é considerado pelo autor como alternativa ao design estruturado, pois a decomposição de sistema empregada na OO é antagônica à decomposição no design estruturado, ou decomposição algorítmica, onde cada módulo denota uma etapa principal num processo. Booch sugere uma decomposição baseada nas abstrações-chave do domínio do problema, composta por objetos que exibem um comportamento bem definido. A decomposição por algoritmos enfatiza o ordenamento de eventos, enquanto a decomposição por objetos destaca os agentes autônomos que enviam e reagem às mensagens recebidas.

No OOD não há ordenamento fixo de fases, pelo contrário, recomenda que analistas e designers trabalhem interativamente e incrementalmente, combinando diagramas formais e técnicas informais. O foco é o design e implementação.

O OOD propõe um processo em quatro etapas:

1) Identificar as classes e os objetos.
2) Identificar a semântica de classes e objetos, usando técnicas como ciclo de vida.
3) Identificar o relacionamento entre as classe e os objetos (herança entre classes e cooperação entre objetos).

4) Implementar as classes e objetos, construindo os detalhes internos e definições de comportamento (serviços); alocar as classes e objetos aos módulos (como definido no ambiente da linguagem destino) e alocar os programas aos processadores.

Booch sugere que os objetos podem ser identificados a partir do documento de especificação, onde os substantivos podem indicar os objetos do problema. Booch considera porém que não existe uma "receita" pronta. Para ilustrar, o autor cita um fato ocorrido durante uma conferência em engenharia de software, onde diversos desenvolvedores foram argüidos sobre quais regras eles aplicavam para identificar classes e objetos. Stroustrup, o projetista de C++, respondeu: "Isso é o Santo Graal! Não há panacéia". Gabriel, um dos projetistas do CLOS, declarou: "Essa é uma questão fundamental, para a qual não há resposta fácil. Eu tento coisas!" (BOOCH, 1991).

Booch oferece uma notação icônica bastante rica ao processo de design, descrevendo em detalhes as atividades a serem executadas. Inclui quatro diagramas básicos: classe, objeto, módulo e processo, para mostrar as propriedades estáticas do sistema. Os diagramas de classe e de objeto são partes da visão lógica pois descrevem a existência e o significado das abstrações principais do design. Os diagramas de módulo e processo são partes da estrutura física e auxiliam a descrever os componentes reais de software e hardware de uma implementação. Dois diagramas complementares, transição de estados e tempo, mostram a visão dinâmica. As classes podem ter uma indicação de como o ordenamento de eventos externos pode afetar o estado das suas instâncias. Adicionalmente, o diagrama de tempo ou uma PDL (Program Definition Language) podem ser usados para documentar a dinâmica de como os processos são organizados num diagrama de processos.

• OBJECT MODELING TECHNIQUE (OMT - Rumbaugh)

A *OMT* (RUMBAUGH et al,1991) suporta o ciclo de vida completo de um software. Os conceitos centrais são *cenário*, *evento* e *estado*. Esses conceitos e a notação são apresentados em termos de três modelos: objetos, dinâmico e funcional. O método apresenta também o processo de formulação desses modelos.

O modelo de objetos descreve a estrutura dos objetos: identidade, relacionamentos, atributos e operações. É representado graficamente pelo diagrama de objeto (Figura 2.6).

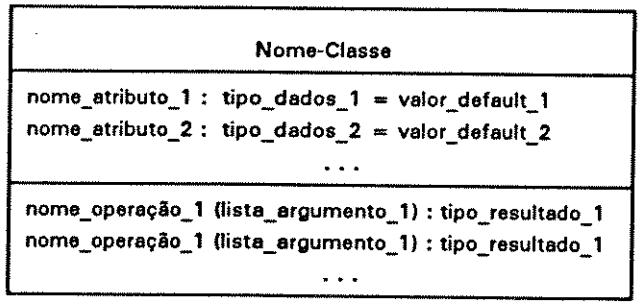


Figura 2.6 - Notação para classes de objetos (RUMBAUGH et al, 1991).

O modelo dinâmico descreve, através de *evento, estado* e *cenário,* os aspectos associados ao tempo e à seqüência de operações que ocorrem devido aos estímulos externos. Evento é a transmissão de informação de um objeto para outro. Estado representa o intervalo entre os eventos. Cenário é uma seqüência de eventos que ocorre durante uma execução particular do sistema. Esses conceitos refletem a visão dos autores em relação aos objetos: "Os objetos estimulam-se uns aos outros, resultando numa série de mudanças em seus estados internos. No mundo real os objetos existem concorrentemente. Um objeto envia um evento a outro objeto e pode aguardar a resposta. A resposta é outro evento separado, sob controle do objeto receptor, que pode decidir entre retornar ou não a resposta" (RUMBAUGH et al, 1991).

O modelo dinâmico é representado graficamente por vários diagramas de estado, um para cada classe com comportamento dinâmico relevante. Cenários, por sua vez, são mostrados num diagrama denominado *event trace*. Um exemplo de *event trace* é mostrado na Figura 2.7.

Na Figura 2.7, os eventos transmitem a informação de um objeto para outro ou a ambos os objetos. Por exemplo, "inicia o sinal de linha" transmite um sinal do objeto Linha Telefônica ao objeto Pessoa que inicia a chamada. A Linha Telefônica envia eventos para ambos os objetos concorrentemente. O diagrama representa os objetos

como linha vertical e os eventos como linha horizontal. A seta indica a direção da transmissão. Somente a seqüência de eventos é mostrada.

O modelo funcional descreve a transformação de valores, isto é, como os valores de saída originam-se dos valores de entrada de dados. É representado graficamente pelo DFD, que inclui um novo elemento para simbolizar a criação de objetos. Os conceitos principais são objetos ativos, ou *actors*, caracterizados por produzir ou consumir dados, e objetos passivos, caracterizados por somente responder aos pedidos de armazenamento ou acesso a dados. Os fluxos de dados também são objetos, mas com valores puros, como por exemplo, um número inteiro.

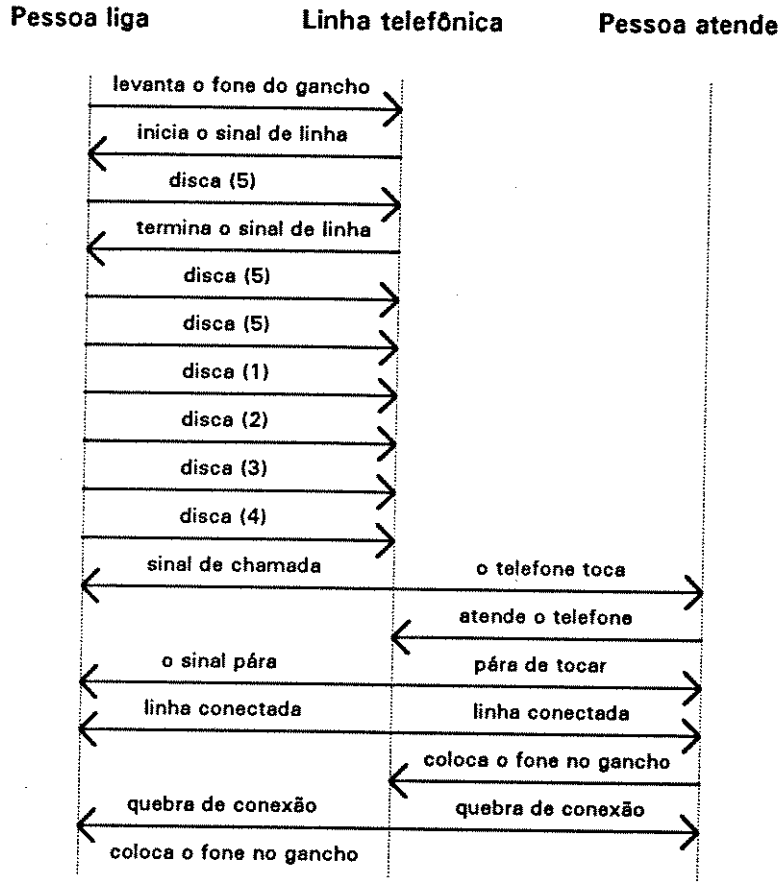


Figura 2.7 - *Event trace* de uma ligação telefônica (RUMBAUGH et al, 1991).

Os três modelos estão relacionados entre si, porém a base é o modelo de objetos. Segundo os autores: "Os relacionamentos temporais são difíceis de serem

compreendidos. Um sistema pode ser melhor compreendido examinando-se inicialmente a sua estrutura estática, isto é, a estrutura de seus objetos e seus relacionamentos com outros objetos num determinado instante. Depois examinam-se as mudanças nos objetos e seus relacionamentos durante um intervalo maior de tempo" (RUMBAUGH et al, 1991).

Os autores sugerem um processo interativo de desenvolvimento dos modelos de objeto, dinâmico e funcional, composto por três fases: análise, design de sistema e design de objetos. Na análise tem-se os três aspectos do sistema: objetos e seus relacionamentos, fluxo de controle e transformações funcionais. No design de sistema tem-se a arquitetura geral do sistema. No design de objetos propõem escolher os algoritmos básicos para implementar as funções principais do sistema.

• OBJECT-ORIENTED SOFTWARE ENGINEERING (OOSE - Jacobson)

A ***OOSE*** (JACOBSON et al, 1992) é uma versão simplificada do método *Objectory* (*Object Factory for Software Development*). O método utiliza as recomendações do "Commité Consultatif International Télegraphique et Téléphonique". A OOSE originou-se de três áreas técnicas: programação OO, modelagem conceitual e *block design*. Os conceitos centrais são *actor* e *use case*.

O método utiliza cinco modelos e três processos principais. Os modelos são: requisitos, análise, design, implementação e teste. Os processos, que realizam a transição entre os modelos, são: análise, construção e teste (Figura 2.8).

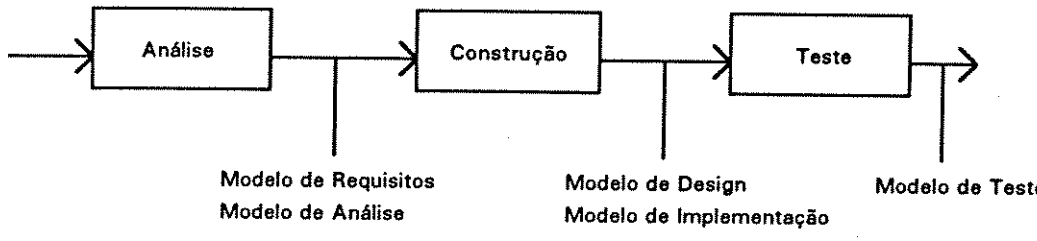


Figura 2.8 - Modelos e processos da OOSE (JACOBSON et al,1992).

O processo de análise procura criar uma imagem conceitual do sistema a ser desenvolvido, incluindo a descrição de como o usuário irá utilizar o sistema. Assume um ambiente de implementação ideal. Este processo desenvolve dois modelos: requisitos e análise.

O processo de construção desenvolve o sistema a partir dos modelos gerados na análise, além de criar dois outros modelos: design (para integrar os elementos do ambiente de implementação) e implementação (codificação).

O processo de teste integra o sistema, verifica-o e decide pela continuidade e entrega do produto.

Um quarto processo, não mostrado na Figura 2.8, desenvolvimento de componentes, associado principalmente ao processo de construção, desenvolve e mantém os componentes que podem ser usados em diferentes aplicações.

Os autores consideram que o desenvolvimento de sistemas é uma transformação gradual de uma seqüência de modelos. O primeiro modelo descreve os requisitos do usuário e o último testa totalmente o produto. Para cada modelo, diferentes tipos de objetos são definidos, obtendo-se então todos os benefícios da OO, principalmente a facilidade de localizar os pontos de manutenção, encapsulamento e reuso.

Os autores consideram também que o critério mais importante para incluir um objeto num contexto é saber como, e sob quais condições, o objeto coopera com os demais. Como bem destacam: "Um objeto não existe por si próprio, ele pode ser perfeitamente correto num modelo mas totalmente errado em outro. Um bom objeto deve ser robusto diante de uma provável modificação e ajudar a compreender o sistema" (JACOBSON et al, 1992). Neste aspecto eles se opõem às estratégias para se descobrir os objetos, tais como as propostas por Coad & Yourdon, Shlaer & Mellor e outros autores.

O modelo de requisitos é determinado totalmente pelos requisitos funcionais do usuário. Consiste de *actor* e *use case* (Figura 2.9), suportados por um modelo de objetos do domínio e descrições de interface. Esses conceitos servem para definir o que existe fora do sistema (*actor*) e o que deve ser executado pelo sistema (*use case*). *Actor* representa uma função que um usuário pode exercer no sistema, enquanto *use*

case é uma maneira específica de utilizar o sistema. *Use case* significa somente uma parte da funcionalidade do sistema. Cada *use case* constitue um curso completo de eventos iniciado por um *actor*. *Use case* representa, portanto, a interação entre *actor* e sistema.

O sistema é representado pelo retângulo. A*ctor* é representado por pessoas fora do retângulo e *use case* é representado pelas elipses.

O modelo de *use case*, o modelo de objetos do domínio e a descrição em detalhes das interfaces definem totalmente os requisitos funcionais do sistema a partir da perspectiva do usuário.

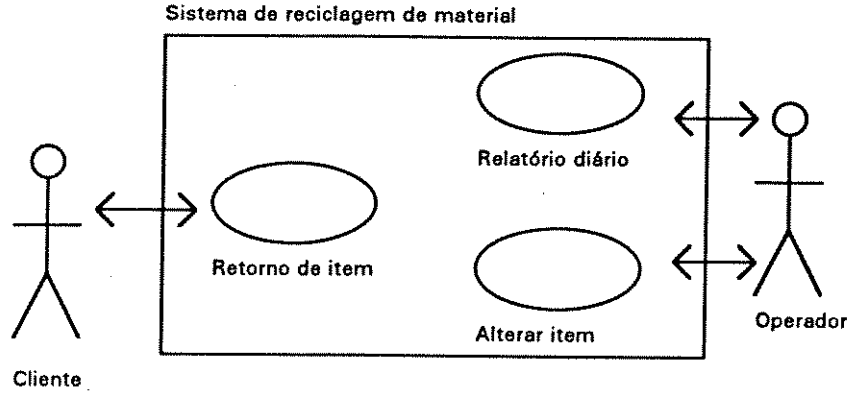


Figura 2.9 - Um modelo de *use case* (JACOBSON et al, 1992).

O modelo de análise é desenvolvido a partir do modelo de requisitos. O objetivo é obter uma estrutura lógica e robusta que seja manutenível durante o ciclo de vida do sistema. Três tipos de objetos são definidos: entidade (são os objetos mais estáveis como, por exemplo, objeto Pessoa); interface (são os objetos dependentes da interface de usuário); e controle (representam funções não vinculadas especificamente a algum objeto).

O modelo de design refina e formaliza o modelo de análise, preparando-o para a implementação. Utiliza o conceito de *bloco* para descrever a implementação do sistema. Os blocos são os objetos de design que implementam os objetos da análise.

São especificados usando o modelo de *use case*, o conceito de estímulo[7] e diagramas de interação (Figura 2.10). Para cada *use case*, descreve-se como e qual estímulo deve ser enviado, além da ordem de transmissão. Um *use case* é descrito então como uma seqüência de estímulos transmitidos entre os blocos. O design fica completo quando todos os *use cases* forem descritos.

O modelo de implementação consiste principalmente do código-fonte necessário para implementar os blocos. Aqui deve haver uma associação entre o conceito de bloco da OOSE e outros conceitos similares nas linguagens de programação, tais como pacote, classe, módulo etc..

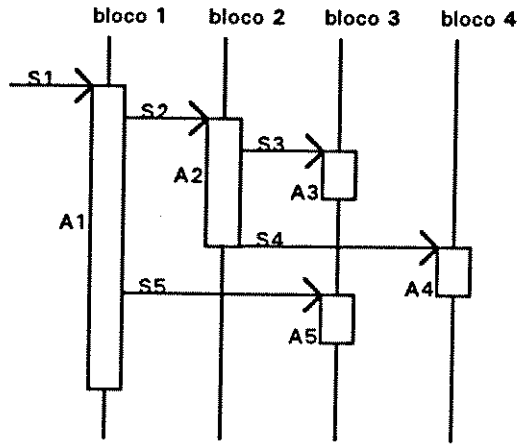


Figura 2.10 - Exemplo de diagrama de interação. Os estímulos (Sn) são transmitidos entre os blocos, desencadeando as atividades (An) nos blocos (JACOBSON et al, 1992).

Outra ferramenta de implementação é a habilidade de usar componentes, que oferecem maior abstração do que as linguagens de programação e podem ser usados diretamente na implementação. O método oferece os detalhes de como usar os componentes.

[7] Os autores utilizam o termo "estímulo" ao invés de "mensagem". Estímulo indica que algum comportamento vai se transformar em ação no objeto receptor e não necessariamente inclui as informações normalmente associadas a mensagem.

O modelo de teste descreve o resultado do teste, que é desenvolvido em diferentes níveis de granularidade. No teste de unidade são testados os objetos de menor nível. O modelo de *use case* é usado como ferramenta primária no teste de integração.

• OBJECT-ORIENTED ANALYSIS (OOA - De Champeaux)

A ***OOA*** (DE CHAMPEAUX et al, 1993) oferece uma notação gráfica (para análise) e textual (para design), e diretrizes de como progredir dos requisitos para a implementação. A notação também é indicada como "front-end" para quaisquer outros métodos de análise/design OO, linguagens de programação ou notações específicas.

Para construir um modelo de sistema, a OOA sugere um "corte vertical" na análise, onde a validade do modelo pode ser testada com a execução de cenários entre analistas e clientes. Além de cenários, a OOA enfatiza *use case* e subsistemas[8], e introduz o conceito de *ensemble.*

Ensemble são geralmente grandes encapsulamentos de objetos com propriedades especialmente definidas para representar diferentes camadas de abstração e permitir a decomposição *top-down.* Ensembles assemelham-se a objetos pois possuem atributos, podem ter uma máquina de transição de estados associada e podem interagir com outros objetos. Diferem dos objetos pelo fato de agrupar objetos (ou outros ensembles) de menor nível denominados *componentes.* Os componentes ficam ocultos de outros objetos e interagem somente com os componentes pertencentes ao mesmo ensemble. Desta maneira, um ensemble pode ser visto como uma ponte entre seus componentes e o resto do sistema.

A construção do modelo envolve duas atividades. A primeira baseia-se em protótipo e a segunda é o processo real da análise.

[8] O significado de *use case*, cenário e subsistema são os mesmos já definidos nesta seção.

O modelo de protótipo:

1. Um fragmento de requisitos, em geral correspondendo a uma narrativa em alto nível que explica a funcionalidade do sistema.
2. Alguns cenários (*use cases*) para descrever as partes do sistema do ponto de vista do usuário.
3. Alguns subsistemas.
4. Um vocabulário consistindo de uma descrição estruturada de algumas classes, como preparação para a sua posterior caracterização no formalismo gráfico.
5. Uma expansão, baseada em *use cases*, de alguns elementos do vocabulário.
6. Criar e testar um protótipo com algumas instâncias, classes, ensembles e relacionamentos, usando os diagramas de interação. Realizar mentalmente alguns *walk-throughs* (ensaio) para verificar se o modelo satisfaz os cenários de *use cases*.

O modelo de análise:

1. Obter os requisitos "completos".
2. Descrever a interação no contexto do sistema através de *use cases*.
3. Delinear os subsistemas. Pode-se adotar uma decomposição que já foi introduzida manualmente. (Um exemplo de subsistema pode ser: ler dados, processar e retornar a resposta).
4. Desenvolver o vocabulário identificando-se as instâncias com suas classes, ensembles e relacionamentos. Pode-se desenvolver um vocabulário para cada subsistema. Os objetos podem ser: os substantivos em um documento de especificação; as entidades externas e depósito de dados no DFD; os elementos que apresentam diversos estados num diagrama de estado; uma entidade no diagrama entidade-relacionamento; abstrações em geral (pessoas, organizações, eventos etc.).
5. Aperfeiçoar as classes. Descrever os elementos estáticos (atributos), os relacionamentos entre classes (incluindo herança), e a interação entre as classes. A interação pressupõe: estilos de interação; transição de estados e seus componentes (guarda, ação e eventos); primitivas de comunicação (rendezvous); e protocolos (acknowledgment, callback, fowarding etc.).
6. Construir um modelo onde a dinâmica dos objetos estejam todas interligadas.

A OOA oferece um conjunto de diagramas e mecanismos para implementar as interações, classes, instanciação, ensembles, herança múltipla etc..

2.5 Considerações finais

A revisão de literatura apresentada neste capítulo focalizou aspectos da situação atual da OO em termos da motivação, tendências, conceitos básicos, produtos e métodos.

O interesse na OO decorre do fato de oferecer uma base para entendimento e representação de sistemas complexos. Proporciona também uma transição suave entre análise, design e implementação, favorece a obtenção de modelos estáveis, e permite a construção de sistemas flexíveis, diante dos quais pode-se realizar a manutenção de maneira rápida e eficiente.

Para a OO se afirmar, no entanto, alguns problemas devem ser resolvidos, tais como descobrir meios de medir a qualidade da análise/design, testar a OO em grandes projetos, especialmente em sistemas de informação, e criar um consenso sobre particionamento. O elemento principal de mudança está em enfatizar a filosofia de objetos independentemente do meio técnico de usar o computador.

Em relação às linguagens de programação e aos métodos, dois grandes enfoques são identificados: um enfoque híbrido onde a OO convive com as técnicas estruturadas, e um enfoque puro.

Uma diferença básica nas linguagens puras, tal como a Smalltalk, é que o conceito de objetos é tomado ao pé da letra. Isso pode ser um incômodo quando, por exemplo, pretende-se portar um sistema já existente para uma arquitetura de objetos. Neste caso o custo pode ser muito grande, dado que seria difícil aproveitar o que já existe (procedimental) em Smalltalk. A linguagem, no entanto, força o conceito de objeto. Em seu ambiente sabe-se que todos os métodos são públicos e todos os dados são privados, e que ambos devem estar associados a algum objeto. Isso evita a prática errônea de se deixar esses elementos soltos no programa.

Quanto aos métodos de análise/design, a abordagem híbrida consiste em aproveitar a experiência considerada madura da análise estruturada, e os sistemas já desenvolvidos, e caminhar em direção ao design OO. Essa idéia é explorada nos

primeiros trabalhos (Bailin, Ward e Wasserman). Nos métodos mais recentes a ênfase é inicialmente colocada nos objetos (Coad & Yourdon, Booch, Jacobson, Wirfs-Brock, De Champeaux, entre outros). Aqui as técnicas estruturadas são usadas normalmente, porém ficam num segundo plano. Este enfoque parece oferecer um suporte melhor pois os objetos, uma vez identificados na análise, passam naturalmente ao design e à implementação. Em qualquer fase, a base é sempre a mesma: objetos. No outro caso, devido à mudança de perspectiva, deve haver um esforço adicional para realizar a transição da análise estruturada, em geral feita com DFD, ao design OO sem perdas.

De uma maneira geral os métodos têm pontos em comum e diferenças. Os principais pontos comuns são os conceitos básicos (classe, herança, polimorfismo etc.), a descrição dos elementos estáticos e dinâmicos, a identificação do papel ou função de um objeto no modelo, o rompimento com o ciclo de vida clássico e a preferência pelos ciclos *fontain* ou a prototipação, e ênfase em reuso, principalmente através de criação de *framework* em nível corporativo.

As principais diferenças são a notação, o foco e alguns conceitos exclusivos. Métodos como OOA-Coad & Yourdon propõem uma notação simples enquanto outros, como OOD-Booch e OOA-De Champeaux, usam uma variedade de diagramas, alguns difíceis de desenhar. Existem métodos cujo foco é a análise, ou o design, ou os que visam a preparação para a programação. Existem ainda métodos que cobrem o ciclo de vida completo. Conceitos exclusivos, definidos no âmbito de um método específico, tais como *ensemble*, *subject*, *cenário* e *use case*, é outro ponto onde as diferenças são visíveis.

Ferramentas CASE são decisivas em função da variedade de diagramas, modelos e check-lists propostos na metodologia. A única provável exceção é o RDD - Wirfs-Brock, que parece ser feita para trabalhar com papel. Os CASEs atuais estão, em geral, vinculados a algum método. Os métodos mais adotados pelos CASEs são OOD-Booch, OOA-Coad & Yourdon, OOA-Shlaer & Mellor, OOSE-Jacobson e OMT-Rumbaugh. Três características das ferramentas são notação gráfica, verificação de erros ou omissões e geração de código (especialmente C++ e Smalltalk).

A metodologia atual continua em evolução. Alguns pontos possivelmente fracos deverão ser resolvidos por amadurecimento. Uma notação padronizada é desejável

para expressar classes, instâncias, relacionamentos, modelos estáticos, modelos dinâmicos e unidades de particionamento.

Algumas iniciativas já existem no sentido de aproveitar propostas bem sucedidas e combinar a outras mais recentes. Por exemplo, a idéia de *use case*, inicialmente formulada por JACOBSON et al (1992), que parece ser uma evolução do particionamento de eventos, é também empregada na OOA (DE CHAMPEAUX et al, 1993). Isso cria um ponto de referência importante para os analistas e usuários.

---

3 - Detalhes de projeto

Neste capítulo são apresentados os detalhes da ferramenta proposta. O capítulo é dividido em quatro seções. Na Seção 3.1 são declarados os requisitos do software, obtidos a partir da revisão de literatura e sugestões pessoais. A Seção 3.2 apresenta o modelo de objetos que serviu de base para a construção da ferramenta. A Seção 3.3 mostra a arquitetura geral do software. A Seção 3.4 presta informações técnicas sobre a implementação.

3.1 Requisitos do software

3.1.1 Objetivos gerais

O objetivo principal da ferramenta é auxiliar a análise orientada a objeto através da descrição das características estáticas. Deverá permitir a realização de operações típicas de base de dados (inserir, excluir, alterar, procurar) com nomes de objetos, atributos, serviços, tipos de dados e relacionamentos, e incorporar o mecanismo de herança.

Deseja-se que os dados armazenados devido à operação em um modelo possam ser reutilizados na criação de outros modelos.

A ferramenta deverá prover a verificação do relacionamento inverso, ou seja, dada uma propriedade (um nome de atributo ou de um serviço já armazenado na base de dados), listar os objetos que apresentam tal propriedade.

O software deverá ser executado em um microcomputador compatível com o IBM-PC. Deseja-se, entretanto, que também possa ser executado em outras máquinas.

É desejável ainda que os dados armazenados possam ser usados por outros sistemas aplicativos.

3.1.2 Requisitos funcionais

1. A ferramenta deverá oferecer recursos de gerência de dados no sentido de permitir ao usuário incluir, excluir, alterar, pesquisar e exibir os objetos, atributos, serviços, relacionamentos e tipos de dados desejados num modelo.

2. A operação de *inserção* dos elementos declarados no item 1 deverá ser facilitada por um estilo onde o usuário possa consultar livremente uma lista ordenada de elementos. A tentativa de inserir um objeto já existente deverá ser bloqueada no ato. O usuário deverá ser informado sobre esta possível causa de erro através de uma mensagem clara.

3. Deverá ser criado um repositório de tal forma que os objetos referidos no item 1, à medida em que forem definidos num modelo, fiquem automaticamente disponíveis para outros modelos. A operação no repositório não necessariamente deverá estar associada a um modelo.

4. A ferramenta deverá oferecer recursos simples de edição de texto para auxiliar a especificação dos atributos e serviços associados a um objeto. Em geral o texto é breve, porém deverá haver um espaço suficiente para escrever textos de tamanho variado. Se necessitar impor um limite, esse limite deverá ficar claro ao usuário.

5. A herança poderá ser simples ou múltipla. A transmissão de herança deverá ocorrer da seguinte forma: quando o usuário escolher o relacionamento "generalização / especialização" entre dois objetos, os atributos e os serviços (com as respectivas especificações) do objeto "geral" devem ser copiados para o objeto "especializado", sem criar conflito com o exposto no item 2 (tentar inserir algo já existente). Os atributos e serviços copiados para o objeto "especializado" deverão ser tratados normalmente pelo editor de texto.

6. A ferramenta deverá oferecer um recurso utilitário que informe, dado um atributo ou serviço existente, para qual ou quais objetos eles foram definidos.

7. A saída impressa deverá ocorrer na forma de texto estruturado. O usuário poderá optar por imprimir um modelo qualquer ou o repositório completo.

3.2 Modelo de objetos

O modelo de objetos que serviu como ponto de partida para o desenvolvimento do software é mostrado a seguir (Figura 3.1). Utilizou-se a notação de COAD & YOURDON (1990) e a idéia de ART *frames* (FOLEY et al, 1989).

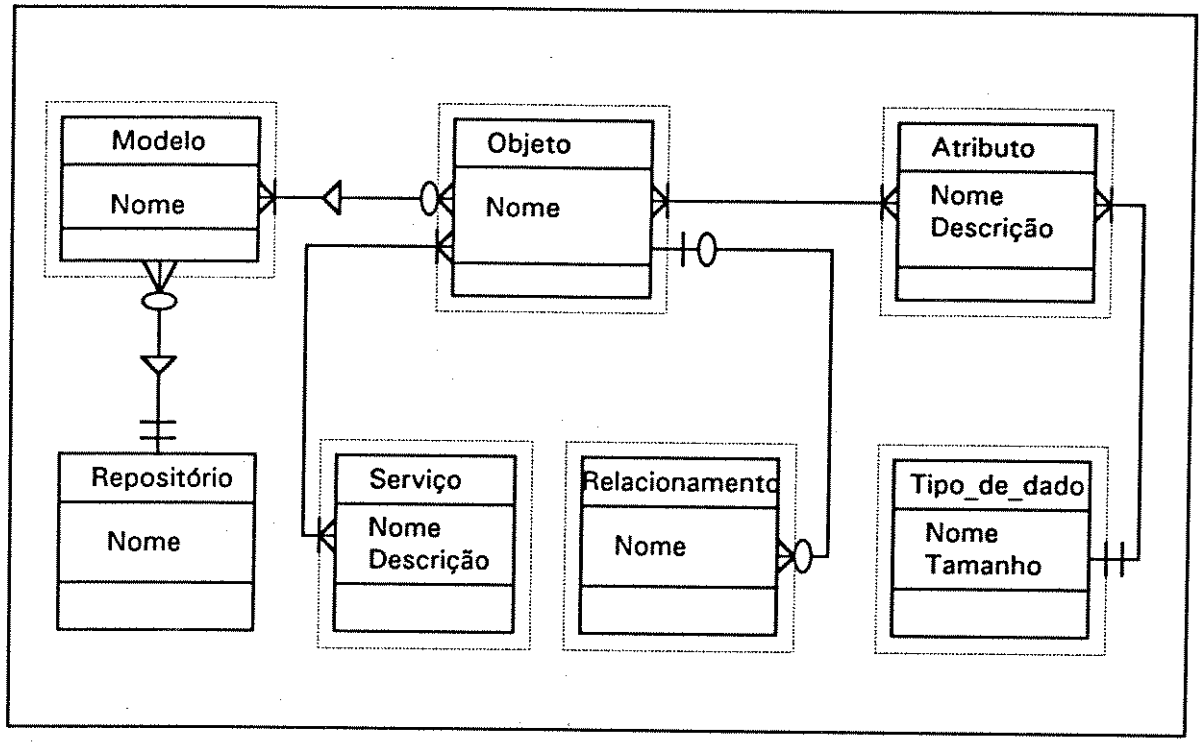
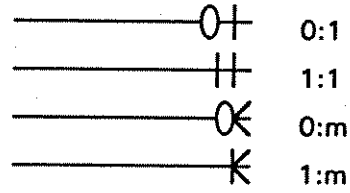


Figura 3.1 - Modelo de objetos.

A razão da escolha da notação de Coad & Yourdon deve-se à adequação ao problema em questão, além de ser simples e clara.

Todo objeto é representado por um retângulo dividido por duas linhas. No topo do retângulo anota-se a identidade do objeto, na parte intermediária anotam-se os atributos e na parte inferior os serviços. A linha pontilhada em torno de um objeto indica que o mesmo gera instâncias. Coad & Yourdon referem-se a este objeto como Classe-Objeto. A ausência da linha pontilhada significa que o objeto representa uma Classe, isto é, uma entidade abstrata não instanciada. Na Figura 3.1 nota-se que o modelo é composto pelas classes-objeto Modelo, Objeto, Serviço, Relacionamento, Atributo e Tipo de Dados, e pela classe Repositório.

As linhas de conexão e os símbolos terminais refletem o mapeamento entre as instâncias de objetos:



O triângulo sobre uma linha de conexão representa o relacionamento todo/partes ou coleção (conjunto, composição).

Instâncias de Atributo são descritas por instâncias de Tipo de Dados. Aqui cabe um comentário sobre a OOA de Coad & Yourdon. Eles não expressam tipo de dados. A ferramenta proposta no presente trabalho, porém, impõe que um atributo seja descrito por um tipo de dados.

Na notação de Coad & Yourdon, os serviços Inserir, Eliminar, Pesquisar, ou Alterar os nomes dos objetos, por serem considerados serviços essenciais, não precisam aparecer explicitamente no diagrama. Essa é a razão da parte inferior do retângulo estar em branco.

A estrutura de arquivos foi derivada diretamente do modelo de objetos. Para representar esta estrutura, utilizou-se o diagrama entidade-relacionamento (CHEN, 1990). Um trecho deste diagrama é mostrado na Figura 3.2.

Na linguagem de implementação, as entidades (representadas pelo quadrado) e os relacionamentos (representados pelo losângo) foram mapeados em arquivos de dados.

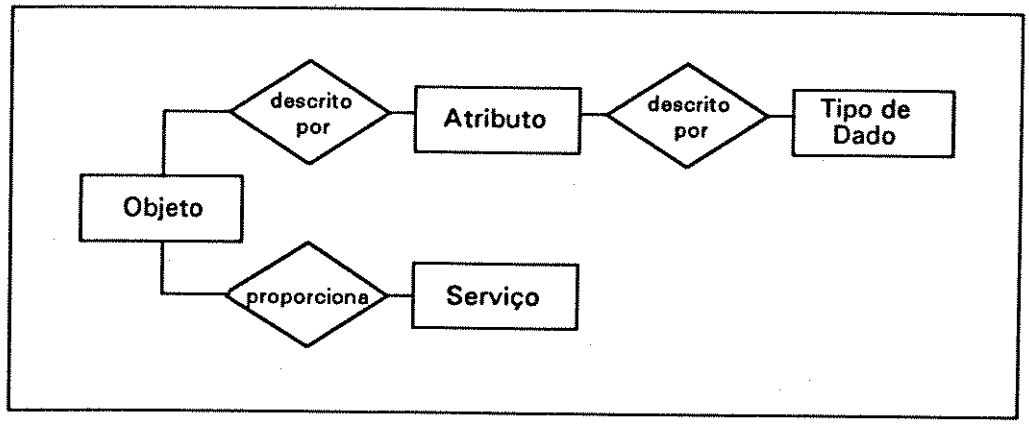


Figura 3.2 - Trecho do diagrama entidade-relacionamento.

3.3 Arquitetura do software

O software é formado por cinco componentes principais: Interface de Usuário, Controle, Validação, Base de Dados e Herança. A interação entre esses componentes é mostrada a seguir (Figura 3.3).

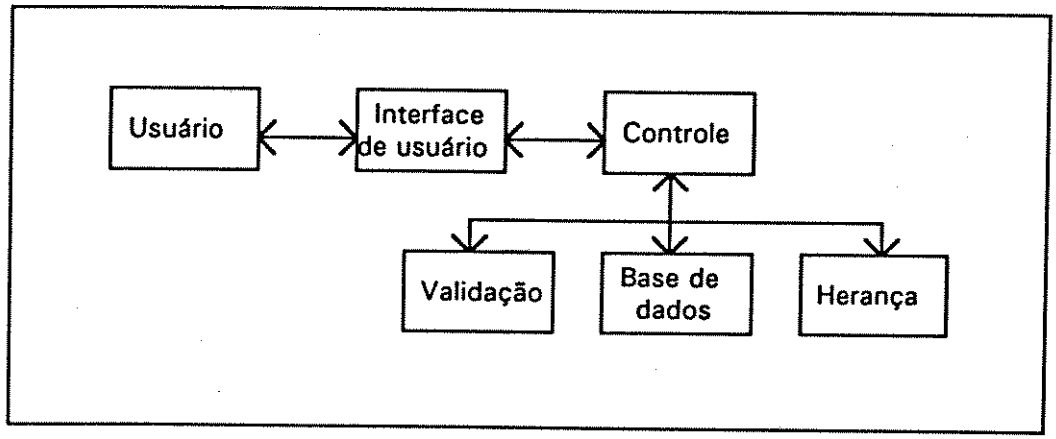


Figura 3.3 - Arquitetura do software

A *interface de usuário* é o componente responsável pelas atividades de diálogo com o usuário, tais como exibir menus, mensagens e *prompts*. A interface recebe as entradas do usuário via teclado e exibe no vídeo ou impressora o processamento realizado pelo software.

O componente de *controle* é responsável pela gerência da funcionalidade do software. Este componente "sente" a ação do usuário e a transmite para os demais componentes.

A *validação* verifica a entrada de dados do usuário, não permitindo a duplicação de nomes de objetos já existentes num mesmo modelo, ou a duplicação de nomes de atributos e serviços no contexto de um objeto.

O componente de *base de dados* realiza as operações usuais em arquivos de dados, tais como armazenar, modificar, classificar, recuperar e eliminar dados. Expressa também a cardinalidade entre os objetos do sistema. Na atual implementação, faz a verificação de atributos e serviços (para qual ou quais objetos eles foram definidos).

O componente de *herança* implementa o mecanismo de herança.

3.4 Informações técnicas

O software pode ser classificado como uma aplicação de base de dados pois baseia-se em operações típicas de base de dados. Foi desenvolvido num microcomputador compatível com o IBM-PC, processador 386 DX 40 Mhz, RAM 4 Mb, HD 120 Mb, monitor de vídeo VGA, e sistema operacional MS-DOS 6.2. Os relatórios impressos foram obtidos em sistema laser.

O código-fonte contém atualmente 3.300 linhas e o código-executável 497.687 bytes. Entre arquivos de sistema, de dados, de índice, arquivos auxiliares e arquivos de

ponteiros, tem-se um total de 45 arquivos. Todos esses arquivos podem ser manipulados durante uma sessão de trabalho.

Utilizou-se o Clipper 5.1 na implementação, um ambiente bem conhecido para o desenvolvimento de aplicativos em microcomputadores IBM-PC. É um sistema gerenciador de base de dados completo, tendo sido originado do interpretador dBase III ("data base"), um dos software mais comercializados na categoria de base de dados. Duas características do Clipper são indicadas para tratar o problema do presente trabalho. A primeira é o fato da linguagem ser eficiente na manipulação de dados discretos e textuais. Possui, por exemplo, um tipo MEMO que armazena dados caractere de tamanho variável, num limite de 65.535 caracteres. Esse aspecto é relevante. Pode-se escrever livremente dentro desse limite sem se preocupar com a administração de espaço fisico, pois a linguagem só armazena o texto efetivamente digitado.

A outra característica são as primitivas simples para programar interfaces de usuário. Um exemplo é o objeto TBROWSE, um mecanismo de uso geral para manipular dados na forma de tabelas. Esses objetos são de boa qualidade para a aquisição, formatação e exibição de dados em vídeo.

4 - Apresentação de resultados

Este capítulo é dividido em 4 seções. Na Seção 4.1 é feita uma exposição sobre a funcionalidade do software através de exemplos com alguns objetos. Na Seção 4.2 são discutidos os resultados obtidos. Na Seção 4.3 o sofware é comparado a outros sistemas de apoio à análise OO. Na Seção 4.4 são feitas as considerações finais.

4.1 Operação do software

O software pode ser executado a partir do *prompt* do sistema operacional, digitando-se OBJ e pressionando em seguida <ENTER>. Inicialmente será solicitada a data e exibida a tela de trabalho (Figura 4.1).

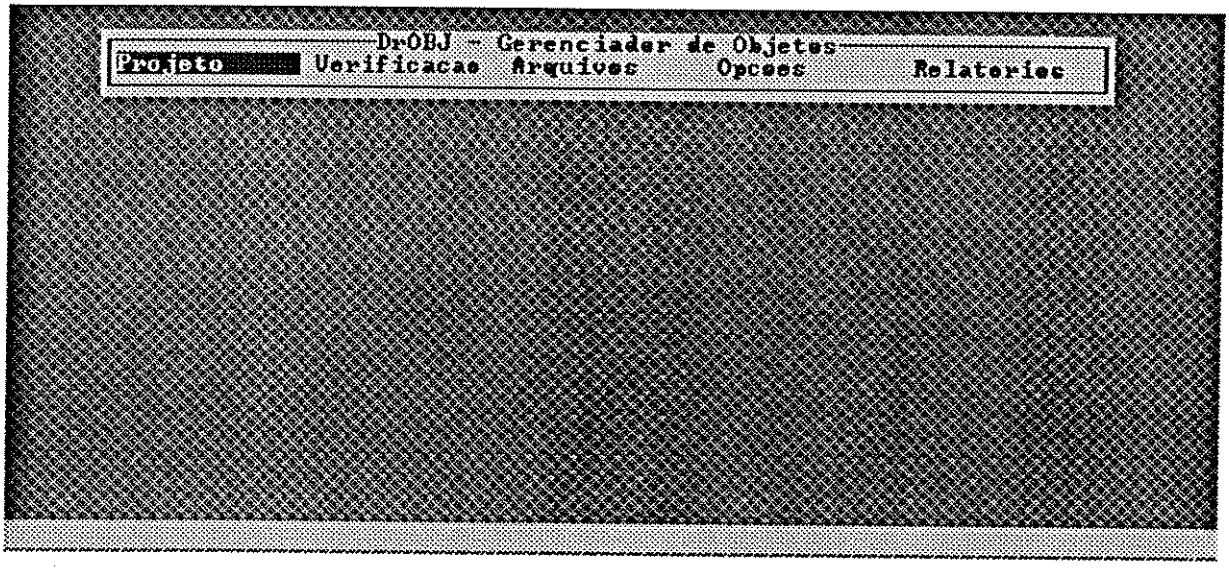


Figura 4.1 - Tela de trabalho.

A tela é dividida em três áreas: área de menu, localizada no topo da tela; área de janelas, que ocupa toda a região central da tela; e uma linha de comandos na parte inferior. Esta linha mostra os comandos que podem ser acionados num dado instante.

O menu principal fica sempre visível. As janelas e os comandos são exibidos após o início de uma sessão, quando opta-se por um dos itens do menu principal.

As opções do menu principal são: *Projeto*, *Verificação*, *Arquivos*, *Opções* e *Relatórios*. *Projeto* permite ao usuário iniciar um novo modelo (ou aplicação) ou abrir um já existente. *Verificação* permite obter o relacionamento inverso entre atributos e serviços armazenados na base de dados. *Arquivos* permite realizar operações com os arquivos de dados principais do sistema. *Opções* realiza uma tarefa de configuração interna. *Relatórios* permite obter resultados impressos.

Para selecionar um item de menu basta pressionar a letra inicial, ou posicionar o cursor sobre o mesmo e pressionar <ENTER>.

A seguir serão apresentados mais detalhes sobre a operação do software.

Opção *Projeto*

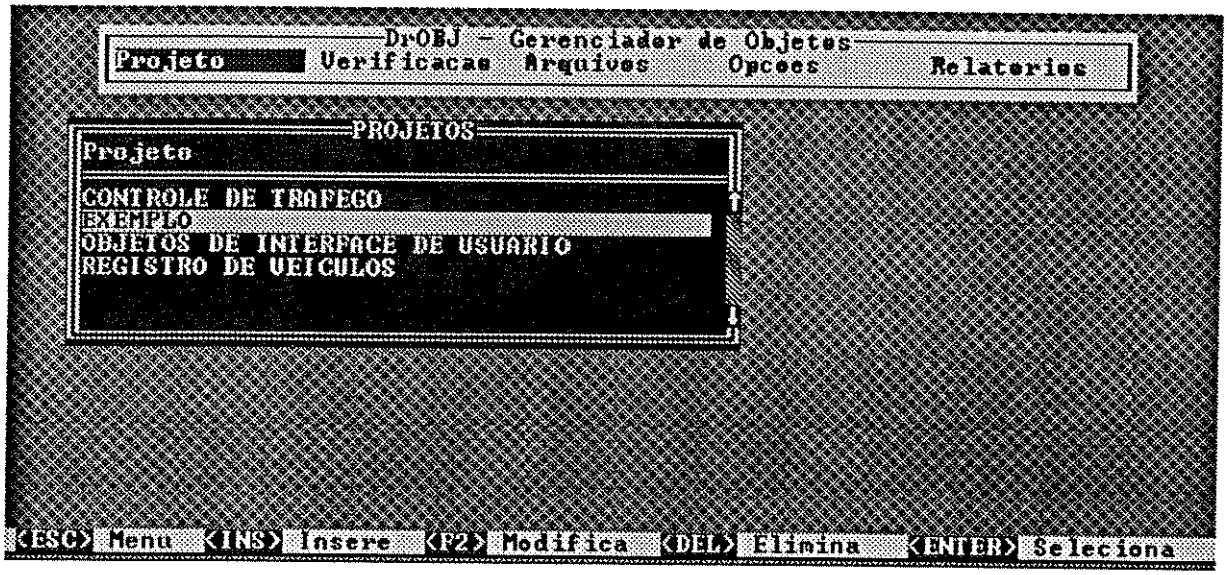


Figura 4.2 - Janela de projetos.

A opção PROJETOS concentra a maior parte da funcionalidade do software. Ao selecionar *Projeto* serão exibidos a janela de projetos e os comandos possíveis de serem acionados (Figura 4.2).

Quatro projetos fictícios foram incluidos para compor o cenário de ilustração: Controle de Tráfego, Exemplo, Objetos de Interface de Usuário e Registro de Veículos. Os projetos são apresentados em ordem alfabética. A escolha de um item na janela é similar à escolha de um item de menu.

As operações de retorno ao menu principal, inserção, modificação, eliminação e seleção de projetos são feitas respectivamente pelas teclas <ESC>, <INS>, <F2>, <DEL> e <ENTER>, como indica a linha de comandos na Figura 4.2.

A seguir serão ilustradas algumas operações no projeto EXEMPLO. Pressionando-se <ENTER> com o cursor sobre este item, será aberta a janela de OBJETOS associados ao projeto EXEMPLO (Figura 4.3).

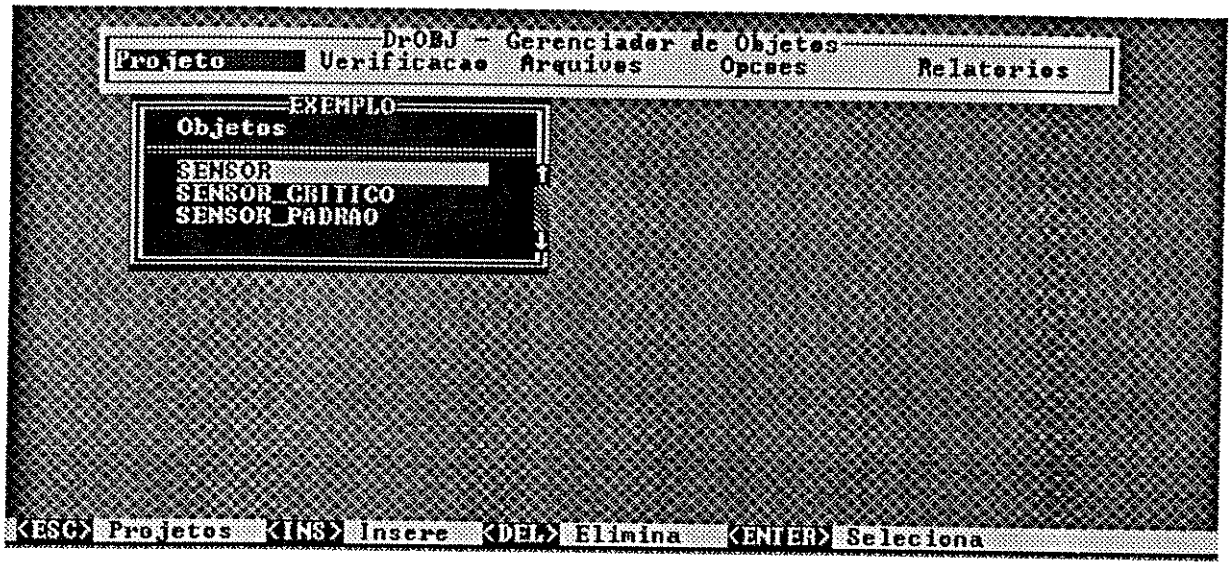


Figura 4.3 - Janela de objetos associados ao projeto EXEMPLO.

A janela na Figura 4.3 mostra a existência de três objetos: SENSOR, SENSOR_CRÍTICO e SENSOR_PADRÃO.

Suponha que se deseja inserir em EXEMPLO um novo objeto. Para inserir um objeto, basta pressionar <INS>. Uma nova janela é exibida, mostrando alguns objetos já definidos anteriormente e que podem ser usados no projeto em questão (Figura 4.4).

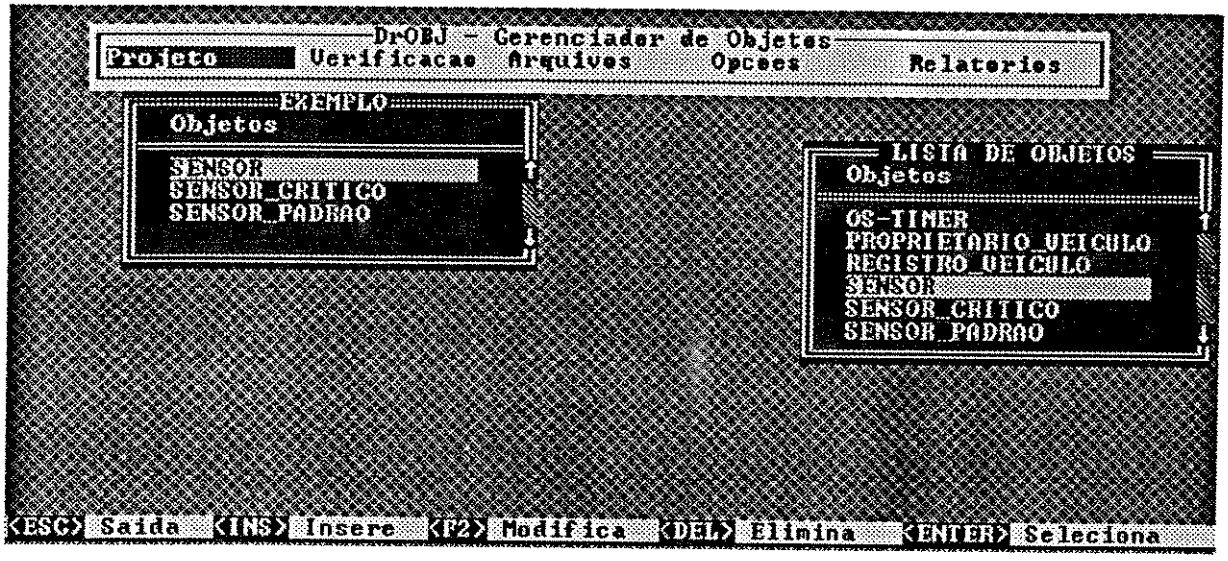


Figura 4.4 - Janela de objetos do projeto atual (à esquerda) e lista de objetos já existentes (à direita).

Se o objeto desejado não existir, pode-se criá-lo através de outro <INS>. Um novo objeto é criado pelo usuário e fica automaticamente disponível para outros projetos. A tentativa de inserir um objeto já existente (por exemplo, o objeto SENSOR) é logo barrada, como pode ser notado na Figura 4.5.

A especificação de atributos e serviços, e a definição de relacionamentos entre objetos são feitas posicionando-se o cursor num objeto e pressionando-se <ENTER>. Um menu é sobreposto à janela de objetos, aguardando a decisão do usuário (Figura 4.6).

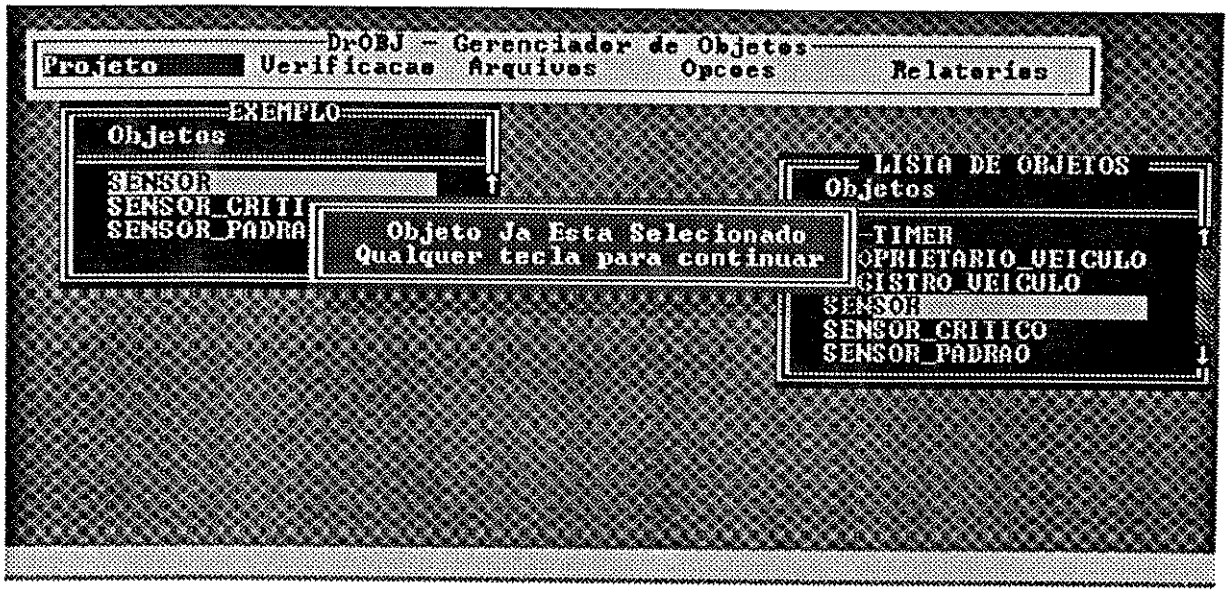


Figura 4.5 - O software não permite inserir um objeto já existente.

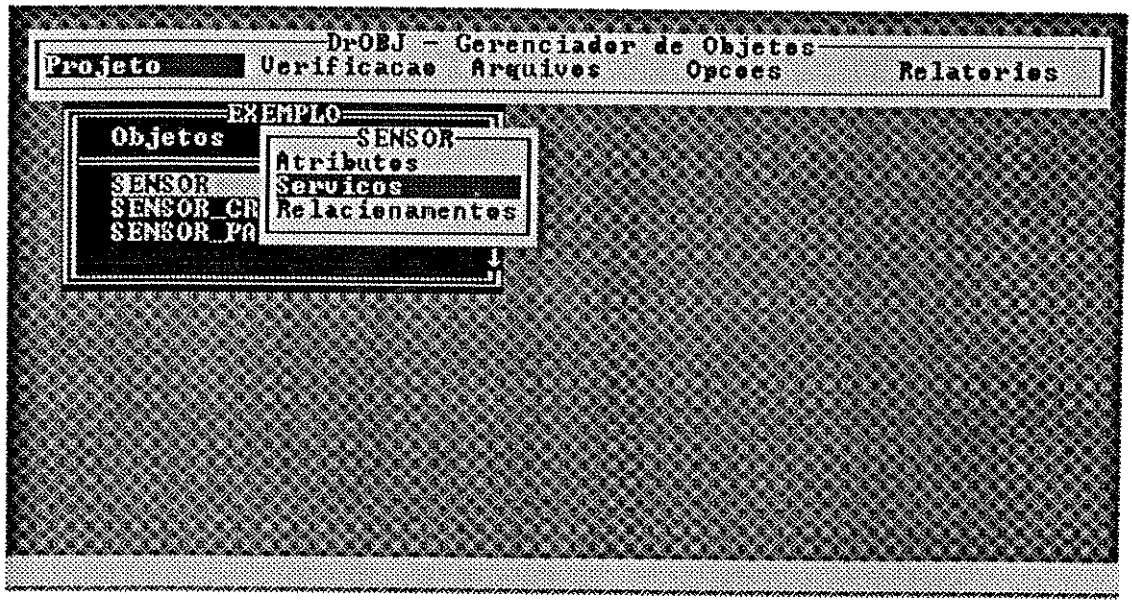


Figura 4.6 - Menu para definir atributos, serviços e relacionamentos.

No exemplo da Figura 4.6, selecionou-se o objeto SENSOR e o item SERVIÇOS. Uma nova janela é exibida, mostrando os serviços já existentes. Neste caso só há um serviço: MONITOR (Figura 4.7).

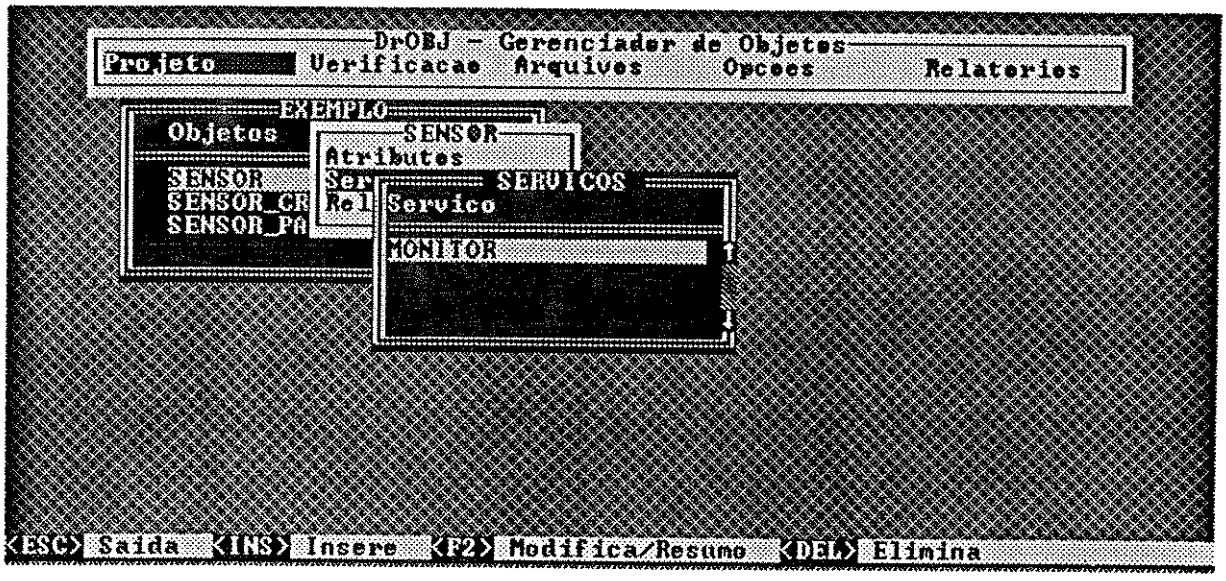


Figura 4.7 - Janela de serviços associados ao objeto selecionado.

A inserção de novos serviços é similar à inserção de objetos descrita anteriormente, isto é, pressionar <INS>, consultar na nova janela os serviços existentes; se o serviço desejado não existir, criar o serviço com outro <INS>.

Para especificar um serviço ou alterar uma especificação existente, basta posicionar o cursor sobre o serviço desejado e pressionar <F2>. Uma janela de edição de texto é exibida na tela com espaços apropriados para descrever algumas características de um serviço, tais como *resumo, parâmetros, algoritmo* e *anotações gerais* (Figura 4.8).

A Figura 4.8 mostra um trecho da especificação do serviço MONITOR. Na janela de edição de texto pode-se escrever livremente dentro dos limites de cada campo: *resumo* (150 bytes), *parâmetros* (60 bytes), *algoritmo* (65 635 bytes) e *obs* (65 635 bytes). Todas as informações nesta tela ficam armazenadas na base de dados.

A especificação de atributos é similar à especificação de serviços, isto é, para cada atributo pode-se ativar uma janela de edição de texto e descrever algumas características de um atributo, tais como *resumo, valor default* e *tipo de dados*.

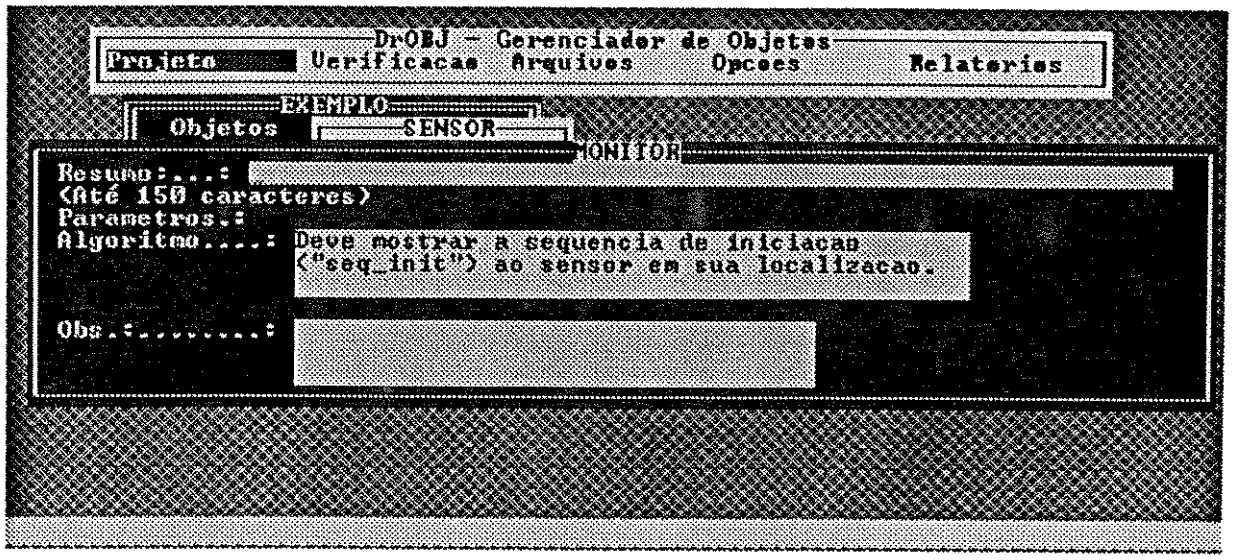


Figura 4.8 - Janela de especificação de serviços.

A definição de relacionamentos entre objetos é ilustrada a seguir. Ao escolher Relacionamentos (Figura 4.6), a janela de objetos é exibida na tela, permitindo que se escolha o segundo objeto participante do relacionamento (Figura 4.9).

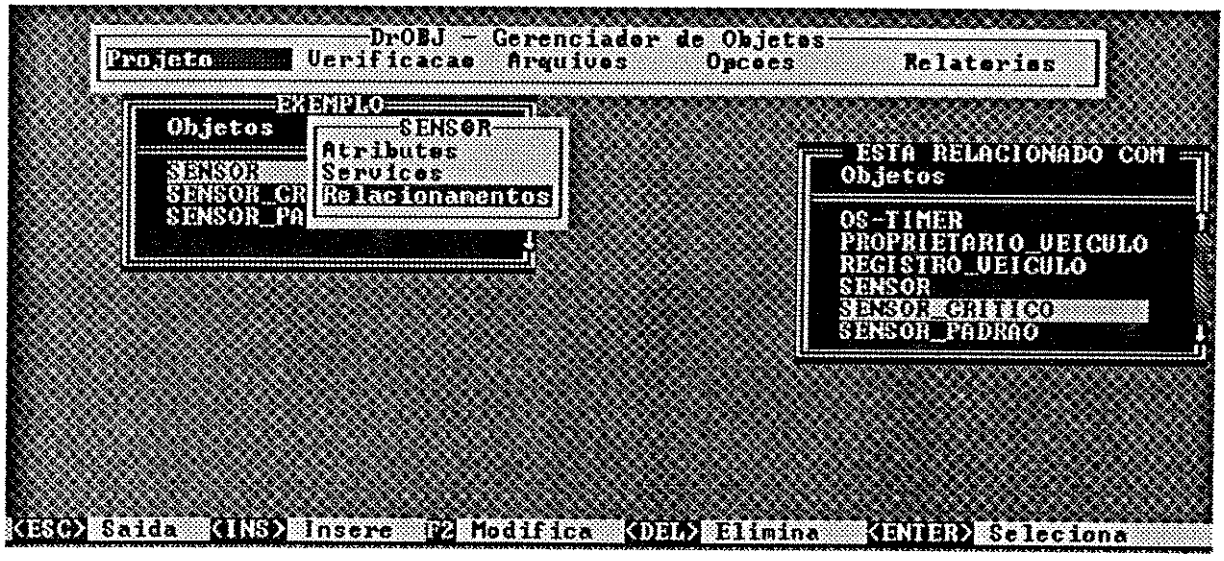


Figura 4.9 - A janela de objetos (à direita) permite escolher o objeto com o qual o objeto selecionado anteriormente (SENSOR) irá se relacionar.

Como pode ser notado na Figura 4.9, deseja-se relacionar o objeto em questão SENSOR com outro objeto, SENSOR_CRÍTICO. Pressionando-se <ENTER> com o cursor posicionado em SENSOR_CRÍTICO, abre-se uma nova janela para que se escolha o relacionamento desejado (Figura 4.10).

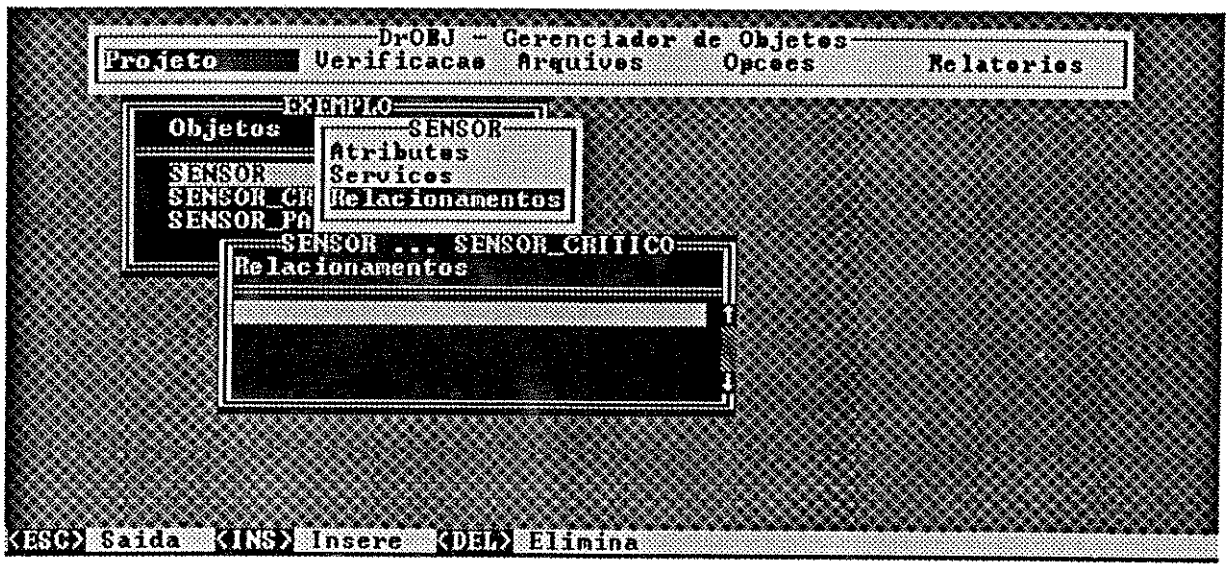


Figura 4.10 - Janela de relacionamentos entre objetos.

Na Figura 4.10 nota-se que a janela de relacionamentos está vazia. Isso significa que nenhum relacionamento foi ainda definido entre SENSOR e SENSOR_CRÍTICO. Para definir um relacionamento basta pressionar <INS>, como indica a linha de comandos, operação esta similar à já vista anteriormente. Ao pressionar <INS> abre-se uma janela com uma lista de relacionamentos previamente armazenados (Figura 4.11). Cada relacionamento escolhido passa a figurar na janela de relacionamentos. É importante destacar que os objetos podem relacionar entre si de diversas formas.

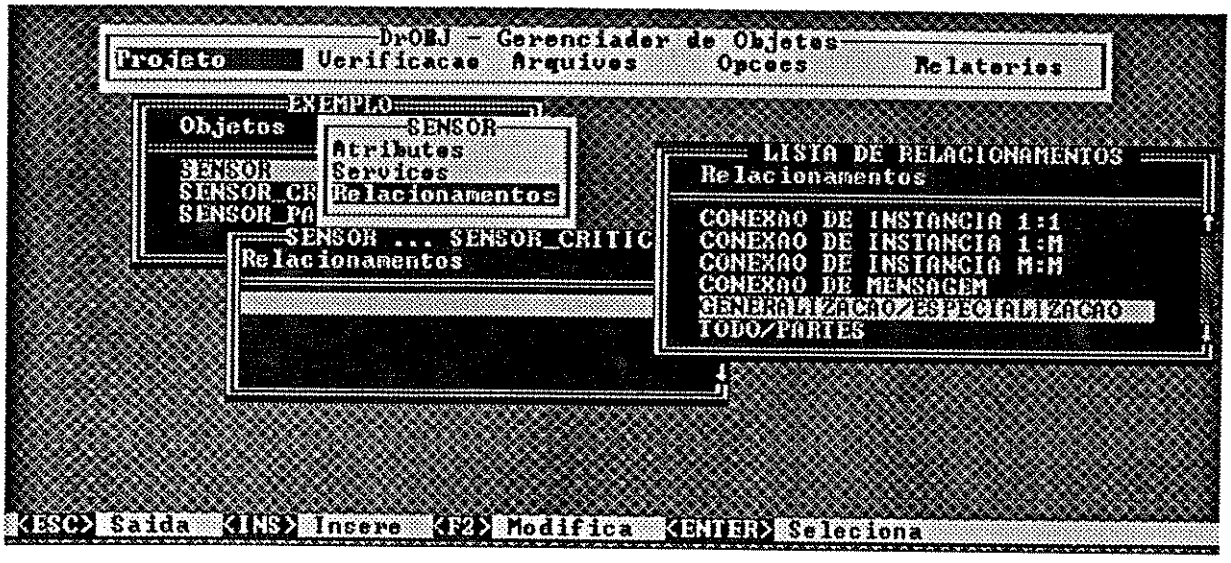


Figura 4.11 - Janela de relacionamentos (à direita).

Nesta implementação, quando escolhe-se o relacionamento Generalização / Especialização, automaticamente todos os atributos e serviços do primeiro objeto (SENSOR) são copiados para o segundo objeto (SENSOR_CRÍTICO), incluindo a especificação de cada propriedade.

Opção *Verificação*

A opção *Verificação* fornece uma informação sobre o relacionamento inverso de atributos e serviços. Dada uma propriedade (atributo ou serviço), esta opção mostra para quais objetos a propriedade foi definida.

A seguir será ilustrada a operação no menu Verificação / Serviço. A operação no menu Verificação / Atributos é similar. Ao optar por Verificação e em seguida por Serviço, a janela de serviços é exibida na tela (Figura 4.12).

Como pode ser notado na Figura 4.12, a partir do menu Verificação/Serviços escolheu-se o serviço CALC_SEGURO.

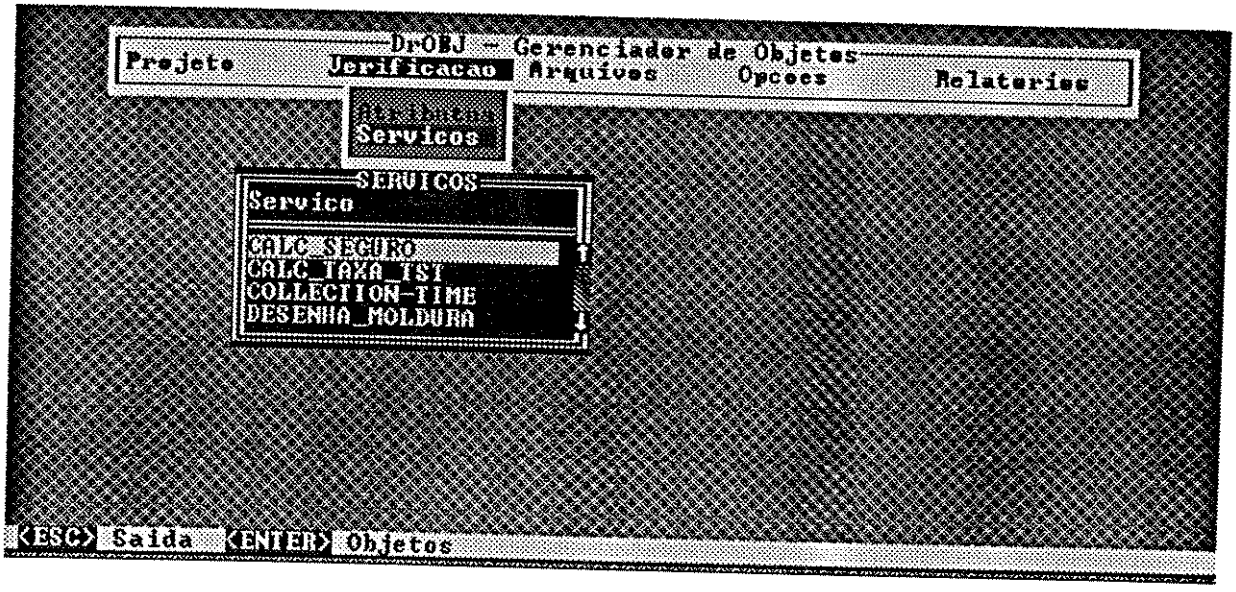


Figura 4.12 - Janela de serviços. O posicionameno do cursor em CALC_SEGURO mostra a intenção de verificar para quais objetos este serviço foi definido.

É importante destacar que a janela de serviços mostra todos os serviços armazenados na base de dados. Esta opção faz uma verificação em todos os modelos existentes. Ao pressionar <ENTER> são exibidos os objetos que apresentam o serviço selecionado. Este resultado é mostrado na Figura 4.13.

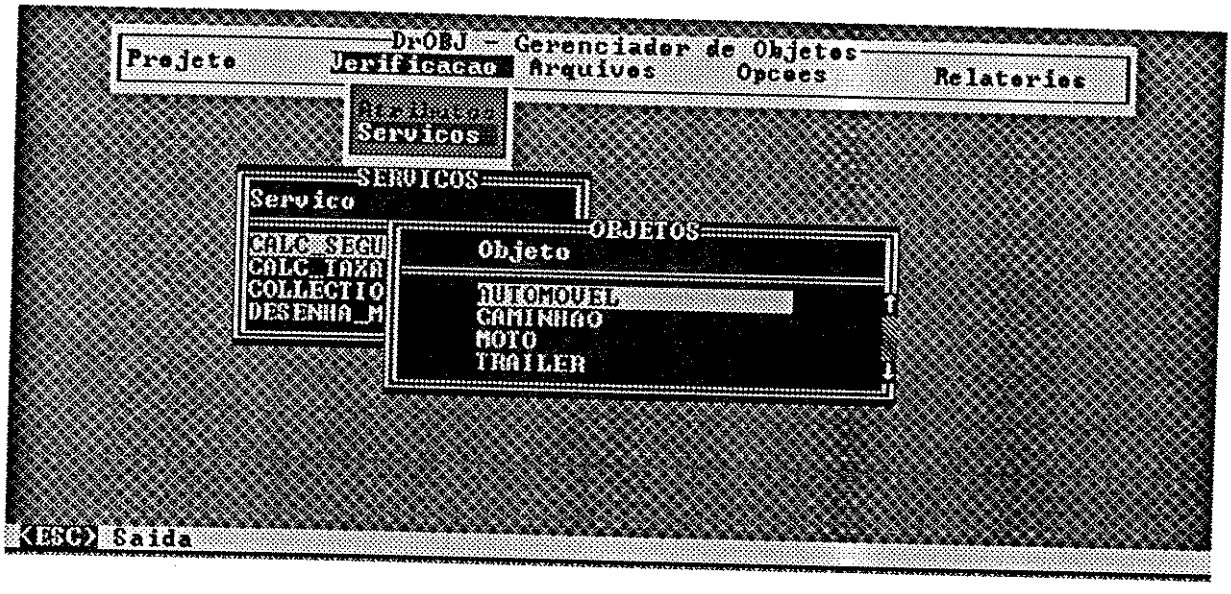


Figura 4.13 - Lista de objetos que apresentam o serviço CALC_SEGURO.

Opção *Arquivos*

A opção *Arquivos* permite operar diretamente com os arquivos de dados principais do software: *objetos*, *atributos*, *serviços*, *relacionamentos* e *tipo de dados*. A operação nesses arquivos independe da abertura de um projeto em particular. O objetivo é permitir a inserção de dados para posterior aproveitamento, sem estar necessariamente operando em um projeto.

A seguir será ilustrada a operação no menu Arquivos/Tipo de Dados. A operação nos demais itens é similar. Ao optar por Arquivos e em seguida por Tipo de Dados, é exibida a janela com diversos tipos primitivos já armazenados (Figura 4.14).

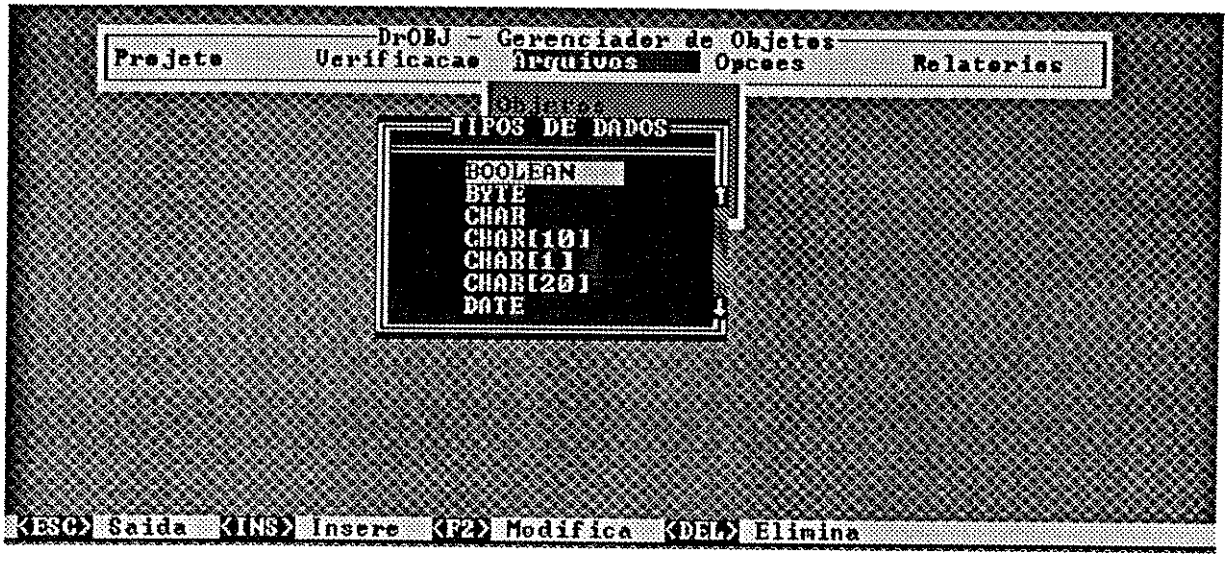


Figura 4.14 - Janela de tipo de dados.

Os tipos de dados armazenados pelo software ficam sempre disponíveis para quaisquer modelos.

O comando Modifica pode ser executado normalmente. Se um dado for modificado, essa modificação irá refletir em todos os objetos sem problemas. O comando Elimina, porém, transmite um sinal de advertência ao usuário, pois um dado

pode estar associado a vários objetos. Se o usuário desejar, o comando elimina o dado em todos os objetos onde ele foi definido.

Opção *Opções*

Como já foi declarado no início deste capítulo, esta opção realiza uma configuração nos arquivos de índice do software, não havendo detalhes importantes que mereçam destaque.

Opção *Relatórios*

Esta opção permite obter relatórios impressos. Dois tipos de relatórios estão disponíveis: o relatório de *todos* os projetos ou o relatório de *um* projeto.

A seguir será ilustrada a operação no menu Relatórios / Seleciona. A operação no menu Relatório / Todos os Projetos é similar. Ao optar por Relatório e em seguida por Seleciona, é exibida a janela de Projetos (Figura 4.15).

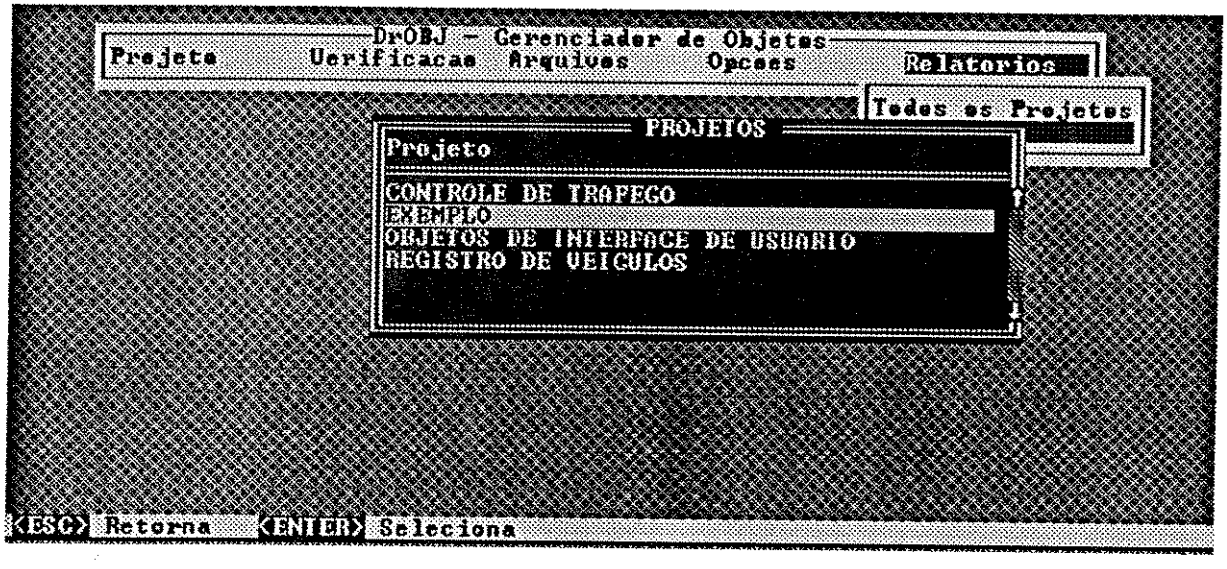


Figura 4.15 - Janela de projetos.

Como pode ser notado na Figura 4.15, escolheu-se o projeto EXEMPLO para imprimir o relatório. Um trecho do relatório é mostrado a seguir (Figura 4.16).

Pagina: 001
Data: 29/11/94

Listagem de Projetos

Nome do Projeto: EXEMPLO

OBJETO: SENSOR
ATRIBUTO: MODELO
Resumo: marca e modelo do sensor.
Default:
Tipo de dados: char[20]
Obs:

SEQ_INIC
Resumo: a sequencia de iniciacao para este tipo de sensor.
Default:
Tipo de dados: char[20]
Obs:

CONVERSAO
Resumo: consiste do fator de escala, direcao e unidade de medida.
Default:
Tipo de dados: real
Obs:

INTERVALO
Resumo: intervalo de amostra para o sensor.
Default:
Tipo de dados: real
Obs:

THRESHOLD
Resumo: o valor de threshold
Default:
Tipo de dados: real
Obs:

ESTADO

Resumo: estado operacional do sensor
Default:
Tipo de dados: char[10]
Obs: os estados possiveis:
"on";
"standby"
"off"

TOLERANCIA
Resumo: tolerancia de amostragem de intervalo (para sensores criticos).
Default:
Tipo de dados: real
Obs:

SERVICO: MONITOR
Resumo:
Parametros:
Algoritmo: deve mostrar a sequencia de iniciacao ("seq_init") ao sensor em sua localizacao.

repetir com os valores dos atributos:

--> amostrar o sensor em sua localizacao, no intervalo correspondente.

--> converter a leitura do sensor com a equacao

valor = (direcao + leitura do sensor) * offset

--> se o valor de "threshold" for encontrado ou excedido, retornar um alarme.

==> (estender para sensor critico)

amostrar o sensor em sua localizacao, em intervalos correpondentes, dentro de uma "tolerancia".

--> a partir de uma alteracao na sequencia de iniciacao ("init_seq"), mostrar a sequencia de iniciacao ao ...

Obs:

RELACIONAMENTO: <generalizacao/especializacao>... SENSOR_CRITICO
<generalizacao/especializacao>... SENSOR_PADRAO

Figura 4.16 - Trecho do relatório do projeto EXEMPLO.

O relatório completo mostra o contexto do projeto: a lista de objetos, a especificação dos atributos e dos serviços de cada objeto, e o relacionamento existente entre os objetos.

4.2 Discussão

O protótipo, cuja operação foi ilustrada na seção anterior, atendeu em grande parte aos requisitos pré-estabelecidos.

A operação mostrou ser simples e fácil. A interface de janelas teve um bom efeito. A busca em cada janela na forma de *browser* foi simplificada pela digitação da letra inicial de cada elemento, o que trouxe mais rapidez ao processo de localização.

A possibilidade de trabalhar separadamente com objetos, atributos, serviços, relacionamentos e tipos de dados, mais a possibilidade de criar esses elementos num modelo qualquer e depois usá-los em outros modelos, apresentou resultados satisfatórios. Pôde-se incluir antecipadamente uma lista de objetos primitivos ou mais comuns e depois escolher aqueles que fizeram parte da ilustração.

O relacionamento inverso de atributos e serviços foi outro resultado positivo desta implementação. O principal benefício que pode advir refere-se ao refinamento de serviços. Os serviços que aparecem em muitos objetos podem ser escolhidos como candidatos à otimização do algoritmo.

Outro ponto relevante observado foi a implementação de herança, que trouxe flexibilidade no que tange ao reuso de especificações de atributos e serviços. Uma vez transmitidas as especificações dos atributos e serviços de uma classe geral para as classes especializadas, pôde-se alterar livremente cada especificação para atender as características particulares de cada classe especializada. Um problema notado foi o tamanho da especificação resultante, que pode levar à redundância de documentação.

Em se tratando de operações em base de dados, realizou-se um teste com o software para verificar alguns limites quanto ao armazenamento externo. Duzentos objetos foram introduzidos em um projeto fictício e a cada objeto foram alocados 10 atributos (incluindo a especificação de cada atributo) e 10 serviços (incluindo a especificação de cada serviço). À medida em que os objetos eram introduzidos, não se notaram problemas operacionais como, por exemplo, tempo excessivo para realizar o armazenamento ou a recuperação de informações.

Cabe comentar dois aspectos associados aos requisitos do software: o uso dos dados por outros sistemas e a migração para outros equipamentos. Qualquer sistema que interprete o formato de dados armazenado pelo Clipper (formato DBF) como, por exemplo, Access e FoxPro, pode fazer uso da base de dados sem custo adicional, independentemente do software aqui apresentado. O outro aspecto é a migração. A conversão para o Unix, por exemplo, é direta, através de um compilador/tradutor denominado FlagShip. Basta copiar todos os arquivos (de programa, de dados, de índice, etc.) para uma máquina Unix e rodar o FlagShip.

A migração e o uso da base de dados por outros sistemas não foram testados nesta implementação, porém testes realizados em sistemas de porte equivalente mostraram resultados positivos.

4.3 Comparação com outras ferramentas

Foi feita uma tentativa de examinar versões *demo* de algumas ferramentas citadas no Capítulo 2, Seção 2.3.6 CASE. Por não poder contar com essas versões dentro de um prazo conveniente e devido à indisponibilidade de plataforma apropriada para instalar e rodar cada software, foram examinadas somente duas ferramentas: OOTher e MetaEdit.

Comparada à OOTher e MetaEdit, o protótipo, que será referido nesta seção por OBJ, apresenta pontos comuns e diferenças. Todas têm por objetivo apoiar a análise OO. OBJ trabalha com a descrição textual, enquanto as outras oferecem recursos

adicionais, como a notação gráfica. OBJ é independente de algum método de análise OO. OOTher oferece uma combinação de desenhos: diagrama de objetos de Coad & Yourdon; *use case* e diagrama de interação de Jacobson; e diagrama de estados. MetaEdit permite escolher a notação de um método específico dentre uma série de métodos.

Uma diferença significativa é a interface de usuário e o ambiente operacional. MetaEdit e OOTher são sistemas para o Windows. OBJ roda no DOS, com interface de caracteres.

OOTher e OBJ fazem uma verificação no ato da entrada de dados, não deixando repetir nomes de atributos e serviços num mesmo objeto. MetaEdit não oferece este recurso, apenas não deixa repetir nomes de objetos.

A introdução de objetos num modelo, bem como a alocação de atributos e serviços aos objetos, é mais ágil na OBJ do que nas outras. Supondo que a OBJ tenha sido usada em outros modelos, é provável que a quantidade de nomes de objetos armazenados seja grande. Isso implica na redução da possibilidade de reescrever nomes. É provável que os nomes de objetos desejados num novo modelo já estejam armazenados, portanto basta pegá-los e inserí-los no modelo.

As três ferramentas apresentam relatórios na forma de texto estruturado, com variados níveis de informação sobre os modelos. Um exemplo é o histórico das propriedades herdadas pelos objetos. Ainda que elas não dêem um suporte ideal para um método específico, suporte no sentido de forçar a aplicação do método, o relatório é um bom ponto de partida para qualquer método.

4.4 Considerações finais

Procurou-se oferecer uma ferramenta que reunisse algumas características de apoio à análise OO. Dentre elas destaca-se o mecanismo de herança, uma das principais, senão a principal, característica da OO. Em geral, as linguagens de

programação OO dão suporte à herança. Atualmente nota-se que a herança também deve se expressar na análise.

A herança está diretamente associada à classificação que é, em geral, arbitrária. Atributos e serviços definidos nas classes de mais alto nível, dificilmente são compartilhados sem modificações pelas subclasses. É mais provável que, à medida em que se caminha em direção às classes especializadas, haja alteração nas propriedades herdadas, inclusão de novas propriedades e exclusão das não desejadas.

Uma ferramenta flexível para lidar com a herança foi um dos principais aspectos que motivou a concepção deste protótipo. A forma como ela foi implementada encontra-se, porém, num nível experimental.

Buscou-se ainda neste protótipo uma forma de apoiar a análise no sentido de armazenar previamente um conjunto de nomes de objetos, atributos, serviços, relacionamentos e tipo de dados considerados primitivos, que tendem a aparecer com mais freqüência, e deixá-los sempre disponíveis. Desta maneira, a atividade principal fica mais concentrada em localizar e encaixar as peças que irão satisfazer as peculiaridades de um modelo.

---

5 - Conclusão e perspectivas

Procurando focalizar a situação atual da OO e colher subsídios para desenvolver uma ferramenta de apoio ao processo da análise OO, selecionou-se extenso material bibliográfico referente a algumas áreas de interesse como conceitos básicos, linguagens de programação, interface de usuário, sistema operacional, base de dados, *framework*, CASE e métodos de análise/design.

A importância de conceitos básicos está em estabelecer um vocabulário comum para termos e conceitos considerados centrais em diferentes domínios de aplicação, sendo este um ponto importante e amplamente reconhecido na literatura.

As linguagens de programação, principalmente Smalltalk e C++, estão se popularizando. Com essas linguagens pode-se praticar a OO de duas formas: através de um enfoque puro, onde o ambiente da linguagem proporciona um suporte direto a todos os benefícios da OO; ou através de uma abordagem híbrida, que permite aos programadores com habilidade no método procedimental transitar gradualmente para a OO.

A importância de sistema operacional OO decorre do fato de que o código pode crescer e mudar segundo as exigências de mercado. Esse é um aspecto bastante enfatizado: o enfoque de objetos permite a construção de sistemas flexíveis, diante dos quais a manutenção fica mais fácil e eficiente. O mesmo pode-se dizer das interfaces de usuários.

Base de dados OO (OODB) estende as capacidades das linguagens de programação, tais como classe e herança, principalmente para a persistência de objetos. Em vários sistemas, após uma sessão de trabalho, as instâncias de objetos e seus relacionamentos deixam de existir. Nas OODBs elas ficam armazenadas. Outra característica de interesse é que as OODBs permitem aplicações distribuídas.

Framework é citado na literatura como uma área que se tornará indispensável na maioria das aplicações, mas que ainda demanda pesquisa. Os relacionamentos de objetos mais comuns, generalização e agregação, não conseguem expressar o grau de relacionamento que as classes pertencentes a um *framework* apresentam.

CASE, especialmente para a análise, é uma necessidade porque esta metodologia gera um vocabulário diferente e uma rede de relacionamentos maior do que em outras metodologias. Diversos produtos estão disponíveis no mercado comercial a preços que variam de US$ 100,00 a US$ 25 000,00, nas mais diversas plataformas. Em geral, as ferramentas dão suporte aos métodos mais conhecidos, combinam a notação gráfica com a notação textual e geram código.

Os métodos de análise e design estão evoluindo. É possível que outros apareçam e procurem combinar os pontos fortes dos já existentes, propondo uma base comum para o desenvolvimento orientado a objeto. Conceitos básicos, identificação das propriedades estáticas e dinâmicas, análise e design interativos, CASE, ênfase em reuso (principalmente através de *framework* para reutilização em nível corporativo) e prototipação são alguns dos pontos comuns entre os métodos. Notação, apoio total ou parcial ao ciclo de vida, elaboração de modelos e conceitos exclusivos são algumas das diferenças entre os métodos.

Ainda não existe um método que seja aceito amplamente, porém existem indícios de que *use case* (JACOBSON et al, 1992) caminha no sentido de tornar-se um padrão industrial. O que mais se nota atualmente é uma disputa de mercado.

A revisão de literatura apresentada neste trabalho permitiu colher subsídios para construir um protótipo dedicado a alguns pontos de interesse, tais como a descrição estática dos objetos, a verificação de entrada de dados, o relacionamento inverso de atributos e serviços, herança, e relatório estruturado. O protótipo possibilita o exercício da análise orientada a objeto, sem estar vinculado a algum método. O relatório é um bom ponto de partida para ser usado por qualquer método já conhecido.

Procurou-se verificar como a *herança* era abordada nos métodos e em algumas ferramentas CASE. A conclusão é que a herança, cujo poder de expressão já está incorporado nas linguagens de programação, também deve ocorrer na análise. Foi feita uma tentativa de implementar, em nível experimental, esse mecanismo. Deve-se tentar

outras formas de trabalhar com a herança, pois este é um dos pontos mais importantes da orientação a objeto.

O protótipo, entretanto, apresenta algumas limitações. A versão é monousuário, feita para rodar somente sob o DOS e grava dados no formato DBF, quando poderia seguir o código ASCII.

Como perspectivas de continuidade, algumas sugestões são apontadas a seguir.

É importante obter informações sobre o modelo como, por exemplo, a quantidade de propriedades (atributos e serviços) que foi definida para cada classe. Classes gerais com baixa quantidade de propriedades tendem a gerar classes especializadas com o mesmo conjunto de propriedades, sem cancelamentos. O problema é descobrir a quantidade ideal de propriedades que uma classe deve apresentar!

A herança pode ser aprimorada no sentido de permitir ao usuário decidir quais propriedades serão transmitidas de uma classe geral para uma classe especializada.

Outra sugestão é elaborar um quadro geral de verificação. Um exemplo é verificar a existência de classes sem alguma propriedade, ou a existência de alguma propriedade sem especificação. COAD & YOURDON (1990) sugerem uma lista interessante de itens.

Um ponto relevante é estender o software para um ambiente multiusuário e implementar outras formas de localizar objetos no repositório. Nesta versão, o protótipo é monousuário e a localização de objetos pelo nome é limitada e insuficiente. Pode-se necessitar de um objeto e não se lembrar do nome. Um outro método deve ser implantado para permitir a localização segundo diversos critérios como, por exemplo, através de tipos de objetos e serviços.

Em um estudo referente a tipologia de objetos, MONARCHI & PUHR (1992) classificam os objetos em quatro categorias: *semânticos*, *interface*, *aplicação* e *base*. Objetos semânticos são aqueles que descrevem ou fazem parte do domínio do problema como, por exemplo, eventos ou ocorrências, localizações, objetos tangíveis, funções exercidas por pessoas etc.. Os demais são objetos que ajudam a encontrar a

solução do problema. Objetos *interface* mostram a visão do usuário dos objetos semânticos, objetos *aplicação* (não têm estado que persiste no tempo, só apresentam comportamento) podem ser os mecanismos de controle como, por exemplo, iniciar uma passagem de mensagem a outros objetos, e *base* são os objetos mais primitivos tais como array, string etc..

Um outro critério de localização é através dos serviços proporcionados pelos objetos. Também nesse caso deve-se encontrar referências que se dedicam ao exame da classificação de serviços. Um ponto que merece reflexão é tentar incorporar um método de identificação inteligente que permita o acesso a determinado objeto através da descrição das características mais significativas dos serviços dos objetos.

O acesso por palavras-chave ou por uma expressão que faça parte do nome do objeto são também outros critérios que podem ser úteis para localizar objetos.

Um repositório de objetos em ambiente multiusuário, com um modelo completo de verificação de erros e omissões e vários critérios de identificação de objetos pode ser útil principalmente se se pensar em nível corporativo. Não é toda ferramenta que apresenta essas características. Se desejar implementar a reutilização ao nível corporativo, deve existir um CASE para administrar objetos corporativos, evitando alternativas limitadas do tipo "importar/exportar", que parece ser a mais comum nas ferramentas atuais.

Referências

BAILIN, S. C. An object-oriented requirements specification method. *Communications of the ACM*, v. 32, n. 5, p. 608-623, May. 1989.

BARANAUSKAS, M. C. C. Criação de ferramentas para o ambiente PROLOG e o acesso de novatos ao paradigma da programação em lógica. Campinas, 1993. 355p. Tese (Doutorado). Faculdade de Engenharia Elétrica / UNICAMP.

BEARD, M. NeXTstep 3.1 para Intel. *Personal Computer World*, v.7, n. 81, p. 38-40. Set. 1993.

BECK, K.; CUNNINGHAM, W. A laboratory for teaching object-oriented thinking. In: Proceedings of OOPSLA'89. *SIGPLAN Not. (ACM)*, v. 24, n.10, p.1-6, Oct. 1989.

BELLIN, D.; SUCHMAN, S. *Manual de desenvolvimento de sistemas estruturados*. São Paulo, Makron Books do Brasil, 1993. 223 p.

BERRY, J. The Waite Group's C++ programming. Indianapolis, *Howard W. Sams & Company*, 1988. 381p.

BOOCH, G. *Object-oriented design with applications*. Menlo Park, Benjamin/Cummings, 1991. 580p.

BROWN, A. *Object-oriented databases: applications in software engineering*. London, McGraw-Hill, 1991. 184p.

CHEN, P. *Gerenciando banco de dados: a abordagem entidade-relacionamento para projeto lógico*. São Paulo, McGraw-Hill, 1990. 80p.

COAD, P.; YOURDON, E. *Object-oriented analysis*. Englewood Cliffs, Prentice-Hall, 1990. 232p.

DE CHAMPEAUX, D.; LEA, D.; FAURE, P. *Object-oriented system development*. Reading, Addison-Wesley, 1993. 532p.

DEMARCO, T. *Structured analysis and system specification.* Reading, Prentice-Hall, 1978. 352p.

FICHMAN, R. G.; KEMERER, C. F. Adaptation of software-engineering process innovations: the case of object-orientation. Cambridge, *Center for Information Systems Research*, WP n. 242, Sloan School of Management (MIT). 1992 (a).

FICHMAN, R. G.; KEMERER, C. F. Object-oriented and conventional analysis and design methodologies: comparison and critique. *IEEE Computer*, v. 25, n. 10, p. 22-39, oct. 1992 (b).

FISHER, A. S. *CASE - Utilização de ferramentas para desenvolvimento de software.* Rio de Janeiro, Campus. 1990. 264p.

FLAVIN, M. *Fundamental concepts of information modeling.* New York, Yourdon Press, 1981.

FOLEY, J.; KIM, W. C.; KOVACEVIC, S.; MURRAY, K. Defining interfaces at a righ level of abstraction. *IEEE Software*, v. 6, n. 1, p. 25-32. 1989.

GANE, C.; SARSON, T. *Structured systems analysis: tools and techniques.* Englewood Cliffs, Prentice-Hall, 1979. 241p.

GOLDBERG, A.; ROBSON, D. *Smalltalk-80: the language and its implementation.* Reading, Addison-Wesley, 1983. 714p.

GOLDBERG, A. Object-oriented Smalltalk. *DBMS*, p. 38-42, Oct. 1992.

HAREL, D. On visual formalisms. *Communications of the ACM*, v. 31, n. 5, p. 514-530, May. 1988.

HENDERSON-SELLERS, B.; EDWARDS, J. M. The object-oriented systems life cycle. *Communications of the ACM*, v. 33, n. 9, p.142-159, Sep. 1990.

JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; ÖVERGAARD, G. *Object-oriented software engineering: a use case driven approach.* Reading, Addison-Wesley, 1992. 524p.

JACOBSON, I. Is object technology software's industrial plataform? *IEEE Software*, v. 10, n. 1, p. 24-30, Jan. 1993.

KORSON, T.; MCGREGOR, J. D. Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, v. 33, n. 9, p. 40-60, Sep. 1990.

KOZACZYNSKI, W.; KUNTZMANN-COMBELLES, A. What it takes to make OO work. *IEEE Software*, v. 10, n. 1, p. 20-23, Jan. 1993.

LEE, L. S. Mudança de paradigma: pedra fundamental de OOP. São Paulo, *COMDEX- SUCESU*, 1993. (Anotações de palestra)

LINTON, M. A.; VLISSIDES, J. M.; CALDER, P. R. Composing user interfaces with InterViews. *IEEE Computer*, v. 22, n. 2, p. 8-22, Feb. 1989.

LUCCA, V. L. Uma avaliação do processo de seleção de ferramentas CASE no mercado brasileiro. São Carlos, 1992. 177p. Tese (Mestrado). Centro de Ciências Exatas e de Tecnologia / Universidade Federal de São Carlos.

MARTIN, J.; MCCLURE, C. *Técnicas estruturadas e CASE*. São Paulo, Makron Books do Brasil, 1991. 854p.

MARTIN, J. *Principles of object-oriented analysis and design*. Englewood Cliffs, Prentice-Hall, 1993. 412p.

MCMENAMIN, S. M.; PALMER, J. F. *Essential systems analysis*. Englewood Cliffs, Yourdon Press, 1984. 392p.

METAEDIT, version 1.1. Ylistönmäentie, *Metacase Consulting Oy*. Obtida através de FTP "Anonymous". 1993.

MEYER, B. Lessons from the design of the Eiffel libraries. *Communications of the ACM*, v. 33, n. 9, p. 68-88, Sep. 1990.

MEYER, B. *Object-oriented software construction*. New York, Prentice-Hall, 1988. 534p.

MONARCHI, D. E.; PUHR, G. I. A research typology for object-oriented analysis and design. *Communications of the ACM*, v. 35, n. 9, p. 35-47, Sep. 1992.

O'BRIEN, L. Issues of programming. *Computer Language*, v. 10, n. 1, p. 45-52, Jan. 1993.

OBJECT TOOLKIT. Houston. *TechnoJock Software, Inc.* Obtida através de FTP "Anonymous". 1991.

OOTHER - Documention Tool, release 1.06a. Norsborg, *Zielinski Metod & System Utveckling.* Obtida através de FTP "Anonymous". 1994.

PITTMAN, M. Lessons learned in managing object-oriented development. *IEEE Software*, v. 10, n. 1, p. 43-53, Jan. 1993.

PRESSMAN, R. S. *Software engineering: a practitioner's approach.* 3. ed. New York, McGraw-Hill, 1992. 793p.

REENSKAUG, T.; NORDHAGEN, E. The description of complex object-oriented systems: Version 1. Oslo, *Senter for Industriforskning,* 1989.

RINE, D.; BHARGAVA, B. Object-oriented computing. *IEEE Computer,* v. 25, n. 10, p. 6-10, Oct. 1992.

ROWAN, L. B. Visual programming: application design for end users and professional developers. Boston, *Office Computing Report,* v. 7, n. 2, p. 3-10, Fev. 1992.

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.; LORENSEN, W. *Object-oriented modeling and design.* Englewood Cliffs, Prentice-Hall, 1991. 500p.

SHLAER, S.; MELLOR, S. J. *Análise de sistemas orientada para objetos.* São Paulo, Makron Books do Brasil, 1990. 178p.

SHLAER, S.; MELLOR, S. J. *Object life cycles: modeling the world in states.* Englewood Cliffs, Yourdon Press, 1992.

SIGRIST, S. R.; DALTRINI, B. M. Partição de sistemas no paradigma de objeto. *Revista do Instituto de Informática,* Campinas, PUCCAMP, 1994. (no prelo)

SNYDER, A. The essence of objects: concepts and terms. *IEEE Software*, v. 10, n. 1, p. 31-42, Jan. 1993.

SOUZA, E. M. Comparação entre metodologias de análise orientada a objetos. São Paulo, *Sucesu,* 1993. (Anotações de palestra)

STEELE Jr., G. L. *Common Lisp - The language*. 2. ed. Lexington, Digital Press, 1990. 1029p.

STROUSTRUP, B. *The C++ programming language*. 2. ed., Reading, Addison-Wesley, 1991. 691p.

TAKAHASHI, T. *O paradigma de objetos: introdução e tendências*. Campinas, Projeto ETHOS, CPqD-TELEBRÁS, 1989. 96p.

VALDÉS, R. Application frameworks and class libraries. *Dr. Dobb's Journal*, p.18-35, Oct. 1992 (a).

VALDÉS, R. Application frameworks and class libraries. *Dr. Dobb's Journal*, p. 86-87, Oct. 1992 (b).

WARD, P. T.; MELLOR, S. J. *Structured development for real-time systems*. Englewood Cliffs, Yourdon Press, 1985. 3v.

WARD, P.T. How to integrate object-orientation with structured analysis and design. *IEEE Software*, v. 6, n. 2, p. 74-82, Mar. 1989.

WASSERMAN, A. I.; PIRCHER, P. A; MULLER, R. J. An object-oriented structured design method for code generation. *Software Eng. Notes*, v. 14, n. 1, p. 32-55, Jan. 1989.

WINBLAD, A. L.; EDWARDS, S. D.; KING, D. R. *Software orientado ao objeto*. São Paulo, Makron Books do Brasil, 1993. 314p.

WIRFS-BROCK, R. J.; JOHNSON, R. E. Surveying current research in object-oriented design. *Communications of the ACM*, v. 3, n. 9, p. 104-124, Sep. 1990.

WIRTH, N. Program development by stepwise refinement. *Communications of the ACM*, v. 14, n. 4, p. 221-227, Apr. 1971.

YAMAZAKI, S.; KAJIHARA, K.; ITO, M.; YASUHARA, R. Object-oriented design of telecommunication software. *IEEE Software*, v. 10, n. 1, p. 81-87, Jan. 1993.

YOURDON, E.; CONSTANTINE, L. *Structured design: fundamentals of a discipline of computer program and systems design*. 2. ed. New York, Yourdon Press, 1978. 446p.

YOURDON, E. *Modern structured analysis.* Englewood Cliffs, Prentice-Hall, 1989. 672p.