Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

Instituto de Ciências Matemáticas
Biblioteca
São Carlos

REENGENHARIA DE SOFTWARE

Marcos Antonio Quinaia

Rosely Sanches

DEDALUS - Acervo - ICMSC

30300022353

Nº 84

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
MARÇO/1999

Índice

Índice ii
Figuras ii
Quadros ii
Capítulo 1 Introdução 1
1.1 - Motivação para Reengenharia 1
1.2 - Objetivo 4
Capítulo 2 Reengenharia de Software 5
2.1 - Considerações Iniciais 5
2.2 - Definições de Reengenharia 5
2.3 - Engenharia Reversa 9
2.3.1 - Definições de Engenharia Reversa 12
2.3.2 - Visualização de Software 13
2.3.3 - Categorias de Engenharia Reversa 15
2.4 - Processo de Reengenharia 17
2.5 - Categorias de Reengenharia de Software 20
2.6 - Planejamento para Reengenharia 22
2.7 - Ferramentas de Auxílio à Reengenharia 24
2.8 - Considerações Finais 25
Capítulo 3 Tendências em Reengenharia de Software 27
3.1 - Motivação Para Transformar Sistemas Orientados a Função Para Sist. Orientados a Objeto . 27
3.2 - Tendências de Reengenharia nas Atividades das Empresas que Influenciam na Reengenharia de Software 28
3.3 - Considerações Finais 29
Bibliografia 30
Referências 30
Complementar 33

Figuras

Fig. 2.1 - Reengenharia de Software 10
Fig. 2.2 - Relacionamentos no Ciclo de Desenvolvimento de Software 11
Fig. 2.3 - Ciclo de Vida Genérico 12
Fig. 2.4 - Nível e Grau de Abstração no Ciclo de Vida de Software 13
Fig. 2.5 - Engenharias Progressiva e Reversa no Ciclo de Vida de Software 13
Fig. 2.6 -Visualizações de Software no Ciclo de Desenvolvimento 16
Fig. 2.7 - Visões das Categorias de Engenharia Reversa 19
Fig.2.8 - O Processo de Reengenharia de Dados 24
Fig. 2.9 - Matriz de Decisão Sobre o Destino de Software Legado 25
Fig. 2.10 - Abordagens de Reengenharia 26

Quadros

Quadro 2.1 - Escopo das informações utilizadas por cada ferramenta de reengenharia, e as respectivas visões e outras saídas produzidas 26

CAPÍTULO 1

INTRODUÇÃO

1.1 - Motivação para Reengenharia

Todos os sistemas têm um tempo de vida limitado e cada alteração efetuada no mesmo pode degenerar a sua estrutura, fazendo com que as manutenções se tornem cada vez mais difíceis e dispendiosas. Isso ocorre principalmente em *software legado* [Jacobson & Lindström, 1991].

Software legado pode ser informalmente definido como aquele que executa tarefas úteis para a organização, mas que foi desenvolvido utilizando-se técnicas atualmente consideradas obsoletas [Ward & Bennett, 1995]. A quantidade de código em sistemas legados é imensa. Em 1990, estimava-se [Ulrich, 1993] que existia 120 bilhões de linhas de código fonte de sistemas desse tipo. A maioria deles escrito em COBOL ou FORTRAN, linguagens com facilidades limitadas de estruturação de programas e estruturas de dados. A migração e / ou alteração desse tipo de software gera desafios técnicos (por exemplo: verificar efeitos colaterais de uma alteração) e não-técnicos (por ex.: estimar o custo de alteração) a todos os envolvidos com manutenção

de software.

Existe um grande dilema na decisão sobre o futuro de software legado. Ao mesmo tempo em que ele traz incorporado o acúmulo de anos de experiência e refinamento, traz também todos os vícios e defeitos vigentes na época de seu desenvolvimento, mesmo que naquela época, o que hoje é chamado de vício e defeito fosse a melhor indicação para o desenvolvimento de software [Bennett, 1995]. Por exemplo, o *tamanho do programa*, que devido a limitação imposta pelo custo do hardware, tinha que ser reduzido. Atualmente, com o barateamento dos componentes de hardware, o fator tamanho de programa já não é crucial. Houve ainda a época em que se prezava a *eficiência*, em detrimento da clareza e estruturação do programa. Adicione-se a isso as sucessivas manutenções e conseqüentes degenerações sofridas pelo software. Sabe-se hoje que todos esses fatores estão diretamente relacionados com a dificuldade de entendimento de qualquer software legado.

Muitos programas legados são críticos para os negócios das organizações que os possuem. Eles têm embutidos informações dos negócios e procedimentos, que podem não estar documentados. O risco de remover e reescrever tais programas é grande, pois muita informação teria que ser redescoberta por tentativa e erro. Consequentemente, as organizações, de um modo geral, não "aposentam" seus sistemas legados, preferindo mantê-los em operação, com adaptações às novas necessidades. As dificuldades devem ser enfrentadas, pois qualquer software que não evoluir continuamente (portanto passar por manutenções) tornar-se-á menos útil no mundo real [Lehman, 1980].

Segundo Bennett [1991], os maiores problemas em manter software legado são:

- Desestruturação e dificuldade de entendimento do código, muitas vezes porque o software foi desenvolvido antes da introdução dos métodos de programação estruturada.
- Programadores que não participaram do desenvolvimento de um produto de software sentem dificuldade em entender e mapear a funcionalidade para o código fonte.
- Documentação desatualizada, não auxiliando em nada a equipe de manutenção.
- Dificuldade de predizer as conseqüências de efeitos colaterais.
- Dificuldade de administrar múltiplas alterações concorrentes.

A complexidade e tamanho dos programas legados têm influência direta no custo

de manutenção dos mesmos [Ramamoorthy & Tsai, 1996]. Pesquisas revelam que mais de 50% do custo de um produto de software está relacionado com as atividades de manutenção, havendo casos deste percentual chegar até a 85% [Layzell *et al.*, 1995].

Em alguns negócios, estima-se que 80% de todos os gastos com software é consumido pelas manutenções e evoluções dos sistemas [Yourdon, 1990]. O número de sistemas que precisam ser modificados está aumentando gradualmente. Existe uma grande fila de espera para pedidos de manutenção. Isto significa que, algumas vezes, é impossível para as organizações investir em novos sistemas para melhorar a eficiência organizacional.

Uma pesquisa realizada em 100 grandes empresas norte americanas mostrou que, anualmente, estas empresas mantêm 35 milhões de linhas de código e, somente em atividades de manutenção, adicionam 3,5 milhões de linhas novas. Com isso deduziu-se que a quantidade de linhas de código de software dobra a cada 7 anos somente nessas 100 companhias. Estima-se ainda que o mercado de manutenção de software movimenta anualmente 30 bilhões de dólares [Mackenzie & Henshaw, 1994 apud Melymuka, 1991] e que somente no Reino Unido são gastos anualmente 1 bilhão de Libras em manutenção [Bennett, 1991].

O problema de manutenção de sistemas legados é complexo porque, geralmente, esses sistemas não são simples programas, desenvolvidos e mantidos. Muitos deles são compostos de diferentes programas que, de alguma forma, compartilham dados. Também ocorre que estes programas foram desenvolvidos por diferentes pessoas ao longo dos anos, os quais não estão mais na organização. Muitas vezes eles usam um sistema de administração de bases de dados que pode estar obsoleto, ou dependem de arquivos armazenados separadamente. No caso de arquivos separados, cada um tem seu próprio formato e freqüentemente as mesmas informações são duplicadas e representadas em diferentes formas e em diferentes arquivos. Esta duplicação usualmente acontece porque as informações são fortemente integradas com as estruturas de dados dos programas.

Com o objetivo de minimizar os problemas gerados por manutenções difíceis e algumas vezes degenerativas da estrutura do sistema, muitas organizações estão optando por refazer seus sistemas. A idéia básica dessa reconstrução ou *reengenharia* é que as informações de projeto e especificação sejam extraídas do código fonte, reformuladas e reconstruídas, resultando em um software mais fácil de ser mantido [Bennett, 1991].

Reengenharia de software consiste na reimplementação de um sistema em uma forma mais fácil de ser mantido. Com isso, o sistema pode ser redocumentado ou reestruturado, os programas podem ser traduzidos para uma linguagem de programação mais moderna e implementados em uma plataforma distribuída, bem como seus dados podem ser migrados para uma base de dados diferente. A reengenharia de software objetiva fazer sistemas flexíveis, fáceis de modificar, frente as constantes mudanças das necessidades dos usuários.

O campo da reengenharia está crescendo rapidamente em resposta à necessidade crítica que existe na indústria de software por tecnologia que suporte a manutenção de sistemas legados e o desenvolvimento evolutivo de novos sistemas. Existe uma firme e crescente demanda para migrar programas legados de mainframes monoprocessados, para estações de computadores, multiprocessadas, distribuídas e ligadas em rede, visando acompanhar os avanços das técnicas de programação tais como: interfaces gráficas para o usuário, comunicação interprogramas, desenvolvimento orientado a objeto, reusabilidade, etc. [Rugaber&Wills, 1996; Sneed&Nyáry, 1995]. Também o desenvolvimento de novos projetos de software deverão se deparar, cada vez mais, com processos de reengenharia, visto que desenvolvimentos de software, que não envolvam códigos antigos, são raros na prática atual [Coleman *et al.*, 1994].

1.2 - Objetivo

A reengenharia ou reconstrução de um sistema é indicada quando o software possui alta utilidade, porém é de difícil manutenção. A reengenharia envolve primeiramente um processo de engenharia reversa para a recuperação de informações importantes sobre o sistema e, posteriormente, a reconstrução do mesmo (engenharia progressiva), utilizando-se métodos que proporcionam maior qualidade ao sistema. Atualmente, os métodos orientados a objeto mostram ser a melhor perspectiva, tanto para o desenvolvimento quanto para a manutenção de software.

O objetivo deste trabalho é mostrar algumas possíveis tendências em reengenharia de software atualmente; para tanto, um capítulo de revisão bibliográfica sobre reengenharia é introduzido primeiramente.

CAPÍTULO 2

REENGENHARIA DE SOFTWARE

2.1 - Considerações Iniciais

A reengenharia de software envolve basicamente duas etapas: engenharia reversa e engenharia progressiva. Para melhor compreensão dos conceitos e propósitos da reengenharia, faz-se necessário o entendimento dos conceitos de engenharia reversa, categorias de reengenharia, o seu processo, definições e aplicações.

2.2 - Definições de Reengenharia

O tema reengenharia está relacionado com a reconstrução de algo do mundo real, e independentemente da aplicação da reengenharia, o seu principal propósito é a busca por melhorias que permitam produzir algo de qualidade melhor ou pelo menos de qualidade comparável ao produto inicial.

Sage [1995], indica diversas categorias de melhorias relacionadas à reengenharia,

entre elas: melhorias de sistemas administrativos, melhorias de processos e melhorias de produtos.

A *reengenharia de processos administrativos* é direcionada para alterações potenciais em todos os negócios ou processos organizacionais. Esta definição de reengenharia de processos administrativos é a mesma dada por Hammer e Champy [1993]: "reengenharia é o repensar fundamental e o reprojeto radical dos processos de uma organização, para realizar uma dramática melhoria em termos de performance, custo, qualidade e serviços".

A *reengenharia de processos produtivos* consiste de modificações em qualquer ciclo de processos padrão, que esteja em uso em uma dada organização, para melhor acomodar as tecnologias novas e emergentes ou requisitos dos clientes para um produto ou sistema. Indiretamente, melhoria nos processos de produção resulta em aumento de eficácia do produto pelo mesmo custo, ou na redução do custo pela mesma eficácia, ou ambos.

A *reengenharia de sistemas de software ou produtos* é o exame, estudo, captura e modificação de mecanismos internos ou funcionalidade de um sistema existente ou produto, visando reconstituí-lo em uma nova forma e com novas características, freqüentemente para tomar vantagem das novas e emergentes tecnologias, mas sem grandes alterações na funcionalidade e propósito inerentes do sistema.

Quando se efetua a reengenharia de processos administrativos provavelmente será necessário efetuar reengenharia de software, visto que existe uma dependência implícita entre os processos administrativos e o software [Sommerville, 1995].

Neste trabalho, é dado enfoque à reengenharia de sistemas de software a qual também é definida por outros autores:

- Para Chikofsky e Cross [1990], IEEE CS-TCSE [1997] e GT-REG [1998], a reengenharia, conhecida também como renovação ou reconstrução, é o exame e alteração de um sistema de software, para reconstituí-lo em uma nova forma, e a subseqüente implementação dessa nova forma. Um processo de reengenharia geralmente inclui alguma forma de engenharia reversa, seguida por uma forma de engenharia progressiva ou reestruturação.
- Para Warden [1992], a reengenharia tem como principal objetivo melhorar um sistema de alguma maneira; através de alterações significantes que proporcionem melhoria, porém, sem alterar suas funções. A extração

automática da descrição de uma aplicação e sua implementação em outra linguagem não é considerada, segundo o autor, reengenharia e sim tradução de código. Do mesmo modo que Chikofsky, Warden considera que a reengenharia pode ser dividida em duas fases principais: a Engenharia Reversa e a Engenharia Progressiva; e cada uma destas fases pode ser dividida em uma série de atividades.

- O padrão IEEE P1219/D14 [apud Sage, 1995], para manutenção de software define a reengenharia como um subconjunto da engenharia de software, composta por engenharia reversa e engenharia progressiva.
- Premerlani e Blaha [1994] citam que o objetivo da reengenharia é reutilizar automaticamente os esforços de desenvolvimento passados, objetivando reduzir custos de manutenção e melhoria na flexibilidade do software.
- Segundo Pressman [1995], a reengenharia, também chamada de *recuperação* ou *renovação*, recupera informações de projeto de um software existente e usa essas informações para alterar ou reconstituir o sistema, preservando as funções existentes, ao mesmo tempo em que adiciona novas funções ao software, num esforço para melhorar sua qualidade global.
- Para Sommerville [1995], a reengenharia de software é descrita como a reorganização e modificação de sistemas de software existentes, parcial ou totalmente, para torná-los mais manuteníveis.

Dessas definições percebe-se que existe clara distinção entre o desenvolvimento de um novo software e reengenharia de software. A distinção está relacionada ao ponto de partida de cada um dos processos. O desenvolvimento de um novo software (definido como *engenharia progressiva* por Chikofsky e Cross [1990]) inicia-se com uma especificação escrita do software que será construído, enquanto que a reengenharia inicia-se tomando como base um sistema já desenvolvido.

Das definições nota-se também que existe distinção entre os objetivos da reengenharia e os da engenharia reversa. O objetivo da engenharia reversa é derivar o projeto ou especificação de um sistema, partindo-se de seu código fonte [Sommerville, 1995]. O objetivo da reengenharia é produzir um sistema novo mais manutenível e a engenharia reversa é usada como parte do processo de reengenharia, para o entendimento do sistema a ser reconstruído. Estas distinções estão na figura 2.1.

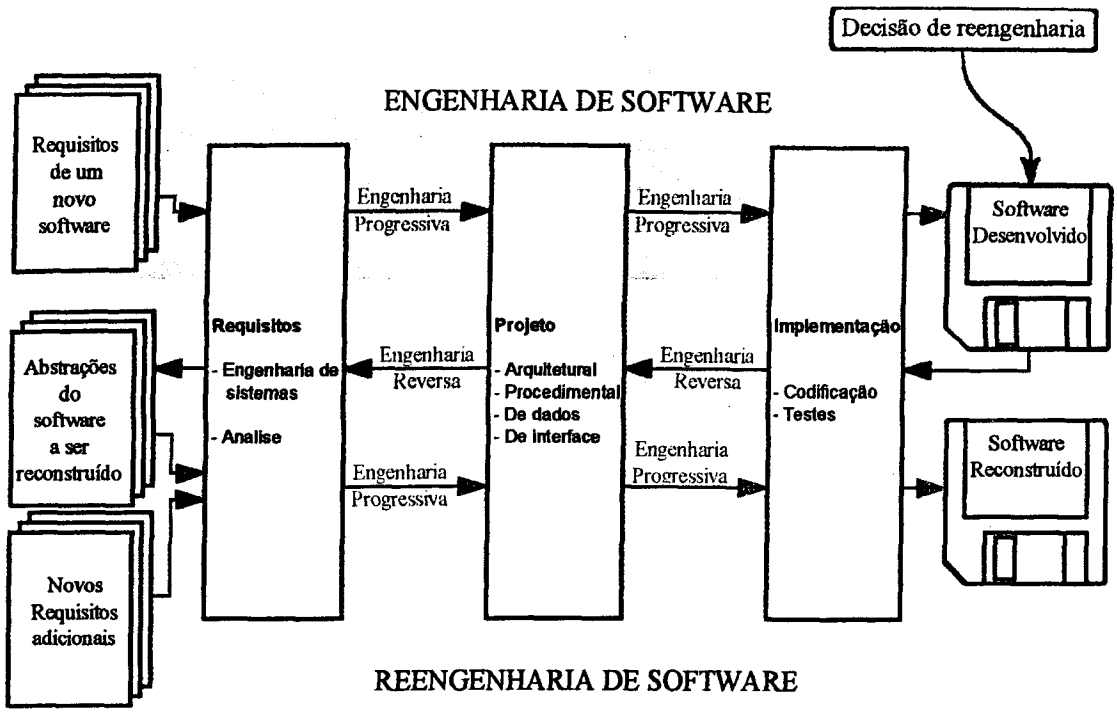

FIG. 2.1 - REENGENHARIA DE SOFTWARE

Outros conceitos relacionados à reengenharia são: redocumentação, reestruturação, recuperação de projeto e engenharia reversa (figura 2.2) [Chikofsky & Cross, 1990].

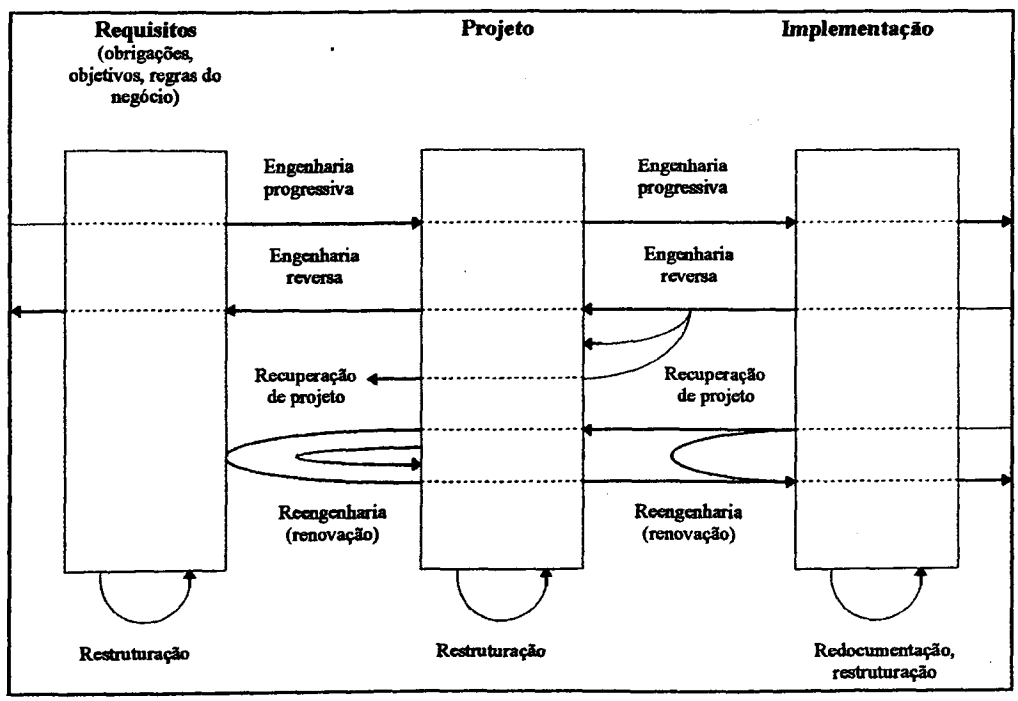

FIG. 2.2 - RELACIONAMENTOS NO CICLO DE DESENVOLVIMENTO DE SOFTWARE [CHIKOFSKY & CROSS, 1990]

- *Redocumentação.* Como uma subárea da engenharia reversa, a redocumentação é a criação ou revisão de uma representação semanticamente equivalente, dentro do mesmo nível relativo de abstração, sendo que as formas resultantes de representação são consideradas como visões alternativas, utilizadas para uma melhor compreensão humana do sistema analisado.

- *Recuperação de projeto.* É uma subárea da engenharia reversa na qual o conhecimento do domínio da aplicação, informações externas, e dedução são adicionadas às observações referentes ao programa, para se extrair abstrações significativas de mais alto nível, além daquelas obtidas através da observação direta do sistema.

- *Reestruturação.* É a transformação de uma forma de representação, para outra no mesmo nível de abstração relativo, preservando o comportamento externo do sistema (funcionalidade e semântica). Geralmente usada como uma forma de manutenção preventiva, a reestruturação aplica-se em sistemas que tenham sido desenvolvidos de forma desestruturada, resultando em uma representação que preserva as características do sistema, porém de forma mais bem estruturada.

2.3 - Engenharia Reversa

O ciclo de vida de software refere-se às etapas (ou estágios) da existência do software. Existem diversos modelos de ciclo de vida: o modelo em cascata [Royce, 1970], o evolutivo, a quarta geração e o espiral [Boehm, 1988], entre outros.

Independentemente do modelo de ciclo de vida adotado, o processo de desenvolvimento de software começa pela especificação dos *requisitos*, passa pelo *projeto* e termina em *implementação* do software (figura 2.3).

No desenvolvimento de software, quando se parte dos requisitos, para a implementação, tem-se um aumento do nível de detalhes e, portanto uma diminuição na abstração.

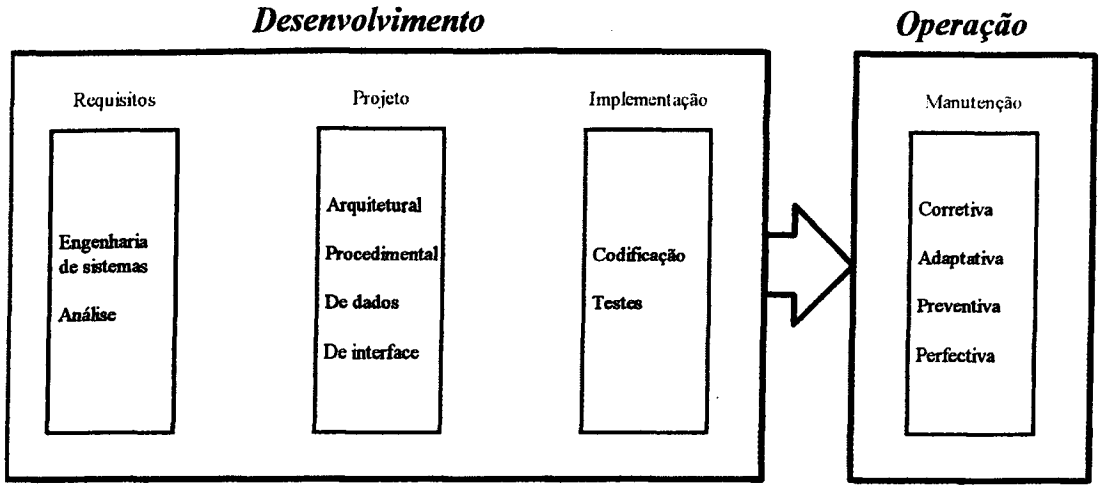

FIG. 2.3 - CICLO DE VIDA GENÉRICO

Abstração é definida como a habilidade de se ignorar os aspectos de assuntos não relevantes para o propósito em questão, tornando possível uma concentração maior nos assuntos principais [Oxford, 1986]. Dois conceitos relacionam abstração com as atividades do ciclo de vida:

Nível de abstração: quanto mais próximo o projeto estiver do nível de implementação em máquina, menor será o nível de abstração do projeto. Por outro lado, quanto mais próximo o projeto estiver da fase de especificação dos requisitos do sistema, maior será o nível de abstração do mesmo. Geralmente no desenvolvimento de software, parte-se de informações em alto nível de abstração, e à medida que o projeto for caminhando através das etapas do ciclo de vida, o nível de abstração vai diminuindo [Chikofsky & Cross, 1990].

Grau de abstração: dentro de uma mesma etapa (ou fase) do ciclo de vida, se as informações forem pouco detalhadas, diz-se que é alto o grau de abstração, já se as informações forem muito detalhadas, diz-se que é baixo o grau de abstração [Chikofsky & Cross, 1990].

Resumindo, o grau de abstração é interno a cada fase do ciclo de vida, e o nível de abstração acontece entre as fases do ciclo de vida (figura 2.4).

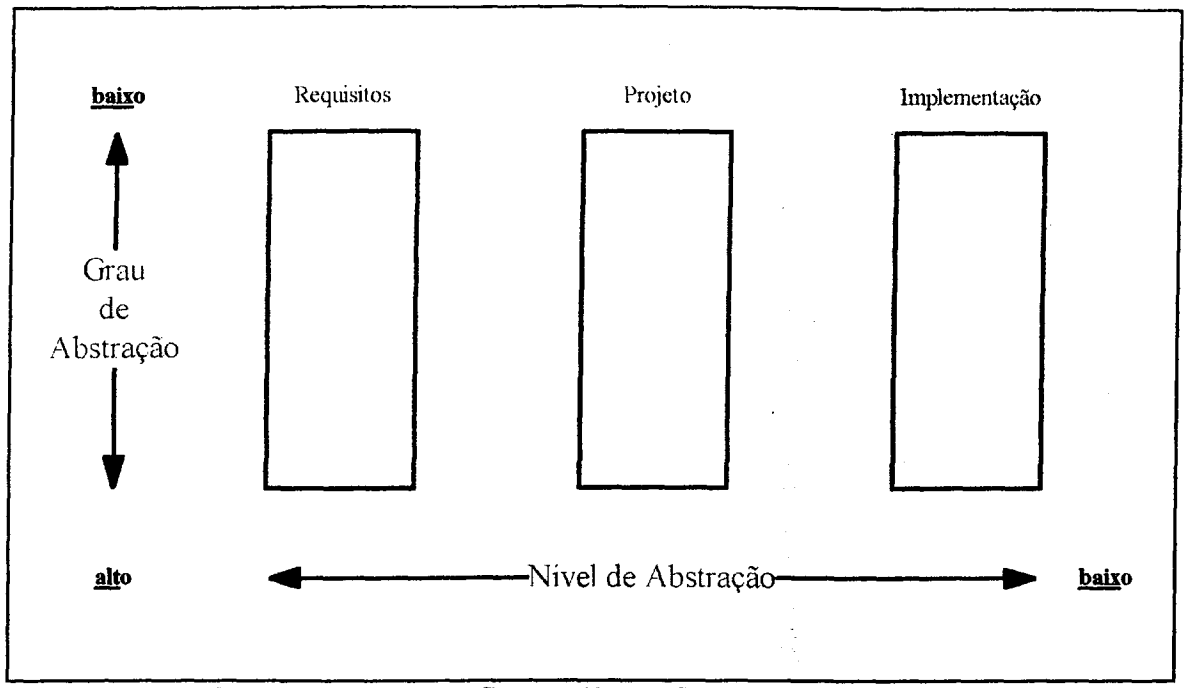

FIG. 2.4 - NÍVEL E GRAU DE ABSTRAÇÃO NO CICLO DE VIDA DE SOFTWARE [COSTA, 1997]

O processo tradicional de engenharia de software, caracterizado pelas atividades progressivas do ciclo de vida, que partem de um alto nível de abstração, para um baixo nível de abstração, é conhecido como *engenharia progressiva*. Já o processo inverso à engenharia progressiva, caracterizado pelas atividades retroativas do ciclo de vida, que partem de um baixo nível de abstração para um alto nível de abstração, é conhecido como *engenharia reversa* (figura 2.5) [Chikofsky & Cross, 1990].

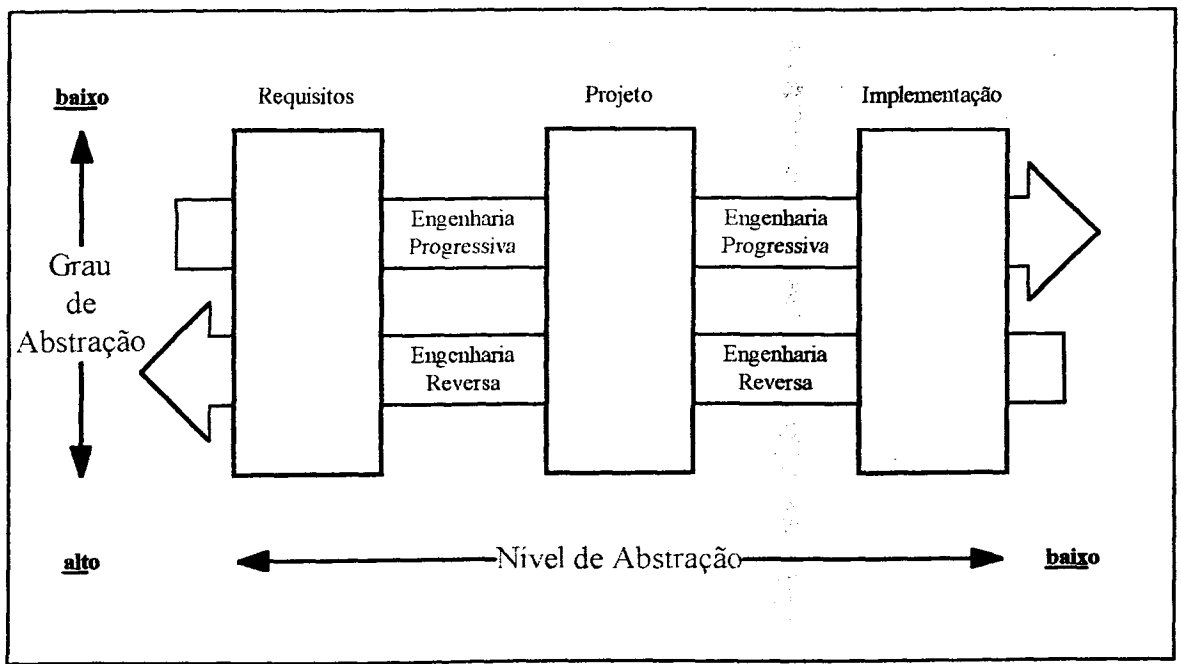

FIG. 2.5 - ENGENHARIAS PROGRESSIVA E REVERSA NO CICLO DE VIDA DE SOFTWARE [COSTA, 1997]

2.3.1 - Definições de Engenharia Reversa

O termo engenharia reversa originou-se na engenharia de hardware, onde o mesmo é considerado como o processo de desenvolvimento de especificações para um sistema de hardware complexo, através de exame minucioso de exemplares do sistema de hardware em estudo [Rekoff, 1985].

O conceito de Engenharia Reversa de Software é similar. Porém, tradicionalmente o objetivo da engenharia reversa de hardware é duplicar o sistema, enquanto o objetivo da engenharia de software freqüentemente é obter um entendimento do sistema em um nível de abstração mais alto.

Para Pressman, a engenharia reversa de software é um processo de recuperação de projeto, consistindo em analisar um programa, na tentativa de criar uma representação do mesmo, em um nível de abstração mais alto que o código-fonte [Pressman, 1995].

Para Gannod e Cheng, [1996], engenharia reversa de um programa é o processo de construir uma abstração de alto nível objetivando facilitar o entendimento de um sistema legado ou geriátrico.

Segundo Benedusi, pode-se definir engenharia reversa como uma coleção de teorias, metodologias, e técnicas capazes de suportar a extração e abstração de informações de um software existente, produzindo documentos consistentes, quer seja a partir somente do código fonte, ou através da adição de conhecimento e experiência que não podem ser automaticamente reconstruídos a partir do código [Benedusi *et al.*, 1992 apud Ramamoorthy *et al.*, 1986; Chikofsky & Cross, 1990].

Para Samuelson, engenharia reversa é geralmente entendida como a ação de criar um conjunto de especificações funcionais para um sistema, por alguém que não foi o projetista original, baseado na análise de um sistema existente e suas partes componentes [Samuelson, 1990 apud Rekoff, 1985].

Segundo Waters e Chikofsky, o processo de análise de um sistema para identificar os componentes de um software e seus inter-relacionamentos, e para criar uma representação do software em outra forma, provavelmente num nível mais alto de abstração que o código fonte, é engenharia reversa [Waters & Chikofsky, 1994].

Stephen chama de engenharia reversa, entender, analisar, e abstrair o sistema para uma nova forma, num alto nível de abstração [Stephen & Lynn, 1995].

A partir da engenharia reversa e com base nos diferentes níveis e graus de abstração o software pode ser visualizado de diferentes maneiras.

2.3.2 - Visualização de Software

As visões que se obtêm do software podem ser classificadas em 4 *categorias*: visão em nível-implementacional, visão em nível-estrutural, visão em nível-funcional, visão em nível-de-domínio [Harandi & Ning, 1990].

Visão em nível-implementacional. Abstrai características da linguagem de programação e características específicas da implementação. Exemplos de visões em nível implementacional são informações a respeito da sintaxe e da semântica da linguagem e informações da implementação.

Visão em nível-estrutural. Abstrai detalhes da linguagem de programação para revelar sua estrutura a partir de diferentes perspectivas. O resultado é uma representação explícita das dependências entre os componentes do sistema. Exemplos de visões em nível estrutural são grafos de fluxo de dados, grafos de fluxo de controle [Cleveland, 1989], o projeto procedimental expresso através de uma linguagem de projeto de programação [Caine & Gordon, 1975; Harband, 1990], o projeto arquitetural expresso através de gráficos de estruturas [Cross, 1990], ou através de uma linguagem de interconexão modular, como NuMIL [Choi & Scacchi, 1990].

Visão em nível-funcional. Abstrai a função de um componente, isto é, *o que* o componente faz. Essa visão relaciona partes do programa às suas funções procurando revelar as relações lógicas entre elas (diferentemente das relações sintáticas ou das estruturais). Exemplos de visões em nível funcional podem ser descrições da função do sistema expressos de maneira formal, usando linguagens tais como VDM [Jones, 1990], Z e Z^{++} [Spivey, 1988; Bennet, 1991]; diagramas de fluxo de dados, que apresentam os processos e o fluxo de dados entre eles [DeMarco & Sarson, 1979]; diagramas de fluxo de controle [Hatley & Pirbhai, 1987], que apresentam os processos e o fluxo de controle entre eles e diagramas de entidade-relacionamento [Ross, 1988], que mostram os dados e seus relacionamentos.

Visão em nível-de-domínio. Abstrai o contexto em que o sistema está operando, ou seja, o *porquê* do sistema a ser desenvolvido.

Na realidade poucas representações são restritas somente a uma fase do ciclo de vida ou consideradas como pertencentes a uma categoria de visualização. Uma linguagem de projeto de programa [Caine & Gordon, 1975] pode ser usada para representar as funções na fase de requisitos ou o projeto procedimental na fase de projeto. Um diagrama de fluxo de dados [DeMarco & Sarson, 1979], pode ser usado para descrever o que o sistema faz ou como o processo interage. Assim, se uma representação extraída é uma representação de requisitos ou uma representação de projeto depende principalmente do contexto em que a representação será usada.

É relevante ressaltar que uma forma de representação extraída do código pode diferir de uma representação similar que foi desenvolvida no processo de engenharia progressiva. A forma extraída irá refletir a idiossincrasia da representação do código muito mais do que a representação original, que reflete a compreensão do problema pelo analista ou projetista.

A figura 2.6 mostra a correspondência entre as categorias de visualização do software e as diferentes atividades do ciclo de desenvolvimento de software.

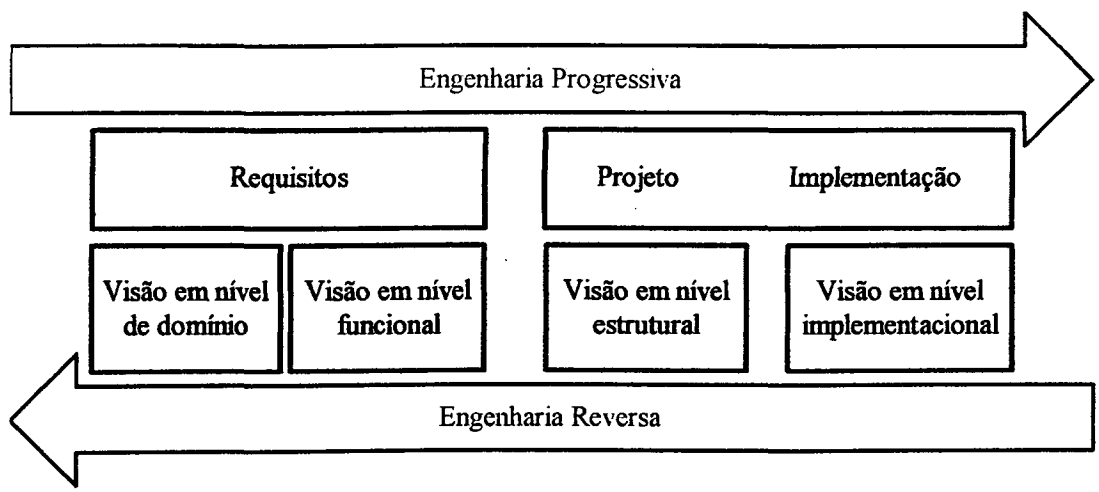

FIG. 2.6 -VISUALIZAÇÕES DE SOFTWARE NO CICLO DE DESENVOLVIMENTO [COSTA, 1997]

Muitas vezes é necessário acrescentar às informações contidas no código outras informações provenientes de conhecimentos e experiências humanas, para se obter

"visões" diferenciadas do software. Conforme o escopo das informações usadas que resultarão em um nível de entendimento obtido do sistema, pode-se formular uma categorização dos métodos de engenharia reversa.

2.3.3 - Categorias de Engenharia Reversa

De acordo com o nível de entendimento obtido do sistema e o escopo das informações usadas, duas categorias de engenharia reversa são definidas: *visualização de código* [Oman, 1990] e *entendimento de programa* [Chikofsky & Cross, 1990].

Visualização de código: Também denominada *redocumentação*. É a criação ou revisão de representações semanticamente equivalentes num mesmo nível de abstração [Chikofsky & Cross, 1990]. O processo de visualização de código cria as representações a partir de informações obtidas apenas da análise do código fonte, embora a apresentação dessas informações possa se diversificar. As formas das representações são consideradas visões alternativas, cujo objetivo é melhorar a compreensibilidade do sistema global.

A forma mais simples e mais antiga de engenharia reversa é a visualização de código. A intenção é recuperar a documentação que já existiu, ou que deveria ter existido, sobre o sistema. A ênfase, de fato, é a criação de visões adicionais, especialmente visões gráficas, que não foram criadas durante o processo original de engenharia progressiva.

A visualização de código não transcende a visão em *nível-estrutural* e não atribui significados ao sistema analisado. Recuperações mais ambiciosas tais como a função, os propósitos ou a essência do sistema exigem um nível de entendimento maior e são definidas como *entendimento de programa*.

Entendimento de programa: Nesta categoria de engenharia reversa, também denominada *recuperação de projeto*, o conhecimento do domínio das informações externas e as deduções são adicionadas às observações feitas sobre o sistema através do exame do mesmo, de modo a obter informações com nível mais alto de abstração [Chikofsky & Cross, 1990].

Segundo Biggerstaff [1989], *entendimento de programa* recria abstrações do

projeto a partir de uma combinação de código, documentação existente do projeto (se disponível), experiências pessoais, e conhecimentos gerais sobre o problema e o domínio de aplicação. Sintetizando, entendimento de programa deve produzir todas as informações necessárias para se entender completamente *o que*, *como*, e *por que* o sistema faz.

Até o momento pesquisas [Ramamoorthy & Tsai, 1996], revelam que os métodos de engenharia reversa não recuperam de modo automático todas as visões do software. Isto acontece basicamente porque a fase de implementação - caracterizada principalmente por programas fonte e descrição de arquivos - não contém todas as informações essenciais para o processo de engenharia reversa, as quais são providas pela fase de análise, sendo necessária a intervenção humana para extrair boas representações de projetos de software, principalmente em níveis mais altos de abstração [Tangorra & Chiarolla, 1995]. Segundo Pressman [1995], as ferramentas baseadas em engenharia reversa estão ainda "engatinhando" e as pesquisas nesta área de entendimento de código são muito importantes e muitas outras ainda acontecerão.

Entendimento de programa distingue-se de visualização de código porque objetiva *entender* o sistema, em vez de simplesmente fornecer visões alternativas para *auxiliar* o usuário a entender o sistema. Esse entendimento vai além do conhecimento em *nível implementacional e estrutural*, buscando obter o conhecimento em *nível-funcional* e até mesmo em *nível-de-domínio* (ambiente de operação do sistema).

Um completo entendimento de programa busca reconstruir não somente a função do sistema, mas também o processo pelo qual o sistema foi desenvolvido. Rugaber *et al.* [1990] enfatizam a importância da recuperação de decisões de projeto tomadas durante o desenvolvimento original para uma completa estrutura de entendimento.

Entendimento de programa é a forma mais crítica de engenharia reversa porque tenta aproximar-se do raciocínio humano na busca do entendimento.

A figura 2.7 apresenta a amplitude de alcance das categorias de engenharia reversa, relacionadas com o escopo das informações utilizadas (código fonte ou base de conhecimento) e o nível de visualização pretendida (implementacional, estrutural, funcional e de domínio).

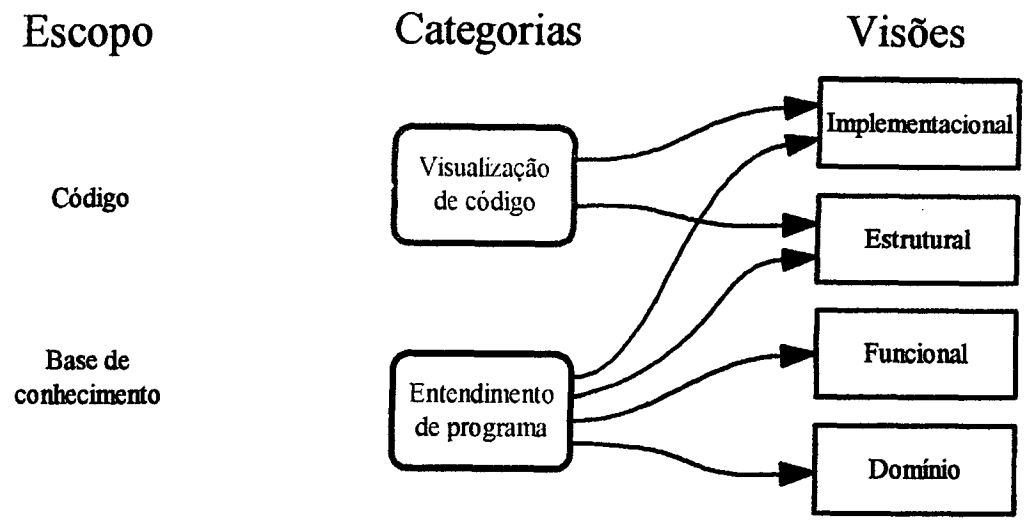

FIG. 2.7 - VISÕES DAS CATEGORIAS DE ENGENHARIA REVERSA

2.4 - Processo de Reengenharia

O processo de reengenharia de software é constituído de duas fases distintas, a primeira "desmonta" o software objeto de reconstrução, para seu entendimento e a segunda o constrói novamente, em uma nova forma desejada, a partir do produto da primeira fase juntamente com os ajustes que se fizerem necessários.

O processo de Reengenharia pode ser traduzido como [Jacobson & Lindström 1991]:

Reengenharia = Engenharia Reversa + Δ + Engenharia Progressiva

Onde Δ pode ser de dois tipos:

1. *Alterações de funcionalidade* (Alteram o objetivo principal do sistema): Ocorrem devido a mudanças nos negócios, ou necessidade do usuário.
2. *Alterações de implementação* (Alteram a forma de implementação do sistema): ocorrem devido a alterações no ambiente de operação do software e ou linguagem de implementação (protocolos, sistema operacional, portabilidade, linguagens, etc.).

Existem alguns pontos que devem ser considerados para um processo de reengenharia [Warden, 1992]:

- Reengenharia deve ser executada somente se existir um argumento aceitável de custo/benefício.
- Reengenharia implica melhoria através de reprojeto.

- Reengenharia deve remover projetos ruins, mas reconhecer e manter projetos bons e simples, mesmo se ele for desestruturado.
- Engenharia Reversa é dirigida por tipos de problemas, os quais necessitam ser identificados.
- Problemas são identificados e expressos como violações às técnicas de projeto estruturado e regras de programação, ou outras que o usuário pode definir.
- Ferramentas devem ser adequadas aos processos de reengenharia, e não os processos adequados às ferramentas.

Jacobson e Lindström [1991] declaram que para executar um processo de reengenharia de um sistema é necessário:

1. *Engenharia reversa*: Identificar como os componentes do sistema se relacionam uns aos outros e então criar uma descrição mais abstrata do sistema. Um exemplo de identificação de relacionamentos entre componentes pode ser a identificação de dependências entre os arquivos e as funções, entre as funções e as descrições da base de dados, etc.. Um exemplo da criação de uma descrição mais abstrata do sistema pode ser um diagrama de fluxo de dados para as funções e o modelo entidade-relacionamento para as descrições da base de dados. Com esse primeiro passo obtém-se um modelo abstrato o qual mostra a funcionalidade do sistema (propósito para o qual o sistema foi construído) e um número de mapeamentos entre os diferentes níveis de abstração. Os mapeamentos compreendem as decisões de projeto que ocorrem quando se transforma uma representação abstrata em uma representação concreta.

2. *Decidir sobre alterações na funcionalidade*: as alterações de funcionalidade são as alterações nos requisitos que o usuário determina que sejam implementadas no sistema. Esse passo é executado utilizando-se as abstrações de mais alto nível, obtidas no passo anterior. Sem o modelo abstrato é necessário decidir questões de alto nível, utilizando-se comandos de baixo nível, por exemplo, um comando de alto nível tal como "Altere a associação entre as entidades X e Y" deveria ser traduzido para um outro de baixo nível tal como "Adicione uma tabela que contenha as referências entre X e Y".

3. *Reprojetar o sistema*: reprojeta-se o sistema partindo-se das abstrações de alto nível, obtidas nos passos anteriores, para uma representação mais concreta, ou seja, executa-se a engenharia progressiva reimplementando o sistema. Neste

processo, deve-se levar em consideração as alterações de técnicas de implementação. Se apenas parte do sistema for alterado, deve-se considerar questões sobre a integração/comunicação entre as partes velhas e novas do sistema.

Gall e Klosch [1995], relatam um processo para encontrar objetos em programas procedimentais. Esse processo de identificação de objetos é baseado nas informações derivadas do código fonte, integradas com conhecimento específico do sistema e do domínio da aplicação, os quais resultam em representações das semânticas das aplicações em objetos. A identificação de objetos inicia-se com a geração de documentos de projeto de baixo nível, como diagramas estruturados, e diagramas de fluxo de dados, sendo estes a base para o processo de identificação dos objetos.

Sneed e Nyáry [1995], relatam uma abordagem para extrair automaticamente documentação de projeto orientada a objeto a partir de programas escritos em COBOL para mainframe; mapas ou painéis para comunicação com o usuário; bases de dados armazenados e cartões de controle de execução do programa.

Yeh *et al.* [1995], abordam a recuperação de representações arquiteturais de software, partindo do código fonte. A experiência inclui desenvolvimento, implementação e teste de uma abordagem interativa para recuperação de tipos abstratos de dados (TADs) e instâncias de objetos extraídos de programas escritos em linguagens convencionais como C.

Newcomb e Kotik [1995], descrevem uma ferramenta de reengenharia para transformar automaticamente um sistema procedimental em sistema orientado a objeto sem alteração de funcionalidade O processo de transformação em forma orientada a objeto localiza dados e procedimentos redundantes, duplicados e similares e abstrai-os em classes e métodos.

Lewis e McConnell [1996], descrevem um processo geral de reengenharia de software usado em experimentos com software de míssil. O experimento envolve reimplementação partindo-se de monoprocessamento, para multiprocessamento escalável e distribuído. O processo tem os seguintes passos: 1) configuração da engenharia reversa e reengenharia; 2) tradução para ADA95; 3) validação e correção; 4) reestruturação; 5) transformação arquitetural e ligação do novo código e 6) reimplementação para processadores distribuídos.

2.5 - Categorias de Reengenharia de Software

Jacobson e Lindström [1991], categorizam a reengenharia segundo alguns *cenários,* onde ocorre a transformação de sistemas orientados à função para sistemas orientados a objeto:

a) Reengenharia completa da técnica de implementação, sem alteração de funcionalidade: este cenário raramente ocorre em um grande sistema, porque demanda uma mudança radical de paradigma, devido às alterações efetuadas em todo o sistema. Os principais passos do processo de reengenharia para este cenário são:

- Preparar um modelo de análise, com base no sistema a ser reconstruído;
- Mapear cada objeto de análise para a implementação do sistema velho;
- Reprojetar o sistema usando uma técnica orientada a objeto para desenvolvimento de sistemas de software.

b) Reengenharia parcial sem alteração de funcionalidade: Neste cenário o sistema antigo é alterado parcialmente. O objetivo é fazer a aplicação orientada a objeto "acreditar" que o sistema inteiro consiste de objetos. Este cenário requer alguns passos discriminados a seguir:

- Identificar as partes do sistema que serão reimplementadas usando técnica orientada a objeto;
- Preparar um modelo de análise da parte a ser remodelada e seu ambiente;
- Mapear cada objeto da implementação antiga do sistema;
- Iterar os passos anteriores até a interface entre as parte velhas e novas do sistema ser aceitável;
- Executar em paralelo:
  - ◊ Projetar o novo subsistema e sua interface com o resto do sistema antigo;
  - ◊ Modificar o antigo sistema e adicionar uma interface para o novo subsistema.
- Integrar e testar as partes novas e antigas do sistema.

c) Reengenharia com alteração de funcionalidade: Este cenário é um processo normal de engenharia progressiva, onde se adiciona as novas funcionalidades que se quer no sistema e o implementa usando-se uma técnica orientada a objeto. Os principais passos deste cenário são:

- Alterar o modelo de análise de acordo com os novos requisitos para o sistema.
- Reprojetar o sistema.

A categorização de reengenharia, segundo Sommerville [1995] está relacionada à

profundidade (grau) com que será feita a reengenharia do software:

a) Conversão de programas fonte: É a mais simples das formas de reengenharia, onde o código fonte escrito em uma linguagem de programação é traduzido para código fonte em outra linguagem. A linguagem destino pode ser uma versão mais nova da mesma linguagem, ou uma linguagem completamente diferente. Este tipo de reengenharia pode ser útil quando, por exemplo, houver alteração da plataforma de hardware, falta de experiência do pessoal de manutenção na linguagem, ou alterações nos padrões da organização.

b) Reestruturação de programas: aplicável quando a estrutura do sistema está corrompida, dificultando o seu entendimento, como por exemplo: existência de funções ou procedimentos nunca chamados ou duplicados, existência de desvios *gotos*, existência de estruturas condicionais complexas, etc. Esta categoria se subdivide em:

- Reestruturação automática de programas
- Reestruturação automática de programa com alterações manuais
- Reestruturação de programas e dados.

c) Reengenharia de dados: Envolve a análise e reorganização de estruturas de dados em um programa. A reengenharia de dados pode também envolver, além da reorganização das estruturas de dados, a conversão dos valores dos dados. Para o processo de reengenharia de dados é necessária a criação de programas especiais para proceder a conversão dos dados. A reengenharia de dados é geralmente executada quando se procede a transformação de um software para utilizar um sistema de administração de bases de dados, ou migrar de uma base de dados para outra. A fig. 2.8 mostra o processo de reengenharia de dados, incluindo a conversão de dados.

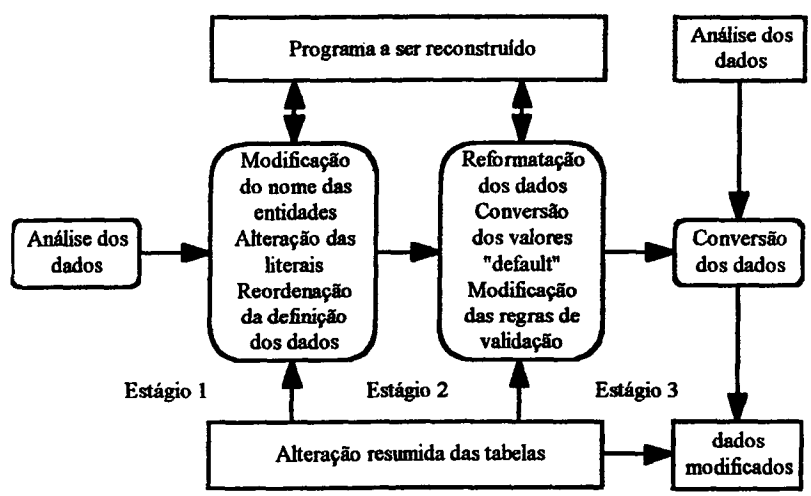

FIG.2.8 - O PROCESSO DE REENGENHARIA DE DADOS [SOMMERVILLE, 1995]

Instituto de Ciências Matemáticas - São Carlos - Biblioteca

2.6 - Planejamento para Reengenharia

O planejamento para a reengenharia configura uma atividade que busca estabelecer os passos para o sucesso de um empreendimento de reengenharia.

Em vista da complexidade encontrada em se gerenciar um processo de reengenharia, dado o fato de que é difícil migrar ou atualizar programas e dados, Sneed [1995] apresenta um processo de planejamento de projetos de reengenharia em cinco passos:

1. *Justificativa do projeto.* Proceder um minucioso estudo sobre os sistemas da organização: seu processo de manutenção e sua importância, visando justificar o empreendimento do processo de reengenharia. Esta justificativa deve projetar retorno do investimento em reengenharia, enfatizando o aumento da qualidade do software, a melhoria no processo de manutenção e o conseqüente aumento do valor do software para a organização.
2. *Análise do portifólio.* Estabelecer critérios para a escolha das aplicações que são candidatas a reengenharia de acordo com sua qualidade técnica e valor para a organização. Essa análise pode ser feita utilizando-se a figurar 2.9 para a determinação das aplicações de software candidatas a reengenharia.
3. *Estimativa de custo.* Estimar o custo de um projeto de reengenharia, pela identificação avaliação-ponderação de todos os componentes de software a serem reconstruídos. Estes componentes podem incluir: reengenharia do software, teste e reengenharia de dados.
4. *Análise de custo-benefício.* Com base nos custos estimados no passo anterior, proceder uma comparação do custo de fazer reengenharia com o custo esperado de manutenção. Se os custos de reengenharia forem altos, deve-se procurar outras alternativas como compra de software pronto, ou redesenvolvimento.
5. *Contrato externo.* Existem basicamente dois tipos de contratos que podem ser usados em processos de reengenharia: contrato por tempo e material e contrato por resultados. Os contratos visam alcançar um nível máximo de distribuição de projetos para evitar gargalos.

Para a tomada de decisão sobre reengenharia de software é importante fazer-se uma análise de custo-benefício. A matriz da figura 2.9 [Jacobson & Lindström, 1991] mostra como analisar a viabilidade entre substituir, manter, melhorar ou efetuar a reengenharia de um sistema de software legado.

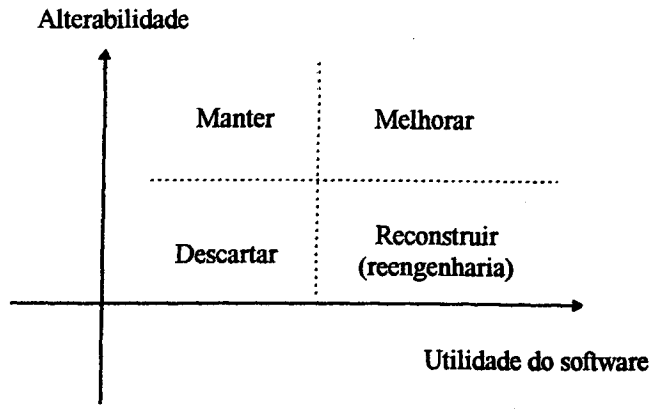

FIG. 2.9 - MATRIZ DE DECISÃO SOBRE O DESTINO DE SOFTWARE LEGADO

Pode-se visualizar na figura 2.9 que sistemas fáceis de alterar, porém com pouca utilidade para a empresa, devem ser mantidos sob manutenção. Os difíceis de alterar e com menos utilidade, são indicados a serem descartados ou substituídos por outros. Por outro lado, os sistemas que tiverem um bom índice de alterabilidade, e forem de grande utilidade, devem continuar evoluindo. Somente os que forem de baixa alterabilidade e de alta utilidade para a empresa devem ser reconstruídos.

Os custos de um processo de reengenharia têm uma relação direta com a categoria (tipo) de reengenharia. A profundidade e extensão com que for executada a reengenharia, definirão os seus custos. Outros fatores que terão influência sobre os custos de reengenharia são: a qualidade do software a ser reconstruído; as ferramentas de suporte disponíveis para auxiliar a reengenharia; a extensão da conversão dos dados requerida; a disponibilidade de pessoal qualificado para reengenharia.

A figura 2.10 [Sommerville, 1995], mostra as categorias de reengenharia de software e a relação do seu custo com a profundidade com que é executada.

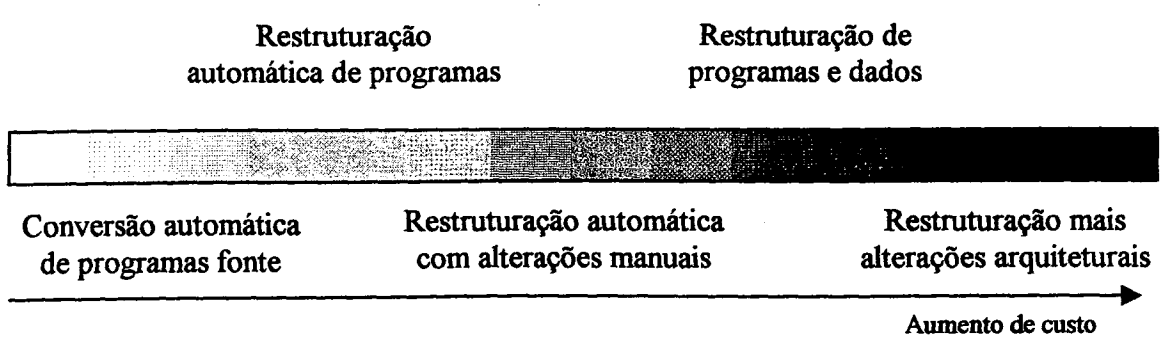

FIG. 2.10 - ABORDAGENS DE REENGENHARIA [SOMMERVILLE, 1995]

Se, em uma análise de custo/benefício, os custos de reengenharia de um sistema

de software forem considerados maiores que o redesenvolvimento de sistemas usando técnicas modernas de engenharia de software; então essa alternativa à reengenharia deve ser considerada. No entanto, a reengenharia tem duas vantagens chaves sobre o redesenvolvimento de sistemas [Sommerville, 1995]:

- Risco reduzido - Quando um software é essencial para uma determinada empresa, seu redesenvolvimento é uma atividade de risco, se comparada com a reengenharia do mesmo.
- Custo Reduzido - O custo de reengenharia é significativamente menor que o custo de desenvolvimento, talvez da ordem de quatro vezes ou menos.

No âmbito comercial, existe uma variedade de motivos para a aplicação de reengenharia [Warden, 1992]:

- Necessidade de aumentar a vida econômica de um sistema;
- Necessidade de aumentar significativamente a manutenibilidade de um sistema, o qual está se tornando rapidamente não manutenível.
- Necessidade de remover problemas complexos para permitir expansão ou integração para alguns sistemas específicos.

2.7 - Ferramentas de Auxílio à Reengenharia

Existem ferramentas com a finalidade de auxiliar a execução da reengenharia. A maioria das ferramentas são utilizadas na etapa de engenharia reversa do sistema a ser reconstruído.

Segundo Pressman [1995], as ferramentas baseadas em engenharia reversa estão ainda "engatinhando", ficando claro que as pesquisas na área de entendimento de código são muito importantes e muitas outras pesquisas ainda acontecerão.

Ferramentas devem ser adequadas aos processos de reengenharia, e não os processos adequados às ferramentas. As ferramentas de suporte disponíveis para auxiliar a reengenharia têm influência sobre os custos de reengenharia.

Existem muitas ferramentas de reengenharia com aplicabilidade em sistemas de software. O quadro 2.1 apresenta várias dessas ferramentas, mostrando o escopo das informações que cada uma utiliza (código fonte, base de conhecimento), bem como as saídas que cada ferramenta produz (visões produzidas e outras saídas).

2.8 - Considerações Finais

Apresentou-se neste capítulo, definições sobre o processo de reengenharia de software, suas divisões, propósitos e conceitos relacionados.

Quadro 2.1 - Escopo das informações utilizadas por cada ferramenta de reengenharia, e as respectivas visões e outras saídas produzidas

Ferra-Menta	Escopo das informações utilizadas		Visões produzidas				Outras Saídas produzidas	Refe-rência
	Código	Base de conhecimento a partir de	Implementacional	Estrutural	Funcional	Domínio		
Art	C		✓	✓				[Tonella *et al.*, 1996]
Decode	Cobol		✓	✓			Base de conhecimento sobre o sistema;	[Chin & Quilici, 1996]
Desire	C	Documentação, Biblioteca de componentes padrões de software, Informações informais	✓	✓	✓			[Biggerstaff, 1989]
Docket	Cobol e C	Documentação, Interação com usuário	✓	✓	✓	✓	Base de conhecimento sobre o sistema;	[Layzell *et al.*, 1995]
Macs	C	Repositório com conhecimentos do domínio de aplicação e implementação; Experiência do programador	✓	✓	✓			[Bennett, 1991]
Mandrake	Assembly		✓	✓				[Morris & Filman, 1996]
Newcomb	Cobol		✓	✓	✓			
Pat	C++		✓	✓			Padrões de código (clichês);	[Krämer & Prechelt, 1996]

Re^2	Genérico		✓	✓				[Canfora *et al.*, 1994]
Redo	Cobol e Fortran	Biblioteca de componentes padrões de software	✓	✓	✓			[Bennett, 1991]
Reform	Assembly	Base de conhecimento; Biblioteca de transformações						[Bennett, 1991]
Rigi	Cobol, C		✓	✓	✓			[Wong *et al.*, 1995]
Sneed	Cobol	Base de conhecimento do programador	✓	✓	✓		Classes e métodos (COBOL-OO);	[Sneed, 1996]

CAPÍTULO 3

TENDÊNCIAS EM REENGENHARIA DE SOFTWARE

3.1 - Motivação Para Transformar Sistemas Orientados a Função Para Sistemas Orientados a Objeto

Atualmente a nova fase de transição que a comunidade de informática está enfrentando para migrar software das plataformas centralizadas (*mainframes*) para ambientes de processamento distribuído em arquitetura cliente/servidor, mostra a importância da reengenharia e implica, muitas vezes, mover de uma arquitetura orientada a ação (paradigma procedimental) para uma arquitetura orientada a objeto (paradigma de orientação a objeto). [Sneed, 1992; Mittra, 1995].

Para Newcomb [1995], a programação orientada a objeto tem muitas vantagens sobre programação procedimental: possibilita construir sistemas altamente flexíveis, adaptáveis e extensíveis; possui uma coleção rica de mecanismos composicionais para formação de classes, instanciação de objetos, propriedades de herança, polimorfismo e ocultamento de informações, que estão presentes em linguagens de programação orientadas a objeto, as quais provêm apoio para criar sistemas que exibem um alto grau

de reuso e facilidade de manutenção.

Gall e Klosch [1995], citam vários motivos para migração do paradigma procedimental para o paradigma de orientação a objeto:

- Em programas procedimentais é muito difícil prever efeitos colaterais, pois freqüentemente os relacionamentos não são visíveis.
- Manutenções sucessivas requerem um conhecimento detalhado dos componentes do sistema, como foram criados e modificados e seus inter-relacionamentos;
- A reusabilidade de software também é seriamente afetada pela estrutura própria dos programas procedimentais;
- A orientação a objeto oferece algumas características úteis, tais como meios de abstração bem definidos, conceito de encapsulamento e comportamentos que efetivamente apóiam o processo de manutenção de software;
- A identificação de objetos nos programas procedimentais, exibindo explicitamente suas dependências, ajuda a entender o projeto do sistema, evita a degradação do projeto original durante as manutenções e facilita o processo de reuso.

3.2 - Tendências de Reengenharia nas Atividades das Empresas que Influenciam na Reengenharia de Software

A tendência de reengenharia dos processos das empresas são influenciados por fatores tais como: necessidade de melhoria da qualidade dos serviços e produtos oferecidos, compressão das margens de lucro, diminuição do ciclo de vida dos produtos, Diminuição da interferência dos governos, diminuição dos subsídios, a explosão tecnológica, fazendo que o conhecimento humano seja cada vez maior e mais acelerado, maturidade dos mercados de consumo, globalização da economia; o próprio complexo das atividades empresariais tais como: busca por produtividade, flexibilidade frente as constantes mudanças, concentração no ramo de negócio, relacionamentos com clientes, com meio ambiente, com governos, apoio de consultores, parceirização, gestão por resultados, remuneração dos recursos humanos por resultados. São todos fatores que influenciam diretamente a reengenharia dos softwares existentes em empresas [Coopers&Lybrand, 1998].

3.3 - Considerações Finais

Este capítulo procurou apresentar as principais tendências verificadas no campo de reengenharia de software. Estas tendências são do momento, e é lógico, que com o passar do tempo, novas mudanças sejam necessárias para atender as novas necessidades. Poderão ocorrer também novas mudanças tecnológicas, novas descobertas que propiciem alterações no campo da reengenharia de software.

BIBLIOGRAFIA

Referências

BENEDUSI, P.; CIMITILE, A.; CARLINI, U. *Reverse Engineering Processes, Design Document Production, and Structure Charts.* Journal Systems and Software, v.19, pp.225-45, 1992.

BENNETT, K. H. *Automated Support of Software Maintenance.* Information and Software Technology, v.33, n.1, pp.74-85, 1991.

BENNETT, K. H. *Legacy Systems: coping with success.* IEEE Software, v.12, n.1, pp.19-23, 1995.

BIGGERSTAFF, T. J. *Design Recovery for Maintenance and Reuse.* IEEE Computer, v.22, n.7, pp.36-49, 1989.

BOEHM, B. W. *A Spiral Model of Software Development and Enhancement.* Computer, v.21, n.5, pp.61-72, may, 1988.

CAINE, S.; GORDON, K. *PDL - A Tool for Software Design. In: Proc. National Computer Conference.* AFIPS Press, pp.271-276, 1975.

CANFORA, G.; CIMITILE, A.; MUNRO, M. *RE^2: Reverse-engineering and Reuse Re-engineering.* Journal of Software Maintenance: Research and Practice, v.6, n.2, pp.53-72, 1994.

CHIKOFSKY, E. J. e CROSS II, J. H. *Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software*, v.7, n.1, pp.13-17, 1990.

CHIN, D. N. e QUILICI, A. *DECODE: A Co-operative Program Understanding Environment.* Journal of Software Maintenance: Research and Practice, v.8, n.1, pp.3-33, 1996.

CHOI, S. C.; SCACCHI, W. *Extracting and Restructuring the Design of Large Systems. IEEE Software,* v.7, n.1, pp.66-71, 1990.

CLEVELAND, L. *A Program Understanding Support Environment.* IBM Systems Journal. v.28, n.2, pp.324-344, 1989.

COLEMAN, D. *et al. Object-Oriented Development: THE FUSION METHOD.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

COOPERS&LYBRAND. *Reengenharia & Process Management Excellence.* http://www.sisnema.com.br/news/eventos/apresentacoes/reengenharia/REENG12/sld001.htm. 10/07/1998.

COSTA, R. M. *FUSION-RE/I Um Método de Engenharia Reversa para Auxiliar a Manutenção de Software.* ICMSC-USP, São Carlos, 1997. Dissertação (mestrado).

CROSS, J. H. *Grasp/Ada Uses Control Structure.* IEEE Software, v.7, n.3, pp.62, 1990.

DeMARCO, T.; SARSON, C. *Structured Analysis and Systems Specification.* Prentice-Hall, 1979.

GALL, H. e KLOSCH, R. Finding Objects in Procedural Programs: *An alternative Approach.* In: Proceedings of Second Working Conference on Reverse Engineering, Monterey, EUA, pp.208-216, july, 1995.

GANNOD, G. C.; CHENG, B. H. C. *Using Informal and Formal Techniques for the Reverse Engineering of C Programs.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.249-258, nov, 1996.

GT-REG (Georgia Tech's – Reverse Engineering Group). *Glossary of Reengineering Terms.* Georgia, http://www.cc.gatech.edu/reverse/glossary.html, 1998.

HAMMER, M.; CHAMPY, J. *Reengenharia Revolucionando a Empresa.* Editora Campus, Rio de Janeiro, 1993

HARANDI, M. T.; NING, J. Q. *Knowledge-Based Program Analysis.* IEEE Software, v.7, n.1, pp.74-81, 1990.

HARBAND, J. *Seela Aids Maintenance with Code-Block Focus.* IEEE Software, v.7, n.3, pp.61, 1990.

HATLEY, D. J.; PIRBHAI, I. A. *Strategies for Real Time Systems Specifications.* Dorset House, 1987.

IEEE CS-TCSE (IEEE Computer Society - Technical Council on Software Engineering). *Reengineering & Reverse Engineering Terminology.* Washington, http://www.tcse.org/revengr/taxonomy.html, 1997.

JACOBSON, I.; LINDSTRÖM, F. *Re-engineering of old systems to an object-oriented architecture.* SIGPLAN Notices, v.26, n.11, pp.340-350,nov., 1991.

JONES, C. B. *Systematic Software Development Using VDM.* 2.ed., Prentice-Hall, 1990.

KRÄMER, C.; PRECHELT, L. In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.208-215, nov, 1996.

LAYZELL, P. J.; FREEMAN, M. J.; BENEDUSI, P. *Improving Reverse-engineering through the Use of Multiple Knowledge Sources.* Journal of Software Maintenance: Research and Practice, v.7, n.4, pp.279-299, 1995.

LEHMAN, M. M. *Programs, Life-Cycles, and the Laws of program Evolution.* Proc. *IEEE* pp.1060-1076, 1980.

LEWIS, B. e McCONNELL, D. J. *Reengineering Real-Time Embedded Software onto a Parallel Processing Platform.* In: Proceedings of Third Conference on Reverse Engineering, Monterey, EUA, pp.11-19, nov., 1996.

MACKENZIE, C. L.; HENSHAW, J. *Understanding the Legacy.* Software Engineering Quality Consultants, Ontario, april, 1994. /TR-74.137/

MITTRA, Sitansu S. *A Road Map for Migrating Legacy Systems to Client/Server.* Journal of Software Maintenance: Research and Practice, v.8, n.2, pp.117-130, 1995.

MORRIS, P.; FILMAN, R. *Mandrake: A Tool for Reverse-Engineering IBM Assembly Code.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.57-66, nov, 1996.

NEWCOMB, P. e KOTIK G. *Reengineering Procedural Into Object-Oriented Systems.* In: Proceedings of Second Conference on Reverse Engineering, Toronto, Canada, pp.237-249, jul., 1995.

OMAN, P.W. *Maintenance Tools.* IEEE Software, v.7, n.3, pp.59-65, 1990.

OXFORD. *Dictionary of Computing.* Oxford University Press, 1986.

PENTEADO, R. A. D. *Um Método para Engenharia Reversa Orientada a Objetos.*, IFSC-USP, São Carlos, 1996. Tese (Doutorado)

PREECE, J. *et al. Human-Computer Interaction.* Wokinghan, Addison-Wesley, 1994.

PREMERLANI, W. J. e BLAHA, M. R. *An Approach for Reverse Engineering of Relational Databases.* Communications of the ACM, v.37, n.5, pp.42-49, 1994.

PRESSMAN, R. S. *Engenharia de Software*, Makron Books, 1995.

RAMAMOORTHY, C. V.; TSAI, W. *Advances in Software Engineering.* IEEE Computer v.29, n.10, pp.47-58, 1996.

REKOFF Jr., M. G. *On Reverse Engineering.* IEEE Transaction on Systems, Man, and Cybernetics, v.15, n.2, mar/apr, 1985.

ROSS, R. G. *Entity Modeling: Techniques and Application.* Data Research Group, 1988.

ROYCE, W. W. *Managing the Development of Large Software Systems: Concepts and Techniques.* in Proc. WESCON, pp.1-9, 1970.

RUGABER, S. e WILLS, L. M. *Creating a Research Infrastructure for Reengineering.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.98-102, nov, 1996.

RUGABER, S. *et al.. recognizing Design Decision in Programs.* IEEE Software, v.7, n.1, pp.46-54, 1990.

SAGE, A.P. Systems Engineering and Systems Management for Reengineering. *Journal Systems and Software,* v.30, n.1, pp.03-25, 1995.

SAMUELSON, P. *Reverse-Engineering Someone Else's Software: Is It Legal?* IEEE Software, v.1, n.1, pp.90-96, 1990.

SNEED, H. M. *Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture.* In: Proceedings of Conference on Software Maintenance, Orlando, EUA, pp.105-116,nov., 1992.

SNEED, H. M. *Object-Oriented COBOL Recycling.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.169-178, nov, 1996.

SNEED, H. M. e NYÁRY, E. *Extracting Object-Oriented Specification from Procedurally Oriented Programs.* In: Proceedings of Second Conference on Reverse Engineering, Toronto, Canada, pp.217-226,jul., 1995.

SNEED, H. M. *Planning the Reengineering of Legacy Systems. IEEE Software,* v.12, n.1, pp.24-34, 1995.

SOMMERVILLE, I. *Software Engineering (International Computer Science Series).* 5ª Edição. Addison-Wesley Pub Co. 1995.

SPIVEY, J. M. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

STEPHEN, R. M.; LYNN, M. M. *Software Migration and Reengineering: A Pilot Project in Reengineering.* Journal Systems and Software, v.30, n.1, pp.137-50, 1995.

TANGORRA, F.; CHIAROLLA, D. *A methodology for reverse engineering hierarchical databases.* Information and Software Technology, v.37, n.4, pp.225-231, nov., 1995.

TONELLA, P. *et al. Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic - A Case Study.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.198-208, nov, 1996.

TURINE, M. A. S. *Especificação de Requisitos: Uma Introdução.* Relatório Técnico do ICMSC n º 39. ICMSC-USP. 1996.

ULRICH, W. M. Re-engineering: *Defining an integrated migration framework,* in Software Reengineering (R. S. Arnold, ed.), IEEEE Computer Society Press, Los Altos, California, pp. 108-118, 1993.

WARD, M. P.; BENNETT, K. H. *Formal Methods for Legacy Systems.* Journal of Software Maintenance: Research and Practice, v.7, n.3, pp.203-219, 1995.

WARDEN, R. *Re-engineering - A Practical Methodology With Commercial Applications.* in Applied Information Technology 12 (Software Reuse and Reverse Engineering in Practice). (P. A. V. Hall, ed.) - Chapman@Hall. 1992.

WATERS, R. C.; CHIKOFSKY, E. J. *Reverse Engineering: Progress Along Many Dimensions.* Communications of the ACM. v.37, n.5, pp.23-24, 1994.

WONG, K. *et al. Structural Redocumentation: A Case Study. IEEE Software,* v.12, n.1, pp.46-54, 1995.

YEH, A. S.; HARRIS, D. R. e REUBENSTEIN, H. B. *Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language.* In: Proceedings of Second Conference on Reverse Engineering, Toronto, Canada, pp.227-236, jul., 1995.

YOURDON, E. *Análise Estruturada Moderna.* Editora Campus, Rio de Janeiro, 1990.

Complementar

ALBRECHT, A. J. *Measuring Application Development Productivity.* In: Proc. IBM Application Development Symposium. Monterey, C. A., pp.83-92, 1979.

BENDIFALLAH, S.; SCACCHI, W. Understanding software Maintenance Work. *IEEE Transaction on Software Engineering,* v.13, n.3, pp.311-23, 1987

BOLDYREFF, C. *Reuse, Software Concepts, Descriptive Methods and the Practitioner Project.* ACM Sigsoft Software Engineering Notes, v.14, n.2, pp.25-31, april, 1989.

BURD, E.; MUNRO, M.; WEZEMAN C. *Extracting Reusable Modules from Legacy Code: Considering the Issues of Module Granularity.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.189-196, nov, 1996.

CORBI, T.A. Program Understanding: Challenge for the 1990s. *IBM Systems Journal.* v.28, n.2, 1989

CURTIS, B. *et al.* Measuring the Psychological Complexity of Software Maintenance Tasks with the Halsted and McCabe Metrics. *IEEE Conference on Software Engineering*, v.5, n.2, pp.96-104, 1979

DEKLEVA, S. Delphi Study of Software Maintenance Problems. In: *Proc. of 8th International Conference on Software Maintenance*, 1992

EDELSTEIN, D.V. Standard for Software Maintenance - Report on the IEEE STD 1219-1993, *Software Engineering Notes,* v.18, n.8, pp. 94-5, 1994

FORTE, G. Tools Fair: Out of the lab, onto the shelf - Reverse Engineering and Maintenance. *IEEE Software,* V.9, N.3, pp.76, 1992

GALLAGER, K.B.; LYLE, J.R. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering,* v.17, n.8, pp.751-61, 1991

HALL, P. A. V. *Overview of reverse engineering and reuse research.* Information and Software Technology, v.34, n.4, pp.239-249, april, 1992.

HOLZNER, Steven. *Programando em Visual C++.* Berkeley, 1995.

HTTP://www.microsoft.com/visualc/

LIENTZ, B.P.; SWANSON, E.B. Problems in Application Software Maintenance. *Communications of ACM,* v.24, n.11, pp.31-7, 1981

LIU, C.C. A Look at Software Maintenance. *Datamation*, nov., pp.51-5, 1976

LIVADAS, P. E.; JOHNSON, T. *A New Approach to Finding Objects in Programs,* Journal of Software Maintenance: Research and Practice, v.6, n.5, pp.249-260, 1994.

MALAN, R.; LETSINGER, R e COLEMAN, D. *Object-Oriented Development at Work - Fusion In the Real World.* Prentice Hall. 1996.

MAMONE, S. The IEEE Standard for Software Maintenance. *Software Engineering Notes*, v.19, n.1,1994

MAYER, R. C. *Visual Basic X Delphi: duelo de gigantes?.* Byte Brasil, pp.114, 1997.

McCABE, T. Jr.. *Battle-Map, ACT Show Code Structure, Complexity.* IEEE Software, v.7, n.3, pp.62, 1990.

MISAKA, A. P. S. *Modelagem Pelo Método Fusion de um Sistema de Agendamento de Reuniões.* Relatórios Técnicos - ICMSC-USP. 1996.

MOORE, M. M. *Rule-Based Detection for Reverse Engineering User Interfaces.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey-CA, pp.42-48, nov, 1996.

OGANDO, R. M. *et al. An object Finder for Program Structure Understanding in Software Maintenance.* Journal of Software Maintenance: Research and Practice, v.6, n.5, pp.261-283, 1994.

OSBORNE, W.M.; CHIKOFSKY, E.J. Fitting Pieces to the Maintenance Puzzle. *IEEE Software,* v.7, n.1, pp.10-1, 1990

PENTEADO, R. A. D.; GERMANO, F. S. R.; MASIERO, P. C. *An Overall Process Based on Fusion to Reverse Engineer Legacy Code.* In: Proceedings of Third Working Conference on Reverse Engineering, Monterey, EUA, pp.170-188, nov, 1996.

PENTEADO, R. A. D.; GERMANO, F. S. R.; MASIERO, P. C. *Engenharia Reversa Orientada a Objetos do Ambiente StatSim: método utilizado e resultados obtidos.* In: IX Simpósio Brasileiro de Engenharia de Software, Recife, pp.345-360, 1995.

PENTEADO, R. A. D.; GERMANO, F. S. R.; MASIERO, P. C. *Melhoria da Manutenibilidade de Software Legado com Base em Engenharia Reversa.* In: Anais da Conferência Internacional de Tecnologia de Software: Qualidade de Software, Curitiba-PR, pp.155-169, jun, 1996.

ROBSON, D. J. *et al. Approaches to Program Comprehension.* The Journal of Systems and Software, v.14, n.2, pp.79-84, 1991.

SANCHES, R., *A Influência do Software e de seu Processo de Manutenção no Esforço de Manutenção.* Universidade de São Paulo, Faculdade de Economia, Administração e Contabilidade, São Paulo, 1993. Tese (Doutorado)

SCHACH, S.R. *The Economic Impact of Reuse on Maintenance.* Journal of Software Maintenance: Research and Practice. v.6, n.4, pp.185-96, 1994.

SCHNEIDEWIND, N.F., The State of Software Maintenance. *IEEE Transaction on Software Engineering,* v.13, n.3, pp. 303-10, 1987.

SILVA, C. R. e SANTOS P. C. F. *Delphi X Visual Basic.* Byte Brasil, pp.24-28, 1997.

SNEED, H. M.; NYARY, E. *Downsizing Large Application Programs.* Journal of Software Maintenance: Research and Practice, v.6, n.5, pp.235-247, 1994.

SOMMERVILLE, I.; SAWYER, P. *Requirements Engineering : A Good Practice Guide.* John Wiley & Sons. 1997.

STAPLETON, L. *Reverse Engineering Puzzles Out The Past.* Open Computing, v.11, n.3, pp.77-82, 1994.

YANG, H. *The Supporting Environment for A Reverse Engineering System - The Maintainer's Assistant.* In: Proc. IEEE Conference on Software Maintenance, Sorrento, Italy, pp.13-22, nov., 1991.

YAU, S.S.; CHANG, P.S. Design Stability Measures for Software Maintenance. *IEEE Transaction on Software Engineering,* v.11, n.9, pp.849-56, 1985.

YOUNGER E. J.; WARD, M. P. *Inverse-Engineering a Simple Real-Time Program.* Journal of Software Maintenance: Research and Practice, v.6, n.4, pp.197-234, 1994.