MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-6891-TDI/652

DESENVOLVIMENTO DE SOFTWARE POR ENGENHEIROS: DIRETRIZES E METODOLOGIAS

Adriana Cursino Thomé

Dissertação de Mestrado em Computação Aplicada, orientada pelo Dr. Tatuo Nakanishi e pelo Dr. João Bosco Cunha, aprovada em agosto de 1998.

INPE
São José dos Campos
1998

Publicado por:

Coordenação de Ensino, Documentação e
Programas Especiais - CEP

Instituto Nacional de Pesquisas Espaciais - INPE
Caixa Postal 515
12201-970 - São José dos Campos - SP - Brasil
Fone: (012) 345.6911
Fax: (012) 345.6919
E-Mail: marciana@sid.inpe.br

- Solicita-se intercâmbio
- We ask for exchange
- Si sollecita intercambio
- On demande l'échange
- Mann bittet un Austausch
- Pidese canje
- Просим обмену
- 歡迎交換
- 出版物交換のお願い

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-6891-TDI/652

DESENVOLVIMENTO DE SOFTWARE POR ENGENHEIROS: DIRETRIZES E METODOLOGIAS

Adriana Cursino Thomé

Dissertação de Mestrado em Computação Aplicada, orientada pelo Dr. Tatuo Nakanishi e pelo Dr. João Bosco Cunha, aprovada em agosto de 1998.

INPE
São José dos Campos
1998

681. 3. 06

THOMÉ, A.C.

Desenvolvimento de Software por engenheiros: diretrizes e metodologias / A.C.Thomé. – São José dos Campos: INPE, 1998.

352p. – (INPE-6891-TDI/652).

1.Engenharia de software. 2.Metodologias.3.Técnicas orientadas para objeto. 7.Qualidade de softwares. I.Título.

Aprovado pela Banca Examinadora em cumprimento a requisito exigido para a obtenção do Título de **Mestre** em **Computação Aplicada.**

Dr. Solon Venâncio de Carvalho
Presidente

Dr. Tatuo Nakanishi
Orientador

Dr. João Bosco Schumann Cunha
Orientador

Dr. Glauco Augusto de Paula Caurim
Membro da Banca Convidado

Dr. Sdnei de Brito Alves
Membro da Banca Convidado

Candidata: Adriana Cursino Thomé

São José dos Campos, 20 de agosto de 1998

Dos sóis da imensidão às últimas gotas d'água no centro da Terra, tudo o que há de bom e belo nasce e vive do trabalho constante.

Emmanuel

"No Portal da Luz"

Psicografado por Chico Xavier

Ao meu esposo e companheiro Thomé.

Ao meu filho Ruan, fonte de alegria e de luz.

Aos meus pais, fonte de incentivo e compreensão.

AGRADECIMENTOS

Ao meu orientador e amigo Tatuo pela dedicação e confiança.

Ao meu co-orientador João Bosco pela colaboração.

Aos Engenheiros e Analistas por mim entrevistados, pela imensa contribuição que prestaram ao desenvolvimento deste trabalho.

RESUMO

Esta dissertação propõe diretrizes para auxiliar Engenheiros a desenvolver sistemas de software de pequeno, médio, e grande porte, usando o Paradigma Clássico e o Paradigma da Orientação a Objetos. Este trabalho traça portanto, uma linha de desenvolvimento de software que vai desde a concepção do sistema até a fase de implementação e testes; indicando metodologias, técnicas, e cuidados a serem considerados durante o processo de desenvolvimento, facilitando dessa maneira a construção de softwares de qualidade, sem precisar se tornar um especialista em Engenharia de Software. Isto permite que os Engenheiros, que muitas vezes tem pouco conhecimento sobre Engenharia de Software, possam ter uma referência de como proceder na construção de softwares, podendo desta maneira construir, mais facilmente, softwares apropriados para seus objetivos.

SOFTWARE DEVELOPMENT BY ENGINEERS : ROUTES AND METHODOLOGIES

ABSTRACT

This dissertation proposes routes to help Engineers to develop small, medium, and large software systems, using the Classical Paradigm and the Object-Oriented Paradigm. In such case, this work traces a line of software development that goes since its conception until the implementation and tests phase; indicating methodologies, techniques, and precautions to be considered during the development process, helping thus the construction of quality softwares, without the need of becoming a Software Engineer specialist. This way, it allows Engineers, who many times have little knowledge about Software Engineering, to have a source of reference on how to proceed in software construction, and can in this manner easier build software systems appropriate to their goals.

SUMÁRIO

LISTA DE TABELAS

Pág.

LISTA DE FIGURAS

Pág.

CAPÍTULO 1

INTRODUÇÃO

Computadores tornaram-se ferramentas indispensáveis para um grande número de aplicações comerciais, tecnológicas, de comando e controle, pesquisa científica e outras; e muitas tarefas hoje desempenhadas rotineiramente, seriam virtualmente impossíveis de serem realizadas sem o emprego de computadores digitais devidamente programados.

Porém, computadores sofisticados, sistemas operacionais poderosos e softwares envelhecem com uma rapidez difícil de acompanhar. Isto força investimentos constantes em projetos que nem sempre chegam ao fim - ou quando chegam, às vezes não acrescentam nenhum benefício substancial ao negócio da empresa.

Vejamos o que mostra uma pesquisa feita pelo Instituto Americano "Standish Group" com 365 companhias espalhadas pelo mundo. Descobriu-se que 31% dos projetos de informática são cancelados antes de chegar ao fim, e que 53% superam a estimativa inicial dos gastos e de prazo de implantação. Somente 16% chegam ao final no tempo estipulado e sem estourar o orçamento (Militello,1997).

Além disso, um estudo da consultoria americana "Gartner Group" indica que 70% dos projetos de informática não geram os benefícios esperados, e isto acontece principalmente porque a maioria das corporações não sabe o que realmente pode esperar dos sistemas que implantaram (Militello,1997).

Tudo isto nos faz crer que a maneira pela qual softwares devem ser construídos deve sofrer uma série de modificações. Com o crescimento da potência de cômputo e com a drástica redução do custo dos equipamentos; hoje, o desenvolvedor de software tende a ser muito mais caro que o tempo de processamento e a memória de computador. Isto

porque cada vez mais tem se exigido a construção de softwares de qualidade, com elevada confiabilidade e manutebilidade; tudo isto com economicidade (Staa,1987).

Isto faz com que se procure, cada vez mais, utilizar instrumentos (ferramentas, metodologias, e técnicas) que facilitem o desenvolvimento de softwares, mesmo que eles consumam um pouco mais de recursos de computação. Em decorrência dessas novas exigências, começa a surgir o que podemos chamar de "Engenharia de Software".

Grande parte das pessoas que desenvolvem sistemas de software não lançam mão das facilidades que a Engenharia de Software pode proporcionar, e constróem seus sistemas sem nenhum planejamento. Isto se dá principalmente porque a história da arte de se desenvolver software enfatiza a individualidade do programador e o código que ele produz, encorajando a idéia de que o conhecimento do desenvolvedor está na habilidade de se produzir código sem a necessidade de nenhuma técnica de apoio. Por isto, apesar de os benefícios que as técnicas para o desenvolvimento de software podem trazer serem bem conhecidos, o seu uso na produção de software continua limitado (Rout,1992).

O principal problema da falta de planejamento é o aparecimento freqüente de software de pouca qualidade. Um planejamento mal elaborado gera definições pobres, que são a maior causa das falhas futuras do software; pois elas podem acarretar muitos outros problemas, como dificuldade de manutenção e baixa reutilização. Além disso, é importante lembrar que quanto mais avançado o projeto, maior o impacto no custo proporcionado por modificações, sendo portanto vital para o sucesso de um software, que se faça um mínimo de planejamento.

Para dificultar ainda mais o interesse das pessoas pelo uso da Engenharia de Software, existem inúmeras técnicas, ferramentas e metodologias para desenvolvimento de softwares, e quais delas selecionar e como usá-las depende do tipo de sistema que se quer desenvolver.

Além disso, a bibliografia existente nessa área dificilmente apresenta um caminho direto para a construção do software, de maneira que o leitor tem que ser capaz de tomar suas próprias decisões na escolha do caminho que é melhor para o desenvolvimento do seu sistema.

Em vista disso, até mesmo para um profissional da área de Engenharia de Software muitas vezes é difícil tomar uma decisão de quais técnicas, metodologias e ferramentas adotar, a não ser que se tenha certas diretrizes em mente durante o processo de seleção (Bell,1994).

Dessa maneira, para um profissional que não é dessa área pode ser ainda mais difícil tomar decisões de projeto; já que isto exige que se tenha um conhecimento razoável sobre a área de Engenharia de Software, o que na maioria das vezes não interessa para ele. Na verdade, na maior parte dos casos o que interessa para ele é apenas selecionar uma técnica, metodologia ou ferramenta que seja boa para o desenvolvimento do seu software, e se aprofundar nela; sem ter que conhecer as demais.

Então, muitas vezes, profissionais que não são da área de Engenharia de Software até pensam em fazer um certo planejamento do seu software, mas, desanimam ao perceber a quantidade de metodologias, ferramentas e técnicas existentes; e ao perceber que para escolher uma delas é necessário que se conheça um pouco de cada uma, para que se possa fazer comparações, e, finalmente chegar a uma conclusão do que seria mais adequado para o caso. Assim, muitas vezes, essas pessoas ou acabam desistindo de usar alguma dessas técnicas, ferramentas ou metodologias, ou adotam qualquer uma, o que não é nem um pouco apropriado.

Nesta dissertação falaremos um pouco sobre a Engenharia de Software, seus conceitos e paradigmas; discutiremos o porquê de ela ser tão importante; e mostraremos como é possível se utilizar da Engenharia de Software para obter sistemas mais confiáveis, manutíveis e com uma maior qualidade.

Em seguida traçaremos diretrizes que poderão auxiliar os Engenheiros em geral na construção de sistemas de pequeno, médio e grande porte, abrangendo o paradigma clássico e o paradigma da orientação a objetos.

Optou-se por desenvolver este tipo de dissertação pois é muito comum Engenheiros necessitarem de softwares para otimizar o seu trabalho, então, é natural que eles gerem softwares que os auxiliem; porém isto nem sempre é feito de maneira adequada, gerando-se muitas vezes softwares com uma qualidade discutível.

Pode-se dizer que os sistemas de software gerados para dar apoio à Engenharia são, na maioria das vezes, constituídos por (Shoup,1979) :

1) Computações similares a cálculos feitos à mão, mas que precisam ser feitos muitas vezes;
2) Computações que são extensões de cálculos manuais, mas que são muito complexos para serem feitos à mão por razões de tempo e acurácia, e
3) Manipulação de dados para propósitos de visualização, manufatura ou documentação.

Estes sistemas são na maior parte das vezes simples de serem construídos, e em sua maioria podem ser classificados como Sistemas de Pequeno Porte.

Por isto, nesta dissertação os sistemas de pequeno porte são abordados com uma maior profundidade, já que entendemos que para construir sistemas de médio e grande porte o Engenheiro precisaria deter um conhecimento razoável de engenharia de software, e seria aconselhável, pelo menos no caso dos sistemas de grande porte, que ele procurasse ajuda de um profissional da área.

Gostaríamos de enfatizar que os caminhos para a construção de softwares indicados nesta dissertação não são os únicos, e nem necessariamente os mais eficientes, mas com

certeza se seguidos, levam à construção do sistema proposto com uma grande garantia de qualidade.

Finalmente, falaremos um pouco das conclusões que chegamos após o desenvolvimento deste trabalho de dissertação, das dificuldades encontradas para desenvolvê-lo, e das perspectivas e sugestões de trabalhos futuros baseados no trabalho aqui desenvolvido.

CAPÍTULO 2

SOFTWARE E ENGENHARIA DE SOFTWARE

Software pode ser entendido como sendo **(1)** instruções (programas de computador) que quando executadas, produzem a função e o desempenho desejados; **(2)** estruturas de dados que possibilitam que os programas manipulem adequadamente a informação e **(3)** documentos que descrevem a operação e o uso dos programas (Pressman,1995).

O software é formado, portanto, pelo conjunto de programas, bases de dados e documentos (manual do usuário, documentação interna e externa dos programas, modelos, diagramas etc...) gerados durante o processo de desenvolvimento e manutenção deste software.

Isso significa que em contraposição ao que normalmente é feito, a geração de documentos é uma atividade que deve fazer parte do processo de desenvolvimento do software, pois os documentos possibilitam que se tenha informações sobre o sistema e sobre o software disponíveis para uso posterior.

Os documentos são, portanto, muito importantes para a compreensão e entendimento de todas as informações que estão armazenadas e que são manipuladas pelo sistema, possibilitando assim, uma melhor compreensão dos programas e bases de dados; e também uma melhor realização das alterações (correção de defeitos, adaptações e acréscimos) que podem ocorrer após o software ter entrado em funcionamento.

2.1 - A crise do Software

Segundo Pressman (1995), a "crise" do software vem nos acompanhando praticamente desde que se começou a se desenvolver softwares.

Podemos defini-la como sendo uma série de problemas que estão sendo encontrados no processo de criação do software. Esses problemas não se limitam a softwares que não funcionam; ao contrário, a "crise" abrange problemas associados a como desenvolver os softwares, a como manter um volume crescente de softwares já existentes, e a como lidar com uma demanda cada vez maior; além de problemas como planejamento mal elaborado, estimativas de custo abaixo do necessário, cronogramas curtos, dificuldade de manutenção e baixa reutilização.

Na verdade, esses problemas são a manifestação mais visível de outras dificuldades do software, como a falta de tempo dedicado para se fazer uma coleta adequada dos dados sobre o processo de desenvolvimento do software, e a falta de comunicação entre o cliente e o desenvolvedor , acarretando em requisitos pobres, que nos dizem pouco a respeito do software que deverá ser desenvolvido; gerando portanto, produtos que não correspondem na maioria das vezes às expectativas do usuário. Além disso, a qualidade do software freqüentemente é suspeita, já que a especificação e a estruturação do sistema não são vistas como algo digno de muita atenção; e como também não é dada a devida importância para os testes, geralmente eles tem pouca confiabilidade.

Tudo isso faz com que se construa freqüentemente softwares inadministráveis, insatisfatórios, improdutivos, não confiáveis, ineficientes, inflexíveis e de difícil manutenção.

Portanto, como podemos observar a "Crise do Software" se dá muito mais por falhas humanas, atribuídas às pessoas encarregadas da criação do software, do que por falhas atribuídas à máquina. Na verdade, podemos dizer que a maioria dos problemas citados acima acontecem devido à uma falta de abordagem de "Engenharia" no processo de criação dos softwares (Pressman,1995).

Porém, cada um dos problemas citados acima pode ser corrigido, ou pelo menos minorado, através de soluções para o desenvolvimento de software apontadas pela área

de Engenharia de Software, aliadas a um uso contínuo de técnicas, metodologias e ferramentas apropriadas.

Enfim, a solução para a crise do software está na melhoria da qualidade do software e da produtividade do seu desenvolvimento, através da melhoria do processo de desenvolvimento. Por processo de desenvolvimento pode-se entender o conjunto de todas as atividades e etapas do desenvolvimento do software.

2.2 - Engenharia de Software : Definição

Os problemas que vem aparecendo durante o desenvolvimento de software não desaparecerão da noite para o dia. Reconhecer os problemas e suas causas são os primeiros passos em direção às soluções. Porém, as próprias soluções devem oferecer assistência prática ao desenvolvedor de software, melhorar a qualidade do software e, por fim, permitir que o software acompanhe a evolução do *hardware*.

Não existe uma única abordagem particular que seja a melhor para a solução da "crise" que abala o desenvolvimento de software. Entretanto, ao combinarmos metodologias abrangentes para todas as fases de desenvolvimento do software, melhores ferramentas para automatizar essas metodologias, blocos de construção mais poderosos para a implementação do software, melhores técnicas para a garantia da qualidade do software, e uma filosofia de coordenação predominante de controle e administração; podemos conseguir uma disciplina para o desenvolvimento do software - disciplina esta chamada "Engenharia de Software".

A Engenharia de Software propõe diferentes meios para a abordagem do problema, e cabe aos desenvolvedores escolher aqueles que serão utilizados para o desenvolvimento do seu software particular. A escolha nem sempre é simples, pois ela envolve um conjunto de aspectos tais como o tipo de sistema a ser desenvolvido, o perfil da equipe

de desenvolvimento, o nível de qualidade desejado, o rigor do cronograma, as ferramentas disponíveis, as formas de utilização, entre outros.

A Engenharia de software trata, portanto, a construção e a manutenção dos sistemas de software propondo técnicas, metodologias e ferramentas para abordar o processo de desenvolvimento, bem como suas utilizações. Ela abrange também a organização do ambiente de trabalho, e o controle e gerenciamento de todas as atividades relacionadas ao processo de desenvolvimento, visando sempre a produtividade das tarefas e a qualidade dos produtos.

2.3 - Engenharia de Software : Metodologias, Técnicas e Ferramentas

A Engenharia de Software abrange um conjunto de três elementos fundamentais - metodologias, técnicas e ferramentas - que possibilita ao gerente o controle do processo de desenvolvimento do software e oferece ao profissional uma base para a construção de software de qualidade.

As técnicas da Engenharia de Software proporcionam os detalhes de "como fazer" para construir o software. Elas envolvem um amplo conjunto de tarefas que incluem planejamento e estimativa do projeto, análise de requisitos de software e de sistemas, projeto da estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste e manutenção. As técnicas da Engenharia de Software muitas vezes introduzem uma notação gráfica ou orientada à linguagem, além de introduzirem um conjunto de critérios para a qualidade do software.

As ferramentas da Engenharia de Software proporcionam técnicas de apoio automatizado ou semi-automatizado para o processo de desenvolvimento do sistema. Quando as ferramentas são integradas de forma que a informação criada por uma ferramenta possa ser usada por outra, é estabelecido um sistema de suporte ao

desenvolvimento de software chamado Engenharia de Software Auxiliada por Computador ("Computer-Aided Software Engineering" (CASE)).

As metodologias da Engenharia de Software constituem o elo de ligação que mantém juntas as técnicas e as ferramentas. Elas definem a seqüência em que as técnicas serão aplicadas, os produtos que serão entregues (documentos, relatórios, formulários, etc...), os controles que ajudam a assegurar a qualidade e a coordenar as mudanças, e os marcos de referência que possibilitam aos gerentes de software avaliar o progresso do desenvolvimento (Pressman,1995).

2.4 - Qualidade do Software

Em qualquer ramo industrial, a qualidade do produto é um objetivo de projeto, sendo raras as ocasiões em que a qualidade poderá ser incorporada ao produto de forma não consciente ou, então, após o produto ter sido construído.

Assim, ao desenvolvermos um software, devemos sempre ter em mira o objetivo de qualidade previamente estabelecido. Sem conhecer este objetivo e sem trabalhar conscientemente para alcançá-lo, é virtualmente impossível assegurar a qualidade do software durante sua construção.

De uma forma genérica, podemos dizer que um software de boa qualidade produz resultados úteis e confiáveis na oportunidade certa; é ameno ao uso, é mensurável, corrigível, modificável e evolutível; opera em máquinas e ambientes reais; foi desenvolvido de forma econômica e no prazo estipulado; e opera com economia de recursos. Qualidade de software é, pois, um conceito muito mais amplo do que software correto e bem documentado, requerendo para ser conseguida, metodologias e técnicas de desenvolvimento específicas (Staa,1987).

2.4.1 - Fatores de Qualidade do Software

São atributos, condições ou características da construção do software que são determinantes de sua qualidade. Eles avaliam o software a partir de três pontos distintos, **(1)** operação do produto (usando-o); **(2)** revisão do produto (mudando-o) e **(3)** transição do produto (modificando-o para funcionar num ambiente diferente). São eles (Staa,1987):

- **Utilidade.** Visa assegurar a confiabilidade e utilidade dos resultados; isto é, verifica se o software está realmente atendendo aos requisitos do usuário, se os resultados que estão sendo gerados estão corretos, e se o programa protege-se contra condições ambientais e de uso diversas.

- **Utilizabilidade.** Visa assegurar a facilidade de uso do software nas diversas condições de contorno, bem como, segundo os pontos de vista de seus diferentes tipos de usuários. Isto é, mede o quão o software é fácil de operar, instalar, fornecer dados, interpretar resultados, verificar a ocorrência de falhas, treinar novos usuários, etc...

- **Economicidade.** Visa assegurar a economicidade no consumo dos diversos recursos (pessoal, papel, *CPU*, memória, etc...) necessários para obter o serviço do produto, bem como para adquirir o software em si.

- **Alterabilidade.** Visa assegurar a capacidade de adaptação e alteração do software, durante e depois do desenvolvimento.

- **Validabilidade.** Um produto possui validabilidade na medida em que facilita a certificação, testes, verificação e aceitação completa das alterações incorporadas.

• **Portabilidade.** Um produto possui portabilidade na medida inversa do esforço necessário para transferi-lo de uma situação ambiental para outra, isto é, mede o quanto o software está preparado para uma mudança de ambiente de execução.

• **Flexibilidade.** Dizemos que um software é flexível se ele está preparado para diferentes formas de uso.

• **Reusabilidade.** Mede o quão fácil ou difícil é o reuso desse software.

2.4.2 - Garantia de Qualidade do Software

Atualmente, a única forma de se ter uma certa garantia de qualidade do software é através do processo de desenvolvimento, o que significa que é difícil melhorar a qualidade de um software após sua construção.

A garantia da qualidade de software compreende uma variedade de tarefas associadas às atividades de desenvolvimento e manutenção, como **(1)** aplicação de métodos técnicos; **(2)** realização de revisões técnicas formais; **(3)** atividades de teste de software; **(4)** aplicação de padrões; **(5)** controle de mudanças; **(6)** manutenção de registros e documentos técnicos e **(7)** gerenciamento.

A qualidade de software inicia-se de fato, com o conjunto de metodologias, técnicas e ferramentas que ajudam o analista a conseguir uma especificação de elevada qualidade, e o projetista a desenvolver um projeto também de elevada qualidade.

Assim que uma especificação (ou protótipo) de um projeto tiverem sido criados, cada um deve ser avaliado quanto à qualidade. A atividade central que leva a efeito a avaliação da qualidade é a Revisão Técnica Formal. A revisão técnica formal é um encontro estilizado realizado pelo pessoal técnico com o propósito único de descobrir problemas de qualidade.

A atividade de Teste de Software combina uma estratégia de múltiplos passos com uma série de métodos de projeto de casos de testes que ajudam a garantir uma detecção efetiva de erros. Muitos desenvolvedores de software usam a atividade de teste de software como uma "rede de segurança" de garantia de qualidade. Ou seja, os desenvolvedores pressupõem que uma atividade de teste cuidadosa revelará a maioria dos erros, abrandando a necessidade de outras atividades de controle de qualidade. Infelizmente, a atividade de testes, mesmo quando bem realizada, não é tão efetiva quanto gostaríamos que fosse para todas as classes de erros.

O grau em que Padrões e Procedimentos Formais são aplicados no processo de engenharia de software varia de empresa para empresa. Em muitos casos, os padrões são determinados pelos clientes ou por imposições reguladoras. Em outras situações, os padrões são auto-impostos. Se existirem padrões formais (escritos), uma atividade de controle de qualidade deve ser estabelecida para garantir que eles sejam seguidos.

Uma grande ameaça à qualidade de software vem de uma fonte aparentemente benigna, as "mudanças". Toda mudança no software tem potencial para introduzir erros ou criar efeitos colaterais que propagam erros. O processo de Controle de Mudanças contribui diretamente para a qualidade do software ao formalizar pedidos de mudança, avaliar a natureza da mudança e controlar o impacto da mudança. O controle de mudanças é aplicado durante a fase de desenvolvimento do software e, posteriormente, durante a fase de manutenção do software.

A Manutenção dos registros e documentos técnicos, para a garantia da qualidade de software, deve oferecer procedimentos para a coleta e disseminação de informações de controle de qualidade. Os resultados de revisões, auditorias, controle de mudanças, testes e outras atividades de controle de qualidade devem tornar-se parte do registro histórico de um projeto e devem ser levados ao conhecimento do pessoal de desenvolvimento, tendo-se como base a necessidade de conhecimento. Por exemplo, os resultados de cada revisão técnica formal de um projeto procedimental são registrados e

podem ser colocados numa pasta que contenha todas as informações técnicas e de controle de qualidade sobre um módulo (o que é módulo será explicado no item **2.10.1.1 - Técnicas Estruturadas**, situado mais adiante neste mesmo Capítulo) (Pressman,1995).

Dessa maneira, para garantirmos a qualidade de um software, e portanto garantir que ele siga os fatores de qualidade citados no item anterior; é necessário definir e seguir um processo de desenvolvimento e um processo de manutenção para o software, usando para isso metodologias, técnicas e ferramentas que nos auxiliem a realizar esse trabalho. Além disso, é necessário também, controlar esse processo de desenvolvimento e esse processo de manutenção; tornando-se vital que haja um gerenciamento que coordene essas atividades, permitindo que elas interajam da melhor maneira possível.

Quando se deseja ter uma qualidade muito elevada do software é necessário adicionar a atividade de medição dos diferentes fatores de qualidade. Algumas medidas diretas podem ser utilizadas, tais como o número de erros detectados em um certo período de tempo, a velocidade de execução, o número de linhas de código produzidas, a quantidade de memória utilizada etc... Outros fatores tais como a facilidade de uso, a alterabilidade, a flexibilidade, e outros; só podem ser avaliados indiretamente, sendo que essa avaliação é difícil de ser realizada por envolver aspectos de subjetividade.

2.5 - Sistema de Software e Domínio do Problema

O sistema de software está contido em um domínio do problema que pode ser definido nada mais nada menos como os elementos que se relacionam com o software, e que portanto, o afetam de alguma maneira. Esses elementos podem ser pessoas (usuários, operadores), equipamentos e *hardware* (*CPU*, memória, equipamentos eletromecânicos), procedimentos (etapas que definem o comportamento e/ou uso dos elementos do domínio), e outros sistemas. Além disso, o domínio do problema poderá conter (ou estar contido em) outros domínios do problema.

Ao construirmos um software é necessário que investiguemos o "domínio do problema" desse software e as "responsabilidades" desse software dentro dele. Essa investigação para identificar as características do domínio do problema nem sempre é trivial, na verdade, ela é um dos maiores problemas encontrados pelos analistas (Coad,1992).
A Figura 2.1 mostra a relação entre o sistema de software e o domínio do problema.

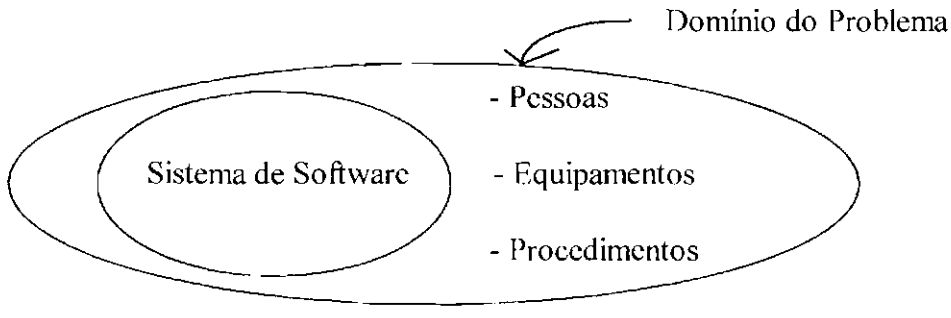


Fig. 2.1 - Sistema de Software e Domínio do Problema.
FONTE: Nakanishi (1995A, p. 3.1).

Sendo assim, é muito importante que ao iniciarmos o desenvolvimento de um software nos situemos em seu domínio do problema, isto é, é muito importante que definamos os elementos com os quais o software se relaciona, e que pesquisemos as características desses elementos; para que possamos assim, modelar da melhor maneira possível o relacionamento do software com esses elementos.

2.6 - Ciclo de Vida do Software

O ciclo de vida do software é uma seqüência de fases (ou etapas) que se iniciam ao conceber-se o software, e terminam ao descontinuá-lo.

De uma maneira genérica, independentemente do paradigma, da área de aplicação, do tamanho e da complexidade do software; o seu ciclo de vida contém três fases : Análise, Projeto e Implementação, e Uso e Manutenção (Pressman,1995).

A fase de **Análise** focaliza "o quê". Ela procura, portanto, identificar os requisitos do sistema e do software; identificando quais informações tem que ser processadas, qual função e desempenho são desejados, quais interfaces devem ser estabelecidas, quais são as restrições de projeto, e quais são os critérios de validação que são exigidos para se definir um sistema bem sucedido. Não obstante as técnicas aplicadas durante a fase de especificação variem dependendo da metodologia de engenharia de software adotada, a fase de análise é composta das seguintes etapas :

- **Estudo de Viabilidade.** Nessa etapa é feito um estudo da viabilidade do sistema em termos econômicos, técnicos e legais.

- **Especificação.** A especificação possui duas etapas. A primeira, que podemos chamar de "Definição" ou "Especificação Inicial", consiste de uma especificação detalhada da função e dos requisitos do software. A segunda consiste em se formalizar as informações obtidas na fase de "Definição" usando técnicas de engenharia de software para representá-las.

A fase de **Projeto e Implementação** focaliza "o como". Ela procura, portanto, estabelecer a estrutura do software, definindo as estruturas de dados e a arquitetura do software que tem que ser projetadas, definindo como os detalhes procedimentais tem que ser implementados, como o projeto será traduzido numa linguagem de programação, e como os testes tem que ser realizados. As técnicas aplicadas durante a fase de projeto podem variar, mas em geral, ela é composta das seguintes etapas :

- **Projeto do Software.** O projeto traduz os requisitos do software num conjunto de representações (algumas gráficas, outras tabulares ou baseadas em linguagem) que descrevem a estrutura de dados, a arquitetura, o procedimento algorítmico e as características de interface.

- **Codificação.** As representações do projeto devem ser convertidas numa linguagem artificial que resulte em instruções que possam ser executadas pelo computador. A etapa de codificação realiza essa conversão.

- **Testes.** Logo que é implementado em uma forma executável por máquina o software deve ser testado para que se possa descobrir defeitos de função, de lógica, e de implementação.

A fase de **Uso e Manutenção** abrange as alterações que podem ocorrer no uso do software após ele ter entrado em funcionamento. Os tipos de alterações que podem ocorrer são : Correção de defeitos, adaptações exigidas à medida que o ambiente de software evolui, e acréscimos exigidos pelo cliente.

A Figura 2.2 mostra a visão genérica do ciclo de vida de um software.

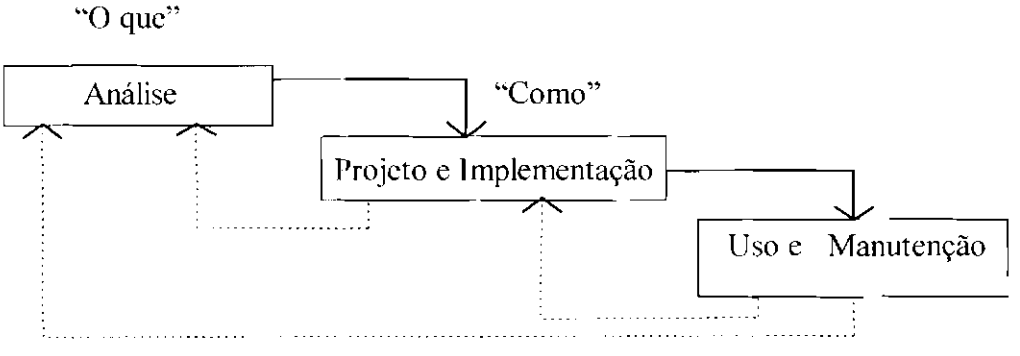


Fig. 2.2 - Visão Genérica do Ciclo de Vida de um Software.

As fases e passos correlatos descritos acima, são complementados por uma série de atividades como revisões para garantir que a qualidade seja mantida à medida que cada etapa é concluída; desenvolvimento e controle da documentação do sistema e do software, para que estejam disponíveis para uso posterior; e instituição do controle de mudanças, de forma que elas possam ocorrer apenas com aprovação e acompanhamento.

Entendemos por Desenvolvimento de Sistemas de Software, para escopo desse trabalho, a realização de todas as etapas do ciclo de vida de um software especificadas neste tópico.

2.7 - Estudo de Viabilidade do Software

O desenvolvimento de um sistema baseado em computador é afetado pela escassez de recursos e datas de entrega críticas. Portanto, é necessário e prudente avaliar a viabilidade de um projeto o mais cedo possível. Meses ou anos de esforços, milhares ou milhões de dólares e um grande embaraço profissional podem ser evitados se um sistema mal concebido for reconhecido logo na fase de definição. Somente gastando-se tempo para avaliar a viabilidade do sistema, é que conseguimos reduzir as chances de um impedimento de continuar o desenvolvimento em estágios posteriores do projeto.

Durante o trabalho de engenharia de sistemas, devemos nos concentrar nas seguintes áreas de estudo de viabilidade :

Viabilidade Econômica. Uma avaliação do custo de desenvolvimento confrontada com o benefício proporcionado pelo sistema sendo desenvolvido. A justificação econômica é geralmente a consideração primária para a maioria dos sistemas. Ela envolve uma ampla variedade de preocupações, que incluem análise de custo-benefício (discutida no próximo tópico), estratégias de renda corporativa à longo prazo, impacto sobre outros centros de lucros ou produtos, custo dos recursos necessários ao desenvolvimento e crescimento em potencial de mercado.

Viabilidade Técnica. Um estudo da função, do desempenho e das restrições que possam afetar a capacidade de se conseguir um sistema aceitável. A viabilidade técnica é a área mais difícil de ser avaliada nesta etapa do processo de desenvolvimento do sistema; pois os objetivos, funções e desempenho são um tanto vagos; e qualquer escolha parece possível se forem feitas pressuposições corretas.

Entre as considerações que normalmente são associadas à viabilidade técnica incluem-se os riscos de desenvolvimento, a disponibilidade de recursos, e a tecnologia existente para suportar tal sistema.

Viabilidade Legal. Uma determinação de qualquer infração, violação ou responsabilidade legal que possa resultar do desenvolvimento do sistema.

Alternativas. Uma avaliação das abordagens alternativas ao desenvolvimento do sistema. O grau em que alternativas são consideradas muitas vezes é limitado por restrições de tempo e de custo.

Um estudo de viabilidade não tem razão de ser para sistemas em que a justificação econômica seja óbvia, os riscos técnicos sejam baixos, poucos problemas jurídicos sejam esperados e não exista nenhuma alternativa razoável (Pressman,1995).

2.8 - Análise de Custo X Benefício do Software

A análise de Custo X Benefício é muito importante no estudo de viabilidade de um sistema. Ela é uma justificativa econômica para o projeto, e esboça os custos do desenvolvimento em relação aos benefícios que poderão ser obtidos.

A análise de Custo X Benefício é dificultada por critérios que variam de acordo com as características do sistema a ser desenvolvido, pelo tamanho do projeto, e pelo retorno esperado ao investimento como parte do plano de estratégia da empresa (Pressman,1995).

Na Figura 2.3, podemos observar a relação Custo X Benefício X Qualidade de um sistema.

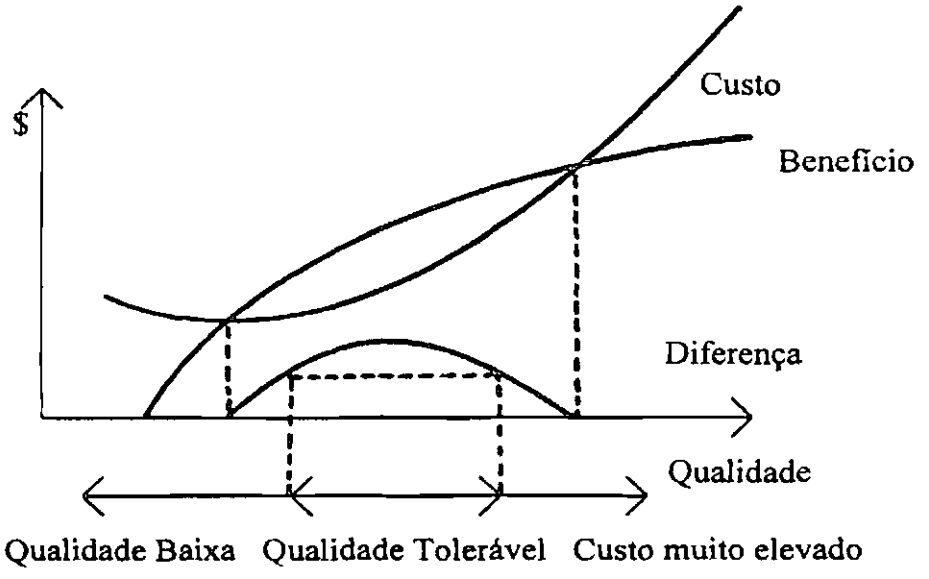


Fig. 2.3 - Análise de Qualidade X Custo X Benefício.

FONTE: Nakanishi (1995A, p. 1.9).

Assim, podemos notar que quanto maior for a qualidade do software, maior também é o benefício que ele proporcionará. Este benefício cresce rapidamente quando se melhora a qualidade, porém, até um certo ponto. A partir desse ponto é preciso aumentar muito a qualidade para se obter um pequeno incremento no benefício.

As providências para se melhorar a qualidade do software como a utilização de metodologias, técnicas e ferramentas para o seu desenvolvimento; podem, com o aumento da qualidade do software, até reduzir o seu custo. Porém, a partir de um certo ponto uma qualidade melhor implica em investimentos mais pesados, e conseqüentemente o custo para o desenvolvimento desse software irá crescer. Um software de qualidade muito alta poderá implicar então, em custos de desenvolvimento tão elevados que não justificariam os benefícios que o software iria proporcionar.

A curva resultante da diferença entre a curva de Benefício e a curva de Custo, serve de indicação para mostrar que existe uma faixa de qualidade tolerável onde o benefício é maior do que o custo, de uma forma compensadora. Porém, se a qualidade do software

desenvolvido for muito baixa, o benefício oferecido por ele será tão pequeno que não compensará tê-lo desenvolvido. Por outro lado, se sua qualidade for excessivamente elevada, o custo de seu desenvolvimento será muito grande, e também não compensará tê-lo desenvolvido.

2.9 - Modelagem e Abstração

Criamos modelos para obter uma melhor compreensão da entidade real a ser construída. Quando a entidade é um objeto (por exemplo, um prédio, um avião, uma máquina), podemos construir um modelo que seja idêntico em sua aparência e em sua forma, mas em escala menor. Porém, quando a entidade a ser construída é um software, nosso modelo deve assumir uma forma diferente. Ele deve ser capaz de modelar a informação que o software transforma, as funções que possibilitam que as transformações ocorram, e o comportamento do sistema quando a transformação está se desenvolvendo (Pressman,1995).

O principal motivo da modelagem é facilitar a compreensão de sistemas que são demasiadamente complexos. A mente humana só consegue tratar um limitado volume de informações a cada momento, e os modelos reduzem a complexidade dividindo-a em um pequeno número de pontos importantes a serem tratados de cada vez (Rumbaugh,1994).

Durante a análise de requisitos do software, criamos modelos do sistema a ser construído. Os modelos concentram-se naquilo que o sistema deve fazer; não em como ele o faz. Em muitos casos, os modelos que criamos fazem uso de uma notação gráfica que descreve as informações, o processamento, o comportamento do sistema e outras características, usando ícones distintos e reconhecíveis. Outras partes do modelo podem ser meramente textuais. Informações meramente descritivas podem ser oferecidas mediante o uso de uma linguagem natural ou de uma linguagem especializada que descreva as exigências (Pressman,1995).

Os modelos criados durante a análise de requisitos cumprem papéis importantes :

- o modelo ajuda o analista a entender a informação, a função e o comportamento de um sistema, tornando a tarefa de análise de requisitos mais fácil e mais sistemática;

- o modelo torna-se o ponto focal para a revisão e, portanto, a chave para a determinação da completeza, consistência e precisão da especificação, e

- o modelo torna-se base para o projeto, fornecendo ao projetista uma representação essencial do software, a qual pode ser "mapeada" num contexto de implementação.

Então, para a Engenharia de Software, um modelo é uma representação limitada da realidade (pessoas, fatos, objetos, organizações), e pode assumir diferentes formas, como por exemplo: texto descritivo, diagramas, software, e concepção mental.

Um modelo depende dos objetivos para os quais ele está sendo construído, isto é, para uma mesma realidade pode-se construir diferentes modelos para diferentes objetivos.

Os modelos são construídos utilizando a capacidade de abstração das pessoas. A abstração é o exame seletivo de determinados aspectos de um problema. O objetivo da abstração é isolar os aspectos que sejam importantes para algum propósito e suprimir os que não o forem. Muitas abstrações diferentes da mesma entidade são possíveis, dependendo do propósito para o qual forem feitas.

Todas as abstrações são incompletas e inexatas; pois tudo que se disser sobre a realidade, ou qualquer descrição dela, em geral é apenas uma parte dela. Na construção de modelos portanto, não se deve procurar a verdade absoluta, e sim, a adequação à algum propósito. Um bom modelo incorpora os aspectos fundamentais de um problema e omite os demais (Rumbaugh,1994).

Para se sair da realidade e se chegar ao software, a abstração poderá ser realizada em níveis, para que a complexidade possa ser abordada em etapas e para que os trabalhos possam ser sistematizados.

Os níveis de abstração mais utilizados como referência são (Setzer,1986) :

- **Descritivo.** O Modelo Descritivo é caracterizado pela coleta de informações informais (exemplo : texto em português).

- **Conceitual.** O Modelo Conceitual é caracterizado pela formalização das informações, e pelo uso de regras de manipulação (Modelo em Diagrama DFD, E-R, (explicados mais adiante)).

- **Operacional ou Lógico.** O Modelo Operacional é caracterizado pela definição da estrutura externa dos dados, e pela definição das formas de manipulação dos dados (Modelo Relacional, Diagrama de Módulos, Português Estruturado (explicados mais adiante)). Deve ficar clara a diferença entre as informações formais do nível conceitual e entre as estruturas externas dos dados definidas no nível operacional. As primeiras podem seguir qualquer formalismo matemático, podendo existir no papel ou em nossa mente. As segundas devem ser expressas de tal forma que o computador possa recebê-las e tratá-las.

- **Físico.** O Modelo Físico é caracterizado pela implementação (arquivos, programas).

A Figura 2.4 mostra os níveis de abstração da realidade mais utilizados e sua correspondência com as fases do ciclo de vida de um software.

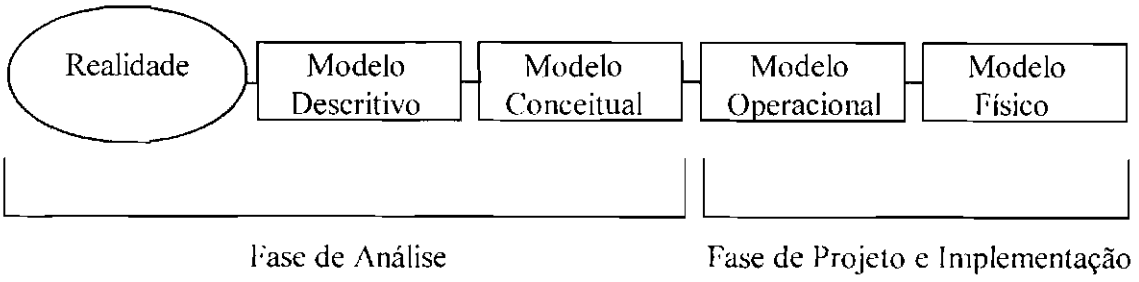


Fig. 2.4 - Níveis de Abstração X Fases do Ciclo de Vida.
FONTE: Adaptada de Nakanishi (1995A, p. 3.8).

Além disso, para a criação de software podemos ver o mundo real sob a óptica de funções, dados e eventos. Por exemplo :

- **Funções.** Calcular a órbita de um foguete, cadastrar os novos funcionários de um departamento, atualizar os dados recebidos de um satélite.

- **Dados.** Cadastro de dados sobre órbitas de foguetes já lançados, cadastro de dados sobre os funcionários de um departamento, cadastro dos dados recebidos por um satélite.

- **Eventos.** Recebimento de uma ordem de cadastramento dos funcionários novos, recebimento de uma ordem de produção numa fábrica, recebimento de uma solicitação de troca de produto numa loja, emissão de uma carta de cobrança.

Isto é, podemos ver o mundo real em termos das funções que o sistema deverá realizar, dos dados que ele utilizará, ou dos eventos que poderão influenciá-lo. A Figura 2.5, mostra a relação dos Modelos de Abstração e das Ópticas de desenvolvimento de softwares.

Na maioria das metodologias para desenvolvimento de softwares, entre a modelagem de dados, a modelagem de eventos, e a modelagem de funções do sistema, existe um grande degrau; pois elas modelam cada óptica separadamente. Isso na verdade não deveria

acontecer, pois num sistema, os dados que ele possui estão intimamente ligados com as funções que ele realiza e com os eventos que o influenciam. Então, o ideal seria se ter metodologias onde a Figura 2.5 fosse representada por um cilindro, mostrando uma passagem suave de um modelo para o outro.

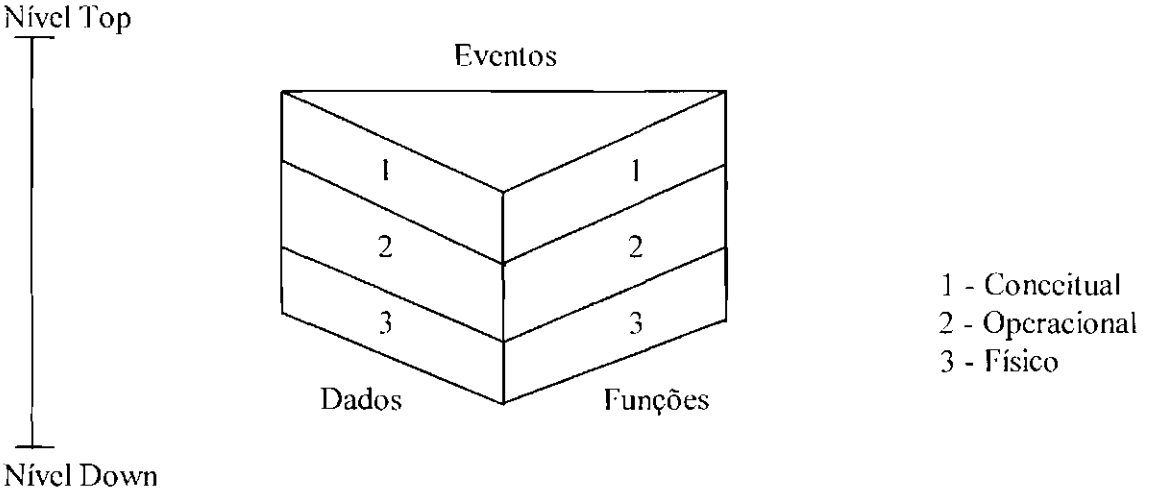


Fig. 2.5 - Modelos de Abstração e as Ópticas de Desenvolvimento de Softwares.
FONTE: Nakanishi (1995B).

2.10 - Paradigmas da Engenharia de Software

As técnicas para desenvolvimento de softwares baseiam-se no princípio de "dividir para conquistar". Esse princípio, nada mais é que dividir problemas complexos em problemas menores para resolvê-los por partes.

Existem inúmeras técnicas que podem ser adotadas, e quais delas escolher e como usá-las irá depender do tipo de software que se deseja desenvolver. Além disso, essas técnicas podem compor diversas metodologias de desenvolvimento de softwares, e a maneira como elas serão empregadas dependem da metodologia escolhida.

As técnicas para desenvolvimento de software podem ser enquadradas dentro do Paradigma Clássico, ou dentro do Paradigma da Orientação a Objetos.

2.10.1 - Paradigma Clássico

As técnicas enquadradas no Paradigma Clássico podem ser ainda subdivididas em Técnicas Estruturadas e Técnicas não Estruturadas.

2.10.1.1 - Técnicas Estruturadas

Nas técnicas estruturadas pode-se aplicar o princípio de dividir para conquistar recursivamente quebrando partes em partes ainda menores, como mostra a Figura 2.6.

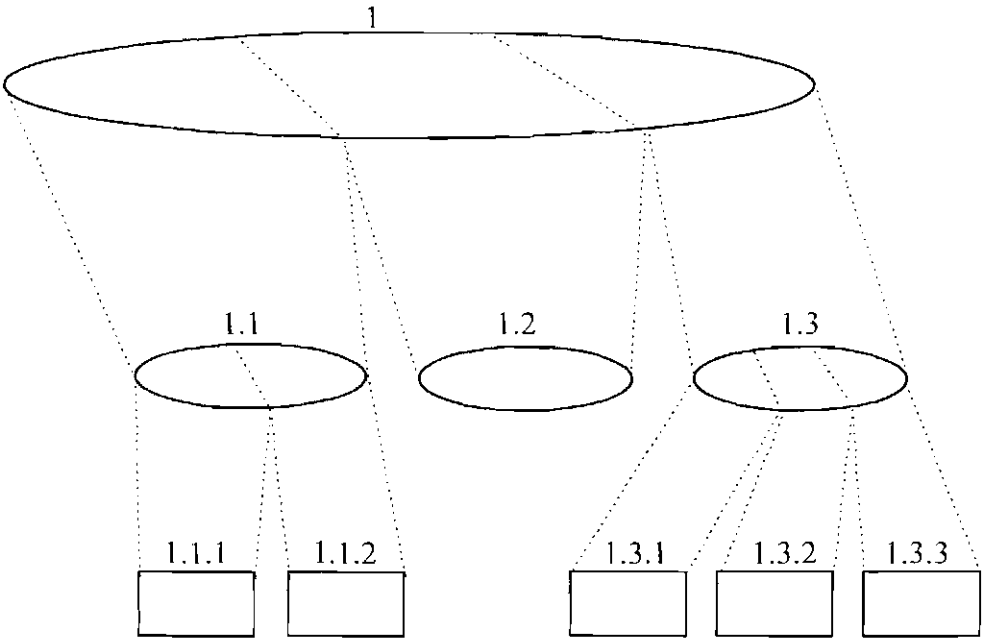


Fig. 2.6 - Técnicas Estruturadas e o Princípio de Dividir para Conquistar.
FONTE: Nakanishi (1995B).

Ao se fazer isso, porém, deve-se estar atento se as partes juntas realmente compõe o problema inicial. Além disso, deve-se verificar também se as interfaces entre as partes são simples, e não estão aumentando a complexidade do problema ao invés de reduzi-la.

Por exemplo, suponhamos que temos um problema matemático complexo para resolver. Para simplificar sua resolução, nós o dividimos em problemas menores e mais simples e resolvemos um de cada vez. Porém, para que o problema inicial tenha realmente sido resolvido, é necessário que os problemas menores juntos realmente representem o problema maior, sem complicá-lo e sem faltar nada; só assim poderemos garantir que a solução do problema complexo proposto é formada pelo conjunto de soluções dos problemas menores.

As técnicas estruturadas estabelecem a melhor maneira de quebrar o problema em partes, e estabelecem também um limite para a recursividade; garantindo assim, que as partes componham o todo e que as interfaces entre as partes sejam simples. Dentre as técnicas estruturadas podemos citar :

- **Diagrama de Fluxo de Dados (DFD).** É a principal técnica da análise estruturada. À medida que se movimenta pelo software, a informação é modificada por uma série de transformações. Um diagrama de fluxo de dados (DFD) é uma técnica gráfica que descreve o fluxo de informação e as transformações que são aplicadas à medida que os dados se movimentam da entrada para a saída. A forma básica de um diagrama de fluxo de dados, segundo a notação de Gane (1983), é ilustrada na Figura 2.7.

O DFD é composto de Entidades Externas (Um produtor ou consumidor de informações que resida fora dos limites do sistema a ser modelado), de Processos (Um transformador de informações que resida dentro dos limites do sistema a ser modelado), Fluxos de Dados (Informações de Entrada, Informações de Saída, Dados para atualização do depósito de dados, e Dados para serem processados) e Depósitos de Dados (Um repositório de dados que são armazenados para serem usados em um ou mais processos).

O diagrama de fluxo de dados pode ser usado para representar um sistema em qualquer nível de abstração. De fato, os DFDs podem ser divididos em partições de

acordo com níveis que representem um crescente detalhamento funcional e do fluxo de informação. Isto é, cada Proceso de um DFD pode ser expandido para tornar-se um novo DFD que representa um nível de abstração com maiores detalhes (DFD pai de DFDs filhos), como mostra a Figura 2.8. É justamente por essa possibilidade de se ter uma expansão *top-down* que o DFD é considerado uma técnica estruturada.

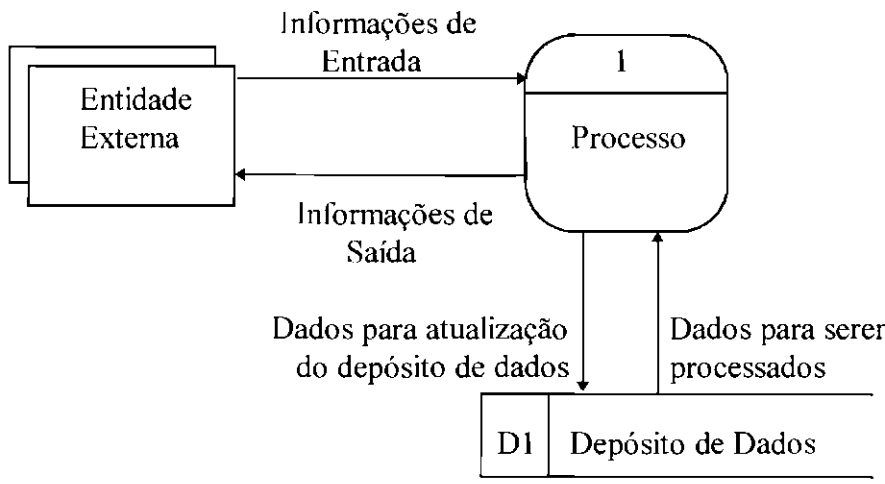


Fig. 2.7 - Diagrama de Fluxo de Dados.

- **Diagrama de Estrutura dos Módulos.** É a principal técnica do projeto estruturado. Ele é, nada mais nada menos, que um gráfico que ilustra a segmentação de um sistema em módulos, mostrando a hierarquia, organização e comunicação entre eles.

Um módulo é uma parte do sistema que será transformada em um conjunto de instruções de programa. Ele é caracterizado por quatro atributos básicos :

1) Entrada e Saída. Informações que o módulo necessita e fornece.

2) Função. O que o módulo faz para a entrada produzir a saída.

3) Lógica. Procedimentos ou algoritmos que executam a função.

4) Dados internos. Dados que somente o módulo referencia na sua área de trabalho.

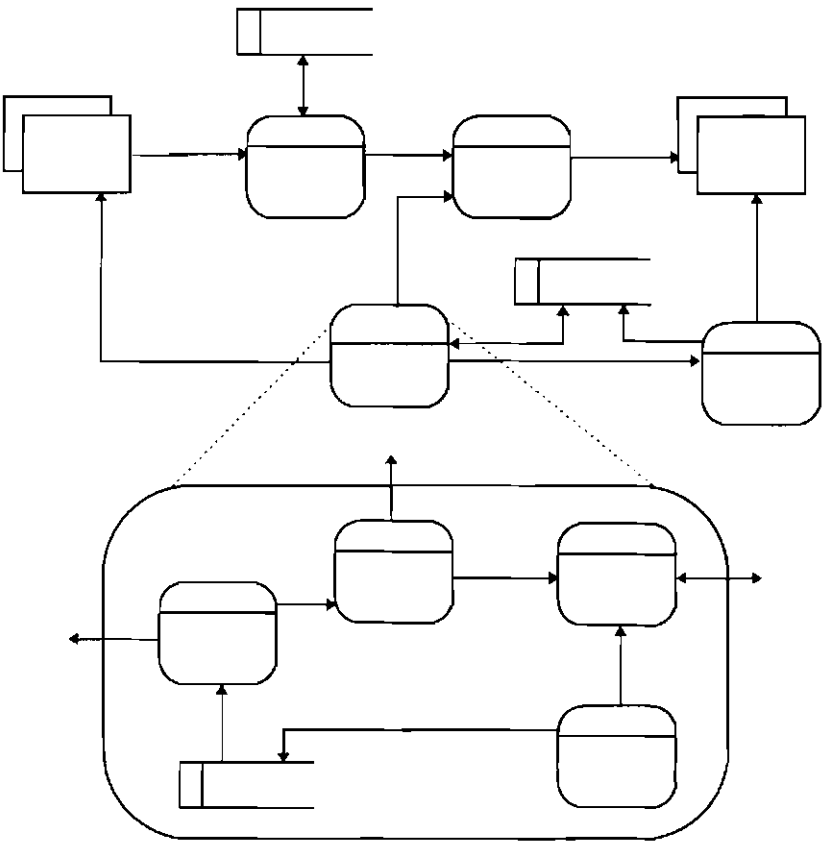


Fig. 2.8 - DFD e sua Expansão *Top-Down.*

O módulo também tem outros atributos, como por exemplo, o nome pelo qual ele é referenciado. Existem duas ópticas para se enxergar um módulo : a visão externa, e a visão interna. A visão externa é composta dos atributos entrada/saída, função e nome. A visão interna é formada pelos atributos lógica e dados internos. Cada módulo será posteriormente transformado em uma unidade de programa, ou seja, um *programa principal*, uma *sub-rotina*, uma *procedure*, etc...

O Diagrama de Estrutura dos Módulos, assim como o DFD, é considerado uma técnica estruturada porque ele pode representar vários níveis de abstração do sistema, já que um módulo pode ser ainda mais detalhado em seus módulos filhos, como mostra a Figura 2.9, que usa a notação de Page-Jones (1988).

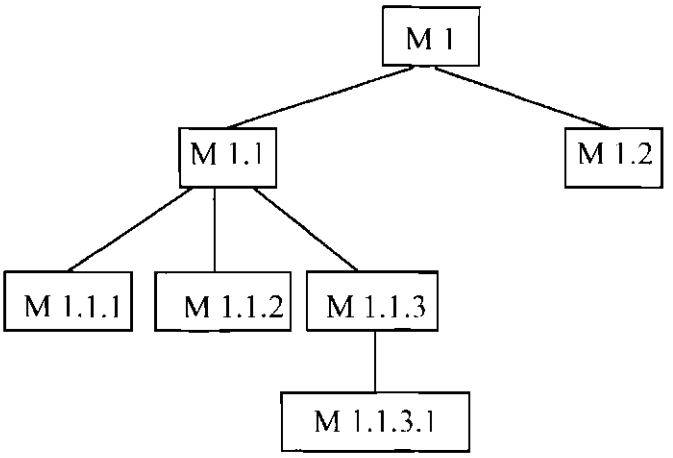


Fig. 2.9 - Diagrama de Estrutura dos Módulos.

2.10.1.1.1 - Metodologias Estruturadas

As metodologias são na verdade um conjunto de técnicas, e as metodologias Estruturadas são aquelas que usam como base uma (ou mais) técnica Estruturada, isto é, dizer que uma metodologia é Estruturada não quer dizer que todas as técnicas que a compõe sejam Estruturadas, quer dizer apenas que pelo menos a técnica principal que a compõe é Estruturada.

Dentre as principais metodologias Estruturadas podemos citar as metodologias de Análise e Projeto Estruturado Clássico, e algumas metodologias de Análise e Projeto Orientados a Eventos.

As metodologias de Análise e Projeto Estruturado Clássico enfocam as funções que o software realiza, dando pouca atenção à estrutura dos dados. Dentre elas podemos citar as metodologias de Yourdon, Constantini, De Marco, Gane, etc...

As metodologias de Análise e Projeto Orientados a Eventos enfocam como o sistema responde aos eventos externos. Dentre elas podemos citar as metodologias de Stephen Mc Menamin e John Palmer, Stephen Mellor, Paul Ward, etc...

As metodologias Estruturadas normalmente apresentam um grande degrau entre a análise e o projeto do sistema.

2.10.1.2 - Técnicas não Estruturadas

As técnicas não estruturadas não utilizam o princípio de dividir para conquistar de maneira recursiva; e portanto não abordam o problema em diferentes níveis de abstração. Elas propõem apenas uma única quebra do problema em partes, o que significa que todos os detalhes são abordados de uma só vez. Dentre as técnicas não estruturadas podemos citar :

- **Diagrama Entidade-Relacionamento (E-R).** O Diagrama de Entidade-Relacionamento é uma técnica de modelagem de dados. A modelagem de dados considera os dados independentemente dos processos que os transformam; e por isso o Diagrama de Entidade-Relacionamento concentra-se apenas nos dados, representando uma "rede de dados" de um determinado sistema. Isto é, ele é uma ferramenta gráfica que representa os relacionamentos entre os conjuntos de entidades do sistema. Sua representação básica segundo a notação de Setzer (1986) é dada na Figura 2.10 abaixo.

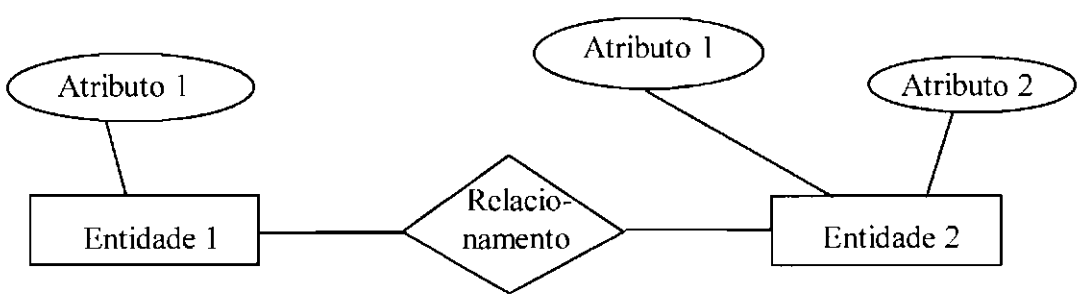


Fig. 2.10 - Representação Básica do Diagrama de Entidade-Relacionamento.

Entidades são representações abstratas de objetos, seres, e organizações do mundo real; que precisam ter dados a seu respeito armazenados para que o sistema consiga cumprir seus objetivos. As entidades podem representar também coisas não concretas como por exemplo eventos, desde que para satisfazer os objetivos do sistema deva-se armazenar dados sobre eles. Os dados que armazenamos sobre uma entidade são chamados de "atributos" dessa entidade.

O relacionamento nada mais é que uma estrutura abstrata que indica as associações entre os elementos de um conjunto de entidades, com elementos de um outro conjunto de entidades.

O diagrama de entidade-relacionamento é considerado uma técnica não estruturada porque ele aborda o problema em apenas um nível de abstração, isto é, os detalhes não são abordados gradativamente como nas técnicas estruturadas.

2.10.1.2.1 - Metodologias não Estruturadas

As metodologias não Estruturadas são aquelas que não tem como base uma (ou mais) técnica Estruturada, e dentre as metodologias não Estruturadas, podemos citar a metodologia de Jackson, para análise, projeto e implementação de sistemas de tempo real.

A metodologia de Jackson divide o desenvolvimento de softwares em duas etapas: implementação e especificação. Ela não faz distinção entre análise e projeto do software, em vez disso, reúne as duas fases como especificação.

Um modelo nessa metodologia descreve o mundo real em termos de entidades, ações e ordenação de ações.

A metodologia de Jackson é orientada principalmente para aplicações onde o tempo é importante, isto é, softwares de tempo real, pois a modelagem é bastante detalhada e se concentra no tempo; para softwares concorrentes, onde os processos devem estar sincronizados entre si; para programação de computadores em paralelo; e para microcódigo. Porém, ela é inadequada para (Rumbaugh,1994) :

- análise de alto nível, pois não fornece o amplo entendimento de um problema, sendo ineficiente para abstração e simplificação, pois manipula meticulosamente os detalhes, mas não auxilia o desenvolvedor a dominar a essência do problema;

- banco de dados, porque o projeto de banco de dados é um tópico mais complexo do que a metodologia Jackson sugere, já que a metodologia Jackson é orientada às ações e distancia-se de entidades e atributos, e

- software convencional rodando sob um *sistema operacional*, pois a abstração de centenas ou milhares de processos na metodologia Jackson é confusa e desnecessária.

2.10.2 - Paradigma da Orientação a Objetos

No Paradigma Clássico a informação é dividida em dois tipos distintos: funções e dados. No Paradigma da Orientação a Objetos isso não acontece, pois um software baseado em objetos é organizado como uma coleção de objetos separados que incorporam tanto o comportamento funcional, quanto os dados manipulados.

Para entender o termo "orientado para objetos", consideremos um exemplo de objeto do mundo real, um Boing 747. **Boing 747** é um membro (ou instância) de uma classe muito maior de objetos, a classe **Avião**. Um conjunto de atributos genéricos pode ser associado a cada objeto da classe **Avião**. Por exemplo, todo avião tem um **Número de Identificação**, **Modelo**, **Posição**, **Estado** (Em uso ou não), entre muitos outros

possíveis. Esses atributos se aplicam, quer se esteja falando de um Avião Monomotor ou de um Boing 747, de um Helicóptero ou de um Planador. Uma vez que **Boing 747** é um membro da classe **Avião**, **Boing 747** herda todos os atributos definidos para a classe, como ilustra a Figura 2.11 a seguir.

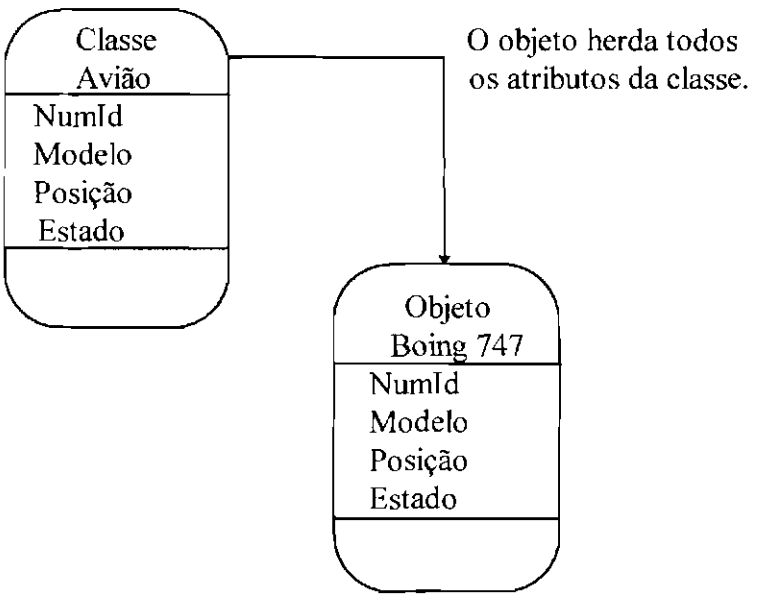


Fig. 2.11 - Herança de Classe para Objeto.

Logo que a classe é definida, os atributos podem ser reusados quando novas instâncias da classe são criadas. Por exemplo, suponhamos que fôssemos definir um novo objeto chamado **Avião Monomotor** que seja membro da classe **Avião**. **Avião Monomotor** herda todos os atributos de **Avião**.

Todo objeto da classe **Avião** também pode ser manipulado de diversas maneiras. Ele pode ser escalado para um vôo, pode ser liberado, pode mudar sua posição. Cada uma dessas operações (ou serviços) modificará um ou mais atributos do objeto. Por exemplo, se o atributo **Posição** for composto de **Latitude + Longitude + Altitude**, então uma operação denominada **Mudar**, modificará um ou mais dos itens de dados (latitude, longitude, altitude) que compreendem o atributo **Posição**. Para fazer isso, **Mudar** deve ter "conhecimento" desses itens de dados. A operação **Mudar** poderia ser usada para um **Boing 747** ou um **Avião Monomotor**, visto que ambos são instâncias da classe **Avião**.

Todas as operações válidas (por exemplo, **Comprar, Vender, Pesar**) para a classe **Avião** estão ligadas à definição de objeto, e são herdadas por todas as instâncias da classe como mostra a Figura 2.12 a seguir.

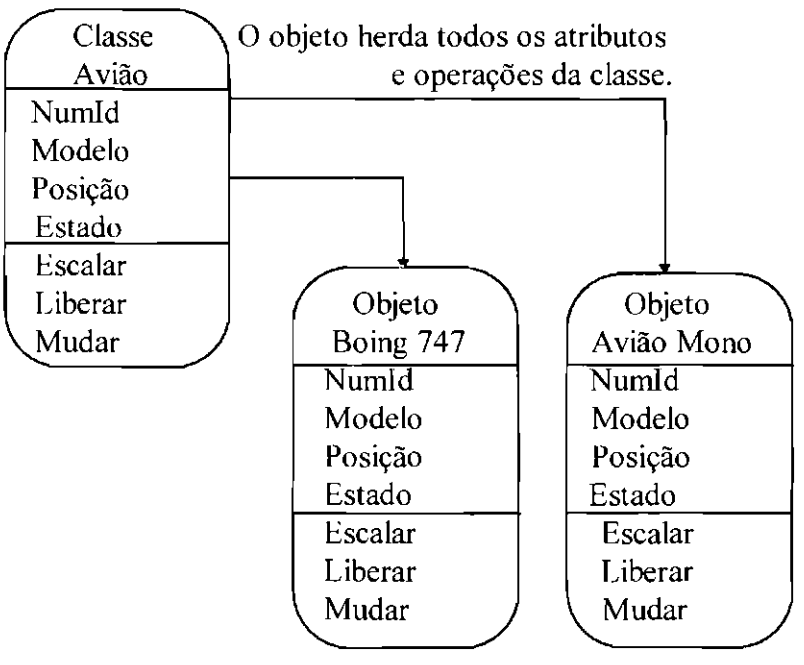


Fig. 2.12 - Herança de Operações da Classe para o Objeto.

O objeto **Boing 747** (e todos os objetos em geral) encapsula dados (os valores dos atributos que definem **Boing747**), operações (as ações que são aplicadas para mudar os atributos de **Boing 747**), outros objetos (objetos compostos), constantes (valores estabelecidos) e outras informações correlatas.

Agora que foram introduzidos alguns dos conceitos básicos, uma definição mais formal de "orientado para objetos" se demonstrará mais significativa. Em Coad (1992) tem-se :

Orientado para Objetos = Classes e Objetos + Herança + Comunicação

Podemos dizer que um objeto é uma abstração de alguma coisa em um domínio de problemas, exprimindo as capacidades de um sistema de manter informações sobre ela,

interagir com ela, ou ambos; um encapsulamento de valores de atributos e de seus serviços exclusivos (Coad,1992).

Os objetos que compartilham de um mesmo conjunto integrado de atributos e serviços compõe uma classe, isto é, podemos dizer que uma classe é a definição de um conjunto de objetos quase idênticos, onde os objetos são instâncias desta classe (Coad,1992).

Na verdade, a abordagem orientada para objeto gerencia a complexidade do mundo real abstraindo conhecimento dele, e encapsulando esse conhecimento em objetos. Ela concentra-se em achar os objetos do sistema e suas conexões; e para fazer isso, ela verifica quais operações devem ser realizadas, e quais informações resultam dessas operações, e atribui a responsabilidade por tais operações e informações para os objetos. Cada objeto sabe, portanto, como realizar suas próprias operações e se lembra de informações a seu respeito.

Isso significa que a abordagem orientada para objeto decompõe o sistema em entidades que sabem como executar seus papéis no sistema, ou seja, elas sabem como ser elas mesmas. Porém, o que significa saber ser ela mesma ? Claramente tal conhecimento envolve tanto funções como dados. Objetos sabem certos dados a seu respeito e sabem como realizar certas funções. Ou seja, objetos podem mandar mensagens para outros objetos, executar um serviço quando uma mensagem é recebida de outros objetos, e manter dados sobre si mesmo.

Na verdade, a abordagem orientada para objetos encoraja uma visão do mundo como um sistema de agentes cooperadores. O trabalho em um sistema orientado para objeto é iniciado por um objeto enviando uma requisição para outro objeto, pedindo a ele que realize uma de suas operações, ou que ele revele algumas de suas informações. A primeira requisição encadeia uma longa e complexa cadeia de requisições, fazendo com que os objetos sejam agentes de seus próprios sistemas (Wirfs-Brock,1990).

A tecnologia baseada em objetos está fundamentada em características como :

- **Abstração.** No desenvolvimento de sistemas baseados em objetos isso significa concentrar-se no que um objeto é e faz, antes de decidir como ele é implementado. O uso da abstração preserva a liberdade de se tomar decisões, evitando tanto quanto possível, comprometimentos prematuros com detalhes (Rumbaugh,1994).

- **Encapsulamento.** O encapsulamento é nada mais nada menos que um ocultamento de informações. O conhecimento ou inteligência encapsulado em um objeto possibilita a separação dos seus aspectos externos (que são acessíveis por outros objetos), dos detalhes internos da implementação daquele objeto (que ficam assim ocultos dos demais objetos).

Pode-se dizer então, que os objetos têm um lado particular que sabe como realizar operações; fazendo com que "como" ele realiza tais operações não diga respeito a outras partes do sistema. Dessa maneira, os objetos são livres para mudar seus lados particulares sem afetar o resto do sistema (Wirfs-Brock,1990). Então, como ilustrado na Figura 2.13 a seguir, um objeto encapsula inteligência, que é representada no software por dados e processos.

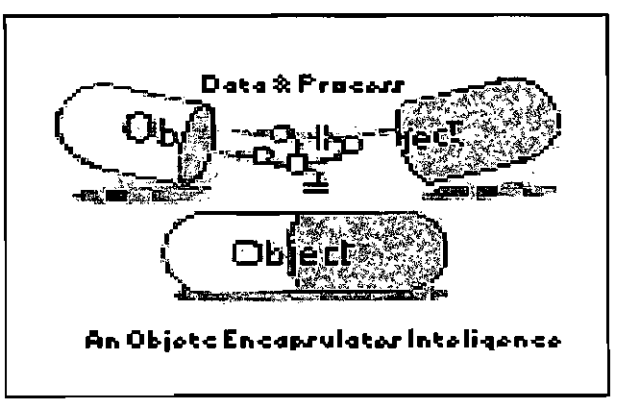


Fig. 2.13 - Um Objeto Encapsula Inteligência.

FONTE : Cohen (1993, p. 61).

O encapsulamento portanto, permite que um modelo de objetos seja constituído de objetos que são planejados de maneira que cada objeto seja visto de fora, e de maneira que cada um seja entendido em termos de seu procedimento intríseco, que é expresso em termos dos serviços que ele provém (Cohen,1993).

• **Compartilhamento.** Esse princípio dirige a criação de estrututras de herança onde procedimentos e dados que são compartilhados por um grupo de classes são descritos apenas uma vez, numa classe que é um antecessor comum para cada classe no grupo.

As técnicas baseadas em objetos promovem o compartilhamento em diversos níveis. A herança da estrutura de dados e do seu comportamento permite que a estrutura comum seja compartilhada por diversas subclasses semelhantes sem redundâncias. O compartilhamento de código com utilização de herança também é possível, e é uma das principais vantagens das linguagens baseadas em objetos. Além disso, o desenvolvimento baseado em objetos oferece a possibilidade de reutilização de modelos e códigos em projetos futuros (Rumbaugh,1994).

• **Ênfase na Estrutura de Objetos e Não na Estrutura de Procedimentos.** A tecnologia baseada em objetos preocupa-se em especificar o que um objeto é, e não como ele é utilizado. Os usos de um objeto são altamente dependentes dos detalhes de aplicação e freqüentemente mudam durante o desenvolvimento. À medida que os requisitos evoluem, as características de um objeto permanecem muito mais estáveis do que os modos como ele é utilizado, e por causa disso, os softwares construídos com base na estrutura de objetos são mais estáveis a longo prazo (Rumbaugh,1994).

• **Polimorfismo.** É a habilidade de adquirir mais de uma "forma". Um atributo de um objeto pode ter mais de um conjunto de valores, e uma operação pode ser implementada por mais de um serviço. Então, com o polimorfismo uma mensagem recebida por um objeto pode resultar em ações diferentes por parte desse objeto,

fazendo com que o objeto que enviou a mensagem não precise se preocupar com detalhes de implementação do receptor.

Um exemplo comum de polimorfismo vem da computação gráfica. Nesse exemplo, serviços de desenho diferentes podem ser implementados por uma operação "Desenhar". O serviço "Desenhar" (por exemplo, Desenhar_Arco, Desenhar_Retângulo) apropriado ao objeto a ser desenhado, é acionado em tempo de execução do programa (*Run Time*) (Kung,1995).

- **Incorporação Dinâmica.** Significa que o serviço que implementa uma operação só é conhecido em tempo de execução do programa (*Run Time*). Esse mecanismo possibilita a implementação do polimorfismo.

2.10.2.1 - Técnicas e Metodologias Orientadas para Objetos

As técnicas enquadradas no Paradigma da Orientação a objetos ainda não foram subdivididas nem classificadas, pois constituem ainda um campo novo de observação, necessitando para tal serem melhor estudadas e analisadas. Dentre elas podemos citar :

- **Diagrama de Classe&Objeto.** Um diagrama de Classe&Objeto é uma técnica gráfica que mostra as Classes&Objetos de um modelo de objetos, pode ou não mostrar seus respectivos atributos e serviços; mostra as relações de herança existentes entre as classes (relações de Generalização-Especialização), e as conexões (Conexões Todo-Parte e Associações) existentes entre os objetos.

O Diagrama de Classe&Objeto fornece apenas uma visão estática do sistema, sua forma básica, segundo a notação de Coad (1997), é ilustrada na Figura 2.14. Ele pode ser dividido em dois níveis : Diagrama de Classe&Objeto do Modelo Conceitual, e Diagrama de Classe&Objeto do Modelo Lógico.

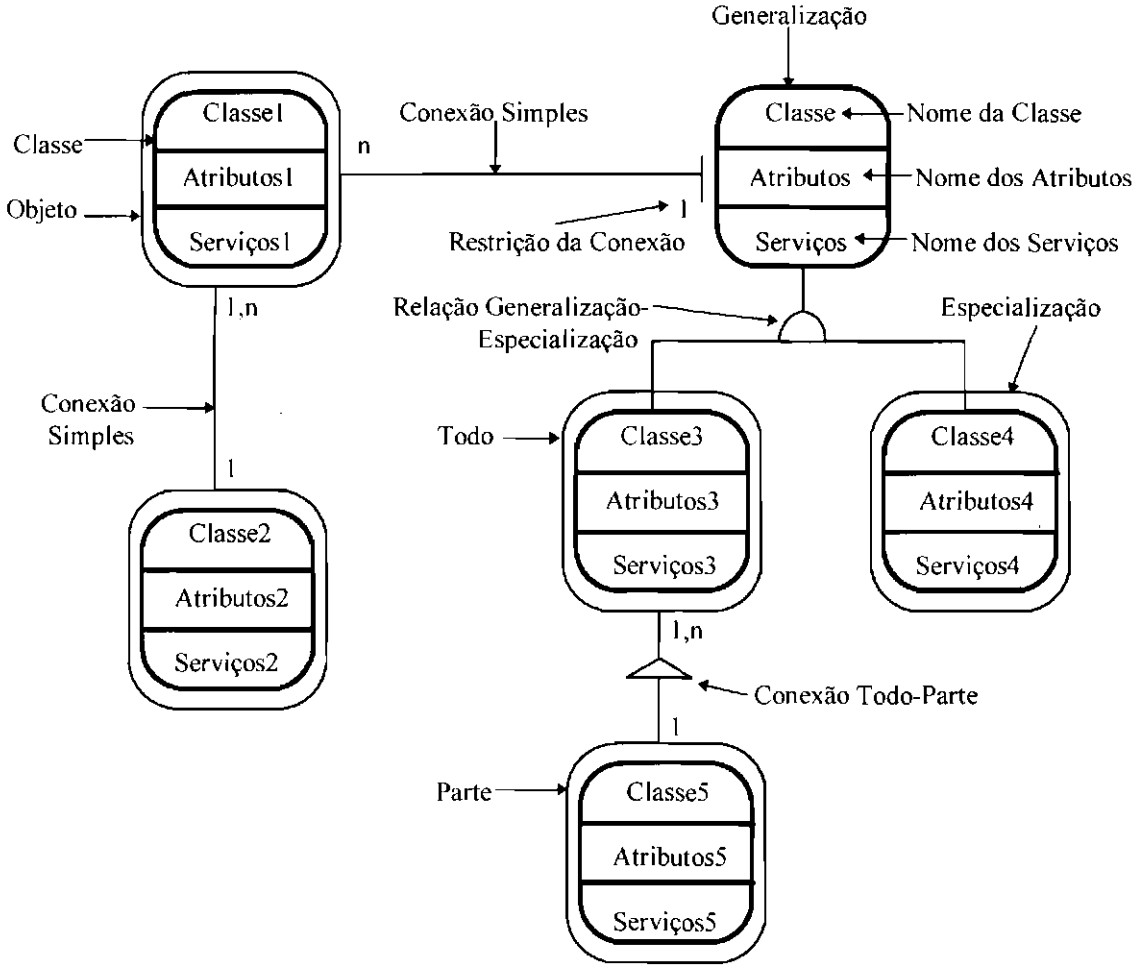


Fig. 2.14 - Diagrama de Classe&Objeto.

• **Cenário.** Um cenário é uma descrição de objetos e de uma seqüência de eventos relacionadas a eles, que acontecem para que se cumpra uma determinada necessidade. Cada cenário, portanto, especifica as interações entre objetos ordenadas no tempo que são requeridas para responder aos eventos relacionados a uma determinada habilidade do sistema sob consideração. Cada seqüência de interações geradas por um evento dentro de um cenário é chamada "Cena" (Dai,1997).

Ao contrário dos Diagramas de Classe&Objeto, os cenários nos dão uma visão da dinâmica do sistema. Cada cenário pode ser desenvolvido e descrito com um

"Roteiro de Cenário", ou com "Visões de cenário". Fazer um roteiro de cenário significa identificar para cada cena de um cenário, os serviços emissores e receptores de mensagens numa seqüência ordenada no tempo.

A visão de cenário é uma técnica gráfica que mostra os objetos que participam de uma determinada cena de um cenário, seguidos de uma seqüência ordenada no tempo de serviços emissores, setas de mensagens, serviços receptores e *argumentos* (entradas, saídas). O nome do cenário e da cena também são representados. A notação gráfica para uma Visão de cenário, segundo a notação de Coad (1997), é mostrada na Figura 2.15.

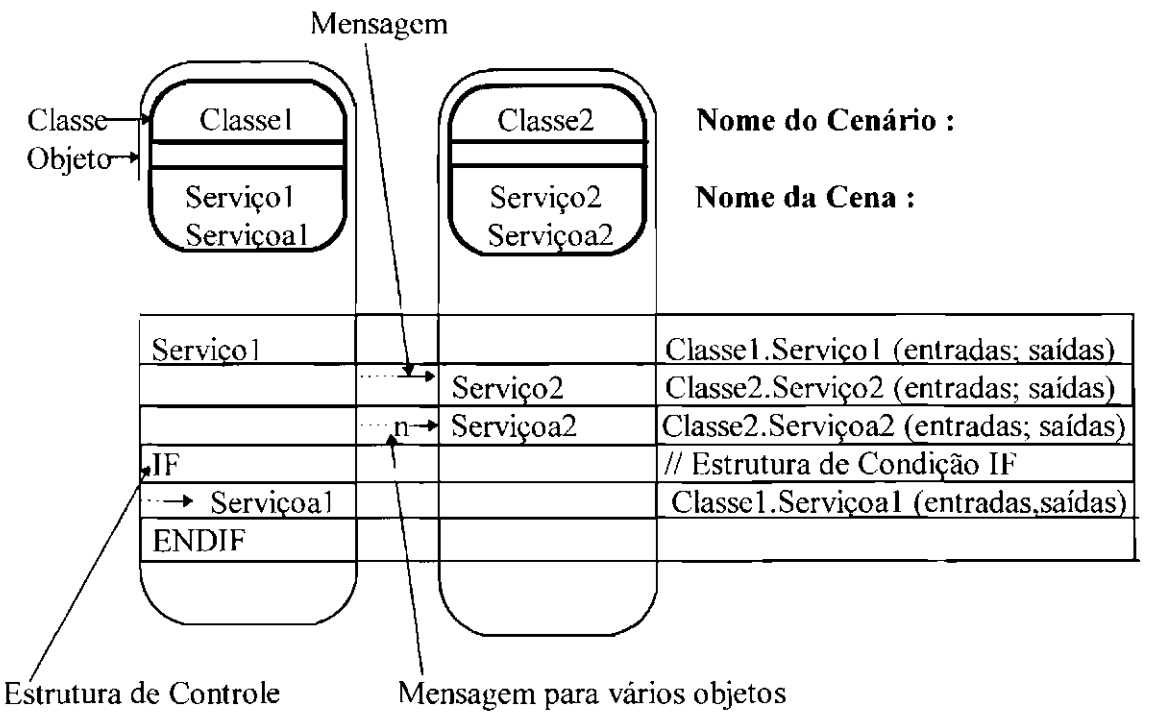


Fig. 2.15 - Notação para uma Visão de Cenário.

As várias técnicas Orientadas para Objetos existentes compõe as diversas metodologias Orientadas para Objetos. Essas metodologias estudam o desenvolvimento de softwares utilizando modelos fundamentados em conceitos do mundo real e conceitos abstratos

criados pelo projetista, onde a estrutura básica é o objeto. Dentre as metodologias Orientadas para Objetos, podemos citar as metodologias de Booch, Coad & Yourdon, Rumbaugh, Jacobson etc....

As metodologias orientadas para objetos têm um enfoque voltado para os dados que o objeto contém. Esses dados são manipulados por serviços dos próprios objetos. Esse tipo de abordagem suaviza bastante os degraus entre análise, projeto e implementação, geralmente bem acentuados nas metodologias tradicionais. Além disso, o enfoque de dados produz resultados mais estáveis, pois os processos estão mais suscetíveis a mudanças do que os dados.

Portanto, o desenvolvimento baseado em objetos inverte as metodologias do Paradigma Clássico; onde a ênfase principal repousa na especificação e decomposição da funcionalidade do sistema. Tal abordagem pode parecer a maneira mais direta de implementar o que se pretende, mas o sistema resultante pode ser frágil. Se os requisitos mudarem, um sistema baseado na decomposição funcional pode exigir uma maciça restruturação (Rumbaugh,1994).

Em contraste, a abordagem baseada em objetos preocupa-se primeiro em identificar os objetos contidos no domínio da aplicação e depois em estabelecer os procedimentos relativos a eles. Embora isso possa parecer mais indireto, o software baseado em objetos mantém-se melhor à medida que os requisitos evoluem, por se apoiar na própria estrutura fundamental do domínio da aplicação, ao invés de apoiar-se nos requisitos funcionais de um único problema (Rumbaugh,1994).

Então, como o conceito de objeto engloba modularidade e encapsulamento, e as relações de herança facilitam as alterações e reutilizações, pode-se gerar softwares mais flexíveis para se alterar; valorizando a reutilização, a modularidade e o encapsulamento; e fazendo com que o degrau entre análise, projeto e implementação seja bem mais suave do que nas metodologias do Paradigma Clássico (Coad,1992).

2.11 - Por que Fazer Uso da Engenharia de Software

Como já dissemos anteriormente, a caracterização imprópria do projeto afeta indiscutivelmente a sua qualidade. A falta de coleta de dados e de requisitos do usuário gera definições pobres, e conseqüentemente o produto gerado terá menos chances de corresponder às expectativas do usuário. Além disso, uma definição pobre das fronteiras, restrições e limitações do sistema; e uma análise imprópria da viabilidade do projeto e da relação Custo X Benefício, pode fazer com que o projeto fique impedido de seguir adiante numa fase já adiantada, causando um desperdício enorme de tempo e investimento.

É muito importante lembrar que quanto mais avançado o projeto, maior o impacto no custo proporcionado por modificações, pois, quanto mais tardias, as modificações consomem mais tempo e capital para serem introduzidas. Para ilustrar o impacto de custo da detecção de erros feita tardiamente, consideremos uma série de custos reais compilados de grandes projetos de software. Suponhamos que um erro descoberto durante a fase de projeto custe 1,0 unidade monetária para ser corrigido. Em relação a esse custo, o mesmo erro, descoberto logo antes que as atividades de teste se iniciem, custará 6,5 unidades; durante os testes, 15 unidades; e, após o lançamento, entre 60 e 100 unidades (Pressman,1995).

Por isso, é importante uma definição formal e detalhada dos procedimentos, interfaces, limitações e critérios de validação; e é justamente aí que a Engenharia de Software vem nos ajudar, oferecendo metodologias, técnicas e ferramentas que nos orientarão nas etapas do ciclo de vida do software; permitindo que tenhamos softwares de boa qualidade, isto é, softwares que estejam dentro das expectativas do usuário, que operem com economicidade de recursos, que sejam fáceis de se manter e de alterar, tudo isso dentro dos custos e cronograma esperados.

Vejamos na Figura 2.17 a seguir, a relação de impacto do custo de uma mudança em relação ao tempo de projeto.

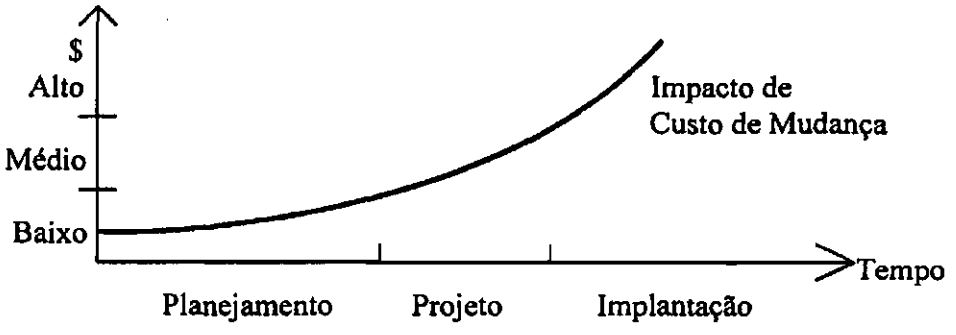


Fig. 2.17 - Impacto do Custo de Mudança X Tempo de Projeto.
FONTE: Nakanishi (1995B).

CAPÍTULO 3

SISTEMAS DE PEQUENO PORTE

O que é um sistema de pequeno porte ? Não é uma pergunta fácil de se responder. A verdade é que não existem regras precisas para a classificação de sistemas, já que ela depende muitas vezes do ponto de vista do analista. Isto é, pode haver casos em que um sistema seja classificado como um sistema de pequeno porte por um dado analista; e que o mesmo sistema, seja classificado como um sistema de médio porte por um outro analista. Essas variações acontecem principalmente nos sistemas que se encontram na fronteira entre o pequeno e o médio porte.

Em geral, os sistemas de pequeno porte são constituídos por poucos módulos de programa (para mais detalhes sobre módulos de programa ver o item **2.10.1.1 - Técnicas Estruturadas** , no **Capítulo 2 - Software e Engenharia de Software**, desta dissertação), e a equipe de desenvolvimento é pequena, sendo que, freqüentemente, ela é formada por uma única pessoa.

Apesar disso, não podemos simplesmente associar um sistema de pequeno porte a um sistema que seja de tamanho pequeno, pois a noção de pequeno porte está mais relacionada à complexidade de seu desenvolvimento. Na verdade, diversos fatores devem ser levados em conta ao se mensurar um sistema; onde o tamanho é apenas um deles.

A Tabela 3.1 é uma tabela de análise preliminar que pode ser usada para guiar a classificação do porte dos sistemas de software. Ela está organizada de acordo com três níveis de complexidade das características comumente encontradas nos sistemas de informação.

TABELA 3.1 - ANÁLISE PRELIMINAR PARA A CLASSIFICAÇÃO DO PORTE DE UM SISTEMA DE SOFTWARE

Característica	**Baixo**	**Moderado**	**Alto**
1.Número de Funções a serem realizadas	Poucas (<10)	Média (de 10 a 30)	Muitas (acima de 30)
2.Inovação da Função	Aplicação Básica (Cadastros e Relatórios)	Similar aos sistemas existentes, mas com acréscimo de algumas funções	Nova abordagem para solução do problema (automatização dos processos)
3.Número de Usuários	1	(<10)	mais que 10
4.*Processos : On-Line* X *Batch*	*batch*	*on-line*	*on-line*
5.Tempo de Resposta	não crítico	Interativo, na ordem de segundos	tempo-real
6.Quantidade de dados armazenados	um único disco	dois ou mais discos	requer gerenciamento dos discos
7.Precisão dos dados	baixa	média	alta
8.Tamanho das Transações	pequenas	médias	grandes
9.Processamento Distribuído	não	não	sim
10.Acesso Remoto X Local	somente local	local e remoto	remoto
11.Tolerância ao Sistema ficar fora do ar	várias horas	alguns minutos	não pode ficar fora do ar
12.Segurança	nenhuma	moderada	alta
13.Necessidade de Manual do Usuário	não	não	sim
14.Nível de Cultura do Usuário	responsável e familiarizado com sistemas de informação	alguma familiarização	ignorante
15.Nível de Experiência dos Desenvolvedores	experiências com sistema similar e nas ferramentas a serem utilizadas	experiência com a ferramenta, mas não com a aplicação	nenhuma experiência

FONTE: Chioccarello (1997, p. 168).

Desse modo, para se fazer uma análise preliminar do porte de um sistema, cada característica especificada na tabela deve ser analisada em relação a este sistema, e classificada quanto ao seu nível de complexidade (baixo, moderado, alto). Por exemplo, se o "tempo de resposta" do processamento e/ou recuperação de dados for "não crítico", então o sistema, quanto a essa característica, será classificado como de "baixa complexidade". Ao se analisar posteriormente a classificação de todas as características do sistema em conjunto, pode-se ter uma idéia da complexidade deste sistema.

Uma outra tabela que pode ser usada para guiar a classificação do porte dos sistemas é apresentada abaixo.

TABELA 3.2- CATEGORIAS DE TAMANHO DE SOFTWARES

Categoria	**Número de Desenvolvedores**	**Duração**	**Tamanho do Produto (Linhas de Código)**
Trivial	1	1 - 4 semanas	500
Pequeno	1	1 - 6 meses	1.000 a 2.000
Médio	2-5	1 - 2 anos	5.000 a 50.000
Grande	5 -20	2 - 3 anos	50.000 a 100.000
Muito Grande	100 -1.000	4 - 5 anos	1 milhão
Extremamente Grande	2.000 - 5.000	5 - 10 anos	1 a 10 milhões

FONTE : Fairley (1985, p. 11).

Podemos dizer então, que um Sistema de Pequeno Porte é um sistema que tem pouca complexidade; isto é, um sistema cuja maioria de suas características tem um nível de complexidade baixo; e que portanto, o seu entendimento global como um todo e seus detalhes, podem ser mantidos na mente de uma única pessoa. Pode-se dividir os sistemas de pequeno porte em :

1) **Sistemas Triviais.** Um sistema trivial envolve um desenvolvedor trabalhando alguns dias ou algumas semanas, e resulta em um programa de menos de 500

instruções agrupadas em 10 a 20 *sub-rotinas*. Tais programas freqüentemente são softwares pessoais, são desenvolvidos para uso exclusivo do programador, e são usualmente descartados depois de alguns meses. Nesses sistemas há pouca necessidade de uma análise formal, elaboração de documentação de projeto, elaboração de plano de testes extensivo, documentação para os programas; contudo, até mesmo os sistemas triviais podem ser melhorados aplicando-se um pouco de análise, projeto sistemático, programação estruturada e testes metódicos (Fairley,1985).

2) Sistemas Pequenos. Requer um desenvolvedor trabalhando de um a seis meses, e resulta num produto contendo de 1.000 a 2.000 linhas de código distribuídas em talvez 25 a 50 *sub-rotinas*. Sistemas pequenos não tem interações com outros programas. Exemplos de sistemas pequenos incluem aplicações científicas escritas por engenheiros para resolver problemas numéricos, aplicações comerciais pequenas para manipulação de dados e geração de relatórios, projetos de estudantes escritos em cursos de *compiladores* ou *sistemas operacionais*. Um sistema pequeno requer pouca interação entre desenvolvedor e usuário. Técnicas e notações padronizadas, documentos padronizados, revisões sistemáticas de projeto devem ser aplicadas até mesmo nesses sistemas, porém o grau de formalidade pode ser menor que o requerido em sistemas maiores (Fairley,1985).

Portanto, mesmo durante o desenvolvimento de um sistema de pequeno porte é útil lançar mão de metodologias, técnicas e ferramentas da engenharia de software. Aplicações simples de pequeno porte podem freqüentemente usar caminhos simples para o seu desenvolvimento, como por exemplo, fazer uma definição do sistema antes de partir para o projeto (para mais detalhes sobre as etapas do desenvolvimento de um software ver item **2.6 - Ciclo de Vida do Software**, no Capítulo 2 desta dissertação), e mesmo assim obter um sistema com uma boa qualidade (Bell,1994).

Ao longo deste Capítulo forneceremos diretrizes para se construir um Sistema de Pequeno Porte usando técnicas estruturadas para o projeto, implementação e testes do sistema (para mais detalhes sobre técnicas estruturadas para o desenvolvimento de sistemas ver o item **2.10.1.1 - Técnicas Estruturadas**, no Capítulo 2 desta dissertação). Para a modelagem dos dados utilizaremos técnicas não estruturadas (para mais detalhes sobre técnicas não estruturadas para o desenvolvimento de sistemas ver o item **2.10.1.2 - Técnicas não Estruturadas**, no Capítulo 2 desta dissertação).Estaremos neste Capítulo, portanto, traçando uma rota que pode ser seguida por aqueles que queiram construir um sistema desse tipo.

A construção de Sistemas de Pequeno Porte usando o paradigma da orientação a objetos (para mais detalhes sobre o paradigma da orientação a objetos ver o item **2.10.2 - Paradigma da Orientação a Objetos**, no Capítulo 2 desta dissertação) será abordada no **Capítulo 4 - Construção do Sistema de Pequeno Porte Usando o Paradigma da Orientação a Objetos**, desta dissertação.

3.1 - Definição do Sistema

Antes de começar a construção propriamente dita, deve-se fazer uma definição do sistema a ser construído.

Fazer a definição do sistema consiste em construir o seu modelo descritivo, onde as informações podem ser representadas na forma de um texto informal; e em iniciar a construção do modelo conceitual do sistema, onde formaliza-se as informações obtidas no modelo descritivo (para mais detalhes sobre modelos de abstração do software ver o item **2.9 - Modelagem e Abstração**, no Capítulo 2 desta dissertação).

Porém, como começar a construir o modelo descritivo do sistema ? A seguir são apresentadas algumas diretrizes.

Comece identificando os propósitos do sistema. A declaração de propósito deve ser curta, de preferência uma única frase longa (Shiller,1992). Além disso, ela deve ser uma declaração completa sobre o sistema, e não deve incluir detalhes sobre as formas dos dados ou sobre a tecnologia utilizada para implantá-lo. Lembre-se de usar as palavras "ajudar", "facilitar", "suportar"; mantendo sempre a meta do sistema e os fatores críticos para que ele tenha sucesso em mente (Coad,1997). A declaração de propósito deve começar assim : "O propósito do sistema é ..."

Feita a declaração de propósito do sistema, deve-se continuar a construção do modelo descritivo perguntando a especialistas do domínio, de preferência àqueles que irão usar o sistema; como eles conduzem o seu trabalho (Coad,1997). Pergunte quais são os maiores problemas que eles enfrentam todas as vezes que vão executar seu trabalho, ou partes dele; e pense em maneiras de diminuir o impacto desses problemas. Pergunte também em quê o novo sistema poderá ajudá-los; ou seja, como o novo sistema poderá facilitar o seu trabalho. Peça para que eles dêem exemplos, pois isso o ajudará muito a compreender o domínio em questão.

Se você achar que estão sendo usados termos de maneira diferente, ou seja, que estão sendo usadas palavras diferentes para designar a mesma coisa, ou que está sendo usada a mesma palavra para designar coisas diferentes, ou até mesmo que está se dando significados diferentes para uma mesma palavra; construa um glossário usando uma tabela que contenha : o termo usado, sua definição no dicionário de língua portuguesa e sua definição no projeto.

Além disso, leia sobre o domínio do problema, e pense se você já conhece algum sistema de software que seja semelhante ao software que você está querendo desenvolver. Se você puder lembrar de algum outro sistema que se pareça com o seu, use-o, se possível. Estude como ele funciona e aprenda com suas qualidades e defeitos. Se julgar necessário pergunte aos usuários desse sistema o que eles acham dele, isto é, em que ponto eles o

acham falho, em que pontos eles o acham bom, quais facilidades ele oferece que eles julgam desnecessárias.

Finalmente, usando as informações que se obteve até agora, deve-se construir o modelo descritivo para o sistema. O texto deve conter uma descrição de como o sistema deverá funcionar, seus principais objetivos e metas, as informações que servirão de entrada para o sistema, assim como as informações de saída desejadas. É importante também, descrever um pouco do ambiente no qual o sistema deverá operar, e com quais outros sistemas (computadorizados ou não) ele irá interagir.

O próximo passo na definição do sistema é construir o Diagrama de Contexto. Ele define com o quê, ou quem o sistema faz interface, e qual o conteúdo dessa interface, ajudando portanto a se definir as fronteiras do sistema.

Para se construir o diagrama de contexto deve-se primeiramente modelar uma identificação para o sistema escolhendo um nome significativo para designá-lo. Uma vez que se tenha uma identificação aceitável, desenhe um grande círculo no meio de uma folha de papel em branco, escrevendo a identificação dentro do círculo.

Feito isso deve-se listar todas as entidades que fazem interface com o sistema. Ao tentar identificar uma entidade deve-se lembrar que ela deve ser uma fonte ou um receptor de dados. A lista de entidades provavelmente incluirá entidades que não estão apontadas na declaração de propósito, mas qualquer entidade referenciada nela tem que estar na lista de entidades. Uma vez completada a lista, envolva com caixas retangulares cada uma das entidades, e desenhe-as ao lado do círculo. Feito isso, para cada entidade da lista, escreva uma ou duas frases que descrevam a sua função. Se encontrar problemas para descrever a função de uma entidade, provavelmente ela está desempenhando muitas funções, portanto, divida-a em duas ou mais.

A última etapa da construção do diagrama de contexto é a adição dos fluxos de dados. Fluxos de dados devem ser adicionados entre as entidades e o círculo representativo do sistema, indicando quais dados são entradas para o sistema e quais dados são saídas para o sistema. Os fluxos de dados devem ser representados por uma linha com uma seta indicando o sentido do caminho. Um exemplo da notação usada para se construir o diagrama de contexto é dado na Figura 3.1 adiante.

Após a construção do diagrama de contexto deve-se continuar a definição do sistema definindo-se um modelo de eventos para o sistema. Esse modelo inclui a lista de eventos externos que causam reação do sistema, e o conjunto de reações do sistema para cada evento. Esteja certo de estar identificando eventos que são relevantes para o sistema em questão.

A fim de ilustrar e esclarecer as diretrizes para a construção de sistemas de software que são propostas nessa dissertação, adotou-se um sistema exemplo que foi retirado de Gomaa (1984).

Esse sistema exemplo é um Sistema Controlador de um Robô, que controla os movimentos de até 6 eixos de um robô, e interage com sensores digitais em entrada/saída (E/S) de dados, através de um painel de controle constituído de um conjunto de botões e de um seletor de programas.

Apesar de realizar poucas funções, e de ter sido simplificado e adaptado para se tornar um exemplo didático; o seu aspecto de tempo real, que exige um controle preciso de todas as atividades do software, tanto durante o desenvolvimento, quanto durante a fase de operação; e suas interações com outros dispositivos, fizeram com que ele tenha sido classificado como um sistema de médio porte; e por isso, durante este Capítulo abrangeremos apenas o desenvolvimento de uma parte dele, já que essa parte pode ser vista por si só como um sistema de pequeno porte. O sistema como um todo será abordado no **Capítulo 5 - Sistemas de Médio Porte**, desta dissertação.

O Sistema Controlador de um Robô foi, portanto, quebrado em partes ou subsistemas, onde um subsistema é composto de uma grande função, ou por um conjunto de funções correlatas com um contorno lógico bem definido.

Neste Capítulo abrangeremos apenas o desenvolvimento do que podemos chamar de "Subsistema Gerenciador de Painel", que é o subsistema responsável pela interação humana do Sistema Controlador de um Robô, ou seja, ele é responsável por fazer a interação entre o usuário e o robô.

Por que o Sistema Controlador de um Robô foi quebrado em subsistemas, e qual o critério que foi utilizado não interessa por hora, e isso será abordado no Capítulo 5 desta dissertação. O que interessa por enquanto é apenas saber que o Subsistema Gerenciador de Painel interage com o "Subsistema Gerenciador de Entradas do Painel", com o "Subsistema Gerenciador de Saídas do Painel", com o "Subsistema Gerenciador de Eixos", e com o "Subsistema Interpretador", que também fazem parte do Sistema Controlador de um Robô.

Abaixo segue a definição do Subsistema Gerenciador de Painel :

- **Propósito do Sistema :** O propósito do subsistema é obter e validar as entradas do painel de controle do robô, e informar os subsistemas apropriados sobre as entradas válidas.

- **Modelo Descritivo do Sistema :** O objetivo do subsistema é obter (de uma fila de entradas) e validar as entradas do painel de controle do robô que são fornecidas pelo Subsistema Gerenciador de Entradas do Painel; e informar o Subsistema Gerenciador de Saídas do Painel, o Subsistema Interpretador, e o Subsistema Gerenciador de Eixos sobre as entradas válidas, para que eles possam realizar o processamento adequado às situações escolhidas pelo usuário.

Para saber se uma entrada é válida ou não, o subsistema deve consultar uma tabela de estados que contém todas as entradas possíveis para cada um dos estados em que o robô possa se encontrar. Para determinadas entradas, dependendo do estado em que o robô se encontre, o subsistema deverá gerar ações diferentes. Por exemplo, ao obter a entrada "Power", o subsistema deverá executar a ação "Power On" se o robô deve ser ligado; e deverá executar a ação "Power Off", se o robô deve ser desligado. Ao obter a entrada "Run", o subsistema deverá executar a ação "Run Start" se o programa deve ser iniciado, e deverá executar a ação "Run Resume", se o programa deve ser reativado.

Se a ação "Power On" for ativada, o robô deve passar para o estado manual e enviar a saída (Manual On, Power On) para o Subsistema Gerenciador de Saídas do Painel (as saídas enviadas para o Subsistema Gerenciador de Saídas do Painel devem ser colocadas em uma fila). Se a ação "Power Off" for ativada, o robô será desligado e o subsistema deve enviar a saída (Manual Off, Power Off) para o Subsistema Gerenciador de Saídas do Painel. O robô só pode ser desligado se estiver no estado manual.

Se a ação "Run Start" for ativada, o subsistema deve enviar uma mensagem para o Subsistema Interpretador, contendo o ID do programa que deverá ser executado, e um sinal de início de execução. O subsistema deve ainda enviar a saída (Manual Off, Run On) para o Subsistema Gerenciador de Saídas do Painel.

Se a ação "Stop" for ativada, o subsistema deve enviar um sinal de interrupção para o Subsistema Gerenciador de Eixos para que a execução do programa corrente seja suspensa, e deve enviar a saída (Run Off, Stop On) para o Subsistema Gerenciador de Saídas do Painel.

Se a ação "Run Resume" for ativada, o subsistema deve enviar um sinal de reativação para o Subsistema Gerenciador de Eixos, para que a execução do

programa corrente seja reativada; e deve enviar a saída (Stop Off, Run On) para o Subsistema Gerenciador de Saídas do Painel.

Se a ação "End" for ativada, o subsistema volta para o estado manual, e deve enviar um sinal de término para o Subsistema Interpretador, para que a execução do programa corrente seja abortada. Ele deve enviar ainda a saída (Manual On, Run Off) para o Subsistema Gerenciador de Saídas do Painel.

Se a entrada do painel for a seleção de um novo programa, o subsistema deverá convertê-la no ID desse programa. Só será possível selecionar um novo programa se o subsistema estiver no estado manual.

A tabela de estados do robô é apresentada a seguir :

TABELA 3.3 - ESTADOS DO ROBÔ

Entrada do Painel	**Estado do Robô**	**Ação**	**Novo Estado do Robô**
POWER	Desligado	POWER ON	Manual
	Manual	POWER OFF	Desligado
RUN	Manual	RUN START	Executando
	Suspenso	RUN RESUME	Executando
END	Suspenso	END	Manual
	Executando	END	Manual
STOP	Executando	STOP	Suspenso
Novo Programa	Manual	Número do Programa	Manual

- **Identificação do Sistema :** Subsistema Gerenciador de Painel

- **Lista de Entidades do Sistema :**

  - **Entidade Gerenciador de Entradas do Painel.** Envia as entradas lidas do painel de controle para o Subsistema Gerenciador de Painel.

• **Entidade Gerenciador de Saídas do Painel.** Recebe do Subsistema Gerenciador de Painel dados a respeito das saídas que devem ser geradas no painel de controle.

• **Entidade Gerenciador de Eixos.** Recebe do Subsistema Gerenciador de Painel a ordem para interromper e para reativar a execução do programa corrente.

• **Entidade Interpretador.** Recebe do Subsistema Gerenciador de Painel a ordem para iniciar e terminar o programa escolhido pelo usuário.

• **Diagrama de Contexto do Sistema :**

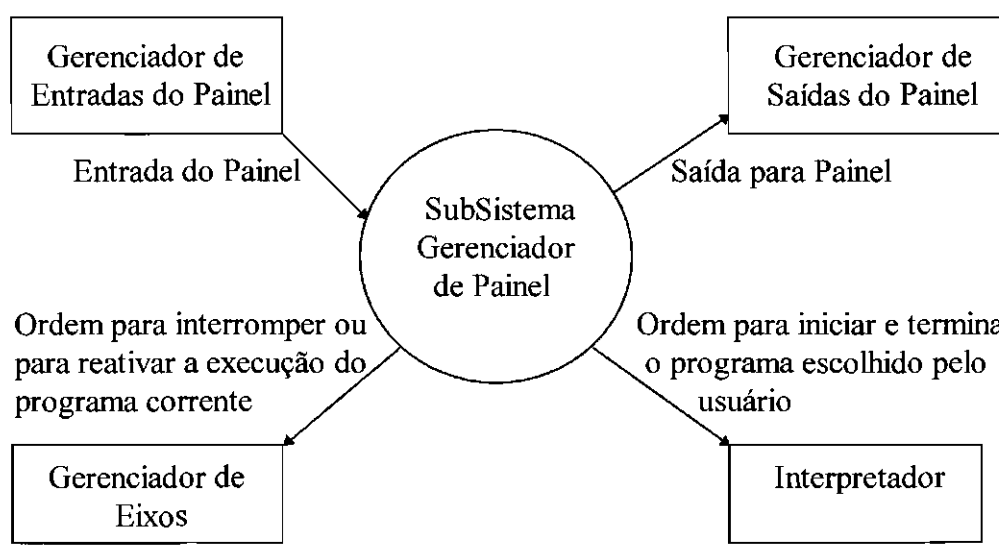


Fig. 3.1 - Diagrama de Contexto do Subsistema Gerenciador de Painel.

• **Eventos do Sistema e Reações Provocadas por eles :**

• **Chegada de entrada do painel.** O subsistema deve validar essa entrada de acordo com uma tabela de estados, que contém todas as entradas possíveis para

cada um dos estados em que o robô possa se encontrar. Se a entrada for um novo programa a ser executado, depois de validada ela deve ser convertida para o ID do programa correspondente

- **Ação "Power On" foi ativada.** O subsistema deve ser ligado. Ele passa para o estado manual, e envia a saída (Manual On, Power On) para o Subsistema Gerenciador de Saídas do Painel.

- **Ação "Power Off" foi ativada.** O subsistema será desligado, e envia a saída (Manual Off, Power Off) para o Subsistema Gerenciador de Saídas do Painel.

- **Ação "Run Start" foi ativada.** O subsistema envia uma mensagem para o Subsistema Interpretador contendo o ID do programa que deverá ser executado, e um sinal de iniciar execução. O subsistema envia a saída (Manual Off, Run On) para o Subsistema Gerenciador de Saídas do Painel.

- **Ação "Stop" foi ativada.** O subsistema envia um sinal de interrupção para o Subsistema Gerenciador de Eixos, e envia a saída (Run Off, Stop On) para o Subsistema Gerenciador de Saídas do Painel.

- **Ação "Run Resume" foi ativada.** O subsistema envia um sinal de reativação para o Subsistema Gerenciador de Eixos, e envia a saída (Stop Off, Run On) para o Subsistema Gerenciador de Saídas do Painel.

- **Ação "End" foi ativada.** O subsistema envia um sinal de término para o Subsistema Interpretador, e envia a saída (Manual On, Run Off) para o Subsistema Gerenciador de Saídas do Painel.

Deve-se lembrar que a definição deve ser vista como um processo de representação. Ela deve ser, na verdade, uma descrição daquilo que é desejado, e não de como tem que ser

realizado (implementado). Isto é, ela deve ser um modelo cognitivo em vez de um modelo de projeto e implementação, descrevendo portanto, o sistema da forma como ele é percebido pela comunidade de usuários. Ela deve abranger o sistema do qual o software é apenas um componente; pois um sistema é composto de componentes interagentes, e somente dentro do contexto de todo o sistema e da interação entre suas partes é que o comportamento de um componente específico pode ser definido.

Deve-se lembrar também que a definição deve ser completa o bastante para que possa ser usada para determinar se a implementação proposta a satisfaz em situações de testes arbitrariamente escolhidas. Ou seja, obtidos os resultados de uma implementação sobre algum conjunto de dados arbitrariamente escolhido, é preciso que seja possível usar a definição para validar esses resultados.

Após a definição do sistema, pode-se começar sua construção propriamente dita; e cabe ao desenvolvedor decidir de que maneira ele deseja desenvolver o sistema, se pela maneira tradicional, ou se pela orientação a objetos.

A maneira tradicional ou paradigma clássico envolve o uso de técnicas estruturadas e não estruturadas e linguagens de programação tradicionais como Fortran, Pascal, C, Visual Basic, etc... O paradigma da orientação a objetos envolve a construção de modelos orientados para objeto e linguagens também orientadas para objeto, tais como C++, Delphi, etc...

Se o desenvolvedor não possuir os conceitos de orientação a objetos, e não estiver disposto a estudar e absorver uma maneira totalmente nova de se desenvolver software; que inclui o aprendizado de técnicas orientadas para objetos para o desenvolvimento de sistemas, e de uma nova linguagem de programação, então recomenda-se que ele siga o desenvolvimento de seu sistema usando as técnicas estruturadas.

Se a opção feita for por desenvolver o sistema usando técnicas orientadas para objetos, vá para o **Capítulo 4 - Construção do Sistema de Pequeno Porte Usando o Paradigma da Orientação a Objetos**, desta dissertação.

3.2 - Construção do Sistema de Pequeno Porte Usando o Paradigma Clássico

Se o sistema a ser desenvolvido for usar técnicas tradicionais, após a definição pode-se partir direto para a construção do modelo lógico do sistema (para mais detalhes sobre modelos de abstração do software, ver o item **2.9 - Modelagem e Abstração**, no Capítulo 2 desta dissertação), o que significa partir direto para a construção do diagrama de estrutura dos módulos.

O diagrama de estrutura dos módulos deve mostrar o particionamento do sistema em módulos; mostrando a hierarquia e organização dos módulos, como ilustra o exemplo da Figura 3.2 a seguir.

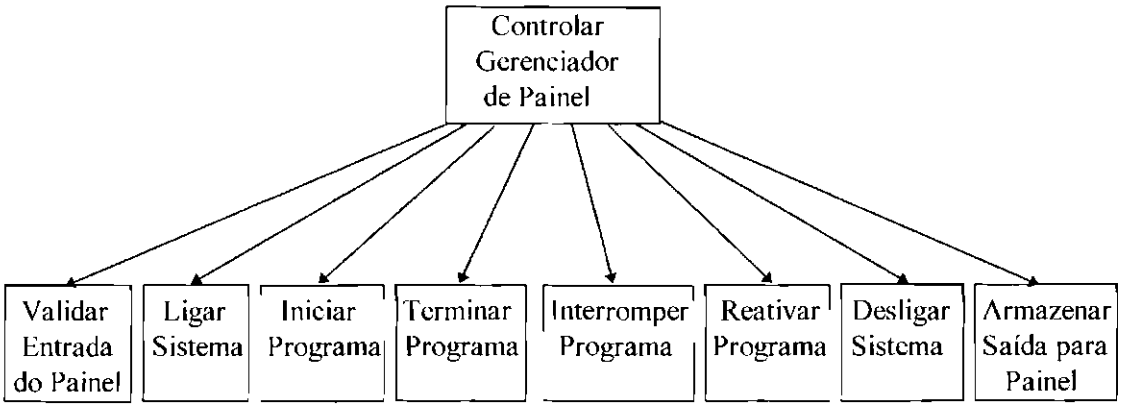


Fig. 3.2 - Diagrama de Estrutura dos Módulos - Subsistema Gerenciador de Painel.

Um módulo é representado no diagrama de estrutura por uma caixa retangular. Na Figura 3.2, "Controlar Gerenciador de Painel", "Validar Entrada do Painel", "Ligar Sistema", "Iniciar Programa", "Terminar Programa", "Interromper Programa", "Reativar Programa", "Desligar Sistema", e "Armazenar Saída para Painel", são módulos.

Quando um módulo tiver parte de sua função executada por um módulo de nível mais baixo, uma seta deve ser desenhada entre eles como mostra a Figura 3.2. Por exemplo, o módulo "Controlar Gerenciador de Painel" tem seu processamento interno, mas parte de sua função é processada pelos módulos "Validar Entrada do Painel", "Ligar Sistema", "Iniciar Programa", "Terminar Programa", "Interromper Programa", "Reativar Programa", Desligar Sistema", e "Armazenar Saída Para Painel".

No nível de implementação do sistema, um módulo que coopera com outro módulo de nível superior, será na verdade uma *procedure* ou *sub-rotina* que é chamada pelo módulo superior. Por "ser chamada" deve-se entender executar a sua função e retornar o controle da execução para o módulo que a chamou.

Nessa fase de estruturação do sistema não se deve preocupar com detalhes como : em que situação um módulo chama outro módulo, quando a chamada ocorre, quantas vezes uma chamada se repete, etc... Enfim, nessa fase não se deve preocupar com detalhes de implementação.

Porém, como identificar e projetar os módulos que comporão o diagrama de estrutura do sistema ?

É importante lembrar que nessa fase de projeto dos módulos, deve-se ater somente na visão externa do módulo, isto é, o que o módulo faz deve interessar mais do que como ele faz. Na verdade, o que o módulo faz deve ser, sempre que possível, uma única função completa, sem partes faltando ou partes sobrando (Page-Jones,1988).

Uma boa maneira de se começar a procurar os módulos que comporão o sistema, é rever a lista de reações provocadas pelos eventos externos feita na definição do sistema, grifando os verbos que forem encontrados. Esses verbos, aliados a seus complementos, são sérios candidatos a se transformarem em módulos.

Ao nomearmos os módulos do sistema devemos tomar alguns cuidados. Primeiramente, deve-se estabelecer um padrão para os nomes desses módulos. Esse padrão deve ser, como dito acima, um nome na forma de verbo + complemento; como ilustrado na Figura 3.2.

Segundo, o nome do módulo deve lembrar o que ele faz cada vez que é chamado, portanto, é importante que se adote nomes que realmente representem a função que o módulo realiza.

3.2.1 - Projetando Módulos Coesos

Um dos critérios para se fazer um bom projeto do sistema é projetar módulos coesos. A coesão é um critério para avaliar como as atividades ou partes de um módulo estão relacionadas umas às outras. Isto é, o termo coesão tem o sentido de "colocar as partes fortemente associadas no mesmo módulo".

Na verdade, o que se deseja durante o projeto dos módulos é que eles sejam altamente coesos, cada um deles constituído de elementos genuinamente relacionados uns aos outros, contribuindo para a execução de uma mesma função.

Uma exceção à regra de coesão de um módulo é o módulo superior do diagrama de estrutura dos módulos. Ele deve abranger o sistema como um todo, e por isso nem sempre é coeso, já que o sistema pode ter mais que uma função.

Para se obter módulos coesos deve-se projetar módulos que tenham todas as partes necessárias, e somente essas, para realizar somente uma função específica relacionada ao problema. Por exemplo, se observarmos o módulo "Validar Entrada do Painel" da Figura 3.2, podemos considerar que ele realiza uma única função completa que é validar as entradas do painel; se, e somente se, "Validar Entrada do Painel" puder ser entendido como englobando todas as partes necessárias para executar essa função (etapas de

cálculo, sub-funções, processamento de entrada/saída), e somente essas. Outros exemplos de módulos coesos podem ser : calcular ponto de impacto de um míssil, calcular corrente elétrica, ler registros de funcionários.

Alguns módulos, apesar de estarem executando uma única função, poderão possuir sub-funções, que juntas realizarão o trabalho total, como por exemplo o módulo "Calcular a Conta Total do Cliente" da Figura 3.3 a seguir. Ele tem uma única função, embora tenha parte dela executada pelo módulo "Calcular Total Hospedagem" e parte dela executada pelo módulo "Calcular Total Restaurante".

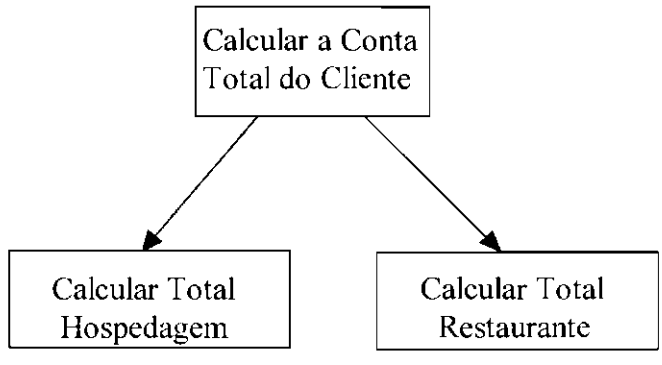


Fig. 3.3 - Um Módulo Coeso e suas Sub-Funções.

Diante disso, como saber se o módulo é realmente coeso? É necessário ter em mente que não importa o quão complexo seja o módulo, e nem quantas sub-funções realizam partes de seu trabalho; se essas sub-funções puderem ser resumidas em uma única função relacionada com o problema, então o módulo é coeso.

Portanto, uma maneira prática de verificar se um módulo é realmente coeso, é descrever a função do módulo em uma frase, e verificar se pode-se derivar dessa frase, o nome do módulo na forma de verbo + complemento (Page-Jones,1988).

Por exemplo, para o módulo "Calcular a Conta Total do Cliente" da Figura 3.3, pode-se descrever sua função na frase : Calcula o valor total da conta de um cliente. Dessa frase,

pode-se derivar o nome desse módulo na forma de verbo + complemento, como já mostrado; o que é um bom indicativo de que ele é coeso.

Deve-se observar que a frase que descreve a função do módulo deve ser derivada sob a visão superior dele, isto é, deve-se descrever a função do módulo observando-se o mesmo, do ponto de vista do quê ele faz para seu superior.

Tomemos como exemplo o módulo "Calcular Total Restaurante" da Figura 3.3. Deve-se descrever sua função sob o ponto de vista do que ele faz para "Calcular Conta Total do Cliente", e não sob o ponto de vista de seus subordinados.

Ao particionar a função de um módulo em sub-funções deve-se observar o nível de complexidade que isso acarretará para o sistema. Isto é, deve-se observar o tamanho dos módulos que são projetados. Eles não podem ser grandes demais, pois a nossa habilidade para entender um módulo e encontrar erros depende de sermos capazes de compreende-lo de uma só vez. Ao mesmo tempo, eles também não podem ser pequenos demais.

Portanto, deve-se projetar módulos que possam ter sua codificação visível em uma página de listagem, ou no máximo em duas páginas consecutivas; caso contrário, deve-se separar a função contida nele em sub-funções com módulos próprios.

Essa separação porém, como dito acima, deve ter um limite, pois um módulo de mil linhas é confuso, mas mil módulos de uma linha são ainda mais confusos. Deve-se, portanto, parar o particionamento quando não se puder encontrar uma função bem definida para compor resultados, ou quando a interface de um módulo for tão complexa quanto o próprio módulo.

Outro ponto importante durante a divisão da função de um módulo em sub-funções é projetar módulos que sejam úteis para o sistema. Isto é, deve-se fazer o possível para projetar módulos que possam ser usados em várias partes do sistema.

3.2.2 - Diretrizes para a Construção do Diagrama de Estrutura dos Módulos

À medida que os módulos são identificados, eles devem ser integrados, formando o diagrama de estrutura dos módulos.

Porém, como organizar os módulos dentro do diagrama de estrutura ? Existem algumas medidas que podem ser tomadas para melhor organizar o diagrama de estrutura dos módulos, tornando-o mais claro, e portanto, mais fácil de ser entendido.

Primeiramente, embora o diagrama de estrutura não mostre necessariamente em qual seqüência os módulos são chamados, deve-se sempre que possível, desenhar os módulos seguindo sua ordem de chamada da esquerda para a direita. Isto faz com que naturalmente os módulos de entrada de dados fiquem mais à esquerda no diagrama de estrutura, os módulos transformadores de dados fiquem ao centro, e os módulos de saída de dados fiquem mais à direita no diagrama de estrutura.

Podemos notar na Figura 3.2, que posicionando os módulos de acordo com sua ordem de chamada da esquerda para a direita no diagrama de estrutura, o módulo de entrada de dados, nesse caso "Validar Entrada do Painel", ficou posicionado mais à esquerda no diagrama; o módulo "Armazenar Saída Para Painel", que é um módulo de saída de dados, ficou posicionado mais à direita no diagrama; e os módulos transformadores ficaram mais ao centro. Isso deixa o diagrama mais claro, ficando portanto mais fácil de ser entendido e modificado.

Segundo, deve-se representar um módulo apenas uma vez no diagrama de estrutura, mesmo que ele tenha mais de um superior.

Terceiro, o nome do módulo deve sintetizar não só sua própria função de codificação, como também de todos os seus subordinados imediatos. Um exemplo disso é dado pelo módulo "Controlar Gerenciador de Painel" da Figura 3.2. O nome desse módulo sintetiza

a função do módulo, independentemente de ele realizar a função por si, ou de ele chamar subordinados para ajudá-lo. O importante é que ele exerce a função de Controlar o Gerenciador de Painel, como se toda a codificação estivesse dentro dele.

Quarto, deve-se evitar sempre que possível, linhas de comunicação cruzadas dentro do diagrama de estrutura, pois elas obscurecem o entendimento.

Quinto, os módulos já existentes que forem ser reaproveitados de outros sistemas devem ser marcados como mostra a Figura 3.4 a seguir.

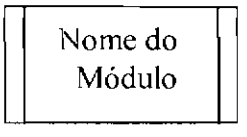


Fig. 3.4 - Módulo já Existente.

Sexto, deve-se sempre que possível construir sistemas balanceados, onde os módulos superiores do diagrama de estrutura lidem mais com dados de natureza lógica do que com dados de natureza física. Quando analisar seu sistema em relação ao balanceamento lembre-se que ele não é uma característica visual. Portanto, o que deve ser levado em conta ao se analisar o sistema em relação ao balanceamento é a organização lógica dos módulos, e não o tamanho dos ramos do diagrama.

A Figura 3.2, por ser uma primeira abordagem do diagrama de estrutura dos módulos, não mostra a comunicação entre eles. Numa abordagem subseqüente entretanto, os nós de comunicação devem ser acrescentados ao diagrama.

A comunicação entre os módulos corresponde nada mais nada menos que à passagem de informações entre nós durante as chamadas. Na fase de implementação corresponderia à

passagem de *parâmetros* entre um programa e suas *sub-rotinas*, ou entre duas *sub-rotinas*.

A representação da comunicação entre módulos no diagrama de estrutura pode ser feita como mostra a Figura 3.5 adiante. Na Figura 3.5 :

- **Controlar Gerenciador de Painel** envia dados (Entrada) para **Validar Entrada do Painel**;

- **Validar Entrada do Painel** (após executar sua função) envia dados (Ação) e envia uma chave (*flag*) (Entrada é Válida) para **Controlar Gerenciador de Painel**;

- **Ligar Sistema** (após executar sua função) envia dados (Saída para Painel) para **Controlar Gerenciador de Painel**;

- **Controlar Gerenciador de Painel** envia dados (ID do Programa) para **Iniciar Programa**;

- **Iniciar Programa** (após executar sua função) envia dados (Saída para Painel) para **Controlar Gerenciador de Painel**;

- **Terminar Programa** (após executar sua função) envia dados (Saída para Painel) para **Controlar Gerenciador de Painel**;

- **Interromper Programa** (após executar sua função) envia dados (Saída para Painel) para **Controlar Gerenciador de Painel**;

- **Reativar Programa** (após executar sua função) envia dados (Saída para Painel) para **Controlar Gerenciador de Painel**;

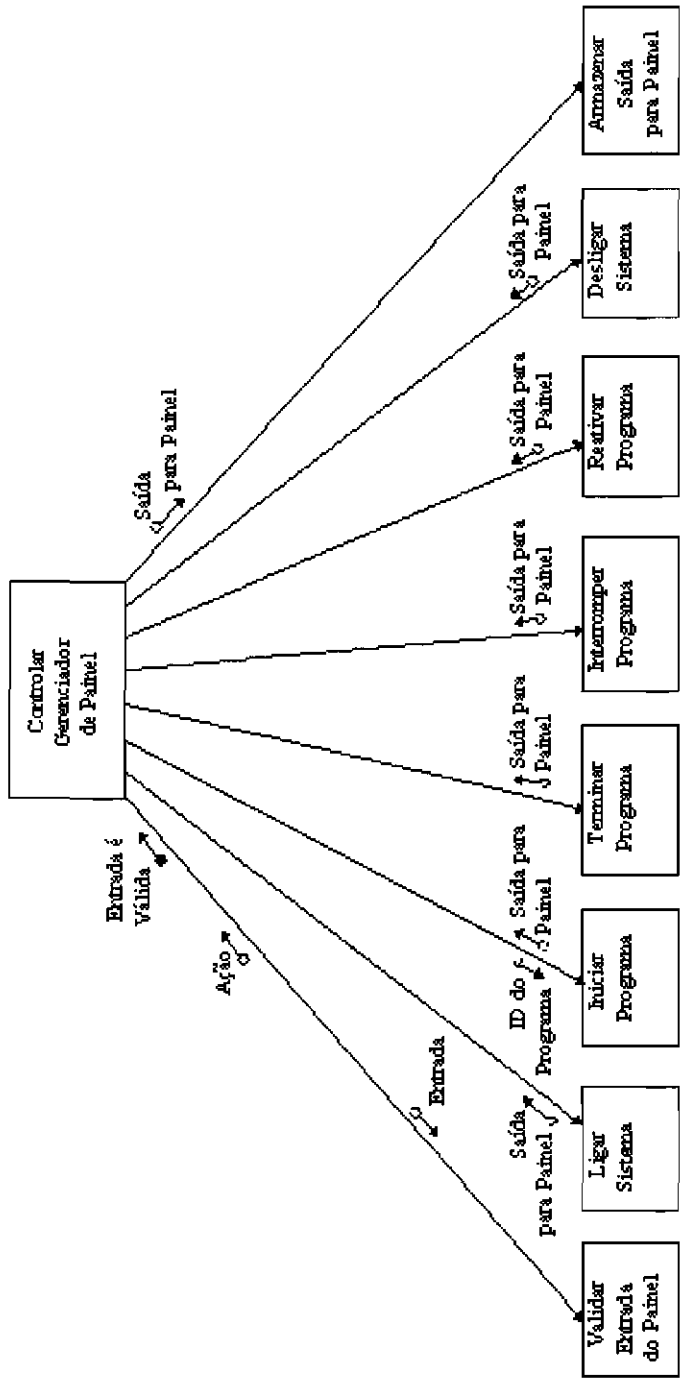


Fig. 3.5 - Comunicação entre Módulos.

• **Desligar Sistema** (após executar sua função) envia dados (Saída para Painel) para **Controlar Gerenciador de Painel**, e

- **Controlar Gerenciador de Painel** envia dados (Saída para Painel) para **Armazenar Saída para Painel.**

Os módulos podem trocar informações de dados e de controle, e portanto os símbolos de comunicação entre os módulos tem dois significados diferentes. Um controle não é processado, e sim testado. Ele tende a ser um passo externo ao mundo do problema, pois ele comunica informações sobre uma parte de dados. Já os dados são processados e se relacionam com o problema. A Figura 3.6 abaixo mostra os dois símbolos de comunicação entre módulos.

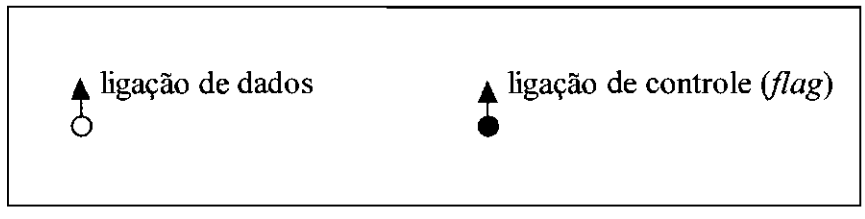


Fig. 3.6 - Símbolos de Comunicação entre Módulos.

Além disso, o sentido da seta indica qual módulo envia informação para o outro, como mostra a Figura 3.7 abaixo.

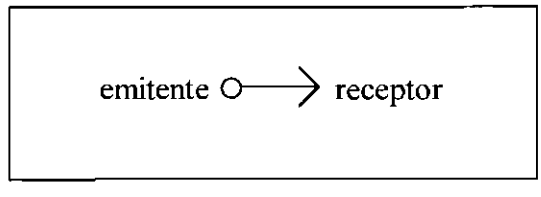


Fig. 3.7 - O módulo Emitente Envia Informação para o Módulo Receptor.

3.2.3 - Acoplamento entre Módulos

A estruturação do sistema em módulos hierarquizados é um grande passo para a qualidade do sistema, contudo, não é suficiente para garantir que os módulos estejam se comunicando de maneira adequada. Para tal, é necessário verificar o quanto os módulos dependem uns dos outros, ou seja, é necessário cuidar do acoplamento entre os módulos.

O acoplamento mede, então, o grau de interdependência entre os módulos. O objetivo do projeto é minimizar o acoplamento, isto é, tornar os módulos tão independentes quanto possível. Para que isso aconteça deseja-se que os acoplamentos sejam os mais fracos possíveis, e que os acoplamentos desnecessários sejam eliminados (Page-Jones,1988).

Para minimizar o acoplamento entre os módulos deve-se policiar atentamente o tipo de informação que está sendo passada de um módulo para outro. O ideal é que os módulos se comuniquem por *parâmetros*, sendo que cada *parâmetro* deve ser um único elemento de dados ou uma tabela homogênea (uma tabela na qual cada entrada contém o mesmo elemento de dados).

Na Figura 3.5 pode-se notar que há um baixo acoplamento entre os módulos, pois nenhum dos dados podem ser dispensados e não há nenhum elemento adicional.

Uma vez que os módulos tem que se comunicar, a ligação de dados é inevitável, mas deve ser mantida a taxas mínimas; isto é, deve-se evitar o passeio de dados através dos módulos. Um exemplo de passeio de dados é mostrado na Figura 3.8.

Na Figura 3.8, observamos que "Registro Funcionário" percorre uma longa trajetória desde o ponto que ele entra no sistema até atingir o módulo onde é usado. Uma solução melhor é dada na Figura 3.9.

Um caso de passeio de dados típico ocorre quando há divisão de decisão. Isso acontece quando os dados requeridos pela parte de reconhecimento de uma decisão (qual ação tomar), e os dados requeridos pela parte de execução, não estão no mesmo módulo. Ao se notar que houve uma divisão de decisão, deve-se fazer um novo arranjo dos módulos para trazer o reconhecimento mais perto da execução.

Outro caso de passeio de dados típico acontece na detecção de erros. O passeio de dados nesse caso pode ser evitado fazendo com que um erro seja relatado pelo módulo

que o detectou e que conhece sua natureza. As mensagens de erro também devem ser posicionadas dessa maneira. Em relação às mensagens de erro, deve-se lembrar que deve-se manter um formato padrão para elas, evitando que o usuário venha a receber mensagens diferentes sobre um mesmo erro.

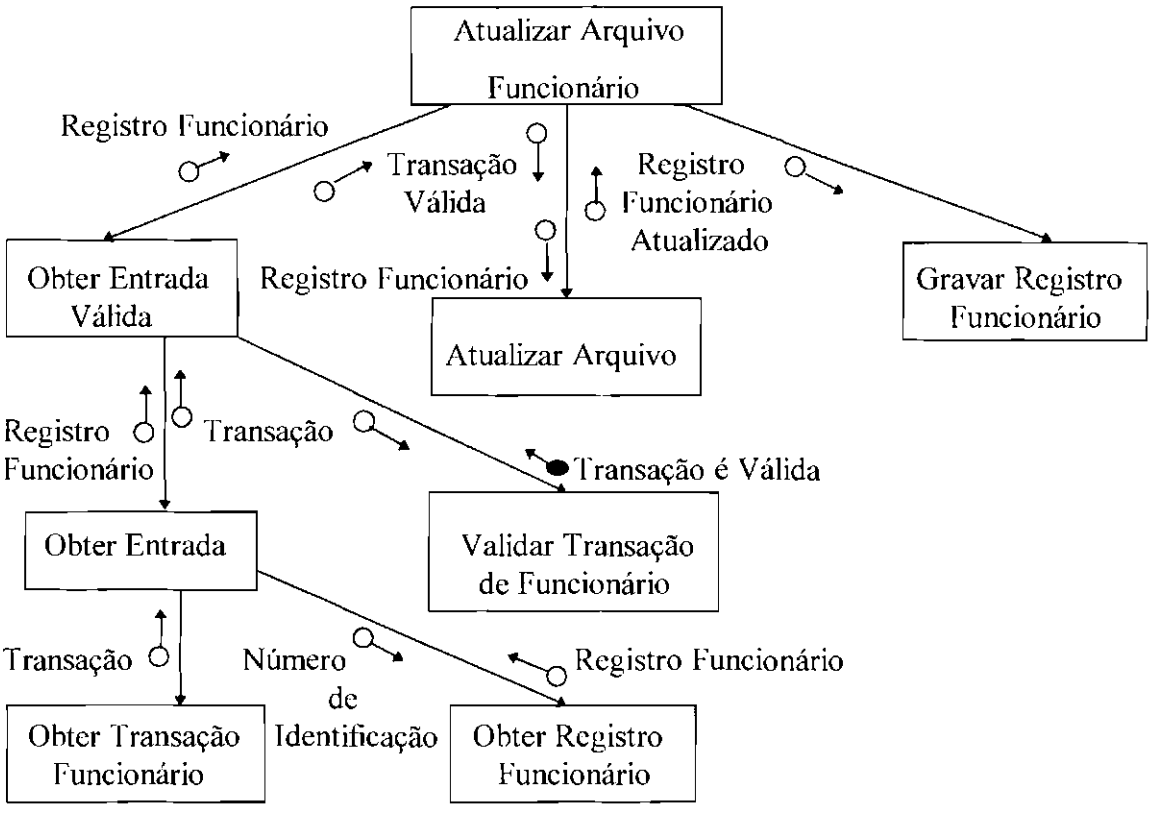


Fig. 3.8 - Passeio de Dados.

Outra maneira aceitável de comunicação entre módulos é quando eles se referem à mesma estrutura de dados. Por estrutura de dados deve-se entender um conjunto de dados tal como um registro constituído por vários campos. Um exemplo disso é dado na Figura 3.10. Nessa figura, o registro de funcionários é composto de :

- número do funcionário;
- nome do funcionário;
- RG do funcionário;

- endereço do funcionário;
- departamento que trabalha;
- ramal;
- salário e
- valor do tiquet refeição que recebe.

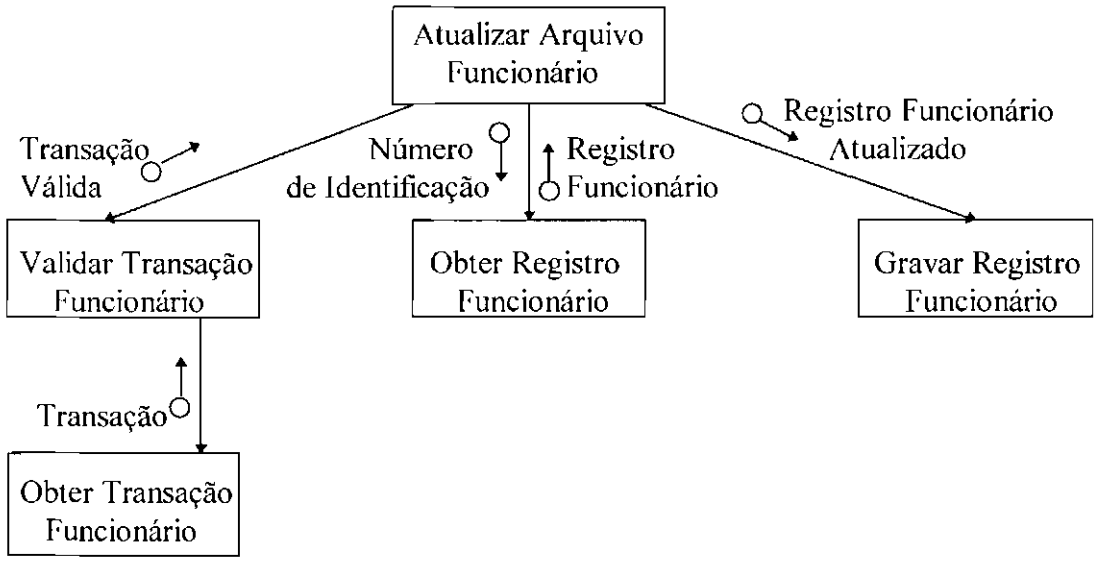


Fig. 3.9 - Eliminando o Passeio de Dados.

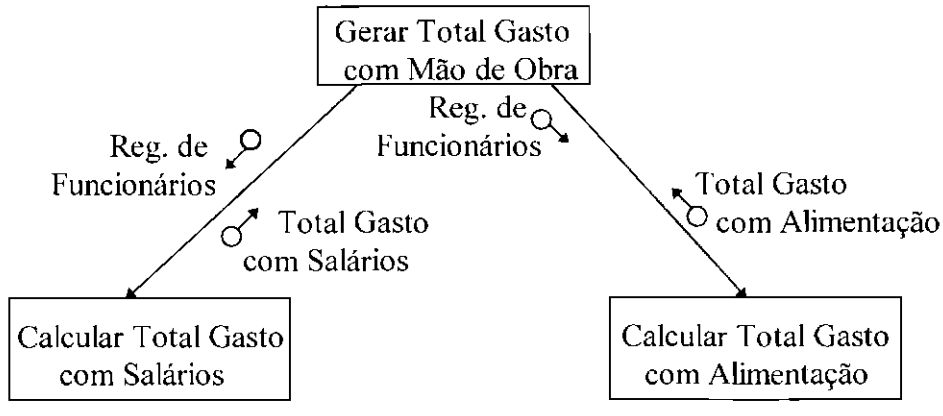


Fig. 3.10 - Acoplamento Aceitável.

Como pode-se notar na Figura 3.10, apesar do módulo "Calcular Total Gasto com Salários" requerer somente o campo "Salário" do registro de funcionários, ele recebe todo o registro. O mesmo acontece com "Calcular Total Gasto com Alimentação", que necessita apenas do campo "Valor do Tiquet Refeição que Recebe". Qualquer alteração no registro de funcionários, tanto em formato como em estrutura, afetará todos os módulos que a ele se refiram, mesmo aqueles que não utilizem os campos modificados.

Por isso, apesar de aceitável, esse tipo de comunicação entre módulos deve ser evitado sempre que possível, pois ele cria dependência entre módulos originalmente não relacionados.

Após minimizar acoplamento entre os módulos, deve-se verificar se o diagrama de estrutura dos módulos está abrangendo todos os objetivos que foram determinados na definição do sistema. Se isso não estiver acontecendo é hora de rever em que ponto há falhas, e refazer os módulos problemáticos.

3.2.4 - Especificação dos Módulos

O diagrama de estrutura dos módulos mostra somente a estrutura geral de um sistema, e deliberadamente reprime quase todos os detalhes de procedimento. No entanto, o programador que usará o diagrama de estrutura para derivar a codificação do módulo, precisa de detalhes de procedimento que deverão ser fornecidos pela especificação de cada módulo do diagrama de estrutura.

Fazer a especificação dos módulos consiste em descrever a visão externa do módulo, criando uma documentação para cada módulo. Esta atividade descreve os dados de entrada que são utilizados pelo módulo, os dados de saída que são esperados quando o módulo for processado, e a função que o módulo deve executar. A função deve ser definida como uma simples frase que estabelece a relação entre as entradas e as saídas do módulo.

A Figura 3.11 a seguir mostra a especificação para o módulo **Iniciar Programa**, da Figura 3.2.

módulo - Iniciar Programa
função - Enviar um sinal de início de execução e o ID do programa a ser executado para o Subsistema Interpretador, e enviar a saída (Manual Off, Run On) para o Subsistema Gerenciador de Saídas do Painel.
usa - ID do Programa
retorna - Saída para Painel

Fig. 3.11 - Especificação para um Módulo.

3.2.5 - Programação Estruturada

Antes de começar a programação propriamente dita de um módulo, deve-se detalhar sua especificação usando o português estruturado. Isso significa escrever a lógica interna de um módulo usando expressões em português que contenham apenas as estruturas básicas da programação estruturada e suas combinações. São elas (Gane,1983) :

1. **Instruções Seqüenciais.** Esta estrutura cobre qualquer instrução, ou grupo de instruções, que não tenha repetição ou ramificação englobada nela própria, como por exemplo :

   "Pegar a primeira entrada da Fila de Entradas do Painel."
   "Procurar ID do programa."
   "Enviar Saída para Painel."

   A Figura 3.12 a seguir, mostra a representação gráfica da estrutura de instruções seqüenciais.

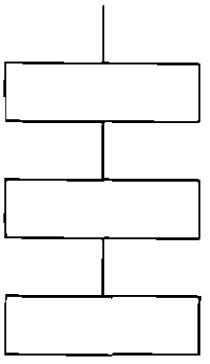


Fig. 3.12 - Instruções Seqüenciais.

2. Instruções de Decisão. O formato padrão para uma decisão é a estrutura :

SE Condição-1
ENTÃO Ação-A
SENÃO (Não Condição-1)
LOGO Ação-B

Nas estruturas de decisão, cada ação (Ação-A, Ação-B) pode ser um conjunto de instruções seqüenciais, um ciclo, outra decisão, ou ainda uma combinação de instruções seqüenciais, ciclos e decisões. Por isso, deve-se usar convenções para alinhar cada SENÃO com o SE ao qual ele está associado, pois à medida que a lógica se complica pode se tornar muito difícil compreender o aninhamento dessas estruturas.

A Figura 3.13 mostra a representação gráfica da estrutura de instruções de decisão.

Um tipo especial de estruturas de decisão aparece quando existem várias possibilidades de uma condição, sendo que elas representam casos mutuamente exclusivos. Esta estrutura especial é conhecida como a estrutura "caso", e seu formato padrão é :

SE Caso-1

Ação-1

SENÃO SE Caso-2

Ação-2

...

SENÃO Caso-n

Ação-n

A Figura 3.14 mostra a representação gráfica da estrutura "caso".

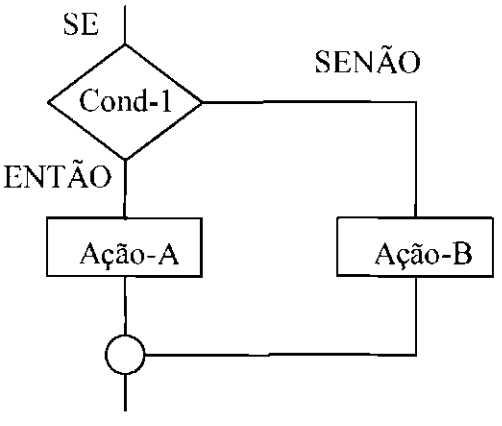


Fig. 3.13 - Instruções de Decisão.

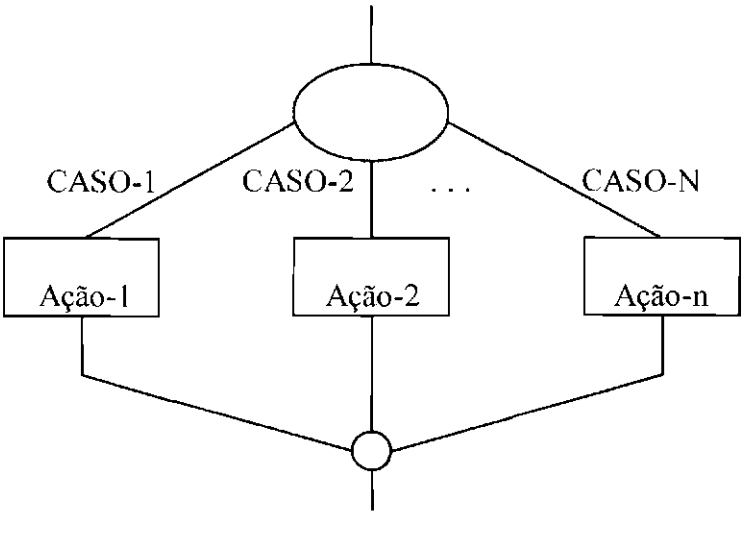


Fig. 3.14 - Estrutura "Caso".

3. Instruções de Repetição (*loop*). Esta estrutura é aplicada a qualquer situação em que uma instrução, ou grupo de instruções, é repetida até que o resultado desejado seja obtido. O formato padrão para as instruções de repetição é a estrutura :

REPETIR Ação **ATÉ** que Condição seja Satisfeita

A Figura 3.15 mostra a representação gráfica da estrutura de instruções de repetição.

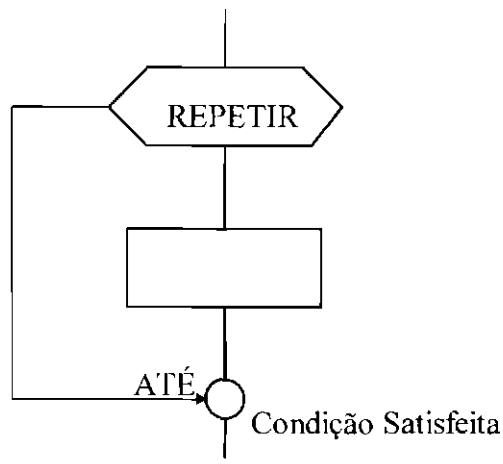


Fig. 3.15 - Instruções de Repetição.

Após a especificação dos módulos deve-se escrever a lógica interna de todos os módulos do diagrama de estrutura usando o português estruturado. Deve-se ter em mente, porém, que o português estruturado não é um programa de computador, pois nele não deve haver especificação de leitura e gravação de arquivos físicos, nenhuma preparação de contadores ou chaves, ou qualquer detalhe do modelo físico do sistema (para mais detalhes sobre modelos de abstração do software, ver o item **2.9 - Modelagem e Abstração**, no Capítulo 2 desta dissertação).

Um exemplo de módulo escrito em português estruturado é dado a seguir. O módulo superior do Subsistema Gerenciador de Painel, "CONTROLAR GERENCIADOR DE PAINEL", foi escrito como uma hierarquia de blocos de instruções. Se o desejo for

apenas obter um resumo desse módulo, precisa-se apenas ler o bloco de cima; se o desejo for obter detalhes de qualquer parte em particular, deve-se ler a especificação daquela parte.

Deve-se observar que o módulo como está escrito, poderia muito bem ser seguido e executado por uma pessoa qualquer, e não pelo computador, o que o deixa livre de detalhes de implementação.

CONTROLAR GERENCIADOR DE PAINEL

REPETIR

Obter entrada da fila de entradas

EXECUTAR VALIDAR ENTRADA DO PAINEL

SE entrada é válida

ENTÃO CASO Ação seja :

"Power On" : EXECUTAR LIGAR SISTEMA

"Run Start" : EXECUTAR INICIAR PROGRAMA

"End" : EXECUTAR TERMINAR PROGRAMA

"Stop" : EXECUTAR INTERROMPER PROGRAMA

"Run Resume" : EXECUTAR REATIVAR PROGRAMA

"Power Off" : EXECUTAR DESLIGAR SISTEMA

SENÃO Converter ação para o ID do programa correspondente

FIM_CASO

EXECUTAR ARMAZENAR SAÍDA PARA PAINEL

FIM_SE

ENQUANTO chegarem entradas

FIM_CONTROLAR GERENCIADOR DE PAINEL

VALIDAR ENTRADA DO PAINEL

Obter estado atual do robô

Consultar tabela de estados para verificar se a entrada é válida

SE entrada é válida

ENTÃO SE entrada é uma seleção de um novo programa

ENTÃO ação é o número do programa escolhido

SENÃO Verificar na tabela de estados qual ação deve ser tomada

Atualizar Estado do Robô

FIM_SE

FIM_SE

FIM_VALIDAR ENTRADA DO PAINEL

LIGAR SISTEMA

Setar Saída do Painel para Manual On, Power On

FIM_LIGAR SISTEMA

INICIAR PROGRAMA

Enviar ID do Programa para Subsistema Interpretador

Enviar sinal de Iniciar para Subsistema Interpretador

Setar Saída do Painel para Manual Off, Run On

FIM_INICIAR PROGRAMA

TERMINAR PROGRAMA

Enviar sinal de Terminar para Subsistema Interpretador

Setar Saída do Painel para Manual On, Run Off

FIM_TERMINAR PROGRAMA

INTERROMPER PROGRAMA

Enviar sinal de Interromper para Subsistema Gerenciador de Eixos

Setar Saída do Painel para Run Off, Stop On

FIM_INTERROMPER PROGRAMA

REATIVAR PROGRAMA

Enviar sinal de Reativar para Subsistema Gerenciador de Eixos

Setar Saída para Painel para Stop Off, Run On

FIM_REATIVAR PROGRAMA

DESLIGAR SISTEMA

Setar Saída para Painel para Manual Off, Power Off

FIM_DESLIGAR SISTEMA

ARMAZENAR SAÍDA PARA PAINEL

Colocar a saída na fila de saídas

FIM_ARMAZENAR SAÍDA PARA PAINEL

Para facilitar a compreensão e visualização da lógica dos módulos, ao escrevê-los em português estruturado deve-se usar as palavras SE, ENTÃO, SENÃO, LOGO, REPETIR, ENQUANTO, FIM-SE e ATÉ em letras maiúsculas; e deve-se deslocar as estruturas verticalmente para mostrar sua hierarquia lógica, como mostrado no exemplo anterior.

Pode-se também usar a representação gráfica do português estruturado para detalhar os módulos, como mostra a Figura 3.16.

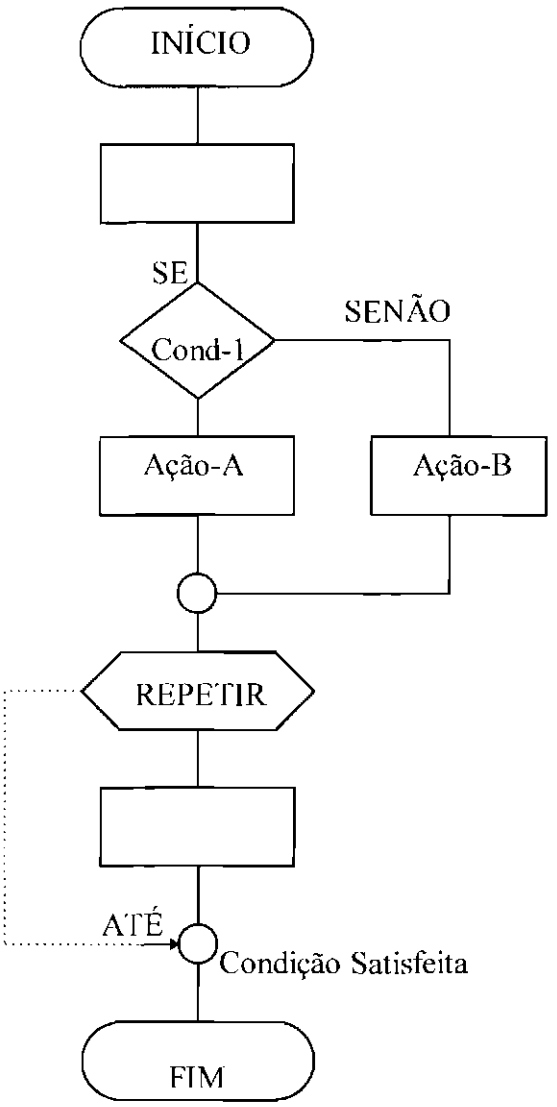


Fig. 3.16 - Combinação das Estruturas Básicas da Programação Estruturada.

3.2.6 - Modelagem de Dados

A modelagem de dados consiste em organizar os dados para que eles possam ser armazenados, recuperados e processados com segurança e eficiência. Por "organizar" pode-se entender agrupar os dados altamente relacionados e dependentes. Por exemplo, os dados que caracterizam uma entidade como "Funcionário", ou seja, "número", "nome", "ramal", "endereço" do funcionário, etc...; são dados fortemente relacionados e devem ser armazenados de forma agrupada.

A modelagem de dados é útil para aplicações em que os dados e as relações que regem esses dados sejam complexas. Ela proporciona ao analista e ao projetista de banco de dados o necessário esclarecimento sobre os dados e as relações que os regem. Porém, sistemas que apresentam poucos tipos de entidades de dados, na maioria das vezes dispensam o modelo de dados, já que os relacionamentos entre as entidades, se existirem, serão simples de serem implementados.

Um exemplo disso é o Subsistema Gerenciador de Painel que adotamos como exemplo no início deste Capítulo. Ele não necessita que se modele seus dados, pois os únicos dados que ele manipula, que são uma tabela de estados do robô e uma tabela de programas possíveis, não têm nenhum relacionamento entre si.

Ao se fazer a modelagem de dados deve-se modelar apenas os dados básicos do sistema, ou seja, os dados que o sistema precisa manter armazenados para que ele atinja seus objetivos. Dados intermediários decorrentes de processamentos não devem ser modelados nessa fase.

Se não for necessário para o seu sistema fazer a modelagem de dados, vá para o item **3.2.7 - Definição do Plano de Implementação e Testes**, neste mesmo Capítulo.

Para não se perder em detalhes recomenda-se que o modelo de dados seja feito em duas etapas. Inicialmente deve-se modelar os dados em entidades-relacionamentos e depois deve-se construir o modelo relacional.

3.2.6.1 - Construção do Modelo de Entidade-Relacionamento

O modelo entidade-relacionamento é o modelo conceitual dos dados do sistema (para mais detalhes sobre modelos de abstração, ver o item **2.9 - Modelagem e Abstração**, no Capítulo 2 desta dissertação), portanto não se deve nesse momento preocupar-se com detalhes de como os dados serão armazenados, e nem como eles serão manipulados.

Deve-se preocupar apenas em identificar os dados básicos do sistema, alguns relacionamentos que existam entre eles, e os agrupamentos em entidades e relacionamentos.

Modelar dados em entidades-relacionamento consiste em construir um diagrama de entidade-relacionamento (para mais detalhes sobre o diagrama de entidade-relacionamento, entidades, relacionamentos e atributos, ver o item **2.10.1.2 - Técnicas não Estruturadas**, no Capítulo 2 desta dissertação). Cada componente do diagrama entidade-relacionamento deve ser rotulado com seu nome correspondente, um exemplo é dado na Figura 3.17 a seguir. Na Figura3.17 :

- **Retângulos.** Representam o conjunto de entidades.
- **Elipses.** Representam os atributos.
- **Losangos.** Representam conjuntos de relacionamentos.
- **Linhas.** Ligam atributos a conjuntos de entidades e conjuntos de entidades a conjuntos de relacionamentos.

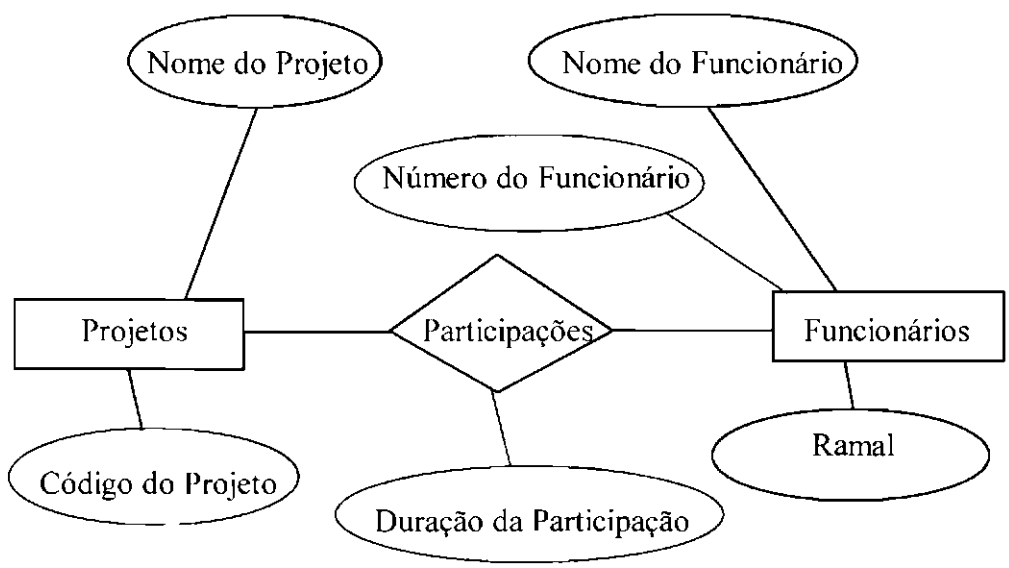


Fig. 3.17 - Diagrama de Entidade-Relacionamento Funcionários - Projetos.

O diagrama de entidade-relacionamento da Figura 3.17 consiste de dois conjuntos de entidades, que são "Funcionários" e "Projetos"; relacionados por um conjunto de relacionamentos "Participações". Os atributos associados a Funcionário são "Nome do Funcionário", "Número do Funcionário" e "Ramal"; e os atributos ligados a Projeto são "Nome do Projeto", e "Código do Projeto".

Como se pode observar, os relacionamentos também podem ter atributos, e no caso do Relacionamento "Participações", o seu atributo é "Duração da Participação". Isso acontece porque o atributo "Duração da Participação" não pode ser de "Funcionário", já que um "Funcionário" poderá estar em vários projetos, e seu tempo de participação pode ser diferente em cada um deles; e porque o atributo "Duração da Participação" também não pode ser de "Projeto", pois um projeto pode ter vários "Funcionários", cada um com um tempo de participação diferente. Então, se o atributo não pode ser de nenhum dos conjuntos de entidades, ele só pode ser do relacionamento. No exemplo dado não dizemos apenas "Tempo de Participação", mas sim "Tempo de Participação de um Funcionário em um Projeto".

Para se construir o diagrama de entidade-relacionamento deve-se inicialmente identificar as entidades do sistema. Deve-se preocupar em identificar apenas as entidades que são relevantes para o sistema em questão. Por exemplo, para um determinado sistema pode interessar o fato de uma pessoa ser funcionário, sendo necessário armazenar informações referentes ao seu nome, seu código, seu ramal. Para um outro sistema, no entanto, pode interessar o fato de uma pessoa ser cliente, sendo necessário armazenar informações referentes ao seu nome, endereço, telefone, cic, banco com o qual trabalha. Uma maneira prática de identificar entidades é procurar na definição do sistema (Modelo Descritivo) por substantivos, pois eles são sérios candidatos a serem entidades.

Após identificar as entidades relevantes para o sistema, deve-se identificar os atributos referentes à cada uma delas. Uma maneira prática de identificar atributos de entidades é

procurar no texto da definição do sistema por adjetivos ou qualidades, pois eles são sérios candidatos a serem atributos de entidades.

Feito isso deve-se identificar os relacionamentos entre cada par de entidades. Uma maneira prática de identificar relacionamentos é procurar no texto da definição do sistema por verbos, pois a presença de um verbo é uma forte indicação da existência um relacionamento. Os tipos de relacionamentos mais comuns são:

1. Relacionamentos um-para-um (1:1) :

Fig. 3.18 - Relacionamento 1 : 1.

Cada departamento é gerenciado no máximo por um funcionário; e cada funcionário gerencia no máximo um departamento.

2. Relacionamentos um-para-muitos (1:N) :

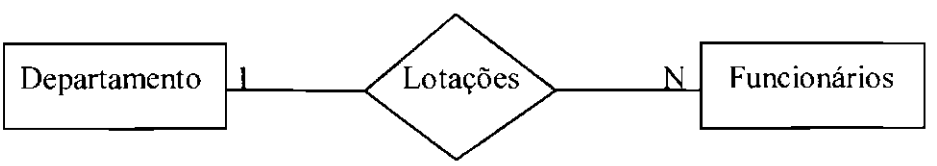


Fig. 3.19 - Relacionamento 1 : N.

Cada departamento lota um número qualquer de funcionários; e cada funcionário é lotado no máximo em um departamento.

3. Relacionamentos muitos-para-muitos (N:N) :

Fig. 3.20 - Relacionamento N : N.

Um projeto poderá ter a participação de vários funcionários; e um funcionário poderá participar de vários projetos.

4. Agregação : Freqüentemente um conjunto de entidades que se relacionam deve ser considerado como uma nova entidade mais complexa. Como uma entidade agregada ela poderá ser relacionada com outras entidades. Um exemplo disso é dado na Figura 3.21 a seguir.

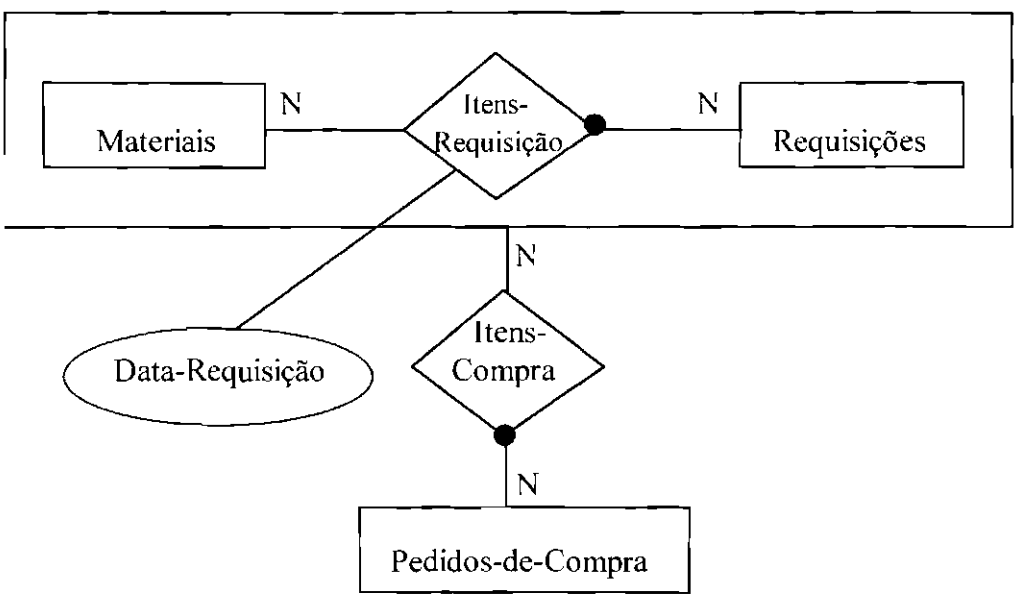


Fig. 3.21 - Agregação.

Na Figura 3.21 a esfera de cor preta indica um relacionamento total, isto é, todos os elementos participam obrigatoriamente no relacionamento. Por exemplo, um pedido

de compra é formado de ao menos uma requisição, e contém ao menos um material; e uma requisição contém ao menos um material.

Nesse exemplo, um pedido de compra consta de vários materiais e é necessário que se conheça para cada um deles em qual requisição ele consta (para controlar, por exemplo, as quantidades pedidas e as requisitadas). Portanto, um pedido de compra relaciona-se com ambos : material e requisição.

Com as requisições de materiais o que ocorre é diferente, pois elas nem sempre geram pedidos de compra. Isso acontece porque parte do material requisitado pode estar, por exemplo, no almoxarifado ou a emissão de pedidos de compra pode ser adiada por um motivo qualquer. Portanto, os materiais relacionam-se com as requisições independentemente dos pedidos.

Então, a Figura 3.21 nos diz que as requisições contém materiais, e que alguns itens de requisição ocorrem em pedidos de compra, mas não obrigatoriamente.

Estabelecidos os relacionamentos entre os conjuntos de entidades deve-se identificar os atributos desses relacionamentos, se existirem. Uma maneira prática de identificá-los é procurar na definição do sistema por advérbios que qualifiquem o verbo que representa o relacionamento.

Finalmente, deve-se colocar os pares de entidades e relacionamentos em um mesmo diagrama como mostra a Figura 3.22. Juntou-se nesse diagrama a Figura 3.18, a Figura 3.19, e a Figura 3.20 para se formar o diagrama de entidade-relacionamento desse determinado sistema.

Quando se tiver uma determinada entidade que tenha muitos atributos de descrição, para ganhar eficiência no acesso pode-se criar uma outra entidade separada para guardar esses

atributos. A entidade acessaria seus atributos através de um código quando fosse necessário, um exemplo é dado na Figura 3.23.

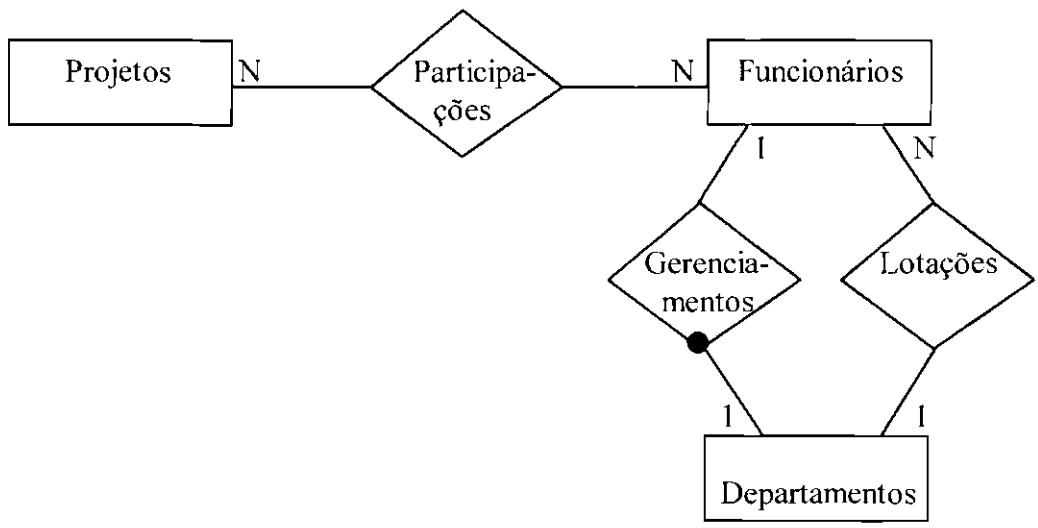


Fig. 3.22 - Diagrama de Entidade-Relacionamento do Sistema.

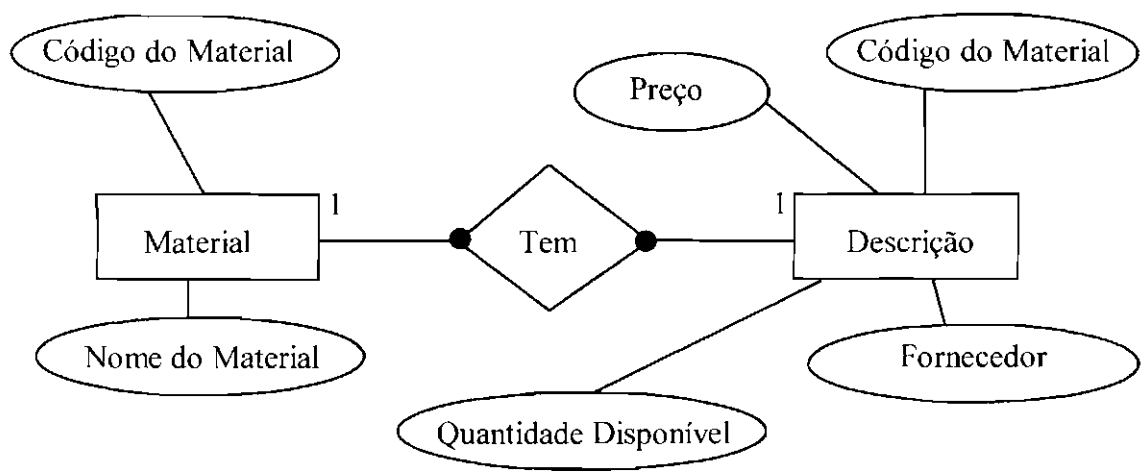


Fig. 3.23 - Entidade Separada para Guardar os Atributos de uma outra Entidade.

O próximo passo na modelagem dos dados é construir o modelo relacional.

3.2.6.2 - Construção do Modelo Relacional

O modelo relacional é o modelo operacional ou lógico (para mais detalhes sobre modelos de abstração, ver o item **2.9 - Modelagem e Abstração**, no Capítulo 2 desta dissertação) dos dados do sistema. Nessa fase portanto, deve-se definir a maneira que os dados serão armazenados e manipulados, isto é, nessa fase começa-se a se preocupar com detalhes de implementação.

Construir o modelo relacional significa organizar os dados do sistema sob a forma de tabelas, também chamadas de relações. As tabelas são organizadas em linhas e colunas, e cada elemento de um conjunto de dados é representado por uma linha na tabela. Cada item de dados desse elemento é representado pelo valor que se encontra na coluna correspondente ao item. Um exemplo de tabela é dado na Figura 3.24 abaixo. Nessa tabela cada elemento do conjunto de dados "Funcionários" (Bernardo, Sueli, Marcelina e Marildo) é representado por uma linha na tabela. Um item de dados de um elemento da tabela, por exemplo, "Ramal" da "Sueli", é representado por um valor que se encontra na coluna correspondente ao item, no caso "65".

TABELA FUNCIONÁRIOS

Número-Func	Nome-Func	Ramal
789	Bernardo	23
438	Sueli	65
567	Marcelina	45
951	Marildo	34

Fig. 3.24 - Exemplo de Tabela.

O esquema de uma tabela é descrito pelo seu identificador e pelo conjunto de nomes de suas colunas. Tomando-se a Figura 3.24 como exemplo, temos para a tabela "Funcionários" o seguinte esquema :

FUNCIONÁRIOS (Número-Func, Nome-Func, Ramal)

Uma coluna, ou uma concatenação de colunas de uma tabela é denomindada "chave", se para uma linha qualquer da tabela não existir uma outra linha com o mesmo valor nessa coluna (ou nessa concatenação de colunas). As chaves são utilizadas para localizar uma linha com eficiência. "Número-Func" pode ser a chave da tabela "Funcionários".

Porém, como mapear o diagrama de entidade-relacionamento para as tabelas do modelo relacional ? A seguir são apresentadas algumas diretrizes de como fazer isso.

3.2.6.2.1 - Mapeamento de Entidades

De uma maneira geral cada conjunto de entidades dá origem à uma tabela, onde cada atributo corresponde a uma coluna, e os dados de cada entidade correspondem a uma linha. Por exemplo, o conjunto de entidades "Funcionários" da Figura 3.17 deu origem à tabela "Funcionários" da Figura 3.24.

3.2.6.2.2 - Mapeamento dos Relacionamentos 1 : N

Existem duas maneiras de mapear os relacionamentos 1 : N para o modelo relacional :

- **Caso 1.** No primeiro caso transpõe-se a chave da tabela correspondente ao conjunto de entidades do lado "1", para a tabela correspondente ao conjunto de entidades do lado "N". (Por "transpor" deve-se entender criar uma coluna com a chave da tabela correspondente a um determinado conjunto de entidades, em outra tabela). Deve-se transpor também os atributos do relacionamento, se existirem. Então, o mapeamento do relacionamento da Figura 3.19, seria como mostra a Figura 3.25 a seguir. A chave para a tabela "Departamentos" é "Número-Depto".

TABELA DEPARTAMENTOS

Número-Depto	Nome-Depto	Localização
01	Física	Prédio II
02	Mecânica	Prédio III
03	Matemática	Prédio I

TABELA FUNCIONÁRIOS

Número-Func	Nome-Func	Ramal	Número-Depto
789	Bernardo	23	01
438	Sueli	65	01
567	Marcelina	45	02
951	Marildo	34	03

Fig. 3.25 - Mapeamento dos Relacionamentos 1 : N com Associação Implícita.

- **Caso 2.** No segundo caso cria-se uma tabela correspondente ao relacionamento transpondo para ela as chaves das duas entidades. Se existirem atributos do relacionamento, eles também deverão fazer parte desta nova tabela. Então, o mapeamento do relacionamento da Figura 3.19 seria como mostra a Figura 3.26.

Porém, quando deve-se usar o **Caso 1**, e quando deve-se usar o **Caso 2** ? A seguir são apresentadas algumas vantagens e desvantagens de cada abordagem, elas devem ser analisadas de acordo com os requisitos estabelecidos pelo sistema que está sendo desenvolvido (Setzer,1986) :

1. A Figura 3.25 tem uma tabela a menos. Isso significa menor espaço ocupado pelos arquivos resultantes e, muito mais importante, maior eficiência (rapidez) no processamento de operações que envolvam o relacionamento. As transações são também expressas de maneira mais simples. Portanto, se eficiência for um requisito do sistema que está sendo desenvolvido, deve-se optar por essa abordagem.

2. A Figura 3.26 apresenta uma solução com maior independência de dados, pois a tabela "Funcionários" contém exclusivamente seus dados próprios, ao contrário da Figura 3.25 em que recebe a chave de "Departamentos".

3. Uma vantagem da Figura 3.26 é que uma mudança estrutural do tipo do relacionamento, passando de 1 : N para N : N, não requer nenhuma alteração estrutural nas tabelas, ao contrário da Figura 3.25.

TABELA DEPARTAMENTOS

Número-Depto	Nome-Depto	Localização
01	Física	Prédio II
02	Mecânica	Prédio III
03	Matemática	Prédio I

TABELA FUNCIONÁRIOS

Número-Func	Nome-Func	Ramal
789	Bernardo	23
438	Sueli	65
567	Marcelina	45
951	Marildo	34

TABELA LOTAÇÕES

Número-Func	Número-Depto
789	01
567	01
951	02
438	03

Fig. 3.26 - Mapeamento dos Relacionamentos 1 : N com Associação Explícita.

3.2.6.2.3 - Mapeamento dos Relacionamentos 1 : 1

Para se fazer o mapeamento dos relacionamentos 1 : 1 deve-se criar uma tabela para cada entidade, e transpor a chave de uma tabela para a outra. O sentido da transposição depende das formas de acesso para ganho de eficiência.

Os exemplos da Figura 3.27 e da Figura 3.28 a seguir mostram as duas maneiras de se mapear os relacionamentos 1 : 1 em tabelas, usando o exemplo dado na Figura 3.18. O exemplo da Figura 3.27 é melhor pois como praticamente todos os departamentos têm gerentes, porém poucos funcionários são gerentes, a coluna "Número-Func" na tabela "Departamentos" produz menos valores vazios, do que a coluna "Número-Depto" na tabela "Funcionários". São eles :

- **Caso 1.** Transpõe-se a chave de "Funcionários" para "Departamentos", como mostra o exemplo da Figura 3.27 a seguir.

TABELA DEPARTAMENTOS-GERENCIAMENTOS

Número-Depto	Nome-Depto	Localização	Número-Func
01	Física	Prédio II	789
02	Mecânica	Prédio III	567
03	Matemática	Prédio I	951

TABELA FUNCIONÁRIOS

Número-Func	Nome-Func	Ramal
789	Bernardo	23
438	Sueli	65
567	Marcelina	45
951	Marildo	34

Fig. 3.27 - Primeira Alternativa para Mapear os Relacionamentos 1 : 1 em Tabelas.

- **Caso 2.** Transpõe-se a chave de "Departamentos" para "Funcionários", como mostra o exemplo da Figura. 3.28 a seguir.

TABELA DEPARTAMENTOS

Número-Depto	Nome-Depto	Localização
01	Física	Prédio II
02	Mecânica	Prédio III
03	Matemática	Prédio I

TABELA FUNCIONÁRIOS-GERENCIAMENTOS

Número-Func	Nome-Func	Ramal	Número-Depto
789	Bernardo	23	01
438	Sueli	65	----
567	Marcelina	45	02
951	Marildo	34	03

Fig. 3.28 - Segunda Alternativa para Mapear os Relacionamentos 1 : 1 em Tabelas.

3.2.6.2.4 - Mapeamento dos Relacionamentos N : N

Cria-se uma tabela correspondente ao relacionamento e transpõe-se para ela as chaves das duas entidades. Se o relacionamento possuir atributos, eles passarão a ser colunas da tabela criada. Então, o mapeamento do relacionamento da Figura 3.20 apresentada no item anterior, seria como mostra a Figura 3.29.

3.2.6.2.5 - Mapeamento de Agregações

Existem duas maneiras de mapear as agregações para o modelo relacional. Tomemos a Figura 3.21 como exemplo :

TABELA PROJETOS

Código-Proj	Nome-Proj
123	Alfa I
456	Beta III
789	Centauro

TABELA FUNCIONÁRIOS

Número-Func	Nome-Func	Ramal
789	Bernardo	23
438	Sueli	65
567	Marcelina	45
951	Marildo	34

TABELA PARTICIPAÇÕES

Código-Proj	Número-Func	Duração-Participação
123	789	1 mês
456	438	2 anos
789	567	5 meses
123	951	3 anos
456	951	8 meses
789	789	5 anos
789	438	4 meses

Fig. 3.29 - Mapeamento dos Relacionamentos N : N.

• **Caso 1.** Cria-se uma tabela correspondente ao relacionamento agregado (que está dentro do retângulo), e tranpõe-se para ela as chaves das entidades agregadas (que estão dentro do retângulo), e os atributos do relacionamento, se existirem. Cria-se uma outra tabela correspondente ao relacionamento que está fora do retângulo, e

transpõe-se para ela as chaves das entidades que estão fora e dentro do retângulo, e os atributos da relação, se existirem. Veja a Figura 3.30 a seguir.

TABELA MATERIAIS

Código-Material	Nome-material	Preço-Material	Fornecedor
01-A	Prego	0.5	Casas Cem
02-A	Broca	0.8	Lojas Andrade
03-B	Cano	1.00	Armazém Legal

TABELA REQUISIÇÕES

Número-Requisição	Item-Requerido	Quantidade-Requerida
01	Prego	03
01	Broca	02
01	Cano	01
02	Prego	10
02	Cano	90
03	Broca	50

TABELA PEDIDOS DE COMPRA

Número-Pedido	Itens-Compra	Quantidade-Compra
01	Prego	05
02	Broca	40

TABELA ITENS DE REQUISIÇÃO

Número-Requisição	Código-Material	Data-Requisição
01	01-A	25/09/97
01	02-A	25/09/97
01	03-B	25/09/97
02	01-A	03/10/97
02	03-B	03/10/97
03	02-A	19/11/97

TABELA ITENS DE COMPRA

Número-Pedido	Número-Requisição	CódigoMaterial
01	01	01-A
02	01	02-A
---	01	03-B
01	02	01-A
---	02	03-B
02	03	02-A

Fig. 3.30 - Mapeamento de Agregações em Duas Tabelas.

• **Caso 2.** Cria-se uma tabela correspondente aos relacionamentos que estão fora e dentro do retângulo, contendo as chaves das entidades que estão fora e dentro do retângulo, além dos atributos dos dois relacionamentos, se existirem. Veja a Figura 3.31 a seguir.

TABELA MATERIAIS

Código-Material	Nome-material	Preço-Material	Fornecedor
01-A	Prego	0.5	Casas Cem
02-A	Broca	0.8	Lojas Andrade
03-B	Cano	1.00	Armazém Legal

TABELA REQUISIÇÕES

Número-Requisição	Item-Requerido	Quantidade
01	Prego	03
01	Broca	02
01	Cano	01
02	Prego	10
02	Cano	90
03	Broca	50

TABELA PEDIDOS DE COMPRA

Número-Pedido	Itens-Compra	Quantidade-Compra
01	Prego	05
02	Broca	40

TABELA ITENS DE REQUISIÇÃO-ITENS DE COMPRA

Número-Pedido	Número-Requisição	Código-Material	Data-Requisição
01	01	01-A	25/09/97
02	01	02-A	25/09/97
---	01	03-B	25/09/97
01	02	01-A	03/10/97
---	02	03-B	03/10/97
02	03	02-A	19/11/97

Fig. 3.31 - Mapeamento de Agregações em uma Única Tabela.

As considerações sobre quando usar o **Caso 1** e quando usar o **Caso 2** para mapeamentos de agregações são análogas às considerações feitas para os mapeamentos 1 : N apresentadas anteriormente.

3.2.6.3 - Implementação dos Dados

Pode-se usar diferentes mecanismos de armazenamento na implementação dos dados modelados em tabelas. Esses mecanismos podem ser arquivos simples, arquivos indexados, ou sistemas gerenciadores de bancos de dados relacionais.

Os arquivos simples e os arquivos indexados são implementados através de linguagens de programação, e o gerenciamento desses arquivos para consulta, inserção e remoção de informações contidas neles é de responsabilidade do programador. Isso significa que todos os mecanismos de gerenciamento para evitar inconsistência de dados devem ser projetados e controlados dentro do código do sistema, via programação.

Como o gerenciamento desses arquivos, por melhor que sejam implementados, oferece apenas capacidades de manipulação e ordenação rudimentares dos dados; eles devem ser evitados quando o volume dos dados e o número de entidades de dados forem grandes, pois pode-se encontrar dificuldades para manter e atualizar as informações contidas neles.

Para maiores informações sobre arquivos simples e arquivos indexados consultar Ullman (1980), Teorey (1982).

Os sistemas gerenciadores de bancos de dados relacionais são pacotes de software prontos que podem ser integrados ao sistema, para armazenar e manipular os dados. Como o ambiente dos bancos de dados relacionais baseia-se no modelo relacional, ao usá-los o usuário estará manipulando várias tabelas de dados.

Os sistemas gerenciadores de bancos de dados relacionais se encarregam de gerenciar os dados contidos nas tabelas, provendo operações especiais que tornam possível manipular esses dados. Para isso eles fornecem uma linguagem de consulta de alto nível, normalmente baseada na álgebra relacional, pela qual o usuário requisita a informação no banco de dados.

A linguagem de consulta mais comum é a "Structured Query Language" (SQL). Outras linguagens de consulta podem ser por exemplo, Quel, QBE.

Dentre os sistemas gerenciadores de banco de dados relacionais que estão em maior evidência atualmente pode-se citar Oracle, SQL Server, Ingres, Informix, Progress, Access, Dbase, etc...

Para mais informações sobre sistemas gerenciadores de banco de dados relacionais consultar Korth (1989), e Setzer (1986).

Após a construção do modelo relacional, se o sistema for usar arquivos simples ou indexados para armazenar os dados, deve-se definir na fase de implementação um arquivo para cada tabela do modelo relacional. Se o sistema for usar um sistema de gerenciamento de bancos de dados relacionais, deve-se definir na fase de implementação uma tabela do banco de dados para cada tabela do modelo relacional.

3.2.7 - Definição do Plano de Implementação e Testes

O sistema deve ser testado para certificar-se de que ele cumpre todos os requisitos estabelecidos na sua definição. Na verdade, a atividade de testes deve acontecer em paralelo com a implementação, ou seja, à medida que o sistema é implementado ele é também testado.

Porém, quais são os objetivos dos testes ? Em Pressman (1995) pode-se encontrar uma série de regras que servem bem como objetivos de teste :

1) a atividade de teste é o processo de executar um programa com a intenção de descobrir um erro;
2) um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto e
3) um teste bem sucedido é aquele que revela um erro ainda não descoberto.

Convém observar que os objetivos acima contrapõe-se ao ponto de vista comumente defendido de que um teste bem sucedido é aquele em que nenhum erro é encontrado.

Portanto, o objetivo nessa fase é realizar testes que descubram sistematicamente diferentes classes de erros, e façam-no com uma quantidade de tempo e esforço mínimos. Por essa razão, deve-se definir um conjunto de passos em que se pode alocar técnicas de projeto de casos de teste, e métodos de teste específicos para o sistema sendo desenvolvido.

A seguir apresentaremos algumas diretrizes para implementar e testar um sistema de pequeno porte desenvolvido usando o paradigma clássico. Essas diretrizes abrangem a adoção de uma estratégia para implementação e testes, e o projeto de casos de testes.

3.2.7.1 - Adoção de uma Estratégia para Implementação e Testes

Antes de começar a implementação e testes do sistema, deve-se adotar uma estratégia para que essas atividades sejam realizadas da melhor maneira possível.

Nesse caso a estratégia recomendada é a integração incremental *top-down*. Esta abordagem envolve primeiro a implementação do módulo superior do diagrama de estrutura dos módulos, com cada um de seus subordinados simulados por um *stub*. Um

stub é um módulo cujas funções são rudimentares. Ele é usado no lugar de um módulo verdadeiro que ainda não tenha sido implementado.

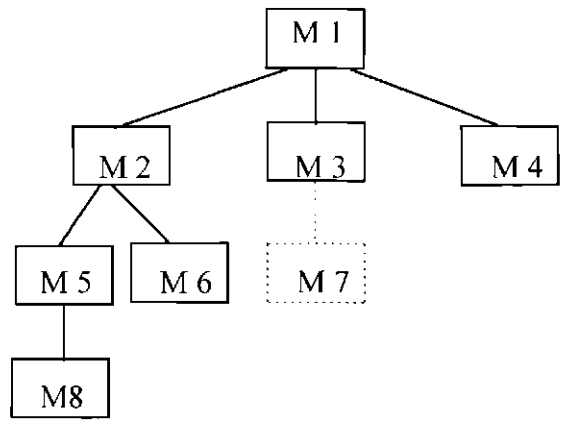
Pode-se criar *stubs* para atuar de diversas maneiras. Eles podem exibir uma mensagem-sinal do tipo : "01, eu fui chamado", que pode ser usada como uma trilha para análise do fluxo de controle do sistema na *depuração (debbuging)*; eles podem retornar um valor constante, por exemplo, R$ 4,00; eles podem ser uma versão simplificada do módulo real; ou até mesmo podem não executar nada.

Portanto, no Subsistema Gerenciador de Painel que adotamos como exemplo, deve-se começar a implementação do sistema pelo módulo "Controlar Gerenciador de Painel", com seus subordinados imediatos "Validar Entrada do Painel", "Ligar Sistema", "Iniciar Programa", "Terminar Programa", "Interromper Programa", "Reativar Programa", "Desligar Sistema", e "Armazenar Saída para Painel" simulados por *stubs*, ver Figura 3.2.

Assim que o módulo superior for implementado ele deve ser testado individualmente, e o *stub* do lado aferente do diagrama de estrutura dos módulos (de entrada de dados) diretamente subordinado ao módulo superior deve ser substituído por um módulo verdadeiro. E assim, cada *stub* do lado aferente do diagrama deve ser substituído por um módulo verdadeiro em profundidade (*depth-first*), com seus subordinados simulados por *stubs*, até que a parte inferior do diagrama seja atingida. À medida que cada módulo verdadeiro é construído, ele deve ser testado individualmente.

Depois que os módulos do lado aferente forem implementados e testados, deve-se adotar a mesma abordagem para os módulos do lado eferente (de saída de dados).

Feito isso, deve-se implementar e testar os módulos transformadores de dados (ao centro) em largura (*breadth-first*) até que a parte inferior do diagrama seja atingida, e o sistema esteja completo. A Figura 3.32 a seguir ilustra o processo.



M1 : Módulo Superior

M2, M5, M8, M6 : Módulos Aferentes

M4 : Módulo Eferente

M3, M7 : Módulos Transformadores

Fig. 3.32 - Integração Incremental *Top-Down*.

Então, inicialmente os testes devem focalizar cada módulo individualmente, garantindo que ele funcione adequadamente como uma unidade.

Após integrar todos os módulos ao sistema, o software está completamente montado como um pacote, erros de interface foram descobertos e corrigidos, e uma série final de testes de software, os testes de aceitação, pode iniciar-se.

Nesses testes, os critérios de validação (estabelecidos durante a análise de requisitos) devem ser testados. O teste de aceitação deve oferecer a garantia final de que o software atende a todas as exigências funcionais, comportamentais e de desempenho.

3.2.7.2 - Projeto de Casos de Teste

Definida a estratégia que será usada para a implementação e testes do sistema, deve-se projetar os casos de testes que o sistema deverá abranger, estabelecendo também a ordem em que eles deverão ser realizados.

Os casos de testes devem ser fundamentados na definição inicial do sistema e no modelo de implementação. No primeiro caso, poderá ser avaliado se o software está operando de acordo com os requisitos estabelecidos pelo usuário; no segundo caso, poderão ser checados detalhes pertinentes à codificação propriamente dita (Yourdon,1990).

É muito importante projetar um conjunto completo e abrangente de casos de teste, pois isso ajuda muito a validar o software sendo construído, isto é, pode-se projetar casos de teste que chequem funções chaves que o sistema deverá realizar, ou até mesmo projetar casos de teste que chequem condições de contorno que são importantes para que o software opere de maneira correta.

A seguir, apresentaremos algumas diretrizes que podem auxiliar no projeto de casos de testes de unidade e de aceitação.

3.2.7.2.1 - Projeto de Casos de Teste de Unidade

A atividade de teste de unidade deve se concentrar no esforço de verificação da menor unidade de projeto de software - o módulo.

O projeto de casos de testes de unidade deve ser considerado um adjunto da etapa de codificação. Depois que o código fonte de cada módulo foi desenvolvido, revisado e verificado a fim de conseguir uma sintaxe correta, deve-se iniciar o projeto de casos de teste de unidade daquele módulo.

Os testes que devem acontecer como parte da atividade de teste de unidade são ilustrados esquematicamente na Figura 3.33 a seguir. Uma revisão da especificação do módulo proporciona orientação para estabelecer os casos de teste que tenham a probabilidade de descobrir erros em cada uma das categorias citadas na Figura 3.33.

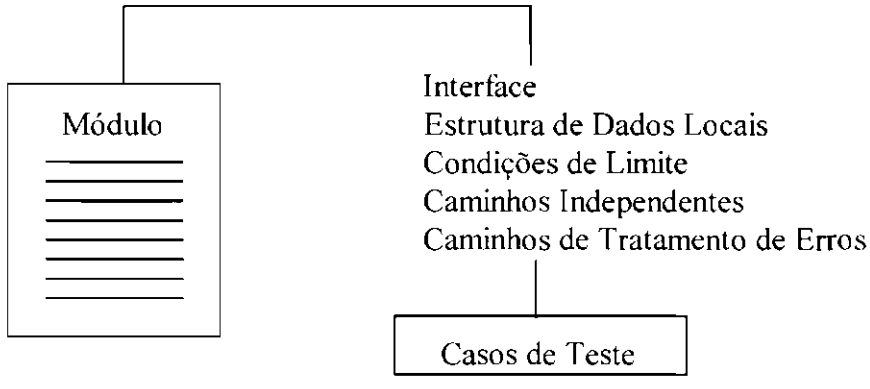


Fig. 3.33 - Teste de Unidade

FONTE : Adaptada de Pressman (1995, p. 845).

Os casos de testes de fluxo de dados ao longo da interface de um módulo devem ser os primeiros a serem identificados e efetuados, pois se os dados não entrarem e saírem adequadamente do módulo, todos os demais testes serão discutíveis. Abaixo segue uma lista de conferência para testes de interfaces :

1) O número de *parâmetros* de entrada é igual ao número de *argumentos* ?

2) Os atributos de *parâmetro* e de *argumento* são compatíveis ?

3) Os sistemas de unidades de *argumentos* e de *parâmetros* são compatíveis ?

4) O número de *argumentos* transmitidos aos módulos chamados é igual ao número de *parâmetros* ?

5) Os atributos dos *argumentos* transmitidos aos módulos chamados são iguais aos atributos dos *parâmetros* ?

6) O sistema de unidades dos *argumentos* transmitidos aos módulos chamados é igual ao de unidades de *parâmetros* ?

7) Os atributos numéricos e a ordem dos *argumentos* para funções embutidas estão corretas?

8) Existem quaisquer referências a *parâmetros* não associados ao ponto de entrada atual?

9) *Argumentos* de entrada/saída alterados ?

10) Definições de *variáveis globais* consistentes ao longo dos módulos ?

11) Restrições passadas como *argumentos* ?

Quando um módulo executa entradas e saídas (E/S) externas, devem ser realizados testes adicionais de interface. São eles :

1) Os atributos de arquivo estão corretos ?

2) As instruções *OPEN/CLOSE* estão corretas ?

3) A especificação de formato é compatível com a instrução de E/S ?

4) O tamanho do *buffer* é compatível com o tamanho do registro ?

5) Arquivos são abertos antes do uso ?

6) Condições de final de arquivo são tratadas ?

7) Erros de E/S manipulados ?

8) Existem quaisquer erros textuais nas informações de saída ?

A estrutura de dados local para um módulo também é uma fonte comum de erros. Casos de teste devem ser projetados para descobrirem erros nas seguintes categorias :

1) digitação inconsistente ou imprópria;

2) iniciação ou valores *default* errôneos;

3) nomes de variáveis incorretos (mal redigidos ou truncados);

4) tipos de dados incorretos e

5) *underflow* ou *overflow*, exceções de endereçamento.

Além das estruturas de dados locais, o impacto de dados globais (por exemplo, COMMON do FORTRAN) sobre um módulo deve ser verificado, pois eles são uma fonte muito provável de erros.

Em se tratando do COMMON do FORTRAN, o caso é ainda mais sério pela própria estrutura de armazenamento das *variáveis globais* em FORTRAN. Na verdade um programa FORTRAN é composto de uma ou mais unidades (programa principal e sub-programas (*sub-rotinas* e funções)). Cada unidade é *compilada* em separado, e associada a um *registro de ativação* que pode ser alocado antes da execução, em outras palavras, variáveis podem ser criadas antes da execução, e seu tempo de vida se estende por toda a duração do programa (variáveis estáticas). O escopo de uma variável entretanto, é limitado à unidade onde a variável é declarada. As unidades podem acessar *variáveis globais* declaradas por meio de comandos COMMON. Estas variáveis podem ser consideradas como pertencentes a um *registro de ativação* fornecido pelo sistema, que é global a todas as unidades de programa. Quando uma unidade é traduzida, posições consecutivas no *registro de ativação* da unidade, são reservadas para as variáveis locais cujas declarações estão sendo processadas, isto é, deslocamentos são amarrados estaticamente às variáveis (Ghezzi,1985).

Então, quando um comando COMMON é declarado no programa principal, um espaço na memória é alocado para essas *variáveis globais* no *registro de ativação* do sistema, em ordem de precedência, como mostra o exemplo da Figura 3.34 a seguir. Quando uma *sub-rotina* deseja acessar essas variáveis, ela pode faze-lo também através do comando COMMON, e o problema aparece justamente aí, pois apesar de acessar a mesma área de memória, variáveis com nomes, dimensões e tipos diferentes podem ser declaradas, propagando erros por todo o programa. Por exemplo, na Figura 3.34 as variáveis A, B, e C são declaradas no programa principal. A subrotina SUB1 deseja acessá-las e por isso declara também um comando COMMON, seguido das variáveis A, X, Y. A mesma área de memória no *registro de ativação* do sistema é acessada por essas variáveis; o que quer dizer que se ocorrer alguma desatenção, como por exemplo, uma declaração de tipo

ou dimensão diferentes, pode-se alterar completamente o resultado do programa. Por exemplo, no exemplo abaixo, B foi declarado no programa principal como tendo 5 elementos, e X foi declarado na *sub-rotina* como tendo 10 elementos; o que ocasionará uma associação errada de variáveis.

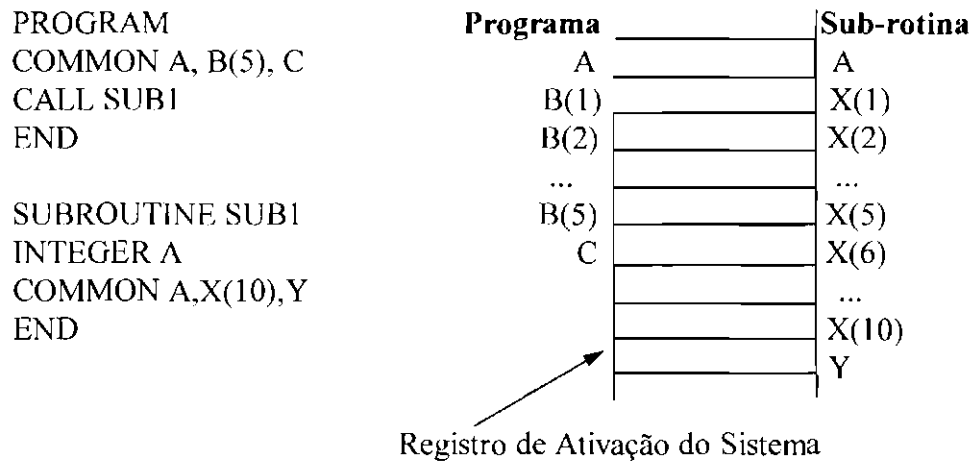


Fig. 3.34 - Esquema de Alocação de Memória do FORTRAN.

O teste seletivo de caminhos de execução também deve ser uma tarefa essencial durante o teste de unidade; e casos de teste devem ser projetados para descobrir erros devido a computações errôneas, comparações incorretas ou fluxo de controle impróprio. Os teste dos *loops* e dos caminhos básicos que as informações devem seguir durante o programa são técnicas efetivas para descobrir um amplo conjunto de erros de caminho; pois o fluxo de controle e as comparações realizadas no programa estão estreitamente relacionados (isto é, a mudança de fluxo acontece freqüentemente depois de uma comparação).

Para realizar o teste seletivo dos caminhos de execução de um módulo, é muito útil dispor de alguma ferramenta de *depuração* (*Debbuger*); pois elas facilitam tremendamente o acompanhamento da alteração dos valores das variáveis ao longo dos *loops* e condições que devem ser testados.

Se não for possível a utilização de um *Debbuger*, existe uma outra alternativa, que embora seja mais trabalhosa, também funciona muito bem, e pode ser muito útil durante a realização dos testes de caminhos de execução. Ela consiste, na verdade, em mandar imprimir os valores das variáveis que devem ser verificadas, para que se possa acompanhar a sua alteração ao longo de um determinado trecho do programa. Por exemplo, vamos supor que se queira acompanhar a mudança de valor da variável Y do trecho de código abaixo :

```
Y = 0;
DO X = 1 TO 5;
    Y = Y + (X * X);
END;
```

Para que isso seja possível, adiciona-se um comando de escrita ao trecho de código, como mostrado a seguir :

```
Y = 0;
DO X = 1 TO 5;
    Y = Y + (X * X);
    PRINTF ("%-d\n", Y);
END;
```

Assim, a cada vez que Y for acionada dentro do *loop*, seu valor será impresso (na tela, ou no papel); o que possibilitará uma checagem mais eficiente dos valores que ele está assumindo durante a execução do programa.

Essa estratégia pode ser usada ao longo de todo o programa, e se ele for extenso, pode ser útil mandar imprimir além do valor da variável, a *sub-rotina* à que ela pertence, para que ela possa ser localizada mais facilmente no código.

Deve-se portanto, projetar casos de teste de caminhos de execução que descubram erros tais como: **(1)** comparação de diferentes tipos de dados; **(2)** operadores lógicos ou precedência incorretos; **(3)** erro de precisão das variáveis, tornando uma igualdade improvável quando na verdade ela deveria acontecer; **(4)** comparação ou variável incorreta; **(5)** término de *loop* impróprio ou inexistente; **(6)** falha para sair quando encontrada iteração divergente e **(7)** variáveis de *loop* impropriamente modificadas.

Um bom projeto também deve determinar que as condições de erro sejam antecipadas e que caminhos de tratamento de erros sejam estabelecidos para reorientar ou terminar o processamento de forma clara quando um erro ocorrer. Entre os erros em potencial que devem ser testados quando o tratamento de erros é avaliado encontram-se :

1) a descrição do erro é ininteligível;
2) o erro apontado não corresponde ao erro encontrado;
3) a condição de erro provoca intervenção no sistema antes do tratamento de erros;
4) o processamento das condições de exceção é incorreto e
5) a descrição do erro não oferece nenhuma informação que ajude na localização da causa do erro.

A atividade do teste dos limites deve ser a última (e provavelmente a mais importante) tarefa da etapa de teste de unidade. O software freqüentemente falha em seus limites. Ou seja, freqüentemente ocorrem erros quando o n-ésimo elemento de um *array* n-dimensional é processado; quando a i-ésima repetição de um *loop* com i passagens é invocada; quando o valor máximo ou mínimo permitido é encontrado. Os casos de teste que exercitam a estrutura de dados, o fluxo de controle e os valores logo abaixo e logo acima dos valores máximo e mínimo têm muita probabilidade de descobrirem erros.

Para o módulo "Validar Entrada do Painel" do Subsistema Gerenciador de Painel, pode-se, por exemplo, projetar casos de teste de unidade que verifiquem a interface que ele faz com o outros módulos, verificando se o número de *argumentos* de entrada é igual ao

número de *parâmetros*; se os *parâmetros* e os *argumentos* têm tipos e dimensões compatíveis, etc... Pode-se também, projetar casos de teste que exercitem os caminhos de execução, verificando, por exemplo, se a entrada "Run", está gerando as ações : "Run Start", se o robô estiver no estado "Manual"; "Run Resume", se o robô estiver no estado "Suspenso". Além disso, pode-se projetar casos de teste para acompanhar a alteração dos valores de uma determinada variável ao longo da execução do programa, por exemplo, acompanhar a variável "Estado do Robô" ao longo do programa e verificar se ela está sendo corretamente alterada etc...

3.2.7.2.2 - Projeto de Casos de Teste de Aceitação

É difícil para o desenvolvedor do software prever como o usuário realmente usará o programa. As instruções de uso podem ser mal interpretadas, combinações estranhas de dados podem ser regularmente usadas, saídas que pareciam claras ao desenvolvedor podem ser ininteligíveis para o usuário em campo (Pressman,95).

Portanto, quando um software é construído, uma série de testes de aceitação deve ser realizada, para capacitar o usuário a validar todos os requisitos. A validação dos requisitos é feita pelo usuário final, e não pelo desenvolvedor do sistema; e para um sistema de pequeno porte consiste de um *test drive* informal. Sendo assim, casos de teste devem ser projetados para que o usuário final possa avaliar a conformidade do sistema com os requisitos estabelecidos por ele, no início do desenvolvimento.

No caso do Subsistema Gerenciador de Painel, deve-se projetar casos de teste de aceitação que verifiquem a interação com os outros subsistemas do Sistema Controlador de um Robô, já que ele não interage diretamente com o usuário final. Para realizar esse tipo de teste, no caso do Subsistema Gerenciador de Painel, deve-se simular os outros subsistemas por *stubs*, pois como já dito, o Subsistema Gerenciador de Painel faz parte de um sistema maior, que é o Sistema Controlador de um Robô, onde cada *subsistema* pode ter velocidade própria de desenvolvimento.

3.2.8 - Implementação do Sistema de Pequeno Porte Usando Técnicas Estruturadas

Todos os passos para o desenvolvimento de software que foram apresentados até este ponto estão dirigidos a um objetivo final : traduzir representações de software para uma forma que possa ser "entendida" pelo computador. A etapa de codificação é portanto, um processo que transforma o resultado do projeto numa linguagem de programação.

O passo de tradução inicial - desde o projeto detalhado do software para uma concepção de linguagem de programação - é uma preocupação primária no contexto da engenharia de software, pois "ruídos" podem entrar no processo de tradução de muitas maneiras; e por isso nessa fase é muito importante padronizar o que está sendo feito e ter um bom controle do desenvolvimento, pois isso pode ajudar muito a se obter um sistema que tenha mais garantia de manutebilidade, confiabilidade e reusabilidade.

Existem algumas providências que podemos tomar para fazer com que a etapa de codificação não afete a qualidade do sistema de uma maneira negativa. Por exemplo, uma das preocupações primordiais nessa fase deve ser a padronização e a documentação do código fonte (Pressman,1995).

Além disso, outro fator que pode ajudar a se obter um código mais limpo é usar o número mínimo de programadores possível, pois isso torna mais fácil para se fazer um controle de qualidade no código sendo gerado (Joch,1995).

Outro ponto interessante é a utilização de ferramentas automatizadas que ajudem no processo de codificação. Deve-se lançar mão delas sempre que possível, pois elas podem facilitar muito o trabalho, principalmente no sentido de ajudar a se policiar o que está sendo feito.

A seguir, mostraremos alguns cuidados que devem ser tomados ao se implementar os módulos para se obter as características de qualidade citadas anteriormente. Esses

cuidados relacionam-se principalmente com o estilo de programação que deve ser adotado, pois um código padronizado é muito mais fácil de ser entendido, mantido, alterado e reusado. O estilo de codificação abrange uma filosofia de codificação que salienta a simplicidade e a clareza. Entre os elementos de estilo incluem-se a documentação interna (nível de código fonte) e externa dos módulos, métodos para declaração de dados, e uma abordagem à construção das instruções.

3.2.8.1 - Documentação Externa do Módulo

Antes de se começar a programação propriamente dita de um módulo, deve-se fazer uma documentação externa que deve aparecer no início de cada módulo, como se fosse um cabeçalho. O formato de tal documentação deve ser :

1) Uma declaração de propósitos que indique a função do módulo.

2) Uma descrição de interface que inclua :

a) uma amostra da "seqüência de chamada";

b) uma descrição de todos os *argumentos* e

c) uma lista de todos os módulos subordinados.

3) Uma discussão de dados pertinentes, tais como variáveis importantes e suas restrições e limitações de uso, e outras informações importantes.

4) Um histórico de desenvolvimento que inclua :

a) o nome do autor do módulo;

b) o nome do revisor (auditor) e data, se houver e

c) datas e descrições das modificações.

Um exemplo de documentação externa para um módulo é mostrado na Figura 3.35.

NOME : Validar Entrada do Painel

PROPÓSITO : Validar as entradas do painel, de acordo com o estado do robô.

CHAMADA AMOSTRA : Validar_Entrada_Painel (Entrada, &Estado, &Ação, &Flag)

ENTRADAS : Entrada = Entrada do painel

SAÍDAS : Estado = Estado do Robô
Acao = Ação que deverá ser executada
Flag = Variável que diz se a entrada é ou não válida

SUB-ROTINAS REFERENCIADAS : ----------

DADOS PERTINENTES :

Dependendo do estado do robô, a Entrada poderá ou não ser válida.

Sendo ela válida, também dependendo do estado do robô, podem ser geradas ações diferentes, e portando a variável Acao assumirá valores também diferentes.

Se a Entrada for um novo programa, então Acao será o número do programa escolhido.

Após uma entrada válida, o estado do robô deve ser alterado (menos para um novo programa).

OBS: Valores assumidos pelas variáveis :

Entrada :	P = Power	**Estado:**	D = Desligado	**Acao:**	L = Power On	E = End
	R = Run		M = Manual		D = Power Off	S = Stop
	E = End		E = Executando		I = Run Start	1,2,3 = Programas
	S = Stop		S = Suspenso		R = Run Resume	

Flag: 0 = Entrada Inválida, 1 = Entrada Válida

AUTOR :
AUDITOR : ----------
DATA : 09/12/97
MODIFICAÇÕES : ----------

Fig. 3.35 - Documentação Externa para um Módulo.

3.2.8.2 - Codificação de um Módulo

Existem algumas medidas que podem ser tomadas durante a codificação de um módulo para torná-lo mais fácil de ser compreendido e alterado. Dentre essas medidas podemos

citar a escolha adequada de nomes identificadores (variáveis e rótulos), a padronização da declaração de dados, o formato claro das instruções de programa e a documentação interna do código do módulo através de comentários.

3.2.8.2.1 - Escolha de Nomes Identificadores Significativos

Deve-se selecionar nomes identificadores significativos, pois isso é crucial para facilitar a compreensão do código. Consideremos as três abstrações seguintes :

D = V*T
DIST = VELHOR*TEMPO
DISTÂNCIA = VELOCIDADE.HORIZONTAL * TEMPO.PERCORRIDO.EM.SEGS

A primeira expressão é inegavelmente concisa, mas o significado de D=V*T é impreciso, a menos que o leitor tenha informações prévias. A segunda expressão oferece mais informações, mas o significado de DIST e VELHOR poderia ser mal interpretado. Já a última expressão deixa poucas dúvidas em relação ao significado do cálculo.

Essas instruções ilustram a maneira pela qual os identificadores podem ser escolhidos para ajudarem a documentar o código.

Pode-se argumentar que expressões prolixas (como a última expressão acima) obscurecem o fluxo lógico e dificultam modificações. Obviamente, o bom senso deve ser aplicado quando os identificadores são selecionados. Identificadores desnecessariamente longos, na verdade, constituem um potencial para erros. Estudos indicam, porém, que, mesmo para programas pequenos, identificadores significativos melhoram a compreensão.

3.2.8.2.2 - Declaração e Inicialização de Dados

Uma série de diretrizes relativamente simples pode ser estabelecida a fim de tornar os dados mais compreensíveis e a sua manutenção mais simples.

Por exemplo, a ordem da declaração de dados deve ser padronizada, mesmo que a linguagem de programação não tenha requisitos obrigatórios. Por exemplo, a organização das declarações para um módulo FORTRAN poderia ser :

1) Todas as declarações explícitas (todas as variáveis devem ser declaradas) :
INTEGER, REAL, DOUBLE PRECISION

2)Todos os blocos de dados globais :
COMMON/ Block-name/...

3) Todos os *arrays* locais :
DIMENSION nomes e dimensões dos *arrays*

4) Todas as declarações de arquivo :
DEFINE FILE, OPEN, CLOSE

A disposição em ordem torna os atributos mais fáceis de encontrar, agilizando a atividade de testes, *depuração* e manutenção.

Quando múltiplos nomes de variáveis são declarados com uma única instrução, uma disposição em ordem alfabética dos nomes é valiosa. Similarmente, dados globais com rótulos (por exemplo, blocos COMMON FORTRAN) devem ser ordenados alfabeticamente, porém, deve-se lembrar que seu uso deve ser evitado, pois eles destróem a modularidade do programa, aumentando muito sua complexidade.

Na realidade, deve-se evitar ao máximo o uso de *variáveis globais*, pois elas podem ser uma fonte grande de erros que nem sempre são facilmente detectados. Além disso, o seu uso obscurece o código, já que o programador precisa conhecer o comportamento global da variável dentro do programa para poder usá-la adequadamente (Ledgard,1975).

Além da declaração, deve-se prestar atenção na inicialização dos dados, pois ela também é uma fonte comum de erros em programas. Veja o exemplo FORTRAN abaixo :

Sem Inicialização

```
      INTEGER SUM
      DO 10  I = 1, 100
10    SUM = SUM + 1
      AVG = FLOAT (SUM)/100.0
```

Com Inicialização

```
      INTEGER SUM
      SUM = 0
      DO 10  I = 1, 100
10          SUM = SUM + 1
      AVG = FLOAT(SUM)/100.0
```

No bloco de código da esquerda a variável "SUM" não foi inicializada. A linguagem FORTRAN não assume o valor 0 para as variáveis não inicializadas, o que significa que o código da esquerda pode levar a um resultado errôneo.

Portanto, não se deve assumir que a linguagem de programação tenha certas convenções *default*. Quando houver dúvida a esse respeito o melhor a fazer para evitar erros é uma inicialização cuidadosa das variáveis em uso.

3.2.8.2.3 - Construção das Instruções de Programa

A construção de instruções deve ser fiel a uma regra fundamental : cada instrução deve ser simples e direta.

Muitas linguagens de programação não tem um formato rígido para o texto do programa. Por exemplo, muitas permitem que as expressões sejam arbitrariamente espaçadas, permitindo que se tenha muitas instruções por linha. Os aspectos de economia

de espaço dessa característica dificilmente são justificados pela legibilidade ruim que resulta. Consideremos os dois seguimentos de código seguintes :

```
DO I = 1 TO N-1; T = I; DO J = 1 + 1 TO N; IF A(J) < A(T) THEN DO T = J; END;
IF T <> I THEN DO H = A(T); A(T) = A(I); A(I) = T; END; END;
```

A estrutura de laço (*loop*) e as operações condicionais contidas no segmento acima são mascaradas pela construção de múltiplas instruções por linha. Reorganizando-se a forma do código temos :

```
DO I = 1 TO N-1;
    T= I;
    DO J = I + 1 TO N;
            IF A(J) < A(T) THEN DO
               T = J;
            END;
    IF  T <> I  THEN DO
        H = A(T);
        A(T) = A(I);
        A(I) = T;
    END;
END;
```

Aqui, a construção de instruções e identação simples iluminam as características funcionais e lógicas do segmento.

Portanto, ao se escrever o código de um módulo deve-se organizar as instruções de uma maneira clara, posicionando os corpos de instruções de repetição como o "DO", os corpos de instruções de decisão como o "IF", e os corpos de *sub-rotinas* mais para a direita do cabeçalho do bloco de programa, como mostra o exemplo anterior.

Além disso, deve-se evitar usar instruções de transferência de controle incondicional do tipo "GOTO", pois elas obscurecem o código. Em vez de usar instruções desse tipo deve-se substituí-las sempre que possível pelas estruturas da programação estruturada. Veja o exemplo FORTRAN abaixo :

Usando a Instrução "GOTO"

```
      INTEGER S
      S = 0
      I = 1
10  IF (I .GT. 100) GOTO 20
          S = S + 1
          I = I + 1
          GOTO 10
20 END
```

Eliminando o "GOTO" em Favor do "DO"

```
      INTEGER S
      S = 0
      DO 10 I = 1, 100
10          S = S + 1
```

Note que a eliminação do "GOTO" em favor do "DO" tornou o código mais claro e fácil de compreender.

Além disso, as instruções de código fonte individuais podem ser simplificadas tomando-se algumas medidas preventivas como (Pressman,1995) : evitar o uso de testes condicionais complexos; eliminar os testes em condições negativas; evitar um intenso aninhamento de laços ou condições; usar parênteses para esclarecer expressões lógicas ou aritméticas; usar símbolos de espaçamento e/ou legibilidade para esclarecer o conteúdo da instrução; e pensar : será que eu poderia entender isto se não tivesse sido eu a pessoa que o codificou ?

3.2.8.2.4 - Documentação Interna do Código

A documentação interna do código nada mais é que a colocação e composição de comentários que devem estar embutidos no corpo do código fonte, e que são usados para descrever as funções de processamento. Os comentários devem:

1. Comentar cada linha ou blocos de instrução, isto é expressar em português o significado de uma ou mais linhas de código. De acordo com Emden (1992), a pessoa que estiver fazendo os comentários no código deve decidir quantas linhas expressar de uma vez. Não é aconselhável que esse número exceda a duas ou três linhas, porém isso depende da complexidade do código.

2. Usar linhas em branco ou recuos, para que os comentários possam ser prontamente distinguidos do código.

3. Serem corretos, um comentário incorreto ou enganoso é pior do que nenhum comentário.

A seguir são mostrados alguns exemplos de comentários de código. O primeiro exemplo refere-se a um código em Visual Basic. Os comentários aparecem precedidos pelo símbolo " ' ".

```
Private Sub Command1_Click()

'*********************************************************************
'                        DECLARACAO DAS VARIAVEIS
'*********************************************************************
    Dim I As Integer, J As Integer

    For I = 1 To 10
'*********************************************************************
'                          IMPRIMINDO LINHA
'*********************************************************************
      For J = 1 To 10
        Print J;
      Next J
'*********************************************************************
'                     PASSANDO PARA PROXIMA LINHA
'*********************************************************************
      Print
    Next I
End Sub
```

O próximo exemplo refere-se a um código em C. Os comentários aparecem entre os símbolos " /* " e " */ ".

```
#include <stdio.h>

void main()
{
/*************************************************************************
                        DECLARACAO DAS VARIAVEIS
*************************************************************************/
 int c;

/*************************************************************************
                    PEGANDO UM CARACTER DO TECLADO
*************************************************************************/
 c = getchar();
 while (c != EOF)
 {
/*************************************************************************
              IMPRIMINDO O CARACTER E PEGANDO O PROXIMO
*************************************************************************/
  putchar(c);
  c = getchar();
 }
}
```

O último exemplo refere-se a um código em Pascal. Os comentários aparecem entre os símbolos " { " e " } ".

```
PROGRAM SOMA;
{*************************************************************************}
                        DECLARACAO DAS VARIAVEIS
*************************************************************************}
VAR MAT : ARRAY [1..4, 1..5] OF INTEGER;
    SOMALINHA, I, J, TOTAL : INTEGER;

BEGIN
TOTAL : = 0;
FOR I : = 1 TO 4 DO
    BEGIN

{*************************************************************************
              CALCULO DA SOMA DOS ELEMENTOS DA LINHA
*************************************************************************}
    SOMALINHA : = 0;
    FOR J : = 1 TO 5 DO
          SOMALINHA : = SOMALINHA + MAT [ I , J ];
{*************************************************************************
              CALCULO DA SOMA DE TODOS OS ELEMENTOS
*************************************************************************}
    TOTAL : = TOTAL + SOMALINHA;
    END;
END.
```

CAPÍTULO 4

CONSTRUÇÃO DO SISTEMA DE PEQUENO PORTE USANDO O PARADIGMA DA ORIENTAÇÃO A OBJETOS

Ao longo deste Capítulo forneceremos diretrizes para se construir um Sistema de Pequeno Porte usando o paradigma da orientação a objetos (para mais detalhes sobre o paradigma ver o item **2.10.2 - Paradigma da Orientação a Objetos**, no **Capítulo 2 - Software e Engenharia de Software**, desta dissertação). Estaremos neste Capítulo, portanto, traçando uma rota que pode ser seguida por aqueles que queiram construir um sistema desse tipo.

A primeira etapa do desenvolvimento de um sistema de software é a construção do modelo descritivo e do modelo conceitual do sistema (para mais detalhes sobre modelos de abstração, ver o item **2.9 - Modelagem e Abstração** no Capítulo 2 desta dissertação), que no paradigma da orientação a objetos corresponde à construção do modelo de objetos.

Pode-se identificar quatro atividades principais na construção do modelo de objetos de um sistema :

1) identificar os propósitos e as características do sistema;
2) estabelecer um modelo de componentes para o sistema;
3) selecionar e estabelecer responsabilidades para as classes&objetos do sistema e
4) adicionar as interações entre os objetos.

A primeira atividade, é nada mais nada menos que fazer a especificação inicial do sistema. Como realizar esta atividade está explanado no item **3.1 - Definição do Sistema**, no **Capítulo 3 - Sistemas de Pequeno Porte**, desta dissertação. A especificação dada como exemplo nesse item (Especificação do Subsistema Gerenciador

de Painel) não será adotada neste Capítulo, pois o Paradigma da Orientação a Objetos não usa os mesmos critérios que o Paradigma Clássico para quebrar o sistema em subsistemas. Na verdade, no caso do Sistema Controlador de um Robô, não será nem mesmo possível quebrá-lo em subsistemas dentro desse paradigma, e por isso serão adotados neste Capítulo outros exemplos mais adequados. As demais atividades serão explanadas a seguir.

4.1 - Estabelecendo um Modelo de Componentes para o Sistema

O próximo passo na construção do modelo de objetos do sistema é dividi-lo em componentes; pois eles nos guiam na procura dos objetos e nos ajudam a organizar o trabalho. São eles :

- **Componente Domínio do Problema (CDP).** O Componente Domínio do Problema contém objetos que correspondem diretamente ao problema sendo modelado. Os objetos nesse componente são independentes da tecnologia de implementação. Eles têm pouco conhecimento sobre os objetos dos outros componentes.

- **Componente Interação Humana (CIH).** O Componente Interação Humana contém objetos que fazem interface entre os objetos do domínio do problema e os usuários. Num modelo de objetos, tais objetos freqüentemente correspondem a janelas e relatórios.

- **Componente Interação com outros Sistemas (CIS).** O componente Interação com outros Sistemas contém objetos que fazem interface entre os objetos do domínio do problema, e outros sistemas ou dispositivos. Um objeto de interação com outro sistema encapsula protocolos de comunicação, deixando os objetos do domínio do problema livres desses detalhes de implementação.

- **Componente Gerenciamento de Dados (CGD).** O Componente Gerenciamento de Dados contém objetos que fazem interface entre os objetos do domínio do problema que precisam de suporte para persistência e recuperação, e os *sistemas de banco de dados* ou de gerenciamento de arquivos.

- **Componente Gerenciamento de Cenários (CGC).** O Componente Gerenciamento de Cenários contém objetos coordenadores que cuidam da criação dos objetos dos outros componentes num determinado cenário. Um objeto do gerenciamento de cenários, portanto, encapsula a coordenação da criação dos objetos de um cenário, deixando os outros objetos livres desse tipo de comportamento (Cunha,1997).

Então, o modelo de objetos do sistema é formado de cinco grandes componentes, como mostra a Figura 4.1 adiante.

Cada componente por sua vez é constituído de objetos, que juntos, compõe os objetos do modelo de objetos do sistema.

Nem sempre é necessário desenvolver todos os componentes do modelo de componentes, e só os componentes que forem relevantes para o sistema em questão devem ser desenvolvidos. Por exemplo, só é relevante para um sistema desenvolver o Componente Gerenciamento de Dados, se o sistema precisar armazenar e recuperar dados ao longo do tempo, caso contrário não faz sentido tê-lo no modelo.

Um sistema de pequeno porte, considerado nesse trabalho, não tem interação com outros sistemas e dispositivos, portanto não é necessário nesse caso desenvolver o Componente Interação com Outros Sistemas. Isso será abordado no **Capítulo 6 - Construção do Sistema de Médio Porte Usando o Paradigma da Orientação a Objetos**, desta dissertação.

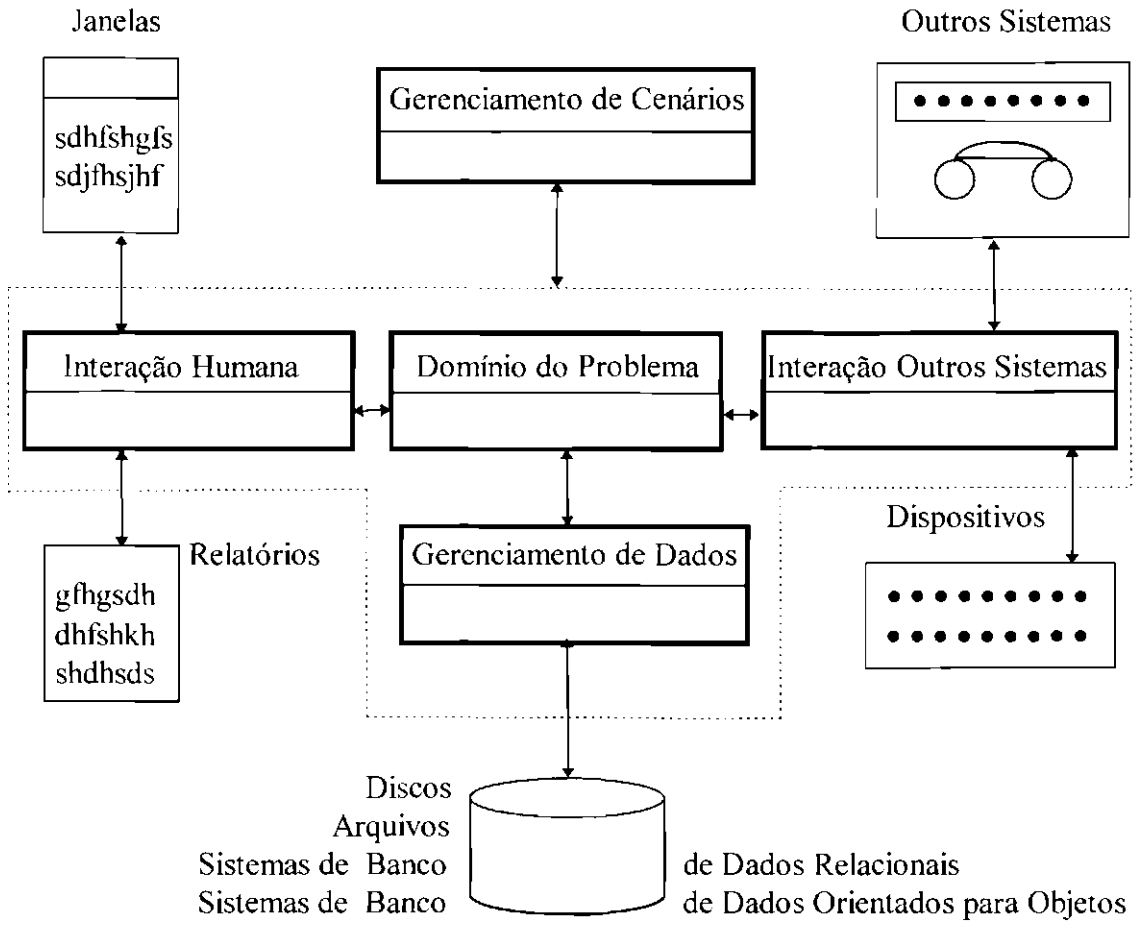


Fig. 4.1 - Modelo de Componentes do Sistema.

FONTE : Adaptada de Coad (1997, p. 6).

A seguir apresentaremos estratégias e padrões que irão anos ajudar a selecionar e estabelecer responsabilidades para os objetos, e a adicionar as interações entre os objetos, para os componentes Domínio do Problema, Interação Humana, Gerenciamento de Dados, e Gerenciamento de Cenários.

4.2 - Selecionando e Estabelecendo Responsabilidades para as Classes&Objetos

Quais objetos fazem parte do modelo de objetos do sistema ? Por onde começar a procurá-los ?

Pode-se começar a localização dos objetos do sistema por qualquer componente. Na verdade, a localização dos objetos de cada componente deve ser vista como sendo atividades, e não passos seqüenciais; pois na prática pode-se identificar objetos pertencentes a qualquer um dos componentes de uma só vez. O que se torna importante é classificar esses objetos encontrados e posicioná-los no componente adequado.

Nesta dissertação começaremos a localização dos objetos pelo componente domínio do problema, pois acreditamos ser essa a maneira mais didática de introduzirmos os conceitos envolvidos na construção do modelo de objetos de um sistema.

Porém, se for necessário um retorno rápido de resultados para o usuário, aconselha-se começar a procura e estabelecimento das responsabilidades dos objetos pelo componente interação humana; pois dessa maneira os resultados poderão ser gerados mais rapidamente, já que a interface pode ser implementada e apresentada para o usuário de maneira simulada, antes do sistema todo ter sido projetado.

4.2.1 - Notação para Classes e Objetos

No modelo de objetos, um objeto é representado graficamente pelo símbolo de Classe&Objeto proposto por Coad (1992), como mostra a Figura 4.2 abaixo. Esse símbolo representa, na verdade, um conjunto de objetos e a classe à que eles pertencem.

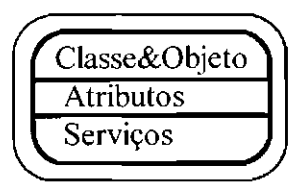


Fig. 4.2 - Notação para Classe&Objeto.

A classe é representada pelo retângulo arredondado em negrito dividido em três seções horizontais; e seu(s) objeto(s) representado(s) pelo retângulo mais leve. Na seção

superior deve-se colocar o nome da classe, na seção do meio os atributos aplicáveis a cada objeto na classe, e na inferior, os serviços aplicáveis a cada objeto na classe.

O nome da classe deve ser composto de um substantivo, ou de um substantivo seguido de um adjetivo. Ele deve descrever um objeto único na classe, por exemplo, quando cada objeto descreve algo sendo transportado, deve-se usar "Item-Transportado" (cada objeto é um item), em vez de "Transporte" (que deve descrever um transporte inteiro, como por exemplo um transporte rodoviário ou um transporte aéreo).

Uma variação do símbolo Classe&Objeto é o símbolo Classe, mostrado na Figura 4.3 a seguir.

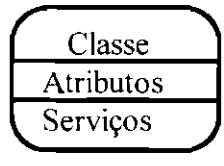


Fig. 4.3 - Notação para Classe.

4.2.2 - Localizando Objetos do Componente Domínio do Problema

Numa primeira abordagem, deve-se procurar no domínio do problema (para mais detalhes sobre o que é domínio do problema ver o item **2.5 - Sistema de Software e Domínio do Problema**, no Capítulo 2 desta dissertação) por muitos objetos, construindo-se uma lista dos objetos potenciais.

Para localizar esses objetos potenciais deve-se lembrar que objetos podem ser melhor entendidos pensando-se neles na primeira pessoa. Por exemplo: eu sou um objeto , eu sei informações a meu respeito, eu conheço outros objetos com os quais me relaciono, eu realizo serviços para mim e para outros objetos. Portanto, deve-se pensar sobre um objeto na primeira pessoa, falar sobre um objeto na primeira pessoa, isto é, o desenvolvedor deve se colocar no problema (Coad,1993B).

Procurar especificamente por objetos do domínio do problema significa ler a definição do sistema, procurando por objetos que estejam relacionados com o sistema em si, ou seja, objetos que correspondam diretamente ao problema sendo modelado; sempre lembrando que os objetos desse componente devem ser identificados independentemente da tecnologia de implementação a ser adotada.

Portanto, considerações sobre arquiteturas de sistemas de computadores (centralizadas, distribuídas, duplicadas), *drives* de disco, monitores de vídeo, e outras afins, devem ser feitas nas atividades de implementação.

Então, para selecionar objetos do domínio do problema deve-se procurar na definição do sistema, por objetos que o sistema deva conhecer para conseguir cumprir suas responsabilidades. Portanto, deve-se procurar por (Coad,1997) :

1) Atores. Um ator é uma pessoa ou uma organização que age ativamente no sistema realizando ações importantes para o sistema. Exemplos de atores podem ser, por exemplo, "comprador", "vendedor", "fornecedor" etc...

2) Lugares. Deve-se procurar por lugares que possam conter outros objetos, por exemplo, "banco", "hospital", "aeroporto", "região" etc...

3) Objetos Tangíveis. Deve-se procurar por objetos tangíveis usados no domínio do problema, como por exemplo, "caixa registradora", "procedimento", "produto", "horário", "plano", "item" etc...

4) Eventos Lembrados. Deve-se procurar por eventos lembrados, ou seja, eventos que o sistema deva se lembrar através do tempo. Um evento lembrado é, portanto, um momento no tempo; por exemplo uma venda, ou um intervalo de tempo, como um aluguel. Outros exemplos de eventos lembrados podem ser : acordo, autorização, contrato, entrega, depósito, pagamento, reserva, etc... Um objeto do

tipo evento lembrado sabe sobre um evento significativo, sabe os atuantes que participam desse evento, e processa informações relacionadas a esse evento. Os eventos lembrados podem vir de uma janela (o evento é baseado na interação humana em algum ponto do tempo); de outro objeto que esteja monitorando para encontrar um evento significativo, e guardando o evento ocorrido; e de outro sistema, um que o sistema em questão possa interagir com eventos lembrados que ele guarda. Então, para localizar eventos lembrados deve-se perguntar : Quais são os eventos notáveis que devem ser guardados pelo sistema ?

4.2.3 - Estabelecendo Responsabilidades para os Objetos do Domínio do Problema

Cada objeto em um modelo de objetos tem responsabilidades específicas (Coad,1997) :

- um objeto sabe informações a seu respeito;
- um objeto realiza serviços, por si mesmo ou em colaboração com outros objetos e
- um objeto conhece outros objetos.

Deve-se portanto estabelecer responsabilidades para cada objeto que reflitam o que ele sabe, o que ele faz, e quem ele conhece.

Os atributos de um objeto refletem "o que" ele sabe sobre si mesmo, os serviços de um objeto refletem "o que" ele "faz" para ele e para os outros objetos com que ele interage, e as conexões entre os objetos refletem "quem" eles "conhecem".

4.2.3.1 - Estabelecendo Atributos para os Objetos

Um atributo é um dado para o qual cada objeto de uma classe tem seu próprio valor. Pode-se dizer então, que os atributos de um objeto refletem o que ele "sabe" a seu respeito.

Para identificar atributos é fundamental que se pense como um objeto, isto é, o desenvolvedor deve se colocar no lugar do objeto e pensar : "Eu sou um objeto, eu sei informações que descrevem o objeto real do qual eu sou uma abstração (Coad,1993B)."

Pensando dessa maneira, cada objeto deve perguntar a si mesmo (Coad,1992) :

- Como sou descrito em geral ?
- Como sou descrito nesse domínio de problemas ?
- Como sou descrito no contexto das responsabilidades do sistema ?
- O que preciso saber ?
- De quais informações de estado preciso sempre me lembrar ?
- Quais podem ser meus estados ?

Ao estabelecer atributos deve-se preocupar em fazer com que cada atributo capture um "conceito de atomocidade", ou seja, um valor único (por exemplo, número do vôo), ou um agrupamento de valores fortemente relacionados (por exemplo, endereço).

Ao escolher nomes para os atributos deve-se procurar também ser mais abrangente e menos específico (por exemplo, deve-se escolher "Combustível" ao invés de "Diesel"), pois os nomes específicos são menos estáveis.

Abaixo segue-se uma relação dos atributos mais comuns para os objetos do domínio do problema. É lógico que dependendo da aplicação os atributos irão variar, mas basicamente pode-se ter :

Atores. Para atores normalmente considera-se atributos do tipo nome, endereço, telefone etc...

Lugares. Para lugares normalmente considera-se atributos do tipo número, nome, endereço (talvez latitude, longitude, altitude).

Eventos Lembrados. Para eventos lembrados normalmente considera-se atributos do tipo número, data, tempo, estado.

Objetos Tangíveis. Para objetos tangíveis normalmente considera-se atributos do tipo número, nome, estado operacional etc...

4.2.3.2 - Estabelecendo Serviços para os Objetos

Os serviços detalham ainda mais a abstração da realidade sendo modelada, indicando qual comportamento será oferecido por um objeto de uma classe. Pode-se dizer então que os serviços refletem o que um objeto "faz".

Para identificar serviços é fundamental que se pense como um objeto, isto é, o desenvolvedor deve se colocar no lugar do objeto e pensar : "Eu sou um objeto, eu faço serviços que são normalmente feitos para o objeto real do qual eu sou uma abstração (Coad,1993B)."

Pensando dessa maneira, cada objeto deve perguntar a si mesmo (Coad,1997) :

- Quais questões posso responder ?
- O que eu posso fazer baseado no que eu sei ?
- Que cálculos eu posso fazer ?
- Que monitoramento eu posso fazer ?

Os nomes dos serviços devem ser compostos de um verbo, ou um verbo seguido de um complemento. Os complementos devem especificar os nomes dos serviços de acordo com o domínio do problema. Por exemplo, para o cálculo de uma taxa de um evento legal pode-se usar "CalcularTaxa", para a monitoração de um sensor pode-se usar "MonitorarCondiçãoDeAlarme".

Para os serviços que possam ser aplicados em diferentes intervalos de tempo, pode ser útil adicionar "sobre um intervalo" complementando o seu nome. Por exemplo, "CalcularTaxaSobreUmIntervalo".

Deve-se colocar os serviços na classe que "sabe" o suficiente para realizá-los. Ou seja, deve-se colocar os serviços juntos com os atributos que eles modificam, pois assim, obtem-se objetos que são responsáveis por si mesmos. Por exemplo, obtem-se um objeto "Sensor" que monitora a si mesmo, um objeto "Consumidor" que qualifica a si mesmo, um objeto "alvo" que destrói a si mesmo.

Existem alguns serviços básicos que se aplicam a todos os objetos no modelo. São eles:

1) **Criar.** Este serviço cria e inicializa um novo objeto em uma classe.
2) **ConectarAo.** Este serviço conecta um objeto a outro objeto.
3) **DesconectarDe.** Este serviço desconecta um objeto de outro objeto.
4) **Acessar.** Este serviço obtém os valores de atributo de um objeto.
5) **Setar.** Este serviço modifica os valores de atributo de um objeto.
6) **Liberar.** Este serviço libera (desconecta e elimina) um objeto.

Esse tipo de serviço não deve ser mostrado explicitamente no diagrama de classe&objeto do modelo conceitual, ou seja, eles devem ser tratados nesse nível de modelagem como serviços implícitos a todos os objetos do modelo.

4.2.3.3 - Procurando por Estruturas de Generalização-Especialização

A estrutura de generalização-especialização (Gen-Espec) deve ser usada para a "distinção entre classes". Nessas estruturas pode-se aplicar a herança, com uma representação dos atributos e serviços mais gerais, seguidos por especializações pertinentes. Um exemplo de estrutura de generalização-especialização é dado na Figura 4.4 a seguir.

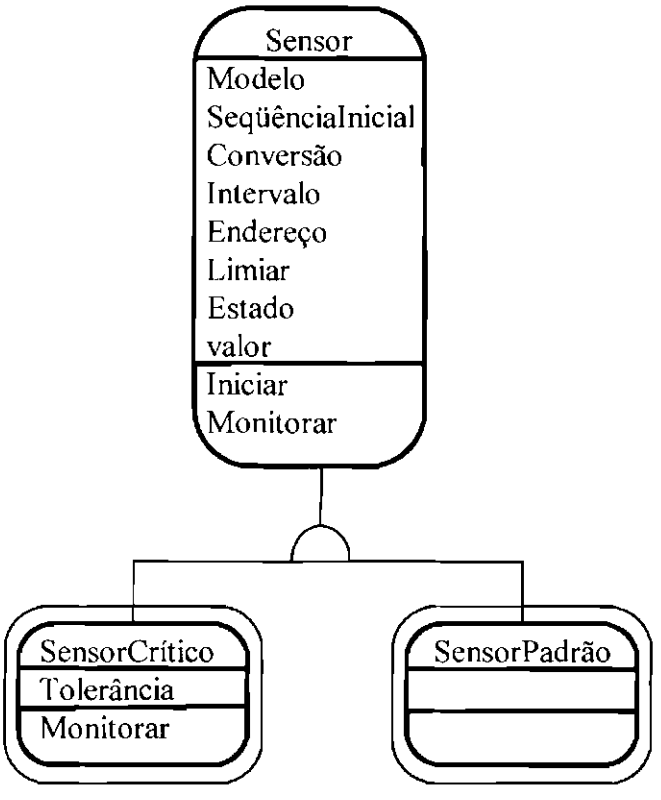


Fig. 4.4 - Estrutura de Generalização-Especialização.

A estrutura de Gen-Espec tem uma classe de "generalização" no topo (no exemplo anterior "Sensor"), e classes de "especialização" abaixo (no exemplo anterior "SensorCrítico" e "SensorPadrão"), com linhas desenhadas entre elas. Um semi-círculo denota que as classes estão formando a estrutura Gen-Espec. Os pontos finais de uma linha de estrutura Gen-Espec são posicionados de forma a refletir um mapeamento entre as classes, e não entre objetos.

Cada especialização deve ser nomeada de forma a ser auto-explicativa. Um nome apropriado para uma especialização deve ser, sempre que possível, o(s) nome(s) de sua(s) generalização(ções) correspondente(s), acompanhado(s) por um nome qualificador que descreva a natureza da especialização. Por exemplo, para a generalização "Sensor" da Figura 4.4, as especializações "SensorCrítico" e "SensorPadrão" devem ser preferidas em relação a apenas "Crítico" e "Padrão".

Numa estrutura Gen-Espec, a classe de generalização pode não conter objetos diretos, como acontece com a classe "Sensor" do exemplo da Figura 4.4. Nesse caso "Sensor" é um "Sensor Crítico" ou um "Sensor Padrão"; e "Sensor" é uma classe sem um objeto diretamente correspondente. Em vez disso, os objetos correspondentes são refletidos nos objetos das Classes&Objetos especializadas.

Como já dito, deve-se usar a generalização-especialização para mostrar, baseado no domínio do problema, o que é "igual" e o que é "diferente" numa aplicação. Por exemplo: equipamento, tipos de equipamento; participantes, tipos de participantes etc..

Portanto, para identificar estruturas Gen-Espec deve-se checar se existem atributos potenciais que pertençam somente a certos objetos em uma classe, ou se existem atributos que contenham um valor não aplicável. Se isso acontecer, deve-se colocar o atributo especializado em uma classe de especialização.

Deve-se checar também se existem serviços potenciais que se aplicam somente a certos objetos numa classe, ou se existem serviços que primeiramente testem uma condição, e depois ajam de acordo com o resultado obtido. Se isso acontecer, deve-se colocar o serviço especializado em uma classe de especialização (Coad,1997).

Uma boa maneira de identificar se existem atributos e serviços adicionais é, para duas ou mais classes de especialização, perguntar o que é igual, e o que é diferente entre elas. Isto é, deve-se perguntar (Coad,1993B) :

- Como eu sou diferente no que eu sei ? Tais diferenças apontam para generalização-especialização.

- Como eu sou diferente no que faço ? Tais diferenças também apontam para generalização-especialização.

• Como eu sou diferente na apresentação de meus valores ? Essas diferenças são apenas um serviço de apresentação adicional, e não é necessário uma especialização para representá-las. Um exemplo disso é dado na Figura 4.5 a seguir.

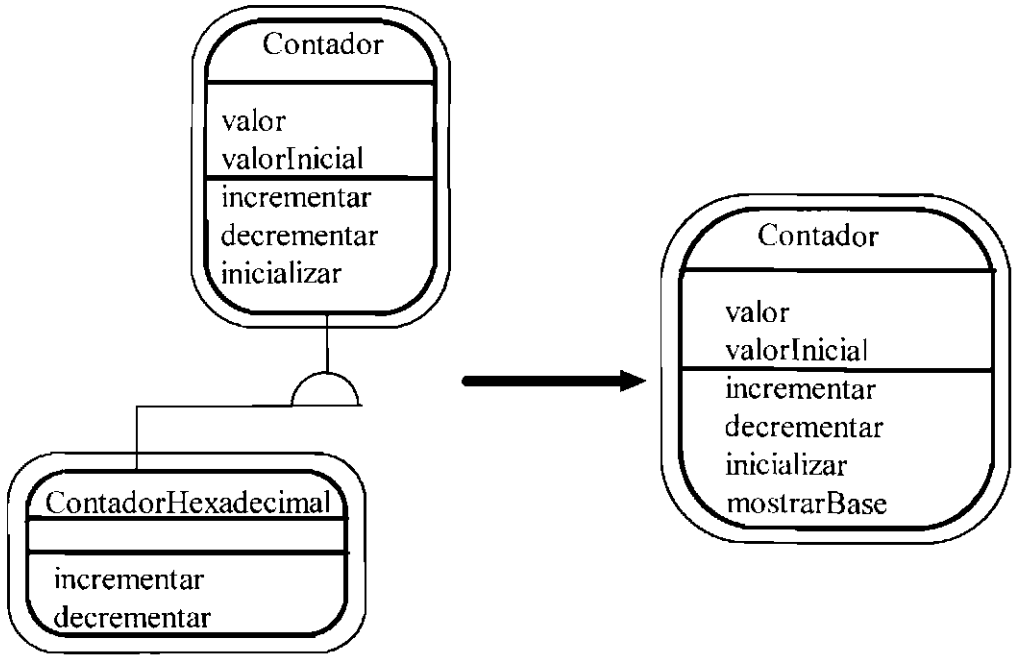


Fig. 4.5 - Diferença na Apresentação dos Valores.
FONTE: Adaptada de Coad (1993A, p.7).

Nesse exemplo, a especialização não é adequada pois as classes "Contador" e "ContadorHexadecimal" da situação 1, só apresentam diferenças na apresentação de seus valores, e por isso não há necessidade de se manter a classe "ContadorHexadecimal" no modelo. Em vez disso, foi adicionado a "Contador" o serviço "mostrarBase", que mostra o valor do contador na base hexadecimal, como mostra a situação 2.

Não se deve usar também a estrutura Gen-Espec apenas para extrair um atributo ou serviço comum. Por exemplo, um "consumidor" tem um nome, um "bote" tem um nome, então adiciona-se uma classe "Coisas-com-nome".

Para classes em uma estrutura Gen-Espec deve-se colocar os atributos e os serviços no ponto mais superior da estrutura possível, ou seja, onde eles permaneçam aplicáveis a cada uma de suas especializações. Se um atributo ou um serviço aplicar-se a um nível inteiro de especializações, então deve-se movê-lo para a generalização correspondente.

A principal vantagem de se usar as estruturas de generalização-especialização está em se aplicar o recurso de herança, ou seja, atributos e serviços generalizados podem ser identificados apenas uma vez, e depois especializados apropriadamente.

Normalmente as linguagens de programação orientadas para objetos fornecem uma biblioteca de classes que podem ser reaproveitadas e estendidas através do mecanismo de herança permitido pelas estruturas gen-espec; ou seja, pode-se definir especializações pertinentes ao domínio da aplicação para classes mais gerais contidas na biblioteca de classes, poupando trabalho e esforço de programação.

4.2.3.4 - Estabelecendo Conexões entre Objetos

As conexões entre os objetos, como já dito, refletem "quem" um objeto "conhece". Então, deve-se estabelecer as conexões que um objeto tem com outros objetos, para que se possa saber para "quem" mandar mensagens, e para poder responder perguntas sobre os objetos diretamente relacionados com o objeto em questão.

As conexões entre objetos podem ser de dois tipos : Conexões Todo-Parte e Associações.

4.2.3.4.1 - Estabelecendo Conexões Todo-Parte entre os Objetos

A conexão todo-parte, a partir da perspectivas do todo, pode ser pensada como uma conexão "tem um". Um exemplo de conexão todo-parte é dado na Figura 4.6 a seguir.

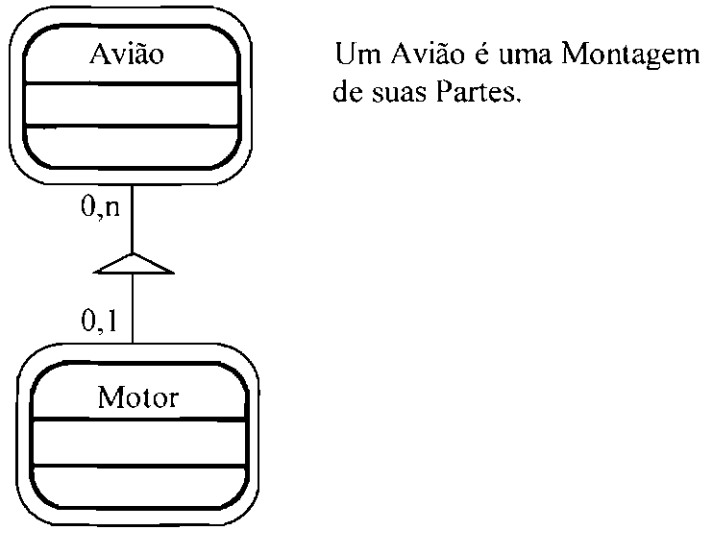


Fig. 4.6 - Conexão Todo-Parte.

A conexão todo-parte tem um objeto "Todo" no topo (no exemplo anterior um objeto da classe "Avião"), e um objeto "Parte" abaixo (no exemplo anterior um objeto da classe "Motor"), com uma linha desenhada entre eles. Um triângulo distingue os objetos que formam a conexão todo-parte. Os pontos finais de uma conexão todo-parte são posicionados para refletir um mapeamento entre objetos (não entre as classes).

Cada final de uma conexão todo-parte é marcado com uma restrição, ou seja, com uma quantidade (n) ou intervalo (m,n), indicando o número de "partes" que um "todo" pode ter, e vice-versa, em qualquer instante. Limites inferiores e superiores podem ser mostrados diretamente, como por exemplo "0,1" do exemplo anterior. Se um número fixo de conexões todo-parte tiver que ocorrer, uma quantidade única pode ser usada, como por exemplo "1", "4", "n".

Ao se estabelecer conexões todo-parte deve-se puxar trabalho (atributos e serviços) para as partes, distribuindo as responsabilidades. Um exemplo disso é mostrado na Figura 4.7 a seguir.

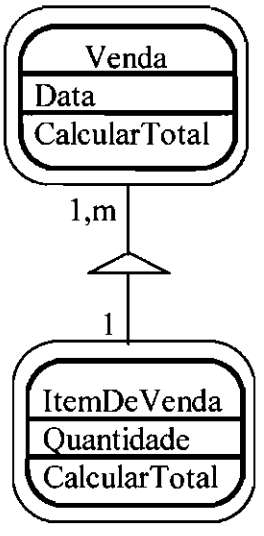


Fig. 4.7 - Distribuindo a Responsabilidade entre as Partes.

Neste exemplo, deve-se notar que como um "ItemDeVenda", eu sei calcular meu total individual; o que é muito melhor do que fazer com que "Venda" faça todo o trabalho de calcular o total por si mesma. Esse princípio é muito importante quando um todo tem diferentes tipos de partes, pois se ele fizer todo o trabalho vai estar cheio de estruturas do tipo *case*, *switch*, ou *if-then-else*, para ajudá-lo a decidir qual processamento aplicar a cada parte (Coad,1993B).

Para localizar conexões todo-parte potenciais deve-se procurar por (Coad,1992) :

- **Montagem - Partes.** Um exemplo de conexão todo-parte do tipo montagem-partes pode ser a classe "Avião" como a montagem, e a classe "Motor" como a parte, mostradas na Figura 4.6. Tal montagem existe fisicamente, e é observável (pode-se tocar em uma).

• **Recipiente - Conteúdos.** Um exemplo de conexão todo-parte do tipo recipiente-conteúdos é dado na Figura 4.8 a seguir. Neste exemplo, "Avião" é considerado como um recipiente, e os "Pilotos" estão dentro dele.

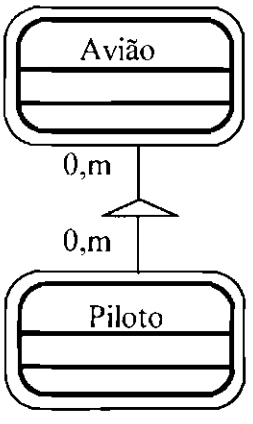


Fig. 4.8 - Conexão Todo-Parte, Recipiente-Conteúdos.

• **Conjunto - Membros.** Um exemplo de conexão todo-parte do tipo conjunto-membros é dado na Figura 4.9 a seguir. Neste exemplo "Vôo" é um conjunto de "Segmentos de Vôo".

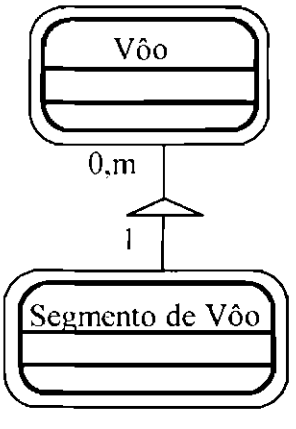


Fig. 4.9 - Conexão Todo-Parte, Conjunto-Membros.

Nas conexões todo-parte conjunto-membros tem-se apenas um modelo mental, onde o método conjunto-membro é aplicado para uma melhor administração da complexidade do domínio do problema. Por exemplo, na Figura 4.9, um vôo é formado por vários

segmentos de vôo. Não se pode tocá-los, nem observá-los; o que se tem na realidade é uma abstração mental.

Na fase de implementação as conexões todo-parte, até agora representadas apenas graficamente, serão representadas por atributos contendo o identificador (endereço) de cada objeto conectado. Por exemplo, cada objeto "todo" terá implicitamente um identificador para cada objeto "parte". Cada objeto "parte" terá um identificador para o seu "todo".

4.2.3.4.2 - Estabelecendo Associações entre os Objetos

As Associações representam mapeamentos entre objetos em um domínio de problema, que são necessários para que esses objetos cumpram suas responsabilidades.

A diferença entre uma Associação e uma Conexão Todo-Parte está na intensidade da semântica implícita, pois "Todo e Parte" é um dos serviços básicos de organização dos seres humanos, e tem um significado muito mais forte do que um simples mapeamento entre objetos em um domínio de problema, que é o caso da associação. Um exemplo de associação é dado na Figura 4.10 a seguir.

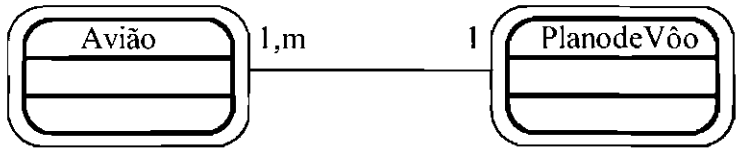


Fig. 4.10 - Associação.

Uma associação entre objetos é representada com uma linha desenhada entre os objetos, como mostra a Figura 4.10. As pontas de uma linha de associação indicam os mapeamentos entre os objetos individuais (em vez de entre classes).

Cada objeto deve ter restrições, ou seja, indicadores de quantidade (m) ou intervalo (m,n) em cada uma de suas associações, refletindo seu relacionamento com os outros objetos. Uma quantidade ou intervalo mostra o número de mapeamentos que podem ocorrer. Da mesma maneira que nas conexões todo-parte, limites inferiores e superiores podem ser mostrados diretamente, e se um número fixo de associações tiver que ocorrer, uma quantidade única pode ser usada.

No exemplo da Figura 4.10, um plano de vôo particular tem que estar associado a exatamente um avião, e um avião particular pode ter desde 1 até muitos planos de vôo associados a ele.

Então, para cada objeto do modelo deve-se adicionar linhas de associações ligando-os a outros objetos; definindo a quantidade ou intervalo dessas associações, a partir das perspectivas de cada objeto.

Para identificar as associações, o desenvolvedor deve se colocar no lugar do objeto e pensar : "Eu sou um objeto, eu conheço outros objetos, àqueles que se relacionam com o objeto do mundo real do qual eu sou uma abstração (Coad,1993B)."

Numa estrutura Gen-Espec deve-se adicionar associações no ponto mais superior da estrutura possível, ou seja, onde elas permaneçam aplicáveis à cada uma de suas especializações. Se uma associação aplicar-se a um nível inteiro de especializações, então deve-se movê-la para a generalização correspondente.

Ao se estabelecer uma associação entre os objetos em uma estrutura Gen-Espec pode acontecer de uma das extremidades da associação não ter objetos diretamente correspondentes. Nesse caso, a associação deve ser representada como mostra a Figura 4.11 a seguir.

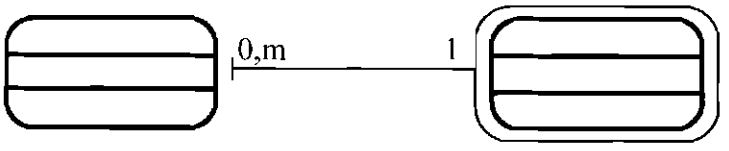


Fig. 4.11 - Variação da Notação de Associação.

Na fase de implementação as associações, até agora representadas apenas graficamente, serão representadas por atributos contendo o identificador (endereço) de cada objeto associado, ou seja, cada objeto terá implicitamente um identificador para cada objeto com o qual ele se associa.

4.2.4 - Classes&Objetos : O que Considerar e o que Recusar

Após a localização das Classes&Objetos potenciais, deve-se fazer uma seleção daquelas que realmente representem o domínio sob consideração. Porém, quais os critérios que devem ser usados para decidir quais Classes&Objetos considerar ?

Abaixo seguem algumas diretrizes para decidir o que considerar e o que recusar na seleção de Classes&Objetos :

- **Lembrança Necessária.** O sistema precisa lembrar de algo sobre os objetos na classe? Um objeto da classe pode ser descrito ? Quais são alguns de seus atributos potenciais ? Se o sistema não precisar saber ou fazer algo sobre os objetos da classe, e se os objetos não puderem ser descritos em termos de seus atributos, então essa classe provavelmente não é necessária para o sistema sob consideração.

- **Processamento Necessário.** Os objetos da classe precisam ter algum tipo de comportamento (processamento) ?

- **Atributos Múltiplos (Normalmente).** Se os objetos da classe tem apenas um atributo algo pode estar errado. Pode ser melhor incluir este objeto como um atributo de alguma outra classe, do que incluí-lo como um objeto individual.

- **Mais de um objeto em uma classe (Normalmente).** Deve-se recusar classes com apenas um objeto. Contudo, se uma classe com apenas um único objeto realmente refletir o domínio do problema, deve-se conservá-la no modelo.

- **Atributos sempre aplicáveis.** Consegue-se identificar um conjunto de atributos que se aplica a cada objeto na classe ? Sempre que um sistema souber que existe um objeto, deve haver um valor para cada atributo. Se existirem atributos que puderem ser aplicados somente a certos tipos de objetos na classe, isso é uma indicação de que uma estrutura de generalização-especialização pode ser desenvolvida.

- **Serviços sempre aplicáveis.** Consegue-se identificar serviços sempre aplicáveis, isto é, um comportamento (processamento) que se aplica a cada objeto em uma classe? Se os serviços variarem para diferentes objetos na classe, isto indica que uma estrutura de generalização-especialização pode ser adicionada.

- **Objetos de Dados e Objetos de Controle.** Deve-se rever os objetos que apenas armazenem valores e os objetos que controlem outros objetos, fazendo com que os dados e as ações sobre esses dados fiquem juntos no mesmo objeto.

4.2.5 - Organizando os Objetos do Domínio do Problema

Foram selecionados um número de objetos do domínio do problema, foram compostas classes desses objetos, e foram estabelecidas responsabilidades para eles.

Agora deve-se agrupar o que se obteve sobre o componente domínio do problema num mesmo Diagrama de Classe&Objeto (para mais detalhes sobre o Diagrama de

Classe&Objeto ver o item **2.10.2.1 - Técnicas e Metodologias Orientadas para Objetos**, no Capítulo 2 desta dissertação).

Para uma melhor organização do Diagrama de Classe&Objeto do sistema (Diagrama de Classe&Objeto de todos os componentes do sistema), deve-se envolver o diagrama de Classe&Objeto do Componente Domínio do Problema com uma quadrado nomeado Componente Domínio do Problema (CDP).

Um exemplo de diagrama de classe&objeto para o componente domínio do problema é dado na Figura 4.12 a seguir.

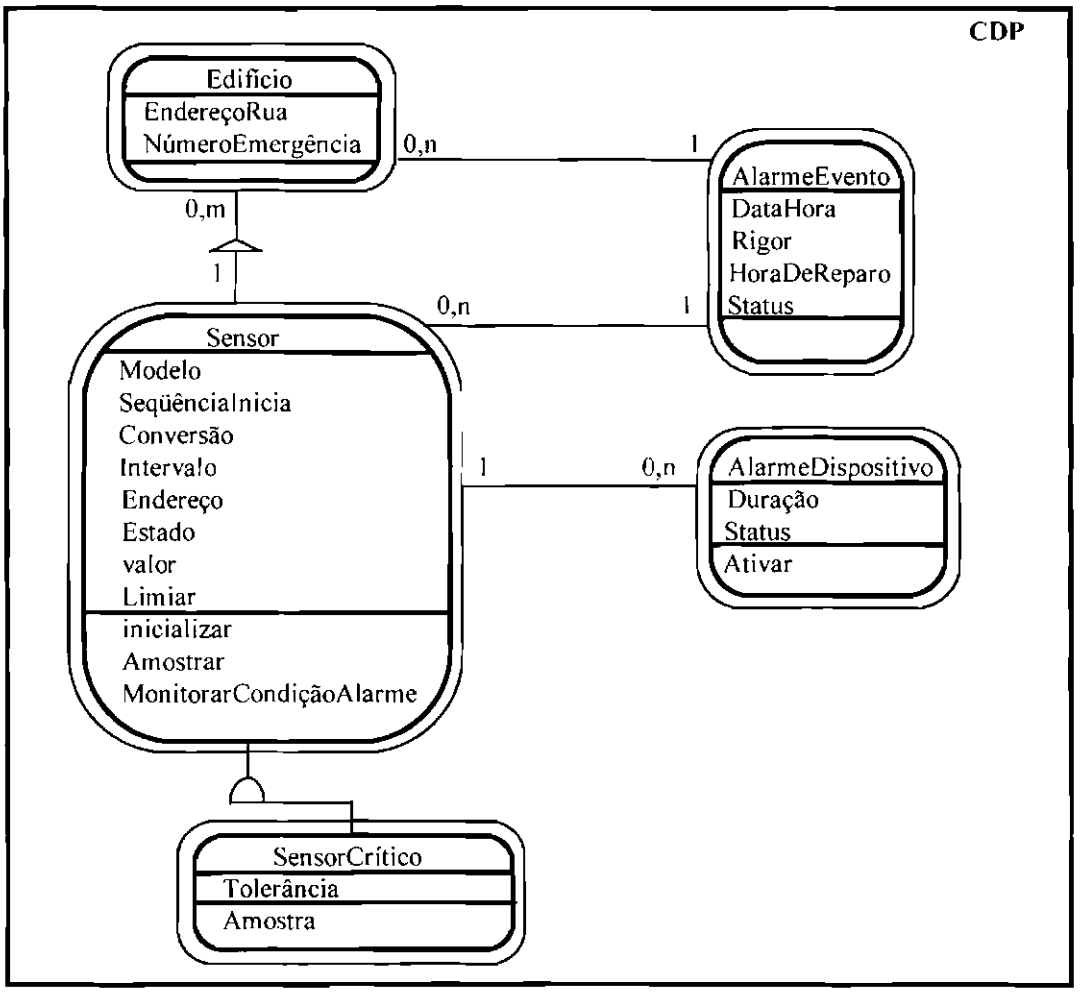


Fig. 4.12 - Diagrama de Classe&Objeto do Modelo Conceitual para o Componente Domínio do Problema.

4.2.6 - Localizando Objetos do Componente Interação Humana

O Componente Interação Humana deve captar como uma pessoa comandará o sistema, e como o sistema apresentará as informações ao usuário. Ele contém, portanto, objetos que provêem a interface entre os objetos do domínio do problema e os usuários. Num modelo de objetos, tais objetos na maioria das vezes correspondem a janelas e relatórios, mas podem variar de acordo com a aplicação.

Para localizar objetos de interação humana, o desenvolvedor deve se colocar no lugar de um objeto e pensar : "Eu, objeto da interação humana, obtenho a informação que eu preciso obter, mostro a informação que eu preciso mostrar, e mudo a informação que eu preciso mudar (Coad,1993B)."

Ao localizar objetos de interação humana deve-se verificar também o que muda se for usada uma interação humana diferente (por exemplo, usar comandos de voz ao invés do teclado do computador). Se os objetos do domínio do problema mudarem com a alteração, deve-se revê-los e separar a apresentação do sistema do que realmente acontece no domínio do problema.

Para uma melhor organização do Diagrama de Classe&Objeto do sistema, deve-se envolver o diagrama de Classe&Objeto do Componente Interação Humana com um quadrado nomeado Componente Interação Humana (CIH), assim como foi feito no Componente Domínio do Problema.

4.2.6.1 - Localizando Janelas

Quais janelas devem ser consideradas no modelo de objetos ? A seguir vão algumas estratégias para localizar janelas :

1) Deve-se adicionar janelas para cada objeto do domínio do problema que tem interface com a interação humana. Por exemplo, para a classe "Contador" do domínio do problema ilustrada na Figura 4.13 a seguir; foi adicionada a classe "Janela de valor" no Componente Interação Humana. Essa classe é uma janela que mostra "o valor" do contador.

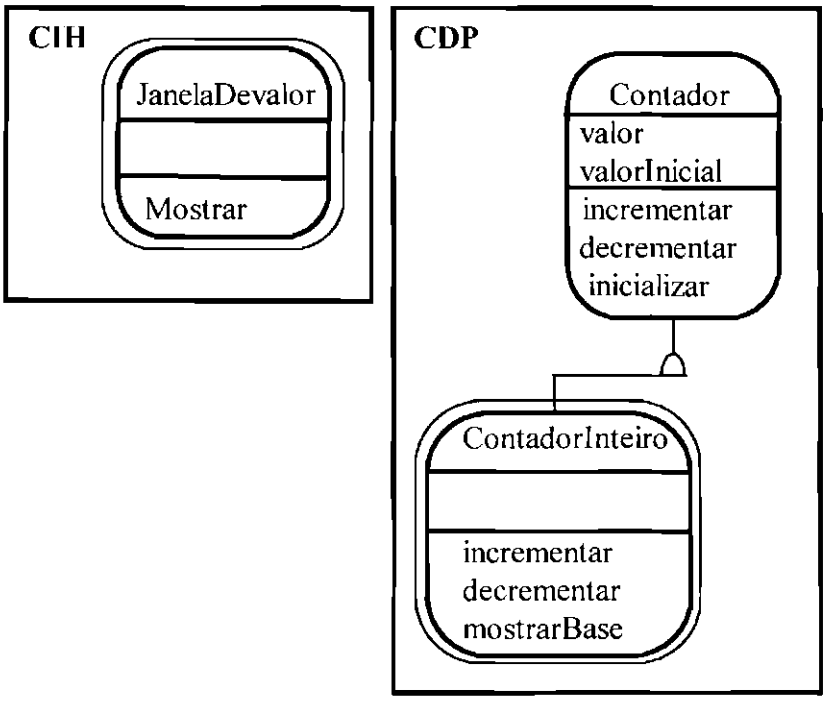


Fig. 4.13 - Adicionando uma Classe de Interação Humana.
FONTE: Adaptada de Coad (1993B, p. 38).

Nesse exemplo, "ContadorInteiro" redefine os serviços "incrementar" e "decrementar" definidos pela classe "Contador". Pode-se perguntar : Porque então não defini-los apenas na classe "ContadorInteiro", já que "Contador" não tem objetos diretos? Esses serviços devem ser definidos na classe "Contador" e redefinidos na classe "ContadorInteiro", para forçar com que os serviços redefinidos nas classes de especialização tenham a mesma interface nas diferentes classes de especialização (por exemplo "ContadorInteiro", "ContadorReal",

"ContadorComplexo"), ou seja, as mesmas quantidades e tipos de *argumentos*, e o mesmo tipo do valor resultante.

2) Deve-se adicionar janelas de inicialização. Elas devem conter serviços de administração do sistema que permitam, se for o caso, adicionar novos usuários e privilégios.

3) Deve-se adicionar janelas de análise de resultados.

4) Deve-se adicionar janelas que contenham objetos relacionados no tempo. Por exemplo, uma venda e seu pagamento.

4.2.4.2 - Localizando Relatórios

Que tipos de relatórios pode-se precisar num sistema ?

Um sistema pode precisar emitir relatórios legais, isto é, relatórios que devem ser emitidos de acordo com a lei, como recibos, notas fiscais, etc...; e relatórios que digam respeito ao sistema em si.

Uma boa maneira de localizar os relatórios que um sistema necessita é agrupar as principais saídas chaves que o sistema deve prover para cumprir as necessidades legais; ou as necessidades que digam respeito ao sistema em si.

Além disso, para cada relatório deve-se verificar se ele mostra apenas valores do domínio do problema e resultados de cálculos, ou se ele engloba algo que o sistema necessita saber e fazer ao longo do tempo. Somente deve-se adicionar um relatório quando o sistema for responsável por produzir e lembrar o conteúdo mostrado nele.

4.2.7 - Estabelecendo Responsabilidades para os Objetos de Interação Humana

Da mesma maneira que no Componente Domínio do Problema, deve-se estabelecer responsabilidades para os objetos de interação humana estabelecendo "o que" eles sabem (atributos), "o que" eles fazem (serviços), e "quem" eles conhecem (conexões todo-parte e associações).

Para janelas e relatórios normalmente inclui-se atributos básicos como campo de procura e campo de entrada de dados, além de outros relacionados com a aplicação.

Os serviços para as janelas são as ações do menu específicas de cada janela. Para relatórios, os serviços são as ações de saída de dados específicas de cada relatório. Em adição, cada objeto de interação humana sabe como se mostrar, e cada classe da interação humana sabe como criar, inicializar e deletar um objeto novo na classe.

Num modelo de objetos um objeto de interação humana conhece o que ele suporta, ou seja, seus componentes e seu *layout* de apresentação.

Considere os componentes de uma janela. Uma janela pode conter muitos objetos, incluindo menus, botões, campos de edição, caixas de listagem etc... Esses objetos trabalham juntos para criar o efeito de um objeto maior, a janela.

Assim sendo, deve-se incluir conexões para os componentes da janela ou relatório; e para os objetos que eles conhecem diretamente, isto é, para os objetos de que eles necessitam para realizar seu trabalho. As conexões que existirem entre objetos de um componente do modelo de objetos, com objetos de outro componente do modelo de objetos, podem ser, se desejado, expressas na forma de atributos para facilitar o entendimento do modelo.

Pode-se construir um modelo complexo para cada janela ou relatório, porém na maior parte das vezes isso não é necessário, porque a maioria das linguagens orientadas para objeto já oferecem mecanismos para que se crie as janelas e relatórios com facilidade.

Isto é, elas fornecem uma biblioteca de classes (Biblioteca "Graphical User Interface" (GUI)), que contém classes como campo de edição, caixa de listagem, menu, botão, que podem ser usadas para compor uma janela ou relatório.

Assim, só é necessário realmente projetar uma classe de interface humana quando a aplicação necessitar de alguma manipulação gráfica especial.

Portanto, ao se modelar as classes de interação humana deve-se procurar na biblioteca de classes da linguagem que será usada para a implementação do sistema, por aquelas classes que melhor se enquadram às classes de interação humana do sistema em questão.

Deve-se procurar primeiramente por uma classe que tenha o mesmo propósito ou objetivo daquela do sistema. Pode-se criar um novo objeto nessa classe e usá-la diretamente.

Se não for possível encontrar um casamento exato deve-se procurar por uma classe de generalização. Pode-se adicionar uma classe de especialização ao modelo, herdando as capacidades da classe de generalização, e adicionando algumas capacidades referentes à aplicação.

Se não for possível encontrar um casamento exato e nem uma generalização adequada, deve-se procurar por uma classe que faça parte do que se precisa, mesmo que a generalização-especialização não se aplique. Pode-se herdar as capacidades dessa classe e adicionar o que se precisa.

Finalmente, se nenhuma das alternativas funcionar, deve-se procurar uma classe que faça mais do que é necessário. Pode-se herdar suas capacidades, ignorar o que não é necessário, e adicionar o que for necessário.

Para o exemplo da Figura 4.13, foram adicionadas as conexões mostradas na Figura 4.14 a seguir. A classe "JanelaDevalor" inicialmente identificada, foi melhor arranjada para suportar o reuso, pois a maioria das bibliotecas GUI oferecem uma classe "View Container" que é responsável por organizar na tela o *layout* dos componentes de uma determinada janela.

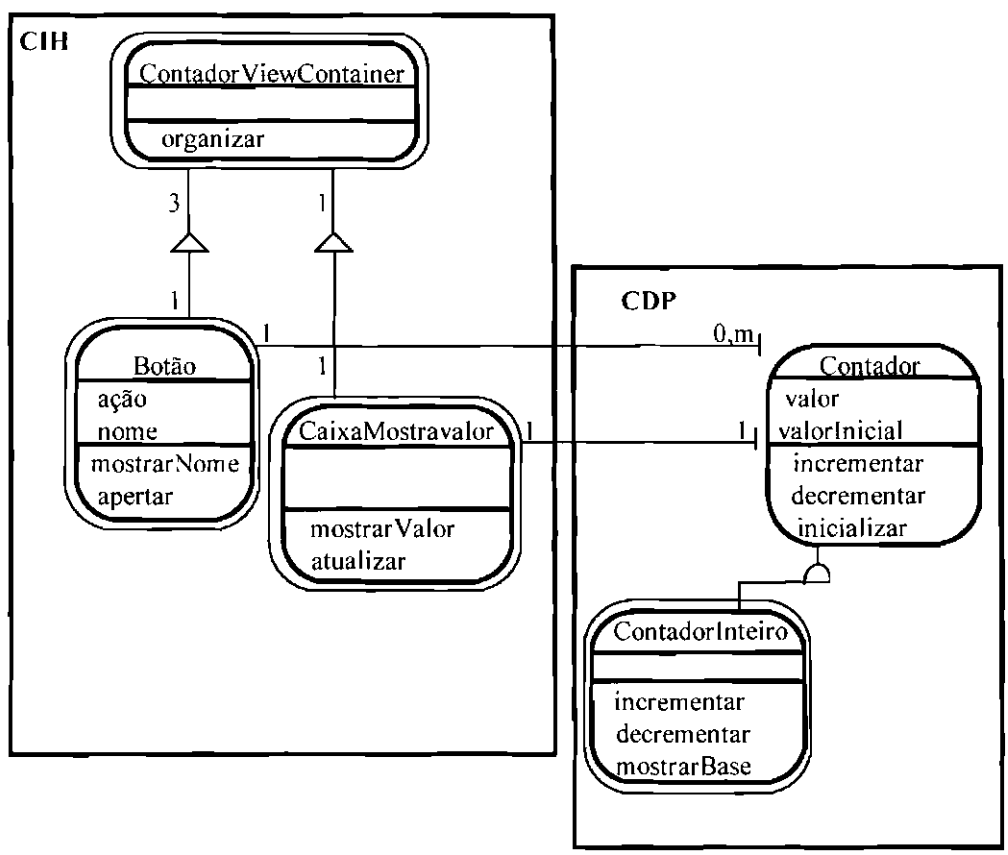


Fig. 4.14 - Adicionando Conexões para os Objetos da Interação Humana.

A Figura 4.14 nos diz que uma janela contêm 3 botões (incrementar, decrementar e inicializar), e uma caixa de apresentação do valor do contador. Ela mostra o nome de cada botão e o valor do contador.

Localizadas as classes de interação humana deve-se revê-las usando os critérios explanados no item **4.2.4 - Classes&Objetos : O que Considerar e o que Recusar**, neste mesmo Capítulo. Feito isso deve-se agrupar o que se obteve sobre a interação humana num mesmo Diagrama de Classe&Objeto.

4.2.8 - Localizando Objetos Persistentes

Objetos persistentes são aqueles objetos que devem ser armazenados ao longo das evocações de programa, ou seja, aqueles objetos que o sistema deve se lembrar ao longo do tempo.

Pode-se usar diferentes mecanismos de armazenamento na implementação dos objetos que devem persistir. Esses mecanismos podem ser sistemas gerenciadores de bancos de dados relacionais, sistemas gerenciadores de bancos de dados relacionais estendidos, ou sistemas gerenciadores de bancos de dados orientados para objetos.

O que são sistemas gerenciadores de bancos de dados relacionais está explanado no item **3.2.6.3 - Implementação dos Dados**, no Capítulo 3 desta dissertação. Portanto aqueles que estejam interessados em mais detalhes e exemplos podem ler esse item.

Os bancos de dados relacionais estendidos acrescentam *tipos abstratos de dados* e herança aos bancos de dados relacionais, além de alguns serviços de uso geral para criar e manipular classes e objetos.

Exemplos de bancos de dados relacionais estendidos podem ser : Oracle 8 da Oracle Corporation, POSTGRESS da Universidade de Berkeley, EXODUS da Universidade de Wisconsin, etc...

Os sistemas gerenciadores de bancos de dados orientados para objetos são uma tecnologia emergente de implementação, e portanto, as plataformas ainda são muito limitadas e os preços conseqüentemente altos.

Os sistemas gerenciadores de bancos de dados orientados para objetos estendem uma linguagem de programação baseada em objetos com a sintaxe e a capacidade de gerenciar a persistência de um objeto ao longo do tempo dentro de um banco de dados. Ele fornece ao desenvolvedor de sistemas, portanto, uma visão contínua de todos os objetos, e torna desnecessária a transferência entre estruturas de dados de armazenamento e estruturas de dados de programa.

Exemplos de bancos de dados orientados para objetos podem ser : O2 da O2 Technology, Poet da Poet Software GmbH, ObjectStore da ODI (Object Design, Inc.), Jasmine da CA (Computer Associates), etc...

Para mais informações sobre bancos de dados relacionais estendidos e bancos de dados orientados para objetos consultar Norman (1998), Deux (1991), Stonebraker (1990), Atkinson (1998).

Se o sistema sendo desenvolvido for usar mecanismos de armazenamento do tipo sistemas gerenciadores de bancos de dados orientados para objetos, o próprio sistema de gerenciamento de banco de dados se encarregará de mapear os dados do objeto assinalado como persistente para o banco de dados, e o próprio sistema de gerenciamento de banco de dados fornecerá o comportamento "armazene a si mesmo", "recupere a si mesmo", para cada objeto a ser armazenado através do tempo. Portanto a única medida que precisa ser tomada é localizar os objetos do domínio do problema que devem persistir, e informar o sistema gerenciador de banco de dados dessa persistência; não sendo, nesse caso, necessário desenvolver o Componente Gerenciamento de Dados.

Se o sistema sendo desenvolvido for usar mecanismos de armazenamento do tipo sistemas gerenciadores de bancos de dados relacionais, e sistemas gerenciadores de bancos de dados relacionais estendidos, essa persistência não acontece diretamente; e é necessário se fazer um projeto dos dados (atributos e conexões) dos objetos persistentes, e um projeto dos serviços "armazenar", "recuperar" que irão manipular esses dados no banco de dados. Portanto, aqueles sistemas que forem usar mecanismos de armazenamento desse tipo, devem desenvolver o Componente Gerenciamento de Dados.

Se não for necessário para o seu sistema desenvolver o Componente Gerenciamento de Dados, seja porque se usará um sistema gerenciador de bancos de dados orientados para objetos para o armazenamento dos objetos persistentes, ou seja porque o sistema sendo desenvolvido não precisa armazenar nem recuperar objetos ao longo do tempo, vá para o item **4.3 - Adicionando Interações entre os Objetos**, neste mesmo Capítulo.

4.2.9 - Localizando Objetos do Componente Gerenciamento de Dados

O Componente Gerenciamento de Dados fornece a infra-estrutura para o armazenamento e recuperação de objetos, isolando o impacto de uma mudança no esquema de gerenciamento de dados.

Para localizar os objetos de gerenciamento de dados (objetos GD) deve-se procurar no componente domínio do problema por aqueles objetos que devem persistir, ou seja, por aqueles objetos que devem ser armazenados entre as evocações de programa.

Para cada objeto persistente encontrado no domínio do problema deve-se adicionar um objeto GD no componente gerenciamento de dados. Dessa maneira os objetos GD encapsulam os mecanismos de procura e armazenamento dos objetos do domínio do problema, isolando as complexidades do gerenciamento dos dados do restante da aplicação.

Por exemplo, para um objeto persistente "Sensor" do domínio do problema, adiciona-se um objeto GD "SensorGD" no Componente Gerenciamento de Dados. O objeto "SensorGD" encapsulará os mecanismos de procura e armazenamento do objeto do domínio do problema "Sensor".

Então, cada objeto GD serve uma classe específica de objetos do domínio do problema, provendo :

- armazenamento persistente para uma única classe;
- isolamento dos mecanismos de armazenamento de dados;
- um lugar para colocar índices, mecanismos de busca, e afins;
- ativação da recuperação e armazenamento para objetos relacionados com aquele sendo recuperado ou armazenado e
- construção das conexões para os outros objetos.

Então, para cada objeto GD pode-se pensar : "Eu, objeto GD, conheço meu objeto correspondente do domínio do problema, sei como guardar seus valores e suas conexões para outros objetos. Eu obtenho a informação que preciso salvar, formato-a e salvo-a. Eu sei também como recuperar esse objeto (Coad,1993B)."

Além das classes GD adicionadas no Componente Gerenciamento de Dados, pode-se adicionar no Componente Domínio do Problema uma classe CDP-CGD que define os serviços "Armazenar um Objeto", "Recuperar um Objeto" e afins, para os objetos persistentes do domínio do problema. As classes persistentes do domínio do problema herdam esses serviços da classe CDP-CGD, que por sua vez, não tem objetos diretos; como mostrado na Figura 4.15.

Quando um objeto persistente do domínio do problema ativa seu serviço "Armazenar" ou seu serviço "Recuperar", sua classe GD correspondente é avisada de que um objeto do domínio do problema precisa ser armazenado ou recuperado. Isso faz com que cada

objeto do domínio do problema saiba "armazenar a si mesmo" e "recuperar a si mesmo", isolando porém, o impacto do esquema de armazenamento adotado.

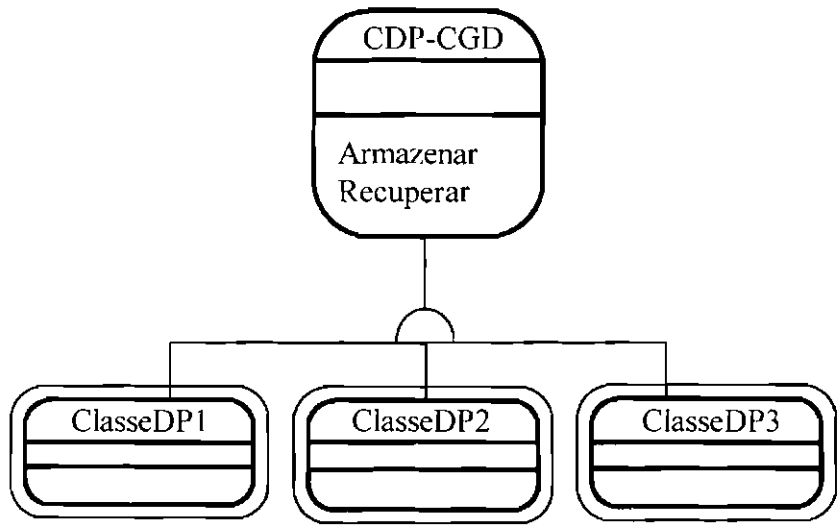


Fig. 4.15 - Classe CDP-CGD.

4.2.10 - Estabelecendo Responsabilidades para os Objetos de Gerenciamento de Dados

Da mesma maneira que nos outros componentes, deve-se estabelecer responsabilidades para os objetos de gerenciamento de dados estabelecendo "o que" eles sabem (atributos), "o que" eles fazem (serviços), e "quem" eles conhecem (conexões todo-parte e associações).

Cada classe GD tem apenas um objeto na classe. Esse objeto conhece seus objetos correspondentes do domínio do problema (todos os objetos em uma classe do domínio do problema), e é responsável por realizar o gerenciamento dos dados desses objetos.

Então, a Figura 4.15 pode ser ainda melhor detalhada adicionando-se as conexões entre o objetos GD e seus objetos do domínio do problema correspondentes, como mostra a Figura 4.16 a seguir. Para uma melhor organização do Diagrama de Classe&Objeto do sistema deve-se envolver o diagrama de classe&objeto do Componente Gerenciamento

de Dados com um quadrado nomeado Componente Gerenciamento de Dados (CGD), assim como foi feito nos outros componentes.

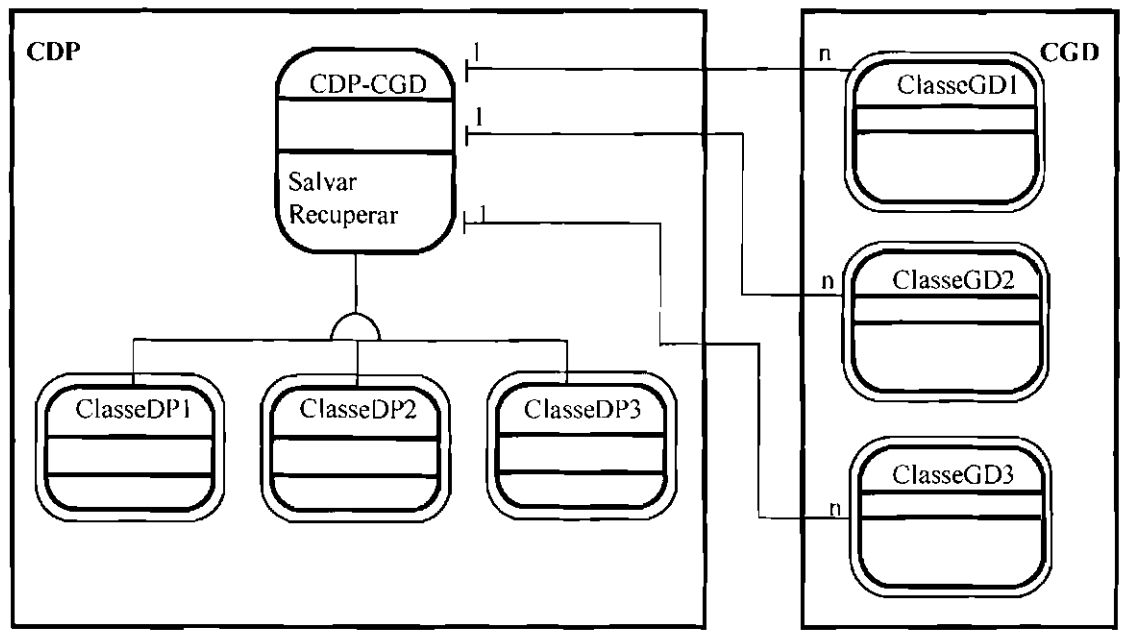


Fig. 4.16 - Conexões entre os Objetos Persistentes do Domínio do Problema e seu Objeto GD Correspondente.

Portanto, para um objeto GD deve-se incluir uma associação para os objetos da sua classe correspondente do domínio do problema. A Classe CDP-CGD deve conter uma tabela de associação que é capaz de associar cada uma de suas especializações ao seu objeto GD correspondente.

Cada objeto GD deve ser responsável basicamente por armazenar e recuperar (recuperar um único objeto ou recuperar um conjunto de objetos, usando um ou mais critérios de busca) de um arquivo persistente, os objetos do domínio do problema que ele conhece.

Portanto, deve-se considerar basicamente para cada objeto GD, serviços do tipo recuperar e armazenar. Esses serviços não devem ser mostrados explicitamente no

diagrama de classe&objeto do modelo conceitual, ou seja, eles devem ser tratados nesse nível de modelagem como serviços implícitos a todos os objetos GD.

4.3 - Adicionando as Interações entre os Objetos

A documentação que foi obtida até esse ponto nos fornece uma visão estática do sistema, o que não é suficiente para entendermos o comportamento dinâmico dos objetos.

Esse comportamento dinâmico acontece através das interações que os objetos têm entre si. Essas interações são necessárias para que os objetos possam realizar seus serviços, pois eles podem necessitar de informações e de serviços que estejam contidos em outros objetos para conseguir cumprir suas responsabilidades.

A comunicação entre os objetos é feita através de mensagens. Uma mensagem entre objetos representa um mapeamento de um objeto para um outro objeto (ou ocasionalmente para uma classe na criação de um novo objeto), no qual um objeto "emissor" envia uma mensagem para um objeto "receptor", para a realização de algum processamento.

As mensagens só existem em função dos serviços, sendo que cada mensagem representa valores enviados no contexto de uma necessidade particular de serviço, e uma resposta recebida como resultado.

Um objeto emissor precisa saber para quem enviar mensagens, e isso está explicitado no Diagrama de Classe&Objeto através das conexões (conexões todo-parte e associações). Então, cada par de objetos conectados normalmente é composto de pelo menos um objeto emissor e de pelo menos um objeto receptor.

Uma boa maneira de capturar a dinâmica dos objetos em um sistema é exercitar as interações que eles têm uns com os outros através de "Cenários". Um cenário é uma descrição de objetos e de uma seqüência de eventos relacionados a eles, que acontecem para que se cumpra uma determinada necessidade. Cada cenário, portanto, especifica as interações entre objetos ordenadas no tempo que são requeridas para responder aos eventos relacionados a uma determinada habilidade do sistema sob consideração (Dai,1997).

Um cenário pode ser comparado a um "ato" de uma peça de teatro, onde a peça é o sistema, e o elenco da peça são os objetos do sistema. Assim como cada ato tem certos atores do elenco que participam dele, cada cenário tem certos objetos do sistema que interagem dentro desse cenário para realizar a habilidade que ele deve prover para o sistema.

O que caracteriza a mudança de um ato para outro ato em uma peça de teatro é basicamente a mudança do assunto sendo tratado, fazendo com que alguns dos atores do elenco da peça sejam afastados, e outros entrem em cena para participar do próximo ato. Com os cenários acontece o mesmo, ou seja, quando o sistema passa de um cenário para outro cenário, outros objetos do sistema entram em cena para cumprir a nova habilidade.

Cada ato em uma peça de teatro, por sua vez, é constituído de um conjunto de "cenas". Com os cenários acontece o mesmo, e cada seqüência de interações geradas por um evento dentro de um cenário é chamada "Cena".

Cada cenário portanto, pode ser constituído de várias cenas. Por exemplo, para o exemplo da Figura 4.14, pode-se definir o cenário "Um botão é Apertado". Esse cenário é constituído das seguintes cenas : "Incrementar Contador", "Decrementar Contador" e "Inicializar Contador".

Para identificar os cenários que devem ser abordados em um sistema, deve-se portanto, procurar na definição do sistema pelas habilidades que ele deve prover. Deve-se exercitar a dinâmica do sistema com cenários, habilidade por habilidade, montando cenários que demonstrem que as habilidades estão de acordo com os requisitos do sistema. Para tal, cada cenário deve especificar as interações entre objetos que são requeridas para responder aos eventos relacionados a uma determinada habilidade do sistema sob consideração.

A maioria dos sistemas de pequeno porte é constituída de apenas um cenário, porém esse número pode chegar a 3 ou 4 dependendo da aplicação. Um exemplo de sistema de pequeno porte com mais de um cenário pode ser um Sistema para Controle de uma Locadora de Fitas de Vídeo. Esse sistema têm dois cenários diferentes : "Cadastrar Clientes" e "Cadastro e Locação de Fitas".

Cada cenário deve ser desenvolvido e descrito com o que chamamos de "Roteiro de Cenário". Um Roteiro de Cenário para o cenário "Um Botão é Apertado" do exemplo da Figura 4.14 é dado a seguir.

Roteiro para o Cenário "Um Botão é Apertado (1)" :

Cena : "Incrementar Contador (A)"

Eu sou um **ContadorViewContainer**.
Eu conheço meus componentes (Botão + CaixaMostravalor).
Eu organizo meus componentes.
Eu aviso meu Botão que ele foi apertado **(Mensagem 1A1)**.
Eu aviso minha CaixaMostravalor que seu valor mudou **(Mensagem 1A2)**.

Eu sou um **Botão**.
Meu nome é "incrementar".

Em algum momento no tempo eu sou apertado.

Eu conheço meu contador inteiro. Eu envio para ele uma mensagem dizendo para ele se incrementar **(Mensagem 1A3)**.

Eu sou um **ContadorInteiro**.

Eu recebo uma mensagem dizendo para eu me incrementar.

Eu me incremento.

Eu sou uma **CaixaMostravalor**.

Eu recebo uma mensagem avisando que uma mudança que é de meu interesse aconteceu.

Eu conheço meu ContadorInteiro. Eu envio uma mensagem para ele para pegar seu valor **(Mensagem 1A4)**.

Eu mostro o novo valor.

Cena : "Decrementar Contador (B)"

Eu sou um **ContadorViewContainer**.

Eu conheço meus componentes (Botão + CaixaMostravalor).

Eu organizo meus componentes.

Eu aviso meu Botão que ele foi apertado (**Mensagem 1B1**).

Eu aviso minha CaixaMostravalor que seu valor mudou **(Mensagem 1B2)**.

Eu sou um **Botão**.

Meu nome é "decrementar".

Em algum momento no tempo eu sou apertado.

Eu conheço meu contador inteiro. Eu envio para ele uma mensagem dizendo para ele se decrementar **(Mensagem 1B3)**.

Eu sou um **ContadorInteiro**.

Eu recebo uma mensagem dizendo para eu me decrementar.

Eu me decremento.

Eu sou uma **CaixaMostravalor**.

Eu recebo uma mensagem avisando que uma mudança que é de meu interesse aconteceu.

Eu conheço meu ContadorInteiro. Eu envio uma mensagem para ele para pegar seu valor **(Mensagem 1B4)**.

Eu mostro o novo valor.

Cena : "Inicializar Contador (C)"

Eu sou um **ContadorViewContainer**.

Eu conheço meus componentes (Botão + CaixaMostravalor).

Eu organizo meus componentes.

Eu aviso meu Botão que ele foi apertado **(Mensagem 1C1)**.

Eu aviso minha CaixaMostravalor que seu valor mudou **(Mensagem 1C2)**.

Eu sou um **Botão**.

Meu nome é "inicializar".

Em algum momento no tempo eu sou apertado.

Eu conheço meu contador inteiro. Eu envio para ele uma mensagem dizendo para ele se inicializar **(Mensagem 1C3)**.

Eu sou um **ContadorInteiro**.

Eu recebo uma mensagem dizendo para eu me inicializar.

Eu me inicializo.

Eu sou uma **CaixaMostravalor**.

Eu recebo uma mensagem avisando que uma mudança que é de meu interesse aconteceu.

Eu conheço meu ContadorInteiro. Eu envio uma mensagem para ele para pegar seu valor **(Mensagem 1C4)**.

Eu mostro o novo valor.

Construído o roteiro para o cenário pode-se representar cada cena no Diagrama de Classe&Objeto, como mostra a Figura 4.17 a seguir. A seta em negrito representa as mensagens enviadas entre os objetos. Dessa maneira consegue-se incorporar o dinamismo do sistema no próprio Diagrama de Classe&Objeto, sem ter que criar uma notação gráfica adicional.

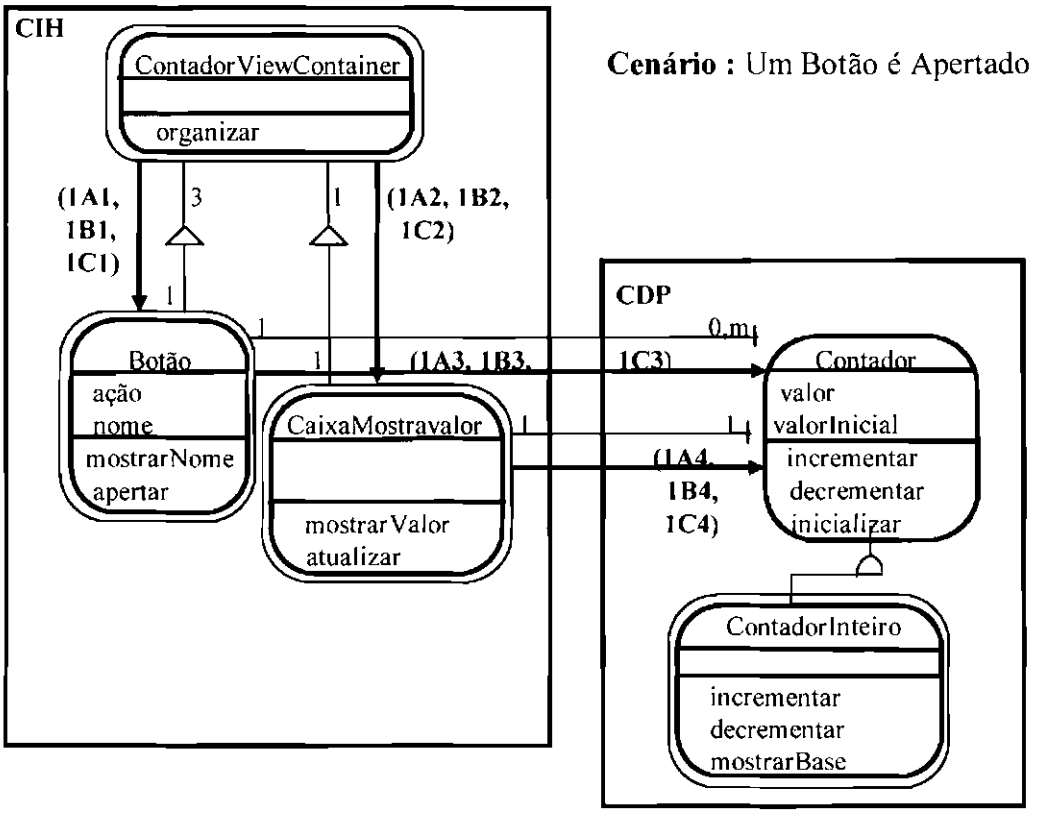


Fig. 4.17 - Adicionando as Interações entre os Objetos ao Diagrama de Classe&Objeto do Modelo Conceitual.

Toda mensagem deve ter um rótulo que deve ser constituído do número representante do cenário, da letra representante da cena do cenário, e do número da mensagem dentro daquela cena. Os números das mensagens devem ser representativos de sua ordem cronológica, ou seja, as mensagens devem estar numeradas de acordo com a ordem em que são geradas dentro de uma cena.

Para uma melhor clareza do diagrama pode-se representar cada cena de um cenário em um diagrama de classe&objeto diferente. Por exemplo, para o cenário "Um Botão é Apertado" geraria-se três diagramas de classe&objeto, um representando a cena "Incrementar Contador", outro representando a cena "Decrementar Contador", e outro representando a cena "Inicializar Contador".

Na fase de implementação as mensagens agora representadas por setas de mensagens serão representadas por chamadas a serviços específicos, onde o serviço emissor chama o serviço receptor para a realização de algum processamento. A comunicação entre eles é feita através da passagem de *parâmetros*.

O excesso de mensagens entre os objetos pode deixar o modelo difícil de se entender e de se implementar. Uma boa maneira de amenizar o tráfego de mensagens é construir objetos que não sejam simplesmente guardadores de dados. Isso significa construir objetos que contenham tanto os dados, quanto as ações que modificam esses dados. Ou seja, um objeto não deve perguntar a um outro objeto por seu(s) valor(es) e depois fazer ele mesmo o trabalho. Ele deve dizer ao outro objeto que faça o trabalho, e retorne para ele um resultado significativo.

Portanto, após acrescentar as interações entre os objetos deve-se rever aqueles objetos que interajam com quase todos os objetos no modelo, tentando ajustar suas responsabilidades (o que eu sei, quem eu conheço, o que eu faço).

4.4 - Adicionando Objetos do Componente Gerenciamento de Cenários

O Componente Gerenciamento de Cenários contém uma Classe Gerenciadora de Cenários (Classe GC) que tem um objeto coordenador que cuida da criação (criar e inicializar) dos objetos dos outros componentes nos cenários do sistema (Cunha,1997).

A classe GC deve conter uma tabela de ativação dos objetos, ou seja, para cada cenário, a classe GC deve saber quais objetos criar, montando um ambiente propício para que um determinado cenário possa ser "encenado".

Pode-se então, comparar a classe GC a um "diretor" de uma peça de teatro. Ou seja, em cada ato (cenário) ela é responsável por colocar em cena (criar e inicializar) aqueles atores que participarão daquele ato.

A Figura 4.18 a seguir, mostra a adição da classe GC "ContadorGC" para o cenário apresentado na Figura 4.17. O serviço "ativarCenário" da classe ContadorGC, cria e inicializa um objeto do tipo "ContadorInteiro" e um objeto do tipo "ContadorViewContainer"; e conecta o contador inteiro criado à sua janela "ContadorViewContainer".

O diagrama de classe&objeto para o Componente Gerenciamento de Cenários deve ser envolvido com um quadrado nomeado Componente Gerenciamento de Cenários (CGC).

4.5 - Projetando o Modelo de Objetos

A segunda etapa do desenvolvimento de um sistema corresponde à construção de seu modelo lógico, que no paradigma da orientação a objetos consiste em se projetar o modelo de objetos.

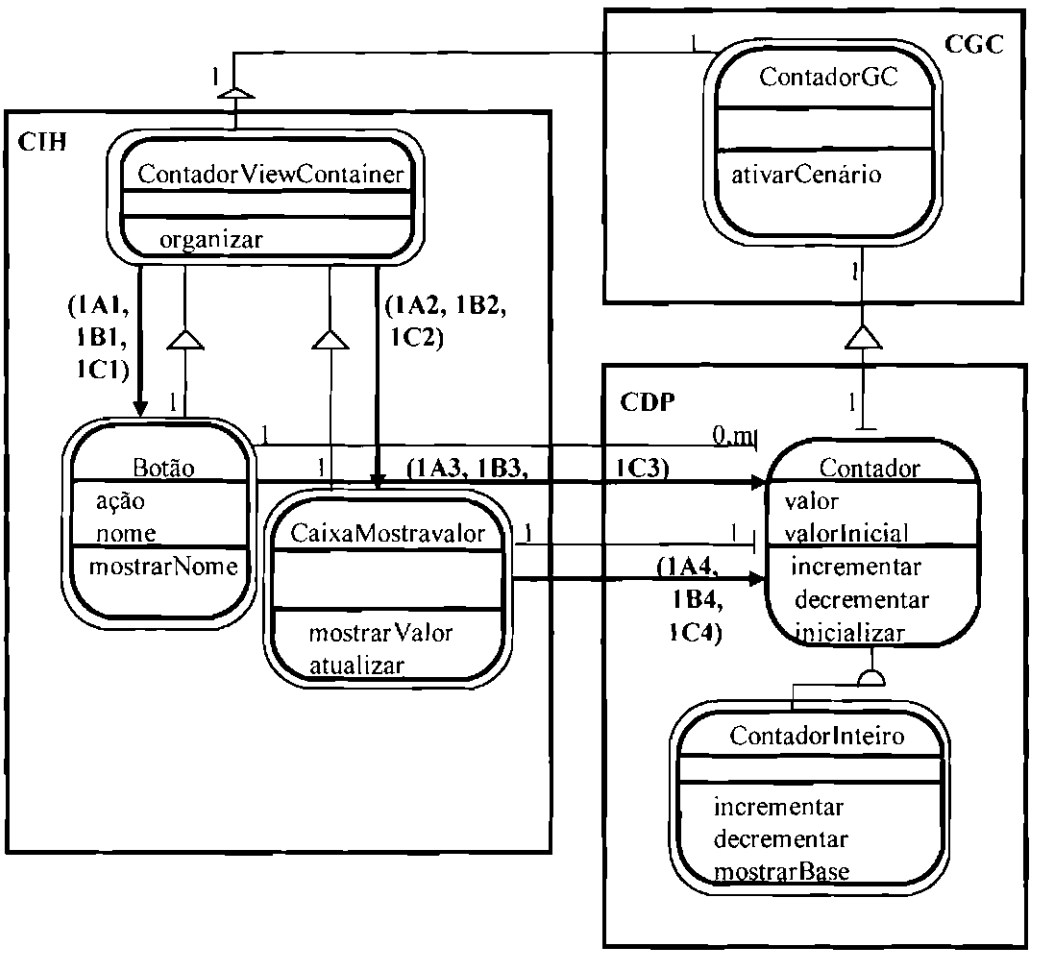


Fig. 4.18 - Adicionando uma Classe Gerenciadora de Cenários.

Projetar o modelo de objetos significa fazer um projeto das classes do sistema detalhando seus atributos, conexões e serviços visando sua implementação, e em se projetar os dados que deverão persistir (quando o Componente Gerenciamento de Dados existir no modelo de objetos).

4.5.1 - Projetando as Classes

Projetar as classes significa adicionar os serviços Criar, Liberar, Acessar e Setar para cada classe, projetar as conexões existentes entre os objetos dessa classe e os outros objetos do modelo, projetar os serviços complexos, e fazer uma especificação para cada classe.

4.5.1.1 - Projetando as Conexões entre os Objetos

Projetar as conexões entre os objetos significa adicionar um atributo de conexão à cada objeto emissor conectado a um objeto receptor, adicionando também serviços que irão conectar/desconectar o objeto emissor ao/do objeto receptor.

Ou seja, além de ser representada graficamente, uma conexão que um objeto emissor tem com um objeto receptor deve também ser representada por um atributo que contém o identificador (endereço) do objeto receptor conectado a ele. Esses atributos que são decorrentes das conexões devem aparecer no diagrama de classe&objeto do modelo lógico com um símbolo "(p)", indicando que eles são atributos de projeto.

Isso significa que antes de adicionar aos objetos conectados os atributos decorrentes das conexões, deve-se examinar cada conexão verificando quem é o objeto emissor e quem é o objeto receptor. Para cada objeto emissor conectado deve-se adicionar um atributo da conexão. O lado da conexão que corresponde ao objeto receptor deve ter sua restrição eliminada do diagrama de classe&objeto do modelo lógico, para indicar que ele não precisa conhecer o objeto emissor conectado a ele.

Para cada objeto emissor conectado deve-se também adicionar serviços que irão ser responsáveis por conectar e desconectar o objeto emissor ao objeto receptor. Esses serviços são "ConectarAo" e "DesconectarDe" respectivamente. Esses serviços devem aparecer no diagrama de classe&objeto do modelo lógico com um símbolo "(p)", indicando que eles são serviços de projeto.

Ao se projetar o serviço "DesconectarDe", deve-se tomar alguns cuidados. Por exemplo, ao se desconectar um objeto "todo" de suas "partes" deve-se eliminá-las, pois não tem sentido elas existirem por si só. O mesmo acontece com certos objetos associados que só existem em função de outros objetos.

Portanto, para a Figura 4.18 foram adicionados os atributos e os serviços decorrentes das conexões como mostra a Figura 4.19 adiante. As restrições das conexões do lado do objeto receptor foram eliminadas do diagrama, indicando que o objeto receptor não precisa conhecer seu objeto emissor.

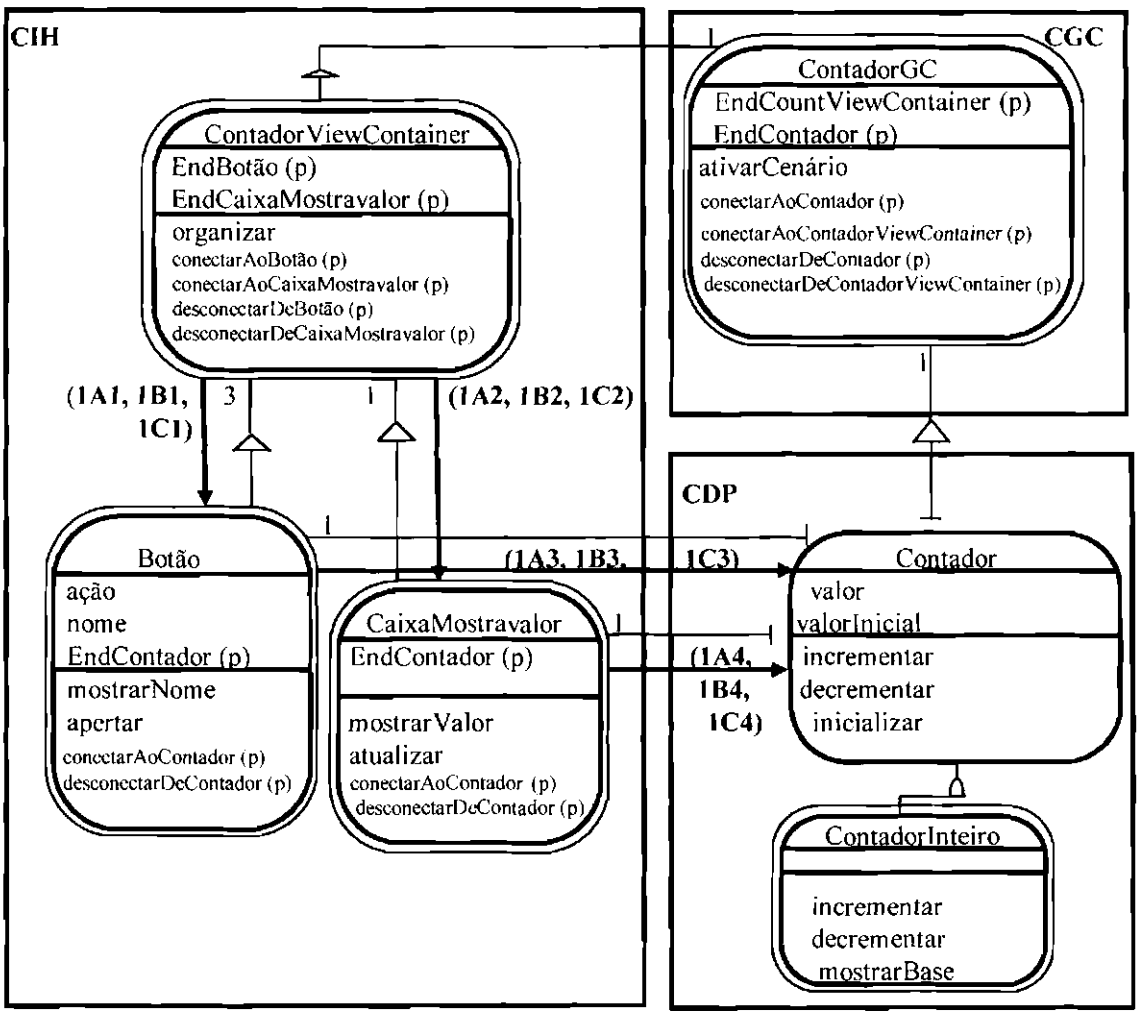


Fig. 4.19 - Acrescentando Atributos e Serviços Decorrentes das Conexões ao Diagrama de Classe&Objeto do Modelo Lógico.

As conexões que têm restrições maiores que "1", geram atributos de conexões multivalorados, isto é, atributos que podem ter mais de um valor.

Um exemplo disso é o atributo "EndBotão". Ele é considerado multivalorado porque como cada objeto "ContadorViewContainer" tem conectado a ele mais de um objeto "Botão", o atributo "EndBotão" deve conter os endereços de todos os objetos "Botão" conectados a "ContadorViewContainer".

Na fase de implementação, os atributos multivalorados dos objetos serão transformados em listas ou filas que armazenam os endereços dos objetos conectados a eles.

4.5.1.2 - Adicionando os Serviços Criar, Liberar, Acessar, Setar, e Serviços de Restrição de Integridade

Para cada classe deve-se adicionar um serviço "Criar" que deve criar e inicializar um novo objeto na classe; e um serviço "Liberar" que deve deletar e desconectar um objeto na classe.

Ao se liberar um objeto, deve-se se assegurar de não ter deixado nenhuma referência ao objeto, ou seja, existe o perigo de se deixar referências pendentes a outros objetos eliminados incorretamente (uma referência pendente é a referência feita a um objeto que não existe mais), bem como de perda de memória para objetos inacessíveis que não foram eliminados (Rumbaugh,1994).

Isso quer dizer que ao se liberar um objeto "todo" deve-se cuidar para que todas as suas "partes" também sejam liberadas. Podem existir também objetos associados a outros objetos que só existam em função destes, e eles também devem ser liberados quando estes últimos forem liberados.

Cada objeto em uma classe tem seus próprios valores para os atributos da classe, e em geral, é um bom costume conservar esses atributos como "privados". Um atributo privado não pode ser acessado por serviços de outras classes, ou seja, eles só podem ser acessados por serviços de sua própria classe.

Assim sendo, se for necessário que objetos de outras classes leiam ou modifiquem esses valores, adiciona-se a essa classe os serviços "Acessar" e "Setar" respectivamente, para cada atributo da classe.

Deve-se também, se necessário, adicionar à cada classe, serviços de restrição de integridade. Os serviços de restrição de integridade podem se referir à integridade da própria entidade, ou seja, pode ser que no domínio da aplicação não possa existir dois objetos com os mesmos valores para seus atributos. Isso significa que antes de criar um novo objeto deve-se verificar se um outro objeto que tenha os mesmos valores para os atributos que os dele, já existe. Pode acontecer também de não poderem ser criados dois objetos com o mesmo valor para um determinado atributo (pode-se chamá-lo de atributo "chave"), e isso também deve ser verificado.

Além disso, os serviços de restrição de integridade podem referir-se à integridade referencial, ou seja, a um objeto "todo" que tem no máximo 4 "partes", por exemplo, não se pode permitir conectar uma quinta "parte". Isso significa que as restrições estabelecidas nas conexões todo-parte e nas associações devem ser observadas nessa fase, adicionando-se serviços necessários para tratá-las.

Os serviços Criar, Liberar, Acessar, Setar, e os Serviços de Restrição de Integridade, não devem aparecer no diagrama de classe&objeto do modelo lógico.

4.5.1.3 - Projetando os Serviços Complexos

Os serviços podem ser classificados em serviços simples e serviços complexos. Os serviços complexos possuem sub-serviços que os ajudam a realizar o trabalho total. Um exemplo disso é o serviço "Calcular Total" da Figura 4.20 a seguir. Ele invoca os serviços "CalcularSubTotal", e "CalcularTaxa" para ajudar no seu trabalho.

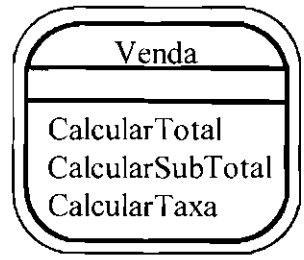


Fig. 4.20 - Serviços Complexos.

Na fase de implementação, cada sub-serviço se transformará em um serviço protegido da classe, ou seja, um serviço que não pode ser acessado diretamente pelos objetos de outras classes.

Cada serviço, na verdade, pode ser visto como um módulo (Khan,1995). Isso significa dizer que cada serviço complexo irá formar por si só um diagrama de estrutura dos módulos (para mais detalhes sobre módulo e diagrama de estrutura dos módulos ver o item **2.10.1.1 - Técnicas Estruturadas**, no Capítulo 2 desta dissertação); onde ele é o módulo superior, e seus sub-serviços seus módulos subordinados.

Então, para cada serviço complexo deve-se construir seu diagrama de estrutura dos módulos como explicado nos itens **3.2 - Construção do Sistema de Pequeno Porte Usando o Paradigma Clássico**, à **3.2.3 - Acoplamento entre Módulos** (inclusive) do Capítulo 3 desta dissertação. Ao ler esses itens deve-se substituir a palavra "módulo" por "serviço".

4.5.1.4 - Fazendo uma Especificação para as Classes

A Especificação para uma classe consiste de uma descrição da classe em uma ou duas linhas; da especificação de seus atributos, da especificação de seus serviços, e da especificação das restrições de criação dos objetos da classe.

As especificações das classes de cada componente do sistema devem ser reunidas num documento de Especificação das Classes&Objetos daquele componente. Por exemplo, as especificações das classes do domínio do problema devem ser reunidas num documento de Especificação das Classes&Objetos do Domínio do Problema; e assim por diante.

Um exemplo de especificação para uma classe é dado na Figura 4.21 a seguir. Ele refere-se às classes "Contador " e "ContadorInteiro" da Figura 4.19.

A especificação da classe deve conter ainda uma especificação para cada atributo, e uma especificação para cada serviço que ela tem. Como fazer isso é mostrado nos itens **4.5.1.4.1 - Especificação para os Atributos**, e **4.5.1.4.2 - Especificação para os Serviços**, a seguir.

4.5.1.4.1 - Especificação para os Atributos

A especificação para os atributos deve incluir o nome do atributo, uma descrição de uma ou duas linhas de cada atributo, seu tipo (inteiro, ponteiro, real, caracter), seu valor *default* (se houver), se ele é um atributo discreto (se ele for, incluir seu domínio de valores), ou se ele é um atributo contínuo (se ele for, incluir detalhes como intervalo, limite, precisão, tolerância); e se ele é um atributo monovalorado ou multivalorado.

Além disso deve-se incluir alguns detalhes adicionais como : sua obrigatoriedade (ou seja, se cada objeto deve ter obrigatoriamente um valor para ele; um exemplo de atributo não obrigatório é telefone, pois nem todos tem um telefone), se ele é um atributo determinante (ou seja, se para cada objeto ele tem um valor diferente), sua unidade de medida (se houver), e se ele é um atributo público ou privado (se a linguagem usada para a implementação suportar tal distinção).

Nome da Classe : Contador
Componente à que Pertence : CDP
Descrição : Classe que representa um contador, que é capaz de se incrementar, se decrementar, e de se inicializar.
Restrições de Criação : Não possui objetos diretos.
Atributos : valor valorInicial
Serviços : incrementar decrementar inicializar criarContador (p) liberarContador (p) acessarValor (p) acessarValorInicial (p) setarValor (p) setarValorInicial (p)

Nome da Classe : ContadorInteiro
Componente à que Pertence : CDP
Descrição : Classe que representa um contador inteiro, que é capaz de se incrementar, se decrementar, se inicializar, e de mostrar seu valor na base hexadecimal. É uma especialização da classe "Contador".
Restrições de Criação : Só pode existir "1" objeto ContadorInteiro.
Atributos : valor (Herdado de "Contador") valorInicial (Herdado de "Contador")
Serviços : incrementar (Redefinido de "Contador") decrementar (Redefinido de "Contador") inicializar (Herdado de "Contador") mostrarBase criarContadorInteiro (p) liberarContadorInteiro (p) acessarValor (p) (Herdado de Contador) acessarValorInicial (p) (Herdado de Contador) setarValor (p) (Herdado de Contador) setarValorInicial (p) (Herdado de Contador)

Fig. 4.21 - Especificação para uma Classe.

Um exemplo de especificação para um atributo é dado na Figura 4.22. Ele refere-se ao atributo "valor" da Classe "Contador". Neste exemplo, o campo "Tipo" do atributo "valor" ficou indefinido porque ele é dependente da classe de especialização, ou seja, se o "valor" se referir a um contador inteiro o seu tipo será "inteiro", se ele se referir a um contador real o seu tipo será "Real", e assim por diante.

Nome do Atributo : valor
Descrição : Atributo que contém o valor do contador.
Tipo : Dependente das classes de especialização.
Valor *Default* : ---------
Atributo Discreto : Sim, seu domínio de valores se restringe aos números inteiros.
Atributo Contínuo : ----------
Atributo Monovalorado : [X] **Atributo Multivalorado :** [ ]
Detalhes Adicionais : É um atributo obrigatório, não determinante, público para leitura, privado para escrita.

Fig. 4.22 - Especificação para um Atributo.

4.5.1.4.2 - Especificação para os Serviços

Essa especificação deve conter o nome do serviço, uma descrição do que ele faz, quais são seus *parâmetros* de entrada, quais são seus *parâmetros* de saída, quais são seus sub-serviços (se houver), o diagrama de estrutura dos módulos para os serviços algoritcamente complexos, uma descrição do serviço usando as estruturas da programação estruturada, mais especificamente usando o português estruturado, e se ele é um serviço público ou privado (se a linguagem suportar tal distinção).

As estruturas da Programação Estruturada e o Português Estruturado já foram explicados no item **3.2.5 - Programação Estruturada**, no Capítulo 3 desta dissertação.

Portanto para mais detalhes deve-se consultar esse item. Ao ler o item deve-se substituir a palavra "módulo" por "serviço"".

Um exemplo de especificação para um serviço é dado na Figura 4.23 a seguir. Ele refere-se ao serviço "incrementar" da classe "ContadorInteiro".

Nome do Serviço : incrementar
Função : Incrementa de 1 o valor do contador.
Parâmetros de Entrada : valor
Parâmetros de Saída : valor
Sub-Serviços Referenciados : ----------
Diagrama de Estrutura dos Módulos : ----------
Descrição em Português Estruturado : **INCREMENTAR** incrementar de 1 o "valor" do contador **FIM-INCREMENTAR**

Fig. 4.23 - Especificação para um Serviço.

4.5.2 - Projetando os Dados que Deverão Persistir

Projetar os dados que deverão persistir significa construir o modelo operacional ou lógico para os dados dos objetos persistentes, que é nada mais nada menos que construir o modelo relacional.

A construção do modelo relacional nos ajudará a organizar os dados dos objetos persistentes para que eles possam ser armazenados, recuperados e processados com segurança e eficiência dentro do esquema de armazenamento escolhido.

O que é o modelo relacional está explanado no item **3.2.6.2 - Construção do Modelo Relacional**, no Capítulo 3 desta dissertação. Ao ler esse item deve-se ignorar as

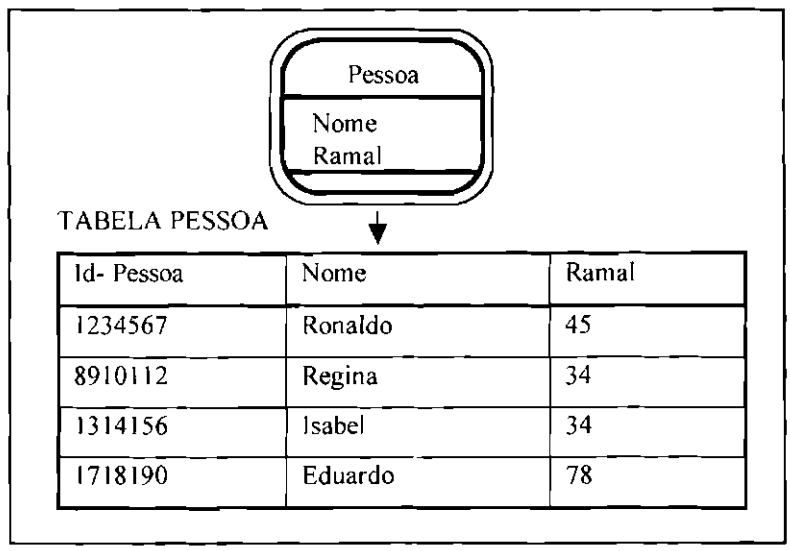
diretrizes de mapeamento do diagrama de entidade-relacionamento para o modelo relacional apresentadas, pois diretrizes apropriadas ao modelo de objetos serão dadas a seguir.

Como mapear os objetos persistentes para as tabelas do modelo relacional ? A seguir são apresentadas algumas diretrizes de como fazer isso.

4.5.2.1 - Mapeamento de Classes

De uma maneira geral cada classe persistente dá origem a uma tabela, onde cada atributo da classe corresponde a uma coluna, e os valores dos atributos para cada objeto na classe correspondem a uma linha.

A Figura 4.24 a seguir mostra um exemplo de conversão de uma classe persistente em uma tabela. Nesse exemplo a classe "Pessoa" possui os atributos "Nome" e "Ramal". Quando ela é convertida para a tabela "Pessoa" acrescenta-se a coluna "Id-Pessoa", pois a classe não possui um atributo ou um conjunto de atributos que possa ser considerado "chave". Essa estratégia é compatível com as linguagens baseadas em objetos, onde os objetos têm identidades independentes das suas propriedades.



Id- Pessoa	Nome	Ramal
1234567	Ronaldo	45
8910112	Regina	34
1314156	Isabel	34
1718190	Eduardo	78

Fig. 4.24 - Mapeamento de Classes em Tabelas.

4.5.2.2 - Mapeamento de Classes de Evento Lembrado e Conexões Relacionadas

As classes de evento lembrado persistentes e as conexões que seus objetos estabelecem com outros objetos persistentes devem ser modeladas de uma só vez. Nesse caso, deve-se criar uma tabela para a classe de evento lembrado, e deve-se inserir nela os atributos da classe de evento lembrado, e as chaves das tabelas das classes cujos objetos são conectados aos objetos da classe de evento lembrado. Um exemplo é dado na Figura 4.25 a seguir.

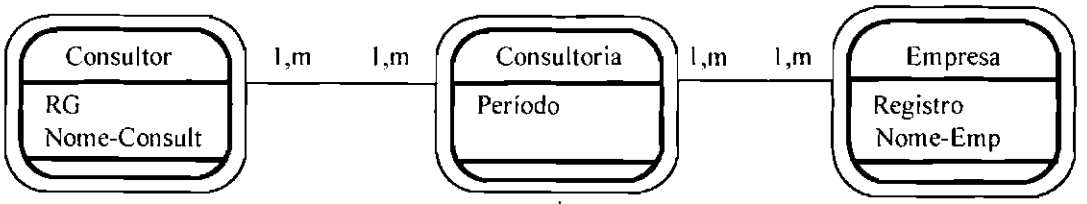


TABELA CONSULTOR

RG	Nome-Consult
3869	Marcelo
9456	Tais

TABELA EMPRESA

Registro	Nome-Emp
123	Alfa
456	Beta
789	Centauro

TABELA CONSULTORIA

RG	Registro	Período
3869	123	12/03/98 à 12/09/98
9456	123	12/03/98/ à 12/09/98
9456	456	17/11//97 à 14/02/98
3869	789	13/06/97 à 20/05/98

Fig. 4.25 - Mapeamento de Classes de Evento Lembrado e Conexões Relacionadas para Tabelas.

4.5.2.3 - Mapeamento de Conexões N : N

As conexões N : N acontecem quando o valor do limite superior em ambas as pontas de uma conexão é maior que "1". Cada conexão N : N entre objetos persistentes deve ser mapeada em uma nova tabela. As chaves das tabelas das classes conectadas transformam-se em atributos dessa nova tabela. Um exemplo do mapeamento de conexões N : N é dado na Figura 4.26 a seguir.

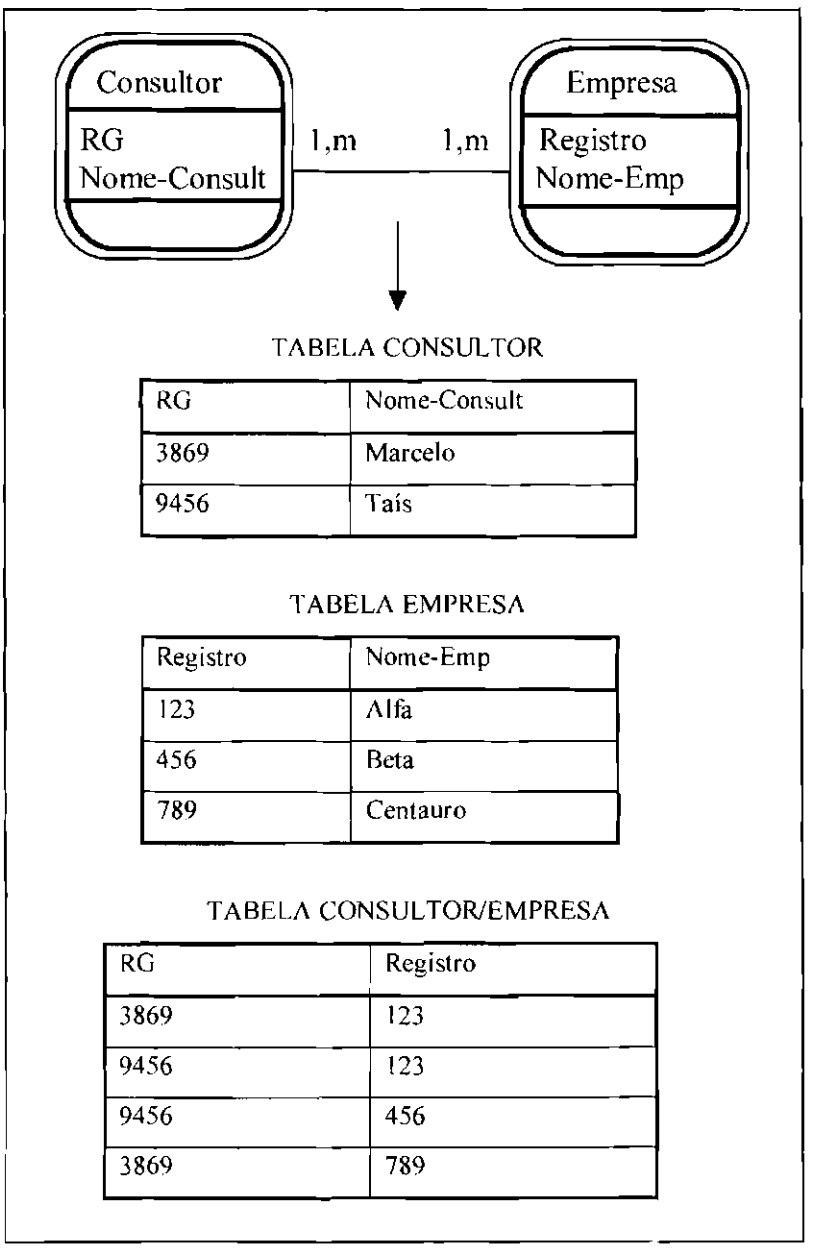


TABELA CONSULTOR

RG	Nome-Consult
3869	Marcelo
9456	Taís

TABELA EMPRESA

Registro	Nome-Emp
123	Alfa
456	Beta
789	Centauro

TABELA CONSULTOR/EMPRESA

RG	Registro
3869	123
9456	123
9456	456
3869	789

Fig. 4.26 - Mapeamento de Conexões N : N para Tabelas.

4.5.2.4 - Mapeamento de Conexões 1 : N

As conexões 1 : N acontecem quando o valor em uma ponta da conexão é "1", e na outra ponta ele tem o limite superior do intervalo maior que "1". Existem duas maneiras de se mapear conexões 1 : N entre objetos persistentes para o modelo relacional :

- **Caso 1.** No primeiro caso transpõe-se a chave da tabela correspondente à classe do lado "N", para a tabela correspondente à classe do lado "1". (Por "transpor" deve-se entender criar uma coluna com a chave da tabela correspondente à uma determinada classe, em outra tabela). Um exemplo é dado na Figura 4.27 a seguir.

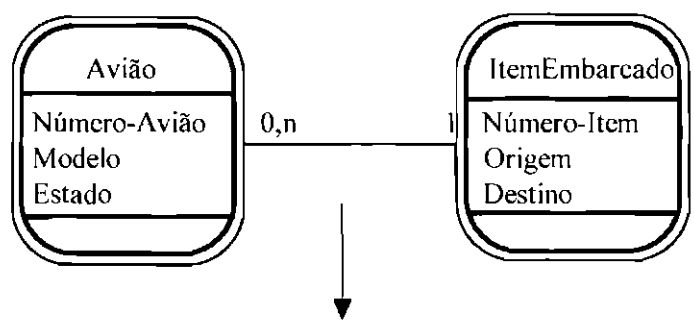


TABELA AVIÃO

Número-Avião	Modelo	Estado
1234	Boing 747	Repouso
5678	Teco-Teco	Voando
9012	AirBus	Manutenção

TABELA ITEM-EMBARCADO/AVIÃO

Número-Item	Origem	Destino	Número-Avião
123	Londres	Praga	5678
456	Nova Iorque	Paris	1234
789	Viena	Edimburgo	5678
012	Istambul	Amsterdam	9012

Fig. 4.27 - Mapeamento de Conexões 1 : N com Associação Implícita.

- **Caso 2.** No segundo caso cria-se uma tabela correspondente à conexão, transpondo para ela as chaves das duas classes. Um exemplo é dado na Figura 4.28 a seguir (foram usadas as classes "Avião" e "ItemEmbarcado" do exemplo anterior).

TABELA AVIÃO

Número-Avião	Modelo	Estado
1234	Boing 747	Repouso
5678	Teco-Teco	Voando
9012	AirBus	Manutenção

TABELA ITEM-EMBARCADO

Número-Item	Origem	Destino
123	Londres	Praga
456	Nova Iorque	Paris
789	Viena	Edimburgo
012	Istambul	Amsterdam

TABELA AVIÃO/ITEM-EMBARCADO

Número-Item	Número-Avião
123	5678
456	1234
789	5678
012	9012

Fig. 4.28 - Mapeamento de Conexões 1 : N com Associação Explícita.

Porém quando deve-se usar o **Caso 1**, e quando deve-se usar o **Caso 2** ? A seguir, são apresentadas algumas vantagens e desvantagens de cada abordagem. Elas devem ser

analisadas de acordo com os requisitos estabelecidos pelo sistema que está sendo desenvolvido (Setzer,1986) :

1) A Figura 4.27 tem uma tabela a menos. Isso significa menor espaço ocupado pelos arquivos resultantes e, muito mais importante, maior eficiência (rapidez) no processamento de operações que envolvam a conexão. As transações são também expressas de maneira mais simples. Portanto, se eficiência for um requisito do sistema que está sendo desenvolvido, deve-se optar por essa abordagem.

2) A Figura 4.28 apresenta uma solução com maior independência de dados, pois a tabela "Item-Embarcado" contém exclusivamente seus dados próprios, ao contrário da Figura 4.27 em que recebe a chave de "Avião".

3) Uma vantagem da Figura 4.28 é que uma mudança estrutural do tipo da conexão, passando de 1 : N para N : N, não requer nenhuma alteração estrutural nas tabelas, ao contrário da Figura 4.27.

4.5.2.5 - Mapeamento de Conexões 1 : 1

As conexões 1 : 1 acontecem quando o valor em ambas as pontas da conexão é "1". Para se fazer o mapeamento das conexões 1 : 1 entre objetos persistentes deve-se criar uma tabela para cada classe, e transpor a chave de uma tabela para a outra. O sentido da transposição depende das formas de acesso para ganho de eficiência.

Os exemplos da Figura 4.29 e da Figura 4.30 a seguir, mostram as duas maneiras de se mapear as conexões 1 : 1 em tabelas. O exemplo da Figura 4.29 é melhor pois como praticamente todos os departamentos têm gerentes, porém poucos funcionários são gerentes, a coluna "Num-Gerente" na tabela "Departamento" produz menos valores vazios, do que a coluna "Número-Depto-Gerencia" na tabela "Funcionário". São elas :

• **Caso 1.** Transpõe-se a chave de "Funcionário" para "Departamento", como mostra a Figura 4.29 a seguir.

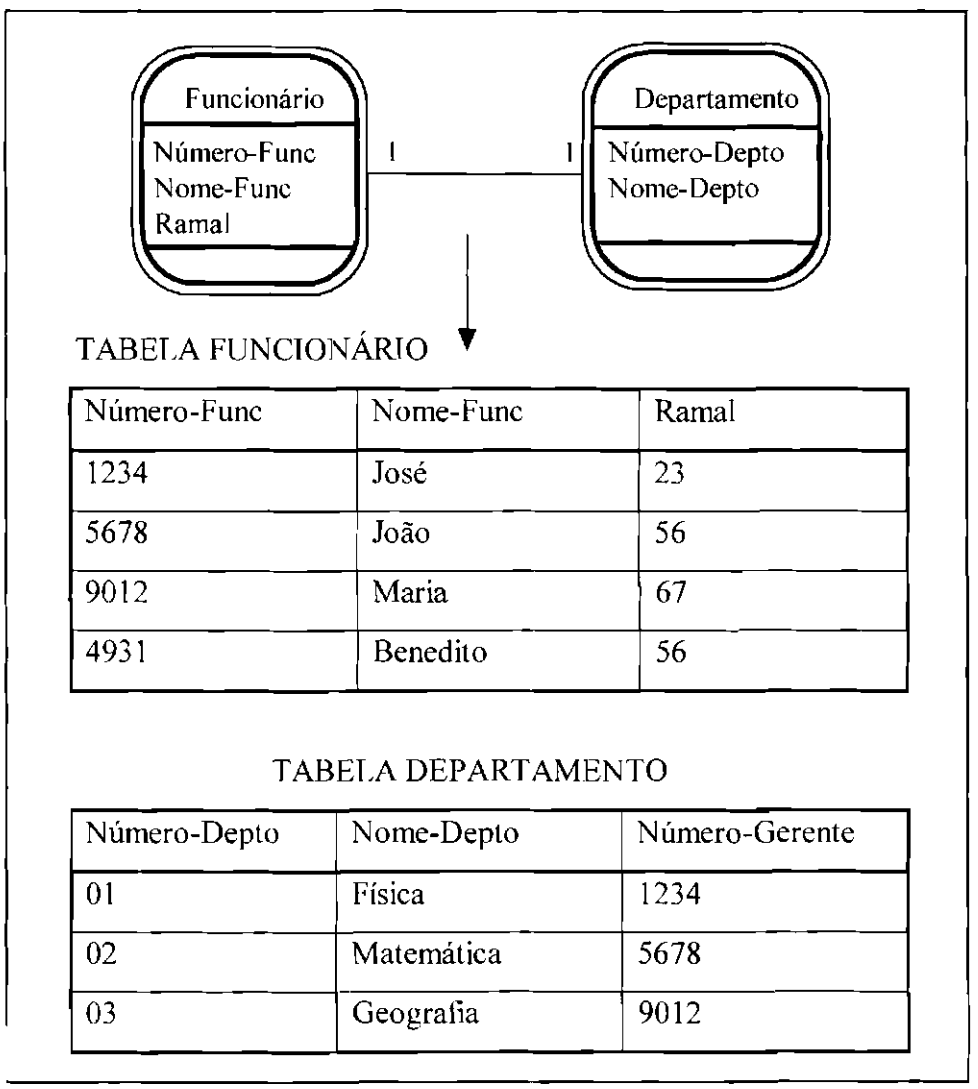


TABELA FUNCIONÁRIO

Número-Func	Nome-Func	Ramal
1234	José	23
5678	João	56
9012	Maria	67
4931	Benedito	56

TABELA DEPARTAMENTO

Número-Depto	Nome-Depto	Número-Gerente
01	Física	1234
02	Matemática	5678
03	Geografia	9012

Fig. 4.29 - Primeira Alternativa para Mapear Conexões 1 : 1 em Tabelas.

• **Caso 2.** Transpõe-se a chave de "Departamento" para "Funcionário", como mostra a Figura 4.30 a seguir.

TABELA FUNCIONÁRIO

Número-Func	Nome-Func	Ramal	Número-Depto-Gerencia
1234	José	23	01
5678	João	56	02
9012	Maria	67	03
4931	Benedito	56	----

TABELA DEPARTAMENTO

Número-Depto	Nome-Depto
01	Física
02	Matemática
03	Geografia

Fig. 4.30 - Segunda Alternativa para Mapear Conexões 1 : 1 em Tabelas.

4.5.2.6 - Mapeamento de Generalizações

Existem três abordagens principais para o mapeamento de generalizações de classes persistentes em tabelas. Serão dados exemplos das três abordagens tomando-se como base a Figura 4.31. São elas :

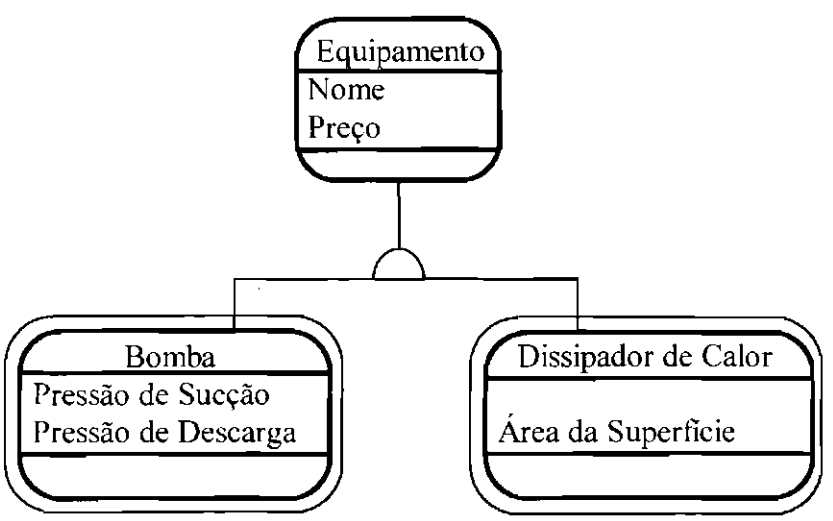


Fig. 4.31 - Equipamento - Bomba - Dissipador de Calor.

- **Caso 1.** A classe de generalização e as classes de especialização são mapeadas cada uma para uma tabela. A chave da tabela da classe de generalização é reproduzida nas tabelas das classes de especialização. Um exemplo é dado na Figura 4.32 a seguir.

TABELA EQUIPAMENTO

Id-Equipamento	Nome	Preço	Tipo
345	Bomba de Recalque	R$ 35,00	Bomba
854	Radiador	R$ 50,00	Dissipador
987	Bomba de Sucção	R$ 40,00	Bomba
097	Ar Condicionado	R$ 25,00	Dissipador

TABELA BOMBA

Id-Equipamento	Pressão de Sução	Pressão de Descarga
345	300 mmHg	4 kgf/cm2
987	-400 mmHg	5 kgf/cm2

TABELA DISSIPADOR

Id-Equipamento	Área de Superficie
097	1 m2
854	2 m2

Fig. 4.32 - Mapeamento de Generalização-Especialização em Tabelas de Generalização e em Tabelas de Especialização.

Nesse exemplo, a identidade de um objeto através de uma generalização é preservada com a utilização de um ID compartilhado. Desse modo, "Bomba de Recalque" pode ter uma linha na tabela "Equipamento" com ID 345, e outra linha na tabela "Bomba" também com ID 345.

A coluna "Tipo" é acrescentada à tabela "Equipamento" para que se possa saber qual tabela pesquisar, se a tabela de bombas ou se a tabela de dissipadores.

A navegação nas tabelas do exemplo anterior poderia ser feita da seguinte forma :

1. o usuário fornece o nome do equipamento;

2. descobre-se a linha da tabela "Equipamento" que corresponde ao nome do equipamento;

3. recupera-se o ID do equipamento e o tipo do equipamento para essa linha da tabela, e

4. vai-se para a tabela da classe de especialização indicada pelo tipo do equipamento, e encontra-se a linha com o mesmo ID da linha da tabela "Equipamento".

- **Caso 2.** Cada classe de especialização é mapeada para uma tabela. A classe de generalização é eliminada, e seus atributos são reproduzidos em cada tabela de especialização. Um exemplo é dado na Figura 4.33 a seguir.

TABELA BOMBA

Id-Equipamento	Nome	Preço	Pressão de Sução	Pressão de Descarga
345	Bomba de Recalque	R$ 35,00	300 mmHg	4 kgf/cm2
987	Bomba de Sucção	R$ 40,00	-400 mmHg	5 kgf/cm2

TABELA DISSIPADOR

Id-Equipamento	Nome	Preço	Área de Superfície
097	Ar Condicionado	R$ 25,00	1m2
854	Radiador	R$ 50,00	2m2

Fig. 4.33 - Mapeamento de Generalização-Especialização em Tabelas de Especialização.

• **Caso 3.** A classe de generalização é mapeada para uma tabela juntamente com os atributos das classes de especialização. Um exemplo é dado na Figura 4.34 a seguir.

TABELA EQUIPAMENTO

Id-Equipamento	Nome	Preço	Tipo	Pressão de Sucção	Pressão de Descarga	Área de Superfície
345	Bomba de Recalque	R$ 35,00	Bomba	300 mmHg	4 kgf/cm2	----------
854	Radiador	R$ 50,00	Dissipador	----------	----------	2 m2
987	Bomba de Sucção	R$ 40,00	Bomba	-400 mmHg	5 kgf/cm2	----------
097	Ar Condicionado	R$ 25,00	Dissipador	----------	----------	1 m2

Fig. 4.34 - Mapeamento de Generalização-Especialização em Tabela de Generalização.

Porém, quando deve-se usar o **Caso 1**, quando deve-se usar o **Caso 2**, e quando deve-se usar o **Caso 3** ? Abaixo seguem-se algumas vantagens e desvantagens de cada abordagem. Elas devem ser analisadas de acordo com os requisitos estabelecidos pelo sistema que está sendo desenvolvido (Rumbaugh,1994) :

1) A Figura 4.32 apresenta a abordagem padrão, que é a mais correta e extensiva logicamente. Contudo ela envolve muitas tabelas, e a navegação das classes de generalização para as classes de especialização pode ser lenta.

2) A Figura 4.33 apresenta uma abordagem que pode ser utilizada se as classes de especialização tiverem muitos atributos, a classe de generalização tiver poucos atributos, e se a aplicação souber qual classe deve ser pesquisada. Ela é menos satisfatória que a abordagem padrão; pois não se pode impor unicidade de nomes de equipamento através das tabelas de especialização.

3) A Figura 4.34 apresenta a abordagem menos satisfatória, porém ela pode ser útil se existirem somente duas ou três classes de especialização com poucos atributos.

4) As abordagens das Figura 4.33 e Figura 4.34, são abordagens alternativas motivadas pelo desejo de eliminar a navegação da classe de generalização para as classes de especialização, melhorando o desempenho (*performance*); contudo a melhora de desempenho incorre em outros problemas como repetição de informações em uma tabela, introdução de valores nulos nas tabelas, e possível perda ou inconsistência de informações.

Após a construção do modelo relacional, se o sistema for usar um sistema de gerenciamento de bancos de dados relacionais, ou um sistema de gerenciamento de bancos de dados relacionais estendidos, deve-se definir na fase de implementação uma tabela do banco de dados para cada tabela do modelo relacional.

4.6 - Definindo um Plano de Implementação e Testes

Após realizar a especificação para as classes, e o projeto dos dados que irão persistir (se for necessário) deve-se definir um plano de implementação e testes para o sistema (para mais detalhes sobre o plano de implementação e testes, ver o item **3.2.7- Definição do Plano de Implementação e Testes**, no Capítulo 3 desta dissertação).

A seguir apresentaremos algumas diretrizes para se implementar e testar um sistema de pequeno porte orientado para objetos. Essas diretrizes abrangem a adoção de uma estratégia de implementação e testes, e o projeto de casos de teste.

4.6.1 - Adotando uma Estratégia de Implementação e Testes

Antes de começar a implementação e testes do sistema deve-se adotar uma estratégia para que essas atividades possam ser realizadas com eficiência. A estratégia

recomendada nesse caso consiste em se implementar e testar cada cenário do sistema separadamente, e depois integrá-los, um por vez, para testar o funcionamento do sistema como um todo (Coad,1997).

Porém, por qual cenário deve-se começar a implementar e testar o sistema ? Deve-se começar pelo cenário que provê a habilidade mais significativa para o usuário, ou seja, deve-se analisar a especificação inicial do sistema e verificar o que é mais importante que o sistema faça. Assim, pode-se definir uma lista de prioridades, determinando uma ordem para a implementação e testes.

Cada classe pertencente a um determinado cenário, por sua vez, também deve ser implementada e testada separadamente. Porém, qual classe do cenário deve ser implementada e testada primeiro ?

Deve-se começar a implementação e testes das classes de um cenário pelas classes de interface humana, e depois seguir implementando e testando as classes que vão tratar os eventos captados pela interface humana. Ao final da implementação e teste individual de todas as classes de um cenário, deve-se integrá-las para ver se o cenário está realmente cumprindo a habilidade a que ele se pré-dispôs.

Por exemplo, para o exemplo da Figura 4.19 deve-se começar a implementação e testes pelas classes "ContadorViewContainer", "Botão", e "CaixaMostravalor", seguidas das classes "Contador" e "ContadorInteiro"; e finalmente da classe "ContadorGC". Depois que todas essas classes forem implementadas e testadas individualmente, elas devem ser integradas e testadas em conjunto para verificar se o cenário está realmente cumprindo sua habilidade.

Implementar e testar cada classe significa implementar e testar seus serviços. Se um serviço invoca serviços de outros objetos, então durante os testes *Stubs* de teste podem

ser necessários para substituir os serviços que ainda não tenham sido implementados (Kirani,1994).

Testar um serviço é semelhante a testar um módulo, o que significa que o teste dos serviços correspondem ao nível de teste de unidade do paradigma clássico (Jorgensen,1994). Para os serviços complexos deve-se adotar a estratégia de implementação e testes denominada "Integração Incremental *Top-Down*". Essa estratégia está explicada no item **3.2.7.1 - Adoção de uma Estratégia de Implementação e Testes**, no Capítulo 3 desta dissertação. Ao ler esse item deve-se substituir a palavra "módulo" por "serviço".

4.6.2 - Projetando Casos de Teste

Definida a estratégia que será usada para a implementação e testes do sistema deve-se projetar os casos de teste que o sistema deverá abranger, estabelecendo também a ordem em que eles deverão ser realizados (para mais detalhes sobre casos de teste ver o item **3.2.7.2 - Projeto de Casos de Teste**, no Capítulo 3 desta dissertação).

O projeto de casos de teste para um sistema de pequeno porte orientado para objetos consiste no projeto de casos de testes para os serviços, no projeto de casos de teste para os cenários, e no projeto de casos de teste de integração dos cenários.

Projetar casos de teste para os serviços é semelhante a projetar casos de teste para os módulos. Como fazer o projeto de casos de teste para os módulos está explicado no item **3.2.7.2.1 - Projeto de Casos de Teste de Unidade**, no Capítulo 3 desta dissertação.

Projetar casos de teste para os cenários consiste em projetar casos de teste que verifiquem se a habilidade que o cenário se pré-dispôs a prover está realmente sendo cumprida (verificando a seqüência de interação entre os serviços), e em projetar casos de

teste de aceitação. O projeto de casos de teste de aceitação já foi abordado no item **3.2.7.2.2 - Projeto de Casos de Teste de Aceitação**, no Capítulo 3 desta dissertação.

Projetar casos de teste de integração dos cenários significa projetar casos de teste que testem as interfaces entre os cenários à medida que eles vão sendo integrados. Testar a interface entre os cenários significa testar se o sistema está "passando" corretamente de um cenário para outro, ou seja, verificar se a classe gerenciadora de cenários está coordenando corretamente a entrada de um novo cenário em "cena", e verificar se as mensagens trocadas entre um cenário e outro estão corretas.

4.7 - Implementando o Sistema de Pequeno Porte Usando Técnicas Orientadas para Objetos

A terceira etapa do desenvolvimento de um sistema de software é a construção do modelo físico do sistema, que é nada mais nada menos que implementá-lo. Essa etapa contribui diretamente para a qualidade do sistema sendo gerado, e por isso, é importante que se tome algumas medidas para que a qualidade possa ser mantida.

É relativamente fácil implementar um projeto baseado em objetos bem feito com uma linguagem baseada em objetos, já que as construções da linguagem são semelhantes às construções do projeto. Além disso, muito do trabalho de codificação pode ser poupado já que muitas classes podem ser reaproveitadas da biblioteca de classes oferecida pela linguagem, como por exemplo as classes de interface gráfica com o usuário (Classes GUI) providas pelas linguagens visuais, por exemplo Visual C++.

Porém, apesar dessas facilidades, medidas relacionadas à padronização e a documentação do código fonte, utilização de um número mínimo de programadores possíveis, e a utilização de ferramentas automatizadas, podem contribuir enormemente para o aumento da qualidade do código gerado (para mais detalhes ver o item **3.2.8 -**

Implementação do Sistema de Pequeno Porte Usando Técnicas Estruturadas, no Capítulo 3 desta dissertação).

A seguir, mostraremos alguns cuidados que devem ser tomados ao se implementar um sistema orientado para objetos. Esses cuidados relacionam-se principalmente com o estilo de programação que deve ser adotado. Entre os elementos de estilo incluem-se a documentação interna (nível de código fonte) e externa dos serviços; além de uma documentação interna (nível de código fonte) e externa para as classes.

4.7.1 - Documentação Externa e Documentação Interna para um Serviço

Como já dito, um serviço pode ser visto como um módulo. Assim sendo, a mesma documentação externa e a mesma documentação interna que foi estabelecida para os módulos pode ser aplicada aos serviços.

Como fazer a documentação externa de um módulo já foi explicado no item **3.2.8.1 - Documentação Externa do Módulo**, no Capítulo 3 desta dissertação. Como fazer a documentação interna de um módulo já foi explicado nos itens **3.2.8.2 - Codificação de um Módulo**, à **3.2.8.2.4 - Documentação Interna do Código**, no Capítulo 3 desta dissertação. Ao ler esses itens deve-se substituir a palavra "módulo" por "serviço", e a palavra "sub-rotina" por "sub-serviço".

4.7.2 - Documentação Externa e Documentação Interna para uma Classe

A documentação externa de uma classe deve aparecer no início de cada classe como se fosse um cabeçalho. O formato de tal documentação pode ser :

1) uma descrição da classe;

2) suas classes de generalização (super-classes) se ela for uma classe de especialização; e suas classes de especialização (sub-classes) se ela for uma classe de generalização;

3) seus atributos e serviços públicos;

4) seus atributos e serviços privados e

5) classes com que interage.

A documentação interna de uma classe consiste em se fazer uma documentação externa e uma documentação interna para os serviços da classe, como explicado no item anterior, e em se fazer uma documentação de código para a classe. A documentação de código da classe consiste em se colocar comentários embutidos no código fonte, usados para descrever os atributos da classe, e para separar as seções de uma classe (seção pública, seção privada, e outros).

Um exemplo de documentação externa para a classe "Contador" da Figura 4.19 é dado na Figura 4.35 adiante.

4.7.3 - Implementação do Sistema Usando o Visual C++

O Visual C++ é um ambiente de programação que combina a linguagem C++ e a biblioteca "Microsoft Foundation Class (MFC)", o que faz com que ele tenha um poderoso conjunto de ferramentas para programação nos ambientes DOS e Windows. A biblioteca "Microsoft Foundation Class" é uma coleção de cerca de cem classes divididas em três categorias (Barkakati,1997) :

1) classes para construir aplicações Windows, incluindo classes que simplificam o uso do dispositivo de interface gráfica (bibliotecas GUI) e suporte a incorporação e *linkagem* de objetos;

2) classes de propósito geral, como listas encadeadas, matrizes dinâmicas, arquivos e *strings*, incluindo recursos para suportar manipulação de exceção e arquivar objetos em arquivos, e

3) macros e globais, que consistem em várias *macros* e objetos globais de propósito geral.

Isso significa que ao se usar o Visual C++, se o programador estiver familiarizado com a biblioteca MFC, pode-se reduzir consideravelmente o tempo e o esforço gastos na programação, já que muitas classes poderão ser reaproveitadas.

Nome da Classe : Contador
Descrição : Classe que representa um contador , que é capaz de se incrementar, se decrementar, e se inicializar. Essa classe não tem objetos diretos.
Super-Classes : -------
Sub-Classes : ContadorInteiro
Atributos Públicos : --------
Atributos Privados : --------
Atributos Protegidos : valor valorInicial
Serviços Públicos : Contador ( ) ~Contador incrementar decrementar inicializar acessarValor setarValor acessarValorInicial setarValorInicial
Serviços Privados : --------
Serviços Protegidos : -------
Objetos para os quais seus objetos mandam mensagens : --------

Fig. 4.35 - Documentação Externa Para uma Classe.

Então, para plenamente utilizar o poder e flexibilidade do ambiente Visual C++, deve-se primeiramente familiarizar-se com três coisas distintas : C++, conceitos e técnicas de modelagem, projeto e programação orientada para objeto, e ferramentas e bibliotecas que são parte do Visual C++.

4.7.3.1 - Algumas Dicas sobre o C++

A seguir são apresentadas algumas dicas sobre o C++. Essas dicas concentram-se principalmente em traçar um paralelo entre o projeto orientado para objeto e a implementação usando C++.

O C++ usa o termo "variáveis" (também chamadas de "data members") para implementar os atributos, usa o termo "funções membro" ("member functions") para implementar os serviços, e usa o termo "membro" ("member") para designar funções membro e variáveis. Uma função membro de um objeto não pode ter o mesmo nome que uma variável desse objeto.

Em C++ primeiramente deve-se "declarar" a classe, e depois definir as funções membro para aquela classe. Declarar uma classe significa declarar seus construtores e destrutores (serviços "Criar" e "Liberar"), declarar as funções que definem seus padrões de comportamento, declarar seus acessadores (serviços "Setar" e "Acessar"), declarar suas variáveis, e declarar seus conectores (serviços "ConectarAo" e "DesconectarDe").

Então, cada atributo e serviço do diagrama de classe&objeto do modelo lógico deve ser declarado, respectivamente, como uma variável e como uma função membro integrante da classe correspondente à sua classe do modelo lógico. É boa prática transportar para a implementação os nomes das classes, atributos e serviços do diagrama de classe&objeto do modelo lógico.

Os membros são declarados com um tipo de acesso específico, ou seja, eles podem ser "públicos" (public), "protegidos" (protected) ou "privados" (private). Qualquer função pode acessar um membro público. Somente as funções membro da classe e de suas classes derivadas podem acessar um membro protegido. Somente funções membro da classe podem acessar um membro privado. Uma classe de especialização herda todos os membros, até mesmo aqueles declarados privados.

A seguir é mostrado a forma geral da declaração de uma classe, para uma melhor clareza as palavras e símbolos reservados do C++ estão em negrito :

```
class nomeClasse {
  // private por default
    tipo1 variável1;
    tipo2 variável2;
    funçãoMembro1;
    funçãoMembro2;
    //....
  public:
    tipo3 variável3;
    tipo4 variável4;
    funçãoMembro3;
        //...
  protected:
    //...
}
```

Após a declaração da classe deve-se definir as funções membro para aquela classe. As funções membro podem ser definidas em um arquivo separado, deste modo, pode-se pensar no arquivo que contém a declaração das classes como sendo um arquivo que

especifica a interface para a classe (arquivo cabeçalho), enquanto que o arquivo contendo as definições das funções é a implementação propriamente dita da classe.

Idealmente, se a interface for definida de forma bastante clara, os programadores poderão usar a classe sem precisar saber os detalhes de implementação daquela classe. É boa prática conservar o mesmo nome para o arquivo de declaração e de definição da classe, com apenas extensões diferentes, por exemplo, pode-se usar a extensão ".CPP" para os arquivos fonte C++, e a extensão ".H" para os arquivos cabeçalho.

O C++ usa os termos "Construtor" e "Destrutor" para os serviços "Criar" e "Liberar" respectivamente. Podem ser definidos múltiplos construtores para uma única classe, identificados pelo número e tipos dos seus *argumentos*. Um construtor é executado sempre que é alocada uma instância de um novo objeto. No momento da alocação, o programador pode especificar os *argumentos* para o construtor; e o construtor com os tipos de *argumentos* iguais é executado. Se nenhum *argumento* é especificado, então é executado o construtor *default*. Os construtores devem ter o mesmo nome da classe.

Cada classe pode ter apenas um destrutor. Os destrutores não exigem nenhum *argumento*. Eles devem ter o mesmo nome da classe, porém precedido por um til (~). A definição de construtores e destrutor é fundamental para classes que contêm *alocação dinâmica* de memória.

Um exemplo de construtores e destrutor para a classe "ContadorInteiro" da Figura 4.19 é dado a seguir :

```
class ContadorInteiro : public Contador<int> {
  public:
    // Construtores e Destrutor
    ContadorInteiro ( ) { valorInicial = 0; inicializar ( ); }  // Construtor default
    ContadorInteiro ( int novo_valor ) { valorInicial = 0; valor = novo_valor; }
    ~ContadorInteiro ( ) { } //Destrutor
}
```

Neste exemplo, os construtores e o destrutor para a classe "ContadorInteiro" foram declarados e definidos ao mesmo tempo. Em C++ quando a função membro é extremamente pequena pode-se declará-la e defini-la ao mesmo tempo (função membro "inline"). Nesse caso o *compilador* inserirá o código "inline" em cada lugar que a função membro é chamada, economizando alguns microsegundos por não ter que "chamar" a função membro; porém adicionando alguns *bytes* a mais no código compilado.

Quando uma classe for uma especialização de alguma outra classe deve-se incluir na primeira linha da sua declaração o símbolo ":" seguido por uma lista de classes de generalização das quais essa classe é uma especialização, como mostra o exemplo anterior.

Nesse exemplo o símbolo "<int>" nos diz que a classe "ContadorInteiro" é uma especialização da classe "Contador" com tipo parametrizado para inteiro. Um tipo parametrizado é um tipo que não pode ser definido na classe de generalização pois ele é dependente da classe de especialização, e por isso deve ser definido posteriormente.

Pode ser necessário usar tipos parametrizados na declaração de uma classe de generalização, pois ela pode conter variáveis que tenham os tipos dependentes das especializações da classe, um exemplo disso são os atributos "valor" e "valorInicial" da classe "Contador" da Figura 4.19.

Quando isso acontece deve-se declarar a classe usando o termo "template". Um "template" permite que se declare variáveis com tipo parametrizado. Pode-se declarar também funções membro que retornem um tipo parametrizado. Um exemplo de declaração usando template para a classe "Contador" é dado a seguir :

```
template<class tipo_contador>
  class Contador {
          public:
            //Construtor e Destrutor
            Count ( ) { }
            virtual ~Count ( ) { }
            // Funções que definem os padrões de comportamento
            virtual void incrementar ( ) = 0;
            virtual void decrementar ( ) = 0;
            void inicializar ( ) { valor = novo_valor; }
            // Acessadores
            tipo_contador acessarValor ( ) { return valor; }
            void setarValor ( tipo_contador novo_valor ) { valor = novo_valor; }
            tipo_contador acessarValorInicial ( ) { return valorInicial; }
            void setarValorInicial ( tipo_contador novo_valor ) { valor = novo_valor; }
          protected:
            // Variáveis
            tipo_contador valor;
            tipo_contador valorInicial;
}
```

Neste exemplo, o tipo "tipo_contador" é um tipo parametrizado, pois ele pode variar de acordo com a classe de especialização, ou seja, se a classe de especialização for um contador inteiro "tipo_contador" será do tipo "int", se a classe de especialização for um contador real "tipo_contador" será do tipo "float", e assim por diante.

Ao se declarar em uma classe de generalização uma função membro que deve ser redefinida em cada classe de especialização, deve-se usar a seguinte sintaxe :

virtual void minha_função_membro ( ) = 0;

A palavra "virtual" diz para o *compilador* C++ que a função membro definida na classe de generalização deve ser usada apenas se as classes de especialização não a redefinirem. A palavra "void" indica que não é esperado que essa função membro tenha algum valor de retorno, o que faz com que essa função membro aja de uma maneira bem parecida com o que a linguagem Pascal ou Ada chamam de *procedure*. A sintaxe "= 0" indica que cada classe de especialização concreta (que tem objetos diretos) deve definir sua própria versão para a função membro. Um exemplo de função virtual que deve ser redefinida na classe de especialização é a função membro "incrementar" da classe "Contador" mostrada no exemplo anterior.

O C++ permite que se tenha funções "friend". Ao declarar uma função membro externa à uma classe como sendo "friend" dessa classe, permite-se que ela acesse diretamente os dados que são privados dessa classe. As funções "friend" devem ser evitadas, pois elas violam um dos princípios básicos da orientação a objetos, que é o encapsulamento.

No C++ constrói-se programas orientados para objetos criando instâncias das classes (os objetos) conforme necessário. Um exemplo de criação de um objeto para a classe "ContadorInteiro" é dado a seguir :

```
int main ( ) {
// Cria um contador inteiro com valor inicial igual a 5
ContadorInteiro meu_contador ( 5 );
return 0;
}
```

Em C++ o programa faz seu trabalho chamando funções membros dos objetos. A sintaxe para chamar funções membros é semelhante à sintaxe usada para chamar qualquer outra função, exceto por se usar operadores "." e "->" para identificar a função membro dentro do objeto. Um exemplo de chamada para a função membro "incrementar" da classe "ContadorInteiro" é dado abaixo :

```
// Dizendo ao contador para ele se incrementar
meu_contador.incrementar ( );
```

O C++ implementa o polimorfismo de duas maneiras, **(1)** através da sobrecarga estática de funções virtuais, e **(2)** através da incorporação dinâmica.

No primeiro caso há uma sobreposição de uma função definida em uma classe de generalização com uma função definida na classe de especialização. Um exemplo de polimorfismo usando sobrecarga estática de funções é dado abaixo :

```
class shape {
  public:
    virtual void draw ( void ) const { }
    // Outras funções membro...
}
```

Neste caso, a classe de generalização define "draw" como uma função membro que não faz nada. Pode-se sobrepor essa função membro nas classes de especialização construindo uma função membro com o mesmo nome. Um exemplo disso é dado a seguir :

```
class circle_shape : public shape {
  // Dados privados...
 public:
 // Outras funções membro
 virtual void draw ( void ) const;
}
```

Mais tarde deve-se realmente definir a função "draw" para a classe "circle_shape". Pode-se fazer o mesmo para a classe "rectangle_shape". Uma vez feito isso, pode-se aplicar a mesma função membro para instâncias de classes diferentes, e o *compilador* C++ vai gerar uma chamada para a função "draw" correta. Um exemplo disso é dado a seguir :

```
// Cria instâncias de circle_shape e rectangle_shape
circle_shape c1 (100.,100.,50.);
rectangle_shape r1 (10.,20., 30., 40.);

c1.draw ( ); // "draw" da classe "circle_shape" é chamada
r1.draw ( ); // "draw" da classe "rectangle_shape" é chamada
```

Embora este seja um comportamento polimórfico, tanto o *compilador* C++ quanto o programador pode determinar, examinando o código, exatamente a função que deve ser chamada.

No segundo caso, uma função virtual é chamada através de um *ponteiro* para um objeto, e o tipo desse objeto não é conhecido em tempo de compilação. Isso significa que a função que é realmente chamada varia de acordo com o tipo que o *ponteiro* determina em tempo de execução. Isso é possível porque o C++ permite que se use um *ponteiro* para uma classe de generalização quando se refere a uma instância de uma classe de especialização. Por exemplo, suponha que se queira criar várias formas, armazená-las em uma matriz, e desenhá-las. Veja a seguir, como isso pode ser feito :

```
int i;
shape *shapes [2]; // Matriz de ponteiros para a classe de generalização

// Criando algumas formas e salvando ponteiros
shapes [0] = new circle_shape (100.,100.,50.);
shapes [1] = new rectangle_shape (80.,40.,120.,60.);

// Desenhando formas
for ( i = 0; i < 2; i++)
    shapes [i] -> draw ( );
```

Repare que pode-se passar pelos ponteiros para as formas e chamar a função membro "draw" de cada objeto. Como "draw" é uma função virtual, a verdadeira função "draw" chamada em tempo de execução depende do tipo de forma que o *ponteiro* "shapes [i]" referencia. Nesse caso, a função membro que é realmente chamada varia de acordo com o tipo que o *ponteiro* determina em tempo de execução.

CAPÍTULO 5

SISTEMAS DE MÉDIO PORTE

Como já dito, uma grande dificuldade para se mensurar um sistema está no fato de não existirem fronteiras precisas que separem sistemas de grande, médio e pequeno porte; ficando muitas vezes difícil saber qual é realmente o porte do sistema que será desenvolvido.

Em geral, sistemas de médio porte requerem de 2 a 5 desenvolvedores, trabalhando por 1 ou 2 anos, e resultam em um software de 10.000 a 50.000 linhas de código, distribuídas em 250 a 1.000 *sub-rotinas* (Fairley,1985).

Sistemas de médio porte geralmente tem poucas, ou nenhuma interação com outros sistemas. Exemplos de sistemas de médio porte podem ser *compiladores*, sistemas de gerenciamento de informações pequenos, aplicações de controle de processos.

Como nesse tipo de sistema a equipe de desenvolvimento tem mais de uma pessoa, o desenvolvimento requererá interações entre esses desenvolvedores para que se possa ter um controle e padronização dos produtos que estão sendo gerados. Interações entre desenvolvedores e usuário também devem ser mais freqüentes, para que se possa assegurar que os requisitos estabelecidos pelo usuário estão sendo cumpridos.

No desenvolvimento desse tipo de sistema o uso de princípios sistemáticos de engenharia pode resultar num grande melhoramento da qualidade, num aumento da produtividade do desenvolvedor, e num aumento da satisfação das necessidades do usuário. Por isso, em sistemas desse tipo é requerido um certo grau de formalidade no planejamento, documentação, e revisões de projeto.

Além disso, num sistema de médio porte, ferramentas automatizadas, como ferramentas CASE, podem ser muito úteis para ajudar no trabalho, portanto é interessante lançar mão desse artifício. CASES para o desenvolvimento de sistemas usando o paradigma clássico podem ser, por exemplo, "Talisman", "EasyCase", "System Architect". Para o paradigma da orientação a objetos podem ser, por exemplo, "Paradigm Plus", "Together C++", "System Architect".

Ao longo deste Capítulo forneceremos diretrizes para se construir um Sistema de Médio Porte usando técnicas estruturadas para a análise, projeto, implementação e testes do sistema (para mais detalhes sobre técnicas estruturadas para o desenvolvimento de sistemas ver o item **2.10.1.1 - Técnicas Estruturadas**, no **Capítulo 2 - Software e Engenharia de Software**, desta dissertação). Para a modelagem dos dados utilizaremos técnicas não estruturadas (para mais detalhes sobre técnicas não estruturadas para o desenvolvimento de sistemas ver o item **2.10.1.2 - Técnicas não Estruturadas**, no Capítulo 2 desta dissertação). Estaremos neste Capítulo, portanto, traçando uma rota que pode ser seguida por aqueles que queiram construir um sistema desse tipo.

A construção de Sistemas de Médio Porte usando o paradigma da orientação a objetos (para mais detalhes sobre o paradigma da orientação a objetos ver o item **2.10.2 - Paradigma da Orientação a Objetos**, no Capítulo 2 desta dissertação) será abordada no **Capítulo 6 - Construção do Sistema de Médio Porte Usando o Paradigma da Orientação a Objetos**, desta dissertação.

Como já dito, a fim de ilustrar e validar as diretrizes para a construção de sistemas de software que são propostas nessa dissertação, adotou-se um sistema exemplo que foi retirado de Gomaa (1984).

No **Capítulo 3 - Sistemas de Pequeno Porte**, desta dissertação, foi abordado o desenvolvimento de apenas uma parte desse exemplo, já que como dito anteriormente, essa parte por si só pode ser considerada como um sistema de pequeno porte. Neste

Capítulo, abrangeremos o desenvolvimento do Sistema Controlador de um Robô como um todo.

5.1 - Definição do Sistema

Antes de começar a construção propriamente dita deve-se fazer uma definição do sistema a ser construído.

Como realizar esta atividade está explanado no item **3.1 - Definição do Sistema** no Capítulo 3 desta dissertação. Ao ler esse item deve-se ignorar o exemplo dado, e substituí-lo pelo exemplo que será apresentado neste Capítulo.

A definição do sistema de médio porte deve incluir também alguns fatores que não eram necessários serem especificados no sistema de pequeno porte. Isso acontece porque num sistema de médio porte começam a aparecer problemas que não apareciam, ou pelo menos eram bem mais suaves, no sistema de pequeno porte; como por exemplo a padronização dos produtos (software e documentos), o controle das versões do software, a comunicação entre os membros da equipe, e o controle do desenvolvimento.

No sistema de pequeno porte a equipe era constituída de uma única pessoa, o que fazia com que esses problemas não tivessem uma importância significativa. No desenvolvimento de um sistema de médio porte porém, onde o sistema pode ser dividido em diversos subsistemas, e esses subsistemas podem ser distribuídos entre os componentes da equipe para que se possa fazer um desenvolvimento em paralelo; esses problemas começam a aparecer.

Por isso, apesar da equipe ser pequena, durante a definição do sistema é necessário estabelecer um padrão para o formato e qualidade dos produtos que serão gerados, e estabelecer também regras para a alteração desses produtos.

Para que a verificação da qualidade dos produtos seja feita de uma maneira organizada, deve-se designar uma pessoa da equipe para ser responsável pelos documentos e softwares produzidos. Essa pessoa será responsável por examinar cada documento e software gerado, verificando se eles estão de acordo com os critérios de qualidade e com os requisitos estabelecidos durante a definição do sistema. Essa pessoa deve ser responsável também pelo controle das versões do sistema. O controle das versões consiste nas atividades de gerar e manter as versões do sistema, sendo que uma versão é definida através de procedimentos que especificam os produtos que a compõe.

As alterações em qualquer documento ou software também devem ser regulamentadas, e só poderão ser feitas com autorização de toda a equipe. Então, para realizar alterações a equipe conversaria entre si e chegaria a um ponto comum, autorizando ou não a mudança.

Antes de apresentarmos a definição para o Sistema Controlador de um Robô, gostaríamos de citar as principais características de um sistema de tempo real (Coad,1997) :

- **Tempo.** Funções devem ser realizadas à tempo.

- **Estrutura Interna Dinâmica.** Ativação e desativação dinâmica dos componentes do software.

- **Resposta a Eventos.** Responder continuamente a diferentes eventos detectados pela aquisição de dados e dispositivos de controle, e pela interação com outros sistemas.

- **Dependência de Estados.** Responder diferentemente dependendo do estado do sistema.

- **Concorrência.** Muitas atividades podem estar acontecendo ao mesmo tempo.

- **Abstrações Múltiplas.** Necessita de modelos que representem abstrações lógicas e físicas.

- **Distribuição.** Sistema opera e pode ser acessado de várias máquinas.

A seguir é apresentada a definição para o Sistema Controlador de um Robô :

- **Propósito do Sistema :** O propósito do sistema é controlar os movimentos de até 6 eixos de um robô, e interagir com sensores digitais em entrada/saída (E/S) de dados, através de um painel de controle constituído de um conjunto de botões e de um seletor de programas.

- **Modelo Descritivo do Sistema :** O objetivo do sistema é controlar os movimentos de até 6 eixos de um robô, e interagir com sensores digitais de entrada/saída (E/S) de dados. O controle dos eixos e E/S é efetuado por um programa inicializado de um painel de controle, que consiste de um conjunto de botões e de um seletor de programas, como mostra a Figura 5.1 a seguir.

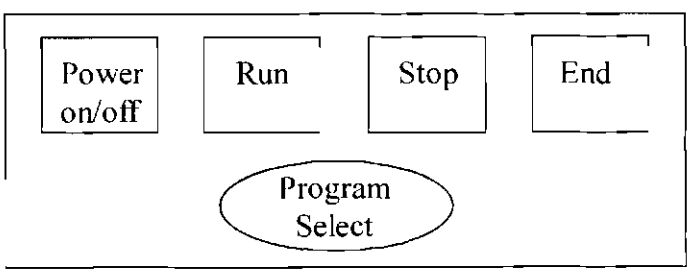


Fig. 5.1 - Painel de Controle do Robô.

Cada vez que um botão do painel é pressionado uma ação diferente será realizada pelo sistema, e luzes se acenderão no painel indicando a operação escolhida. As entradas geradas ao se apertar um botão devem ser convertidas para o formato interno do sistema para poderem ser recebidas e validadas, sendo que as entradas inválidas por questões de simplicidade, devem ser ignoradas.

Quando o botão "Power" é pressionado inicia-se a energização do sistema. Se essa seqüência de energização for completada com sucesso, o sistema passa para o controle manual. Durante o controle manual o operador pode selecionar um programa usando a chave seletora de programas para indicar o número do programa desejado; e quando a posição da chave seletora de programas for alterada, a nova indicação deve ser passada para o sistema.

Ao se pressionar o botão "Run" inicia-se a execução do programa selecionado. As instruções aritméticas e lógicas do programa podem ser processadas diretamente, porém as instruções de movimento dos eixos e as instruções de E/S de sensores devem receber processamentos adicionais. As instruções de movimento devem ser convertidas para "blocos de eixo" para poderem ser entendidas pelos eixos. Da mesma forma as instruções de E/S de sensores devem ser convertidas para o formato do sensor para poderem ser entendidas pelos sensores. Quando os eixos terminam de realizar os movimentos requisitados eles devem informar ao sistema sobre esse término.

A execução do programa pode ser suspensa pressionando-se o botão "Stop", e isso faz com que a execução do programa fique suspensa até que o botão "Run" seja pressionado novamente dando continuidade à execução; ou até que a execução do programa seja abortada pressionando-se o botão "End". Fazendo isso o sistema volta a operar no modo manual esperando uma nova seleção de um programa. Quando o botão "Power" é pressionado novamente o sistema é desligado.

• **Identificação do Sistema :** Sistema Controlador de um Robô.

• **Lista de Entidades do Sistema :**

- **Entidade Botão.** Envia para o sistema dados a respeito do botão pressionado.
- **Entidade Sensor.** Envia e recebe dados dos sensores que interagem com o robô.
- **Entidade Painel.** Mostra ao usuário através de luzes qual botão foi pressionado.
- **Entidade Eixo.** Realiza os movimentos do robô.
- **Entidade Seletor.** Envia para o sistema dados sobre os programas selecionados.

• **Diagrama de Contexto do Sistema :**

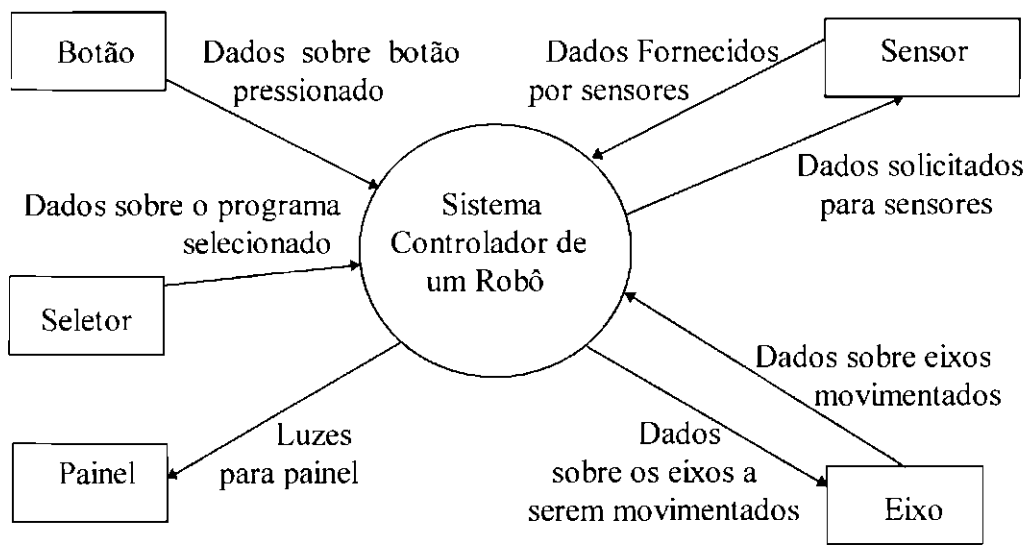


Fig. 5.2 - Diagrama de Contexto do Sistema Controlador de um Robô.

- **Eventos do Sistema e Reações Provocadas por eles :**

  - **Um botão é pressionado.** O sistema lê as entradas e converte-as para seu formato interno. As entradas são validadas e enviadas para o processamento adequado. Luzes no painel são acesas indicando a operação escolhida.

  - **Um novo programa é escolhido.** O sistema muda a indicação do programa a ser executado.

  - **Chegada de dados de sensores.** Os dados de sensores são convertidos para o formato interno do sistema e enviados para processamento adequado.

  - **Chegada de dados sobre os eixos movimentados.** Quando o bloco de eixo executado corresponde ao fim das instruções de movimento, deve-se informar o sistema que as instruções de movimento já foram executadas.

Após a definição do sistema pode-se começar sua construção propriamente dita; e cabe ao desenvolvedor decidir de que maneira ele deseja desenvolver o sistema, se pela maneira tradicional, ou se pela orientação a objetos.

Se a opção for feita por desenvolver o sistema usando técnicas orientadas para objetos, vá para o **Capítulo 6 - Construção do Sistema de Médio Porte Usando o Paradigma da Orientação a Objetos**, desta dissertação.

5.2 - Construção do Sistema de Médio Porte Usando o Paradigma Clássico

Se o sistema a ser desenvolvido for usar técnicas tradicionais, após a definição deve-se partir para a construção do modelo conceitual do sistema (para mais detalhes sobre modelos de abstração do software, ver o item **2.9 - Modelagem e Abstração**, no

Capítulo 2 desta dissertação), o que significa partir para a construção do diagrama de fluxo de dados.

O diagrama de fluxo de dados (DFD) deve mostrar as relações entre os dados e as funções ou processos que transformam esses dados, como ilustra o exemplo da Figura 5.3.

O DFD é constituído de 4 componentes básicos : os fluxos de dados, os processos, as entidades externas, e os depósitos de dados (Gane,1983). Informações como "quando as coisas acontecem", "detalhes físicos", e "fluxos de controle", não são representadas pelo DFD.

Entidades Externas são, com maior freqüência, categorias de objetos, corporações ou pessoas externas ao sistema que representam uma fonte ou destino para transações. Elas representam portanto, a interface entre o sistema e o mundo externo.

Uma entidade externa é representada por um quadrado sólido, cujos lados de cima e à esquerda são representados por traços duplos para destacar o símbolo do resto do diagrama. Como referência à entidade externa, esta deve ser identificada por uma letra minúscula no canto superior à esquerda. No exemplo da Figura 5.3, "Botão", "Seletor", "Painel", "Sensor", e "Eixo" são entidades externas.

Os Fluxos de Dados podem ser considerados como tubos por onde passam pacotes de dados, que fluem em tempo não especificado. Os fluxos de dados são representados por meio de uma seta, de preferência horizontal e/ou vertical, com a ponta indicando a direção do fluxo.

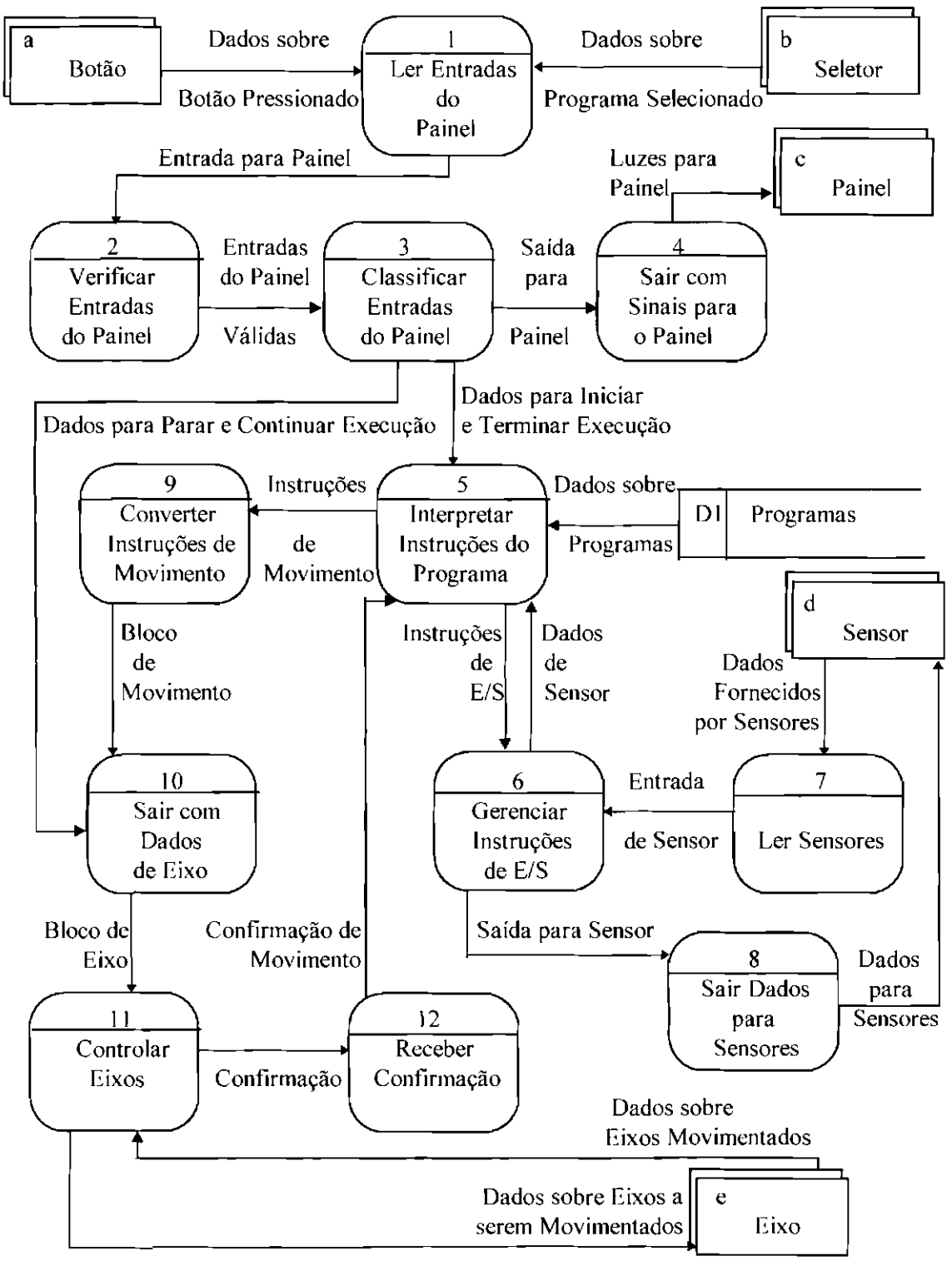


Fig. 5.3. Diagrama de Fluxo de Dados - Sistema Controlador de um Robô.

Cada fluxo de dados deve conter uma descrição de seu conteúdo (nome) ao longo de sua extensão. Essa descrição deve ser mostrada no diagrama com letras minúsculas e maiúsculas. No exemplo da Figura 5.3, "Dados sobre Botão Pressionado", "Dados sobre Programa Selecionado", "Entrada para Painel", etc..., são descrições de fluxos de dados.

Os Processos ou funções, transformam os dados de alguma maneira (manual ou automática). No exemplo da Figura 5.3, "Ler Entradas do Painel", "Verificar Entradas do Painel ", "Classificar Entradas do Painel", etc..., são processos.

Os processos são representados por retângulos "em pé" com vértices arredondados, divididos opcionalmente em três áreas, como mostra a Figura 5.4 abaixo.

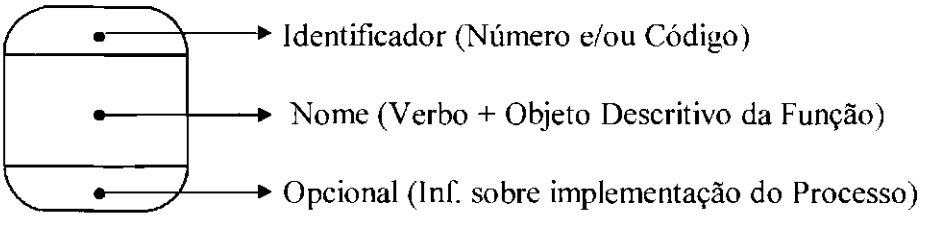


Fig. 5.4 - Símbolo de Processo.

O identificador pode ser um número ou um código, mas não tem nenhum significado além de identificar o processo. O nome deve ser uma sentença imperativa, consistindo num verbo ativo sem ambigüidade (criar, produzir, extrair, recuperar, armazenar, computar, calcular, determinar, verificar, etc...), seguido de uma cláusula objeto, a mais simples possível. A parte inferior, opcional, da caixa de processo, pode conter uma especificação do local físico onde o processo é desempenhado. Um exemplo é dado na Figura 5.5. a seguir.

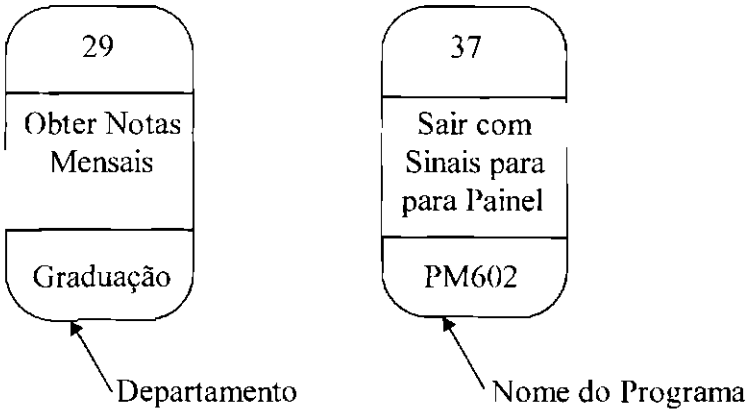


Fig. 5.5. Caixas de Processo com Referências Físicas.

Os Depósitos de Dados são repositórios de dados que são armazenados para serem usados em um ou mais processos. Eles são representados por um par de linhas paralelas horizontais, ligadas em uma das extremidades, largas somente o suficiente para comportar o nome. Para facilitar a referência, cada depósito deve ser identificado por um "D" e um número arbitrário, contidos numa caixa na extremidade esquerda. Na Figura 5.3 acima, "Programas" é um depósito de dados.

Fisicamente um depósito de dados pode ser um arquivo de fichas, uma tabela na memória, um arquivo em disco, um conjunto de arquivos, etc... Em geral eles representam uma ou mais tabelas de uma mesma entidade (para mais detalhes sobre tabelas e entidades ver o item **3.2.6 - Modelagem de Dados**, no Capítulo 3 desta dissertação).

Quando um processo armazena dados, a seta de fluxo de dados que liga o processo ao depósito de dados, aponta para o depósito de dados. Quando o acesso a um depósito de dados é feito de forma a realizar apenas leitura, a seta aponta para o processo, como mostra o fluxo de dados "Dados sobre Programas" da Figura 5.3.

Se for necessário especificar o argumento de pesquisa no depósito de dados, ele deve ser ilustrado do lado oposto da descrição do fluxo de dados; uma ponta de flecha indica que

o argumento de pesquisa é transmitido do processo ao depósito de dados, como na Figura 5.6 abaixo.

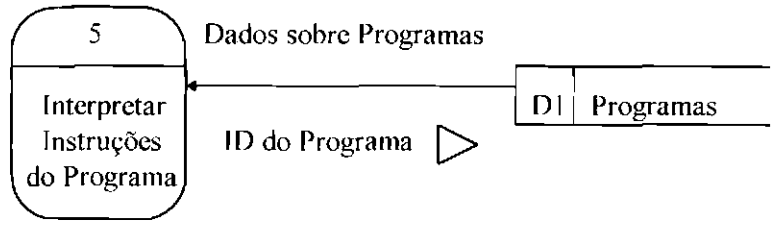


Fig. 5.6. Especificação do Argumento de Pesquisa.

O DFD que representa o sistema como um todo é chamado de DFD de nível de sistema. O diagrama da Figura 5.3 é o diagrama de fluxo de dados de nível de sistema para o Sistema Controlador de um Robô.

No exemplo da Figura 5.3, cada vez que um botão é pressionado a entrada é lida e convertida para o formato interno do sistema pelo processo "Ler Entradas do Painel". As entradas do painel são recebidas e validadas pelo processo "Verificar Entradas do Painel".

O processo "Classificar Entradas do Painel" passa as entradas para os processos apropriados, ou seja, para "Interpretar Instruções do Programa" ou para "Sair com Dados de Eixo". Além disso, ele envia para "Sair com Sinais para o Painel" os dados de controle das lâmpadas do painel.

Quando se altera a posição da chave seletora de programas, a nova indicação é passada para "Classificar Entradas do Painel", que substitui o registro do identificador do programa selecionado.

Quando o botão "Run" é pressionado, os dados para iniciar a execução de um programa chegam até "Interpretar Instruções do Programa". Este então começa a interpretar (executar) o programa selecionado. Ele executa as instruções aritméticas e lógicas

diretamente, e passa as instruções de movimento e as instruções de E/S de sensores para os devidos processos, onde eles receberão processamentos adicionais.

As instruções de movimento sofrem algumas transformações matemáticas pelo "Converter Instruções de Movimento " que passa os resultados sob a forma de "bloco de movimento", para o processo "Sair com Dados de Eixo".

"Sair com Dados de Eixo" converte os dados do bloco de movimento para a forma de "blocos de eixo", que poderão ser recebidos por "Controlar Eixos".

Quando o botão "Stop" é pressionado, o processo "Sair com Dados de Eixo" pára de alimentar o processo "Controlar Eixos" com blocos de eixo; quando o botão "Run" é pressionado ele volta a alimentá-lo.

Quando um bloco de eixo é terminado, "Controlar Eixos" comunica o evento ao processo "Receber Confirmação".

Quando o bloco de eixo corresponde ao fim do bloco de movimento, o evento de seu término é comunicado ao "Interpretar Instruções do Programa" pelo processo "Receber Confirmação".

"Interpretar Instruções do Programa" envia instruções de E/S de sensores para o processo "Gerenciar Instruções de E/S" e recebe dados de Sensores.

"Gerenciar Instruções de E/S" recebe dados de Sensores do processo "Ler Sensores" e passa dados de saída de sensores para o processo "Sair Dados para Sensores".

5.2.1 - Diretrizes para a Construção do Diagrama de Fluxo de Dados

No item precedente vimos que os DFDs são compostos por quatro componentes simples. Munidos dessas ferramentas pode-se começar a construir o DFD do sistema em questão.

Entretanto, existem algumas diretrizes adicionais necessárias para se utilizar o DFD com sucesso. Algumas dessas diretrizes auxiliam a não se construir DFDs incorretos (ou seja, incompletos ou logicamente inconsistentes) e outras destinam-se a ajudar a desenhar DFDs agradáveis à vista, e portanto com mais possibilidades de serem examinados com atenção pelo usuário. As diretrizes são as seguintes (Yourdon,1990) :

1) Deve-se escolher nomes significativos para processos, fluxos, depósitos de dados, e entidades externas. Deve-se rotular os processos de modo a identificar as funções que o sistema executa. Uma boa maneira de se fazer isso, é dar um nome para o processo na forma de verbo + objeto descritivo. No exemplo da Figura 5.3, "Ler Entradas do Painel", é um processo que tem o nome na forma de verbo + objeto descritivo.

Ao se nomear os processos deve-se evitar verbos com significado vago, como por exemplo processar, atualizar, revisar, fazer, manipular; pois isso é um indício de que não se está bem certo de qual função está sendo executada, ou que várias funções foram reunidas. Quando isso for inevitável, significa que o processo deve ser expandido, isto é, deve-se buscar maiores detalhes para serem representados.

Além disso, os nomes escolhidos para os processos (e também para os fluxos de dados, depósitos de dados, e entidades externas) devem provir de um vocabulário conhecido pelo usuário, ou seja, devem provir de um vocabulário oriundo do domínio do problema.

Deve-se evitar também, ao se escolher nomes, usar abreviações e acrônimos, e usar termos provenientes da programação como "sub-rotina" e "função".

Quando houver dúvida em relação ao nome do processo, o melhor é se pensar no nome como se ele fosse uma ordem para um "funcionário" pouco inteligente. Se ele não for ambíguo para o "funcionário", e se se conseguir visualizar o desempenho da função em uma simples ocorrência rotineira em 5-30 minutos, provavelmente tem-se um bom nome para o processo (Gane,1983).

Deve-se lembrar também que dois fluxos de dados não podem ter o mesmo nome mesmo que tenham destino e origem diferentes.

2) Numerar os processos. Não importa a maneira de enumerar os processos, da esquerda para a direita, de baixo para cima, ou de outra maneira qualquer, desde que se enumere os processos de uma maneira consistente. Isso facilita a sua identificação, já que um processo pode ser ainda mais detalhado em processos menos complexos, como será explicado adiante.

3) Evitar DFDs complexos demais. Deve-se construir DFDs que possam ser lidos e compreendidos, não somente por quem o construiu, mas também pelos usuários que são conhecedores do assunto correlacionado. Isso significa que o DFD deve ser prontamente compreendido, facilmente absorvido e agradável aos olhos.

Portanto, não se deve criar DFDs com muitos processos, fluxos de dados, depósitos de dados e entidades externas. Na maioria dos casos isso quer dizer que não se deve incluir mais que sete processos, e depósitos de dados, fluxos de dados, e entidades externas relacionados a eles.

4) Refazer o DFD tantas vezes quantas forem necessárias até obter uma boa estética. Deve-se refazer o DFD até que ele esteja **(1)** tecnicamente correto; **(2)** aceitável pelo usuário e **(3)** esteticamente agradável.

O que torna um DFD esteticamente agradável muitas vezes é uma questão de gosto e opinião, mas existem algumas medidas que podem ser tomadas para facilitar que isso aconteça.

Primeiramente deve-se manter o mesmo tamanho para os retângulos de processos, deve-se evitar fluxos de dados curvos, e deve-se evitar apresentar ao cliente diagramas feitos à mão.

Além disso, deve-se evitar o cruzamento de linhas de fluxos de dados. Ao se fazer isso pode ser necessário desenhar algumas entidades externas e alguns depósitos de dados mais de uma vez no diagrama.

Quando uma entidade externa for duplicada deve-se marcar suas ocorrências com uma linha inclinada no canto inferior à direita do símbolo, como mostra a Figura 5.7 abaixo.

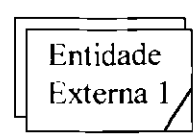


Fig. 5.7 - Símbolo de Duplicação para Entidades Externas.

Se for necessário duplicar outra entidade marca-se suas ocorrências com duas linhas inclinadas, e assim por diante, como mostra a Figura 5.8 a seguir.

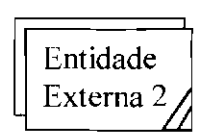


Fig. 5.8 - Duplicação Múltipla de Entidades Externas.

Se for necessário duplicar um depósito de dados marca-se suas ocorrências com uma linha vertical adicional à esquerda do símbolo, como mostra a Figura 5.9 a seguir.

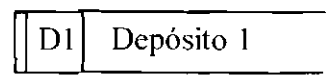


Fig. 5.9 - Símbolo de Duplicação para Depósitos de Dados.

Se for inevitável que um fluxo de dados tenha que cruzar com outro fluxo, deve-se utilizar a convenção do pequeno arco, conforme mostrado na Figura 5.10 a seguir.

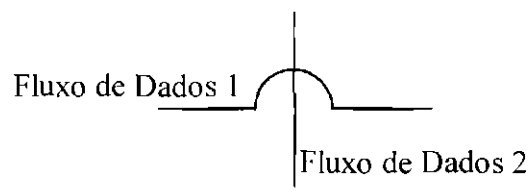


Fig. 5.10 - Convenção do Pequeno Arco.

Para facilitar o entendimento, pode-se usar também uma seta com duas pontas no lugar de duas setas, quando os fluxos de dados aparecem em pares, como mostra a Figura 5.11 a seguir.

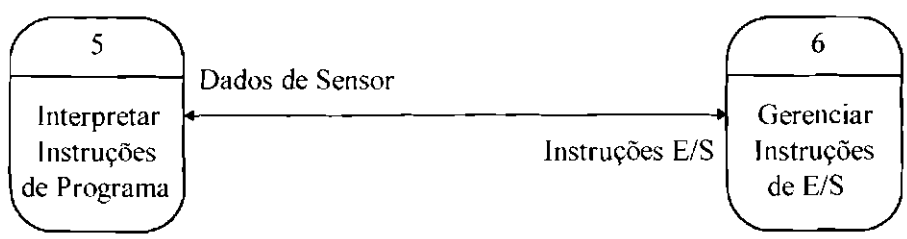


Fig. 5.11. Pares de Fluxos de Dados.

Quando um fluxo de dados seguir para dois ou mais lugares diferentes, para facilitar o entendimento deve-se desenhá-lo como mostra a Figura 5.12 a seguir.

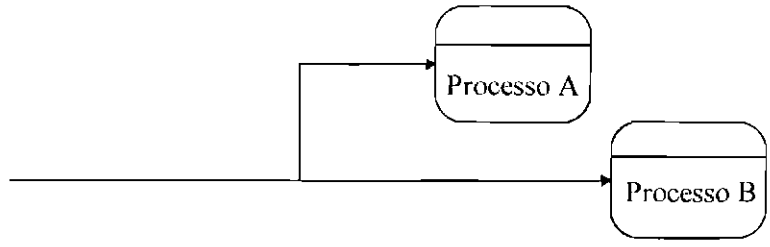


Fig. 5.12 - Fluxo de Dados Indo para Dois Lugares.

5. Certificar-se de que o DFD seja logicamente consistente. Deve-se evitar processos que têm entradas mas não têm saídas; e processos que têm saídas, mas que não têm entradas. Além disso, deve-se tomar cuidado com os processos e fluxos de dados sem rótulo, pois isso pode indicar que não se conseguiu encontrar um nome satisfatório. Deve-se tomar cuidado também, com os depósitos de dados de leitura-apenas ou de escrita-apenas; pois um depósito típico deve ter entradas e saídas.

Além disso, não se deve representar qualquer relacionamento existente entre entidades externas no DFD. Alguns relacionamentos desse tipo podem existir, de fato, mas, por definição, eles não fazem parte do sistema sendo modelado.

Outro ponto importante é lembrar que o DFD não deve representar fluxos de controle. Um controle não é processado e sim testado, ele comunica informações sobre uma parte dos dados. Um exemplo de fluxo de controle é dado na Figura 5.13 a seguir.

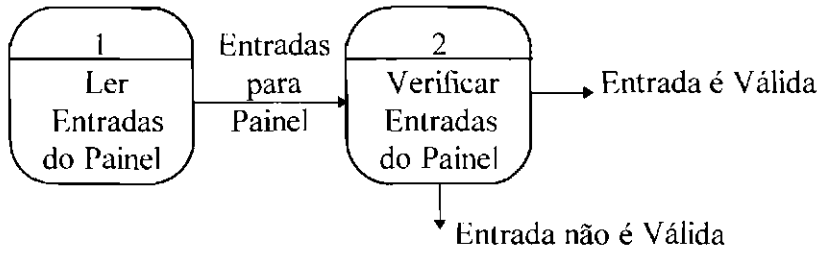


Fig. 5.13 - Fluxo de Controle.

"Entrada é Válida" e "Entrada não é Válida" não são fluxos de dados, e sim fluxos de controle, pois não existem dados que voltem à fonte, o que indica que eles são apenas sinais que informam o sistema sobre o estado de uma determinada entrada.

5.2.2 - Construindo o Diagrama de Fluxo de Dados

Apresentadas as convenções e diretrizes para se construir o diagrama de fluxo de dados, pode-se partir para sua construção propriamente dita. A seguir são apresentados os passos principais envolvidos no projeto de um diagrama de fluxo de dados :

1) Identificar as entidades externas envolvidas. Na verdade, isso já foi feito na especificação inicial do sistema quando construiu-se o diagrama de contexto, como explicado no item **5.1 - Definição do Sistema**. Portanto basta agora passá-las para a notação do DFD.

2) Identificar as entradas e as saídas do sistema. Na verdade isso também já foi feito na construção do diagrama de contexto. Essas entradas e saídas identificadas são nada mais nada menos que fluxos de dados que entram e saem do sistema.

3) Pegar uma folha de papel grande e começar desenhando no canto superior esquerdo a entidade externa que seja a principal geradora de entradas para o sistema. Desenhar os fluxos de dados de entrada relacionados a ela.

4) Identificar as consultas e os pedidos de informação que possam surgir, baseando-se inicialmente nas repostas do sistema aos eventos externos (àquelas que foram detalhadas na especificação inicial do sistema). Deve-se adiar o quanto for possível os detalhes de situações de erros, de falhas, de inicialização, e de término.

5) Identificar os fluxos de dados mais importantes, e para cada um deles perguntar :

- O que eu preciso para obtê-lo ?

- De onde vêm esses componentes ?
- Os fluxos de entrada podem ser transformados em qualquer um desses ?
- Quais processos serão necessários para efetuar essa transformação ?

6) Criar os processos que são necessários para transformar um fluxo ou um conjunto de fluxos em outro. Deve-se deixar inicialmente sem nome os primeiros processos que começarem a aparecer no diagrama, e verificar se se pode imaginar fluxos de dados internos a ele. Se existir algum, deve-se substituir o processo por dois, ou mais, e colocar os fluxos identificados com eles.

7) Criar os depósitos de dados necessários para interagir com os processos.

8) Fazer esboços à mão livre e melhorá-los. Seguir as regras de notação sem enumerar os processos até o esboço final. Aceitar o fato de que serão necessários, pelo menos três esboços do diagrama de fluxo de dados de nível de sistema.

9) Quando o primeiro esboço estiver pronto, verificar se todas as entradas, saídas e entidades externas identificadas no diagrama de contexto foram incluídas.

10) Produzir um segundo esboço mais claro usando uma régua com gabaritos ou uma Ferramenta CASE. Lembrar que o objetivo é um diagrama com processos únicos e um número mínimo de interseções de fluxos de dados.

11) Para minimizar cruzamentos, duplicar primeiro as entidades externas, se necessário. Em seguida, duplicar os depósitos de dados, se necessário. Permitir o cruzamento de fluxos de dados desde que não haja uma estrutura que reduza as interseções.

12) Rever o esboço com o usuário ou alguém que conheça a aplicação, como explicado no item **5.2.2.1 - Revisando O DFD**, que é apresentado a seguir.

13) Produzir a expansão de nível inferior para cada processo definido no segundo esboço, se necessário, de acordo com as convenções que serão especificadas no item **5.2.2.2 - Expandindo o DFD**, que é apresentado a seguir.

14) Se necessário, incorporar as modificações no diagrama de nível superior. As soluções dadas aos tratamentos de erros e exceções devem ficar nos diagramas de nível mais baixo possível.

15) Fazer o desenho definitivo do DFD enumerando os processos.

5.2.2.1 - Revisando o DFD

Deve-se fazer uma revisão de conteúdo do DFD com alguém que conheça a aplicação para verificar se ele está refletindo o que a empresa faz, ou o que se deseja que o sistema faça. Deve-se verificar portanto, se o DFD abrange corretamente o sistema de forma completa e sem redundância.

Deve-se fazer também uma revisão sintática verificando se :

- os processos tem nomes na forma de verbo + objeto descritivo;
- todos os processos e depósitos de dados possuem pelo menos um fluxo de entrada e um fluxo de saída (em caso negativo deve-se verificar se o domínio da aplicação justifica tal situação);
- todos os fluxos de dados possuem pelo menos uma ponta de seta;
- os fluxos sem nome no DFD podem ser explicados e
- todas as convenções e padrões de construção e expansão do DFD foram obedecidos.

A revisão do DFD deve ser realizada em todos os níveis de expansão.

5.2.2.2 - Expandindo o DFD

Conforme explicado no item **2.10.1.1 - Técnicas Estruturadas**, no Capítulo 2 desta dissertação, cada processo no nível superior do diagrama de fluxo de dados pode ser expandido para tornar-se um novo diagrama de fluxo de dados, que representa um nível de abstração com maiores detalhes (DFD pai e DFDs filhos).

Cada processo filho no nível inferior deve se relacionar com o processo pai. Para que isso seja possível, deve-se enumerar os processos filhos de maneira que seu número de identificação seja o valor decimal da caixa de processo pai; por exemplo, o processo pai 29, gerará os processos filhos 29.1, 29.2, 29.3, e assim por diante.

Se for necessário chegar a um terceiro nível, o mecanismo é o mesmo. Por exemplo, o processo pai 29.3 gerará os processos filhos 29.3.1, 29.3.2, e assim por diante, para todos os níveis de particionamento.

Cada processo em expansão deve ser representado desenhando-se diagramas de fluxo de dados de menor nível dentro da divisa que representa a caixa do processo de nível superior. Portanto, todos os fluxos de dados que entram e saem da caixa de processo de maior nível devem entrar e sair da divisa do diagrama de menor nível.

Os fluxos de dados ilustrados pela primeira vez no nível inferior, como condições de erro e exceções podem também sair da divisa, porém não precisam ser balanceados no diagrama de nível superior.

Um fluxo de dados no diagrama de nível superior também pode ser representado no diagrama de nível inferior por uma combinação de fluxos de dados mais detalhados, como ilustra o exemplo da Figura 5.15. Neste exemplo os fluxos de dados "Novo Programa" e "Ação"; que aparecem na expansão do processo "Classificar Entradas do

Painel", são representados no diagrama de nível superior pelo fluxo "Entradas do Painel Válidas".

Os depósitos de dados só devem ser mostrados dentro da divisa, como ilustrado na Figura 5.14 a seguir, quando são criados e processados apenas pelo processo expandido. Eles devem ser identificados com um D seguido do número do processo expandido, e do número do depósito dentro do diagrama de nível inferior. Por exemplo, um depósito de dados desenhado dentro das divisas de expansão processo número 4 deve ser identificado como D4/1; se houver outro depósito dentro dessa divisa ele deve ser identificado como D4/2, e assim por diante. Depósitos de dados externos ao processo expandido devem ser desenhados com uma metade fora e outra metade dentro da divisa, como ilustrado na Figura 5.14 a seguir.

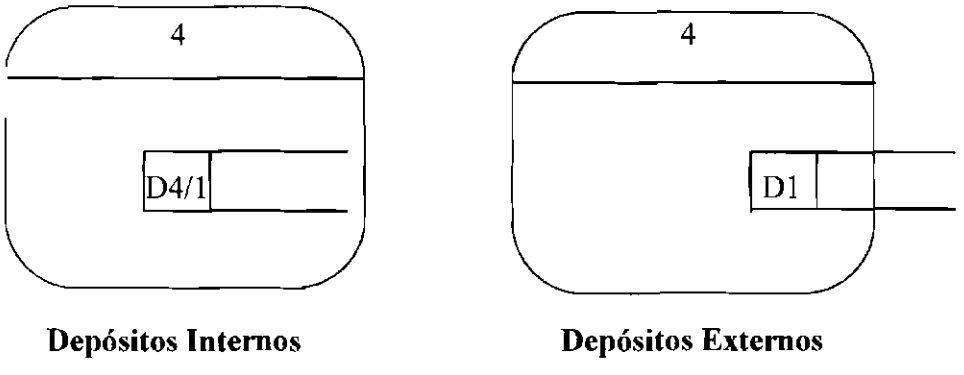


Fig. 5.14. Representação dos Depósitos de Dados na Expansão de Processos.

As entidades externas não devem ser ilustradas dentro das divisas de expansões, mesmo que não estejam envolvidas com qualquer outro processo além do processo expandido.

A Figura 5.15 a seguir ilustra a expansão do processo "Classificar Entradas do Painel".

Porém, até que nível continuar o particionamento do DFD ? Teoricamente pode-se expandir sucessivamente o DFD, porém DFDs de muitos níveis podem tornar o controle excessivamente complexo. Existem algumas regras que devem ser observadas para que o particionamento seja realizado de forma adequada, são elas :

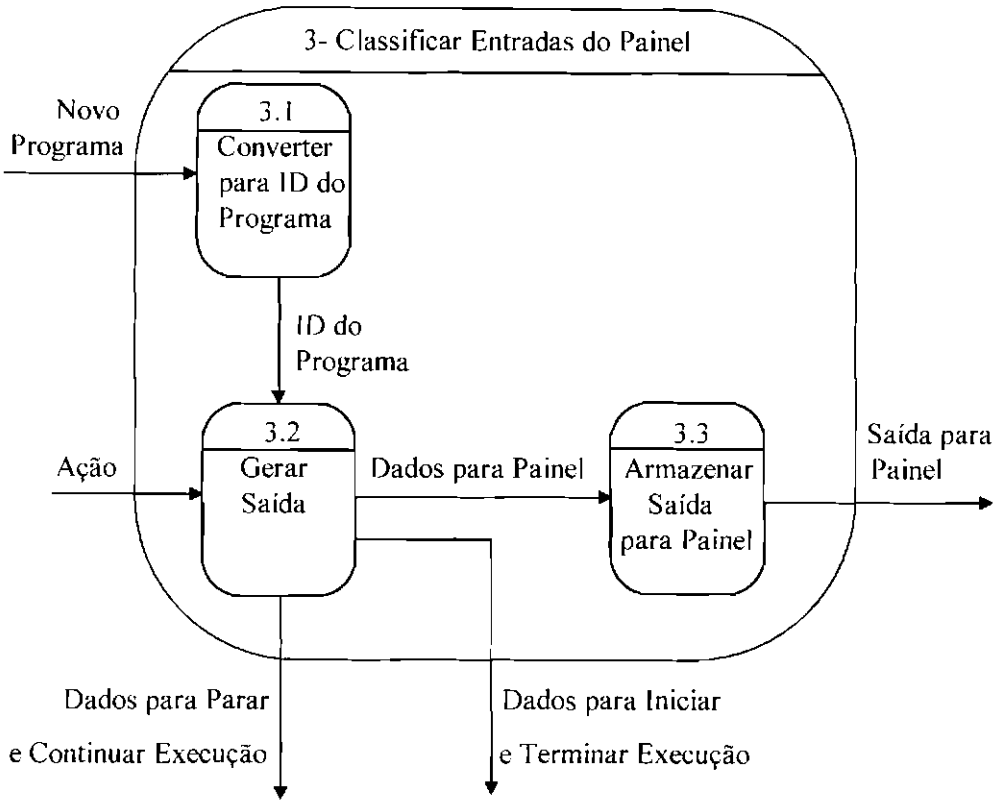


Fig. 5.15 - Expansão do DFD.

1) Em qualquer nível deve-se particionar tanto quanto possível, pois assim se terá menos diagramas para usar e manter. A partição entretanto, deverá produzir DFDs legíveis e perseguir o objetivo de criar interfaces que representem o modelo real.

2) O limite superior de 7 processos, e o limite inferior de 2 processos por DFD é uma regra teoricamente boa, mas não deve ser seguida com muito rigor. O mais importante é a facilidade e entendimento do diagrama. O número de processos nos DFDs de nível mais alto poderá ser maior do que nos de nível mais baixo.

3) A expansão deve parar quando se acredita que cada processo pode ser completamente descrito em uma mini-especificação, que consiste de uma descrição lógica do que o processo deve fazer. Na prática deve-se ficar entre um e quatro

níveis de expansão. Um exemplo de mini-especificação para um processo é dado abaixo :

- **Mini-Especificação para o Processo 3.1 - Converter para ID do Programa:** Ao chegar a indicação de um novo programa escolhido, o processo a converte no número do programa correspondente (ID do programa), e o envia para o processo 3.2- Gerar Saída.

4) Deve-se trabalhar de forma iterativa, isto é, deve-se tentar descrever o processo que se julgar primitivo. Se a descrição for difícil, deve-se retornar e fazer uma nova expansão. Um processo com mais de uma saída pode ser um indicativo de que talvez ele possa ser expandido.

5.2.3 - Quebrando o Sistema em Tarefas

Após a construção do diagrama de fluxo de dados deve-se decompor o sistema de médio porte em tarefas concorrentes, a partir dos resultados obtidos no DFD.

Uma tarefa pode ser definida com sendo um processo computacional ou manual, que pode se comunicar e/ou se sincronizar com outros processos.

Um processo computacional é um programa em execução no ambiente de um sistema de computação, ou seja, uma entidade capaz de causar o acontecimento de eventos. À qualquer instante considerado, o processo computacional está em um certo "estado", ou seja, o "estado" de um processo diz sobre o progresso de sua execução.

A entidade que faz o processo computacional avançar de um estado para o outro é o processador. O processador não é necessariamente um *hardware (CPU)*, ele pode ser também um outro processo computacional, por exemplo um *programa interpretador* em execução.

O estado de um processo computacional é caracterizado por um conjunto de informações que são :

1) o programa que está em execução;
2) a indicação da próxima instrução a ser executada;
3) os valores das variáveis e dos dados do programa e
4) os estados e a posição de todos os equipamentos de E/S em uso.

Essas informações são agrupadas num "Vetor de Estado"; portanto o processo computacional é caracterizado pelo par :

1) programa e
2) vetor de estado.

Considerando-se que a passagem de um estado para outro acontece com a execução de uma instrução do programa, pode-se dizer que um processo computacional passa por uma seqüência de estados ordenados no tempo.

Um processo computacional pode ser "ativado", "suspenso", "reativado", "abortado", e "terminado". Se um processo computacional é ativado mais de uma vez com os mesmos dados ele passará pelos mesmos estados, na mesma seqüência, e dará os mesmos resultados, independente de sua velocidade de execução. Quando um processo computacional é suspenso, o seu vetor de estado é armazenado de modo que ele possa ser reativado mais tarde do ponto onde parou.

Um processo manual é semelhante ao processo computacional, porém no lugar de um programa tem-se uma seqüência de ações; e o processador é uma pessoa ou um conjunto de pessoas (eventualmente usando alguma ferramenta de trabalho).

Um sistema é constituído por um conjunto de tarefas que podem concorrer por dados, e por recursos de execução (*CPU*, memória, equipamentos de E/S, pessoas, etc...). A concorrência ocorre porque as tarefas são executadas em velocidades diferentes, normalmente de maneira assíncrona.

Cada tarefa será composta por um ou mais processos do DFD, e pode ser considerada por si só como um sistema de pequeno porte. Portanto, após a divisão do sistema em tarefas, cada uma delas poderá seguir seu desenvolvimento como especificado no **Capítulo 3 - Sistemas de Pequeno Porte**, desta dissertação. A Figura 5.16 a seguir ilustra o particionamneto do sistema em tarefas.

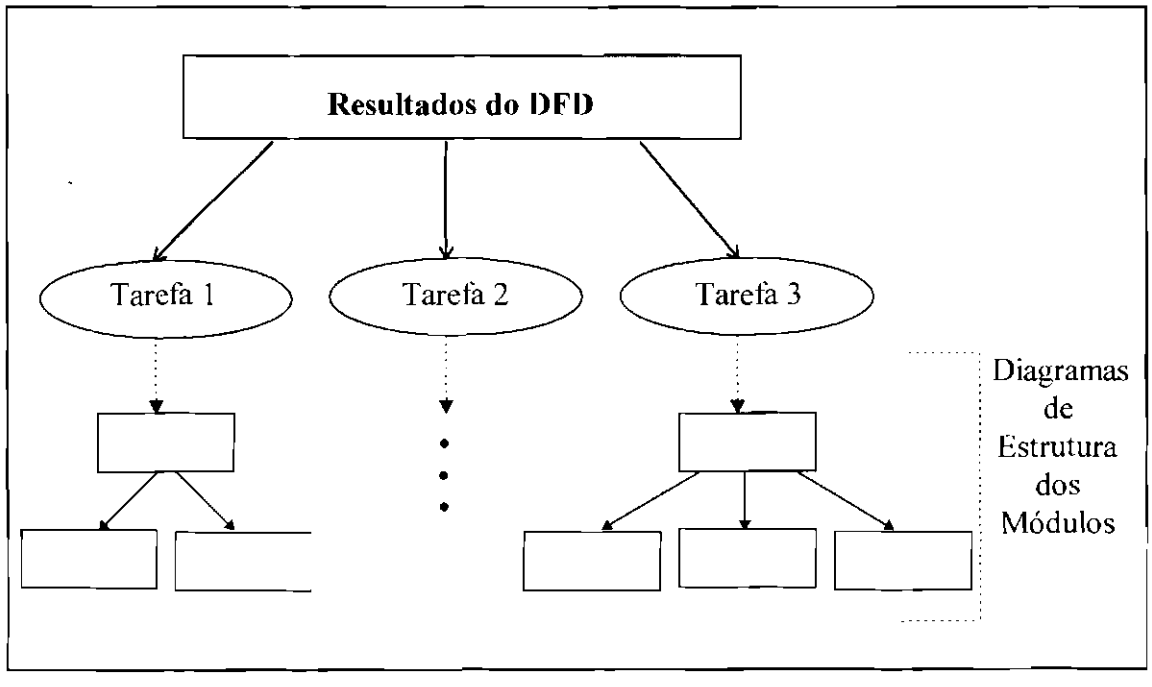


Fig. 5.16 - Particionamento do Sistema em Tarefas.

Neste estágio deve-se definir também as tarefas que serão executadas manualmente, e aquelas que serão executadas pelo sistema de computação. As tarefas que serão executadas pelo sistema podem ser alocadas a um único processador ou serem distribuídas entre dois ou mais processadores. Essa decisão de projeto depende de vários fatores, como por exemplo o desempenho que o sistema deverá ter.

5.2.3.1 - Critérios para a Decomposição do Sistema em Tarefas

Para decompor o sistema em tarefas deve-se primeiramente analisar os processos DFD para identificar quais deles podem ser naturalmente executados seqüencialmente, e quais deles devem ser executados concorrentemente.

Os critérios para decidir se um processo DFD pode se tornar uma tarefa, ou se ele deve ser agrupado com outros processos DFD para juntos formarem uma tarefa, são apresentados a seguir :

1) Processo DFD Dependente de E/S. Necessita se tornar uma tarefa separada pois freqüentemente é obrigado a executar suas instruções em uma velocidade ditada pela velocidade do equipamento de E/S com o qual ele interage.

2) Processo DFD com Funções Críticas no Tempo. Necessita se tornar uma tarefa separada pois uma função crítica no tempo necessita ser executada com uma prioridade alta.

3) Processos DFD que Exigem Requisitos Computacionais. Um processo, ou um conjunto de processos que requer muito processamento, pode ser executado como uma tarefa de baixa prioridade consumindo ciclos ociosos da *CPU*.

4) Processos DFD com Coesão Funcional. Processos DFD que realizam um conjunto de funções fortemente relacionadas devem ser agrupados em uma tarefa, pois o tráfego de dados entre esses funções pode ser intenso se elas forem separadas em tarefas independentes.

5) Processos DFD com Coesão Temporal. Certos processos DFD que realizam funções que são executadas em conjunto, ou seja, sempre que uma for executada as

outras também são, devem ser agrupados em uma tarefa. Desse modo, elas são executadas toda vez que a tarefa for ativada.

6) Processo DFD que têm Execução Periódica. Um processo DFD que precisa ser executado periodicamente deve ser estruturado como uma tarefa separada que é ativada em intervalos regulares.

O DFD do Sistema Controlador de um Robô decomposto em tarefas é apresentado na Figura 5.17 adiante.

Na Figura 5.17 os processos DFD "Ler Entradas do Painel", "Sair Dados para Sensores", "Sair com Sinais para o Painel", e "Controlar Eixos"; foram considerados como tarefas separadas de acordo com o critério **1. Processo DFD Dependente de E/S**. O processo DFD "Controlar Eixos" deve ser executado de forma muito rápida para acompanhar a velocidade de movimento dos eixos, e portanto, pode-se também aplicar neste caso o critério **2. Processo DFD com Funções Críticas no Tempo**.

O processo DFD "Ler Sensores" adquire periodicamente os dados dos sensores, e pelo critério **6. Processo DFD que têm Execução Periódica**, foi considerado em uma tarefa separada que é ativada periodicamente.

Os processos DFD "Verificar Entradas do Painel " e "Classificar Entradas do Painel", foram agrupados em uma tarefa de acordo com o critério **5. Processos DFD com Coesão Temporal**, considerando que a entrada no painel de controle é processada imediatamente após a validação.

Os processos DFD "Interpretar Instruções do Programa", "Converter Instruções de Movimento" e "Gerenciar Instruções de E/S", executam um conjunto de funções fortemente relacionadas, e pelo critério **4. Processos DFD com Coesão Funcional** eles foram agrupados em uma única tarefa.

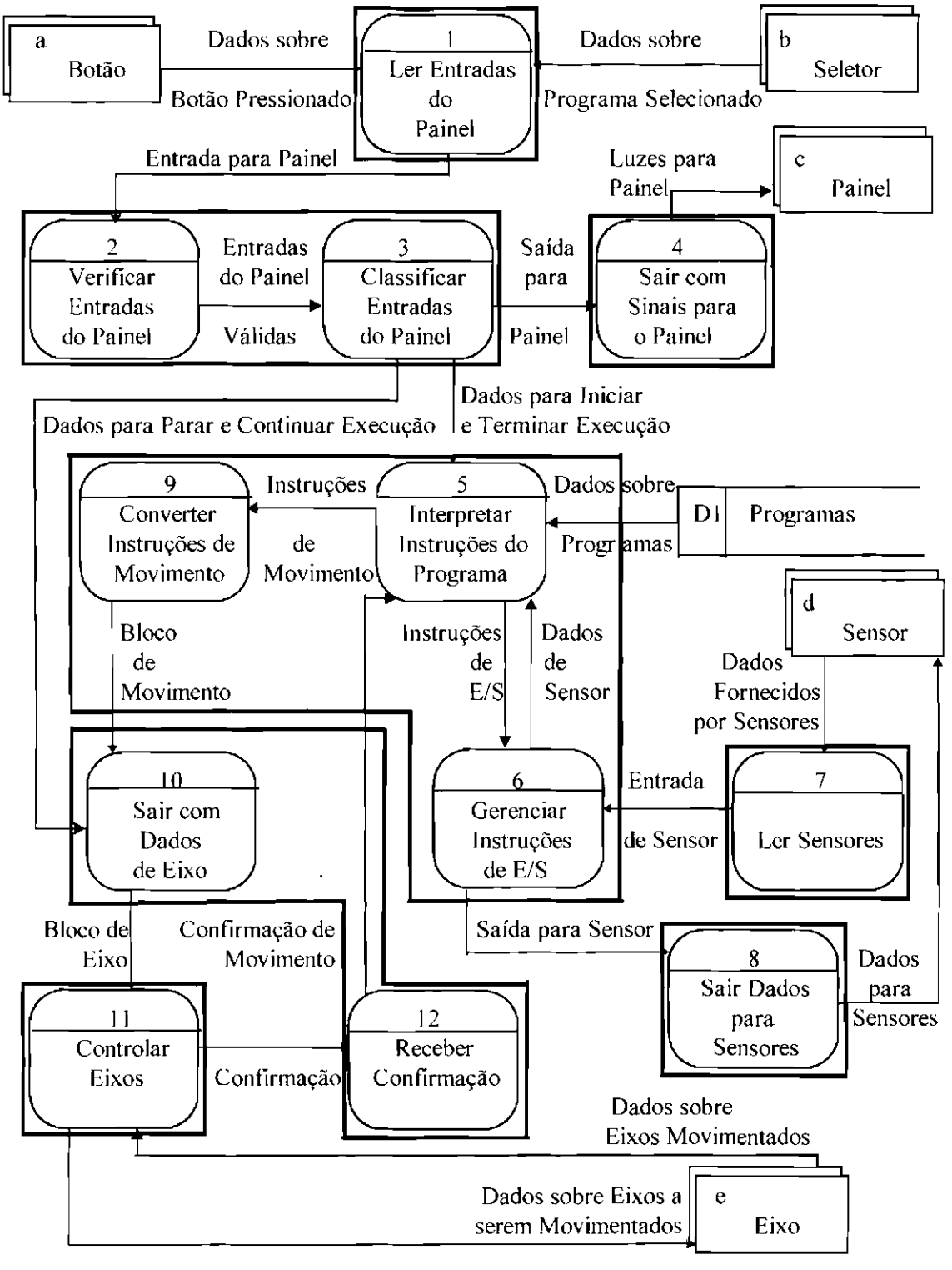


Fig. 5.17 - DFD do Sistema Controlador de um Robô Decomposto em Tarefas.

Cada vez que o processo DFD "Sair com Dados de Eixo" envia um bloco de dados de eixo para o processo DFD "Controlar Eixos", o processo DFD "Receber Confirmação" tem que esperar por uma confirmação antes que o processo DFD "Sair com Dados de Eixo", possa enviar o próximo bloco. De acordo com o critério **5. Processos DFD com Coesão Temporal**, os processos DFD "Sair com Dados de Eixo" e "Receber Confirmação", foram agrupados em uma tarefa.

Após decompor o DFD do sistema em tarefas deve-se nomear cada tarefa e construir o Diagrama de Tarefas do sistema, como mostra a Figura 5.18.

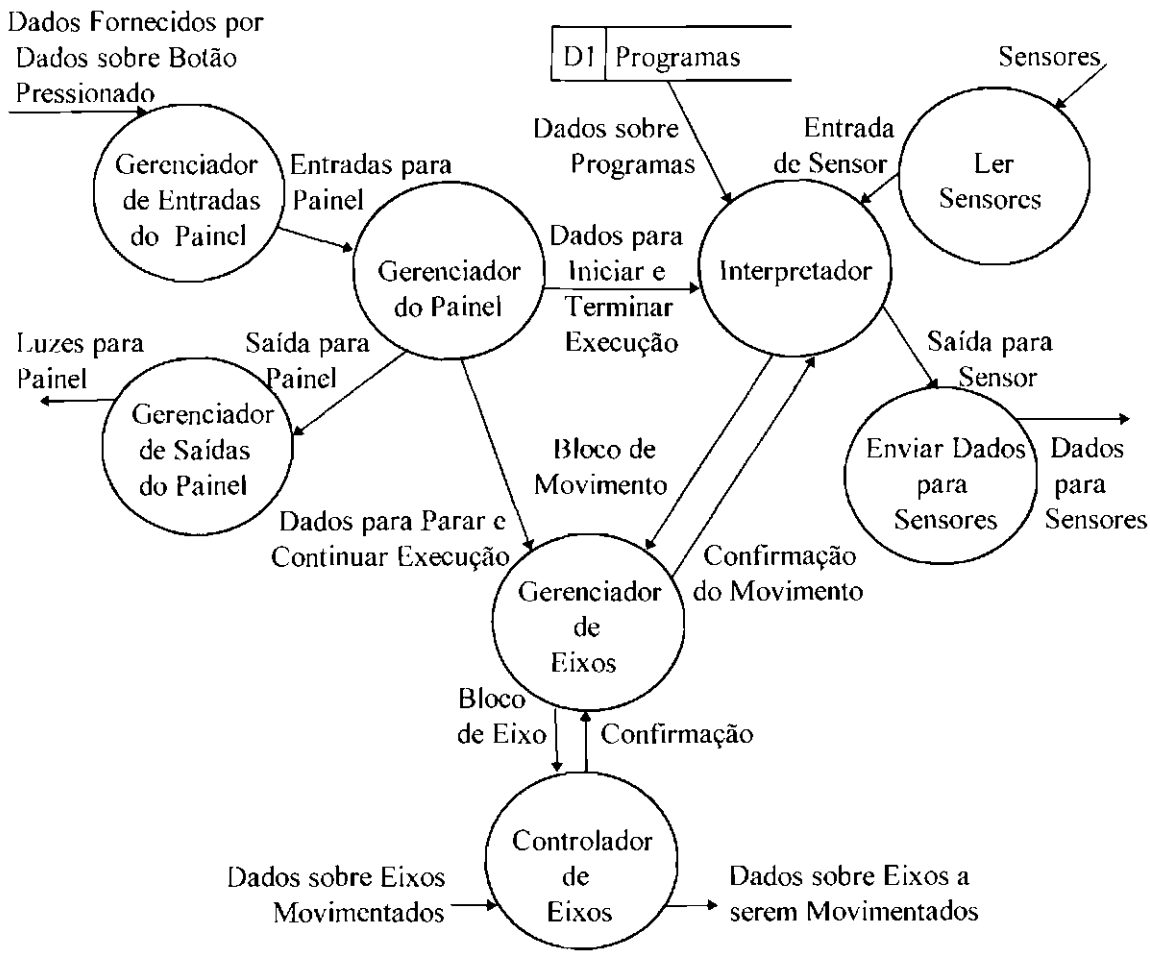


Fig. 5.18 - Diagrama de Tarefas do Sistema Controlador de um Robô.

Como se pode observar na Figura 5.18, os fluxos de dados de entrada e de saída de cada tarefa devem ser mantidos como no DFD. Eles representam as interfaces que devem existir entre uma tarefa e outra, isto é, são informações que devem fluir entre uma tarefa e outra para que o sistema funcione de maneira adequada. Essas informações podem ser mensagens ou dados.

Uma mensagem é um conjunto de informações, geralmente constituído de duas partes. Uma parte contém informações de controle e informações complementares, como por exemplo, a identificação do destinatário. A outra parte contém as informações propriamente ditas, que uma tarefa quer passar para outra tarefa.

A comunicação por dados é uma forma de comunicação entre tarefas onde uma, ou mais estrutura(s) de dados é(são) acessada(s) por tarefas concorrentemente. Os dados compartilhados são acessados pelas tarefas para leitura ou para leitura/escrita.

Um outro tipo de comunicação entre as tarefas é a sincronização. Ela é uma forma de interferência entre tarefas, que se caracteriza pela inexistência de transferência de dados de uma tarefa para outra. A sincronização entre tarefas pode ser de dois tipos : exclusão mútua e sinalização.

A exclusão mútua acontece quando a sincronização é necessária para que um dado recurso seja acessado simultaneamente por duas ou mais tarefas. A sinalização acontece quando uma tarefa sinaliza à outra tarefa da ocorrência de um certo evento. A tarefa que recebe o sinal altera o seu comportamento ou seu estado.

Na fase de implementação do sistema cada tarefa será considerada como um programa do sistema. Cada programa deverá interagir com os outros programas para que o sistema possa ter sua função executada sob a forma de um conjunto de processos computacionais. Não se deve preocupar por enquanto com a maneira pela qual essas

interfaces serão implementadas, deve-se apenas saber as informações que estão envolvidas nesse processo.

5.2.4 - Definição do Plano de Implementação e Testes

Após quebrar o sistema em tarefas deve-se definir um plano de implementação e testes para o sistema (para mais detalhes sobre plano de implementação e testes, ver o item **3.2.7 - Definição do Plano de Implementação e Testes**, no Capítulo 3 desta dissertação).

A seguir apresentaremos algumas diretrizes para se construir um plano de implementação e testes para um sistema de médio porte desenvolvido usando o paradigma clássico. Essas diretrizes abrangem a adoção de uma estratégia para implementação e testes, e o projeto de casos de teste.

Nos sistemas de médio porte pode ser interessante também, adotar ferramentas automatizadas que auxiliem na realização dos testes.

Essas ferramentas têm inúmeras facilidades, que incluem : **(1)** um ambiente visual que permite o planejamento dos testes; **(2)** diagramas de caminhos de testes e do número de testes requeridos, permitindo a identificação visual das partes do software mais complexas onde os esforços de testes devem ser focados com maior ênfase; **(3)** resultados dos testes mostrados tanto graficamente como na forma de relatórios, o que permite identificar os caminhos de testes já testados e os que ainda não foram testados e **(4)** vários níveis de testes de código, inclusive testes a nível de unidade e a nível de integração; entre outras.

Alguns exemplos de ferramentas que auxiliam na elaboração e/ou execução dos testes são : Ms Visual Test, Visual Testing Toolset, CodyVB5, Digital Debbuger, etc...

5.2.4.1 - Adoção de uma Estratégia para Implementação e Testes

A atividade de teste do sistema de médio porte deve focar, inicialmente, cada tarefa individualmente garantindo que ela funcione adequadamente como uma unidade; e à medida que forem sendo implementadas e testadas, elas devem ser integradas para formarem um pacote de software.

Finalizada esta etapa, deve-se realizar a integração do software com os outros elementos do sistema (por exemplo, *hardware*, pessoas, bancos de dados, outros sistemas).

A definição de projetos de casos de teste, e de uma estratégia para testar e implementar cada tarefa individualmente será abordada mais adiante.

Por enquanto, deve-se apenas definir os casos de testes de integração sem se preocupar com os testes individuais de cada tarefa (para mais detalhes sobre casos de teste ver o item **3.2.7.2 - Projeto de Casos de Teste**, no Capítulo 3 desta dissertação).

5.2.4.2 - Projeto de Casos de Teste de Integração

Como já dito, realizados os testes individuais de cada tarefa, elas precisam ser integradas às outras tarefas do sistema. Portanto, devem ser projetados casos de teste de integração que testem as interfaces entre as tarefas à medida que elas vão sendo integradas.

Durante a integração das tarefas pode ser necessário testar o software como um todo antes de ele estar completo, para certificar-se de que as tarefas já integradas até o momento estão operando de maneira correta, ou seja, para certificar-se que a comunicação e o sincronismo entre elas está acontecendo como previsto.

Portanto, deve-se projetar também casos de teste de integração que testem um conjunto de tarefas que formam uma função maior no software (funções que extrapolam a fronteira de uma única tarefa e precisam de mais de uma tarefa para serem realizadas).

Para identificar as tarefas que devem ser testadas em conjunto deve-se focar na especificação inicial do sistema e procurar pelas funções que o sistema deve realizar.

Finalmente, deve-se projetar casos de teste fundamentados na interação que o software realizará com os outros elementos que existem à sua volta (*hardware*, pessoas, banco de dados, outros sistemas), com os quais ele precisa interagir. Para tal, deve-se projetar casos de teste que verifiquem todas as informações que chegam de elementos que estão fora do escopo do software, e projetar casos de teste que verifiquem todas as informações que saem do software para outros elementos.

No Sistema Controlador de um Robô primeiramente deve-se projetar casos de teste que verifiquem as interfaces de cada tarefa integrada ao sistema, e à medida que as tarefas integradas forem formando funções maiores no software, o sistema como um todo deve ser testado. Por exemplo, suponhamos que as tarefas "Gerenciador de Painel", "Gerenciador de Entradas do Painel", e "Gerenciador de Saídas do Painel" sejam integradas ao sistema. Antes de integrar outra tarefa deve-se testar o sistema como um todo para verificar, por exemplo, se a partir das entradas obtidas do painel, as luzes do painel estão acendendo e apagando da maneira desejada. Finalmente, após integrar todas as tarefas, o sistema deverá ser testado como um todo novamente.

Em seguida, deve-se projetar casos de teste que verifiquem se o sistema de software está cumprindo corretamente as suas funções dentro de um contexto maior, isto é, se ele está operando de maneira correta quando colocado em conjunto com os elementos que interagem com ele; ou seja, com as entidades externas com que ele se relaciona. Para isso, deve-se projetar casos de teste que verifiquem se ele está cumprindo em conjunto

com "Sensor", "Eixo", "Botão", "Seletor", e "Painel" os requisitos de função e desempenho (*performance*) estabelecidos na definição do sistema.

5.2.4.3 - Construção do Plano de Testes

Todos os casos de teste identificados devem ser integrados ao plano de testes do sistema. O plano de testes deve ser um documento organizado que descreve as atividades de testes que serão realizadas.

Segundo Yourdon (1990), um plano de teste típico contém, para cada caso de teste identificado, as seguintes informações :

- **Objetivo do teste :** qual é o objetivo do teste e que parte do sistema será testada.

- **Localização e cronograma do teste :** onde e quando o teste será realizado.

- **Descrições de testes :** uma descrição das entradas que serão introduzidas no sistema e das saídas e resultados esperados.

- **Procedimentos de testes :** uma descrição de como os dados de testes devem ser preparados e submetidos ao sistema, como os resultados de saída devem ser colhidos, como os resultados dos testes devem ser analisados, e de quaisquer outros procedimentos operacionais que devam ser observados.

Além disso, deve ser incluído no plano de testes a data que um determinado caso de teste foi realizado, para que se possa ter controle de quais testes já foram realizados.

A Figura 5.19 e a Figura 5.20 mostram exemplos de como os casos de teste podem ser descritos no plano de testes. Esses exemplos referem-se ao Sistema Controlador de um Robô.

É necessário salientar que apenas dois casos de teste foram descritos a título de ilustração, de modo que, um plano de testes real para o Sistema Controlador de um Robô deveria conter todos os casos de teste possíveis de serem realizados.

Tipo do Teste : Teste de Integração do Sistema
Data da Realização do Teste : 03/03/98
Objetivo do Teste : Verificar se o Sistema Controlador de um Robô está recebendo e enviando corretamente os dados de/para sensor, que é fornecido pela entidade externa "Sensor".
Cronograma do Teste : Deve ser realizado após os testes de Integração do Software.
Descrição do Teste **Entradas :** Dados de Sensor **Saídas :** Dados para Sensor
Resultados Esperados : O sistema deve estar lendo periodicamente os dados da entidade externa sensor, e deve estar enviando dados para ela quando um evento de saída de dados para sensor acontece.
Observações : Os dados recebidos devem estar sendo convertidos para o formato interno do sistema, e os dados enviados devem estar sendo convertidos para o formato do sensor.

Fig. 5.19 - Descrição para Caso de Teste de Integração do Sistema.

5.2.5 - Considerações Adicionais para a Construção de Sistemas de Médio Porte

Como dito anteriormente, cada tarefa pode ser considerada como um sistema de pequeno porte. No **Capítulo 3 - Sistemas de Pequeno Porte** desta dissertação, foi abordado o desenvolvimento da tarefa "Gerenciador do Painel" do Sistema Controlador de um Robô. As outras tarefas do Sistema Controlador de um Robô, devem ser desenvolvidas usando-se as mesmas diretrizes.

Tipo do Teste : Teste de Integração do Software
Data da Realização do Teste : 19/01/98
Objetivo do Teste : Verificar se ao se receber uma entrada do painel, as luzes corretas estão sendo acesas e apagadas.
Cronograma do Teste : Deve ser realizado após as tarefas "Gerenciador de Entradas do Painel", "Gerenciador de Painel", e "Gerenciador de Saídas do Painel" terem sido integradas ao sistema.
Descrição do Teste **Entradas :** Entrada do Painel **Saídas :** Luzes para Painel
Resultados Esperados : Quando o botão "Power" é pressionado deve-se acender a luz do mesmo e mantê-la até que o sistema seja desligado. Deve-se acender também uma luz indicando que o sistema está no estado "Manual". Quando o sistema é desligado todas as luzes do painel devem ser apagadas. Quando o botão "Run" ou "Stop" é pressionado deve-se acender a luz do mesmo e apagar as outras luzes, com exceção da luz do botão "Power". Quando o botão "End" é pressionado deve-se acender a luz indicando que o sistema está no estado "Manual". Todas as outras luzes devem ser apagadas, com exceção da luz do botão "Power".
Observações : A chave seletora de programas não têm luzes.

Fig. 5.20 - Descrição para Caso de Teste de Integração do Software.

Portanto, após quebrar o sistema de médio porte em tarefas deve-se abordar cada tarefa como um sistema de pequeno porte. Como as tarefas devem ser integradas para compor o sistema de médio porte, algumas diretrizes que não foram consideradas no desenvolvimento de sistemas de pequeno porte, devem ser consideradas durante o desenvolvimento de sistemas de médio porte. A seguir é apresentado um roteiro para o desenvolvimento de cada tarefa ou subsistema do sistema de médio porte.

Primeiramente, cada subsistema deverá ser especificado como mostrado no item **3.1 - Definição do Sistema** no Capítulo 3 desta dissertação. Então, um sistema de médio porte deve ter, além de sua definição como um todo, as definições de seus subsistemas.

Só após realizada a definição para o subsistema é que se deve, se necessário, expandir os processos DFD desse subsistema, pois somente com uma definição mais detalhada tem-se mais informações para realizar a expansão.

Após expandir os processos DFD do subsistema deve-se ler o item **3.2 - Construção do Sistema de Pequeno Porte Usando o Paradigma Clássico**, do Capítulo 3 desta dissertação.

O primeiro passo na construção do diagrama de estrutura dos módulos a partir do DFD é procurar pelo centro de transformação do DFD. O centro de transformação é, na verdade, a parte do DFD que contém as funções do sistema, e é independente da implementação particular das entradas e saídas. O propósito dessa análise de transformação é derivar um diagrama de estrutura a partir do DFD.

Uma boa maneira para identificar o centro do DFD é através da eliminação de seus ramos aferentes e eferentes. Pode-se fazer isso através de 3 passos : primeiro deve-se traçar cada curso aferente partindo do lado externo do DFD em direção ao centro, e marcar a etapa na qual a entrada foi completamente refinada, mas ainda não foi usada para processamento. Segundo, deve-se traçar o curso eferente do lado externo do DFD em direção ao centro, e marcar a etapa na qual a saída foi produzida, mas ainda não está formatada. Terceiro, deve-se juntar todas as marcas num polígono fechado. O centro de transformação do DFD do Subsistema Gerenciador de Painel é mostrado na Figura 5.21.

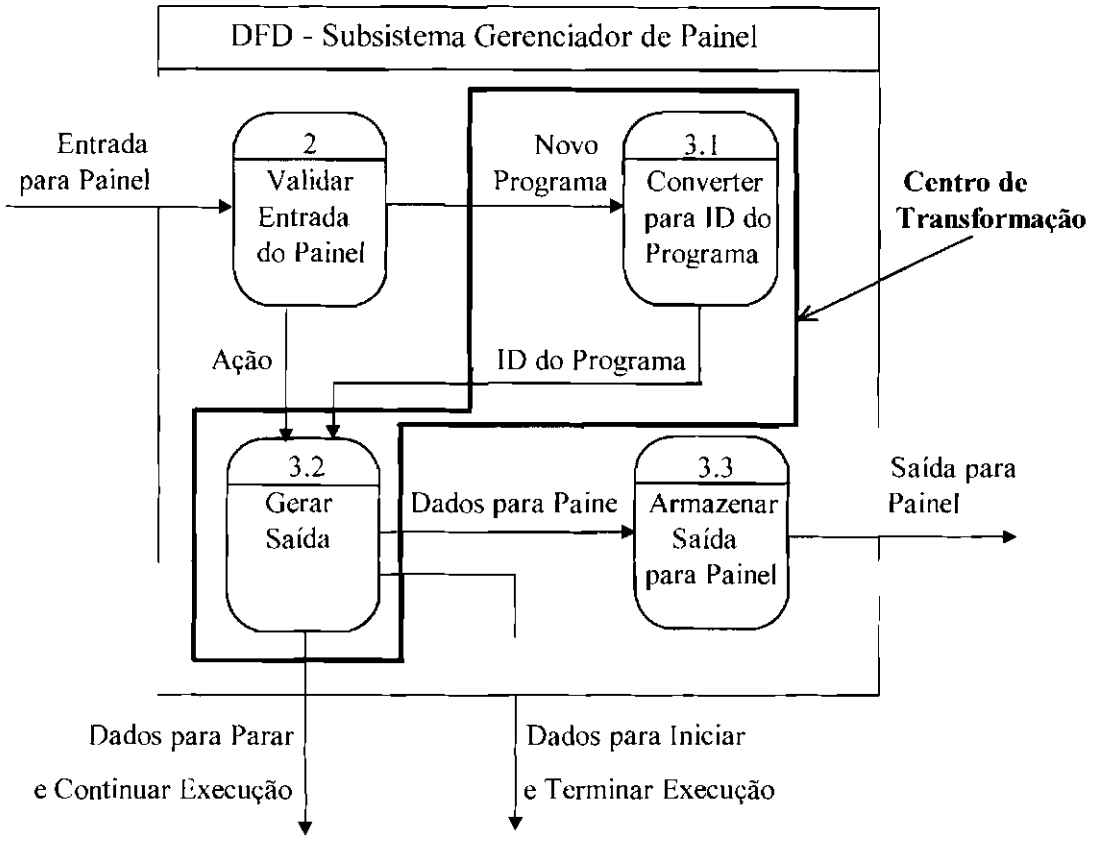


Fig. 5.21 - Centro de Transformação do DFD.

Cada processo do DFD a princípio se tornará um módulo do diagrama de estrutura. Porém, como posicioná-los hierarquicamente se no DFD não existem superiores e subordinados ?

O primeiro passo é criar um módulo para ser o módulo superior do diagrama de estrutura. Esse módulo deve abranger o subsistema como um todo, ou seja, o seu nome deve representar todas as funções que seus módulos subordinados realizam.

Após criar o módulo superior deve-se posicioná-lo ao centro e na parte superior do diagrama, pois todos os demais módulos serão subordinados a ele. Deve-se posicionar os processos da parte aferente do DFD, agora redesenhados como módulos retangulares, do lado esquerdo do diagrama; subordinando-os ao módulo superior. Os processos da parte eferente do DFD, agora também redesenhados como módulos retangulares, devem

ser posicionados no lado direito do diagrama, e da mesma maneira devem ser subordinados ao módulo superior. Os processos do centro de transformação do DFD também devem ser redesenhados como módulos retangulares, e serem posicionados ao centro do diagrama; também subordinados ao módulo superior. Uma primeira versão do diagrama de estrutura dos módulos para o subsistema Gerenciador de Painel é apresentada na Figura 5.22 a seguir.

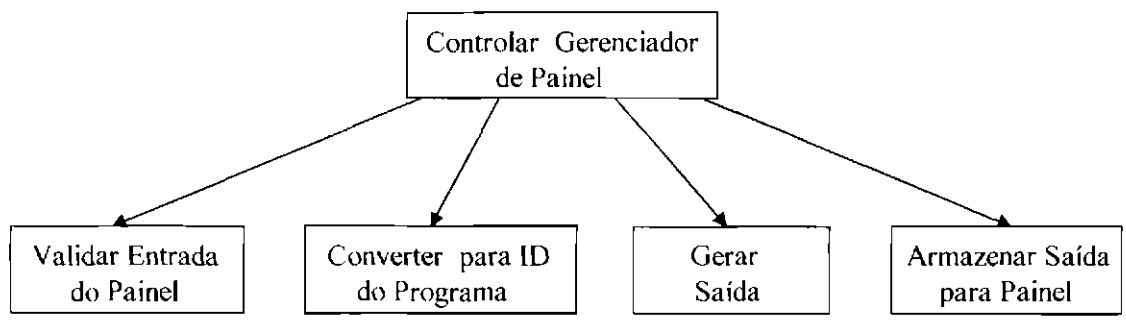


Fig. 5.22 - Primeira Versão do Diagrama de Estrutura dos Módulos.

Numa segunda etapa, deve-se examinar esses módulos de acordo com os critérios apresentados no item **3.2.1 - Projetando Módulos Coesos**, **3.2.2 - Diretrizes para a Construção do Diagrama de Estrutura dos Módulos**, e **3.2.3 - Acoplamento entre Módulos**, do Capítulo 3 desta dissertação; pois pode ser necessário detalhar certos módulos, o que pode resultar na criação ou eliminação de alguns módulos.

Pode ser necessário também durante esse processo mudar o nome de certos módulos para que eles realmente representem a função que o módulo realiza; pois o nome do módulo deve representar o conjunto de atividades de seus subordinados, enquanto que o nome do processo DFD descreve apenas a sua própria atividade.

Ao acrescentar os nós de comunicação entre os módulos deve-se lembrar que as setas que ligam os módulos a seus subordinados no diagrama de estrutura não correspondem

necessariamente às setas do DFD, pois a direção do fluxo de dados não tem o mesmo significado que a direção de uma chamada.

No Subsistema "Gerenciador de Painel", o módulo "Gerar Saída" foi detalhado nos módulos : "Ligar Sistema", "Iniciar Programa", "Terminar Programa", "Interromper Programa", "Reativar Programa" e "Desligar Sistema"; o módulo "Converter para ID do Programa" era muito simples, e por isso foi eliminado e seu processamento incorporado ao módulo superior do diagrama, como mostra a Figura 3.2 do Capítulo 3 desta dissertação.

A versão final do diagrama de estrutura dos módulos para o Subsistema Gerenciador de Painel é apresentada na Figura 3.6, do Capítulo 3 desta dissertação.

Finalizada a construção do diagrama de estrutura dos módulos do subsistema, deve-se especificar os módulos do diagrama conforme explicado nos itens **3.2.4 - Especificação dos Módulos** e **3.2.5 - Programação Estruturada**, no Capítulo 3 desta dissertação.

Em seguida, deve-se fazer a modelagem de dados do sistema. Como faze-la está explanado nos itens **3.2.6 - Modelagem de Dados** à **3.2.6.3 - Implementação dos Dados**, no Capítulo 3 desta dissertação. Deve-se ressaltar que os depósitos de dados do DFD do subsistema, na modelagem de dados serão nada mais nada menos que as entidades do modelo de entidade-relacionamento.

Em seguida, deve-se continuar o desenvolvimento dos subsistemas do item **3.2.7 - Definição do Plano de Implementação e Testes** até o item **3.2.8.1 - Documentação Externa do Módulo** (inclusive).

Nesse item, é mostrado como se deve fazer a documentação externa para os módulos. Os campos "Auditor", "Autor" e "Modificações" do exemplo dado estão em branco

porque num sistema de pequeno porte não faz sentido preenchê-los, já que existe apenas um programador.

Nos sistemas de médio porte porém, ao dividirmos o sistema em subsistemas, mesmo eles sendo consideradas por si só como sistemas de pequeno porte deve-se preencher estes itens, pois deve-se lembrar que os subsistemas serão integrados para compor o sistema de médio porte, e como a equipe contém mais de uma pessoa, deve-se ter um controle da qualidade dos documentos produzidos e das versões do sistema.

Preenchendo esses campos fica fácil saber quem programou o módulo, quem foi o auditor (a pessoa responsável por verificar se o módulo estava de acordo com os critérios de qualidade, e por controlar as versões do software), e fica fácil controlar as versões do sistema, pois o campo "Modificações" deve incluir as datas das modificações e o motivo que levou a equipe a realizar tais alterações.

Finalmente, deve-se continuar o desenvolvimento dos subsistemas seguindo as diretrizes do item **3.2.8.2 - Codificação de um Módulo** no Capítulo 3 desta dissertação, até o final desse Capítulo.

CAPÍTULO 6

CONSTRUÇÃO DO SISTEMA DE MÉDIO PORTE USANDO O PARADIGMA DA ORIENTAÇÃO A OBJETOS

Ao longo deste Capítulo forneceremos diretrizes para se construir um Sistema de Médio Porte usando o paradigma da orientação a objetos (para mais detalhes sobre o paradigma ver o item **2.10.2 - Paradigma da Orientação a Objetos** no **Capítulo 2 - Software e Engenharia de Software**, desta dissertação). Estaremos neste Capítulo, portanto, traçando uma rota que pode ser seguida por aqueles que queiram construir um sistema desse tipo.

Como já dito no **Capítulo 4 - Construção do Sistema de Pequeno Porte Usando o Paradigma da Orientação a Objetos** desta dissertação, a primeira etapa do desenvolvimento de um sistema de software usando o paradigma da orientação a objetos corresponde à construção do modelo de objetos.

Pode-se identificar cinco atividades principais na construção do modelo de objetos de um sistema de médio porte :

1) identificar os propósitos e as características do sistema;
2) estabelecer um modelo de componentes para o sistema;
3) selecionar e estabelecer responsabilidades para as classes&objetos do sistema;
4) selecionar assuntos e
5) exercitar a dinâmica do sistema.

A primeira atividade é nada mais nada menos que fazer a especificação inicial do sistema. Como realizar esta atividade está explanado no item **5.1 - Definição do Sistema** no **Capítulo 5 - Sistemas de Médio Porte**, desta dissertação.

A segunda atividade está explanada no item **4.1 - Estabelecendo um Modelo de Componentes para o Sistema,** no Capítulo 4 desta dissertação. As demais atividades serão explanadas a seguir.

6.1 - Selecionando e Estabelecendo Responsabilidades para os Objetos

Como explicado no item **4.2 - Selecionando e Estabelecendo Responsabilidades para as Classes&Objetos** no Capítulo 4 desta dissertação, pode-se começar a localização e estabelecimento das responsabilidades dos objetos por qualquer componente.

Para aplicações onde os especialistas do domínio estão particularmente interessados na aquisição dos dados e nos aspectos de controle do sistema sob consideração, que é o caso, por exemplo, de sistemas de tempo real, aconselha-se a começar a localização e estabelecimento das responsabilidades dos objetos pelos componentes domínio do problema e interação com outros sistemas.

Como localizar e estabelecer responsabilidades para os objetos do Componente Domínio do Problema, do Componente Interação Humana, e do Componente Gerenciamento de Dados está explicado nos itens **4.2.1 - Notação para Classes e Objetos** à **4.11 - Estabelecendo Responsabilidades para os Objetos de Gerenciamento de Dados**, no Capítulo 4 desta dissertação.

A Figura 6.1 mostra o Diagrama de Classe&Objeto dos Componentes Domínio do Problema, Interação Humana, e Gerenciamento de Dados para o Sistema Controlador de um Robô.

No exemplo da Figura 6.1, um robô tem como suas partes um painel, um intérprete, e eixos. Um robô sabe seu estado (Desligado, Manual, Suspenso, Executando).

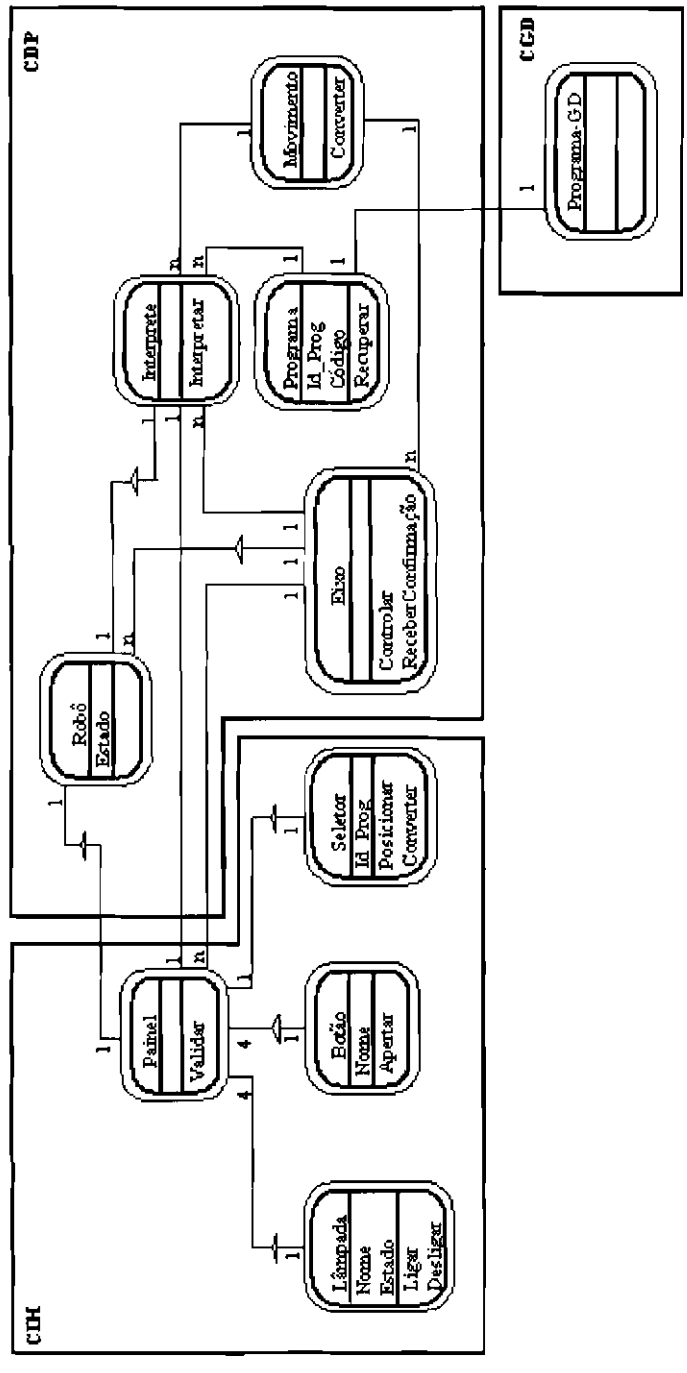


Fig. 6.1 - Diagrama de Classe&Objeto do Modelo Conceitual dos Componentes Domínio do Problema, Interação Humana, e Gerenciamento de Dados para o Sistema Controlador de um Robô.

Um painel tem como suas partes um seletor de programas, botões (Power, Run, End, Stop), e lâmpadas (Power, Run, Stop, Manual). Um painel sabe validar uma entrada fornecida pelo usuário.

Um seletor sabe se posicionar, e sabe converter uma entrada de um novo programa válida no ID do Programa.

Um botão sabe seu nome. Ele também sabe se apertar.

Uma lâmpada sabe seu estado (ligada, desligada). Ela também sabe como se ligar e se desligar.

Um intérprete sabe como interpretar as instruções lógicas e aritméticas de um programa.

Um programa sabe seu ID e seu código. Um programa também sabe se recuperar.

Um programa-GD sabe como recuperar os dados (ID e Código) de um programa, de um sistema de armazenamento de dados.

Um movimento sabe interpretar as instruções de movimento de um programa, ou seja, ele sabe convertê-las para blocos de movimento e para blocos de eixo.

Um eixo sabe se controlar, ou seja, ele sabe sair com dados para os eixos, e receber dados de eixo. Ele sabe também como receber a confirmação de que um bloco de eixo que corresponde ao fim do bloco de movimento já foi executado.

A seguir são apresentadas estratégias e padrões para a localização e estabelecimento das responsabilidades para os objetos do Componente Interação com outros Sistemas.

6.1.1 - Localizando e Estabelecendo Responsabilidades para os Objetos do Componente Interação com outros Sistemas

Os objetos do Componente Interação com outros Sistemas encapsulam a comunicação específica necessária para a interação com outro sistema ou dispositivo.

Para localizar objetos do Componente Interação com outros Sistemas deve-se procurar na especificação inicial do sistema por outros sistemas ou equipamentos com que o sistema tem que interagir. Ou seja, deve-se procurar por sistemas interagentes e por dispositivos (por exemplo : sensor, radar, elevador, controlador; não se deve considerar impressoras e monitores) que necessitem de aquisição e controle de dados. Cada objeto selecionado deve ser adicionado ao Componente Interação com outros Sistemas.

Deve-se estabelecer responsabilidades para os objetos de interação com outros sistemas estabelecendo "quem" eles conhecem (conexões), "o que" eles sabem (atributos), e "o que" eles fazem (serviços).

Para os objetos de interação com outros sistemas devem ser incluídos serviços que expressem uma representação física, ou seja, serviços que expressem detalhes da interação. Dessa maneira pode-se isolar as necessidades específicas das interações dos objetos do domínio do problema.

Estabelecidas as responsabilidades para os objetos de interação com outros sistemas deve-se agrupar o que se obteve sobre a interação com outros sistemas num mesmo diagrama de Classes&Objeto. Para uma melhor organização do Diagrama de Classe&Objeto do sistema (Diagrama de Classe&Objeto de todos os componentes do sistema), deve-se envolver o diagrama de Classe&Objeto do Componente Interação com outros Sistemas com um quadrado nomeado Componente Interação com outros Sistemas (CIS).

O Diagrama de Classe&Objeto para o Sistema Controlador de um Robô incluindo o Componente Interação com outros Sistemas é mostrado na Figura 6.2.

No exemplo da Figura 6.2, um sensor sabe como interpretar as instruções de E/S de sensores, ou seja, ele sabe ler e sair com dados de/para sensores.

6.2 - Selecionando Assuntos

Nos sistemas de médio porte os domínios de problema são mais extensos e complexos do que os domínios de problemas dos sistemas de pequeno porte, fazendo com que sua imediata compreensão torne-se muitas vezes difícil. Por isso a divisão do sistema em subsistemas pode ser uma ferramenta importante para a orientação e distribuição dos trabalhos.

No paradigma da orientação a objetos essa divisão em subsistemas pode ser feita procurando-se por "Assuntos". Para selecionar assuntos recomenda-se procurar no domínio do problema por sub-domínios.

Para tal, deve-se fazer primeiramente uma rápida identificação inicial das classes&objetos do domínio do problema, e depois identificar um conjunto inicial de sub-domínios aplicando o enfoque todo-parte no domínio do problema. Ou seja, os sub-domínios são "partes" usadas para comunicar o "todo" de um domínio de problema das responsabilidades do sistema global.

Uma boa maneira de começar a procurar pelos sub-domínios de um sistema é transformar a classe mais superior em cada estrutura Gen-Espec, e cada objeto "Todo" numa conexão todo-parte, em um sub-domínio. Além disso, deve-se transformar cada classe&objeto não pertencente a uma estrutura Gen-Espec, ou a uma conexão todo-parte, em um novo sub-domínio. Cada sub-domínio deve ser reavaliado, e pode se transformar em um assunto.

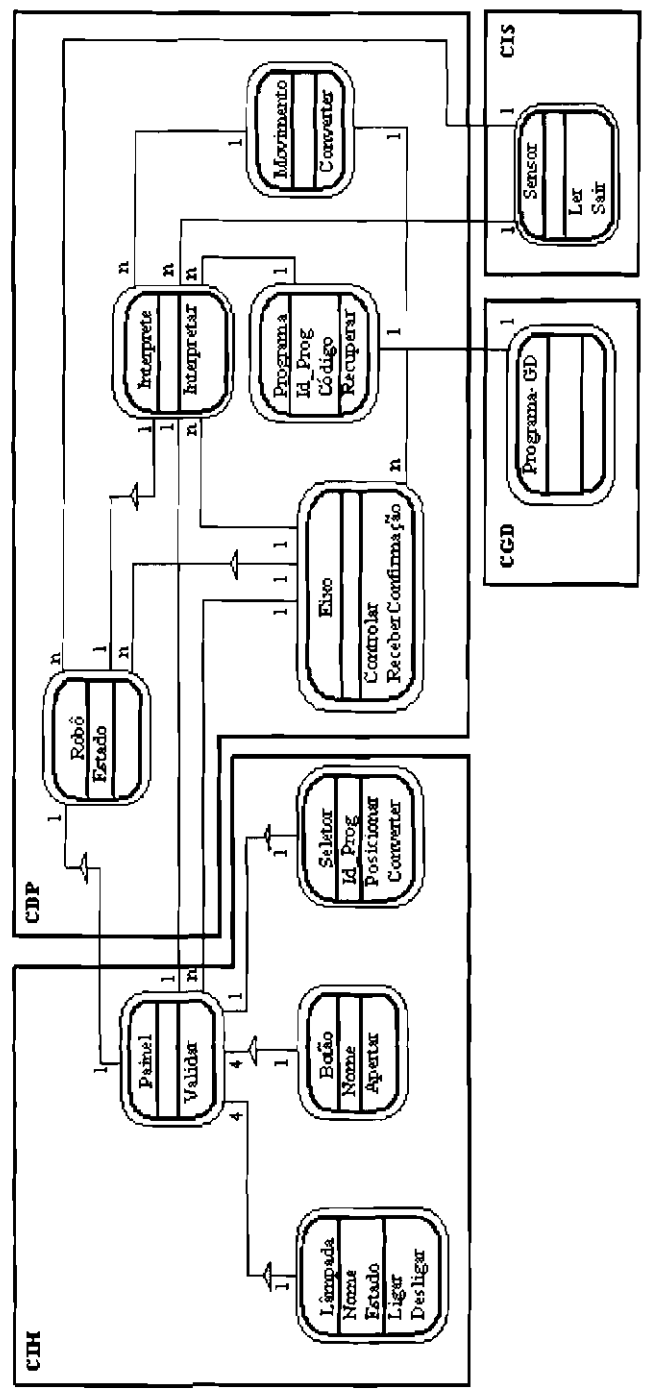


Fig. 6.2 - Diagrama de Classe&Objeto do Modelo Conceitual do Sistema Controlador de um Robô incluindo o Componente Interação com Outros Sistemas.

Esses assuntos podem ser distribuídos para as várias sub-equipes podendo dessa maneira ser abordados paralelamente. Cada assunto pode ser considerado por si só como um sistema de pequeno porte, e pode ser desenvolvido como mostrado no Capítulo 4, salvo algumas considerações adicionais que serão abordadas neste Capítulo.

Nem todo sistema de médio porte pode ter seu domínio de problema dividido em assuntos, pois muitas vezes o que o classifica como sendo um sistema de médio porte é a complexidade de seu domínio de problema, e não a sua extensão (o que é o caso do Sistema Controlador de um Robô). Nesse caso pode-se dividir o trabalho em termos dos componentes do modelo de objetos, ou seja, cada sub-equipe pode se encarregar de desenvolver um ou mais componentes do modelo.

Para cada assunto deve-se construir um Diagrama de Classe&Objeto separado, envolvendo-o com um quadrado pontilhado com a identificação do assunto. Um exemplo disso é dado na Figura 6.3.

Nesse exemplo, o sistema abordado que é um Sistema de Controle de Tráfego Aéreo, foi dividido em dois assuntos (ou subsistemas) : Vôo e Transporte Aéreo. Cada um desses subsistemas pode ser desenvolvido por sub-equipes diferentes, e depois integrados para formar o sistema como um todo.

6.3 - Exercitando a Dinâmica do Sistema

Uma boa maneira de exercitar a dinâmica do sistema é através de cenários. Os cenários de um sistema de médio porte normalmente transcendem um subsistema, o que significa que eles quase sempre envolvem a interação entre vários subsistemas (Onoma,1997).

Um subsistema, por sua vez, também pode ser constituído de cenários menores, que para a facilidade de identificação denominamos sub-cenários. Um cenário portanto, pode ser

constituído de vários sub-cenários, que por sua vez, podem pertencer a subsistemas diferentes.

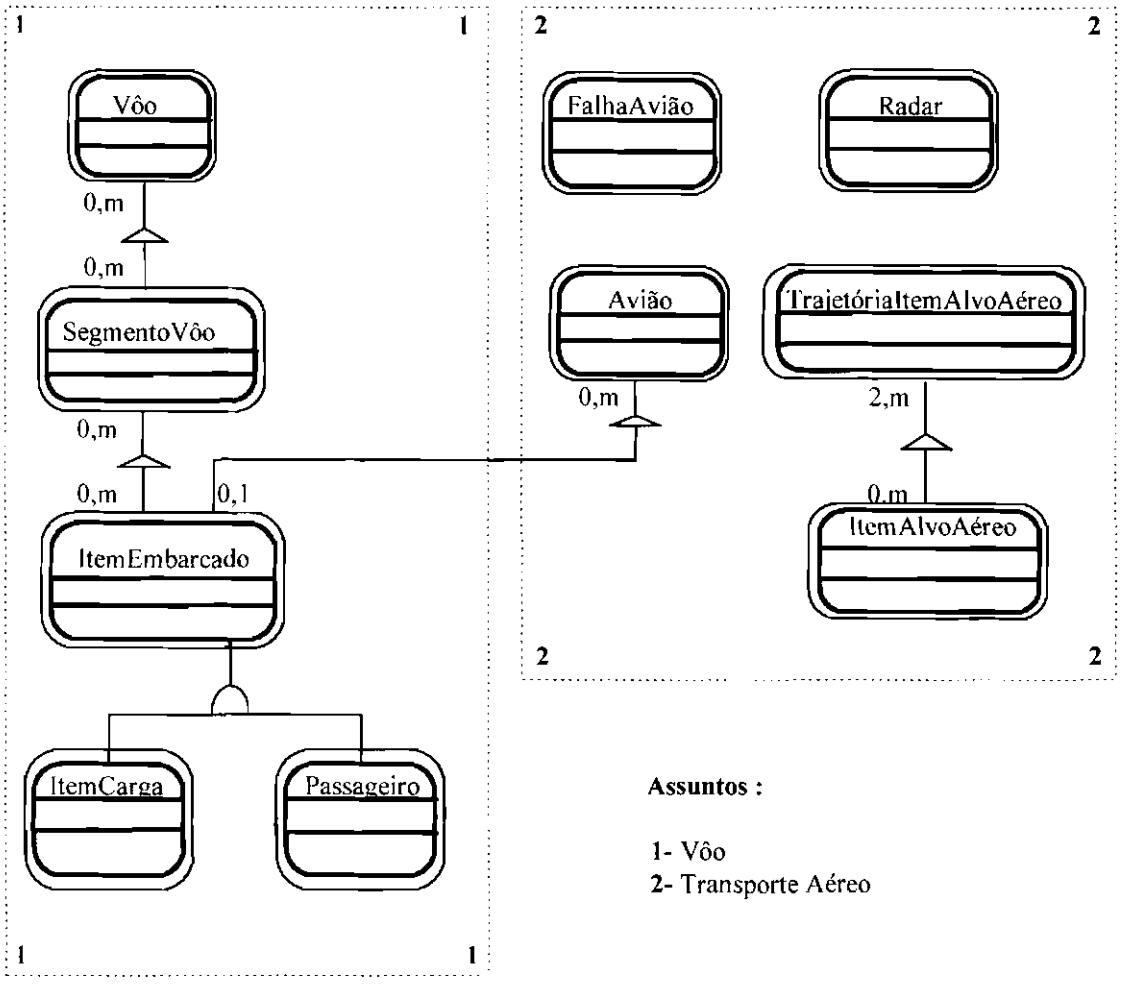


Fig. 6.3 - Assuntos.

Num sistema de médio porte deve-se exercitar a dinâmica de cada subsistema com sub-cenários como explicado no item **4.3 - Adicionando as Interações entre os Objetos**, no Capítulo 4 desta dissertação.

Para exercitar a dinâmica do sistema como um todo, deve-se exercitar a dinâmica de cada cenário do sistema; exercitando a interação entre os diversos sub-cenários que o compõe, para verificar se eles estão interagindo da maneira esperada.

Definidos os cenários e sub-cenários, deve-se incluir no modelo de objetos o Componente Gerenciamento de Cenários. Como fazer isso está explicado no item **4.4 - Adicionando Objetos do Componente Gerenciamento de Cenários**, no Capítulo 4 desta dissertação.

Para o Sistema Controlador de um Robô, foi identificado apenas o cenário "O Usuário Escolhe uma Operação do Painel". Esse cenário por sua vez, é constituído das seguintes cenas : "Seleção de um Novo Programa", "Execução de um Programa", "Suspensão da Execução de um Programa", "Reativação da Execução de um Programa", "Término da Execução de um Programa", "Inicialização do Sistema", e "Desligamento do Sistema".

A seguir é apresentado a título ilustrativo, o Roteiro para o cenário "O Usuário Escolhe uma Operação do Painel" do Sistema Controlador de um Robô. Por questões de praticidade e simplicidade foram abrangidos apenas as cenas "Seleção de um Novo Programa" e "Execução de um Programa".

Roteiro para o Cenário "O Usuário Escolhe uma Operação do Painel (1)" :

Cena : "Seleção de um Novo Programa (A)"

Eu sou um **Seletor**.
Em algum momento no tempo eu sou posicionado.
Eu conheço meu Painel. Eu aviso para ele que uma nova seleção de programa aconteceu **(Mensagem 1A1)**.

Eu sou um **Painel**.
Eu recebo uma mensagem dizendo para eu validar uma nova entrada.
Eu conheço meu Robô. Eu envio uma mensagem para ele perguntando seu estado **(Mensagem 1A2)**.
Eu valido a entrada.

Eu conheço meu Seletor. Eu aviso para ele que a entrada é válida **(Mensagem 1A3)**.

Eu sou um **Robô**.
Eu recebo uma mensagem dizendo para eu informar o meu estado. Eu informo o meu estado.

Eu sou um **Seletor**.
Eu recebo uma mensagem dizendo que a minha entrada é válida. Eu converto a entrada selecionada para o seu respectivo ID do programa.

Cena : "Execução de um Programa (B)"

Eu sou um **Botão**.
Meu nome é "Run".
Em algum momento no tempo eu sou apertado.
Eu conheço meu Painel. Eu aviso para ele que eu fui apertado (**Mensagem 1B1**).

Eu sou um **Painel**.
Eu recebo uma mensagem dizendo para eu validar uma nova entrada.
Eu conheço meu Robô. Eu envio uma mensagem para ele perguntando seu estado **(Mensagem 1B2)**.
Eu valido a entrada e envio uma mensagem para o meu Robô dizendo para ele alterar o seu estado **(Mensagem 1B3)**.
Eu conheço minhas Lâmpadas. Eu envio mensagens para elas dizendo para elas se ligarem ou se desligarem **(Mensagem 1B4)**.
Eu conheço o meu Seletor. Eu envio uma mensagem para ele perguntando o ID do programa selecionado **(Mensagem 1B5)**.

Eu conheço o Intérprete. Eu envio uma mensagem para ele dizendo para ele interpretar o programa selecionado, e dizendo o ID do programa selecionado **(Mensagem 1B6)** .

Eu sou um **Robô**.
Eu recebo uma mensagem dizendo para eu informar o meu estado. Eu informo o meu estado.
Eu recebo uma mensagem dizendo para eu alterar o meu estado. Eu altero o meu estado.

Eu sou uma **Lâmpada**.
Meu nome é "Run".
Eu recebo uma mensagem dizendo para eu me ligar. Eu me ligo.

Eu sou uma **Lâmpada**.
Meu nome é "Manual".
Eu recebo uma mensagem dizendo para eu me desligar. Eu me desligo.

Eu sou um **Seletor**.
Eu recebo uma mensagem dizendo para eu informar o ID do programa selecionado. Eu informo o ID do programa selecionado.

Eu sou um **Intérprete**.
Eu recebo uma mensagem dizendo para eu interpretar o programa selecionado
Eu recebo também o ID do programa selecionado.
Eu conheço o Programa. Eu envio uma mensagem para ele solicitando seu código **(Mensagem 1B7)**.
Eu começo a interpretar o programa selecionado.
Eu conheço o Sensor. Eu passo para ele as instruções de E/S de sensores **(Mensagem 1B9)**.

Eu conheço o Movimento. Eu passo para ele as instruções de movimento **(Mensagem 1B10)**.

Eu sou um **Programa**.
Eu recebo uma mensagem solicitando meu código.
Eu conheço o Programa-GD. Eu envio uma mensagem dizendo para ele recuperar meu código da base de dados **(Mensagem 1B8)**.
Eu recupero meu código.

Eu sou um **Programa-GD**.
Eu recebo uma mensagem dizendo para eu recuperar um código da base de dados.
Eu recupero o código da base de dados.

Eu sou um **Sensor**.
Eu recebo uma mensagem contendo as instruções de E/S para sensores. Eu executo essas instruções.
Eu conheço o Intérprete. De tempos em tempos eu envio uma mensagem para ele contendo dados de sensores **(Mensagem 1B11)**.

Eu sou um **Movimento**.
Eu recebo uma mensagem contendo as instruções de movimento. Eu converto as instruções de movimento para blocos de movimento.
Eu converto os blocos de movimento para blocos de eixo.
Eu conheço o Eixo. Eu envio uma mensagem para ele contendo um bloco de eixo, e digo para ele se esse bloco de eixo corresponde ao fim do bloco de movimento **(Mensagem 1B12)**.

Eu sou um **Eixo**.
Eu recebo uma mensagem contendo um bloco de eixo.
Eu executo o bloco de eixo.

Eu confirmo a execução de um bloco de eixo.

Eu conheço meu Intérprete. Eu envio uma mensagem para ele confirmando a execução de um movimento **(Mensagem 1B13)**.

A Figura 6.4 e a Figura 6.5 mostram respectivamente os diagramas de classe&objeto que foram gerados para as cenas "Seleção de um Novo Programa" e "Execução de um Programa" do cenário "O Usuário Escolhe uma Operação do Painel", do Sistema Controlador de um Robô. Foi usado um diagrama de classe&objeto para cada cena porque o diagrama se tornaria um pouco confuso se elas fossem representadas ao mesmo tempo.

6.4 - Projeto do Modelo de Objetos

A segunda etapa do desenvolvimento de um sistema orientado para objetos consiste em se projetar o modelo de objetos. Salvo algumas considerações adicionais feitas para o projeto de sistemas de tempo real, o projeto de cada subsistema de um sistema de médio porte orientado para objetos pode ser feito como mostrado nos itens **4.5 - Projetando o Modelo de Objetos** à **4.5.2 - Projetando os Dados que Deverão Persistir** (inclusive), no Capítulo 4 desta dissertação.

6.4.1 - Transformando os Serviços de Tempo Real em Tarefas Concorrentes

Cada tarefa (para mais informações sobre o que são tarefas, ver o item **5.2.3 - Quebrando o Sistema em Tarefas**, no Capítulo 5 desta dissertação), num sistema orientado para objetos será composta por um serviço que seja dirigido por eventos, por tempo, ou que precise ter uma execução prioritária ou crítica (Coad,1993A).

Os serviços dirigidos por eventos normalmente são responsáveis pela comunicação com um dispositivo, com uma ou mais janelas numa tela, com uma outra tarefa, com um subsistema, com outro processador, ou com outro sistema. Um serviço dirigido por

eventos deve ser transformado em uma tarefa que dispare um evento, geralmente assinalando o surgimento de algum dado. Quando o sistema está em execução as tarefas dirigidas por eventos ficam adormecidas (não consumindo tempo de processamento), esperando por alguma interrupção. Quando recebe a interrupção a tarefa desperta, captura os dados, coloca-os num *buffer* de memória ou em algum outro destino, notifica quem quer que necessite saber sobre isso, e depois volta a adormecer.

Os serviços dirigidos por tempo são acionados para fazer algum processamento num intervalo de tempo especificado; pois certos dispositivos podem necessitar periodicamente de aquisição e controle de dados; certas interfaces humanas, subsistemas, tarefas, processadores, ou outros sistemas podem precisar de comunicação periódica. Quando o sistema está em execução, a tarefa dirigida por tempo estabelece uma hora para acordar e vai dormir (não consumindo tempo de processamento), esperando por uma interrupção do sistema. Ao receber essa interrupção, a tarefa acorda, faz o seu serviço, notifica quem quer que necessite saber sobre ele, e volta a adormecer.

Os serviços prioritários e críticos acomodam tanto as necessidades de processamento de alta como de baixa prioridade. Um serviço desse tipo deve ser transformado em uma tarefa crítica para isolar o processamento que seja especialmente crítico para o sucesso ou falha do sistema sob consideração, processamento que em geral tem restrições de segurança especialmente rígidas.

Como os objetos encapsulam responsabilidades, todas as considerações sobre tarefas concorrentes devem estar encapsuladas neles. Isso significa que todas as complexidades de uma tarefa, incluindo os mecanismos de comunicação e sincronização (para mais informações sobre o que são mecanismos de comunicação e sincronização, ver o item **5.2.3.1 - Critérios para Decomposição do Sistema em Tarefas**, no Capítulo 5 desta dissertação) devem estar encapsuladas dentro daquela tarefa, e conseqüentemente dentro do objeto que a contém (Coad,1997).

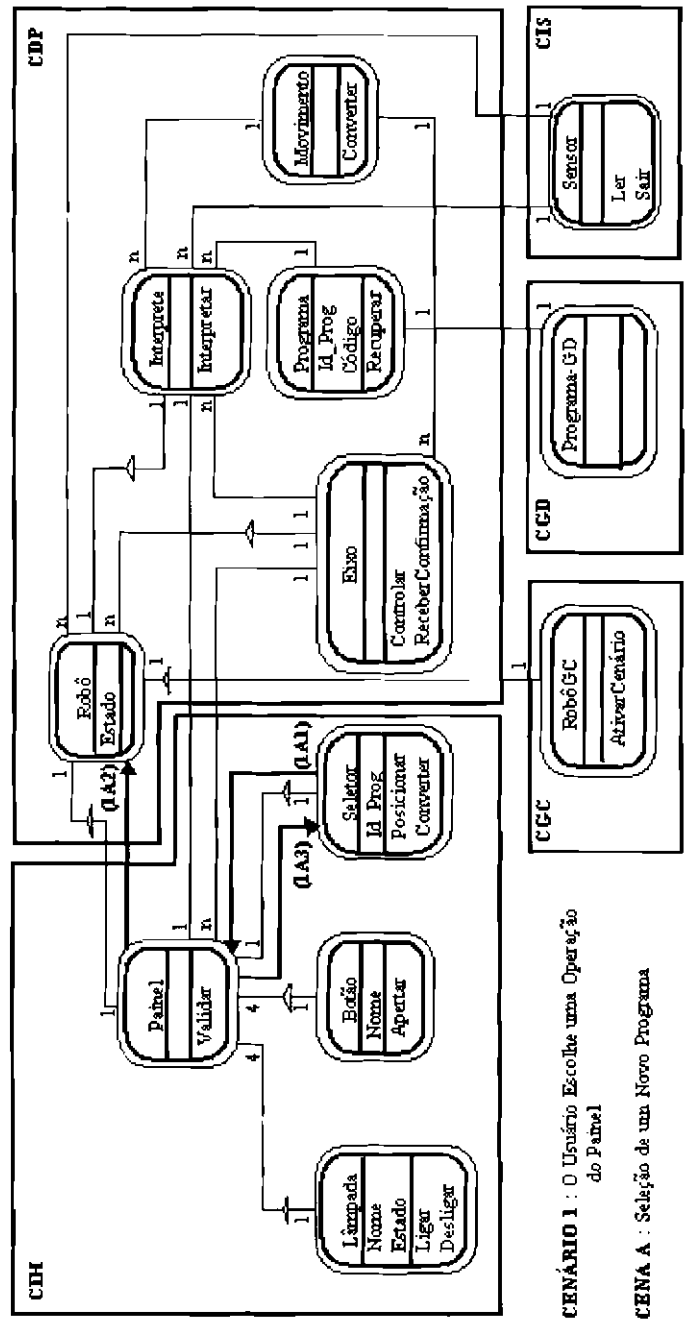


Fig. 6.4 - Cena "Seleção de um Novo Programa" do Cenário "O Usuário Escolhe uma Operação do Painel".

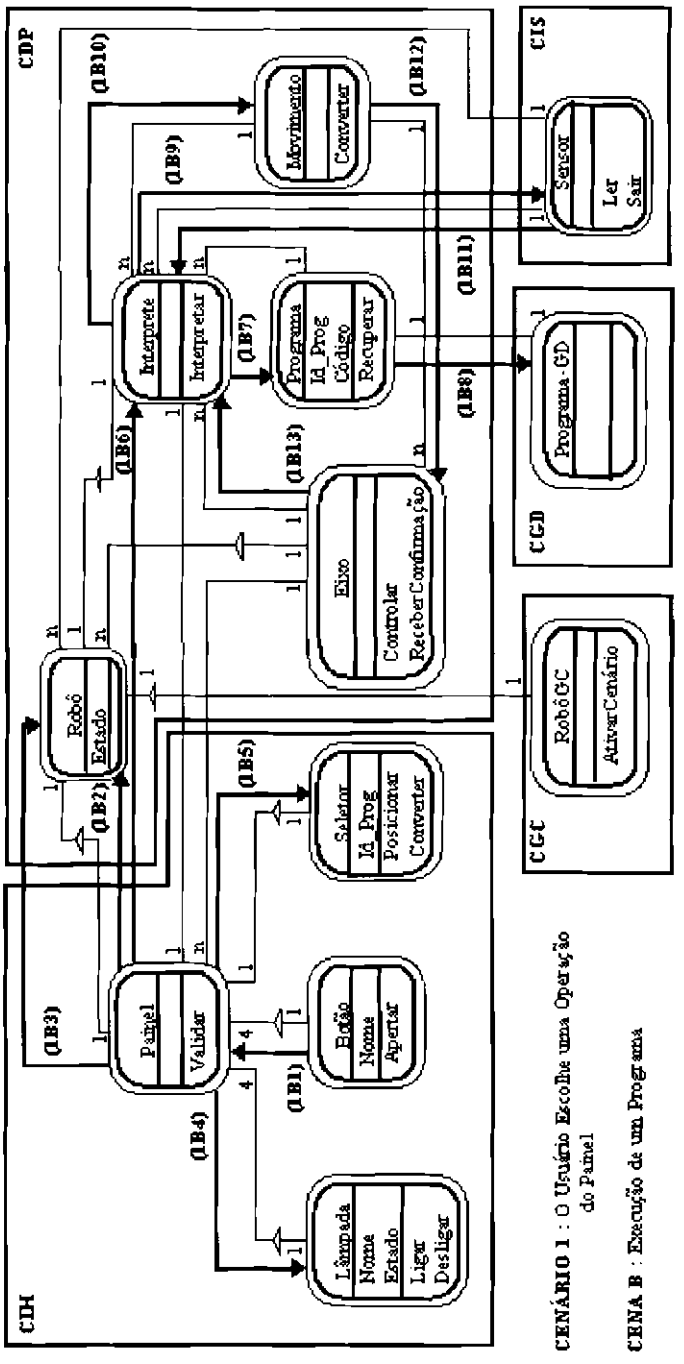


Fig. 6.5 - Cena "Execução de um Programa" do Cenário "O Usuário Escolhe uma Operação do Painel".

Quando um objeto recebe uma mensagem ativando uma tarefa que ele encapsula, ela é executada concorrentemente com os outros processos (inclusive o próprio programa que a contém) de acordo com sua prioridade.

No Sistema Controlador de um Robô, o serviço "Apertar" do objeto "Botão", o serviço "Posicionar" do objeto "Seletor", os serviços "Ligar" e "Desligar" do objeto "Lâmpada", o serviço "Sair" do objeto "Sensor", e o serviço "Controlar" do objeto "Eixo"; foram identificados como serviços dirigidos por eventos, e por isso foram transformados em tarefas separadas. Pode-se considerar também o serviço "Controlar" como sendo um serviço crítico, pois ele deve ser executado de forma muito rápida para acompanhar a velocidade de movimento dos eixos. O serviço "Ler" do objeto "Sensor" foi identificado como um serviço dirigido por tempo, e por isso também foi transformado em uma tarefa separada.

6.4.2 - Fazendo a Especificação dos Atributos de um Sistema de Tempo Real

Se o sistema sendo desenvolvido for um sistema de tempo real, ao fazer a especificação dos atributos, explicada no item **4.5.1.4.1 - Especificação para os Atributos,** deve-se acrescentar para cada atributo um campo especificando se ele é um atributo de estado, se ele é um atributo dependente de estado, ou se ele é um atributo independente de estado.

Um atributo de estado representa os estados que um objeto pode ter ao longo do programa, um atributo dependente de estado é aquele que têm seu valor modificado dependendo do valor de um atributo de estado, e um atributo independente de estado é aquele que tem seu valor independentemente de alguma outra condição.

Um atributo de estado pode ser, por exemplo, considerando o exemplo dado na Figura 6.2, "estado do robô"; um atributo dependente de estado pode ser, por exemplo, "estado da lâmpada"; um atributo independente de estado pode ser, por exemplo, "nome do botão".

Para atributos de estado deve-se acrescentar também à sua especificação, os estados possíveis que ele pode ter. Por exemplo, para o atributo "estado do robô", acrescentaria-se o campo "estados possíveis", que conteria os valores "Desligado", "Manual", "Suspenso", e "Executando".

6.4.3 - Fazendo a Especificação dos Serviços de um Sistema de Tempo Real

Se o sistema sendo desenvolvido for um sistema de tempo real, ao fazer a especificação dos serviços, explicada no item **4.5.1.4.2 - Especificação para os Serviços**, deve-se acrescentar para cada serviço um campo especificando se ele é um serviço dependente de estado, onde um serviço dependente de estado é aquele que á acionado apenas em alguns estados específicos, ou se ele é um serviço independente de estado.

Um serviço dependente de estado pode ser, considerando o exemplo dado na Figura 6.2, o serviço "Interpretar"; um serviço independente de estado pode ser, por exemplo o serviço "Validar".

Além disso, deve-se acrescentar na especificação de um serviço um campo especificando se ele é um serviço dirigido por eventos, se ele é um serviço dirigido por tempo, se ele é um serviço prioritário ou crítico, ou se ele é um serviço que não tem esse tipo de restrições.

6.4.4 - Construindo Visões de Cenários

Nos sistemas de pequeno porte, os cenários foram desenvolvidos com "Roteiros de Cenários", e as setas de mensagens foram representadas graficamente no diagrama de classe&objeto do modelo conceitual e do modelo lógico.

Num Sistema de Médio Porte, onde pode-se ter vários cenários e sub-cenários coexistindo com vários subsistemas, essa abordagem pode não ser suficiente para

representar a complexidade das interações entre os objetos, podendo ser bastante útil para a compreensão da dinâmica do sistema detalhar um pouco mais essas interações na fase de projeto.

Uma abordagem alternativa é representar as setas de mensagens apenas no diagrama de classe&objeto do modelo conceitual; e na fase de projeto, detalhar as interações entre os objetos usando o que chamamos de "Visão de Cenário".

A visão de cenário representa graficamente as interações entre os objetos de cada cena de em um cenário, não sendo necessário portanto, representar as setas de mensagens no próprio diagrama de classe&objeto do modelo lógico. Dessa maneira, o diagrama de classe&objeto do modelo lógico refletiria apenas uma visão estática do sistema, pois a visão dinâmica seria de responsabilidade da visão de cenário.

Uma visão de cenário mostra os objetos que participam de uma cena do cenário, seguidos de uma seqüência ordenada no tempo de serviços emissores, setas de mensagens, serviços receptores e *argumentos.*

Não se deve confundir uma visão de cenário com uma descrição de um serviço. Uma visão de cenário mostra as interações entre objetos. Uma descrição de um serviço apresenta os detalhes de um serviço específico.

Deve-se começar a construir uma visão de cenário de um determinado cenário pelos serviços contidos no objeto "identificador" (objeto que dispara o processo de interação), e depois deve-se expandir essa visão de cenário incluindo também os objetos interagentes.

A Figura 6.6 a seguir, ilustra a primeira versão da "Visão de Cenário" para a cena "Seleção de um Novo Programa" do cenário "O Usuário Escolhe uma Operação do Painel", do Sistema Controlador de um Robô.

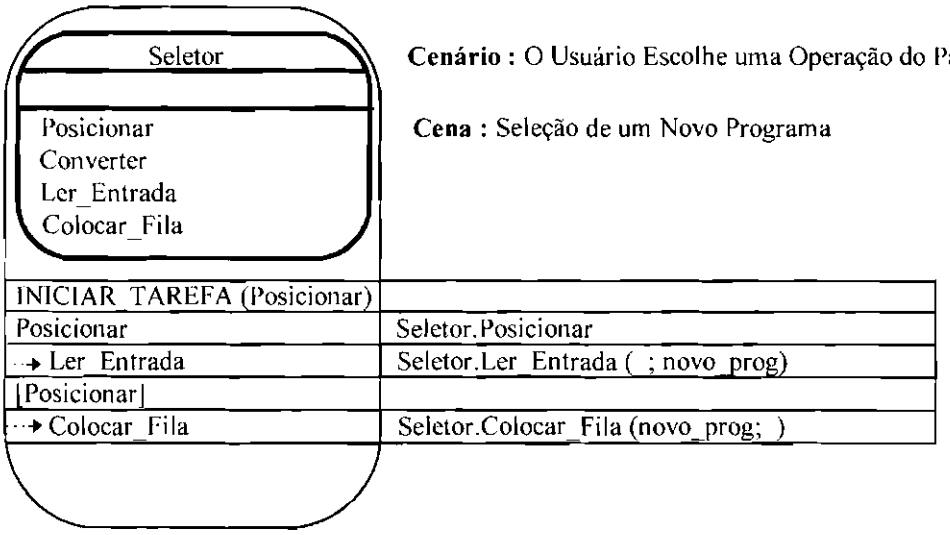


Fig. 6.6 - Construindo uma Visão de Cenário para a Cena "Seleção de um Novo Programa" do Cenário "O Usuário Escolhe uma Operação do Painel".

A visão de cenário ilustrada na Figura 6.6 é lida como :

1) O seletor do painel é posicionado.

2) A tarefa "Posicionar" é acionada.

3) O serviço "Posicionar" chama o seu sub-serviço "Ler_Entrada".

4) O serviço "Ler_Entrada" é executado e retorna o programa selecionado (novo_prog).

5) O serviço "Posicionar" chama seu sub-serviço "Colocar_Fila", enviando para ele o programa selecionado (novo_prog).

6) O serviço "Colocar_Fila" é executado.

Uma visão de cenário deve incluir também a identificação do cenário e a identificação da cena. Além disso, numa visão de cenário o objeto identificador, no exemplo anterior "Seletor", por convenção deve ficar na coluna mais à esquerda.

Os atributos dos objetos, assim como seus serviços que não participam de uma determinada cena, não devem ser listados no símbolo de classe&objeto da visão de cenário daquela cena. O primeiro serviço acionado na visão de cenário, no exemplo

anterior "Posicionar", deve ser listado imediatamente após o símbolo de classe do objeto identificador.

Os serviços complexos devem ter seus sub-serviços também representados na visão de cenário, sendo que uma mensagem entre serviços é representada graficamente por uma seta pontilhada, e é representada também através da passagem de *parâmetros* para um serviço específico.

As listas de *argumentos* dos serviços devem conter as entradas, um símbolo ";", e em seguida as saídas. Além disso, quando é necessário identificar explicitamente qual serviço é o serviço emissor, deve-se envolvê-lo num símbolo "[ ]".

As visões de cenário também podem conter estruturas de controle. As estruturas de controle possíveis de serem representadas são as estruturas da programação estruturada (para mais detalhes sobre as estruturas da programação estruturada ver o item **3.2.5 - Programação Estruturada**, no Capítulo 3 desta dissertação).

Para sistemas de tempo real pode-se usar ainda as estruturas "INICIAR_TAREFA ( )", e "TERMINAR_TAREFA ( )". "INICIAR_TAREFA ( )" indica que a tarefa entre parênteses deve ser ativada, e "TERMINAR_TAREFA ( )" indica que a tarefa entre parênteses deve ser terminada.

Deve-se observar que as estruturas de controle não devem ser usadas numa visão de cenário para detalhar um serviço específico, e sim para detalhar as interações entre os objetos.

A expansão da visão de cenário da Figura 6.6, ilustrada na Figura 6.7 a seguir, mostra a comunicação por mensagens entre objetos numa visão de cenário.

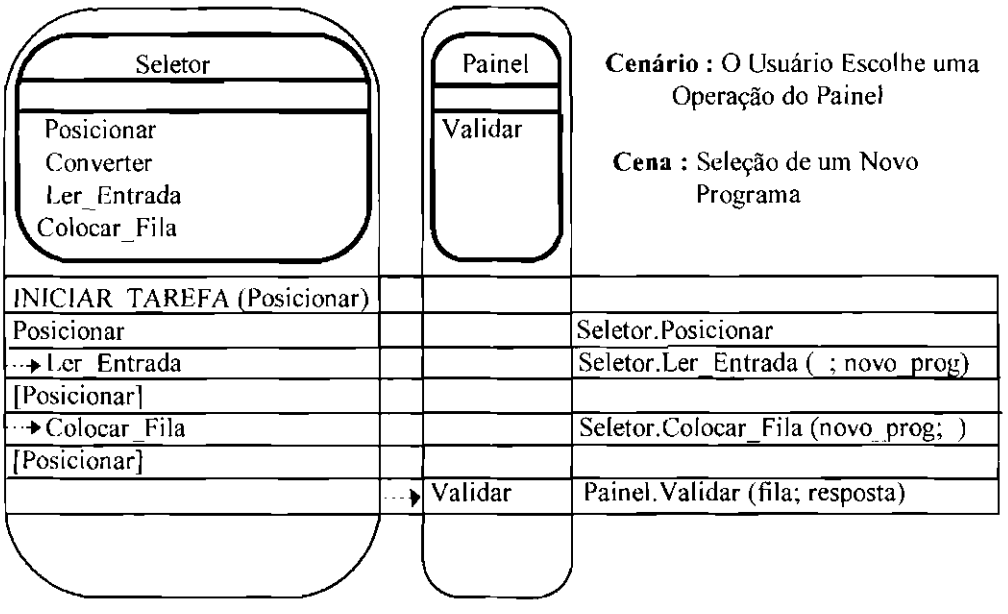


Fig. 6.7 - Expansão da Visão de Cenário para a Cena "Seleção de um Novo Programa" do Cenário "O Usuário Escolhe uma Operação do Painel".

Para a visão de cenário ilustrada na Figura 6.7 acrescentou-se :

7) O objeto seletor envia uma mensagem para o objeto painel perguntando se a entrada válida.

8) O objeto painel aciona seu serviço "Validar".

Deve-se continuar a expansão de uma visão de cenário até se abranger todos os serviços envolvidos naquela cena. A versão final da Visão de Cenário da Figura 6.7, é mostrada na Figura. 6.8.

Quando se desejar colocar algum comentário na visão de cenário deve-se faze-lo colocando se o símbolo "//" antes de iniciar o comentário, como ilustrado no exemplo anterior.

Cenário : O Usuário Escolhe uma Operação do Painel

Cena : Seleção de um Novo Programa

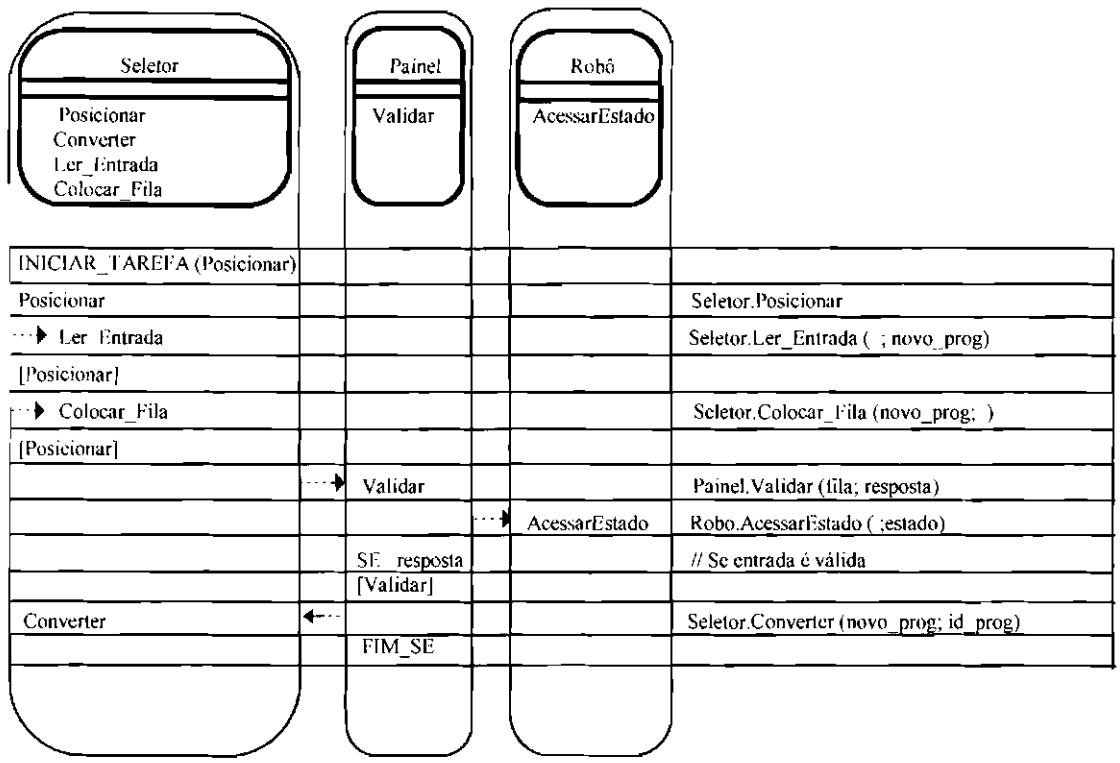


Fig. 6.8 - Versão Final da Visão de Cenário para a Cena "Seleção de um Novo Programa" do Cenário "O Usuário Escolhe uma Operação do Painel".

Para a visão de cenário ilustrada na Figura 6.8 acrescentou-se :

9) O objeto painel envia uma mensagem perguntando ao objeto robô o seu estado.

10) O objeto robô aciona seu serviço "AcessarEstado", que retorna o estado do robô (estado).

11) O serviço "Validar" do objeto painel retorna uma resposta dizendo se a entrada é válida ou não (resposta).

12) Se a resposta for positiva, o objeto painel envia uma mensagem para o objeto seletor dizendo para ele converter o programa escolhido (novo_prog) no ID do programa (id_prog).

Quando um objeto precisar ler ou modificar um atributo de um outro objeto deve-se representar na visão de cenário os serviços básicos "Acessar" e "Setar" que tem acesso a esse atributo, como mostra o exemplo da Figura 6.8. Os serviços "Criar" e "Liberar" também podem aparecer na Visão de Cenário.

Quando uma mensagem tiver que ser passada para vários objetos da classe deve-se usar a notação mostrada na Figura 6.9 a seguir.

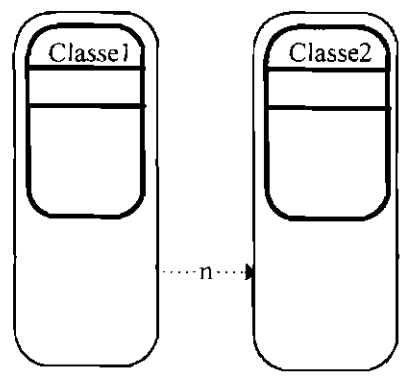


Fig. 6.9 - Mandando uma Mensagem para mais de um Objeto na Classe.

O "n" na seta de mensagem na Figura 6.9 indica que um objeto da Classe1 pode enviar a mensagem para mais de um objeto da Classe2.

6.5 - Construção do Plano de Implementação e Testes

Após realizar o projeto do modelo de objetos deve-se definir um plano de implementação e testes para o sistema (para mais detalhes sobre o plano de implementação e testes, ver o item **5.2.4 - Definição do Plano de Implementação e Testes**, no Capítulo 5 desta dissertação).

A seguir apresentaremos algumas diretrizes para implementar e testar um sistema de médio porte orientado para objetos. Essas diretrizes abrangem a adoção de uma estratégia de implementação e testes, e o projeto de casos de teste.

A estratégia de implementação e testes já foi abordada no item **4.6.1 - Adotando uma Estratégia de Implementação e Testes**, no Capítulo 4 desta dissertação. O projeto de casos de teste foi abordado no item **4.6.2 - Projetando Casos de Teste**, também no Capítulo 4 desta dissertação. Então, para cada subsistema do sistema de médio porte pode-se realizar a etapa de implementação e testes como especificado nesses itens.

À medida que os subsistemas forem sendo implementados e testados deve-se integrá-los ao sistema, e à medida que essa integração for constituindo cenários do sistema, deve-se testar esses cenários e verificar se os sub-cenários que os compõe estão interagindo da maneira esperada.

Num sistema de médio porte deve-se acrescentar também casos de teste fundamentados na interação que o software poderá ter com outros sistemas e dispositivos. Para tal, deve-se projetar casos de teste que verifiquem todas as informações que chegam desses elementos, e todas as informações que saem do software para esses elementos.

Após projetar os casos de teste deve-se construir o plano de testes para o sistema. Como realizar essa atividade está explicado no item **5.2.4.3 - Construção do Plano de Testes**, no Capítulo 5 desta dissertação. Ao ler esse item deve-se ignorar os exemplos dados, e observar os exemplos que serão dados nas Figura 6.10 e Figura 6.11.

É necessário salientar que apenas dois casos de teste foram descritos a título de ilustração, de modo que, um plano de testes real para o Sistema Controlador de um Robô deveria conter todos os casos de teste possíveis de serem realizados.

6.6 - Implementação do Sistema de Médio Porte Usando Técnicas Orientadas para Objetos

A terceira etapa do desenvolvimento de um sistema de software é a construção do modelo físico do sistema, que é nada mais nada menos que implementá-lo. Como realizar

esta atividade está explanado nos itens **4.7 - Implementando o Sistema de Pequeno Porte Usando Técnicas Orientadas para Objetos** em diante, no Capítulo 4 desta dissertação. Então, para cada subsistema do sistema de médio porte pode-se seguir as diretrizes de implementação dadas nesses itens.

Tipo do Teste : Teste de Integração com outros Dispositivos
Data da Realização do Teste : 25/06/98
Objetivo do Teste : Verificar se o Sistema Controlador de um Robô está recebendo e enviando corretamente os dados de/para sensor, que é fornecido pelo dispositivo externo "Sensor".
Cronograma do Teste : Deve ser realizado após os testes de Integração dos Cenários.
Descrição do Teste **Entradas :** Dados recebidos do dispositivo "Sensor". **Saídas :** Dados enviados para o dispositivo "Sensor". **Objeto Responsável pela Interação :** Objeto "Sensor" do CIS. Os dados são enviados pelo serviço "Sair", e são recebidos pelo serviço "Ler".
Resultados Esperados : O sistema deve estar lendo periodicamente os dados do dispositivo sensor, e deve estar enviando dados para ele quando um evento de saída de dados para sensor acontece.
Observações : Os dados recebidos devem estar sendo convertidos para o formato interno do sistema, e os dados enviados devem estar sendo convertidos para o formato do sensor.

Fig. 6.10 - Descrição para Caso de Teste de Integração com Outros Dispositivos.

Tipo do Teste : Teste de Cenário
Data da Realização do Teste : 20/06/98
Objetivo do Teste : Testar a interação dos objetos do cenário "Seleção de um Novo Programa".
Cronograma do Teste : Deve ser realizado após a implementação e teste individual das classes que compõe esse cenário.
Descrição do Teste **Entradas :** Um novo programa do painel. **Saídas :** ID do Programa escolhido.
Resultados Esperados : Ao receber um novo programa, deve-se verificar se a entrada é válida, pois para receber um novo programa o robô deve estar no estado manual. Se a entrada for válida deve-se guardar o ID do programa escolhido.
Observações : A chave seletora de programas não têm luzes.

Fig. 6.11 - Descrição para Caso de Teste de Cenário.

CAPÍTULO 7

SISTEMAS DE GRANDE PORTE

Pode-se dizer que um sistema de grande porte é um sistema que tem uma complexidade grande; ou seja, um sistema que envolve problemas que são difíceis de serem solucionados.

Apesar da complexidade não estar associada diretamente ao tamanho do sistema, em geral os sistemas de grande porte possuem muitas funções (Ghezzi,1985). A equipe e o tempo de desenvolvimento também são maiores que nos sistemas de pequeno e médio porte.

Um sistema de grande porte pode ser classificado em :

1) Sistemas Grandes. Requerem de 5 a 20 desenvolvedores trabalhando por um período de 2 a 3 anos, resultando em um sistema de 50.000 a 100.000 linhas de código, distribuídas em diversos subsistemas. Um sistema grande tem freqüentemente interações significativas com outros programas e com outros sistemas de software. Exemplos de sistemas grandes podem ser *compiladores* grandes, *sistemas de tempo compartilhado* pequenos, pacotes de banco de dados, programas gráficos para aquisição e mostra de dados, sistemas de controle em tempo real.

A comunicação entre desenvolvedores, gerentes e clientes torna-se complexa nesses sistemas. Um sistema grande requer mais do que uma equipe de desenvolvimento (por exemplo, três equipes de cinco pessoas cada) e freqüentemente envolve mais do que um nível de gerenciamento. Em adição, há uma grande probabilidade que existam mudanças na equipe de projeto durante o ciclo de desenvolvimento. Isso

requererá treinamento de novo pessoal, ou a distribuição das responsabilidades da equipe que saiu, para as equipes que ficaram.

O tamanho e a complexidade de um sistema grande tornam difícil, se não impossível prever eventualidades durante o planejamento e a análise. Mudanças nos requisitos do produto podem partir dos usuários e dos desenvolvedores à medida que o projeto evolui. Processos sistemáticos, documentos padronizados, e revisões formais são essenciais ao longo do desenvolvimento de um sistema grande (Fairley,1985).

2) Sistemas Muito Grandes. Requerem de 100 a 1000 desenvolvedores por um período de 4 a 5 anos e resultam em sistemas de software de 1 milhão de instruções. Um sistema muito grande geralmente consiste de muitos subsistemas principais, cada um formando um sistema grande. Os subsistemas normalmente tem interações complexas uns com os outros e com outros sistemas separados.

Sistemas muito grandes geralmente envolvem processamento em tempo real, telecomunicações, *multitarefas*. Exemplos de sistemas desse tipo incluem *sistemas operacionais* grandes, *sistemas de gerenciamento de banco de dados* grandes, sistemas de controle e comando militar (Fairley,1985).

3) Sistemas Extremamente Grandes. Requerem de 2000 a 5000 desenvolvedores por períodos de mais de 10 anos, e resultam em 1 milhão a 10 milhões de linhas de código. Sistemas extremamente grandes consistem de muitos subsistemas muito grandes, e freqüentemente envolvem processamento em tempo real, telecomunicações, *multitarefas*, *e processamento distribuído*. Tais sistemas freqüentemente envolvem requisitos de confiabilidade e envolvem processos de vida ou morte. Exemplos de sistemas extremamente grandes incluem controle de tráfego aéreo, controle de mísseis, sistemas de controle e comando militar. Poucos sistemas extremamente grandes já foram construídos (Fairley,1985).

Como pudemos observar, os sistemas de grande porte exigem muito mais controle e gerenciamento do que os sistemas de pequeno e médio porte; e por isso uma série de medidas adicionais devem ser tomadas durante o processo de desenvolvimento e manutenção de sistemas desse tipo.

Tudo isso acontece porque em sistemas de grande porte não há tempo suficiente para se conversar com cada membro da equipe, para observar de primeira mão o que está acontecendo em cada área, e integrar os dados de uma maneira acurada.

Além disso, aparecem outras dificuldades, como por exemplo; controlar a multiplicidade de versões de um módulo de programa (para mais detalhes sobre módulo ver o item **2.10.1.1 - Técnicas Estruturadas** no **Capítulo 2 - Software e Engenharia de Software**, desta dissertação).

Num sistema de pequeno porte esse problema pode ser ignorado; porém, em sistemas mais complexos, onde o tamanho pode chegar a algumas centenas de milhares de linhas de código, com milhares de módulos de programa, e com uma equipe de dezenas de profissionais; isso certamente será mais difícil de ser controlado; pois, no desenvolvimento desse tipo de sistema, um módulo produzido por um grupo, poderá ser reutilizado por outros grupos em outras partes do sistema; e uma alteração nesse módulo poderá afetar outros módulos.

O mesmo acontece com os documentos produzidos. Eles estão sujeitos a alterações e poderão sofrer influências de alterações realizadas em outros documentos do software (Cunha,1993).

A complexidade de um sistema cresce exponencialmente com o seu tamanho, e por isso, sistemas de grande porte exigem que sejam adotados métodos que ajudem a solucionar os problemas criados pelas linhas crescentes de comunicação (tanto no próprio software, quanto na equipe de desenvolvimento); problemas criados pela dificuldade de se

compreender perfeitamente o papel que o software deverá exercer dentro do contexto existente; e problemas criados pela necessidade de se adotar uma estratégia para integrá-lo incrementalmente, à medida que ele vai sendo desenvolvido. Em outras palavras, o que funciona muito bem em sistemas de pequeno e médio porte, não necessariamente funciona bem em sistemas de grande porte.

Em sistemas desse tipo, traçar um roteiro para o trabalho, e sempre saber em que estágio do desenvolvimento se está, tanto em termos de processos, quanto em termos de produtos, é essencial. Com esse tipo de dado torna-se mais fácil visualizar as áreas que não estão dentro do cronograma estabelecido e, se necessário, redistribuir recursos, focando a atenção nas áreas problemáticas (Lindstrom,1993).

7.1 - Dificuldades Encontradas no Desenvolvimento de Sistemas de Grande Porte

No desenvolvimento de sistemas de grande porte poderão ocorrer problemas que seriam facilmente evitados em sistemas mais simples.

Vejamos por exemplo, como descrito em Lindstrom (1993), o que aconteceu com a "Paramax Eletronic Systems", uma companhia que tinha mais de 30 anos de experiência em desenvolvimento de software.

O grupo de engenharia de software da "Paramax" teve que aprender duras lições sobre engenharia de disciplina, o valor das ferramentas, técnicas e metodologias para o desenvolvimento de software, e a necessidade de um bom gerenciamento de engenharia; durante o desenvolvimento de um sistema de grande porte (mais de 300.000 linhas de código) em C, que deveria ser tolerante a falhas e era caracterizado como um sistema de tempo real. O que começou como algo excitante, novo, de alta tecnologia, rapidamente se tornou um desastre; pois o cronograma não foi cumprido, e os custos se excederam.

Os problemas encontrados ilustram o que acontece se alguns dos princípios da engenharia de software e da engenharia de gerenciamento são violados no desenvolvimento de sistemas de grande porte. Os principais problemas foram :

- **Sistema de engenharia inadequado durante a proposta e o desenvolvimento.** Produziu-se um sistema *protótipo* do que o cliente desejava. Embora o *protótipo* seja uma boa maneira de se entender o problema; infelizmente ele não foi usado de modo adequado, pois foram usados os dados do *protótipo* para estimar o porte do sistema.

Mais tarde foi gerado um modelo realístico do sistema, o que permitiu que se tivesse uma visão melhor dos requisitos. Os desenvolvedores encontraram muitos requisitos inconsistentes, incorretos e incompletos. Esse modelo foi usado para realizar muitas mudanças no *hardware* e no software do sistema; isto é, acabou-se por ter que fazer uma reengenharia do projeto e do código.

Embora freqüentemente houvesse reuniões entre o grupo de desenvolvedores e os representantes técnicos do cliente para determinar como o sistema deveria trabalhar; nem sempre as decisões eram documentadas. Os procedimentos de documentação de mudanças foram ignorados, e muitas dessas mudanças não foram nem sequer implantadas.

As atividades de engenharia do sistema foram completadas, porém com um esforço significativo de reengenharia.

- **Gerenciamento Inadequado dos Requisitos.** No caso do sistema em questão os requisitos em algumas áreas foram pouco mais do que uma descrição de como o *protótipo* trabalhava, ao invés de retratar o quê o sistema deveria fazer. Em outras áreas os requisitos estavam detalhados num nível bem baixo de detalhes de projeto. Essa variação fez com que as revisões se tornassem ineficientes, pois não se podia

ter uma visão completa do sistema em um nível de abstração consistente. Além disso, os requisitos que se alteraram ao longo do tempo não foram documentados, e por conseqüência, muitas dessas alterações não foram refletidas no código. Isso culminou com uma fase de testes muito mais longa do que a planejada, pois não se tinha detalhes consistentes dos requisitos e informações apropriadas sobre eles na especificação.

- **Estimativa imprópria do *hardware*.** O *hardware* que foi adotado para o sistema não atendia a todas às suas necessidades, e mesmo com vários *upgrades* teve que se fazer uma otimização significativa das especificações originais do sistema, causando um maior acoplamento dos componentes do software, aumentando conseqüentemente sua complexidade, e fazendo com que o custo das atividades de integração e testes crescessem proporcionalmente.

- **Seleção de metodologia inadequada.** Foi usada uma metodologia de projeto (o artigo não diz qual) para decompor o problema em módulos pequenos que poderiam ser escritos em C; com interfaces definidas para os outros módulos, para o sistema operacional, e para o *hardware*. Como o programa era muito grande foram criados mais de 2.000 módulos com mais de 6.000 interfaces.

Além disso, começou-se o projeto com engenheiros que tinham pouca experiência em sistemas de grande porte, e que não tiveram nenhum treinamento especial para tal. As técnicas que eles conheciam funcionavam bem com sistemas de pequeno porte, e porque foi assumido que o sistema seria de pequeno porte; foi assumido também que as mesmas técnicas iriam funcionar. Acabou-se descobrindo que o sistema tinha muito mais características de um sistema de grande porte, e que não seria possível adequá-lo às técnicas escolhidas.

- **Perda do Controle do Desenvolvimento.** O projeto em questão não teve um software de *métricas* eficiente que tivesse medidas para prever *performance*, linhas de código etc..., isto é, que ajudasse no controle do desenvolvimento do software.

A seguir, falaremos um pouco dos procedimentos que poderiam ter sido adotados pela "Paramax" para evitar que tantos problemas aparecessem durante o desenvolvimento do software citado acima.

Primeiramente, deveria ter sido feita uma revisão da especificação do sistema proposto comparando os dados obtidos, com os dados do *protótipo*. Se isso tivesse sido feito teria-se notado que o *protótipo* não abrangia muitos dos requisitos do sistema; como tolerância, detecção e recuperação de falhas, *multiprocessamento*, requerimentos de *backup*, tempo real e otimização da *performance* do código; e que na verdade, mais requisitos do software estavam faltando no *protótipo*, do que estavam incluídos nele.

Isso trouxe muitos problemas pois num sistema de grande porte nenhuma atividade de engenharia pode fazer mais para garantir o sucesso de um projeto, do que o gerenciamento apropriado dos requisitos; e por isso, torna-se fundamental que eles sejam especificados num nível aprofundado de detalhes.

Podemos dizer então, que monitorar a volatilidade de um requisito é muito importante, isto é, quando um requisito muda o gerente deve ficar alerta para verificar se essa mudança é realmente necessária, e para garantir que ela não introduza um risco significativo para o projeto. Requisitos que estejam faltando ou que estejam incompletos podem indicar que houve uma análise inadequada desses requisitos; que os processos estão pobres; e que detalhes de projeto apareceram muito cedo no ciclo de vida criando uma tendência a passar rapidamente para o projeto.

Todas as decisões que foram tomadas em reuniões entre o grupo de desenvolvedores e os representantes técnicos do cliente deveriam ter sido documentadas para que pudessem

ser implantadas corretamente, sem correr o risco de se cometer enganos e esquecimentos. Os procedimentos de documentação de mudanças deveriam ter sido seguidos, evitando assim, que se perdesse o controle do que realmente se tinha no software.

Ao se fazer uma análise mais completa dos requisitos do sistema como dito acima, poderia-se perceber também que o *hardware* escolhido não atendia às necessidades do sistema em questão, e a escolha teria sido um *hardware* que se encaixasse melhor com a especificação do sistema.

Com relação à metodologia escolhida para o desenvolvimento do sistema, pode-se dizer que ela não foi adequada porque não proveu uma maneira de integrar os módulos em pacotes de mais alto nível que poderiam ser integrados independentemente, e formar níveis mais altos de abstração, facilitando o entendimento da arquitetura do software.

Pior do que atrasar o desenvolvimento de um projeto é não se conscientizar desse atraso, e por isso o controle do desenvolvimento é muito importante. Num sistema de grande porte não se pode simplesmente aumentar de escala as técnicas de gerenciamento que funcionam bem com sistemas de pequeno e médio porte. Gerentes de projetos grandes dependem da coleta e interpretação correta dos dados do projeto; e por essas razões, um elemento chave para um projeto bem gerenciado é um programa de *métricas* eficiente. As *métricas* podem não fazer com que um programa se livre dos atrasos e atropelamentos, mas elas constituem ferramentas essenciais para que os gerentes monitorem o estado de um projeto, tendo tempo de tomar ações corretivas que minimizem os efeitos de custo e cronograma.

7.2 - Diretrizes para a Construção de um Sistema de Grande Porte

Como pudemos observar, em sistemas de grande porte a necessidade de se controlar tudo o que acontece durante o desenvolvimento e a manutenção é um fator vital para o

sucesso do software. Porém, para que esse esquema de controle funcione é necessário que os membros da equipe de desenvolvimento executem procedimentos de controle adequadamente integrados ao processo de desenvolvimento. Esses procedimentos estabelecem as normas e as responsabilidades que garantem a correta execução do esquema de controle.

Portanto, ao se estabelecer o processo de desenvolvimento torna-se necessário um conjunto de providências que incluem : a organização do ambiente de desenvolvimento, das equipes, da estrutura de diretórios e contas, e de outros recursos computacionais.

7.2.1 - O Controle dos Produtos, Itens e Relacionamentos

Devido às características do sistema de grande porte, controlar o software, seus produtos e itens durante o desenvolvimento e a manutenção, é um fator fundamental na garantia da qualidade e da produtividade do sistema (Nakanishi,1993).

Os produtos incluem *arquivos de dados*, *procedimento* de comandos, *arquivos include*, além dos tradicionais programas, arquivos e documentos. Os itens são partes elementares de produtos, ou agrupamento de elementos, por exemplo : processo DFD, *event flag* e *variável global* são itens pois são partes de documentos e programas; *cluster* de *event flags* e tarefas são itens pois agrupam respectivamente *event flags* e tarefas (Cunha,1993).

Portanto, nesse tipo de sistema além dos produtos tradicionais (software e documentos), outros itens também precisam ser verificados quanto às suas qualidades (corretude, consistência, etc...) e ser colocados sob controle; e uma vez feito isso, todas as alterações e exclusões de qualquer um desses elementos só poderão ser realizadas após a devida análise e aprovação; preservando assim a qualidade do software.

Isso quer dizer que todos esses elementos produzidos pelo software devem passar por uma verificação de qualidade, passando assim a serem "elementos controlados". Como "elementos controlados" eles podem ser consultados livremente, porém não podem ser alterados sem uma prévia autorização.

Os produtos, software e documentos, após verificados e aprovados devem ser armazenados em "Bibliotecas Controladas". Os elementos dessas bibliotecas podem ser acessados para leitura por qualquer equipe, porém, só podem ser acessados para escrita por equipes autorizadas, sendo que essa autorização é feita por um esquema de contas e prioridades.

A determinação de todos os itens e outros produtos relacionados com um dado produto a ser modificado também é necessária. Isso quer dizer que além das informações de cadastro dos produtos e itens, também é necessário manter informações sobre esses relacionamentos, ou seja, associações existentes entre produtos e produtos, produtos e itens, e itens e itens. Esta tarefa não pode ser feita manualmente devido à quantidade de informações e devido ao volume de processamento nos cruzamentos de informações, dessa forma um banco de dados é necessário para manter os cadastros de todos os produtos, itens e relacionamentos. Tanto quanto as bibliotecas de produtos e itens, o banco de dados de cadastro também deve ser controlado.

Então, os cadastros de todos os produtos, itens e relacionamentos, devem ser mantidos em um banco de dados denominado de "Banco de Dados de Elementos Controlados". Ele é semelhante a um *dicionário de dados*, e contém as informações básicas de todos os produtos, itens e relacionamentos. Como dito anteriormente, os cadastros dos elementos só podem ser alterados mediante prévia autorização, mas são liberados para leitura.

Essa necessidade de controle se estende também para a fase de operação, de maneira que a manutenção seja realizada preservando o nível de qualidade que se obteve no

desenvolvimento; pois o relaxamento nas atividades de controle poderá permitir uma rápida degeneração da qualidade e uma drástica redução da vida útil do sistema.

Portanto, os investimentos feitos para a implementação desse controle, além de serem essenciais durante o desenvolvimento, poderão continuar dando um retorno positivo durante toda a existência do sistema, possibilitando a execução eficiente e segura das atividades de manutenção.

7.2.2 - A Organização do Ambiente de Desenvolvimento

Durante o desenvolvimento e a manutenção de um sistema são realizadas diversas atividades. Quando se trata de um sistema de grande porte, a quantidade e a variedade dessas atividades fazem com que se tenha uma necessidade muito grande de que elas sejam executadas de maneira organizada e controlada. Por isso, a organização do ambiente de desenvolvimento é uma das providências importantes a serem tomadas, facilitando também a implementação do esquema de controle comentado anteriormente.

A organização do ambiente de desenvolvimento consiste basicamente na sua quebra em ambientes especializados; sendo que cada um desses ambientes deve agrupar atividades correlatas e pertinentes a um dado objetivo, ou a uma parte do processo de desenvolvimento e/ou manutenção. Essa quebra deve ser realizada levando-se em consideração as características do sistema a ser desenvolvido, a metodologia adotada, a qualidade pretendida e os recursos disponíveis.

Um exemplo de quebra do Ambiente de Desenvolvimento é ilustrado na Figura 7.1. Nesse exemplo o ambiente de desenvolvimento foi quebrado em :

- ambiente de sistema;
- ambiente de controle de configuração;

- ambiente de controle de qualidade;
- ambientes de projeto e implementação;
- ambientes de integração de subsistemas;
- ambiente de integração do sistema e
- ambiente de prototipação.

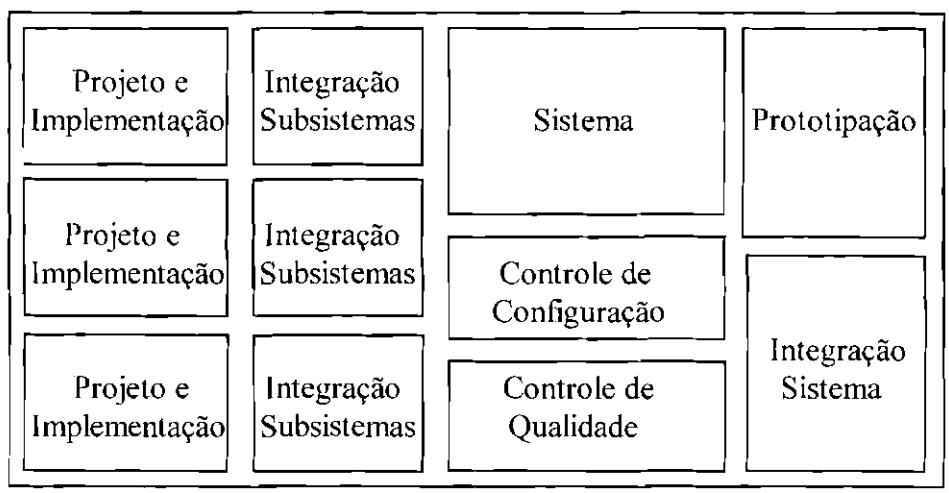


Fig. 7.1 - Um Exemplo de Ambiente de Desenvolvimento.

FONTE : Adaptada de Cunha (1993, p.1316).

7.2.2.1 - Ambiente de Sistema

Neste ambiente são realizadas as atividades de especificação dos requisitos do sistema; definição dos ambientes e do processo de desenvolvimento; estabelecimento de normas e padrões (definições dos formatos de nomes lógicos (nomes para coisas físicas definidas independentemente do programa, por exemplo, nomes de caminhos de diretórios), da estrutura padrão de diretórios e bibliotecas, da estrutura padrão de contas, dos padrões de formatos de telas e teclados, do formato da documentação interna dos programas, dos tipos e dos formatos de documentos e manuais, etc...); especificação funcional; especificação dos dados; divisão do sistema em subsistemas; especificação das interfaces lógicas entre subsistemas; aprovação das especificações dos subsistemas; definição dos

testes de aceitação do sistema; aprovação dos planos de testes; e aprovação das alterações dos elementos controlados.

Estas atividades são de responsabilidade da Equipe de Sistema, que deve ser composta por líderes de equipe e por representantes dos usuários, devendo ser liderada pelo gerente do sistema de software. Essa equipe deve manter o controle da visão global do sistema e de suas interfaces durante todo o desenvolvimento, sendo que um dos objetivos mais importantes a ser perseguido por ela é a divisão do sistema em subsistemas; pois isso possibilita uma primeira quebra do problema em problemas menores, permitindo que a partir de então as atividades de desenvolvimento sejam realizadas em paralelo por diferentes equipes.

A Equipe de Sistema deverá desenvolver a especificação do sistema de forma centralizada até que seja possível a identificação dos subsistemas; e essa atividade de especificação deve ter continuidade com cada equipe responsável por desenvolver os subsistemas; sendo que todos os planos de testes, tanto o de integração do sistema, como o dos subsistemas, tarefas, e módulos, devem ser aprovados pela Equipe de Sistema com base nessas especificações.

Todas as possíveis alterações dos elementos controlados devem passar pela aprovação da equipe do sistema antes de serem realizadas; sendo que as propostas de alteração devem ser analisadas quanto ao seu mérito técnico, quanto aos aspectos de custo/benefício e quanto à alocação de recursos.

7.2.2.2 - Ambiente de Controle de Configuração

Neste ambiente são realizadas as atividades de controle de versões do sistema; manutenção das bibliotecas controladas; transferências e encaminhamentos dos elementos produzidos entre os diversos ambientes; e controle do Banco de Dados de

Elementos Controlados. Estas atividades são de responsabilidade da Equipe de Controle de Configuração, que deve ser constituída de um ou mais profissionais.

O controle de versões do sistema consiste nas atividades de gerar e manter as versões do sistema, sendo que uma versão é definida através de procedimentos que especificam os produtos que a compõe. Esses procedimentos definem um conjunto de regras de dependências e ações, e contém todas as informações necessárias para a construção da versão desejada.

As Bibliotecas Controladas também fazem parte do ambiente de Controle de Configuração, e portanto, a sua manutenção é de responsabilidade da equipe de Controle de Configuração. Elas armazenam produtos aprovados pela Equipe de Sistema, tais como documentos de especificação e planos de testes; e produtos aprovados pela equipe de Controle de Qualidade, tais como programas, dados, etc...

O controle de todas as transferências de elementos produzidos de um ambiente para outro, isto é, do ambiente de origem o elemento é encaminhado para o Controle de Configuração, e este se encarrega de enviá-lo ao seu destino; também é feito pela Equipe de Controle de Configuração, sendo que todas essas transferências devem ser registradas.

O controle do Banco de Dados de Elementos Controlados também é feito pela equipe de Controle de Configuração, e consiste na liberação para alterações do cadastro de elementos controlados mediante prévia aprovação.

Além dessas atividades básicas, a Equipe de Controle de Configuração é responsável pelo controle de abertura das contas, e pela manutenção das estruturas dos diretórios de arquivos e bibliotecas que foram estabelecidas para o desenvolvimento do sistema.

7.2.2.3 - Ambiente de Controle de Qualidade

Neste ambiente são realizadas as atividades de verificação dos elementos produzidos e participação na definição de normas e padrões de qualidade; sendo que essas atividades devem ser realizadas pela Equipe de Controle de Qualidade.

A verificação dos elementos produzidos consiste em checar se eles estão de acordo com as especificações e com as normas e padrões de qualidade estabelecidos, determinando as correções que se fizerem necessárias. Além disso, são analisadas a consistência e a completeza do cadastro dos elementos produzidos no Banco de Dados de Elementos Controlados. Os elementos produzidos que satisfizerem as normas e padrões que foram estabelecidos, e que tenham seus cadastros consistentes no Banco de Dados de Elementos Controlados, são aprovados e passam a ser "elementos controlados". Isso quer dizer que a partir de então esses elementos e seus cadastros somente poderão ser alterados mediante prévia autorização.

A equipe de controle de qualidade também realiza as atividades de assessoria e de geração de propostas para o estabelecimento das técnicas, metodologias, normas e padrões, para a obtenção da qualidade desejada para o sistema. Essas propostas são feitas para a Equipe de Sistema.

7.2.2.4 - Ambientes de Projeto e Implementação

As atividades de Projeto e Implementação de um sistema de grande porte devem ser realizadas em diferentes ambientes de Projeto e Implementação; sendo que para facilitar o controle dos elementos produzidos, em cada um desses ambientes deve ser desenvolvido um único subsistema. Cada subsistema é de responsabilidade de uma equipe de Projeto e Implementação, sendo que a mesma equipe pode ser responsável por um ou mais subsistemas.

Nestes ambientes de Projeto e Implementação são realizadas as atividades de especificação dos subsistemas; projeto dos subsistemas, tarefas e módulos; implementação e testes de módulos de software; confecção dos documentos; alterações do software e dos documentos durante o desenvolvimento e a integração do sistema; alterações do software e dos documentos durante a manutenção do sistema.

7.2.2.5 - Ambientes de Integração de Subsistemas

A Integração e Testes de um subsistema é realizada pela equipe de Projeto e Implementação que o desenvolveu, de acordo com um plano previamente aprovado. Para integrar e testar um dado subsistema, os outros subsistemas e interfaces devem ser substituídos por simuladores (*stubs*); já que seus desenvolvimentos são realizados com velocidades próprias, de maneira que um subsistema poderá chegar na fase de Integração e Testes antes que os outros subsistemas e/ou entidades externas sejam implementados. Os simuladores são desenvolvidos para estes ambientes sob a forma de produtos controlados, e devem possuir interfaces iguais às dos subsistemas e entidades externas que eles simulam.

Os testes realizados nestes ambientes devem ser fiscalizados pela Equipe de Integração do Sistema, já que são parte do conjunto de testes de Integração do Sistema.

7.2.2.6 - Ambiente de Integração do Sistema

As etapas de Integração e Testes do sistema devem ser realizadas pela Equipe de Integração do Sistema, de acordo com um plano de integração e testes aprovado pela Equipe de Sistema; sendo que para cada etapa de integração, a Equipe de Controle de Configuração deve montar uma versão do subsistema, usando produtos controlados. A Integração e Testes para a montagem do sistema deve ser realizada a partir dos subsistemas e entidades externas que já foram integrados e testados individualmente.

7.2.2.7 - Ambiente de Prototipação

Este ambiente deve ser usado quando uma Equipe de Projeto e Implementação precisar, por algum objetivo, desenvolver um *protótipo*. Por exemplo, para completar a especificação das interfaces do sistema com o usuário, para verificar a viabilidade de requisitos operacionais e de desempenho, etc...

Os *protótipos* desse ambiente não precisam ser controlados, ou seja, eles devem ser desenvolvidos e utilizados dentro desse mesmo ambiente, não participando da composição do sistema; evitando assim, a interferência dessas atividades em outras atividades que estão acontecendo paralelamente; como por exemplo, as atividades de Projeto e Implementação.

7.3 - Sistemas de Grande Porte : Considerações Finais

Devido à complexidade dos sistemas de grande porte sugerimos ao engenheiro que, se for necessário desenvolver sistemas desse tipo, procure suporte de profissionais especializados em Engenharia de Software.

Nesse caso, que envolve muito mais do que técnicas triviais de desenvolvimento e controle, acreditamos que só um profissional experiente de engenharia de software pode fornecer diretrizes de desenvolvimento e controle adequadas ao sistema que se quer desenvolver.

Para mais informações sobre sistemas de grande porte e sobre esquemas para controle do desenvolvimento, consultar Lindstrom (1993), Cunha (1993), Nakanishi (1993), Fairley (1985), Ghezzi (1985), Itami (1997).

CAPÍTULO 8

CONCLUSÃO

Neste trabalho foram propostas diretrizes para auxiliar Engenheiros a desenvolver sistemas de software de pequeno, médio, e grande porte, usando o Paradigma Clássico e o Paradigma da Orientação a Objetos. Este trabalho traçou, portanto, uma linha de desenvolvimento de software que vai desde a concepção do sistema até a fase de implementação e testes; indicando metodologias, técnicas, e cuidados a serem considerados durante o processo de desenvolvimento.

Como na maioria das vezes os sistemas de software gerados para dar apoio à Engenharia são sistemas de pequeno porte, esse tipo de sistema foi abordado com uma maior profundidade que os sistemas de médio e grande porte, e teve portanto seu processo de desenvolvimento mais enriquecido de detalhes e conseqüentemente mais completo.

As diretrizes propostas para o desenvolvimento de sistemas de médio e grande porte por sua vez, concentraram-se em um nível mais alto já que entendemos que para construir sistemas desse tipo, é necessário ter um pouco mais de conhecimento de Engenharia de Software, e seria aconselhável, pelo menos no caso de sistemas de grande porte, que se procurasse ajuda de um profissional da área de Engenharia de Software.

A preocupação em sugerir um caminho de desenvolvimento de sistemas de software para Engenheiros se deu à cada vez mais freqüente participação de profissionais de Engenharia em equipes de desenvolvimento de software, e principalmente à geração freqüente por parte de alguns Engenheiros, de software de uso pessoal para ajudar a otimizar seu trabalho.

Como o processo de desenvolvimento é na maior parte das vezes muito complexo para se realizar sem um controle de qualidade adequado, e somente um processo de

desenvolvimento cuidadoso pode minimizar os efeitos dessa complexidade, essas diretrizes ajudam os Engenheiros a conhecer um pouco mais sobre o processo de desenvolvimento, permitindo assim que eles possam construir mais facilmente, softwares apropriados para seus objetivos.

Para enriquecer o nosso universo de informações foram realizadas algumas entrevistas com Analistas e Engenheiros.

Dentre os Engenheiros entrevistados, alguns estavam envolvidos em projetos de sistemas de software de médio e grande porte, outros construíam apenas sistemas de pequeno porte, na maioria das vezes de uso pessoal.

As entrevistas realizadas com os Engenheiros nos permitiram detectar algumas dificuldades encontradas por eles para se construir sistemas de software; e nos permitiram detectar também o pouco uso das técnicas, metodologias e ferramentas que a Engenharia de Software oferece, durante o processo de desenvolvimento de seus softwares.

Essas entrevistas fizeram também com que decidíssemos incluir em nosso trabalho diretrizes para o desenvolvimento de sistemas de software abrangendo o Paradigma Clássico e o Paradigma da Orientação a Objetos, já que pudemos identificar dois perfis de Engenheiros que desenvolvem softwares :

1) aqueles que necessitam e se interessam em estar acompanhando as inovações na área de Engenharia de Software e

2) aqueles que não necessitam e nem se interessam em acompanhar essas mudanças, pois querem apenas melhorar o desenvolvimento de seus softwares dentro do que eles já conhecem.

As diretrizes para o desenvolvimento de software sob o ponto de vista do Paradigma Orientado para Objeto propostas, foram destinadas àqueles que queiram se integrar ao que está acontecendo de mais recente dentro da Engenharia de Software.

As diretrizes para o desenvolvimento de software sob o ponto de vista do Paradigma Clássico propostas, foram destinadas àqueles que não sentem necessidade e nem se interessam em aprender uma nova maneira de pensar e uma nova linguagem de programação, se interessando apenas em melhorar o desenvolvimento do seu software dentro do que ele já conhece.

Consideramos porém, que qualquer um dos caminhos apresentados nesta dissertação para o desenvolvimento de sistemas de software pode levar, se seguido corretamente, à construção de um sistema que atende a requisitos mínimos de qualidade e confiabilidade; ficando a critério do desenvolvedor escolher a maneira mais apropriada de desenvolvimento de acordo com o tipo de software que será desenvolvido, e de acordo com o conhecimento que ele têm e que está disposto a adquirir.

Dentre os Analistas entrevistados, a maioria estava ou esteve envolvida em desenvolvimentos de sistemas de médio e grande porte. Pudemos obter através dessas entrevistas muitas diretrizes práticas para o desenvolvimento de sistemas de pequeno, médio e grande porte, que foram de grande valia para o desenvolvimento desse trabalho.

Consideramos portanto, que a atividade de entrevista foi de grande importância para o desenvolvimento dessa dissertação, pois ela nos permitiu, acima de tudo, estabelecermos o conteúdo que a dissertação deveria abranger; baseado nas dificuldades encontradas durante o desenvolvimento de softwares descritas pelos engenheiros. Ela nos permitiu também através dos especialistas em engenharia de software, delinear as diretrizes gerais para se desenvolver sistemas de pequeno, médio e grande porte.

Acreditamos que com a prática e experiência em usar os processos de desenvolvimento aqui propostos, os Engenheiros serão capazes de começar a tomar suas próprias decisões no processo de desenvolvimento, e queremos enfatizar que os caminhos aqui propostos podem sofrer a inclusão ou desconsideração de algumas das etapas propostas, ou até mesmo a combinação de etapas que foram sugeridas para sistemas de pequeno, médio e grande porte.

O controle de versões e o controle de mudanças para sistemas de pequeno porte, embora não tenham sido abordados nos capítulos referentes ao desenvolvimento desse tipo de sistema, podem contribuir imensamente para o bom andamento do processo de desenvolvimento e de manutenção, ajudando a se ter uma melhor organização dos produtos sendo gerados.

Essas medidas podem, assim como o próprio processo de desenvolvimento, ser simplificadas para sistemas de pequeno porte, o que significa que pode-se realizar um controle de configuração menos formal que o realizado em sistemas de médio e grande porte. Por exemplo, assim que o software tiver sido implementado e testado pode-se gerar a primeira versão desse sistema. À medida que ele for sendo usado e mudanças se fizerem necessárias para corrigir erros, incorporar atualizações, e também para acomodar novas necessidades dos usuários, novas versões do sistema devem ser geradas. Uma versão é composta, nesse caso, pelos documentos e software gerados durante todas as fases do desenvolvimento daquele sistema.

Gostaríamos de dizer também que se após a especificação inicial do sistema, ainda houver dúvida em relação ao seu porte, o Engenheiro deve optar pelo processo de desenvolvimento proposto para o sistema mais complexo, pois consideramos ser melhor gastar um pouco mais de tempo na análise e projeto, mesmo que o sistema não necessite de tanto, do que fazer uma análise e projeto que sejam insuficientes para atender à complexidade do sistema sendo desenvolvido.

Assim sendo, independentemente do paradigma escolhido para o desenvolvimento, somente trabalhando a especificação do sistema, isolando funções críticas, documentando o processo de desenvolvimento, comentando seu código, testando amplamente os componentes individuais e as interações do sistema como um todo; estaremos trabalhando para que a qualidade do software aumente, e conseqüentemente aumente também a sua confiabilidade.

Na Figura 8.1, Figura 8.2, Figura 8.3 e Figura 8.4, são mostrados os principais passos que compõe as metodologias sugeridas para o desenvolvimento de sistemas de pequeno e médio porte sugeridas nesta dissertação, bem como os produtos que vão sendo gerados à medida que essas metodologias vão sendo aplicadas.

Uma proposta de trabalho futuro seria validar os processos de desenvolvimento aqui propostos através da prática, ou seja, submeter as diretrizes e sugestões aqui propostas a engenheiros, para que eles desenvolvessem seus softwares seguindo os caminhos aqui traçados, e avaliassem a utilidade, a praticidade e a clareza com que foram delineados, para que pudéssemos melhorá-los e enriquecê-los.

Outra proposta de trabalho futuro seria delinear caminhos de desenvolvimento de software que explorassem a reutilização dos componentes dos softwares gerados, e a reutilização das etapas do desenvolvimento. Ou seja, incorporar ao sistema sendo desenvolvido resultados de análises, projetos e codificações anteriores.

Uma última proposta de trabalho futuro seria o desenvolvimento de um Sistema de Apoio aos Engenheiros no Desenvolvimento de Sistemas de Software. A idéia seria construir um sistema iterativo que auxiliasse o desenvolvimento de sistemas de software com planejamento, porém sem necessidade de se estender na literatura. O sistema de apoio solicitaria ao usuário informações que permitiriam com que o sistema analisasse o tipo de software que seria desenvolvido, podendo assim, sugerir de acordo com as características desse software; as metodologias, ferramentas, técnicas e procedimentos

de trabalho que mais se adequassem à situação, direcionando assim, o trabalho de desenvolvimento.

FASES	ANÁLISE		PROJETO E IMPLEMENTAÇÃO		
	DEFINIÇÃO	FORMALIZAÇÃO	PROJETO DO SOFTWARE	CODIFICAÇÃO	TESTES
PRODUTOS	Texto Informal	Diagrama de Contexto + Diagrama E-R	Diagrama de Estrutura dos Módulos + Especificação dos Módulos + Descrição dos Módulos em Português Estruturado + Plano de Testes (Estratégia para Implementação e Testes e Casos de Teste de Aceitação) + Tabelas do Modelo Relacional	Documentação Externa dos Módulos + Documentação Interna dos Programas + Programas + Bases de Dados	Casos de Teste de Unidade + Plano de Testes Atualizado

Fig. 8.1 - Resumo da Metodologia Sugerida para o Desenvolvimento de Sistemas de Pequeno Porte Usando o Paradigma Clássico.

FASES	ANÁLISE		PROJETO E IMPLEMENTAÇÃO		
	DEFINIÇÃO	FORMALIZAÇÃO	PROJETO DO SOFTWARE	CODIFICAÇÃO	TESTES
PRODUTOS	Texto Informal	Diagrama de Contexto + Diagrama de Fluxo de Dados + Mini-Especificação para os Processos	DFD do Sistema Decomposto em Tarefas + Diagrama de Tarefas + Plano de Testes (Estratégia para Implementação e Testes e Casos de Teste de Integração) + Seguir o Desenvolvimento de cada Tarefa como sendo um Sistema de Pequeno Porte		

Fig. 8.2 - Resumo da Metodologia Sugerida para o Desenvolvimento de Sistemas de Médio Porte Usando o Paradigma Clássico.

FASES	ANÁLISE		PROJETO E IMPLEMENTAÇÃO		
	DEFINIÇÃO	FORMALIZAÇÃO	PROJETO DO SOFTWARE	CODIFICAÇÃO	TESTES
PRODUTOS	Texto Informal	Diagrama de Contexto + Diagrama de Classe&Objeto do Modelo Conceitual + Roteiros de Cenários	Diagrama de Classe&Objeto do Modelo Lógico + Especificação para as Classes (Especificação para os Serviços e Atributos) + Plano de Testes (Estratégia para Implementação e Testes e Casos de Teste de Cenário e Casos de Teste de Integração de Cenários) + Tabelas do Modelo Relacional	Documentação Externa e Interna dos Serviços + Documentação Interna e Externa das Classes + Códigos + Bases de Dados	Casos de Teste dos Serviços + Plano de Testes Atualizado

Fig. 8.3 - Resumo da Metodologia Sugerida para o Desenvolvimento de Sistemas de Pequeno Porte Usando o Paradigma da Orientação a Objetos.

FASES	ANÁLISE		PROJETO E IMPLEMENTAÇÃO		
	DEFINIÇÃO	FORMALIZAÇÃO	PROJETO DO SOFTWARE	CODIFICAÇÃO	TESTES
PRODUTOS	Texto Informal	Diagrama de Contexto + Diagrama de Classe&Objeto do Modelo Conceitual incluindo Assuntos	Serviços de Tempo Real Transformados em Tarefas + Visões de Cenários + Plano de Testes (Estratégia para Implementação e Testes e Casos de Teste Cenário, de Integração de Cenários, e de Interação com Outros Sistemas e Dispositivos) + Seguir o Desenvolvimento de cada Assunto como sendo um Sistema de Pequeno Porte		

Fig. 8.4 - Resumo da Metodologia Sugerida para o Desenvolvimento de Sistemas de Médio Porte Usando o Paradigma da Orientação a Objetos.

REFERÊNCIAS BIBLIOGRÁFICAS

Arakaki, R.; Arakaki, J.; Angerami, P. M.; Aoki, O. L.; Salles, D. S. **Fundamentos de programação C :** Técnicas e Aplicações. Rio de Janeiro: LTC - Livros Técnicos e Científicos, 1990. 458p.

Atkinson, M.; et al. **The object-oriented database manifesto.** [online]. <http://almond.srv.cs.cmu.edu/user/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html>. May 1998.

Barkakati, N. **Visual C++ guia de desenvolvimento avançado.** Rio de Janeiro: Berkeley Brasil Editora, 1997. 1210p.

Bell, R. Choosing Tools for Analysis and Design. **IEEE Software**, v. 11, n. 3, p. 121-125, May 1994.

Camarão, P. C. B. **Glossário de informática.** Rio de Janeiro: LTC- Livros Técnicos e Científicos, 1989. 730 p.

Chioccarello, R. B. M. **Uma metodologia para o desenvolvimento de sistemas de informação baseada no paradigma da orientação a objetos.** São José dos Campos, 168p. Dissertação (Mestrado em Engenharia Eletrônica de Computação) - Instituto Tecnológico de Aeronáutica - ITA, 1997.

Coad, P.; Yourdon, E. **Análise baseada em objetos.** Rio de Janeiro: Editora Campus, 1992. 225p.

Coad, P.; Yourdon, E. **Projeto baseado em objetos.** Rio de Janeiro: Editora Campus, 1993A. 195p.

Coad, P.; Nicola, J. **Object-oriented programming.** New Jersey: Prentice Hall, 1993B. 582p.

Coad, P. **Object models :** Strategies, Patterns, & Applications. New Jersey: Prentice Hall, 1997. 515 p.

Cohen, S.S.; Sharble, R. C. The Object-Oriented Brewery : A Comparison of Two Object-Oriented Development Methods. **ACM SIGSOFT**, v. 18, n. 2, p. 60-73, April 1993.

Cunha, J. B.; Nakanishi, T. O Controle do Desenvolvimento e Manutenção de Sistemas de Software. In: Congresso Ibero-Latino-Americano de Métodos Computacionais em Engenharia, 14., São Paulo, 1993. **Anais.** São Paulo: Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT, 1993. v. II, p. 1314-1323.

Cunha, J. B. **Metodologias orientadas a objeto :** notas de aula dadas no curso de Metodologias Orientadas a Objeto no INPE - Instituto Nacional de Pesquisas Espaciais. São José dos Campos, 3° trimestre, 1997. Manuscrito.

Dai, K.; Nagata, M. Object-Oriented Analysis Using Scenarios. **Systems and Computers in Japan**, v. 28, n. 6, p. 40-48, June 1997.

Deadline. In: **Michaelis :** pequeno dicionário - inglês - português, português - inglês. São Paulo: Melhoramentos, 1994. 792p.

Deux, O.; et al. The O2 System. **Communications of the ACM**, v. 34, n. 10, p. 34-48, October 1991.

Emden, M. H. Structured Inspections of Code. **Software Testing Verification and Reliability**, v. 2, p. 133-153, 1992.

Fairley, R. E. **Software engineering concepts.** USA: McGraw-Hill, 1985. 364p.

Fragomeni, A. H. **Dicionário enciclopédico de informática.** Rio de Janeiro: Editora Campus, 1986. 727p.

Galante, T. P.; Pow, E. M. **Inglês para processamento de dados.** São Paulo: Editora Atlas, 1986. 166p.

Gane, C.; Sarson, T. **Análise estruturada de sistemas.** Rio de Janeiro: LTC- Livros Técnicos e Científicos Editora, 1983. 255p.

Ghezzi, C.; Jazayeri, M. **Conceitos de linguagens de programação.** Rio de Janeiro: Editora Campus, 1985. 300p.

Gomaa, H. A Software Design Method for Real-Time Systems. **Communications of the ACM**, v. 27, n. 9, p. 938-949, September 1984.

Grogono, P. **Programming in pascal.** USA: Addison-Wesley, 1978. 359p.

Itami, S. N. **Proposta de um esquema de controle de elementos de software para abordagem orientada a objetos.** 132p. (INPE-6394-TDI/610). Dissertação (Mestrado em Computação Aplicada) - Instituto Nacional de Pesquisas Espaciais, 1997.

Joch, A. How Software Doesn't Work - Nine Ways to Make your Code more Reliable. **Byte Magazine**, v. 20, n. 12, p. 49-60, December 1995.

Jorgensen, P. C. Object-Oriented Integration Testing. **Communications of the ACM**, v. 37, n. 9, p. 30-38, September 1994.

Khan, E.; Al-Aáli, M.; Girgis, M. R. Object-Oriented Programming for Structured Procedural Programmers. **IEEE Computer**, v. 28 , n.10, p. 48-57, October 1995.

Kirani, S. Method Sequence Specification and Verification of Classes. **JOOP - Journal of Object-Oriented Programming**, v.7, n.6, p. 28-38, October 1994.

Korth, H. F.; Silberschatz, A. **Sistemas de banco de dados.** São Paulo: McGraw-Hill, 1989. 582p.

Kung, D.; et al. Developing an Object-Oriented Software Testing and Maintenance Environment. **Communications of the ACM**, v. 38, n. 10, p. 75-87, October 1995.

Ledgard, H. F. **Programming proverbs for FORTRAN programmers.** New Jersey: 1975. 130p.

Lindstrom, D. R. Five Ways to Destroy a Development Project. **IEEE Software**, v. 10, n. 5, p. 55-58, September 1993.

Militello, K. Pare de Jogar Dinheiro no Lixo. **Revista Exame Informática**, v. 12, n. 135, p. 39-48, Junho 1997.

Monteiro, M. **Introdução à organização de computadores.** Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora, 1992. 312p. Cap 6: Execução de programas, p. 106.

Nakanishi, T.; Cunha, J. B. S. Quality and Productivity : Control of Software System Elements. In: Structural Optimization 93 - The World Congress on Optimal Design of Structural Systems, Rio de Janeiro, 1993. **Proceedings.** Rio de Janeiro: Coppe/UFRJ - Universidade Federal do Rio de Janeiro, 1993. v. 2, p. 483-490.

Nakanishi, T. **Engenharia de software :** apostila usada durante o curso de Engenharia de Software no INPE - Instituto Nacional de Pesquisas Espaciais. São José dos Campos, 2° trimestre, 1995A.

Nakanishi, T. **Engenharia de software :** notas de aula dadas no curso de Engenharia de Software no INPE - Instituto Nacional de Pesquisas Espaciais. São José dos Campos, 2° trimestre, 1995B. Manuscrito.

Norman, M.; Bloor, R. **The 1996 object/relational summit recommended reading - to universally serve where no database has served before.** [Online] <http://www.dbsummit.com/reading.htm>. May 1998.

Onoma, A. K.; et al. Management of Object Oriented Development Based on Ranked Use Cases. In : Annual International Computer Software & Application Conference - COMPSAC'97, 21., Bethesda, 1997. **Proceedings.** Washington: IEEE Computer Society, 1997. p. 246-251.

Page-Jones, M. **Projeto estruturado de sistemas.** São Paulo: Mc Graw-Hill, 1988. 396p.

Paladino, E. **Novo dicionário técnico de informática com termos e expressões :** Inglês / Português. Rio de Janeiro: Ciência Moderna e Computação Ltda., 1986. 459p.

Pressman, R. S. **Engenharia de software.** São Paulo: Mc Graw-Hill, 1995. 1056p.

Rout, T. P. Quality, Culture and Education in Software Enginnering. **The Australian Computer Journal,** v. 24, n. 3, p. 86-91, August 1992.

Rumbaugh, J.; et al. **Modelagem e projeto baseados em objetos.** Rio de Janeiro, Editora Campus, 1994. 652p.

Selltiz, C.; et al. **Métodos de pesquisa nas relações sociais.** São Paulo: Editora Herder - Editora da Universidade de São Paulo, 1972. 678p.

Setzer, V. W. **Projeto lógico e projeto físico de banco de dados.** Belo Horizonte: UFMG, 1986. 289p.

Shiller, L. **Excelência em software.** São Paulo: Makron Books Editora Ltda., 1992. 295p.

Shoup, T. E. **A Pratical guide to computer methods for engineers.** EUA: Prentice-Hall, 1979. 255p.

Staa, A. **Engenharia de programas.** Rio de Janeiro: LTC- Livros Técnicos e Científicos, 1987. 293p.

Stonebraker, M. Third-Generation Database Manifesto. **SIGMOD Record (ACM Special Interest Group on Management of Data)**, v. 19, n. 3, p. 31-44, September 1990.

Teorey, T. J.; Fry J. P. **Design of database structures.** New Jersey: Prentice-Hall, 1982. 492p.

Test Drive. In: **Longman - dictionary of english language and culture.** England: Longman Group, 1992. 1528p.

Ullman, J. D. **Principles of database systems.** Rockville - EUA: Computer Science Press, 1980. 379p.

Yourdon, E. **Análise estruturada moderna.** Rio de Janeiro: Editora Campus, 1990. 836p.

Wirfs-Brock, R.; Wilkerson, B.; Wiener, L. **Designing object-oriented software.** New Jersey: Prentice Hall, 1990. 341p.

APÊNDICE

ENTREVISTAS REALIZADAS

As entrevistas são uma fonte muito importante para se obter informações a respeito do domínio do problema sendo estudado; pois como entrevistador e entrevistado estão presentes no momento em que as perguntas são apresentadas e respondidas, existe oportunidade para uma maior flexibilidade na obtenção dessas informações, havendo a possibilidade de se repetir perguntas, ou apresentá-las de outro modo para que se possa ter certeza de que são compreendidas, ou fazer outras perguntas a fim de esclarecer o sentido de alguma resposta (Selltiz,1972).

Levando-se em conta que o método mais simples para se obter "fatos" a respeito do que se está estudando, é procurar diretamente as pessoas que estejam em condições de nos fornecer as informações desejadas; é que decidiu-se fazer algumas entrevistas com engenheiros e analistas, a fim de enriquecer o nosso universo de informações.

Foram entrevistados cinco engenheiros, alguns estavam envolvidos em projetos de sistemas de software de médio e grande porte, outros construíam apenas sistemas de pequeno porte, na maioria das vezes de uso pessoal. O mais interessante é que todos eles se utilizavam muito pouco dos recursos da Engenharia de Software para ajudar em seus trabalhos.

Os que estavam envolvidos em projetos maiores sentiam falta de mecanismos que ajudassem a controlar o desenvolvimento do sistema, e de mecanismos que permitissem se ter uma visão geral do sistema antes de partir para a implementação; sendo que, em sua maioria, atribuíram o atraso e as dificuldades encontradas no desenvolvimento de seus projetos, à falta de uma abordagem de engenharia de software desde a concepção do sistema.

Os que construíam sistemas menores usavam, em sua maioria, alguns fundamentos básicos da engenharia de software na construção de seus sistemas, porém não tinham uma idéia muito precisa no que isso poderia ajudá-los.

Foram entrevistados também cinco especialistas em engenharia de software (analistas) que, em sua maioria, estavam ou estiveram envolvidos em desenvolvimentos de sistemas de médio e grande porte. Pudemos obter através deles muitas diretrizes práticas para o desenvolvimento de sistemas de pequeno, médio e grande porte, que foram de grande valia para o desenvolvimento desse trabalho.

Considerando-se que a situação da entrevista raramente é uniforme de uma entrevista para outra, decidimos formular algumas perguntas chaves que serviram de orientação para o trabalho. Essas perguntas puderam ser mais ou menos exploradas de acordo com a experiência do entrevistado no assunto em questão. Dessa maneira, pudemos garantir uma certa padronização de uma entrevista para outra, o que pode ser relativamente importante quando se deseja comparar resultados.

Consideramos portanto, que a atividade de entrevista foi de grande importância para o desenvolvimento dessa dissertação, pois ela nos permitiu, acima de tudo, estabelecermos o conteúdo que a dissertação deveria abranger; baseado nas dificuldades encontradas durante o desenvolvimento de softwares descritas pelos engenheiros. Ela nos permitiu também através dos especialistas em engenharia de software, delinear as diretrizes gerais para se desenvolver sistemas de pequeno, médio e grande porte.

A seguir são apresentadas as perguntas básicas que foram feitas aos engenheiros durante as entrevistas, e uma análise dos resultados gerais obtidos.

1- Qual a sua formação acadêmica ?

TABELA A.1- FORMAÇÃO ACADÊMICA DOS ENGENHEIROS

Engenheiro Eletrônico	**Outros**
60 %	40 %

2- Qual sua experiência no desenvolvimento de softwares ?

TABELA A.2- EXPERIÊNCIA DOS ENGENHEIROS NO DESENVOLVIMENTO DE SOFTWARES

Experiência em Programação	**Experiência em Análise e Projeto**
80 %	20 %

3- Qual o tamanho, em termos de linhas de código, dos sistemas que costuma desenvolver?

TABELA A.3- TAMANHO DOS SISTEMAS DESENVOLVIDOS PELOS ENGENHEIROS

Até 5.000 Linhas	**5.000 - 10.000 Linhas**
80 %	20 %

4- Qual a linguagem de programação que costuma utilizar ?

TABELA A.4- LINGUAGEM DE PROGRAMAÇÃO UTILIZADA PELOS ENGENHEIROS

Fortran	**C e C++**
40 %	60 %

5- Segue alguma norma para o desenvolvimento de software ?

TABELA A.5- UTILIZAÇÃO DE NORMAS PARA O DESENVOLVIMENTO DE SOFTWARES PELOS ENGENHEIROS

Não Utilizam	**Utilizam**
60 %	40 %

6- Usa alguma ferramenta, técnica, ou metodologia para ajudar no desenvolvimento do software ?

TABELA A.6- UTILIZAÇÃO DE FERRAMENTAS, TÉCNICAS E METODOLOGIAS PELOS ENGENHEIROS

Não Usa	**Usa**
60 %	40 %

7- Qual o critério (se usar algum) que usa para quebrar o programa em rotinas ?

TABELA A.7- UTILIZAÇÃO DE CRITÉRIOS PARA QUEBRAR O PROGRAMA EM ROTINAS PELOS ENGENHEIROS

Procura Identificar as Funções do Programa	**Não Usa Critérios**
40 %	60 %

8- Usa alguma técnica para representar a lógica de cada módulo, ou parte direto para a implementação ?

TABELA A.8- UTILIZAÇÃO DE TÉCNICAS PARA REPRESENTAR A LÓGICA DOS MÓDULOS PELOS ENGENHEIROS

Fluxograma	**Português Estruturado**	**Parte Direto para a Implementação**
20 %	20 %	60 %

9- Usa algum critério para implementar as interfaces entre as rotinas (se usa, qual) ?

TABELA A.9- UTILIZAÇÃO DE CRITÉRIOS PARA IMPLEMENTAR AS INTERFACES ENTRE AS ROTINAS PELOS ENGENHEIROS

Procura Trocar o Mínimo de Informações Possíveis entre as Rotinas	**Não Usa**
60 %	40 %

10- Costuma fazer a modelagem dos dados ?

TABELA A.10- UTILIZAÇÃO DE TÉCNICAS DE MODELAGEM DE DADOS PELOS ENGENHEIROS

Sim	**Não**
20 %	80 %

11- Como são mostrados os resultados ?

TABELA A.11- FORMATO DOS RESULTADOS GERADOS PELOS ENGENHEIROS

Gráficos, Animação	**Tabelas, Relatórios**
60 %	40 %

12- Tem problemas com relação à eficiência dos programas que constrói ?

TABELA A.12- EFICIÊNCIA DOS PROGRAMAS CRIADOS PELOS ENGENHEIROS

Estão Satisfeitos com os Programas que Constróem	**Não Estão Satisfeitos**
40 %	60 %

13- Quais são as dificuldades que tem encontrado (se tiver alguma) ao se construir um software ?

TABELA A.13- DIFICULDADES ENCONTRADAS PELOS ENGENHEIROS PARA SE CONSTRUIR SOFTWARES

Nenhuma Dificuldade	**Dificuldades na Construção e Manutenção**
40 %	60 %

14- Que tipo de auxilio gostaria de ter ?

TABELA A.14- AUXÍLIO QUE OS ENGENHEIROS GOSTARIAM DE TER PARA DESENVOLVER SEUS SOFTWARES

Auxílio na Concepção do Sistema	**Auxílio na Programação**
60 %	40 %

15- O que acha do trabalho de dissertação que estamos propondo ?

TABELA A.15- O QUE OS ENGENHEIROS PENSAM SOBRE A PROPOSTA DE DISSERTAÇÃO

Não Vê interesse	**Muito Útil**
0 %	100 %

A seguir são apresentadas as perguntas básicas que foram feitas para os analistas durante as entrevistas, e uma análise dos resultados gerais obtidos.

1- Qual a sua formação acadêmica ?

TABELA A.16- FORMAÇÃO ACADÊMICA DOS ANALISTAS

Graduação em Análise de Sistemas	**Graduação em Engenharia e Especialização em Análise de Sistemas**
40 %	60 %

2- Qual a sua experiência no desenvolvimento de softwares ?

TABELA A.17- EXPERIÊNCIA DOS ANALISTAS NO DESENVOLVIMENTO DE SOFTWARES

Experiência no Desenvolvimento de Sistemas de Médio e Grande Porte	**Nenhuma ou Pouca Experiência em Sistemas desse Tipo**
80 %	20 %

3- Quais as primeiras observações que faz a respeito do sistema que se quer desenvolver, para delinear suas características principais ?

TABELA A.18- PRIMEIRAS OBSERVAÇÕES FEITAS PELOS ANALISTAS PARA DELINEAR AS CARACTERÍSTICAS PRINCIPAIS DE UM SISTEMA

Observa o Tipo de Aplicação à que se Refere e Também o Objetivo do Usuário em Relação ao Software	**Outros**
100 %	0 %

4- Quais os critérios que usa para identificar o porte do sistema que será desenvolvido ?

TABELA A.19- CRITÉRIOS UTILIZADOS PELOS ANALISTAS PARA IDENTIFICAR O PORTE DE UM SISTEMA

Leva em Consideração o Número de Módulos Gerais que Comporão o Sistema, e Características de Complexidade	**Outros**
100 %	0 %

5- Como prevê o tempo que será necessário para o desenvolvimento, bem como o número de pessoas envolvidas ?

TABELA A.20- CRITÉRIOS UTILIZADOS PELOS ANALISTAS PARA MENSURAR O TEMPO E O NÚMERO DE PESSOAS NECESSÁRIOS PARA DESENVOLVER UM SISTEMA

Leva em Consideração as Perspectivas do Usuário em Relação ao Sistema, Aspectos de Manutenção e Qualidade, Necessidade ou Não de Treinamento de Pessoal, Número de Módulos Gerais, Características de Complexidade	**Outros**
100 %	0 %

6- Quando opta por aproveitar o sistema atual ?

TABELA A.21- CRITÉRIOS UTILIZADOS PELOS ANALISTAS PARA DECIDIR SE APROVEITAM OU NÃO O SISTEMA ATUAL

Verifica a Satisfação do Usuário em Relação ao Sistema Atual e a Vantagem de se Aproveitar o Software Existente	**Outros**
100 %	0 %

7- Qual o critério que usa para decidir se deverá ser feito o modelo dos dados ?

TABELA A.22- CRITÉRIOS UTILIZADOS PELOS ANALISTAS PARA DECIDIR SE DEVE SER FEITO O MODELO DE DADOS

Sempre Deve ser Feito	**Procura Identificar o Número de Entidades de Dados Diferentes e o Volume dos Dados**
20 %	80 %

8- Como identificar um sistema de tempo real, e quais os cuidados que se deve ter com sistemas desse tipo ?

TABELA A.23- COMO OS ANALISTAS IDENTIFICAM UM SISTEMA DE TEMPO REAL

Procura Analisar se o Sistema Tem que Fornecer uma Resposta em Tempo Hábil	**Outros**
100 %	0 %

9- Que critérios usa para decidir entre a abordagem tradicional e a abordagem orientada para objetos para o desenvolvimento de sistemas ?

TABELA A.24- CRITÉRIOS UTILIZADOS PELOS ANALISTAS PARA DECIDIR ENTRE A ABORDAGEM TRADICIONAL E A ABORDAGEM ORIENTADA PARA OBJETO

Procura Ver a Experiência da Equipe de Trabalho e o Nível de Ferramentas e Técnicas Disponíveis	**Outros**
100 %	0 %

10- Como escolhe a linguagem de programação que será usada ?

TABELA A.25- CRITÉRIOS UTILIZADOS PELOS ANALISTAS PARA DECIDIR QUAL SERÁ A LINGUAGEM DE PROGRAMAÇÃO UTILIZADA

Procura Ver se a Linguagem Atende à Metodologia Adotada para o Desenvolvimento, se a Linguagem Tem alguma Ferramenta de Apoio, e a Experiência da Equipe	**Outros**
100 %	0 %

11- O que acha do trabalho de dissertação que estamos propondo ?

TABELA A.26- O QUE OS ANALISTAS PENSAM SOBRE A PROPOSTA DE DISSERTAÇÃO

Não Vê Interesse	**Muito Útil**
0 %	100 %

GLOSSÁRIO

Alocação Dinâmica - A alocação dinâmica é o processo de alocar espaço de dados na memória do computador em tempo de execução, ou seja, um programa em execução pode, sempre que necessário e se houver a disponibilidade de espaço em memória, solicitar a alocação do espaço e utilizar convenientemente os dados (Arakaki,1990).

Área de Dados Globais - (Área Comum ou Common Area) Em programação, seção de controle que se emprega para reservar uma área da memória principal à qual outros módulos podem fazer referência (Fragomeni,1986).

Argumentos - Parâmetro que se passa entre um programa e um sub-programa ou procedimento nele embutidos (Fragomeni,1986).

Arquivos de Dados - São um conjunto rígido ou semi-rígido de estruturas (por exemplo, campos de tamanho fixo dentro de registros) (Fragomeni,1986).

Arquivos Include - São arquivos que são incluídos no código fonte pelo pré-processador antes que ele submeta o código fonte ao compilador (Arakaki,1990).

Array - Arranjo, Conjunto ou Matriz. Maneira pela qual estão dispostos, em linhas e colunas, os elementos (Paladino,1986).

Backup - Reserva. Termo dado ao equipamento, programa ou procedimento que serve de substituto ou alternativa (Paladino,1986).

Batch Processing - Processamento tradicional, normalmente executado em equipamento de grande porte. A entrada e o processamento dos dados são realizados por lotes periódicos e contrasta com o modo *on-line* (Camarão,1989).

Breadth-First - Primeiramente pela largura (Pressman,1995).

Buffer - Área de memória para armazenamento temporário, durante transferências de dados de uma parte do sistema para outra (Paladino,1986).

Byte - Conjunto de oito "bits" (dígito binário) usados para representar um dado ou uma informação (Galante,1986).

Case - O *Case* é uma generalização do *If*. Ele faz com que o processamento execute uma de muitas ações possíveis, de acordo com o valor de uma expressão (Grogono,1978).

Cluster - Grupo (Paladino,1986).

Compilador - Programa que converte um programa escrito em linguagem de alto nível, em um programa em linguagem de máquina, ou seja, gera um programa objeto. É um programa que realiza a compilação de outro programa (Paladino,1986).

CPU - Unidade Central de Processamento. Parte do sistema de computação que controla a interpretação e execução das instruções. Abreviatura de "Central Processing Unit" (Paladino,1986).

Deadline - Último prazo para fazer algo, prazo de entrega (Melhoramentos,1994).

Default - Revelia. Valor ou atributo assumido, desapercebidamente, quando não há outra especificação (Paladino,1986).

Depuração (Debug) - Detectar, localizar e corrigir erros em um programa (Galante,1986).

Depth-First - Primeiramente pela profundidade (Pressman,1995).

Dicionário de Dados - Listagem organizada de todos os elementos de dados que são pertinentes ao sistema, com definições precisas e rigorosas, de forma que tanto o usuário como o analista de sistemas tenham uma compreensão comum das entradas, das saídas, dos componentes dos depósitos de dados e até mesmo dos cálculos intermediários (Yourdon,1990).

Drive - (Disk Storage Drive) **(1)** Mecanismo que movimenta e controla os movimentos do disco magnético; **(2)** Dispositivo giratório de que estão dotadas as unidades de discos magnéticos. Em seu movimento de rotação em torno do eixo, permite através das cabeças de leitura e gravação, as operações de leitura/gravação (Fragomeni,1986).

Else - Caso contrário. Uma cláusula que se aplica quando a condição indicada por uma declaração condicional não é verídica (Paladino,1986).

Escalonamento - Distribuição de cargas de trabalho no interior de um computador ou ação de escalonar uma tarefa para sua execução (Fragomeni,1986).

Event Flag - Sinalização de Evento. Um indicador usado para sinalizar a ocorrência de um evento (Paladino,1986).

Fila de Mensagens - Fila de mensagens que esperam o processamento ou o momento de sua transmissão. Geralmente coloca as mensagens para entrada no sistema na ordem adequada e estabelece a seqüência de saída (Fragomeni,1986).

Flag - Sinal. **(1)** Indicador com dois valores, autorizando ou não uma ramificação; **(2)** bits ou caracteres marcando o início ou o fim de um bloco de informação; **(3)** Um indicador usado para sinalizar a ocorrência de alguma condição (Paladino,1986).

Hardware - Partes eletrônicas de um sistema de computador (Paladino,1986).

If - Se. Uma declaração condicional segundo a qual são executadas diferentes seqüências de procedimentos, de acordo com a veracidade da condição (Paladino,1986).

Layout - Organização, disposição. Plano ou desenho global de um projeto (Paladino,1986).

Linkagem - A linkagem é o processo de combinação dos diversos programas requeridos (cuja chamada foi indicada pelo programa de aplicação), como também dos diversos módulos do próprio programa de aplicação, em um único código, completo e pronto para ser executado (Monteiro,1992).

Loop - Ciclo, Laço. Uma seqüência de instruções em um programa, executada repetitivamente, até que uma certa condição ocorra (Paladino,1986).

Macro - É um símbolo que pode ser definido para ser igual a um código C++. Pode-se então usar esse símbolo sempre que se quiser usar aquele código no programa, e quando o código fonte é pré-processado, cada ocorrência do nome da macro será substituída pela definição (Barkakati,1997).

Métrica - Métricas de software referem-se a uma ampla variedade de medidas de software de computador. Dentro do contexto de gerenciamento de projetos de software, estamos preocupados primeiramente com métricas de produtividade e qualidade, ou seja, medidas do resultado do desenvolvimento de software como uma função do esforço aplicado, e medidas da "adequação ao uso" do resultado que é produzido (Pressman,1995).

Multiprocessamento - **(1)** Modo de operação que permite o processamento paralelo por dois ou mais processadores de um multiprocessador; **(2)** Processamento paralelo (Fragomeni,1986).

Multitarefa - Modo de operação que permite a execução concorrente ou intercalada de duas ou mais tarefas, em um computador (Fragomeni,1986).

On-Line Processing - Técnica de operação em que a transmissão de uma informação é enviada no mesmo instante em que é digitada no transmissor (Camarão,1989).

Open/Close - Abertura/Fechamento de arquivo. Inicialização das operações de entrada e saída de um arquivo./ Terminação das operações de entrada e saída de um arquivo (Paladino,1986).

Overflow - Estouro, Excesso. Quantidade que ultrapassa a capacidade de armazenamento de um registro (Paladino,1986).

Parâmetros - **(1)** Variável à qual se atribui um valor constante determinado em cada caso particular e que, eventualmente, identifica esse caso; **(2)** Constante ou variável que permanece sem alterar seu valor durante um determinado cálculo; **(3)** Característica ou atributo do sistema, que tem um único valor em todas as fases previsíveis de operação, mas pode alterar-se quando diferentes alternativas forem usadas; **(4)** Áreas que são passadas a uma rotina de módulo, e cujo valor pode ser diferente em instantes distintos da execução de um programa (Fragomeni,1986).

Performance - Desempenho. O desempenho é um dos principais fatores na análise da produtividade total de um sistema (Paladino,1986).

Ponteiro - Um ponteiro é uma variável cujo conteúdo corresponde a um endereço de dados em vez de ser o próprio dado. Com isso, pode-se utilizar um ponteiro para apontar tipos de dados diferentes, isto é, mudando o conteúdo do ponteiro, pode-se manipular (ler, atribuir ou mudar) dados em vários locais (Arakaki,1990).

Procedimento ou Procedure - Função geral executada como uma subrotina e fazendo parte das linguagens de alto nível (Fragomeni,1986).

Processamento Distribuído - Processamento de dados no qual se dividem fisicamente os recursos computacionais de uma organização, de modo que fiquem tanto geográfica quanto organizacionalmente o mais próximo possível da aplicação (Fragomeni,1986).

Programa Interpretador - Programa Processador de linguagem que traduz instrução por instrução de um programa-fonte, executando-as conforme vão sendo traduzidas (Fragomeni,1986).

Programa Principal - Programa que define as linhas gerais do procedimento que deve ser efetuado com os dados, e cuja função é completada com sub-rotinas, às quais pode ceder o controle (Fragomeni,1986).

Protótipo - Modelo do software que serve como mecanismo para identificar e refinar os requisitos do software. Ele pode assumir três formas : **(1)** um protótipo que retrata a interação homem-máquina de uma forma que capacita o usuário a entender quanta interação ocorrerá; **(2)** um protótipo que implementa algum subconjunto da função exigida do software; **(3)** um programa existente que executa parte ou toda a função desejada, mas que tem outras características que serão melhoradas em um novo esforço de desenvolvimento (Pressman,1995).

Registro de Ativação - Este registro contém toda a informação necessária à execução da unidade, incluindo, entre outras coisas, os objetos associados às variáveis locais de uma ativação da unidade. A posição relativa de um objeto no registro de ativação é chamada "deslocamento". Para acessar um objeto, o processador pode usar o endereço inicial do registro de ativação que contém o objeto e o deslocamento do objeto (Ghezzi,1985).

Run-Time - Tempo durante o qual o programa está sendo rodado (Galante,1986).

Sistema de Tempo Compartilhado - Sistema cujo tempo de computador é compartilhado entre várias tarefas que têm prioridades diferentes (Paladino,1986).

Sistema Gerenciador de Banco de Dados - Um sistema gerenciador de banco de dados consiste numa coleção de dados inter-relacionados e um conjunto de programas para acessar esses dados. A coleção de dados é comumente referenciada como banco de dados, que contem informação de um empreendimento. O principal objetivo desse tipo de sistema é proporcionar um ambiente, conveniente e eficiente, para retirar e armazenar informações no banco de dados (Korth,1989).

Sistemas Operacionais - Software que controla e supervisiona todas as operações internas de um computador (Paladino,1986).

Stub - Um stub é um módulo cujas funções são rudimentares. É usado simplesmente para simular um módulo verdadeiro que ainda não tenha sido implementado (Page-Jones,1988).

String - Strings são matrizes de caracteres com um caracter "null" marcando seu final. Quando uma string é declarada o compilador reserva espaço suficiente na memória para armazenar a string e o caracter terminador "null" (Barkakati,1997).

Sub-Rotina - **(1)** Conjunto seqüencial de sentenças ou declarações que pode ser usado em um ou mais programas de computador e em um ou mais pontos de um mesmo programa; **(2)** Parte de um programa destinada a ter um tratamento diferente (por exemplo, mais freqüente) das partes que a cercam. De um modo geral, é um algoritmo bastante autônomo para merecer, seja ser escrito à parte, seja ser utilizado em condições diferentes do restante do programa. Exemplo: Sub-rotina de menu, que pode ser chamada muitas vezes no mesmo programa (Fragomeni,1986).

Switch - Chave, Interruptor. Um símbolo ou um dispositivo que pode modificar a seqüência de execução de um programa (Paladino,1986).

Tempo de Resposta - O tempo entre o apertar da tecla "entra" no terminal e o início da resposta do computador aparecendo na tela do terminal (Gane,1983).

Test Drive - Dirigir ou operar alguma coisa, para ver se funciona corretamente (Longman Group,1992).

Then - Então. Nas sentenças condicionais, palavra que precede a sentença ou expressão que se deve executar ou atribuir se a relação é verdadeira (Fragomeni,1986).

Tipos Abstratos de Dados - O conceito de tipo abstrato de dados vem do princípio mais geral de esconder informação. Ou seja, os tipos abstratos de dados escondem detalhes de representação e direcionam o acesso aos objetivos abstratos por meio de funções (Ghezzi,1985).

Top-Down - Do mais geral para o mais específico (Camarão,1989).

Underflow - Estouro Negativo. Geração de uma quantidade menor que a menor quantidade possível de ser armazenada em um registro (Paladino,1986).

Upgrade - Atualizar, Ampliar (Paladino,1986).

Variável Global - Variável que se encontra à disposição de qualquer programa do sistema (Fragomeni,1986).