ANTÔNIO SOARES DE OLIVEIRA JÚNIOR

PROGRAMAÇÃO ORIENTADA A OBJETOS: UMA ABORDAGEM PARA A ENGENHARIA DE SISTEMAS DE POTÊNCIA

CAMPINA GRANDE - PB
1998

ANTÔNIO SOARES DE OLIVEIRA JÚNIOR

PROGRAMAÇÃO ORIENTADA A OBJETOS: UMA ABORDAGEM PARA A ENGENHARIA DE SISTEMAS DE POTÊNCIA

Dissertação apresentada ao Curso de PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

ÁREA DE CONCENTRAÇÃO: PROCESSAMENTO DA ENERGIA

ORIENTADORES: BENEMAR ALENCAR, Ds.C.
WELLINGTON SANTOS MOTA, Ph.D.

CAMPINA GRANDE - PB
NOVEMBRO DE 1998

O48p Oliveira Júnior, Antônio Soares de.
Programação orientada a objetos : uma abordagem para a engenharia de sistemas de potência / Antônio Soares de Oliveira Júnior. - Campina Grande, 1998.
102 f.

Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1998.
"Orientação : Prof. Dr. Benemar Alencar de Souza, Prof. Dr. Wellington Santos Mota".
Referências.

1. Sistema de Potência. 2. Análise de Projetos Orientado a Objetos. 3. Software - Desenvolvimento. 4. Dissertação - Engenharia Elétrica. I. Souza, Benemar Alencar de. II. Mota, Wellington Santos. III. Universidade Federal da Paraíba - Campina Grande (PB). IV. Título

CDU 621.31(043)

PROGRAMAÇÃO ORIENTADA A OBJETOS: UMA ABORDAGEM PARA A ENGENHARIA DE SISTEMAS DE POTÊNCIA

ANTONIO SOARES DE OLIVEIRA JÚNIOR

Dissertação Aprovada em 20.11.1998

PROF. WELLINGTON SANTOS MOTA, Ph.D., UFPB
Orientador

PROF. BENEMAR ALENCAR DE SOUZA, D.Sc., UFPB
Orientador

PROF. DAGOBERTO LOURENÇO RIBEIRO, D.Sc., UFPB
Componente da Banca

PROF. MANOEL AFONSO DE CARVALHO JÚNIOR, Ph.D., UFPE
Componente da Banca

CAMPINA GRANDE - PB
Novembro - 1998

AGRADECIMENTOS

Aos Professores Wellington Santos Mota e Benemar Alencar pela valiosa orientação.

Aos amigos Walber Ulisses, Franklin Pamplona, Guilherme Dias e demais companheiros de curso e de trabalho na Escola Técnica Federal de Alagoas, pela ajuda, companheirismo e estímulo durante o desenvolvimento de todo o trabalho.

Ao CNPq pelo auxílio financeiro.

Aos meus irmãos, George e André, pelo grande incentivo, principalmente nas horas mais difíceis.

Um agradecimento muito especial à minha mãe Neumann e a meu pai Antônio, fontes do meu sucesso, que sempre me apoiaram e me estimularam.

Ao nosso Criador, por todas as alegrias e tristezas em minha vida, e por tudo que tenho conseguido realizar.

Finalmente, aos meus adorados filhos, Lucas e Nícolas, razão atual da minha existência e da minha perseverança. A eles eu dedico todo o meu trabalho e esforço.

RESUMO

O crescente aprimoramento dos computadores fornece ao engenheiro máquinas cada vez mais poderosas para tratamento e solução de uma gama de problemas que antes eram tratados manualmente, ou esquecidos por terem uma complexidade que gerava um grande desperdício de tempo deste profissional no processo de cálculo, ficando o trabalho de engenharia em segundo plano. Também, novas metodologias para criação de sistemas computacionais foram sendo desenvolvidas ao longo dos últimos anos, como resposta ao rápido avanço desses equipamentos.

Este trabalho apresenta uma sistemática elaborada para facilitar o processo de desenvolvimento de softwares pelo engenheiro de sistemas de potência, tanto em sua fase acadêmica quanto na profissional, utilizando a tecnologia de análise e projeto orientados a objetos. Além dos conceitos fundamentais definidos por esta nova tecnologia, também é apresentado um software híbrido, criado a partir destes conceitos juntamente com o uso da programação estruturada para enfatizar a transição entre as duas formas de programação. É utilizado o estudo de fluxo de carga como modelo para o sistema computacional desenvolvido, justamente por ser um tema bastante abordado em várias disciplinas do curso de graduação e pós-graduação em engenharia elétrica, na área de sistemas de potência.

ABSTRACT

The computers increasing improvement supplies the engineers with even more powerful machines. This machines are used to the treatment and resolution of an amount of problems which used to be manual treated or even forgotten. This kind of thing happened because these problems were complex enough to waste a great deal of time from the professional during the calculation. Consequentily, the engineering work becomes the background work. Nevertheless, new metodologies, for computational sistem creation, has been developed through the years, as an answer to the fast development in the equipments.

This work introduces an elaborated sistem in order to make it easier for engineer who develops softwares, mainly for power sistem engineers. It uses for this purpose a technology of analysis and project directed to objects. Beyond the fundamental concepts of this new tchnology, a hybrid software is also introduced, which was created from these concepts and joined to the usage of the structure programme to emphatize a transition between the two programming forms. The charge flux is used as a pattern to the developed computational sistem, because it is a well known item to be included in graduation and post graduation courses, in the power sistem area.

LISTA DE TABELAS

CAPÍTULO 4

CAPÍTULO 5

LISTA DE FIGURAS

CAPÍTULO 2

CAPÍTULO 3

CAPÍTULO 4

CAPÍTULO 5

SUMÁRIO

CAPÍTULO 1

Introdução

1.1 Considerações Gerais

O conhecimento de métodos de fluxo de carga, estabilidade, análise de faltas e proteção dos sistemas elétricos sempre fizeram parte das atribuições do engenheiro de sistemas de potência e são necessários ao planejamento e a operação destes sistemas.

O advento da computação digital e da necessidade de modernização destes estudos, impulsionada hoje pela globalização e atuais tendências de redução de gastos e maior eficiência, forneceram ao engenheiro de potência os motivos para se adaptar a essa nova perspectiva. Com isso, o conhecimento em informática e em programação tornaram-se ferramentas importantes para o tratamento de antigos problemas abordados nos sistemas elétricos.

Como exemplo temos os estudos de fluxo de potência que são utilizados nas fases de projeto, planejamento da expansão, planejamento da operação e na própria operação, sendo utilizados para análises de redes ou para estudos mais complexos como os de otimização, estabilidade, controle e supervisão (SOARES, 1993). Todavia, antes de 1930 esses estudos eram feitos manualmente com inúmeras simplificações que impossibilitavam a análise de grandes sistemas. Entre 1930 e 1956 eles passaram a ser feitos a partir de analisadores de rede em corrente alternada, que eram uma réplica monofásica em escala reduzida do sistema real e utilizava a interligação de elementos de circuitos e de fontes de tensão

(STEVENSON, 1986). Contudo, o problema básico da imprecisão e lentidão dos cálculos tornava o estudo cansativo e demorado. Com o surgimento dos computadores digitais foram desenvolvidos métodos matemáticos e consequentemente programas computacionais para melhorar a solução das equações de fluxo de potência. Adaptaram-se então diversas metodologias, algumas das quais tornaram-se de uso bastante amplo como o método de "Gauss-Seidel", "Newton-Raphson" e "Desacoplado Rápido".

1.2 Revisão Bibliográfica

Atualmente, a tecnologia de programação orientada a objetos (Object Oriented Programming – OOP) é vista não apenas como uma nova forma de programar, mas como uma nova maneira de pensar um problema de forma abstrata utilizando conceitos do mundo real e não idéias computacionais (RUMBAUGHT et al., 1995). Assim, algumas abordagens generalistas foram desenvolvidas para possibilitar a aplicação desse pensamento e a criação de softwares orientados a objetos com boa qualidade. A Técnica de Modelagem de Objetos (OMT) é uma delas. Neste método as construções baseadas em objetos são modelos de coisas reais. Ela também utiliza conceitos de tipificação, de classes e de planejamento prévio (RUMBAUGHT et al., 1995). Outra metodologia também bastante conhecida é o Método Fusion (COLEMAN, 1996). Esta é uma abordagem sistemática e prática para o processo de desenvolvimento de software, por integrar e estender os melhores recursos das abordagens orientadas a objetos bem sucedidas como a OMT, Booch e Objectory (COLEMAN, 1996), fornecendo um caminho mais direto ligando a análise e projeto orientados a objetos com a implementação.

Além de dispor de uma ampla bibliografia que trata das abordagens orientadas a objetos, o engenheiro de sistemas de potência encontra várias publicações contendo os conceitos existentes na OOP como: objetos, classes, mensagens, herança, encapsulamento e outros (GENERINI, 1996). Também é vasta a bibliografia referente a utilização de uma

linguagem de programação orientada a objetos como Object Pascal (CONNER et al., 1995), C^{++} (FAISON, 1995) ou FORTRAN 90 (CHAPMAN, 1997).

Com isto, a OOP forneceu a estrutura básica para metodologias de análise orientada a objetos (Object Oriented Analysis – OOA) e projeto orientado a objetos (Object Oriented Design – OOD), que definem uma nova era em termos de programação, suportando uma modularidade com uma interação gráfica a partir de uma interface com usuário e diferente da programação baseada em processos que dominou a análise e o projeto de sistemas de programação durante anos.

1.3 Motivação do Trabalho

Inicialmente e até uma década atrás, o que se tinha em termos de programação eram linguagens estruturadas numa característica passo a passo, limitadas por ambientes operacionais que dificultavam o tratamento com o usuário a partir de uma interface homem-máquina completamente desconfortável e complexa. Contudo, não havia por parte do engenheiro uma maior preocupação em abordar uma metodologia de análise e projeto para desenvolvimento do seu software que suprisse as suas necessidades em termos do problema de engenharia.

Nos últimos anos, a programação orientada a objetos vem tomando o lugar da programação estruturada em diversas áreas do conhecimento. Essa nova tecnologia de programação possibilitou vários benefícios: a maior facilidade de manutenção e posteriores modificações, a possibilidade da *miscigenação* entre linguagens diferentes, o *encapsulamento* de blocos no software que possibilitam a *reutilização* por outros sistemas, são alguns que podem ser citados.

Atualmente, é necessário pelo desenvolvedor de sistemas, a utilização de uma metodologia que forneça a base para o uso racional, adequado e da forma mais eficiente das ferramentas de análise e projeto no processo de desenvolvimento do seu sistema, já que

a OOA e OOD são uma tendência mundial tanto no meio acadêmico de pesquisa como nas empresas de engenharia.

Além disso, muitas vantagens estão sendo alcançadas pelo engenheiro com o desenvolvimento e o uso de novas tecnologias de programação. Entre elas podemos citar: **(i)** o computador em muitos casos fornece meios mais eficientes e econômicos para solução de um problema em engenharia, além de possibilitar estudos que não eram feitos antes da computação digital devido a grande quantidade de cálculos; **(ii)** o computador disponibiliza mais o engenheiro para o trabalho técnico possibilitando o aumento de sua habilidade na análise dos problemas de engenharia, evitando por parte deste, desperdício de tempo com os cálculos.

Por tudo isso e pela falta de uma abordagem direta e específica que fornecesse ao acadêmico e engenheiro uma orientação para a utilização dessa tecnologia, surgiu a necessidade do desenvolvimento desse trabalho.

1.4 Objetivos

Sabe-se que muitos trabalhos desenvolvidos em disciplinas do curso de graduação e em pesquisas de iniciação científica assim como em projetos na pós-graduação, terminam por contemplar o desenvolvimento de um software para tratamento do processo estudado. O objetivo deste trabalho não é o tratamento de nenhum dos métodos de análise e projeto orientados a objetos já criados, tão pouco o de desenvolver uma nova metodologia. O que se deseja é apresentar uma abordagem que facilite ao aluno e engenheiro de sistemas de potência o processo de desenvolvimento de softwares com esta nova tecnologia.

Soma-se a isto a possibilidade de utilizar o software criado como modelo em pesquisas acadêmicas na graduação e como ferramenta de trabalhos em disciplinas do curso.

1.5 Roteiro do Trabalho

Segue-se um breve resumo de cada capítulo desenvolvido neste trabalho.

No capítulo 2 apresenta-se o estudo do fluxo de potência a partir da metodologia de Newton-Raphson com a fundamentação matemática das equações necessárias ao programa.

No capítulo 3 são apresentados os conceitos fundamentais da tecnologia de programação orientada a objetos comentando-se cada terminologia utilizada. É abordado também o uso das bibliotecas de ligação dinâmica.

O capítulo 4 apresenta uma abordagem sistemática para desenvolvimento de softwares para a engenharia com os conceitos vistos anteriormente, utilizando como modelo o próprio software criado para estudos de fluxo de carga.

O capítulo 5 mostra um pouco da teoria de projeto de interface ser humano-computador.

No capítulo 6 são apresentadas as conclusões referentes a tudo que foi comentado nos capítulos anteriores e terminando por indicar sugestões para possíveis trabalhos futuros.

CAPÍTULO 2

Análise de Fluxo de Carga em Sistemas de Potência

2.1 Descrição do Problema

A análise de fluxo de carga faz parte dos estudos da engenharia de sistemas de potência que objetivam fornecer as condições ideais para o planejamento e a operação dos sistemas dentro de critérios de qualidade e confiabilidade estabelecidos em normas.

Os estudos de fluxo de carga são então uma das atividades mais desenvolvidas nas empresas de transmissão e distribuição de energia elétrica, em que um dos objetivos principais é manter as tensões dos barramentos de carga dentro dos níveis permissíveis adequando o sistema ao atendimento das cargas dentro dos critérios de qualidade de fornecimento de energia elétrica.

Com este objetivo são feitos estudos em tempo real na fase de operação e nas fases de planejamento e projeto são realizadas simulações de comportamento futuro, ambos atualmente a partir de métodos computacionais.

Um dos grandes problemas dos sistemas de potência é a contínua mudança nas características das cargas, afetando a geração que também tem que mudar para se adequar ao novo regime de operação (WEEDY, 1979). Outros problemas que podem ocorrer, não com tanta frequência, são perdas de linhas de transmissão ou até mesmo de alguma unidade geradora. A injeção de potência reativa pelas cargas também afeta a operação dos

sistemas e por isso deve ser prevista no projeto de um novo sistema ou no planejamento da expansão de um sistema já existente.

Para tanto, como parte da solução de um fluxo de carga devem ser fornecidas as tensões nos barramentos da rede, as potências injetadas em todas as barras e o fluxo de potência nas linhas de transmissão para determinação das perdas. Esses resultados são obtidos para operações em regime permanente, operando o sistema tanto em condições normais como anormais.

No presente estudo algumas considerações devem ser feitas. São elas: **(i)** será abordada uma solução para fluxo de carga em sistemas de potência operando sob condições balanceadas. Com isso, a representação monofásica do sistema é suficiente; **(ii)** serão estudados os sistemas em condições de regime permanente (WEEDY, 1979).

Com este conjunto de soluções obtido a partir do fluxo de carga para uma determinada rede ou sistema são tomadas as decisões cabíveis. Como por exemplo: para regulação dos níveis de tensão, alteração na quantidade de reativos circulando no sistema, interligação de sistemas com o objetivo de atender manobras para manutenção ou por aumentos de demanda de carga, projeto de um novo sistema adequado às cargas a serem instaladas, planejamento da expansão de um sistema existente pelo aumento da demanda de carga, entre outras decisões necessárias a solucionar um problema específico de um sistema de potência (STAGG & EL-ABIAD, 1968).

O estudo de fluxo de carga realizado neste trabalho como modelo para demonstrar a abordagem orientada a objetos desenvolvida para sistemas de potência, baseia-se em sistemas simples constituídos apenas pelos geradores, linhas de transmissão, transformadores fixos e pelas cargas representadas pelas potências ativa e reativa. Não serão representados capacitores estáticos, reatores de derivação nem transformadores com tap para regulação de tensão.

Para tanto, é empregada uma formulação matemática utilizando-se a análise nodal das redes para se determinar a matriz admitância de barras a partir do diagrama unifilar, que é uma representação monofásica do sistema. São usados as componentes reais e imaginárias das tensões de nós como variáveis independentes nas equações de fluxo de carga. Essa escolha foi feita sem considerar as vantagens de uma abordagem matemática ou outra, o objetivo é apenas definir os passos da metodologia orientada a objetos. Assim, a qualidade final do programa no requisito de cálculo de fluxo de carga não foi considerada.

Nos estudos de fluxo de carga para sistemas de potência existem basicamente três tipos de barras ou nós que representam os barramentos da rede, a saber:

1. Barras de geração ou de tensão controlada, representam as barras que possuem geração com ou sem cargas instaladas. Devem ser fornecidos como dados de entrada o módulo da tensão de geração e a potência ativa nominal na barra. Com isto, o programa determina a potência reativa e o ângulo de fase da tensão na barra.
2. Barras de carga, representam todas as barras que possuem apenas as cargas instaladas. Devem ser fornecidas as potências ativa e reativa. Determinam-se então o módulo e o ângulo de fase das tensões em cada barra deste tipo.
3. Barras de balanço, onde o módulo e o ângulo de fase da tensão são mantidos constantes enquanto a potência ativa e a reativa são determinados com o objetivo, dentre vários, de suprir as perdas nas linhas de transmissão (STEVENSON, 1986).

A partir do diagrama unifilar da rede representando o sistema de potência são numeradas todas as barras ou nós do sistema. No nosso estudo a barra desejada para ser a de balanço foi considerada a última (barra 5 na figura 2.1).

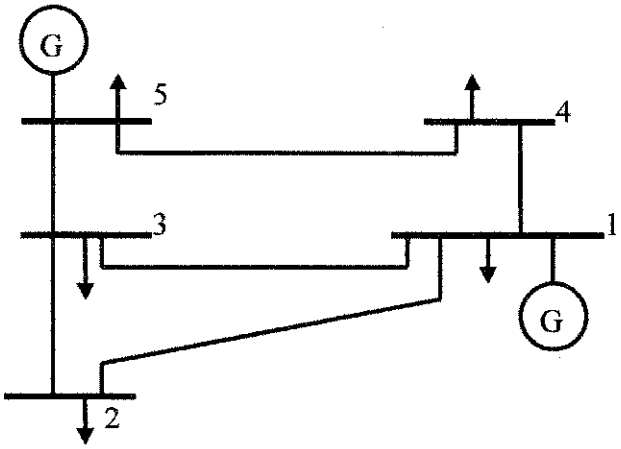

Figura 2.1 – Numeração das barras em um diagrama unifilar

Esta é a abordagem matemática mais utilizada para análise de fluxo de carga na maioria dos sistemas de potência por sua simplicidade na preparação dos dados, facilidade na formação e modificação da matriz admitância de barra quando em mudanças de configuração na rede. Também gasta-se menos tempo para execução dos cálculos pelo computador e a quantidade de memória requerida para armazenar os dados é menor quando comparada com outras abordagens (STAGG & EL-ABIAD, 1968). Não se considerando as características de orientação a objetos.

Como técnica de solução utilizou-se o método de Newton-Raphson para tratamento das equações de fluxo de carga, como será mostrado no item 2.3. Apesar desta já não ser a metodologia mais utilizada por estar sendo substituída pelo método Desacoplado Rápido que propicia melhores vantagens como: menor número de iterações necessárias à convergência do resultado e menor tempo na execução dos cálculos. Isto não foi considerado importante já que apenas um modelo simples era necessário para demonstrar a abordagem orientada a objetos em estudo neste trabalho.

2.2 Formulação Matemática: Equações para o Sistema de Potência

2.2.1 Equações de Rede: Matriz Admitâncias de Barras

Inicialmente, tomemos um sistema com três barras como mostra a figura 2.2. Este sistema é constituído pelas barras ou nós, pelas linhas de transmissão representadas pelas suas respectivas impedâncias série e derivação em por unidade (p.u.), por dois geradores e por um motor representando uma carga, além dos transformadores instalados no sistema.

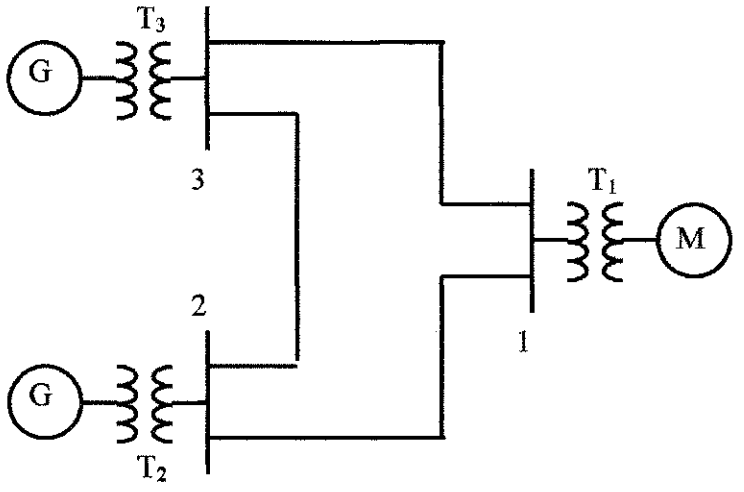

Figura 2.2 – Diagrama unifilar para um sistema de três barras

Os geradores e o motor devem ser representados pelas suas respectivas forças eletromotrizes e os transformadores por suas reatâncias (figura 2.3).

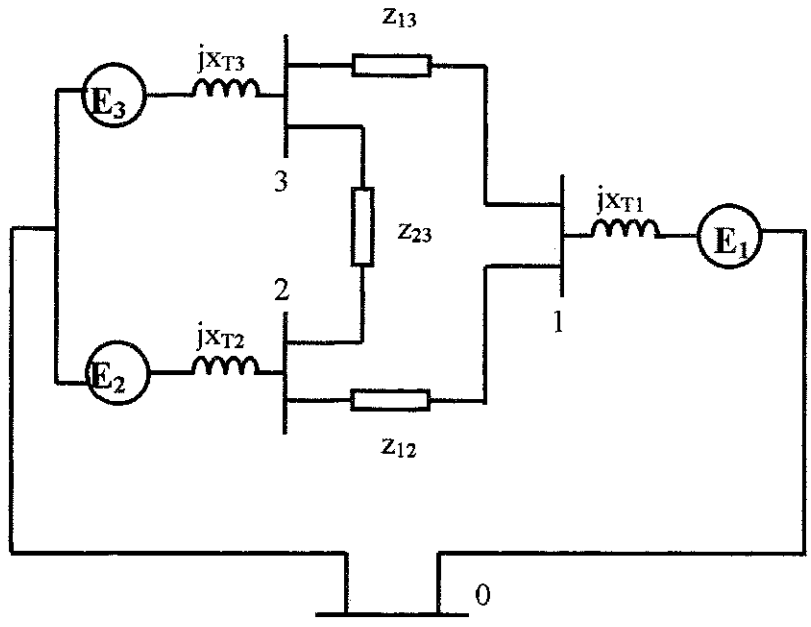

Figura 2.3 – Um sistema representado pelas reatâncias e fontes de tensão.

Para obtermos a matriz admitância de barra do sistema, devemos converter na figura 2.3, as fontes de tensão em fontes de corrente e substituir as impedâncias por suas admitâncias equivalentes.

Pela equivalência entre fontes de tensão e corrente:

$$E_1 = I_1 \times j(x_{T1}) \quad \textbf{(2.1)}$$

$$E_2 = I_2 \times j(x_{T2}) \quad \textbf{(2.2)}$$

$$E_3 = I_3 \times j(x_{T3}) \quad \textbf{(2.3)}$$

Assim, o sistema ficará representado pelo diagrama na figura 2.4.

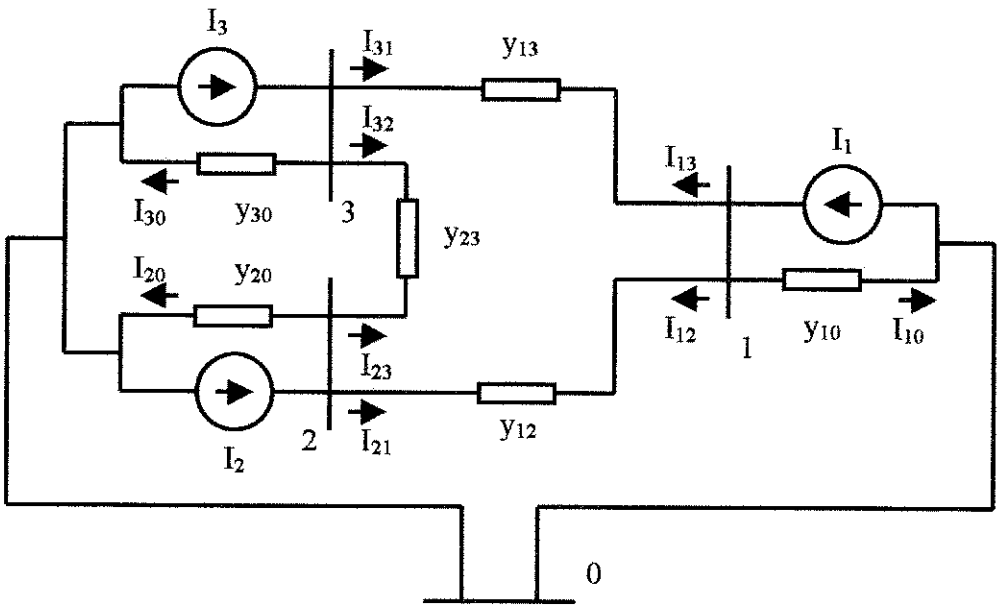

Figura 2.4 – Um sistema representado pelas admitâncias e fontes de corrente.

Sendo:

$$y_{12} = y_{21} = \frac{1}{z_{12}} \tag{2.4}$$

$$y_{13} = y_{31} = \frac{1}{z_{13}} \tag{2.5}$$

$$y_{23} = y_{32} = \frac{1}{z_{23}} \tag{2.6}$$

$$y_{10} = \frac{1}{j(x_{T1})} \tag{2.7}$$

$$y_{20} = \frac{1}{j(x_{T2})} \tag{2.8}$$

$$y_{30} = \frac{1}{j(x_{T3})} \tag{2.9}$$

Tomando-se o neutro como nó de referência para as tensões de barra, temos para cada barra a sua respectiva tensão: V_1, V_2 e V_3

Utilizando-se a lei de Kirchhoff para determinar as correntes em cada nó, temos:

$$I_1 = I_{10} + I_{12} + I_{13} \tag{2.10}$$

$$I_2 = I_{21} + I_{20} + I_{23} \tag{2.11}$$

$$I_3 = I_{31} + I_{32} + I_{30} \tag{2.12}$$

Substituindo as tensões das barras nas equações (2.10) à (2.12), temos:

$$I_1 = y_{10}V_1 + y_{12}(V_1 - V_2) + y_{13}(V_1 - V_3) \tag{2.13}$$

$$I_2 = y_{21}(V_2 - V_1) + y_{20}V_2 + y_{23}(V_2 - V_3) \tag{2.14}$$

$$I_3 = y_{31}(V_3 - V_1) + y_{32}(V_3 - V_2) + y_{30}V_3 \tag{2.15}$$

Logo:

$$I_1 = y_{10}V_1 + y_{12}V_1 - y_{12}V_2 + y_{13}V_1 - y_{13}V_3 \tag{2.16}$$

$$I_2 = y_{21}V_2 - y_{21}V_1 + y_{20}V_2 + y_{23}V_2 - y_{23}V_3 \tag{2.17}$$

$$I_3 = y_{31}V_3 - y_{31}V_1 + y_{32}V_3 - y_{32}V_2 + y_{30}V_3 \tag{2.18}$$

Rearranjando as equações anteriores, temos:

$$I_1 = (y_{10} + y_{12} + y_{13})V_1 - y_{12}V_2 - y_{13}V_3 \tag{2.19}$$

$$I_2 = -y_{21}V_1 + (y_{21} + y_{20} + y_{23})V_2 - y_{23}V_3 \tag{2.20}$$

$$I_3 = -y_{31}V_1 - y_{32}V_2 + (y_{31} + y_{32} + y_{30})V_3 \tag{2.21}$$

Colocando na forma matricial, obtemos:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} \\ Y_{21} & Y_{22} & Y_{23} \\ Y_{31} & Y_{32} & Y_{33} \end{bmatrix} \times \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} \tag{2.22}$$

Sendo:

$$Y_{11} = y_{10} + y_{12} + y_{13} \tag{2.23}$$

$$Y_{22} = y_{21} + y_{20} + y_{23} \tag{2.24}$$

$$Y_{33} = y_{31} + y_{32} + y_{30} \tag{2.25}$$

$$Y_{12} = Y_{21} = -y_{12} = -y_{21} \tag{2.26}$$

$$Y_{13} = Y_{31} = -y_{13} = -y_{31} \tag{2.27}$$

$$Y_{23} = Y_{32} = -y_{23} = -y_{32} \tag{2.28}$$

A matriz das admitâncias na equação (2.22) é conhecida como *Matriz Admitâncias de Barras* e será utilizada para solução das equações de fluxo de carga.

Para um sistema contendo *n* barras, a equação (2.22) pode ser generalizada como segue.

Seja uma barra qualquer *p*, a corrente *I* fluindo para esta barra é dada por:

$$I_p = \sum_{q=1}^{n} Y_{pq} V_q \quad , \text{p} = 1,2,3,\ldots,\text{n} \tag{2.29}$$

Colocando na forma matricial, temos:

$$\begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_n \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} & \cdots & Y_{1n} \\ Y_{21} & Y_{22} & \cdots & Y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ Y_{n1} & Y_{n2} & \cdots & Y_{nn} \end{bmatrix} \times \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix} \tag{2.30}$$

Sendo os elementos na diagonal principal da matriz admitâncias de barras acima, o somatório de todas as admitâncias ligadas ao nó correspondente e os elementos fora da diagonal principal como sendo o negativo da admitância que liga os dois nós em questão.

2.2.2 Equações de Fluxo de Carga

Como vimos na seção anterior, a corrente *I* fluindo para uma barra qualquer *p* do sistema é dada pela equação (2.29).

Logo, a nova equação (2.35*) retificada é dada por:

$$V_p = \frac{1}{Y_{pp}} \times \left(\left(\frac{P_p - jQ_p}{V_p^*} \right) - \sum_{\substack{q=1 \\ q \neq p}}^{n} Y_{pq} V_q \right) \qquad \textbf{(2.35)}$$

Para: $p = 1,2,3,...,n \text{ e } p \neq s$

2.2.3 Equações de Fluxo nas Linhas

Após a obtenção da solução das equações de fluxo de carga, deve-se calcular o fluxo das potências nas linhas de transmissão para determinação das perdas no sistema.

A corrente i_{pq} fluindo de uma barra p qualquer para outra barra q qualquer, é dada por:

$$i_{pq} = y_{pq\,série}(V_p - V_q) + V_p \frac{y_{pq_{shunt}}}{2} \qquad \textbf{(2.36)}$$

Sendo:

$y_{pq\,série}$ = admitância série da linha pq

$y_{pq\,shunt}$ = admitância em derivação da linha pq

A segunda componente da corrente i_{pq} no segundo membro da equação (2.36), refere-se à contribuição da admitância shunt da linha para i_{pq}, devido a sua representação concentrada, metade em cada lado da linha. Como mostra a figura 2.5 a seguir:

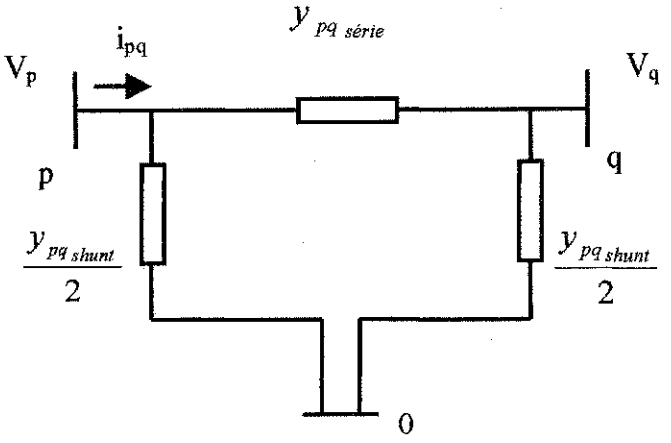

Figura 2.5 – A corrente entre duas barras quaisquer de um sistema

O fluxo das potências ativa P_{pq} e reativa Q_{pq} da barra p para a barra q, é dado por:

$$P_{pq} - jQ_{pq} = i_{pq} V_p^* \tag{2.37}$$

Logo:

$$P_{pq} - jQ_{pq} = V_p^* y_{pq\,série}(V_p - V_q) + V_p^* V_p \frac{y_{pq_{shunt}}}{2} \tag{2.38}$$

Da mesma forma, o fluxo das potências ativa P_{qp} e reativa Q_{qp} da barra q para a barra p, é dado por:

$$P_{qp} - jQ_{qp} = i_{qp} V_q^* \tag{2.39}$$

Logo:

$$P_{qp} - jQ_{qp} = V_q^* y_{pq\,série}(V_q - V_p) + V_q^* V_q \frac{y_{pq_{shunt}}}{2} \tag{2.40}$$

Para se determinar as perdas de potência na linha de transmissão que conecta a barra p à barra q, deve-se somar as potências ativa e reativa determinadas pelas equações (2.38) e (2.40), logo:

Perdas de potência ativa: $P_{ativa} = P_{pq} + P_{qp}$ **(2.41)**

Potência reativa: $P_{reativa} = Q_{pq} + Q_{qp}$ **(2.42)**

2.3 Uma Técnica de Solução: Método de Newton-Raphson

A formulação para solução de um fluxo de carga em um sistema de potência, resulta num conjunto de equações não-lineares, independente da abordagem ser a partir das barras ou das malhas da rede.

O método numérico utilizado para tratamento das equações do fluxo de carga é o de Newton-Raphson, que é iterativo e emprega a expansão por série de Taylor limitada aos termos de primeira derivada para obter a aproximação linear da função.

Sejam as seguintes equações não-lineares com duas funções de duas variáveis cada:

$$y_1 = f_1(x_1, x_2) \quad \textbf{(2.43)}$$

$$y_2 = f_2(x_1, x_2) \quad \textbf{(2.44)}$$

Onde y_1 e y_2 são constantes (HORNBECK, 1975).

As equações de fluxo de carga devem ser colocadas nesta forma para que o método de Newton-Raphson possa obter um conjunto de equações que possam ser tratadas em um computador.

Inicialmente, devemos tomar a equação (2.35) e expandi-la em duas na forma das equações (2.43) e (2.44), logo:

$$V_p = \frac{1}{Y_{pp}} \times \left(\left(\frac{P_p - jQ_p}{V_p^*} \right) - \sum_{\substack{q=1 \\ q \neq p}}^{n} Y_{pq} V_q \right) \tag{2.35}$$

Para: p = 1,2,3,...,n e p ≠ s

Isolando as potências ativa e reativa, temos:

$$\frac{P_p - jQ_p}{V_p^*} = Y_{pp} V_p + \sum_{\substack{q=1 \\ q \neq p}}^{n} Y_{pq} V_q \tag{2.45}$$

Para: p = 1,2,3,...,n e p ≠ s

Assim:

$$P_p - jQ_p = V_p^* \sum_{q=1}^{n} Y_{pq} V_q \tag{2.46}$$

Para: p = 1,2,3,...,n e p ≠ s

Escolhendo-se a forma retangular para expressar as tensões das barras e as admitâncias das linhas, para uma barra p qualquer teremos:

$$V_p = e_p + jf_p \tag{2.47}$$

$$Y_{pq} = G_{pq} - jB_{pq} \tag{2.48}$$

Substituindo as equações (2.47) e (2.48) na equação (2.46), temos:

$$P_p - jQ_p = (e_p - jf_p) \times \left\{ \sum_{q=1}^{n} (G_{pq} - jB_{pq}) \times (e_q + jf_q) \right\} \tag{2.49}$$

Para: p = 1,2,3,...,n e p ≠ s

Assim:

$$P_p - jQ_p = \left(e_p - jf_p\right)\times\left\{\sum_{q=1}^{n}\left(G_{pq}e_q + jG_{pq}f_q\right)+\left(-jB_{pq}e_q + B_{pq}f_q\right)\right\}$$

$$P_p - jQ_p = \left(e_p - jf_p\right)\times\left\{\sum_{q=1}^{n}\left(G_{pq}e_q + B_{pq}f_q\right)+ j\left(G_{pq}f_q - B_{pq}e_q\right)\right\}$$

$$P_p - jQ_p = \sum_{q=1}^{n}\left\{e_p\left(G_{pq}e_q + B_{pq}f_q\right)+ je_p\left(G_{pq}f_q - B_{pq}e_q\right)- jf_p\left(G_{pq}e_q + B_{pq}f_q\right)+ f_p\left(G_{pq}f_q - B_{pq}e_q\right)\right\}$$

Desta forma:

$$P_p - jQ_p = \sum_{q=1}^{n}\left[e_p\left(G_{pq}e_q + B_{pq}f_q\right)+ f_p\left(G_{pq}f_q - B_{pq}e_q\right)\right]+ j\left[e_p\left(G_{pq}f_q - B_{pq}e_q\right)- f_p\left(G_{pq}e_q + B_{pq}f_q\right)\right] \quad \textbf{(2.50)}$$

Para: $p = 1,2,3,\ldots,n$ e $p \neq s$

Separando os termos real e imaginário na equação (2.50), temos:

$$P_p = \sum_{q=1}^{n}\left[e_p\left(G_{pq}e_q + B_{pq}f_q\right)+ f_p\left(G_{pq}f_q - B_{pq}e_q\right)\right] \quad \textbf{(2.51)}$$

$$Q_p = \sum_{q=1}^{n}\left[f_p\left(G_{pq}e_q + B_{pq}f_q\right)- e_p\left(G_{pq}f_q - B_{pq}e_q\right)\right] \quad \textbf{(2.52)}$$

Para: $p = 1,2,3,\ldots,n$ e $p \neq s$

Comparando as equações (2.51) e (2.52) com as equações (2.43) e (2.44) respectivamente, temos que: $P_p = y_1$ $\quad Q_p = y_2$ $\quad e_p = x_1$ $\quad f_p = x_2$

As soluções para as equações (2.51) e (2.52) devem então ser obtidas.

Como o método de Newton-Raphson é iterativo deve ser dada a estimativa inicial para a solução.

Sendo: $V_p^{(0)} = e_p^{(0)} + jf_p^{(0)}$ o chute inicial e sendo: $\Delta V_p^{(0)} = \Delta e_p^{(0)} + j\Delta f_p^{(0)}$ os valores a serem somados às estimativas iniciais para fornecerem as soluções mais corretas, temos que:

$$P_p^{(0)} = f_1\left(e_p^{(0)} + \Delta e_p^{(0)}, f_p^{(0)} + \Delta f_p^{(0)}\right) \tag{2.53}$$

$$Q_p^{(0)} = f_2\left(e_p^{(0)} + \Delta e_p^{(0)}, f_p^{(0)} + \Delta f_p^{(0)}\right) \tag{2.54}$$

Sendo: $P_p^{(0)}$ e $Q_p^{(0)}$ os valores fornecidos das potências líquidas ativa e reativa para a barra em questão.

Expandindo as equações (2.53) e (2.54) na série de Taylor limitada aos termos de primeira derivada, temos:

$$P_p^{(0)} = f_1\left(e_p^{(0)}, f_p^{(0)}\right) + \Delta e_p^{(0)} \left.\frac{\partial f_1}{\partial e_p}\right|_{(0)} + \Delta f_p^{(0)} \left.\frac{\partial f_1}{\partial f_p}\right|_{(0)} + \cdots \tag{2.55}$$

$$Q_p^{(0)} = f_2\left(e_p^{(0)}, f_p^{(0)}\right) + \Delta e_p^{(0)} \left.\frac{\partial f_2}{\partial e_p}\right|_{(0)} + \Delta f_p^{(0)} \left.\frac{\partial f_2}{\partial f_p}\right|_{(0)} + \cdots \tag{2.56}$$

Para: $p = 1,2,3,\ldots,n$ e $p \neq s$

Colocando as equações (2.55) e (2.56) na forma matricial, obtemos:

$$\begin{bmatrix} P_p^{(0)} - f_1\left(e_p^{(0)}, f_p^{(0)}\right) \\ Q_p^{(0)} - f_2\left(e_p^{(0)}, f_p^{(0)}\right) \end{bmatrix} = \begin{bmatrix} \left.\frac{\partial f_1}{\partial e_p}\right|_{(0)} & \left.\frac{\partial f_1}{\partial f_p}\right|_{(0)} \\ \left.\frac{\partial f_2}{\partial e_p}\right|_{(0)} & \left.\frac{\partial f_2}{\partial f_p}\right|_{(0)} \end{bmatrix} \times \begin{bmatrix} \Delta e_p^{(0)} \\ \Delta f_p^{(0)} \end{bmatrix} \tag{2.57}$$

Para: $p = 1,2,3,\ldots,n$ e $p \neq s$

Se denominarmos $f_1\left(e_p^{(0)}, f_p^{(0)}\right) = P_p^{(1)}$ e $f_2\left(e_p^{(0)}, f_p^{(0)}\right) = Q_p^{(1)}$ como sendo os valores calculados de P_p e Q_p para as estimativas iniciais $e_p^{(0)}$ e $f_p^{(0)}$, podemos verificar que:

$$\begin{bmatrix} P_p^{(0)} - f_1\left(e_p^{(0)}, f_p^{(0)}\right) \\ Q_p^{(0)} - f_2\left(e_p^{(0)}, f_p^{(0)}\right) \end{bmatrix} = \begin{bmatrix} P_p^{(0)} - P_p^{(1)} \\ Q_p^{(0)} - Q_p^{(1)} \end{bmatrix} \tag{2.58}$$

E então:

$$\Delta P_p^{(0)} = P_p^{(0)} - P_p^{(1)} \tag{2.59}$$

$$\Delta Q_p^{(0)} = Q_p^{(0)} - Q_p^{(1)} \tag{2.60}$$

Sendo: $\Delta P_p^{(0)}$ e $\Delta Q_p^{(0)}$ as diferenças nas potências ativa e reativa fornecidas pelo sistema $(P_p^{(0)}$ *e* $Q_p^{(0)})$ com relação as potências calculadas $(P_p^{(1)}$ *e* $Q_p^{(1)})$ a partir das estimativas iniciais da tensão $(V_p^{(0)})$, para uma barra qualquer p.

Logo teremos:

$$\begin{bmatrix} \Delta P_p^{(0)} \\ \Delta Q_p^{(0)} \end{bmatrix} = \underbrace{\begin{bmatrix} J_1^{(0)} & J_2^{(0)} \\ J_3^{(0)} & J_4^{(0)} \end{bmatrix}}_{J^{(0)}} \times \begin{bmatrix} \Delta e_p^{(0)} \\ \Delta f_p^{(0)} \end{bmatrix} \tag{2.61}$$

A matriz quadrada das derivadas parciais na equação (2.57) é denominada de Jacobiano *(J)*. No cálculo dos valores numéricos das derivadas parciais do Jacobiano, o termo $J^{(0)}$ indica que foram utilizadas as estimativas iniciais $e_p^{(0)}$ e $f_p^{(0)}$.

Uma equação matricial genérica para tratamento do Jacobiano pode ser obtida levando-se em consideração todas as barras do sistema exceto a de referência (balanço), tida como a n-ézima barra, assim temos:

$$\begin{bmatrix} \Delta P_1 \\ \vdots \\ \Delta P_{n-1} \\ \Delta Q_1 \\ \vdots \\ \Delta Q_{n-1} \end{bmatrix} = \begin{bmatrix} \frac{\partial P_1}{\partial e_1} & \cdots & \frac{\partial P_1}{\partial e_{n-1}} & \frac{\partial P_1}{\partial f_1} & \cdots & \frac{\partial P_1}{\partial f_{n-1}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial P_{n-1}}{\partial e_1} & \cdots & \frac{\partial P_{n-1}}{\partial e_{n-1}} & \frac{\partial P_{n-1}}{\partial f_1} & \cdots & \frac{\partial P_{n-1}}{\partial f_{n-1}} \\ \frac{\partial Q_1}{\partial e_1} & \cdots & \frac{\partial Q_1}{\partial e_{n-1}} & \frac{\partial Q_1}{\partial f_1} & \cdots & \frac{\partial Q_1}{\partial f_{n-1}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial Q_{n-1}}{\partial e_1} & \cdots & \frac{\partial Q_{n-1}}{\partial e_{n-1}} & \frac{\partial Q_{n-1}}{\partial f_1} & \cdots & \frac{\partial Q_{n-1}}{\partial f_{n-1}} \end{bmatrix} \times \begin{bmatrix} \Delta e_1 \\ \vdots \\ \Delta e_{n-1} \\ \Delta f_1 \\ \vdots \\ \Delta f_{n-1} \end{bmatrix} \quad \textbf{(2.62)}$$

Deve-se agora determinar todos os elementos do jacobiano derivando-se parcialmente as equações (2.51) e (2.52) com relação a e_p e f_p para se obter J_1, J_2, J_3 e J_4.

1) Determinação dos elementos de J_1:

A partir da equação (2.51), temos:

$$P_p = e_p\left(G_{pp}e_p + B_{pp}f_p\right) + f_p\left(G_{pp}f_p - B_{pp}e_p\right) + \sum_{\substack{q=1 \\ q\neq p}}^{n}\left[e_p\left(G_{pq}e_q + B_{pq}f_q\right) + f_p\left(G_{pq}f_q - B_{pq}e_q\right)\right] \quad \textbf{(2.63)}$$

Para: p = 1,2,3,...,n-1

1.1) Para os elementos que estão dentro da diagonal principal da sub-matriz J_1.

Derivando parcialmente P_p em relação a e_p na equação (2.63) , temos:

$$\frac{\partial P_p}{\partial e_p} = 2e_pG_{pp} + f_pB_{pp} - f_pB_{pp} + \sum_{\substack{q=1 \\ q\neq p}}^{n}\left(e_qG_{pq} + f_qB_{pq}\right) \quad \textbf{(2.64)}$$

Para: p = 1,2,3,...,n-1

Logo:

$$\frac{\partial P_p}{\partial e_p} = 2e_pG_{pp} + \sum_{\substack{q=1 \\ q\neq p}}^{n}\left(e_qG_{pq} + f_qB_{pq}\right) \quad \textbf{(2.65)}$$

Para: p = 1,2,3,...,n-1

1.2) Para os elementos que estão fora da diagonal principal da sub-matriz J_1.

Derivando parcialmente P_p em relação a e_q na equação (2.63) , temos:

$$\frac{\partial P_p}{\partial e_q} = e_p G_{pq} - f_p B_{pq} \tag{2.66}$$

Para: q≠p

2) Determinação dos elementos de J_2:

2.1) Para os elementos que estão dentro da diagonal principal da sub-matriz J_2.

Derivando parcialmente P_p em relação a f_p na equação (2.63) , temos:

$$\frac{\partial P_p}{\partial f_p} = e_p B_{pp} + 2 f_p G_{pp} - e_p B_{pp} + \sum_{\substack{q=1 \\ q \neq p}}^{n} \left(f_q G_{pq} - e_q B_{pq} \right) \tag{2.67}$$

Para: p = 1,2,3,...,n-1

Logo:

$$\frac{\partial P_p}{\partial e_p} = 2 f_p G_{pp} + \sum_{\substack{q=1 \\ q \neq p}}^{n} \left(f_q G_{pq} - e_q B_{pq} \right) \tag{2.68}$$

Para: p = 1,2,3,...,n-1

2.2) Para os elementos que estão fora da diagonal principal da sub-matriz J_2:

Derivando parcialmente P_p em relação a f_q na equação (2.63) , temos:

$$\frac{\partial P_p}{\partial f_q} = e_p B_{pq} + f_p G_{pq} \tag{2.69}$$

Para: q≠p

3) Determinação dos elementos de J_3:

A partir da equação (2.52), temos:

$$Q_p = f_p\left(e_p G_{pp} + f_p B_{pp}\right) - e_p\left(f_p G_{pp} - e_p B_{pp}\right) + \sum_{\substack{q=1 \\ q\neq p}}^{n}\left[f_p\left(e_q G_{pq} + f_q B_{pq}\right) - e_p\left(f_q G_{pq} - e_q B_{pq}\right)\right] \qquad \textbf{(2.70)}$$

Para: p = 1,2,3,...,n-1

3.1) Para os elementos que estão dentro da diagonal principal da sub-matriz J_3.

Derivando parcialmente Q_p em relação a e_p na equação (2.70) , temos:

$$\frac{\partial Q_p}{\partial e_p} = f_p G_{pp} - f_p G_{pp} + 2e_p B_{pp} + \sum_{\substack{q=1 \\ q\neq p}}^{n}\left(e_q B_{pq} - f_q G_{pq}\right) \qquad \textbf{(2.71)}$$

Para: p = 1,2,3,...,n-1

Logo:

$$\frac{\partial Q_p}{\partial e_p} = 2e_p B_{pp} + \sum_{\substack{q=1 \\ q\neq p}}^{n}\left(e_q B_{pq} - f_q G_{pq}\right) \qquad \textbf{(2.72)}$$

Para: p = 1,2,3,...,n-1

3.2) Para os elementos que estão fora da diagonal principal da sub-matriz J_3.

Derivando parcialmente Q_p em relação a e_q na equação (2.70) , temos:

$$\frac{\partial Q_p}{\partial e_q} = f_p G_{pq} + e_p B_{pq} \qquad \textbf{(2.73)}$$

Para: q≠p

4) Determinação dos elementos de J_4:

4.1) Para os elementos que estão dentro da diagonal principal da sub-matriz J_4.

Derivando parcialmente Q_p em relação a f_p na equação (2.70) , temos:

$$\frac{\partial Q_p}{\partial f_p} = e_p G_{pp} + 2 f_p B_{pp} - e_p G_{pp} + \sum_{\substack{q=1 \\ q \neq p}}^{n} \left(e_q G_{pq} + f_q B_{pq} \right) \qquad \textbf{(2.74)}$$

Para: p = 1,2,3,...,n-1

Logo:

$$\frac{\partial Q_p}{\partial e_p} = 2 f_p B_{pp} + \sum_{\substack{q=1 \\ q \neq p}}^{n} \left(e_q G_{pq} + f_q B_{pq} \right) \qquad \textbf{(2.75)}$$

Para: p = 1,2,3,...,n-1

4.2) Para os elementos que estão fora da diagonal principal da sub-matriz J_4.

Derivando parcialmente Q_p em relação a f_q na equação (2.70) , temos:

$$\frac{\partial Q_p}{\partial f_q} = f_p B_{pq} - e_p G_{pq} \qquad \textbf{(2.76)}$$

Para: q≠p

De posse das equações que relacionam as derivadas parciais de primeira ordem das potências ativa e reativa com relação aos valores reais e imaginários das tensões de cada barra, ou seja, equações (2.65), (2.66), (2.68), (2.69), (2.72), (2.73), (2.75) e (2.76), inicia-se então o processo iterativo de Newton-Raphson para solução das equações de fluxo de carga:

1. O primeiro passo é calcular os valores das potências ativa $(P_p^{(1)})$ e reativa $(Q_p^{(1)})$ em cada barra, com exceção da barra de referência (balanço), para os valores estimados das tensões de cada barra $(V_p^{(0)} = e_p^{(0)} + jf_p^{(0)})$, utilizando-se as equações (2.51) e (2.52),

2. em seguida, calcula-se as diferenças ($\Delta P_p^{(0)}$ e $\Delta Q_p^{(0)}$) entre as potências ativa e reativa fornecidas ($P_p^{(0)}$) e ($Q_p^{(0)}$) com relação as calculadas para cada barra ($P_p^{(1)}$) e ($Q_p^{(1)}$), a partir das equações (2.59) e (2.60),
3. determina-se todos os elementos do jacobiano para a primeira iteração $J^{(0)}$, tendo-se como base as estimativas inicias dos valores das tensões de cada barra ($V_p^{(0)} = e_p^{(0)} + jf_p^{(0)}$). Estimativas iniciais em torno de $V_p^{(0)} = 1.0 + j0.0$, são consideradas satisfatórias para a maioria dos sistemas reais (WEEDY, 1979),
4. de posse do jacobiano, deve-se invertê-lo para se calcular as correções nas tensões de cada barra ($\Delta V_p^{(0)} = \Delta e_p^{(0)} + j\Delta f_p^{(0)}$),
5. os novos valores de tensão ($V_p^{(1)} = e_p^{(1)} + jf_p^{(1)}$) para a próxima iteração são então calculados a partir das correções obtidas anteriormente ($V_p^{(1)} = V_p^{(0)} + \Delta V_p^{(0)}$),
6. O processo é repetido voltando-se ao *passo 1* e utilizando-se os valores das tensões de barra calculados no *passo 5* até que os critérios de convergência tenham sido satisfeitos, isto é, ou todos os valores das potências ativa e reativa ou todos os valores das tensões em cada barra calculados numa iteração e subtraídos dos calculados na iteração anterior, devam estar dentro de uma tolerância pré-estabelecida.

O método é finalizado com a determinação das potências ativa e reativa na barra de referência (balanço) a partir das equações (2.51) e (2.52) e determinação das perdas nas linhas de transmissão, conforme estabelecido na seção 2.2.3.

Os resultados obtidos com as tensões e potências em cada barra e os fluxos das potência nas linhas são então analisados pelos engenheiros de sistemas de potência com vistas a algum objetivo específico, como alguns referenciados na seção 2.1 que trata da descrição do problema.

CAPÍTULO 3

A Tecnologia de Orientação a Objetos

3.1 Introdução

Foi no final dos anos sessenta que se discutiu pela primeira vez sobre programação orientada a objetos. Na época, alguns analistas estavam trabalhando com a linguagem Simula. Na década seguinte, os analistas da Xerox Parc desenvolveram uma linguagem chamada Smalltalk, foi a primeira a implementar os conceitos de OOP até então discutidos. Nesta mesma época, o resto do mundo usava linguagens tais como COBOL e FORTRAN. Ainda demoraria um bom tempo para se discutir sobre projeto baseado em objetos e mais ainda sobre análise baseada em objetos (COAD & YURDON, 1993).

A mudança do enfoque de programação estruturada para análise e projeto estruturado levou mais de uma década para se firmar. Espera-se um intervalo de tempo semelhante para se avançar do tratamento com a programação baseada em objetos para estudos de análise e projeto orientados a objetos.

Isto deve-se a visão errônea de uma metodologia para elaboração de rotinas computacionais e que ainda é adotada quase que invariavelmente, qual seja: as linguagens de programação existentes conseguem influenciar a estrutura da codificação de um programa; estudos para projeto se deixam influenciar pela forma de codificação que é adotada e o próprio projeto tende a influenciar a análise do sistema.

isoladamente se não fizer parte de uma classe, não se pode declarar uma variável global. Exemplos são Smalltalk e Eiffel (CANTÚ, 1996).

Em seguida vieram as linguagens híbridas, que são linguagens OOP montadas sobre linguagens existentes. Nestas pode-se fazer o que quiser, inclusive não aplicar nada dos princípios OOP. Exemplos são: C^{++}, Objective C, CLOS, Object PASCAL e o FORTRAN 90 (CANTÚ, 1996).

Recentemente surgiu a JAVA, uma linguagem OOP pura utilizada com mais ênfase para programação na Internet, mas que já ganha espaço para implementações de softwares em alguns ambientes gráficos.

3.3 Abstração na OOP

A abstração é a base fundamental para OOP. Atualmente, as linguagens de programação orientadas a objetos possibilitam basicamente dois tipos de abstração: a de dados e a procedimental (CANTÚ, 1996). Também existem diferentes níveis de abstração dentro de um projeto de software, desde o mais elevado onde se utiliza a narrativa textual da definição do problema para se criar uma solução, até o nível mais baixo com a geração do código de máquina.

A abstração na representação de dados está fundamentada no conceito de tipo de dado. A primeira geração de linguagens de programação lançou o conceito de tipo e a segunda geração incluiu a idéia de tipos de dados definidos pelo usuário, como registros e matrizes.

Finalmente, surgiu o conceito de tipo de dado abstrato que associa a representação dos dados com a operação usada para manipulá-los. Esta representação normalmente fica oculta aos usuários.

Na OOP uma abstração de dados é um conjunto de dados (uma classe) que descreve uma instância (um objeto).

Uma abstração procedimental é um conjunto de instruções que tem um função específica e limitada (PRESSMAN,1995).

As principais linguagens de programação introduziram declarações para saltos, ampliações e loops utilizando as estruturas de controle. Surgiu a idéia de sub-rotinas (na forma de procedimentos e funções). A abstração de sub-rotinas deve-se ao fato de se poder chamá-las já sabendo o que elas fazem sem precisar saber como fazem.

Na OOP uma abstração procedimental é uma operação ou método ligado a um objeto.

3.4 Conceitos Fundamentais da Tecnologia de Orientação a Objetos

3.4.1 Os Objetos

Objeto é a chave para se entender a tecnologia de orientação a objetos. Em um sistema de transmissão e distribuição de energia elétrica pode-se ter muitos exemplos de objetos reais: geradores, transformadores, chaves, linhas, técnicos, engenheiros, motores, oficinas, etc.

Estes objetos compartilham duas características: todos eles possuem estados e procedimentos. Por exemplo: engenheiros têm estados (nome, cor, fome) e têm procedimentos (beber, comer, dormir). Transformadores têm estados (tensão atual no primário, potência nominal, impedância, etc.) e procedimentos (mudar tap, fornecer tensão no secundário, etc.)

Objetos em softwares são modelados a partir dos objetos reais, por isso eles também possuem estados e procedimentos. Um objeto em software mantém seu estado em uma variável e implementa seus procedimentos com métodos. Logo, um objeto é um conjunto de variáveis e métodos relacionados em um software (CAMPIONE & WALRATH, 1996).

Por exemplo, representar técnicos num software como objetos em um programa de cadastro de pessoal para uma empresa de eletrificação, ou geradores como objetos de software em um programa de manutenção de equipamentos elétricos. Entretanto, pode-se também utilizar objetos em softwares para modelar conceitos abstratos. Um evento é um objeto comumente usado em programas com Interface Gráfica com Usuário (Graphic User Interface – GUI) para representar a ação de um usuário, como por exemplo, pressionar um botão do mouse ou uma tecla no teclado.

A figura 3.1 é uma representação visível de um objeto em um software.

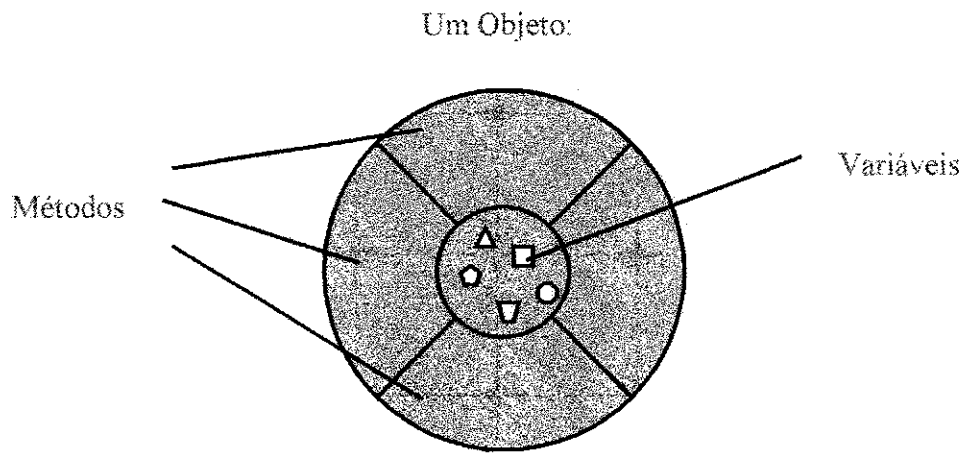

Figura 3.1 – Representação gráfica de um objeto em software

Tudo que um objeto em software possui (estado) e pode fazer (procedimentos) são expressos pelas variáveis e métodos relacionados a ele. Um objeto de software que modela um gerador tem variáveis que indicam o estado atual dele: tensão gerada 240 volts, velocidade de rotação 1800 RPM.

A figura 3.2 ilustra um transformador modelado como um objeto em um software.

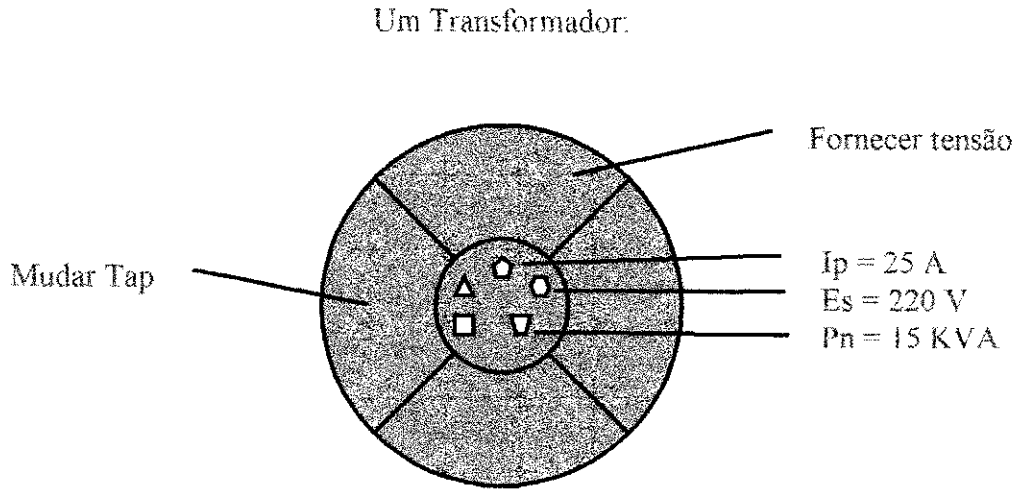

Figura 3.2 – Exemplo de um objeto modelado em software

Um objeto em software que representa um gerador também tem métodos: aumentar a potência de saída, aumentar a tensão gerada, etc.

Estas variáveis e métodos são normalmente conhecidos como variáveis de instância e métodos de instância para distinguí-los de variáveis de classes e métodos de classes.

Algo que um objeto não possui ou não pode fazer é excluído do objeto no software. Por exemplo, uma chave fusível provavelmente não possui um nome próprio, não pode mudar sua corrente nominal e não pode gerar tensão. Então, não existem variáveis ou métodos para estes estados e procedimentos nesta classe de chaves.

3.4.2 O Encapsulamento

Como pode-se ver das figuras anteriores, as variáveis formam o centro ou o núcleo do objeto. Os métodos encobrem e escondem estes núcleos de outros objetos no programa. O empacotamento de variáveis de objetos com seus respectivos métodos é chamado de "Encapsulamento". Tipicamente, o encapsulamento é usado para ocultar detalhes de implementação de outros objetos (CAMPIONE & WALRATH, 1996). Quando se quer mudar o tap de um transformador não se precisa conhecer o mecanismo de funcionamento,

precisa-se apenas saber qual o nível a mover. Similarmente, em programas de software não é necessário saber como uma classe é implementada e sim quais os métodos para invocar. Então, os detalhes da implementação podem mudar em qualquer instante sem afetar outras partes do programa.

Esta imagem conceitual de um objeto: um núcleo de variáveis empacotado por uma membrana de proteção, os métodos, é uma representação ideal de um objeto e é o ideal para projetistas de sistemas com orientação a objetos. Entretanto, frequentemente por razões de implementação ou eficiência, um objeto pode desejar expor algumas de suas variáveis ou esconder alguns de seus métodos.

Em muitas linguagens um objeto pode expor algumas de suas variáveis para outros objetos, permitindo àqueles inspecionar e até modificar as variáveis. Também, um objeto pode esconder métodos de outros objetos proibindo aqueles objetos de invocar os métodos. Um objeto tem completo controle sobre suas variáveis e seus métodos, permitindo ou não a algum outro objeto ter acesso a estes parâmetros. Isto só é possível em linguagens que possibilitem a definição de interfaces públicas e de áreas privadas para os objetos.

Portanto, o encapsulamento de variáveis e métodos dentro de um software possibilita o surgimento de três grandes conceitos para a programação orientada a objetos:

1- Modularidade: o agrupamento de dados e operações de processamento possibilita que o código fonte para um objeto possa ser escrito e mantido independentemente do código fonte para outros objetos.

2- Reutilização: uma classe juntamente com seus objetos podem ser facilmente passados a outro sistema em desenvolvimento, isto possibilita a criação de uma biblioteca de objetos que podem ser reutilizados dependendo da aplicação.

3- Ocultamento de informações: um objeto tem uma interface pública que os outros podem usar para se comunicarem com ele a partir de mensagens. Contudo, ele também é definido por uma área privada que esconde informações e métodos de outros objetos do programa e que podem ser mudados em algum instante sem afetar os que dependem dele.

3.4.3 As Mensagens

Um único objeto sozinho geralmente não é muito útil e usualmente aparece como um componente de um largo programa ou aplicação que contém muitos outros objetos. Um disjuntor sozinho é incapaz de alguma atividade, ele é útil somente quando outro objeto (circuito elétrico) interage com ele (corrente de disparo). Através da interação entre estes objetos, programadores alcançam funcionalidade da mais alta ordem e procedimentos mais complexos.

Objetos em softwares interagem e se comunicam através do envio de mensagens de um para o outro. Quando um objeto **A** quer que o objeto **B** realize um de seus métodos, **A** envia uma mensagem para **B**.

A figura 3.3 demonstra o envio de uma mensagem qualquer de um objeto **A** para **B**.

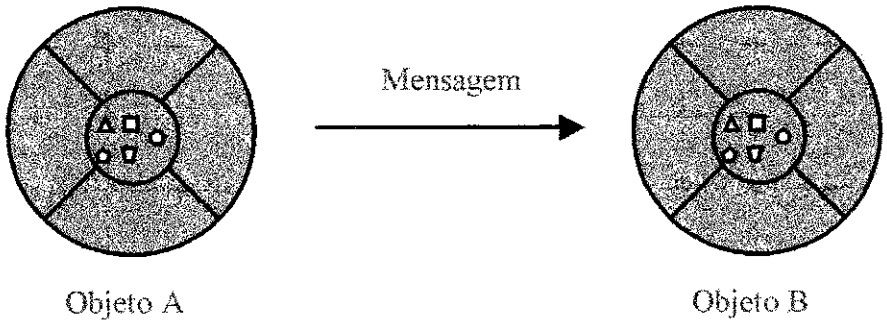

Figura 3.3 – O envio de uma mensagem entre dois objetos

Algumas vezes o objeto receptor precisa de mais informações, tal que ele saiba exatamente o que fazer. Por exemplo, quando se quer mudar a potência de saída em um

gerador, deve-se indicar qual é a nova condição da carga. Esta informação é passada ao longo da mensagem como parâmetro.

Três componentes compreendem uma mensagem:

1- O objeto para quem a mensagem está endereçada (o gerador),

2- o nome do método para performance (Mudar Potência de Saída),

3- alguns parâmetros necessários para o método (corrente de carga, fator de potência, etc.).

A figura 3.4 ilustra a relação entre estes três componentes.

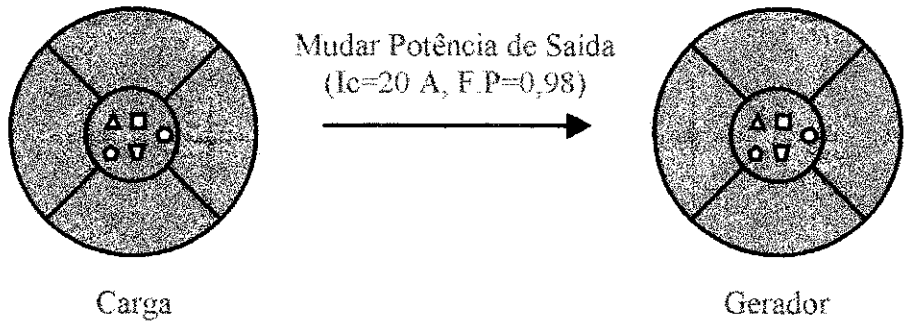

Figura 3.4 – Uma mensagem com parâmetros

Estes três componentes são informações suficientes para o objeto receptor realizar o método desejado. Nenhuma outra informação é necessária.

Fora o acesso direto às variáveis, as mensagens também fornecem suporte a todas as possíveis interações entre objetos.

3.4.4 As Classes

No mundo real as classes representam a união de coisas que compartilham as mesmas características (a classe dos engenheiros, a classe dos transformadores de 15 KVA, etc.). Na programação orientada a objetos isto também ocorre. Assim, temos a representação dos objetos do mundo real e de suas classes em um software.

Por exemplo, quando os fabricantes constróem os transformadores, eles têm a vantagem de que estes compartilham características pela fabricação de muitos a partir de um mesmo modelo. Seria muito ineficiente produzir um novo modelo de projeto para todo transformador individual que eles fabricassem.

Em programas com orientação a objetos também é possível ter-se muitos objetos do mesmo tipo que compartilhem características: geradores idênticos, carros iguais, botões, rótulos, etc.

A vantagem de objetos do mesmo tipo serem similares estar em se poder criar um modelo para estes objetos. Em software, modelos de objetos são chamados de classes.

Uma classe é um modelo de projeto, um protótipo que define as variáveis e métodos comuns a todos objetos de um certo tipo. Logo, um objeto é uma instância de uma classe.

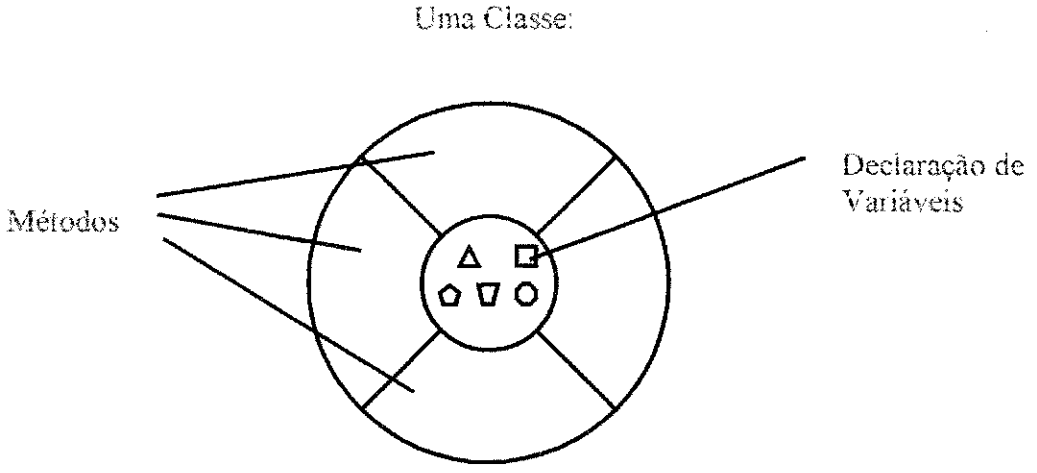

Figura 3.5 – Representação gráfica de uma classe em OOP

Por exemplo, uma classe de chaves declara variáveis de instância para conter: corrente de disparo, tensão nominal, corrente de operação, e outras características para cada objeto. Esta classe também declara e provê implementações para os métodos de instância que permitem ao circuito disparar a proteção, religar a chave, etc.

A figura 3.6 ilustra, graficamente, o modelo dessa representação.

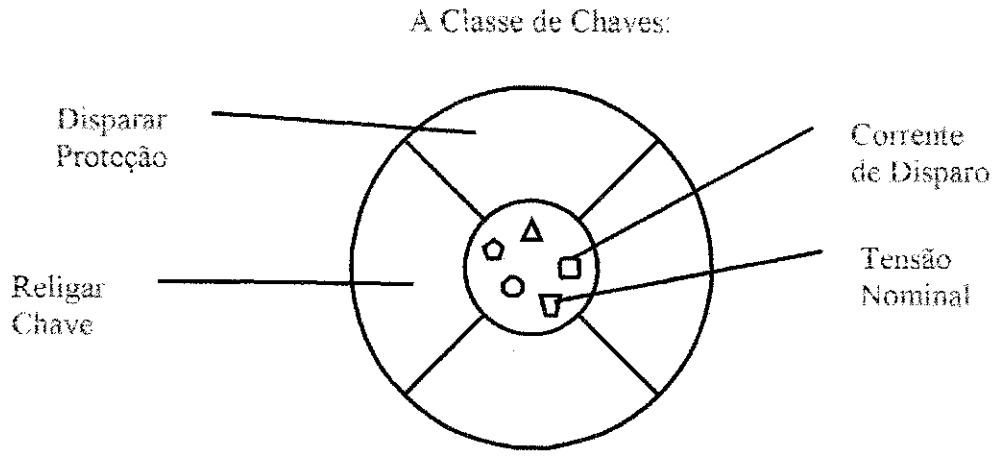

Figura 3.6 – Exemplo de uma classe modelada em OOP

Os valores das variáveis de instância são providos por cada instância da classe. Então, depois que se cria uma classe, criam-se as suas instâncias antes de usá-la. Quando se cria uma instância de uma classe, cria-se um objeto de tipo e o sistema aloca memória para a variável de instância declarada pela classe. Assim, pode-se invocar os métodos de instância do objeto para fazer alguma coisa. Instâncias da mesma classe têm as mesmas implementações de métodos.

Em adição a variáveis e métodos de instância, classes podem também definir variáveis e métodos de classe. Pode-se ter acesso às variáveis e métodos de classe de uma instância de classe (objeto) ou diretamente da classe. Contudo, não é necessário criar instâncias em uma classe para usar variáveis e métodos de classe. Métodos de classe podem operar somente em variáveis de classe, eles não tem acesso a variáveis ou métodos de instância.

O sistema cria uma única cópia de todas as variáveis de classe para a classe a primeira vez que ele a encontra no programa. Todas as instâncias daquela classe tem suas variáveis de classe. Por exemplo, suponha que todos os transformadores tenham a mesma

potência nominal, neste caso definir uma variável de instância é ineficiente. Cada instância teria sua cópia da variável mas o valor seria o mesmo para todas as instâncias. Em uma situação como esta pode-se definir uma variável de classe que contém a potência nominal. Todas as instâncias teriam esta variável. Se um objeto modificar a variável, ela também será modificada para todos os outros objetos daquele tipo.

Como foi exposto na seção 3.3, uma classe pode ser definida como a descrição abstrata de um grupo de objetos, cada um deles com um estado específico mas todos capazes de executar as mesmas operações (CANTÚ, 1996). Um exemplo prático de uma abstração de dados, na linguagem de programação *Visual Basic (versão 3.0)*, pode ser visto no trecho a seguir:

```
TYPE BarraCL
        Nome As String * 10
        Número As Integer
        MóduloTensão As Double
        ÂnguloTensão As Double
        PotênciaAtiva As Double
        PotênciaReativa As Double
End TYPE

:

For I = 1 To TotalBarras
        Dim Barra(I) As BarraCL
Next I
```

Onde *BarraCL* é uma classe que define seus objetos (*Barra(I)*) em função de seus dados: *Nome*, *Número*, etc.

Esta linguagem emprega o termo "tipo de dado" em lugar de "classe" e representa um objeto como um caso do tipo de dado, uma instância (NELSON, 1994). Logo, uma classe é uma definição de tipo que tem alguns campos (os dados que representam o status

de um objeto dessa classe) e alguns métodos (as operações), que geralmente dependem do status do objeto.

Para o exemplo anterior, as abstrações procedimentais ligadas aos objetos Barras poderiam ser implementadas em métodos para mudança de valores em algumas de suas variáveis (dados), como no trecho a seguir:

Procedimento: Determinar MóduloTensão para BarraN;

Repetir até que [PotênciaAtva^{N} - PotênciaAtva^{N-1} ≤ Tolerância];

1.Calcular PotênciaAtiva para estimativas iniciais;

2.Calcular Diferença na PotênciaAtiva fornecida e calculada em 1;

3. Determinar elementos do Jacobiano;

⋮

Fim;

Fim Repetição;

Fim Procedimento.

Este nível de abstração procedimental é intermediário entre o da narrativa da solução vista no método iterativo de Newton-Raphson (seção 2.3) e o nível definido por uma linguagem de programação.

Dentro de um programa as classes têm duas finalidades principais:

1- Elas definem a abstração com as quais se relacionam. Pode-se usá-las para definir entidades do mundo real, não importa o quão complexos sejam. Se a entidade for muito complexa pode-se usá-las para descrever seus sub-elementos (ou sub-sistemas).

2- Elas são a base da modularidade de um programa. De acordo com esta idéia, cada classe deve ocultar ou encapsular alguns de seus elementos.

3.4.5 Classes x Objetos

As ilustrações de objetos e classes são muito similares, a diferença entre classes e objetos é frequentemente a fonte de alguma confusão. No mundo real é óbvio que classes

não são elas próprias os objetos que elas descrevem. O modelo de um tipo de chave pode não ser uma chave fisicamente. Entretanto, é um pouco mais difícil diferenciar classes e objetos em softwares. Isto deve-se porque objetos em softwares são meramente modelos eletrônicos de objetos do mundo real ou conceitos abstratos em primeiro lugar e também porque muitas pessoas usam o termo objeto inconsistentemente para se referir às classes e instâncias.

Nas figuras, as classes não são sombreadas para indicar que elas representam um modelo de um tipo de objeto do que um objeto propriamente. Por outro lado, um objeto é sombreado indicando que ele atualmente existe e que pode ser usado.

Objetos provêm os benefícios da modularidade e do ocultamento de informações. Classes provêm o benefício da reutilização. Fabricantes de geradores usam o mesmo modelo muitas e muitas vezes para construir muitos geradores. Programadores de softwares usam a mesma classe e então o mesmo código muitas e muitas vezes para criar muitos objetos.

3.4.6 A Herança

Os objetos são definidos em termos de suas classes. Conhece-se muito sobre um objeto conhecendo-se a sua classe. Por exemplo, ainda que não se saiba o que é um "corsa", se for dito que é um carro então sabe-se que ele tem quatro rodas, pedais, direção, etc.

Sistemas orientados a objetos permitem que classes sejam definidas em termos de outras classes. Por exemplo, síncronos e de indução são tipos diferentes de geradores de corrente alternada. Em software eles são considerados sub-classes da super-classe *geradores de corrente alternada*.

A figura 3.7 ilustra uma herança em OOA e OOD.

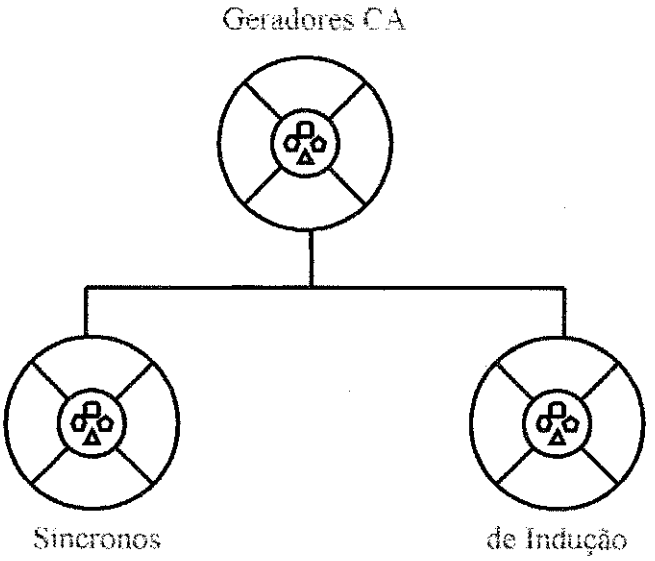

Figura 3.7 – A hierarquia de classes

Cada sub-classe herda estado (na forma de declarações de variáveis) da super-classe. Síncronos e de indução compartilham alguns estados: tensão nominal, potência nominal e semelhanças físicas. Também, cada sub-classe herda métodos da super-classe. Síncronos e de indução compartilham alguns procedimentos: mudar velocidade de rotação, mudar potência de saída, etc.

Entretanto, sub-classes não estão limitadas aos estados e procedimentos providos à elas por sua super-classe, elas podem adicionar ou suprimir variáveis e métodos aos que herdam da classe pai. Em geradores de indução, não há uma variável que represente a tensão fornecida ao campo.

As sub-classes podem também aproveitar métodos herdados e provê implementações especializadas para estes métodos. Por exemplo, em geradores síncronos mudar a potência de saída possui um procedimento diferente que em geradores de indução.

O analista não está limitado exatamente a uma parte da herança, a árvore da herança ou hierarquia de classes pode ser tão profunda quanto necessário. Métodos e variáveis são herdadas para baixo através dos níveis.

Sub-classes são providos de procedimentos especializados com base em elementos comuns providos pela super-classe. Através do uso da herança os programadores podem reusar o código de super-classes muitas vezes.

Na prática, a herança é uma maneira de evitar repetição do código. Em vez de usar técnicas de copiar e colar para montar duas classes similares é muito mais fácil usar uma como classe pai da outra. Uma sub-classe herda os campos e métodos da classe pai. Além de diminuir o código economiza-se parte do tempo em depurar e em lidar com futuras alterações de maneira mais fácil porque há somente uma versão do código.

3.4.7 O Polimorfismo

O polimorfismo permite referenciar objetos de classes diferentes por meio da mesma variável de programa. Ele também permite executar operações com essa variável de diferentes maneiras, de acordo com o objeto e com a classe atualmente associados a essa variável.

Como exemplo, pode-se declarar uma variável genérica da classe pai geradores e associa-la à objetos das sub-classes síncronos e de indução.

Supondo que ambas sub-classes tenham um procedimento chamado *mudar potência de saída*, cada um com implementação diferente, quando se aplica este método à geradores, este procedimento é chamado referente ao tipo atual do objeto gerador. Se este for atualmente da sub-classe síncrono, a potência é modificada de uma forma, se o objeto atual for da sub-classe de indução, a potência é modificada de outra forma.

3.5 As Bibliotecas de Ligação Dinâmica (DLL)

Uma biblioteca de ligação dinâmica (Dynamic Link Library – DLL) é um conjunto de funções e procedimentos organizados de tal forma que possibilitem o seu uso por diversos programas em tempo de execução. É como um arquivo executável, mas que

geralmente não funciona sozinho e deve ser chamado por um programa mestre para auxilia-lo em alguma tarefa usualmente comum e que por isso é utilizado como um arquivo de biblioteca.

Existem duas formas de ligação entre arquivos executáveis e suas bibliotecas: ligações estática e dinâmica. Quando uma sub-rotina não está presente em um arquivo fonte mas é declarada, o compilador deve reconhecer seus parâmetros e tipos, caso contrário aparecerá uma mensagem de erro. Em seguida, para uma ligação estática ocorrer, o programa "linker" busca todas as sub-rotinas compiladas de uma biblioteca estática e as inclui no arquivo executável, este possui então todos os códigos do arquivo fonte e das sub-rotinas envolvidas. Isto não acontece em uma ligação dinâmica, o programa "linker" simplesmente utiliza as informações contidas na declaração "external" da sub-rotina para incluir algumas tabelas internas ao arquivo executável. Estas tabelas irão conter os endereços das funções e procedimentos existentes na DLL e que estão na memória. Então, quando o arquivo executável é carregado na memória, ele carrega todas as DLLs que ele precisa, se alguma DLL não for encontrada o programa não começa e é dada uma mensagem de erro. Assim, cada vez que há uma chamada a uma função ou procedimento externo a chamada é enviada para a DLL correspondente (CANTÚ, 1996).

De fato, a ligação dinâmica não surgiu como uma ferramenta da tecnologia de orientação a objetos. A correlação só é possível com a idéia do encapsulamento, ou seja, agrupar várias seções de código que desempenham uma função ou determinam um procedimento, desde que sejam considerados métodos relativos a objetos dentro de uma determinada classe, de tal forma que não façam parte de um único programa mas que possam ser usados por vários outros e ao mesmo tempo, através da passagem de parâmetros e tipos.

Definir uma DLL como um encapsulamento, fornece várias vantagens:

1. Economia de memória do sistema. As DLLs são carregadas na memória apenas uma vez, podendo ser usadas por diferentes programas. Deve-se entender que o uso de uma mesma DLL por diversos programas não significa o uso em paralelo, ao mesmo tempo de uma única função, mas em série com cada programa esperando sua vez de usá-la.
2. Pode-se utilizar as DLLs escritas em uma linguagem, por programas escritos em linguagens diferentes, desde que a passagem de parâmetros e tipos seja correta e compatível.
3. Fica mais fácil e rápido atualizar apenas uma quantidade pequena de código contida numa DLL, em vez de ter que revisar todo um programa fonte, se este contém o código da DLL repetido em diversos trechos onde se faz necessário. Ou seja, tendo-se um programa muito grande que exija frequentes atualizações e consertos de erros, dividi-lo em diversos arquivos executáveis e diversas DLLs permite que se entregue a um cliente apenas a DLL que tiver sido atualizada e não um único grande arquivo executável.

A implementação de DLLs não é tão fácil, vários autores possuem publicações ensinando alguns procedimentos em diversas linguagens. Para cada linguagem há uma implementação diferente. Contudo, os passos a seguir são praticamente os mesmos. No caso da utilização temos:

1. Os procedimentos e as funções são declaradas juntamente com seus parâmetros.
2. As funções e os procedimentos devem ser referenciados com uma definição externa a uma DLL.
3. Chamar normalmente uma função ou procedimento em alguma parte do código fonte onde se deseja utilizá-la.

Alguns autores preferem o uso de ligações estáticas à criação de DLLs devido a algumas desvantagens destas:

1. A chamada às funções externas em DLLs as vezes pode não ser tão simples, tornando o sistema muito instável em alguns casos. Ou seja, alguma parada crítica pode ocorrer tanto na execução do programa quanto no sistema operacional instalado no computador.
2. Pode-se pensar em poupar espaço em memória com a utilização de DLLs, no entanto pode acontecer justamente o contrário. Às vezes, quando se carrega uma DLL pode-se estar carregando-a completa, se ela for muito grande pode-se ocupar mais espaço que uma biblioteca estática onde a ligação é mais seletiva e permite se incluir no programa somente o código que realmente se vai usar.
3. DLLs tipicamente exportam rotinas, funções e procedimentos e não objetos com seus métodos e atributos. Utiliza-se a orientação a objetos para definir a estrutura do aplicativo, se este for dividido em DLLs pode-se perder esta característica caso estes conceitos não sejam utilizados na forma como são fundamentados na OOP.

Por isso, muitos analistas de sistemas desconsideram o uso das DLLs como uma forma de encapsulamento para a OOP. Para os demais profissionais que desenvolvem seus sistemas para determinados problemas em suas respectivas áreas, essa idéia não é de todo descartável, já que é bastante vantajoso poder utilizar os recursos das DLLs para "encapsular" funções e procedimentos que, por exemplo, poderão ser utilizados por diversos programas na solução de problemas matemáticos. Assim, pode-se ter DLLs com funções de inversão de matrizes, interpolação, obtenção das raízes de equações, integração numérica, entre outras.

CAPÍTULO 4

Desenvolvendo um Software para Engenharia

4.1 Introdução

O capítulo 2 abordou a definição do problema de engenharia, fundamental para a análise do sistema. O capítulo 3 forneceu os conceitos ligados a tecnologia de orientação a objetos, necessários à todas as fases do processo de desenvolvimento de softwares para a engenharia. Processo este denominado de "*Engenharia de Software Orientada a Objetos*". Como o próprio nome sugere, este baseia-se em princípios desenvolvidos em engenharia para aplicar os conceitos adquiridos com esta nova tecnologia de programação, de forma a obter um software dentro dos critérios de qualidade estabelecidos atualmente no mundo: *desempenho, estabilidade, usabilidade, manutenabilidade, e padronização*.

Este processo fundamenta-se no uso de procedimentos para determinar um conjunto de métodos e que a partir de ferramentas possibilitam a criação de um software racional e de alta qualidade (COLEMAN, 1996). Cada etapa é documentada em relatórios contendo seu planejamento.

A figura 4.1 ilustra as etapas do processo de desenvolvimento de softwares a partir desta abordagem.

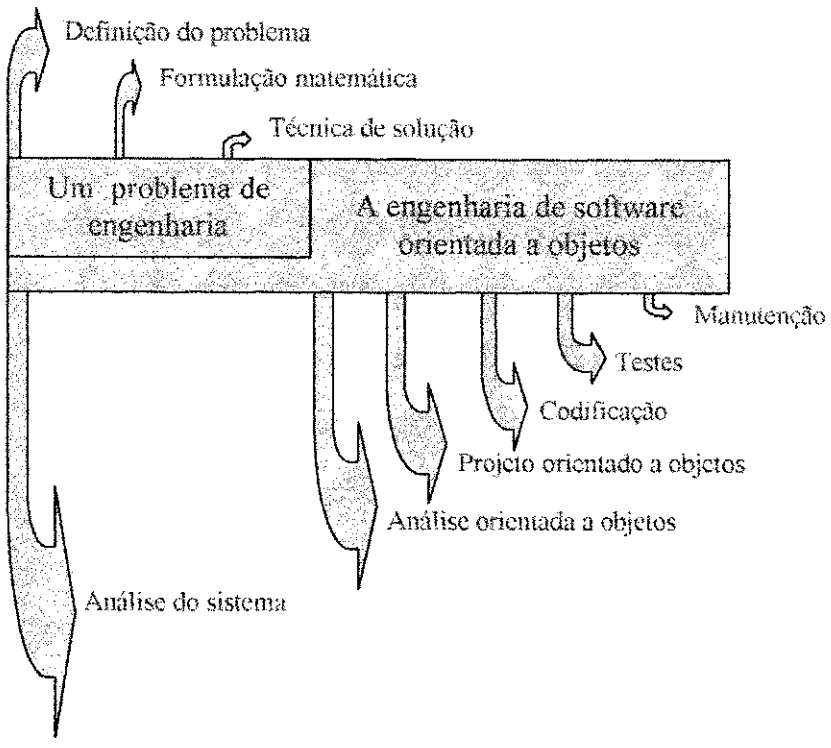

Figura 4.1 – O processo de desenvolvimento de um software para engenharia

4.2 Determinando o Problema de Engenharia

Na primeira fase do processo, o levantamento e tratamento do problema de engenharia é feito em três etapas: *(1) definição do problema, (2) formulação matemática* e *(3) técnica de solução*. O capítulo 2 abordou todas estas, utilizando como modelo o fluxo de carga em sistemas de potência.

Inicialmente, deve-se determinar o problema como um todo. Os objetivos devem ser traçados e os requisitos necessários para solucionar o problema devem ser delineados. Nesta etapa podem ser feitas algumas perguntas que facilitem o processo. Para o modelo proposto temos:

1. O que é fluxo de carga?
2. Para que serve o seu estudo?
3. Quais os problemas no sistema que necessitam de uma análise de fluxo de carga?
4. O que fornece o estudo de fluxo de carga?

5. Em que condições de operação do sistema são feitas as análises de fluxo de carga?
6. Quais são os dados fornecidos inicialmente para solução do problema?
7. Qual a melhor estrutura para a solução?
8. Quais as restrições técnicas para a solução?

Estas são algumas das questões básicas que surgem para o caso exemplificado. Analogamente, questões semelhantes podem ser elaboradas para cada problema estudado em engenharia. O aprofundamento na definição do problema dependerá do caso. Por exemplo, quando existir comunicação não apenas com os seres humanos mas também a partir de dispositivos de coleta e transmissão de dados em tempo real, ou a partir de bancos de dados. Cabe ao engenheiro a tarefa de definir o problema com exatidão, evitando os erros de análise do sistema, a serem abordados na seção 4.3.1.

Determinado o problema, o sistema físico deve ser representado por um modelo matemático adequado. Podem existir diferentes modelos. Contudo, a formulação matemática escolhida deve possibilitar rapidez, precisão e confiabilidade na obtenção dos resultados esperados. Estes podem ser critérios para avaliar a qualidade (neste caso desempenho) dos modelos de análise e projeto.

Como a solução das equações desenvolvidas na formulação matemática será obtida pelo computador, deve ser utilizada uma técnica numérica para transformar as equações, que em muitos casos são não-lineares, diferenciais ou trigonométricas, em equações contendo apenas as quatro operações aritméticas suportadas por esta máquina. No entanto, devem ser observadas as três condições apresentadas anteriormente: rapidez, precisão e confiabilidade, para selecionar-se o método adequado para cada caso.

Um melhor entendimento das etapas desta fase é obtido com o acompanhamento do capítulo 2.

4.3 A Engenharia de Software Orientada a Objetos

A segunda fase do processo utiliza a engenharia de software orientada a objetos para desenvolvimento da rotina computacional que tratará o modelo matemático formulado a partir do sistema físico.

De acordo com a figura 4.1, a segunda fase abrange seis etapas: *(1) análise do sistema, (2) análise orientada a objetos, (3) projeto orientado a objetos, (4) codificação, (5) testes* e *(6) manutenção.*

Uma outra terminologia sugere que a análise do sistema juntamente com a análise orientada a objetos que é uma análise de requisitos para o software formam a *fase de definição;* o projeto, a codificação e os testes fazem parte da *fase de desenvolvimento* e por fim vem a *fase de manutenção.*

4.3.1 A Análise do Sistema

O sistema é o meio onde o software irá interagir. Fazem parte dele os seres humanos (usuários), os bancos de dados, as máquinas ou equipamentos (dispositivos, drivers, sensores, etc.) que fornecem dados em tempo real ou não a partir de interfaces com o programa.

Nesta etapa é preciso:

1. Estabelecer todos os requisitos do sistema;
2. Levantar as tarefas de cada elemento constituinte do sistema para então determinar com exatidão a função do software e com isso planejar as interfaces correspondentes;
3. Executar análise de viabilidade técnica, econômica e legal;
4. Estabelecer os critérios para avaliação da qualidade dos modelos de análise e projeto, e do próprio software.
5. Determinar as restrições do projeto (limitações técnicas, prazos, custos, etc.);

6. Determinar a capacidade do hardware em suportar o software a ser desenvolvido;
7. Levantar as necessidades dos usuários com relação ao software: informações a serem fornecidas ao programa, informações obtidas a partir do programa, etc. Uma análise dos fatores humanos é adequada nesta fase, para posteriormente projetar-se as interfaces para o programa.

A etapa de definição do problema faz parte da análise do sistema, como demonstra a figura 5.1. Nela determinamos todos os aspectos técnicos envolvidos com o problema de engenharia.

Alguns dos estudos envolvidos com a análise do sistema podem não ser necessários dependendo do caso, por exemplo: no projeto do programa para análise de fluxo de carga para o meio acadêmico, idealizado no capítulo 2, não se faz necessário a análise da viabilidade econômica que leva em consideração a relação custo-benefício.

Em outros casos alguns estudos podem fornecer as respostas para outros, por exemplo: a determinação da função, do desempenho requerido e as restrições técnicas do projeto do software para fluxo de carga fornecem uma viabilidade técnica para o propósito do programa desenvolvido, com base nos objetivos traçados no capítulo1.

O relatório da análise deve fornecer um modelo que possa conter entre outros, uma narrativa textual do sistema (definição do problema), uma tabela com cada elemento, suas características, funções e interligações e um diagrama.

A partir do que foi exposto nas etapas 1,2 e 3 da fase de determinação do problema de análise de fluxo de carga, podemos traçar um diagrama de entidade/relacionamento para um modelo do sistema contendo seus elementos e suas interligações (figura 4.2).

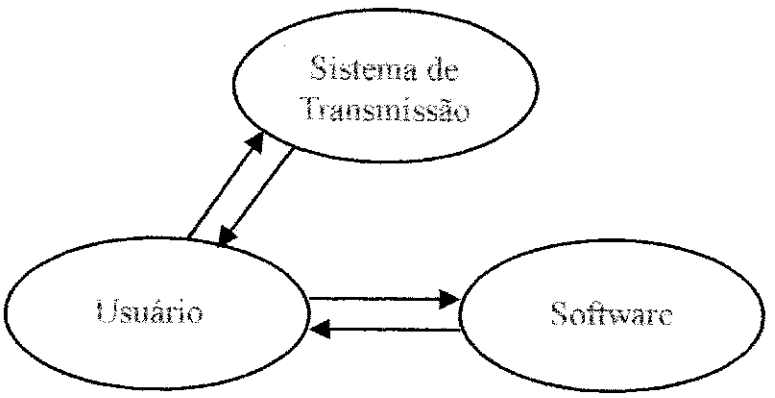

Figura 4.2 – Diagrama de entidade / relacionamento para análise do sistema

Para o problema de engenharia proposto como modelo neste trabalho, a análise do sistema resume-se apenas à três componentes: o sistema de transmissão, o usuário (engenheiro ou técnico) e o próprio software, que é a solução computacional para o problema real.

A análise do sistema é de grande importância para que o engenheiro situe-se no universo do seu problema. Porém, seu objetivo principal é o levantamento das características desse sistema que determinam o problema e quais os requisitos necessários para implementar a solução computacional que é o software. A seção seguinte aborda a análise de requisitos especificamente para este elemento, a partir dos conceitos de orientação a objetos.

4.3.2 Análise de Software Orientada a Objetos

A análise e o projeto orientado a objetos apesar de serem a base para criação de softwares com alto grau de qualidade, ainda estão em fase de amadurecimento com relação às suas aplicações. Foi no final da década passada que o projeto começou a ganhar impulso entre os analistas e programadores. Atualmente, também passaram a aplicar a orientação a objetos na análise de requisitos para o software.

Com base no que foi exposto no capítulo 3, essa nova metodologia de análise consiste em determinar os objetos necessários para o software, definir seus atributos e seus métodos, além de definir as conexões de mensagens entre eles. É criado então um modelo de análise de requisitos para o programa contendo essa estrutura.

Identificar todos os objetos para um software não é tão fácil. Deve-se ter o maior conhecimento do domínio do problema (análise do sistema), mesmo assim, qualquer objeto real é um potencial objeto em software. É necessário se estabelecer requisitos de seleção que possibilitem a escolha daqueles que terão a melhor funcionalidade dentro da solução desejada, evitando duplicidade de funções ou mesmo a criação de blocos obsoletos ou sub-utilizados (ver encapsulamento – cap.3). Contudo, a escolha pode também se basear em aspectos puramente subjetivos e que dependerá do problema.

Neste processo estabelecemos basicamente quatro características de seleção. São elas:

1. Outros elementos do sistema (ser humano, máquinas, dispositivos, etc.) que fornecem ou recebem informações do software essenciais à solução do problema provavelmente serão objetos no programa;
2. Os objetos devem ter um conjunto de atributos e um conjunto de métodos que possibilitem mudar seus valores;
3. Sempre que houver uma solicitação do objeto todos os seus atributos e métodos devem estar aptos a serem aplicados;
4. Os objetos devem se inter-comunicar trocando informações relevantes para que um ou outro possa executar seus métodos.

Como exemplo tomemos o modelo de fluxo de carga. A partir da definição do problema (capítulo 2) contida na análise do sistema, os possíveis objetos para o programa

são: *as cargas, os transformadores, os geradores, as linhas de transmissão, os barramentos (barras ou nós) e os usuários (engenheiros ou técnicos)*.

Os cinco primeiros objetos reais fazem parte do sistema assim como o usuário. Contudo, estes não fornecem ou recebem informações diretamente do software. O usuário é o elemento intermediário que se comunica com o programa, sendo então o único objeto externo visto por este.

Logo, neste caso podem existir dois modelos de análise de requisitos do programa:

- *Modelo 1*: quase tudo pode ser visto como um objeto para o programa: pessoas, cargos, locais, equipamentos, etc. Para o fluxo de carga pode-se modelar apenas o usuário como um objeto. Neste caso ele possuirá todas as variáveis e métodos dos outros que não foram modelados. No entanto, o projeto de um software orientado a objetos que possui apenas um módulo pode não possibilitar o tratamento dos outros conceitos que caracterizam esta metodologia (encapsulamento, mensagens, polimorfismo, hierarquia, etc.);
- *Modelo 2*: como os geradores, os transformadores, as cargas, etc., possuem atributos (tensão, corrente, potência, reatância, etc.) e métodos (mudar tensão, mudar potência, etc.) e existe possibilidade de comunicação entre cada um para aplicar seus métodos e mudar seus atributos, eles podem ser representados individualmente no programa. Aqui, o usuário não pode ser modelado como objeto no software, ele é apenas uma entidade externa que fornece as informações de entrada necessárias para a operação da solução e recebe as informações de saída para atuar nos demais elementos do sistema (geradores, transformadores, cargas, etc.). A comunicação inter-objetos no programa simula uma troca de informações em tempo real entre os objetos e o próprio software, caso estes estivessem diretamente conectados ao programa como objetos reais do sistema.

O projeto do software para fluxo de carga utilizou o primeiro modelo de análise, no qual a execução dos cálculos é realizada por uma rotina estruturada (ANEXO I) em MATLAB® (MATHWORKS, 1995). Tendo sido as interfaces criadas com *Visual Basic 3.0* da Microsoft (NELSON, 1994), que possuía limitações para implementar todos os conceitos de OOP, mas que na época era a única ferramenta disponível.

Por isso, apesar do usuário e do software serem os únicos elementos do sistema a trocarem informações, o usuário não foi modelado realmente como objeto no programa, ele ficou oculto. Ainda assim, possui estados e métodos. Seus atributos são as variáveis de entrada que ele fornece, seus métodos são os eventos que ele dispara para solicitar as mudanças nestes atributos, mudanças estas fornecidas como as variáveis de saída. Estes eventos são interpretados como mensagens de comando entre o "objeto usuário" e outro objeto no programa, como um botão de comando, botão de opção, barra de deslocamento, etc., todos objetos da super-classe Ferramentas, que também possuem estados e procedimentos (métodos) que são solicitados pelo usuário ou por outro objeto. Estas ferramentas são fornecidas pela maioria dos pacotes de programação híbrida e utilizadas para dar suporte ao projeto das interfaces. Assim, cada solicitação do usuário para mudança em algum dos seus atributos é passada como uma mensagem para um objeto ferramenta que executa um método para possibilitar esta mudança.

Para o engenheiro, definir as variáveis para seus objetos é uma função mais técnica e por isso mais fácil. Há duas possibilidades:

1. Os atributos de um objeto são variáveis indispensáveis para estes que precisam ser manipuladas de alguma forma para possibilitar a execução dos cálculos.
2. Os atributos de um objeto são variáveis necessárias à outros para que estes manipulem seus próprios atributos e possibilitem a execução dos cálculos.

De qualquer forma, um atributo para um determinado objeto só pode existir se for criado um método para manipulá-lo.

Como vimos, as operações dos objetos basicamente podem ser agrupadas em duas:

1. Métodos de manipulação de dados.
2. Métodos de monitoração e controle de tarefas ou eventos.

Com base na definição do problema e a partir do primeiro modelo de análise para o fluxo de carga, no qual o único objeto a se comunicar com o software é o usuário, a tabela 4.1 apresenta os dados iniciais que devem ser fornecidos ao programa. Logo, esta tabela pode ser utilizada como referência para se estabelecer os atributos e os métodos para este objeto (tabela 4.3).

Tabela 4.1 – Dados Iniciais Fornecidos ao Fluxo de Carga

Tipos	Parâmetros
Dados Gerais	N.° de Linhas
	N.° de Barras
	Potência Base (MVA)
	N.° Iterações Máximas
	Tolerância
Dados das Linhas (pu)	Barra Inicial
	Barra Final
	Resistência
	Reatância
	Admitância Shunt/2
Dados dos Geradores[1] *(pu)*	N.° da Barra
	Tipo de Barra
	Módulo da Tensão
	Ângulo da Tensão
	Potência Ativa
	Potência Reativa

A análise dos requisitos para o software requer agora a classificação dos objetos, ou seja, defini-los em suas respectivas classes. De acordo com o capítulo 3, a classe é a

[1] Que nesse caso deveria ter sido definido como "Dados das Barras".

generalização do objeto. Todos aqueles que possuem os mesmos estados e procedimentos são agrupados em um único conjunto. Assim classificamos os objetos ferramentas (tabela 4.2).

Também neste ponto, deve-se estabelecer as conexões de mensagem entre cada objeto, que como foi definido, pode ser uma comunicação de serviço para que o recebedor execute um de seus métodos (figura 4.3 e tabela 4.3).

Com os resultados da análise orientada a objetos é criado seu modelo (diagramas de comunicação, tabelas, etc.) e a documentação de todo esse planejamento.

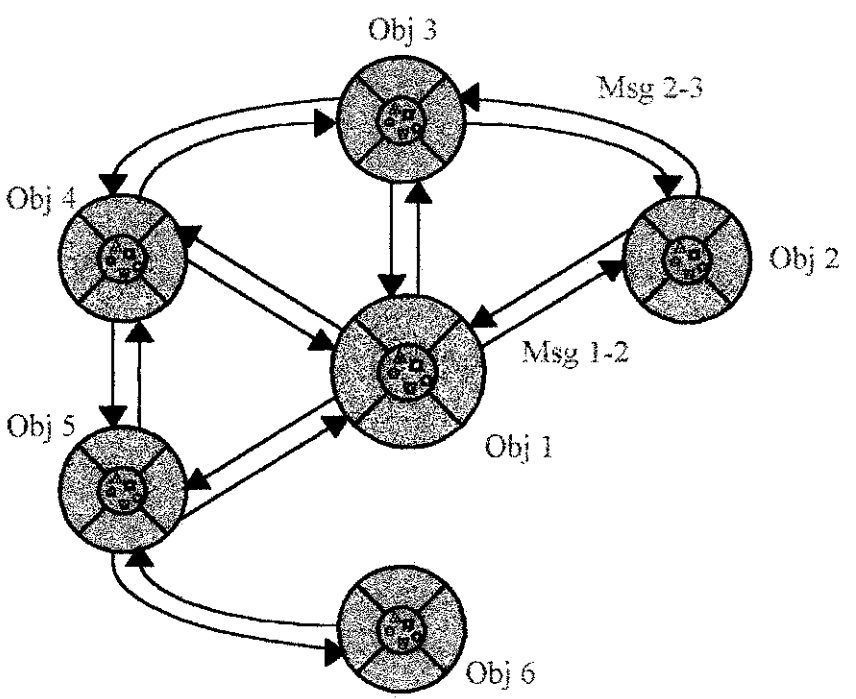

Figura 4.3 – Diagrama de objeto / relacionamento para análise de software

A figura 4.3 mostra um diagrama de objeto/relacionamento para o primeiro modelo simplificado da análise de requisitos com orientação a objetos desenvolvido para a interface do software de fluxo de carga.

Este diagrama quase sempre estará incompleto devido a quantidade de objetos e caminhos de mensagens a serem representados. Por isso deve também haver representação

textual na forma de tabelas, relacionando cada objeto, seus atributos, métodos e suas conexões de comunicação (tabelas 4.2 e 4.3).

Tabela 4.2 – Classificação dos Objetos

Classe	Nome do Objeto	Especificação do Objeto
Usuários	Usuário	Obj 1
Comandos	Comando 1	Obj 2
	Comando 2	Obj 4
	Comando 3	Obj 5
	⋮	⋮
Telas	Tela 4	Obj 3
	Tela 6	Obj 6
	⋮	⋮

Tabela 4.3 – Objetos para a Interface do Software para Fluxo de Carga

Especificação do Objeto	Atributos do Objeto	Métodos do Objeto	Tipo do Método	Conexões de Mensagem
Obj 1	(ver tabela 4.1)	Abrir Arquivo	Evento	Msg 1-2
		Alterar Dados	Evento	Msg 1-3
		Pedir Execução	Evento	Msg 1-4
		Fornecer Saída	Evento	Msg 1-5
Obj 2	Ponteiro Ativado	Click	Procedimento	Msg 2-3
Obj 3	Conteúdo Vazio	Exibir Dados Existentes	Procedimento	Msg 3-4
	Conteúdo Cheio	Receber Alteração	Procedimento	
Obj 4	Ponteiro Ativado	Click	Procedimento	Msg 4-?
Obj 5	Ponteiro Ativado	Click	Procedimento	Msg 5-6
Obj 6	Conteúdo Vazio	Mostrar Saída	Procedimento	Msg 6-?
	Conteúdo Cheio			
⋮	⋮	⋮	⋮	⋮

É evidente que nem todos os objetos foram representados. Contudo, este modelo simplificado demonstra a possibilidade de se utilizar as ferramentas existentes nas linguagens de programação para possibilitar o projeto de um software orientado a objetos.

4.3.3 Projeto de Software Orientado a Objetos

O projeto é a primeira etapa da fase de desenvolvimento de um modelo efetivo de software para solução do problema real. Este modelo deve fundamentar-se em todos os conceitos abordados pela tecnologia de orientação a objetos. Para tanto, o projeto deve dar prosseguimento aos passos que foram iniciados com a análise de requisitos para o software, refinando-os ainda mais e complementando o processo com a representação e o detalhamento dos módulos que contém os objetos, as estruturas de dados que contêm seus atributos e com a implementação procedimental para cada um de seus métodos. Nesta representação pode ser utilizada uma Linguagem de Projeto de Programa (Program Design Language – PDL) desenvolvida pelo próprio projetista ou uma padrão que usa um vocabulário em inglês e a sintaxe de uma linguagem de programação de alto nível.

O refinamento dos passos desenvolvidos na OOA é de fundamental importância para o projeto, já que na análise são especificados os objetos e suas classes que fazem parte do domínio da solução mas que em muitos casos também representam apenas o domínio do problema e que por isso, quando se chega na etapa de implementação de um protótipo, surge a necessidade de se criar objetos e então módulos adicionais necessários para a comunicação e armazenamento de informações entre os objetos já existentes. Por exemplo, como trabalhar com os atributos do objeto *usuário* sem que ele contenha uma funcionalidade que exceda os limites de um módulo coeso? Ou seja, um único objeto teria uma grande quantidade de métodos tornando-se muito "inchado". Uma alternativa é criar outros módulos com outros objetos para desempenhar estas funções (métodos). Como os botões de comando que possibilitam a interação entre o usuário e o software. No caso específico para o fluxo de carga, poderia ser criado um objeto abstrato denominado de *Método Numérico de Newton-Raphson*. Este se comunicaria com os demais a partir de mensagens. Seus atributos seriam as variáveis necessárias para que o objeto opere e seus

métodos poderiam ser: *formação da matriz admitâncias de barras, determinação do jacobiano*, entre outros. Em um nível maior de modularidade, ou seja, módulos cada vez mais coesos, poderíamos "quebrar" este objeto em vários outros: *objeto matriz admitâncias de barras, objeto jacobiano*, etc., cada um contendo seus atributos e suas operações.

Outro aspecto do projeto é o tratamento com dispositivos de entrada e saída (E/S). Quando se projeta um software utilizando os conceitos de orientação a objetos, pode-se representar estes dispositivos como objetos no software para controlar suas funções. Assim, pode ser criado um módulo contendo o objeto impressora e outro contendo o objeto gravador, ambos controlarão seus respectivos equipamentos reais e podem ter além de sua área privada, uma interface pública que possibilite a comunicação com o usuário e com os demais objetos.

Quando se utiliza uma metodologia, como a que será abordada no capítulo 5, para projetar as interfaces gráficas com o usuário, pode-se trabalhar tanto com OOD como com métodos estruturados para desenvolvimento do software. As telas de interfaces apenas trabalham com as informações públicas que são necessárias para que o módulo (objeto em OOD) desempenhe suas funções, como por exemplo o nome de um arquivo para gravação de algum conjunto de dados.

Logo, no projeto deve-se definir todos os objetos necessários para que o software opere completamente e não só aqueles ligados ao domínio do problema da engenharia (geradores, cargas, barras, etc.). O programa é então dividido em módulos que desempenham uma ou um conjunto mínimo e específico de funções correlatas (COOD & YOURDON, 1993). Módulos coesos possibilitam uma maior reutilização por outros sistemas a serem projetados, mas também significam uma quantidade maior de módulos e de interfaces de comunicação. O que inevitavelmente pode levar a um maior consumo de memória para armazenamento de dados e aumento de tempo no processamento.

Diferente de outros métodos, o projeto orientado a objetos depende da linguagem de programação para implementar os conceitos de classe e objeto em seu modelo. Pois algumas linguagens fazem isso diretamente, outras utilizam artifícios como o uso de "pacotes" no lugar das classes e o uso da "determinação de tipos definidos pelo usuário" no lugar dos objetos. Neste caso o projetista tem duas alternativas: **(i)** definir já nesta etapa a linguagem de programação que vai utilizar para implementar o modelo do projeto, então o nível de abstração procedimental e de dados pode ser muito baixo, o que é ideal para a codificação ser estabelecida mais rapidamente, **(ii)** deixar a definição da linguagem para a etapa de codificação e utilizar uma PDL para descrever os módulos e suas interligações, o nível de abstração será mais alto pois deve-se ignorar nesta fase a forma de implementação de classes e objetos, porém não se perderá no projeto o tempo que seria dedicado a uma possível programação.

O trecho a seguir mostra o uso de uma PDL na descrição de um módulo para o objeto *usuário* no primeiro modelo de análise estudado na seção 4.3.2.

```
PACKAGE usuário IS
        TYPE dados do gerador
        PROC ler, mudar
        PRIVATE
                modulotensao IS NUMERIC
                angulotensao IS NUMERIC
                potênciaativa IS NUMERIC
                potenciareativa IS NUMERIC
                reatancia IS NUMERIC
        PACKAGE BODY usuário IS
                PRIVATE
                PROC ler (modulotensao, angulotensao, potenciaativa,
                        potenciareativa, reatancia)
                        { Detalhes de implementação do procedimento ler }
                .
                .
                .
                END
                PROC mudar (modulotensao, angulotensao, potenciaativa,
                        potenciareativa, reatancia)
                        { Detalhes de implementação do procedimento mudar }
                .
                .
                .
                END
END usuário
```

Utiliza-se nesta linguagem de projeto um vocabulário em inglês com uma sintaxe semelhante à linguagem Ada. Esta não implementa diretamente os conceitos de classe e objeto. A classe *usuário* é especificada como um "pacote" (instrução PACKAGE), a declaração de seus dados é feita com TYPE e a das operações com a instrução PROC. Em seguida é reservada a área privada (PRIVATE) onde é feito o ocultamente das estruturas de dados e do processamento.

Todos os módulos são representados desta maneira. A documentação para o modelo do projeto é finalizada com os digramas de módulo/relacionamento, como o da figura 4.3,

com uma especificação tabular de cada objeto e suas mensagens (tabelas 4.2, 4.3 e 4.4) e uma diagramação da estrutura global do software.

Tabela 4.4 – Comunicação Inter-objetos para a Interface

Conexões de Mensagens	Objeto Chamador	Objeto Recebedor	Método a Invocar	Parâmetros
Msg 12	Usuário	Comando 1	Click	Ativado
Msg 23	Comando 1	Tela 4	Exibir Dados Existentes	Atributos Obj 1
Msg 13	Usuário	Tela 4	Receber Alteração	Atributos Obj 1
Msg 14	Usuário	Comando 2	Click	Ativado
Msg 15	Usuário	Comando 3	Click	Ativado
Msg 56	Comando 3	Tela 6	Mostrar Saída	Conteúdo Cheio
⋮	⋮	⋮	⋮	⋮

A tabela 4.4 faz referência ao primeiro modelo de análise de software determinado para o exemplo do fluxo de carga. Juntamente com as anteriores ela completa a notação tabular simplificada para o projeto das interfaces.

Muitos métodos de projeto estruturado se utilizam de sub-divisões para construir um modelo final. Assim existe o projeto de dados, o projeto arquitetural e o projeto procedimental. Mais recentemente, vem sendo desenvolvido em muitas aplicações o projeto de interfaces, que além de estabelecer o layout do software possibilita a interação homem-máquina. Este, por se tratar de uma atividade cujo uso vem crescendo rapidamente entre os analistas e projetistas de sistemas, será discutido no capítulo 5. No projeto orientado a objetos não existe a necessidade dessa sub-divisão explicita pois a modularidade diz respeito às informações (dados) e ao processamento (operações) e não só ao processamento como nos outros métodos. Por isso, em cada módulo pode-se encontrar um projeto de dados quando representamos as estruturas dos dados do objeto e um projeto procedimental quando implementamos os detalhes funcionais das operações. O projeto

arquitetural é estabelecido quando representamos a estrutura global do software para especificar o relacionamento entre os módulos.

Com o conhecimento em análise e projeto adquiridos até então, podemos traçar um exemplo mais concreto de um software completamente orientado a objetos. Tomando como base a rotina desenvolvida em MATLAB® para cálculo do fluxo de carga em sistemas de transmissão de energia elétrica (ANEXO I), utilizando a estrutura da interface já descrita anteriormente, inclusive com seus objetos e a partir do primeiro modelo de análise de requisitos podemos definir todos os objetos para o programa.

Complementando a tabela 4.3 temos os objetos descritos na tabela 4.5 a seguir.

Tabela 4.5 – Objetos para o Núcleo do Projeto do Fluxo de Carga

Objeto	Nome	Atributos do Objeto	Métodos do Objeto	Tipo do Método	Conexões
Obj 7	Gravador	Ativado	Ler Dados Gravar Dados	Function Procedure	Msg 2-7
Obj 8	Printer	Ativado	Imprimir Dados	Function	Msg 1-8
Obj 9	Newton-Raphson	(Ver Tabela 4.1)	Executar	Procedure	Msg 4-9
Obj 10	Comando 4	Ponteiro Ativado	Click	Procedure	Msg 10-6 Msg 10-7
Obj 11	Comando 5	Ponteiro Ativado	Click	Procedure	Msg 1-11

O diagrama de objeto/relacionamento na figura 4.4 demonstra de forma simplificada os caminhos de mensagens entre os objetos existentes no projeto.

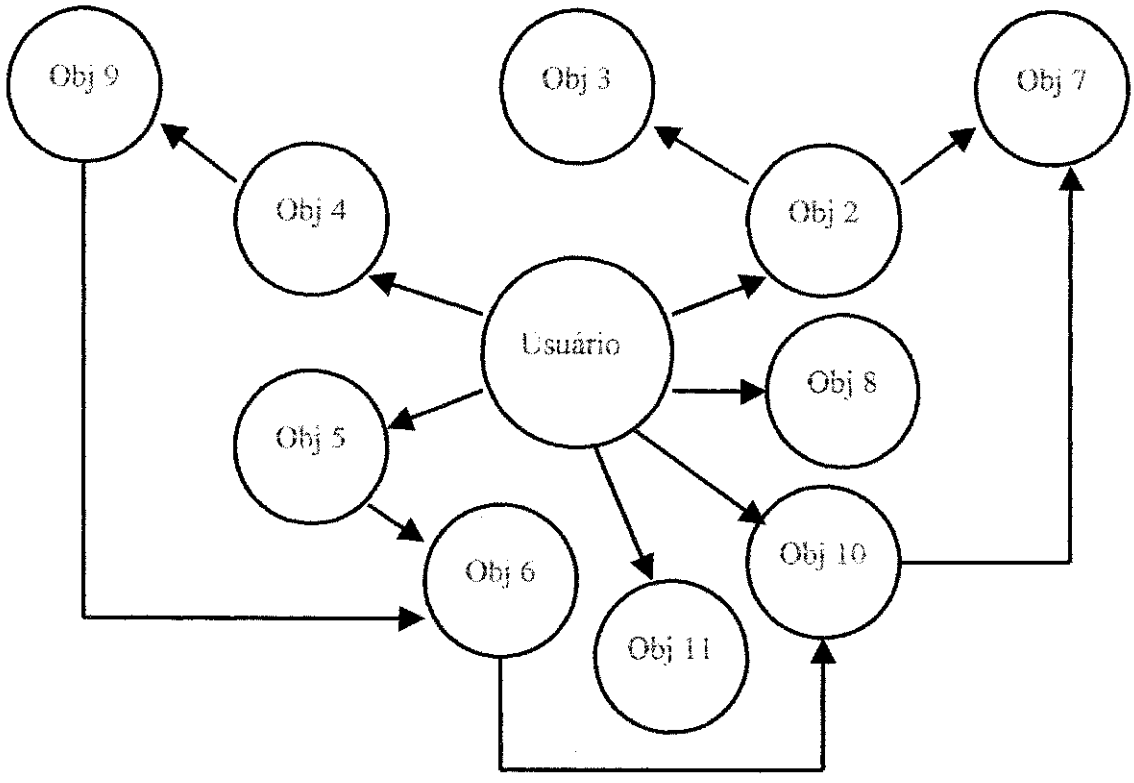

Figura 4.4 – Diagrama de objeto / relacionamento para o projeto completo

O usuário (objeto 1) através do evento "Abrir Arquivo" pressionando o botão de comando 1 (objeto 2), dispara duas mensagens deste objeto: "Ler Dados" direcionado para o gravador (objeto 7) e "Obter Dados Existentes" direcionado para a tela 4 (objeto 3). O usuário pode então a partir da tela 4, alterar os dados existentes ou introduzir novos dados, caso o arquivo não exista. Pressionando o botão de comando 2 (objeto 4), o usuário pede a execução dos cálculos a partir dos dados fornecidos. Este evento dispara mais uma mensagem para execução de uma função: "Executar", existente no objeto Newton-Raphson (objeto 9), com quem o objeto 4 se comunica. Pressionando o botão de comando 3 (objeto 5), o usuário pede a exibição dos dados de saída fornecidos pela tela 6 (objeto 6). Para gravá-los em um arquivo, o usuário deve pressionar o botão de comando 4 (objeto 10) que se comunica com a tela 6 para que esta informe os dados para o objeto gravador efetuar o procedimento de gravação. Para imprimir algum dado, o usuário deve ativar o

objeto "Printer" (objeto 8). Para sair do programa o usuário pressiona o botão de comando 5 (objeto 11) que irá executar o procedimento "exit".

Em qualquer projeto não se faz necessário esta narrativa visto que os caminhos de mensagens podem ser vastos, tornando-se inúmeras as possibilidades de interação do usuário com o software e de certa forma quase imprevisível a sua sequência.

Os gráficos juntamente com as tabelas fornecem os meios para a implementação dos módulos contendo as estruturas de dados e funcional. Utilizando uma PDL com uma sintaxe que implementa diretamente os conceitos de OOP podemos traçar um esboço simplificado de como poderia ser a arquitetura deste modelo de projeto. O ANEXO II contém todos os módulos definidos para cada classe de objetos identificados nesta etapa. Contudo, é inviável esboçar um detalhamento funcional e de dados neste trabalho. Assim não estão presentes todos as rotinas, tais como as de tratamento das telas da interface, de comunicação com dispositivos de entrada e saída, entre outras. Alguns dos métodos podem ser copiados das rotinas em MATLAB® no ANEXO I, acrescentando a eles as mensagens que farão a comunicação entre os objetos e traduzindo-os para OOP com a PDL utilizada no projeto.

Como pode ser visto no ANEXO II, o projeto baseado neste modelo de análise de requisitos é bastante simples. A classe "Newton-Raphson" implementa todas as rotinas antes em MATLAB, tornando-se um módulo de certa forma mono-funcional mas um pouco inchado (várias rotinas). Isto poderia ser eliminado com um projeto baseado no segundo modelo de análise de requisitos: considerando os objetos geradores, cargas, barras, linhas de transmissão e transformadores se comunicando entre si para a solução do método numérico de Newton-Raphson. Este também poderia ser quebrado em várias rotinas (procedimentos e funções) e inseridas dentro de módulos individuais como: "Matriz

Admitâncias de Barras", "Matriz Jacobiana" e dos módulos contendo os objetos do sistema de transmissão.

4.3.4 Codificação, Testes e Manutenção de Softwares

Codificar significa utilizar uma linguagem de programação para traduzir o projeto de forma que o computador possa entender.

A grande questão discutida atualmente pelos membros das áreas científica e de engenharia é: "Que linguagem alia os benefícios da programação orientada a objetos com a rapidez no processamento, confiabilidade e precisão nos resultados exigidos para os softwares destas áreas?"

Quando se define a área de atuação do programa ou sistema a ser desenvolvido e se determinam suas funções e características a partir do que se deseja em critérios de qualidade, limita-se ao mínimo as opções de escolha da linguagem a ser utilizada para implementar este software.

Com o desenvolvimento dos conceitos de orientação a objetos foram criadas dezenas de linguagens com características individuais que propiciam diferentes maneiras para implementar estes conceitos.

Linguagens realmente orientadas a objetos implementam diretamente as definições de classes, objetos, herança, encapsulamento e mensagens, além de possibilitar a criação das interfaces públicas e das áreas privadas para os objetos, apesar dos detalhes de implementação e os termos utilizados para definir estes conceitos variarem de uma para outra. Exemplos são: Smalltalk, Eiffel, versões atuais de Object Pascal (Delphi), C^{++} e Objective-C.

Dentre estas, a mais difundida sem dúvida é C^{++}, por herdar a alta potencialidade e ser de uso geral como a linguagem C. Atualmente acreditamos que seja a mais indicada para uso nas áreas científica e de engenharia.

A fase de testes para o programa é fundamental para a qualidade final do produto. Deve-se eliminar qualquer tipo de erro, seja ele lógico, funcional ou de implementação, que provoque falhas no processamento, instabilidade casual no sistema ou produza resultados errôneos, imprecisos ou não confiáveis.

A modularidade também é uma grande vantagem nesta fase. Os erros podem ser isolados por suas características e origem, assim os módulos que os contêm podem ser determinados e as causas do erro podem ser definidas mais rapidamente.

Atualmente, a maioria das linguagens possuem compiladores que proporcionam uma depuração passo a passo do erro, instrução por instrução.

Uma vez testado o software e seus erros encontrados, passa-se à fase de manutenção. Deve-se reaplicar todos os passos anteriores da engenharia de software para se obter um conjunto de mudanças com o objetivo de: corrigir os erros, adaptar o software às alterações em seu ambiente (sistema operacional, periféricos, etc.) ou mesmo para acrescentar funções exigidas pelo usuário final.

Novamente a modularidade é fundamental para se reduzir o tempo e o trabalho nesta fase. Mudar apenas um ou alguns módulos é bem mais rápido, o custo é menor e a confiabilidade do trabalho bem maior que se tivesse que modificar todo o programa sempre que necessário.

As fases de codificação, testes e manutenção possuem uma ampla literatura, não é objetivo deste trabalho definir detalhadamente cada uma, mesmo porque nestas fases as metodologias de trabalho para orientação a objetos são semelhantes à programação estruturada.

CAPÍTULO 5

Projetando uma Interface com o Usuário

5.1 Introdução

Atualmente, assim como o mundo está voltando mais atenção à analise e ao projeto de sistemas computacionais e não só à codificação e às linguagens de programação utilizadas para tanto, o projeto de interface ser humano-computador (Human Computer Interface – HCI) está também ganhando mais importância a medida que cresce o uso desta máquina. A interface é vista como a embalagem do software.

Com o surgimento da microeletrônica, começou a desenvolver-se entre algumas pessoas um certo pavor a alguns tipos de equipamentos eletrônicos computadorizados, inclusive ao próprio computador. A este último soma-se o fato das interfaces pouco amigáveis, difíceis de usar ou muitas vezes confusas, isto provoca quase sempre frustração e estresse ao se despender muito tempo tentando aprender a utilizar, sem sucesso, o software.

5.2 Conhecendo o Usuário

Um programa de computador que possua uma interface com usuário bastante amigável, fácil de aprender e simples de usar, possibilita satisfação e conforto e pode ser a porta de entrada para o sucesso do próprio software.

Para se criar uma interface efetiva com o usuário, SHNEIDERMAN (1987) declara: "Todo projeto deve iniciar-se com um entendimento dos usuários a que se destina,

inclusive perfis de sua idade, sexo, capacidades físicas, educação, background cultural ou étnico, motivação, metas e personalidade".

Para criar a interface de um determinado programa o projetista precisa conhecer os diversos fatores humanos relacionados aos usuários do seu produto, ou seja, deve ter em mãos alguns dados relativos a quem serão eles. Algumas perguntas podem ser levantadas de início:

1. Quem é o usuário? Apenas uma pessoa, um grupo de uma empresa, ou será toda uma classe específica.
2. Qual a finalidade do programa? A que se destina?
3. Quem fará o usuário aprender a utilizar o novo sistema baseado em computador?
4. O que o usuário espera do sistema?

O projetista deve ter conhecimento do que o sistema baseado em software irá executar para o usuário e as tarefas que serão exigidas deste para poder interagir com o programa.

Além disso, o projetista deve também levar em consideração o nível de habilidade individual bem como as variações de personalidade e distinções comportamentais de seus clientes.

Como exemplo pode-se tomar o caso de um médico e um mecânico de carros. O nível de habilidade entre um médico e o mecânico é inquestionavelmente diferente. O primeiro teve uma formação muito mais aprimorada, enquanto o segundo muitas vezes nem concluiu o segundo grau. O projeto de um software, tipo um editor de textos, deveria levar em consideração ambos os níveis, tentando conciliar habilidades já adquiridas com capacidade de raciocínio dedutivo e indutivo de ambas as classes de usuários.

Contudo, um conhecimento de domínio e contexto específico é mais importante, em certos casos, do que a educação ou inteligência global. No caso de um software específico

para o mecânico que lhe forneça um diagnóstico computadorizado do sistema de injeção eletrônica de combustível, com uma interface especificamente projetada para usuários com conhecimentos em mecânica, possibilitaria o domínio do problema e uma rápida interação por parte do mecânico, enquanto esta mesma interface poderia confundir um médico leigo neste assunto, apesar deste ter mais educação formal e capacidade de raciocínio que o mecânico.

Sabe-se que cada pessoa normal possui personalidade única que gera diferenças comportamentais entre umas e as outras. Uma interface homem-computador ideal deveria ser projetada para ser flexível às diferenças de personalidade ou acomodar uma personalidade típica entre uma classe de usuários finais. Esta última possibilidade pode ser bastante difícil de se executar pela falta de dados para o levantamento da personalidade específica de uma classe.

Logo, um software criado para um consultório médico para diagnóstico por ultra-som deveria considerar o projeto de uma interface que interagisse amigavelmente com cada personalidade médica que utilizasse o programa.

5.3 Estilos de Interfaces

Desde o início da computação o homem vem gradualmente dedicando mais tempo para o estudo de interfaces com usuários. A evolução do hardware possibilitou o surgimento de várias estilos de interfaces, criando diversas tendências de HCI.

O primeiro estilo de interação ser humano-computador, sem falar da era dos cartões, é a interface de comando e consulta que ainda existe até hoje. Uma tela escura onde a comunicação é puramente textual e a partir de comandos e respostas a consultas geradas pelo sistema.

Um exemplo deste tipo de interação pode ser visto na figura 5.1.

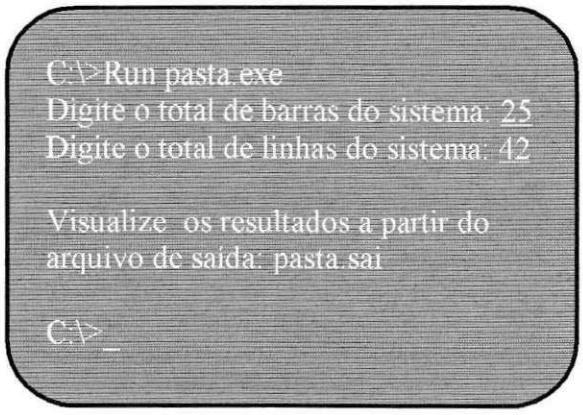

Figura 5.1 – Tela de interface de comando e consulta

A seguir, surgiu a interface de menu simples onde uma lista de opções é oferecida ao usuário que seleciona uma opção a partir de um código digitado.

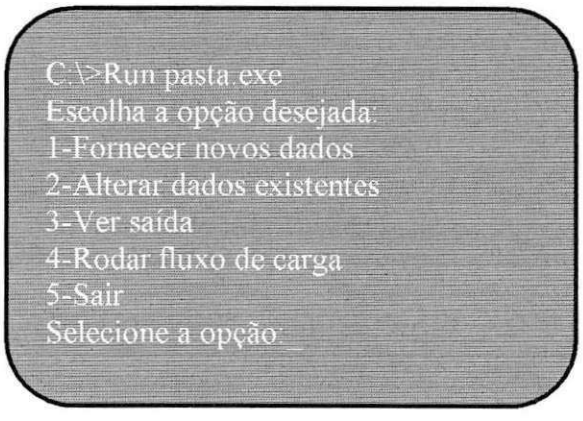

Figura 5.2 – Tela de interface de menu simples

Com o aprimoramento dos estudos sobre os fatores humanos e sua importância sobre o projeto de interfaces, começou a se desenvolver a terceira geração chamada: apontar e escolher (do inglês: "point and pick"), baseada em janelas, menus, ícones e dispositivos de indicação.

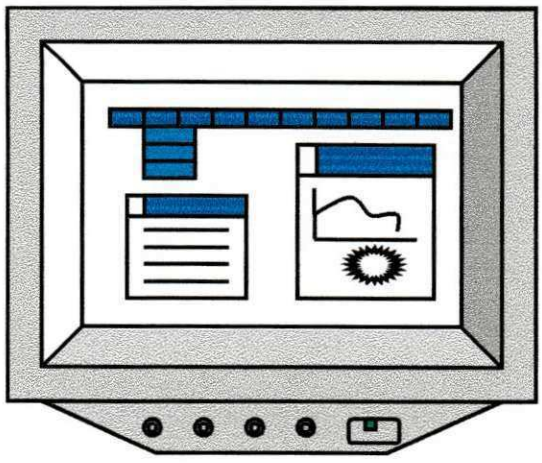

Figura 5.3 – Tela de interface "point and pick"

Segundo PRESSMAN (1995), podem ser observados três benefícios importantes fornecidos por esta geração de interfaces:

1. Diferentes tipos de informação podem ser exibidas simultaneamente, possibilitando que o usuário faça uma comutação de contextos (por exemplo, escrevendo código fonte numa janela, examinando os resultados de saída em outra, escrevendo uma atualização para uma narrativa de processamento numa terceira). As janelas permitem que o usuário execute muitas tarefas cognitivas[2] e de comunicação sem frustração.
2. Muitas tarefas interativas diferentes estão disponíveis por meio de um esquema de menu na tela do monitor, possibilitando ao usuário executar tarefas de diálogo e controle facilmente.
3. O uso de ícones gráficos, menus, botões e técnicas de listagem de valores ou opções reduzem a quantidade de digitação, isso pode aumentar a eficiência de interação dos que não são datilógrafos experientes e tornar o computador acessível a usuários que têm fobia de teclados.

---

[2] Relacionadas com as metas do software.

As interfaces de quarta geração combinam as vantagens da HCI de terceira geração com hipertexto[3] e multitarefa. São a base para os novos sistemas operacionais que estão surgindo bem como para os diversos softwares criados para estas plataformas, desde programas de controle de fluxo de caixa até complexos sistemas multimídia com acesso a redes internas de empresas (intranet) e a própria internet.

5.4 Aspectos de um Projeto de HCI

É evidente que um novo projeto de HCI é sempre diferente dos que já existem. Os estudos dos fatores humanos e a aplicação de diversas metodologias de interfaces são sempre refeitos quando de um novo projeto. Contudo, a indústria de software tem adotado padrões de interfaces que proporcionam a chamada "amigabilidade" entre o usuário e o programa.

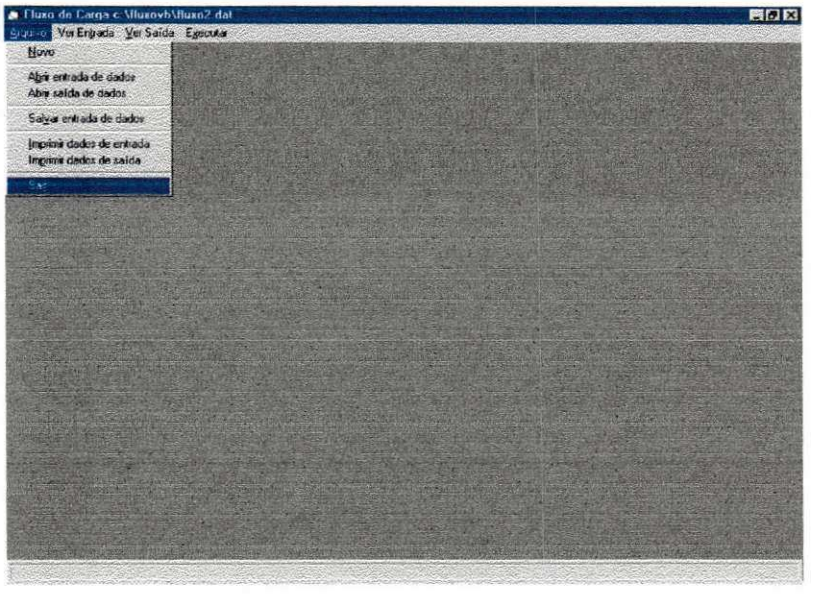

Figura 5.4 – Tela principal do software para análise de fluxo de carga

A padronização de comandos, janelas, formas de menus, tratamento de erros, sistemas de ajuda, entre outros, vem tornando o projeto e a criação de programas numa rotina comum que facilita ao projetista e fornece ao usuário um sistema que lhe é familiar,

[3] Textos que possuem palavras com ligações para se ter acesso a outros textos.

por aceitar por exemplo, comandos que são idênticos aos que ele já tenha utilizado em outros programas bem como por um estilo de interface comum adotada pela maioria dos desenvolvedores. A figura 5.4 exemplifica uma interface padrão também adotada para o projeto do software para fluxo de carga.

Isto não significa que um projeto sério de uma interface ser humano-computador deve ser relegado a uma simples cópia do que já existe (interface padrão). Novas perspectivas surgem a medida que os estudos dos fatores humanos são realizados e novas tecnologias de projeto de interfaces podem ser desenvolvidas a medida que cresce o uso da engenharia de software para a análise e o projeto de sistemas computadorizados.

Assim, alguns aspectos de um projeto de HCI podem ser relevantes quando da criação de uma interface. A frequência de uso (também conhecida como *usabilidade*) que terá um software é um deles. Por exemplo, um programa desenvolvido para uma empresa de eletrificação para estudos de estabilidade de sistemas de potência terá um uso menor que um software criado para monitoração e gerenciamento do tráfego aéreo em um aeroporto. Para o primeiro, a interface poderia ser algo não tão técnico e que pudesse ser compreendida por profissionais que não fossem da área de engenharia. Para o segundo, a interface teria uma aparência muito mais técnica, voltada exclusivamente para os usuários do sistema que detenham o conhecimento na área e que precisam de respostas rápidas para tomada de decisões.

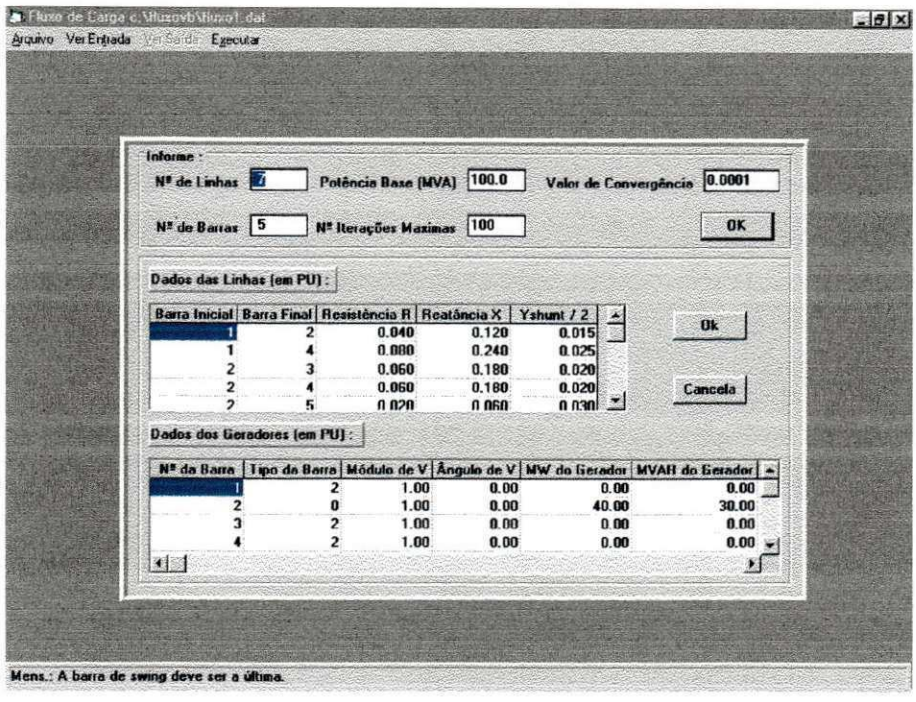

Figura 5.5 – Tela de entrada de dados para o fluxo de carga

Na figura 5.5, a tela de entrada de dados para o programa fluxo de carga mostra o que pode ser certo ou errado em sistemas projetados para engenharia, com respeito a este aspecto. Deve-se evitar que o usuário tenha de digitar as unidades de medida das grandezas, fornecendo-as na interface. Valores padrões para grandezas provavelmente conhecidas também devem ser fornecidos. Em tabelas, o ideal é que cada nova célula a ser preenchida contenha o valor da célula precedente para evitar digitação em casos de repetição de valores, muito comum em engenharia. Não se deve exigir do usuário a entrada de valores que o computador seja capaz de calcular, assim no lugar das impedâncias das linhas poderiam ter-se fornecidos o comprimento e a bitola do condutor. O computador determinaria as demais grandezas através de constantes ou valores armazenadas previamente em um banco de dados. Devem ser fornecidas as informações necessárias para que o usuário interaja com a interface eficientemente. Por exemplo, na figura 5.5 no campo *Tipo de Barra* não foram declaradas as opções existentes. Deve ser permitido omitir grandezas que não sejam relevantes a um determinado caso sem que mensagens de erro

sejam ativadas. Deve-se reduzir a quantidade de digitação necessária pelos usuários para se diminuir os erros provocados por estas ações, como exemplo: restringindo o uso à números em entradas exclusivamente numéricas, evitando a digitação de letras ou qualquer outro caracter.

O uso de mensagens de confirmação de tarefas críticas como apagar um arquivo de dados bem como a utilização, nos menus, dos chamados comandos "undo" para desfazer uma ação, são bastante importantes e as vezes vitais para o usuário.

Algumas outras questões podem ser levantadas durante o projeto de uma interface:

1. Qual a quantidade de dispositivos gráficos, janelas, menus e outros?
2. Qual a melhor forma de entrada de dados e os mecanismos do usuário para o controle do sistema?
3. Como será o fluxo das tarefas para cada janela da interface, ou seja a sequência das ações do usuário?
4. Como passar as informações de erro ao usuário?
5. Como será o acesso do usuário ao sistema de ajuda?
6. Definir e padronizar as formas dos comandos utilizados pelos usuários.
7. Como definir o tempo de resposta do software após um determinado comando do usuário?

As mensagens de erros devem fornecer informações úteis numa linguagem que os usuários possam entender, indicando que procedimento saiu errado, o erro provocado e como sanar o problema gerado. Tudo isso, a partir de sinais visuais (cores destacadas, ou piscamento) e sonoros (bips) que as destaquem do resto do programa. Em todos os casos, mesmo que o erro não possa ser corrigido, devolver se possível, o controle ao programa sem interrupção brusca com a saída do sistema. Uma mensagem de erro típica é mostrada

na figura 5.6. Ela foi gerada a partir de um evento falho de leitura de uma unidade de disco para gravação de dados do programa para fluxo de carga.

Figura 5.6 – Uma mensagem de erro típica

Observe que não foi informado que procedimento falhou e a identificação do tipo de erro gerado não é específica. "Erro 68" pode não indicar nada para o usuário leigo.

Deve-se ter conhecimento de quantas e quais as funções ou tarefas a serem desenvolvidas pelo usuário necessitarão da ajuda do sistema. Seria ideal levar-se em consideração que todos estes fossem principiantes, assim todo o sistema estaria coberto por uma malha de ajuda, fundamental para o sucesso da interação homem-máquina. O sistema de ajuda *integrado*[4] pode ser utilizado de forma abrangente tanto como "caça-palavras", utilizando-se palavras-chaves, bem como um hipertexto. O sistema de ajuda "add-on" (incluído no programa após a criação do modelo) pode ser utilizado para fornecer uma consulta "on-line" limitada.

Existem várias formas de comandos: digitados, menus suspensos, sequências do teclado, teclas de funções, etc. É necessário se conhecer o perfil do usuário (dificuldade de memorizar, impossibilidade de utilizar o mouse, etc.) antes de utilizar um ou outro estilo de comando para desempenhar qualquer tarefa. Contudo, deve-se seguir um padrão, um comando para gravar em uma aplicação deve ser o mesmo nas demais.

---

[4] Projetado com o programa desde o início.

É preciso se considerar a complexidade da resposta e o tipo de usuário (principiante ou instruído) para que o tempo de resposta após o disparo de um evento não seja tão longo que canse ou tão curto que provoque erros do usuário. Em sistemas utilizados continuamente durante um período (todo um dia, por exemplo), como é o caso de programas para atendimento de reclamações de consumidores via telefone. A atendente passa todo o dia interagindo com os mesmos comandos, assim seria ideal um tempo de resposta invariável que possibilitasse um ritmo de acesso ao fluxo de controle. A quantidade de erros seria menor e o tempo gasto com cada consumidor também. Para programas em engenharia, o tempo de resposta para um evento disparado pelo usuário pode estar diretamente ligado ao tempo no processamento de alguma rotina de cálculo e este sendo baixo indica uma boa qualidade para o software.

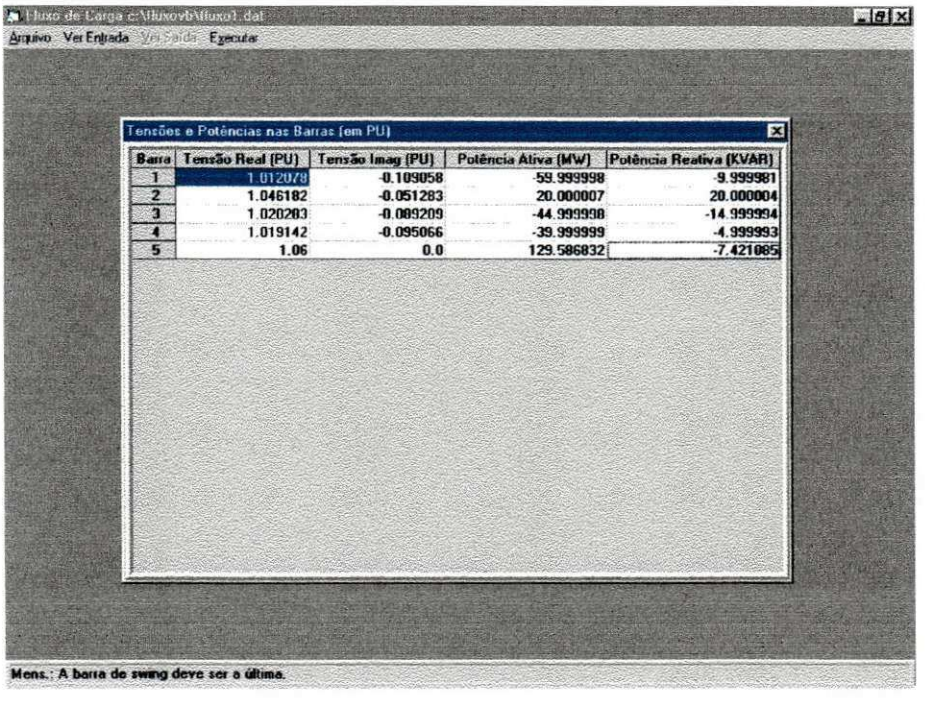

Figura 5.7 – Exemplo de uma tela de saída contendo uma planilha de dados

A maioria dos softwares criados para a área de engenharia são mais pesados e mais complexos e por isso mais sujeitos a instabilidades que os de outras áreas, principalmente

quando nestes são criadas interfaces que possibilitem ao usuário alterar o sistema para solução de um determinado problema. Deve-se portanto na fase de testes, verificar o nível de comprometimento do software com a estabilidade do sistema como já foi mencionado, para limitar o número de casos que provoquem uma parada anormal.

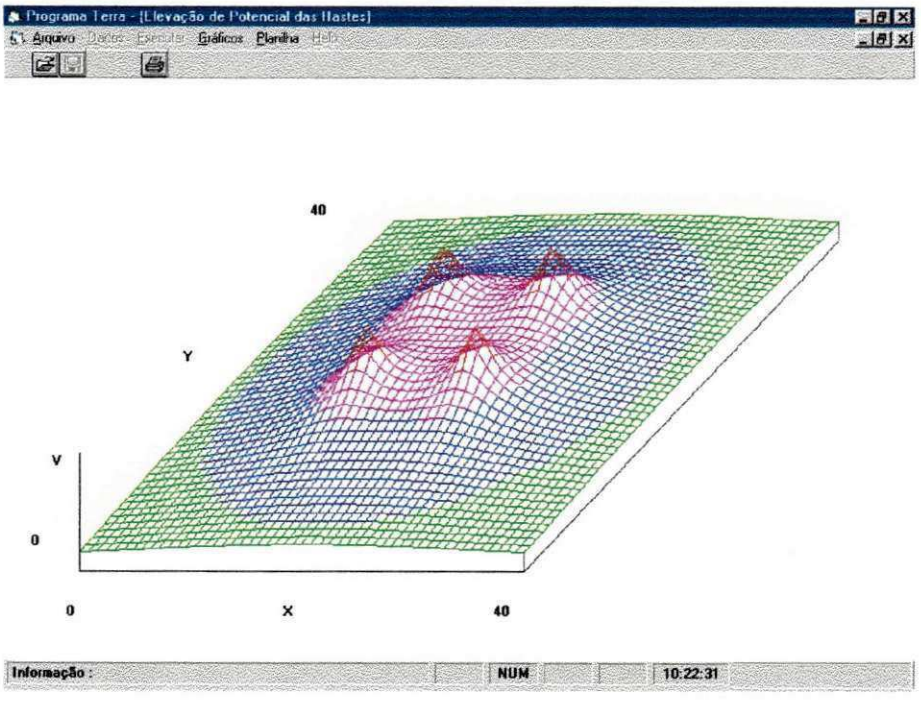

Figura 5.8 – Exemplo de uma tela de saída contendo um gráfico

Além das interfaces padrões de: abertura e fechamento de arquivos, solicitação de ajuda, edição, entre outras, das interfaces específicas: para entrada de dados, controle e definição das características da execução, em programas para engenharia também existem as interfaces de saída e análise dos resultados. Estas telas geralmente fornecem os dados em planilhas apenas para leitura ou em gráficos, como nas figuras 5.7 e 5.8. O uso de um estilo ou de outro baseia-se no que cada um pode fornecer. A planilha na figura 5.7 indica o comportamento de um sistema de transmissão em cada uma de suas barras individualmente. O gráfico da interface na figura 5.8, criada para um programa para análise

de sistemas de aterramento, fornece o comportamento de cada haste numa visão do sistema como um todo.

Em todos os casos, as interfaces de entrada devem satisfazer os requisitos do software em fornecer a solução matemática a partir dos dados que o usuário pode fornecer. As interfaces de saída devem suprir a necessidade do usuário para solucionar o problema físico real. Contudo, ambas também estão sujeitas ao modelo do projeto. Por exemplo: caso o modelo da análise do software fosse baseado nas cinco classes de objetos existentes (barras, linhas, geradores, cargas e transformadores), a interface de entrada de dados na figura 5.5 seria completamente diferente. O usuário poderia definir a classe e as variáveis para cada objeto.

5.5 Uma Metodologia de Projeto de Interface

Raramente um software interativo criado para se executar alguma atividade em engenharia possibilita o surgimento de alguma nova tarefa pelo usuário. Esses sistemas são elaborados, na maioria das vezes, para automatizar processos que antes eram realizados manualmente. Com isso, melhora-se o resultado, tornando-o mais confiável e preciso, e diminui-se o tempo na sua execução.

Um exemplo é a automação dos estudos de fluxo de carga por um sistema computadorizado. A coleta dos dados pode continuar da mesma forma. Entretanto, não é necessário realizar manualmente todo o processo matemático de análise. Através da mesma metodologia de estudo transplantada para o computador, os resultados do fluxo são fornecidos com maior rapidez e precisão nos cálculos.

Neste caso, a tarefa principal da interface com o usuário é de fornecer um ambiente de interação homem-computador da forma mais agradável para que o operador do sistema sinta o mínimo de dificuldade possível no tratamento com o software, desde a entrada dos dados até a coleta e interpretação dos resultados.

Pode-se utilizar a seguinte abordagem para se projetar uma HCI:

1. *Levantar as tarefas dos seres humanos e do computador necessárias para que o sistema funcione corretamente*. Pode-se utilizar uma técnica conhecida como "Análise e Modelagem de Tarefas". Como o nome sugere, esta técnica consiste no levantamento e definição das tarefas que serão realizadas pelos seres humanos e pelo computador, para uma HCI. Para isso, a análise e modelagem de tarefas pode ser aplicada de uma maneira direta: o projetista deve conhecer e entender as tarefas executadas manualmente pelos seres humanos, depois transplantá-las, implementado-as semelhantemente no software. Ou de maneira reversa: a partir de uma especificação existente para uma solução baseada em computador, o projetista pode levantar um conjunto de tarefas de usuário que acomodem o modelo de usuário, o modelo de projeto e a percepção do sistema (PRESSMAN, 1995). Como exemplo, um projeto de um simulador para pilotos de aeronaves. No âmbito da interface com o usuário, seria muito mais racional e eficiente um modelo que leva em consideração, como especificação para uma solução, o painel de um avião e o implementa-se na tela do computador do simulador. Logo, o efeito que o sistema traria para o usuário seria de satisfação, devido a interface conhecida e com um conjunto de tarefas que possibilitasse o provável conhecimento de domínio do problema.
2. *Traçar todas as questões necessárias a um bom projeto de interface*. Além dos fatores humanos, seção 5.2, questões sobre a melhor estrutura da HCI para solução de determinado problema, seção 5.4.
3. *Implementar o modelo do sistema*. Existem quatro tipos para uma HCI:
    a. Modelo de projeto: é a visão do projetista de como a interface deverá ser para acomodar o projeto funcional, arquitetural e de dados contido em todo sistema. Neste caso, não se relacionam os fatores humanos.

b. Modelo de usuário: o projetista, a partir do conhecimento dos fatores humanos traça o perfil dos usuários finais do seu sistema. O modelo de usuário deve refletir este perfil.

c. Percepção do sistema: é a imagem de como deveria ser o sistema, traçada pelo usuário final. Todos nós temos níveis diferentes de conhecimento, baseado nisto, cada um elabora sua própria perspectiva de como deveria ser um sistema para executar determinada tarefa ou grupo de tarefas.

d. Imagem do sistema: é a forma real do sistema que será determinada pelo projetista, baseada nos estudos da percepção pelos usuários.

A interligação destes quatro modelos existe e é necessária para que o projeto de uma HCI tenha resultados consistentes, então as diferenças entre eles devem ser eliminadas para que surja uma única representação ou modelo final.

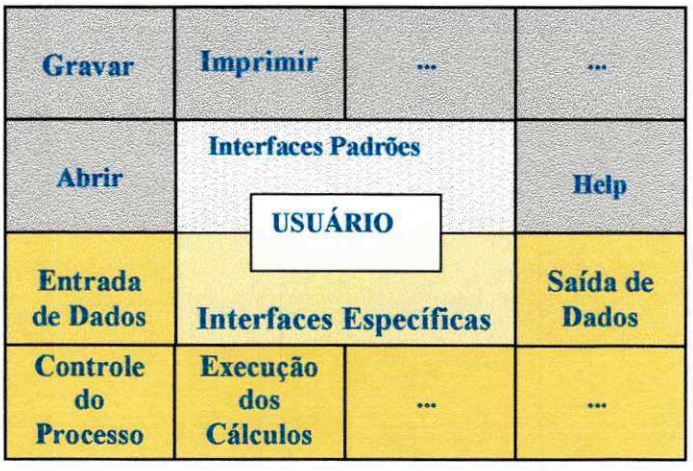

Figura 5.9 – Diagrama para o modelo de projeto de interface com o usuário

4. *Utilizar ferramentas para construção de protótipos.* Linguagens de programação estruturada ou orientadas a objeto. Para estas surgiram pacotes de programas que geram o código fonte para algumas atividades, como criação de janelas, menus, mensagens de erro e outros.

5. *Avaliar o protótipo quanto às suas qualidades.* Para isto, pode-se consultar os próprios usuários. Pode ser uma conversa informal, levantando os possíveis problemas encontrados por estes, ou a partir de avaliações formais utilizando-se questionários. Em todos os casos, são feitas as modificações no projeto e um novo protótipo é criado, este é reavaliado, se algum problema persistir, novas modificações serão feitas até se chegar a um protótipo final.

Novamente, a documentação para este projeto deve conter: uma notação gráfica representado a interação do usuário com o software em termos de interfaces (figura 5.9), um esboço para cada tela de comunicação e uma notação tabular que forneça todos os componentes e eventos necessários ao sucesso desta comunicação, como as tabelas 5.1 e 5.2.

Tabela 5.1 – Especificação das Interfaces

Tela	Módulo/Objeto	Especificação	Conexões
Principal	Gerente	Tela Principal	Todas as Telas
Abrir	Gravador	Tela 1	Telas Principal / 4
Gravar	Gravador	Tela 2	Tela Principal
Imprimir	Impressora	Tela 3	Tela Principal
Entrar Dados	Comando 1	Tela 4	Tela Principal
Executar	Comando 2	Tela 5	Tela Principal
Dados de Saída	Comando 3	Tela 6	Tela Principal
⋮	⋮	⋮	⋮

Tabela 5.2 – Tela Entrar Dados

Componentes	Parâmetros Exigidos	Conexões
Caixa de Texto 1	N.° de Linhas	Botão OK 1
Caixa de Texto 2	N.° de Barras	Botão OK 1
Caixa de Texto 3	Potência Base (MVA)	Botão OK 1
Caixa de Texto 4	N.° Iterações Máximas	Botão OK 1
Caixa de Texto 5	Tolerância	Botão OK 1
Tabela 1	Barra Inicial Barra Final Resistência (pu) Reatância (pu) Admitância Shunt (pu)	Botão OK 2
Tabela 2	N.° da Barra Tipo de Barra Módulo da Tensão (pu) Ângulo da Tensão (rad) Potência Ativa (MW) Potência Reativa (MVAr)	Botão OK 2
Botão OK 1	Evento de pressionamento	Botão OK 2
Botão OK 2	Evento de pressionamento	Tela Principal
Botão Cancela	Evento de pressionamento	Tela Principal

As duas tabelas anteriores fazem parte do projeto para o fluxo de carga, sendo a tabela 5.2 baseada na figura 5.5.

Um módulo pode conter quantas janelas ele precisar para implementar seus métodos. A programação para toda essa comunicação é feita durante a codificação. Entretanto, na OOP o projeto das interfaces é parte integrante e deve estar presente na representação de cada módulo.

Existem vários outros aspectos importantes em um projeto de interface ser humano-computador que devem ser levados em consideração em cada caso. Contudo, é necessário um aprofundamento maior na tecnologia de interfaces, que não é o objetivo deste trabalho. Foram levantados aqui, apenas alguns tópicos necessários para que o engenheiro desperte para a necessidade do estudo dos diferentes fatores envolvidos no projeto de uma HCI.

CAPÍTULO 6

Conclusões e Propostas para Trabalhos Futuros

A metodologia para desenvolvimento de rotinas computacionais apresentada neste trabalho aplica-se não só à sistemas de potência mais a qualquer área científica/de engenharia. Essa versatilidade deve-se a abordagem utilizada que enfatiza a forma de tratamento de estudos nestas áreas. Seja qual for o problema, a diretriz para análise será sempre a mesma.

Os estudos para definição do problema e implementação do método matemático para tratamento computacional devem ser feitos considerando as condições do sistema abordado e as necessidades para a solução.

Para análise e projeto orientados a objetos, a familiarização com os conceitos definidos por esta tecnologia é imprescindível para o desenvolvimento de um sistema computacional de qualidade. Além disso, cada elemento/objeto do sistema a ser tratado no problema, quando representado em módulos individuais eleva ainda mais o nível de qualidade do produto final.

O tratamento externo com o usuário do software feito a partir de um projeto de interfaces adequado possibilita atingir outros critérios de qualidade como: facilidade de uso e de análise dos resultados.

Por fim, a codificação, os testes e a manutenção do software são dependentes da linguagem de programação escolhida, que deve traduzir todas as características do projeto.

Com este trabalho, o engenheiro de potência adquire uma ferramenta de auxílio no tratamento computacional de antigos problemas em sua área. Este também possibilita o direcionamento deste profissional e/ou pesquisador no que hoje é considerado como a tendência mundial no desenvolvimento de softwares, seja em centros de pesquisa ou em empresas de engenharia e sistemas.

Com esta ferramenta surgem algumas propostas para novos trabalhos. Entre elas podemos citar:

- Desenvolvimento de uma biblioteca de objetos/módulos, que possibilite a reutilização em diversos estudos no tratamento de problemas em sistemas de transmissão e distribuição de energia elétrica.
- Desenvolver um estudo mais completo considerando todos os elementos do sistema, para análise de fluxo de carga em sistemas de distribuição utilizando o método Desacoplado Rápido. Verificando alguns critérios de qualidade como o tempo total na execução dos cálculos, entre outros, quando implementado utilizando a orientação a objetos.
- A partir do segundo modelo de análise proposto para o software de fluxo de carga, refazer o projeto para o programa com uma abordagem completamente orientada a objetos, introduzindo estudos de estabilidade, análise de faltas e outros, necessários ao planejamento e a operação dos sistemas elétricos.

REFERÊNCIAS BIBLIOGRÁFICAS

CAMPIONE, Mary, WALRATH, Kathy. *The Java™ Tutorial Object-Oriented Programming for the Internet*. Addison-Wesley Publisshing Company. USA, 1996.

CANTÚ, Marco. *Dominando o Delphi*. Editora Makron Books. São Paulo, 1996.

CHAPMAN, Stephen. *FORTRAN 90/95 For Scientists and Engineers*. McGraw-Hill. USA, 1997.

COAD, Peter, YOURDON, Edward. *Análise Baseada em Objetos*. Editora Campus, Série Yourdon Press. Rio de Janeiro, 1992.

COAD, Peter, YOURDON, Edward. *Projeto Baseado em Objetos*. Editora Campus, Série Yourdon Press. Rio de Janeiro, 1993.

COLEMAN, Derek. *Desenvolvimento Orientado a Objetos*. Editora Campus. Rio de Janeiro, 1996.

CONNER, D. Brookshire et al. *Object Oriented Programming in Pascal*. Addison Wesley Publishing. USA, 1995.

FAISON, Ted. *Object Oriented Programming Boorland C++ 4.5*. Sams. USA, 1995.

GENERINI, Adelize. *Análise, Projeto e Programação Orientados a Objetos*. Bookstore. USA, 1996.

HORNBECK, Robert W. *Numerical Methods*. Quantum Publishers Inc. New York, 1975.

MATHWORKS, Inc. *MATLAB® Versão 4.0 para Estudantes, Manual do Usuário*. USA, 1995.

NELSON, Ross. *Visual Basic for Windows*. Versão 3.0, Guia Autorizado Microsoft. Editora Makron Books. São Paulo, 1994.

PRESSMAN, Roger S. *Engenharia de Software*. Editora Makron Books. São Paulo, 1995.

RUMBAUGHT, James et al. *Modelagem e Projetos Baseados em Objetos*. Editora Campus. Rio de Janeiro, 1995.

SHNEIDERMAN, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publisshing Company. USA, 1987.

SOARES, Alberto. *Planejamento de Sistemas de Distribuição: Um Modelo para Cálculo de Fluxo de Potência em Sistemas de Distribuição. Comitê Coordenador de Operações Norte/Nordeste, II CPDE.* Recife, outubro de 1993.

STAGG, Glenn W., EL-ABIAD, Ahmed H. *Computer Methods in Power Systems Analysis*. McGrall-Hill. USA, 1968.

STEVENSON, William D. *Elementos de Análise de Sistemas de Potência*. 2ª Edição Português. Editora McGraw-hill Ltda. São Paulo, 1986.

THOMAS, Zane, ARNSON, Robert, WAITE, Mitchell. *Visual Basic How-to*. Editora Axcel Books, 2º Edição. Rio de Janeiro, 1995.

WEEDY, B.M. *Eletrical Power Systems*. 3ª Edition. John Wiley & Sons Ltd. London, 1979.

ANEXO I

```
% PROGRAMA EM MATLAB PARA CÁLCULO DO FLUXO DE CARGA EM SISTEMAS DE
% TRANSMISSÃO
% ------------------------------------------------------------------------
% ROTINA 1: ABERTURA DOS ARQUIVOS
%
 clear;
 fid=fopen('c:\fluxovb\flu1.txt');
 file=fscanf(fid,'%s');
 fclose(fid);
 fid=fopen('c:\fluxovb\flu2.txt');
 arq=fscanf(fid,'%s');
 fclose(fid);
%
 fid=fopen(file,'rt');
 var=fscanf(fid,'%f',8);
 fclose(fid);
% ------------------------------------------------------------------------
% ROTINA 2: LEITURA DOS DADOS GERAIS
%
 sb = var(1,1);
 nl = var(2,1);
 nb = var(3,1);
 itmax = var(4,1);
 toler = var(5,1);
 delp=zeros(nb-1,1);
 delq=zeros(nb-1,1);
%
 cont=nl+nb+1;
 fid=fopen(file,'rt');
 car=fscanf(fid,'%f',[8,cont]);
 fclose(fid);
 for i=1:8,
      for j=1:cont,
           var(j,i)=car(i,j);
      end;
 end;
% ------------------------------------------------------------------------
% ROTINA 3: LEITURA DOS DADOS DAS LINHAS
%
 for l=1:nl,
      le(l)=var(l+1,1);
      ld(l)=var(l+1,2);
      r(le(l),ld(l))=var(l+1,3);
      x(le(l),ld(l))=var(l+1,4);
      yshu(le(l),ld(l))=var(l+1,5)*sqrt(-1);
      zser(le(l),ld(l))=r(le(l),ld(l))+x(le(l),ld(l))*sqrt(-1);
%------------------------------------------------------------------------
% ROTINA 4: FORMAÇÃO DA MATRIZ YBUS
%
      yser(le(l),ld(l))=1.0/zser(le(l),ld(l));
      y(le(l),ld(l))=-yser(le(l),ld(l));
      y(ld(l),le(l))=y(le(l),ld(l));
      y(le(l),le(l))=y(le(l),le(l))+yser(le(l),ld(l))+yshu(le(l),ld(l));
      y(ld(l),ld(l))=y(ld(l),ld(l))+yser(le(l),ld(l))+yshu(le(l),ld(l));
 end;
```

```
 for i=1:nb,
      for j=1:nb,
            G(i,j)=real(y(i,j));
            B(i,j)=-imag(y(i,j));
      end;
 end;
% ------------------------------------------------------------------
% ROTINA 5: LEITURA DOS DADOS DAS BARRAS
%
 for l=1:nb,
      b=var(l+1+nl,1);
      tb(b)=var(l+1+nl,2);
      modv(b)=var(l+1+nl,3);
      angv(b)=var(l+1+nl,4);
      pg(b)=var(l+1+nl,5)/sb;
      qg(b)=var(l+1+nl,6)/sb;
      pc(b)=var(l+1+nl,7)/sb;
      qc(b)=var(l+1+nl,8)/sb;
      pl(b)=pg(b)-pc(b);
      ql(b)=qg(b)-qc(b);
      ang(b)=angv(b)/57.295780;
      E(b)=modv(b)*cos(ang(b))+modv(b)*sin(ang(b))*sqrt(-1);
 end;
 for i=1:nb,
      e(i)=real(E(i));
      f(i)=imag(E(i));
 end;
% ------------------------------------------------------------------
% ROTINA 6: CÁLCULO DAS POTÊNCIAS ATIVAS E REATIVAS
%
 a=1.;
 k=0.;
 while a > toler,
      for i=1:nb-1,
            p(i)=0.;
            q(i)=0.;
            for j=1:nb,
                  fator1(i)=  e(i)*(e(j)*G(i,j)+f(j)*B(i,j))+
                              f(i)*(f(j)*G(i,j)-e(j)*B(i,j));
                  p(i)=p(i)+fator1(i);
                  fator2(i)=  f(i)*(e(j)*G(i,j)+f(j)*B(i,j))-
                              e(i)*(f(j)*G(i,j)-e(j)*B(i,j));
                  q(i)=q(i)+fator2(i);
            end;
      end;
      for i=1:nb-1,
            delp(i)=pl(i)-p(i);
            delq(i)=ql(i)-q(i);
% ------------------------------------------------------------------
% ROTINA 7: CÁLCULO DAS CORRENTES NAS BARRAS
%
            ib(i)=(p(i)-q(i)*sqrt(-1))/(conj(E(i)));
            c(i)=real(ib(i));
            d(i)=imag(ib(i));
      end;
% ------------------------------------------------------------------
% ROTINA 8: CÁLCULO DOS ELEMENTOS DA MATRIZ JACOBIANA
%
```

```
      for i=1:nb-1,
            for j=1:nb-1,
                  if i==j
                        j1(i,j)=e(i)*G(i,j)-f(i)*B(i,j)+c(i);
                        j2(i,j)=e(i)*B(i,j)+f(i)*G(i,j)+d(i);
                        j3(i,j)=e(i)*B(i,j)+f(i)*G(i,j)-d(i);
                        j4(i,j)=-e(i)*G(i,j)+f(i)*B(i,j)+c(i);
                  else
                        j1(i,j)=e(i)*G(i,j)-f(i)*B(i,j);
                        j2(i,j)=e(i)*B(i,j)+f(i)*G(i,j);
                        j3(i,j)=f(i)*G(i,j)+e(i)*B(i,j);
                        j4(i,j)=f(i)*B(i,j)-e(i)*G(i,j);
                  end;
            end;
      end;
% ------------------------------------------------------------------------
% ROTINA 9: FORMAÇÃO E INVERSÃO DA MATRIZ JACOBIANA E DA MATRIZ P
%
      for i=1:nb-1,
            for j=1:nb-1,
                  mj(i,j)=j1(i,j);
            end;
            for j=nb:(2*nb-2),
                  mj(i,j)=j2(i,j-(nb-1));
            end;
      end;
      for i=nb:(2*nb-2),
            for j=1:nb-1,
                  mj(i,j)=j3(i-(nb-1),j);
            end;
            for j=nb:(2*nb-2),
                  mj(i,j)=j4(i-(nb-1),j-(nb-1));
            end;
      end;
      MJ=inv(mj);
      P=[ delp
          delq ];
% ------------------------------------------------------------------------
% ROTINA 10: CÁLCULO DAS NOVAS TENSÕES NAS BARRAS
%
      deltensao=MJ*P;
      dele=deltensao(1:(nb-1),:);
      delf=deltensao( nb:(2*nb-2),:);
      for i=1:nb-1,
            e(i)=e(i)+dele(i);
            f(i)=f(i)+delf(i);
            E(i)=e(i)+f(i)*sqrt(-1);
      end;
      a=max(abs(P));
      k=k+1;
 end;
% ------------------------------------------------------------------------
% ROTINA 11: CÁLCULO DO FLUXO DE POTÊNCIA NAS LINHAS
%
```

```
for l=1:nl,
     i=le(l);
     j=ld(l);
     I(i,j)=(E(i)-E(j))*(-y(i,j))+E(i)*yshu(i,j);
     FP(i,j)=conj(E(i))*I(i,j);
     I(j,i)=(E(j)-E(i))*(-y(i,j))+E(j)*yshu(i,j);
     FP(j,i)=conj(E(j))*I(j,i);
     PMW(i,j)=real(FP(i,j))*sb;
     QMVAR(i,j)=-imag(FP(i,j))*sb;
     PMW(j,i)=real(FP(j,i))*sb;
     QMVAR(j,i)=-imag(FP(j,i))*sb;
     X1(:,l)=[ i
               j
               PMW(i,j)
               QMVAR(i,j) ];
     X2(:,l)=[ j
               i
               PMW(j,i)
               QMVAR(j,i) ];
     X=[ X1      X2];
end;
% ------------------------------------------------------------------------
% ROTINA 12: CÁLCULO DA POTÊNCIA NA BARRA DE BALANÇO
%
for i=nb,
     p(i)=0.;
     q(i)=0.;
     for j=1:nb,
          fator1(i)=  e(i)*(e(j)*G(i,j)+f(j)*B(i,j))+
                      f(i)*(f(j)*G(i,j)-e(j)*B(i,j));
          p(i)=p(i)+fator1(i);
          fator2(i)=  f(i)*(e(j)*G(i,j)+f(j)*B(i,j))-
                      e(i)*(f(j)*G(i,j)-e(j)*B(i,j));
          q(i)=q(i)+fator2(i);
     end;
end;
for i=1:nb,
     p(i)=p(i)*sb;
     q(i)=q(i)*sb;
     Z(:,i)=[ e(i)
              f(i)
              p(i)
              q(i) ];
end;
% ------------------------------------------------------------------------
% ROTINA 13: SAÍDA DOS RESULTADOS
%
  z=[ X Z];
%
fid=fopen(arq,'wt');
fprintf(fid,'%14.6f%14.6f%14.6f%14.6f\n',z);
fclose(fid);
%
exit;
```

ANEXO II

```
% PROGRAMA PARA CÁLCULO DO FLUXO DE CARGA EM SISTEMAS DE TRANSMISSÃO
% COM ORIENTAÇÃO A OBJETOS
%
% ------------------------------------------------------------------
% Definição dos Objetos
%
Comando1 IS OBJECT OF CLASS Comandos
      IF Comando1_Ativado=TRUE THEN
            Comando1.Click(1)
      END IF
Comando2 IS OBJECT OF CLASS Comandos
      IF Comando2_Ativado=TRUE THEN
            Comando2.Click(2)
      END IF
Comando3 IS OBJECT OF CLASS Comandos
      IF Comando3_Ativado=TRUE THEN
            Comando3.Click(3)
      END IF
Comando4 IS OBJECT OF CLASS Comandos
      IF Comando4_Ativado=TRUE THEN
            Comando4.Click(4)
      END IF
Comando5 IS OBJECT OF CLASS Comandos
      IF Comando5_Ativado=TRUE THEN
            Comando5.Click(5)
      END IF
Comando6 IS OBJECT OF CLASS Comandos
      IF Comando6_Ativado=TRUE THEN
            Comando6.Click(6)
      END IF
Tela4 IS OBJECT OF CLASS Telas
Tela6 IS OBJECT OF CLASS Telas
Gravador1 IS OBJECT OF CLASS Gravador
Printer1 IS OBJECT OF CLASS Printer
NewtonRaphson1 IS OBJECT OF CLASS NewtonRaphson
% ------------------------------------------------------------------
% Definição dos Módulos
%
MODULE 1

CLASS Comandos
      PUBLIC:
            VAR Ativado IS BOOLEAN
            VAR Entrada(,), Saida(,) IS TYPE MATRIX(,)
            PROC Click(Ponteiro IS INTEGER)
      PRIVATE BODY
            PROCEDURE Click(Ponteiro)
                  IF Ponteiro = 1 THEN
%
% Mensagem para o objeto Gravador1 ler os dados de entrada e gravar na
% variável Entrada(,), com estes dados fornecidos ao objeto Tela4,
% invoca-se seu método de exibição.
%
                        Entrada(,)=Gravador1.LerDados
                        Tela4.ExibirDadosExistentes(Entrada(,))
```

```
                ELSE IF Ponteiro = 2 THEN
%
% Mensagem para o objeto NewtonRaphson1 executar os cálculos com os dados
% de entrada fornecidos. O resultado é armazenado na variável saída(,).
%
                      Saida(,)=NewtonRaphson1.Executar(Entrada(,))
                ELSE IF Ponteiro = 3 THEN
%
% Mensagem para o objeto Tela6 mostrar no monitor o resultados dos
% cálculos.
%
                      Tela6.MostrarDadosSaida(Saída(,))
                ELSE IF Ponteiro = 4 THEN
%
% Mensagem para o objeto Gravador1 gravar os dados de entrada e de saída.
%
                      Gravador1.GravarDados(Entrada(,),Saida(,))
                ELSE IF Ponteiro = 5 THEN
%
% Mensagem para o objeto Printer1 imprimir os dados de entrada e de
% saída.
%
                      Printer1.ImprimirDados(Entrada(,),Saída(,))
                ELSE IF Ponteiro = 6 THEN
                      CLOSE
                      EXIT PROGRAM
                END IF
          END PROCEDURE
END CLASS;

MODULE 2

CLASS Telas
      PUBLIC:
            VAR NL, NB, BI(), BF(), Nbarra(), TipoBarra(),
                IteMax IS INTEGER
            VAR R(), X(), Y(), ModV(), AngV(), PotAtiv(), PotReativa(),
                PotBase, Conver IS REAL
            VAR Vreal(), Vimag(), Watt(), MVAr() IS REAL
            PROC ExibirDadosExistentes(), ReceberAlteracao(),
                 MostrarSaida()
      PRIVATE:
            VAR I,J IS INTEGER
      PRIVATE BODY
            PROCEDURE ExibirDadosExistentes(NL,NB,BI,BF,Nbarra,TipoBarra,
                      IteMax,R,X,Y,ModV,AngV,PotAtiGer,PotReaGer,
                      PotAtiCar,PotReaCar,PotBase)
                  Texto1.Text="NL"
                  Texto2.Text="NB"
                  Texto3.Text="PotBase"
                  Texto4.Text="IteMax"
                  Texto5.Text="Conver"
                  I=0
                  J=0
                  FOR I=1 TO NL
                        FOR J=1 TO 5
                              Grid1.Col(J).Line(I).Text="BI(I)"
                              Grid1.Col(J).Line(I).Text="BF(I)"
                              Grid1.Col(J).Line(I).Text="R(I)"
                              Grid1.Col(J).Line(I).Text="X(I)"
                              Grid1.Col(J).Line(I).Text="Y(I)"
```

```
                NEXT I
            NEXT I
            I=0
            J=0
            FOR I=1 TO NB
                FOR J=1 TO 8
                    Grid2.Col(J).Line(I).Text="NBarra(I)"
                    Grid2.Col(J).Line(I).Text="TipoBarra(I)"
                    Grid2.Col(J).Line(I).Text="ModV(I)"
                    Grid2.Col(J).Line(I).Text="AngV(I)"
                    Grid2.Col(J).Line(I).Text="PotAtiGer(I)"
                    Grid2.Col(J).Line(I).Text="PotReaGer(I)"
                    Grid2.Col(J).Line(I).Text="PotAtiCar(I)"
                    Grid2.Col(J).Line(I).Text="PotReaCar(I)"
                NEXT I
            NEXT I
        END PROCEDURE
%
% Monitorar o teclado para a entrada dos dados pelo usuário. Este método
% é invocado toda vez que o usuário pressiona uma tecla.
%
        PROCEDURE ReceberAlteracao(NL,NB,BI,BF,Nbarra,TipoBarra,
                  IteMax,R,X,Y,ModV,AngV,PotAtiGer,PotReaGer,
                  PotAtiCar,PotReaCar,PotBase)
            NL=Texto1.Text
            NB=Texto2.Text
            PotBase=Texto3.Text
            IteMax=Texto4.Text
            Conver=Texto5.Text
            I=0
            J=0
            FOR I=1 TO NL
                FOR J=1 TO 5
                    BI(I)=Grid1.Col(J).Line(I).Text
                    BF(I)=Grid1.Col(J).Line(I).Text
                    R(I)=Grid1.Col(J).Line(I).Text
                    X(I)=Grid1.Col(J).Line(I).Text
                    Y(I)=Grid1.Col(J).Line(I).Text
                NEXT I
            NEXT I
            I=0
            J=0
            FOR I=1 TO NB
                FOR J=1 TO 8
                    NBarra(I)=Grid2.Col(J).Line(I).Text
                    TipoBarra(I)=Grid2.Col(J).Line(I).Text
                    ModV(I)=Grid2.Col(J).Line(I).Text
                    AngV(I)=Grid2.Col(J).Line(I).Text
                    PotAtiGer(I)=Grid2.Col(J).Line(I).Text
                    PotReaGer(I)=Grid2.Col(J).Line(I).Text
                    PotAtiCar(I)=Grid2.Col(J).Line(I).Text
                    PotReaCar(I)=Grid2.Col(J).Line(I).Text
                NEXT I
            NEXT I
        END PROCEDURE
%
% Exibir os dados de saída na tela do monitor.
%
        PROCEDURE MostrarSaida(NB,Nbarra,Vreal,Vimag,Watt,MVAr)
            I=0
            J=0
```

```
                FOR I=1 TO NB
                      FOR J=1 TO 5
                            Grid3.Col(J).Line(I).Text=Nbarra(I)
                            Grid3.Col(J).Line(I).Text=Vreal(I)
                            Grid3.Col(J).Line(I).Text=Vimag(I)
                            Grid3.Col(J).Line(I).Text=Watt(I)
                            Grid3.Col(J).Line(I).Text=MVAr(I)
                      NEXT I
                NEXT I
          END PROCEDURE
END CLASS;

MODULE 3

CLASS Gravador
      PUBLIC:
            VAR NL, NB, BI(), BF(), Nbarra(), TipoBarra(),
                IteMax IS INTEGER
            VAR R(), X(), Y(), ModV(), AngV(), PotAtiv(), PotReativa(),
                PotBase, Conver IS REAL
            VAR Vreal(), Vimag(), Watt(), MVAr() IS REAL
            PROC GravarDados()
            FUNC LerDados IS TYPE MATRIX(NL+NB+1,8)
      PRIVATE:
            VAR NomeArquivoLeitura NomeArquivoGravar IS STRING
            VAR Dados(,), DadosSaida(,) IS NUMERIC
            VAR I IS INTEGER
      PRIVATE BODY
%
% Esta função deve invocar uma caixa de dialogo para que o usuário
% forneça o nome do arquivo que deseja, para então ler-se os dados e
% retorna-los para o objeto que a chamou.
%
            FUNCTION LerDados
                  NomeArquivoLeitura = SEE DIALOGBOX1
                  OPEN NomeArquivoLeitura FOR READ
                        READ Dados
                        NL=Dados(1,1)
                        NB=Dados(2,1)
                        PotBase=Dados(3,1)
                        IteMax=Dados(4,1)
                        Conver=Dados(5,1)
                        I=0
                        FOR I=2 TO NL
                              BI(I-1)=Dados(1,I)
                              BF(I-1)=Dados(2,I)
                              R(I-1)=Dados(3,I)
                              X(I-1)=Dados(4,I)
                              Y(I-1)=Dados(5,I)
                        NEXT I
                        I=0
                        FOR I=NL+1 TO NB
                              NBarra(I-NL)=Dados(1,I)
                              TipoBarra(I-NL)=Dados(2,I)
                              ModV(I-NL)=Dados(3,I)
                              AngV(I-NL)=Dados(4,I)
                              PotAtiGer(I-NL)=Dados(5,I)
                              PotReaGer(I-NL)=Dados(6,I)
                              PotAtiCar(I-NL)=Dados(7,I)
                              PotReaCar(I-NL)=Dados(8,I)
                        NEXT I
```

```
            CLOSE NomeArquivoLeitura
        END FUNCTION
%
        PROCEDURE GravarDados(NL,NB,BI,BF,Nbarra,TipoBarra,IteMax,
                              R,X,Y,ModV,AngV,PotAtiGer,PotReaGer,
                              PotAtiCar,PotReaCar,PotBase)
            NomeArquivoGravar = SEE DIALOGBOX2
            OPEN NomeArquivoGravar FOR WRITE
                DadosSaida(1,1)=NL
                DadosSaida(2,1)=NB
                DadosSaida(3,1)=PotBase
                DadosSaida(4,1)=IteMax
                DadosSaida(5,1)=Conver
                I=0
                FOR I=2 TO NL
                    DadosSaida(1,I)=BI(I-1)
                    DadosSaida(2,I)=BF(I-1)
                    DadosSaida(3,I)=R(I-1)
                    DadosSaida(4,I)=X(I-1)
                    DadosSaida(5,I)=Y(I-1)
                NEXT I
                I=0
                FOR I=NL+1 TO NB
                    DadosSaida(1,I)=NBarra(I-NL)
                    DadosSaida(2,I)=TipoBarra(I-NL)
                    DadosSaida(3,I)=ModV(I-NL)
                    DadosSaida(4,I)=AngV(I-NL)
                    DadosSaida(5,I)=PotAtiGer(I-NL)
                    DadosSaida(6,I)=PotReaGer(I-NL)
                    DadosSaida(7,I)=PotAtiCar(I-NL)
                    DadosSaida(8,I)=PotReaCar(I-NL)
                NEXT I
                WRITE Dados
            CLOSE NomeArquivoGravar
        END PROCEDURE
END CLASS;

MODULE 4

CLASS Printer
    PUBLIC:
        VAR NL, NB, BI(), BF(), Nbarra(), TipoBarra(),
            IteMax IS INTEGER
        VAR R(), X(), Y(), ModV(), AngV(), PotAtiv(), PotReativa(),
            PotBase, Conver IS REAL
        VAR Vreal(), Vimag(), Watt(), MVAr() IS REAL
        PROC ImprimirDados()
    PRIVATE:
        VAR I IS INTEGER
    PRIVATE BODY
%
% Este procedimento deve invocar uma caixa de dialogo para que o usuário
% informe alguns dados para o status da impressora, do papel, quantidade
% de cópias, etc. Deve também formatar o relatório que será impresso,
% indicando os limites das linhas para as tabelas, o texto a ser escrito,
% o fim de linha e de página, etc. Alguns pacotes de linguagens OOP
% fornecem mecanismos para gerenciar o objeto Printer quando este é
% invocado.
%
        PROCEDURE ImprimirDados(NL,NB,BI,BF,Nbarra,TipoBarra,IteMax,
```

```
                                    R,X,Y,ModV,AngV,PotAtiGer,PotReaGer
                                    PotAtiCar,PotReaCar,PotBase,Vreal,
                                    Vimag,Watt,MVAr)
                  SEE DIALOGBOX3(Configurar o status)
                  PRINTER.PRINT
                  PRINTER.PRINT "Relatório Com os dados para o Arquivo:"+
                                NomeArquivoLeitura
                  PRINTER.PRINT
%
% Imprimir Linhas verticais e horizontais, definindo o papel de 0 a 100
% nos eixos dos x e y.
%
                  PRINTER.LINE (10,10)-(90,10)
                  PRINTER.LINE (90,10)-(90,90)
                  PRINTER.LINE (90,90)-(10,90)
                  PRINTER.LINE (10,90)-(10,10)
                  I=0
                  FOR I=1 TO NL THEN
                        PRINTER.PRINT BI(I),BF(I),R(I),X(I),Y(I)
                  NEXT I
                  I=0
                  FOR I=1 TO NB THEN
                        PRINTER.PRINT NBarra(I),TipoBarra(I),
                                      PotAtiGer(I),PotReaGer(I),
                                      PotAtiCar(I),PotReaCar(I)
                  NEXT I
            END PROCEDURE
END CLASS;

MODULE 5

CLASS NewtonRaphson
      PUBLIC:
            VAR NL, NB, BI(), BF(), Nbarra(), TipoBarra(),
                IteMax IS INTEGER
            VAR R(), X(), Y(), ModV(), AngV(), PotAtiv(), PotReativa(),
                PotBase, Conver IS REAL
            VAR Vreal(), Vimag(), Watt(), MVAr(), P(), Q(), Fator1(),
                Fator2(), E(), F(), G(), B() IS REAL
            FUNC Executar() IS TYPE MATRIX(0,4)
      PRIVATE:
            VAR I,J IS INTEGER
      PRIVATE BODY
%
% Esta função utiliza as rotinas desenvolvidas em MATLAB no ANEXO I,
% devidamente traduzidas para OOP, para cálculo do fluxo de carga. Deve
% ser lembrado que provavelmente ainda não existam métodos de inversão e
% tratamento de matrizes e de números complexos na linguagem OOP
% escolhida, sendo necessário então, criar-se tais funções ou rotinas,
% para que o programa possa executar sua função apropriadamente.
%
            FUNCTION Executar(NL,NB,BI,BF,Nbarra,TipoBarra,IteMax,
                              R,X,Y,ModV,AngV,PotAtiGer,PotReaGer
                              PotAtiCar,PotReaCar,PotBase)
                  Rotina 4
                  Rotina 6
                  Rotina 7
                  Rotina 8
                  Rotina 9
                  Rotina 10
                  Rotina 11
```

```
% ROTINA 12: CÁLCULO DA POTÊNCIA NA BARRA DE BALANÇO
%
                FOR I=NB
                      P(I)=0
                      Q(I)=0
                      FOR J=1 TO NB
                            Fator1(I)=  E(I)*(E(J)*G(I,J)+F(J)*B(I,J))+
                                        F(I)*(F(J)*G(I,J)-E(J)*B(I,J))
                            P(I)=P(I)+Fator1(I)
                            Fator2(I)=  F(I)*(E(J)*G(I,J)+f(J)*B(I,J))-
                                        E(I)*(F(J)*G(I,J)-E(J)*B(I,J))
                            Q(I)=Q(I)+Fator2(I);
                      NEXT J
                END I
                I=0
                J=0
                FOR J=1 TO NB
                      P(J)=P(J)* PotBase
                      Q(J)=Q(J)* PotBase
                      Vreal(I,J)= E(J)
                      Vimag(I,J)= F(J)
                      Watt(I,J)= P(J)
                      MVAr(I,J)= Q(J)
                NEXT J
%
% Fornecer a solução
%
                Executar(0,1)=Vreal(,)
                Executar(0,2)=Vimag(,)
                Executar(0,3)=Watt(,)
                Executar(0,4)=MVAr(,)

          END FUNCTION
END CLASS;
```