

Reengenharia de software: o que, por quê e como

Ana Elisa Tozetto Piekarski e Marcos Antonio Quináia

Departamento de Informática - UNICENTRO
85010-990 Guarapuava, PR

(Recebido em 3 de maio de 2000)

Resumo: Este artigo tem como objetivo fornecer um embasamento sobre o processo de manutenção de software, mais especificamente sobre reengenharia e engenharia reversa, sendo destinado basicamente para apoio didático a essa parte da Engenharia de Software. Devido a isso, são apresentadas as definições - o “o que” são essas modalidades de manutenção, os casos em que se aplicam — o “por que” utilizá-las e a forma de adotá-las — o “como” realizá-las.

Palavras-chave: Manutenção de software, reengenharia de software, engenharia reversa

Abstract: This paper aims to provide a basement about software maintenance process, specially on reengineering and reverse engineering. It was developed regarding pedagogical use for Software Engineering classes. The concepts presented were organized to answer three basic points: “what” are the maintenance categories, “why” use them and “how” to implement them.

Key words: Software maintenance, software reengineering, reverse engineering.

1 Introdução

Segundo Osborne e Chikofsky (1990), a variedade de problemas que envolve manutenção de software cresce constantemente, sendo que as soluções não acompanham essa evolução. Esses problemas são resultantes de código fonte e documentação mal elaborados, além da falta de compreensão do sistema.

A partir do momento em que um sistema começa a ser utilizado, ele entra em um estado contínuo de mudança. Mesmo que tenha sido construído aplicando as melhores técnicas de projeto e codificação existentes, os sistemas vão se tornando obsoletos em vista das novas tecnologias que são disponibilizadas.

Além das correções de erros, as mudanças mais comuns que os sistemas sofrem são migrações para novas plataformas, ajustes para mudanças de tecnologia de hardware ou sistema operacional e extensões em sua funcionalidade para atender os usuários. Em geral, essas mudanças são realizadas sem que haja preocupação com a arquitetura geral do sistema, produzindo estruturas mal projetadas, documentação desatualizada, lógica e codificação ruins, sendo esses os focos que dificultam a manutenção em um sistema (OSBORNE e CHIKOFFSKY, 1990).

Quando o sistema não é fácil de ser mantido sendo, porém, de grande utilidade, ele deve ser reconstruído. Partindo-se do sistema existente (via código-fonte, interface ou ambiente), são abstraídas as suas funcionalidades e são construídos o modelo de análise e o projeto do software. Esse processo é denominado **reengenharia de software**.

Considerando-se que o material sobre essa área da engenharia de software é escasso, principalmente para fins didáticos, este artigo compila os principais artigos do tema em questão. Descreve-se o que é a reengenharia de software (O que), a sua importância (Por quê) e a forma de realizá-la (Como), além de situá-la no processo de manutenção de software. Trata-se, portanto, de uma revisão que tem como objetivo constituir um material de apoio ao ensino dos conteúdos da disciplina Engenharia de Software, mais especificamente Manutenção de Software.

2 O que é reengenharia de software

Para abordar adequadamente as técnicas de manutenção de software, deve-se primeiramente considerar três conceitos dependentes: a existência de um processo de desenvolvimento de software, a presença de um sistema a ser analisado e a identificação de níveis de abstração¹ (OSBORNE e CHIKOFFSKY, 1990).

Qualquer que seja o processo de desenvolvimento de software, espera-se que haja interação entre seus estágios e, talvez, recursão. Em um processo de desenvolvimento de software, os estágios iniciais envolvem conceitos mais gerais, independentes da implementação, enquanto os estágios finais enfatizam os detalhes de implementação. O aumento de detalhes durante o processo de desenvolvimento conceitua os **níveis de abstração**. Estágios iniciais do sistema planejam e definem requisitos de alto nível quando comparados com a própria implementação.

Essa comparação é importante para deixar claro que nível de abstração e grau de abstração são grandezas distintas. Enquanto o nível de abstração é um conceito que diferencia os estágios conceituais do projeto, o **grau de abstração** é intrínseco a cada estágio. A evolução através das fases do processo de desenvolvimento de software envolve transições dos níveis mais altos de abstração nos estágios iniciais, para níveis mais baixos nos estágios posteriores. As informações podem ser representadas em qualquer estágio do desenvolvimento, seja de forma detalhada (baixo grau de abstração), seja de forma mais sucinta ou global (alto grau de abstração).

¹Abstração 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.

Para que as técnicas de manutenção de software (especificamente engenharia reversa e reengenharia) sejam descritas de forma simplificada, serão tomadas como base apenas três fases do processo de desenvolvimento de software, com níveis de abstração bem diferenciados, conforme a Figura 1:

⇒ Requisitos: especificação do problema a ser resolvido, incluindo objetivos, restrições e regras de negociação;

⇒ Projeto: especificação da solução;

⇒ Implementação: codificação, teste e adaptação ao sistema operacional.

A técnica tradicional, que avança progressivamente pelas fases do processo de desenvolvimento de software, é denominada **engenharia progressiva**. A execução dessa técnica consiste em partir de projetos independentes da implementação, que possuem altos níveis de abstração, indo em direção à implementação física do sistema. Em outras palavras, engenharia progressiva segue a seqüência de desenvolvimento estabelecida no projeto, visando à obtenção do sistema implementado (OSBORNE e CHIKOFSKY, 1990).

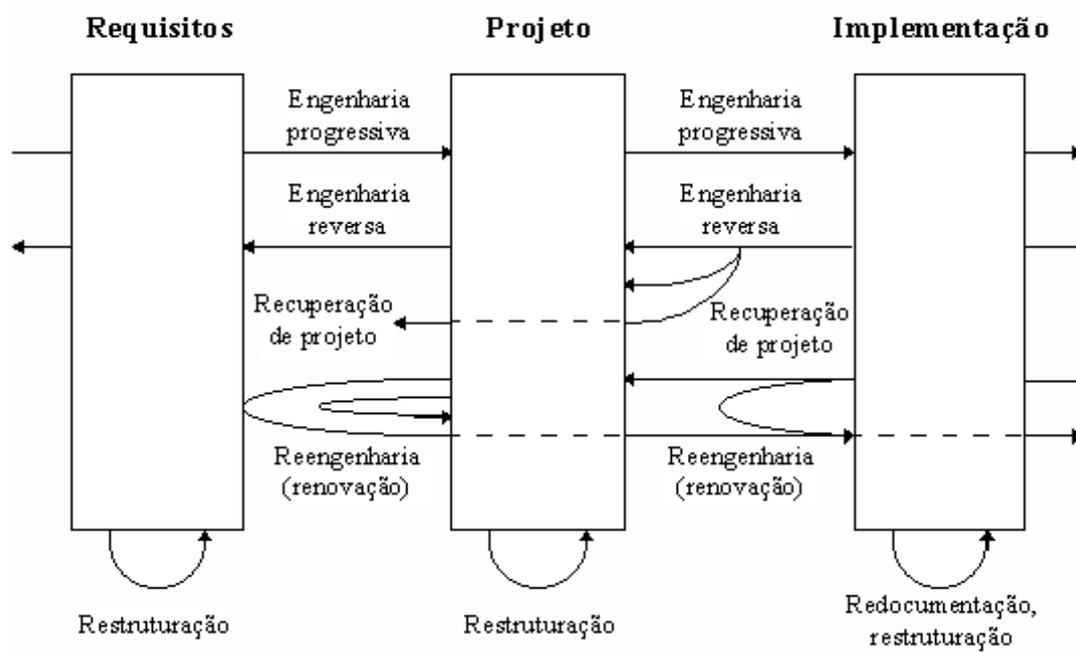


Figura 1: Relacionamentos no Ciclo de Desenvolvimento de Software (CHIKOFSKY e CROSS, 1990)

Na Figura 1, com exceção da engenharia progressiva, as demais transições entre as fases de desenvolvimento são tecnologias utilizadas na manutenção de software (CHIKOFSKY e CROSS, 1990), sendo assim definidas:

⇒ Redocumentação: como uma sub-área da engenharia reversa, 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 sub-área 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 é aplicada em sistemas que tenham sido desenvolvidos de forma desestruturada, resultando uma representação que preserva as características do sistema, porém de forma mais bem estruturada;

⇒ Engenharia reversa: é o processo de analisar um sistema com a finalidade de criar sua representação de uma forma diferente ou em um nível mais alto de abstração do que o código fonte. Essa tecnologia é descrita detalhadamente na seção 2.1;

⇒ Reengenharia: é a reconstrução de algo do mundo real, tendo como propósito a busca por melhorias que permitam produzir algo de qualidade melhor ou comparável ao produto inicial. A reengenharia é detalhada na seção 2.2.

No processo de manutenção, quando se trata de reconstruir um software (ou seja, realizar sua reengenharia), é necessário, portanto, que se proceda à engenharia reversa do sistema em questão, a fim de obter os modelos de análise baseados no software existente. Esses modelos, com as devidas correções/alterações, serão o ponto de partida para a engenharia progressiva.

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.

2.1 Engenharia reversa

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** (CHIKOFFSKY e CROSS, 1990).

O termo “engenharia reversa” originou-se da análise de hardware (CHIKOFFSKY e CROSS, 1990; REKOFF, 1985), que extraía o projeto a partir do produto final. Em geral, é aplicada para melhorar um produto ou para analisar um produto concorrente ou de um adversário (principalmente em situação militar ou de segurança nacional).

Aplicando o conceito inicial de engenharia reversa a sistemas de software, muitas das técnicas utilizadas em hardware servem para obter uma compreensão básica do sistema e sua estrutura. Entretanto, enquanto o objetivo básico para hardware é duplicar o sistema, os objetivos mais frequentes para software são obter uma compreensão suficiente em nível de projeto para auxiliar a manutenção, fortalecer o crescimento do sistema, e substituir o suporte.

2.1.1 Definições

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).

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.*, 1996; CHIKOFISKY e 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, constitui a engenharia reversa (WATERS e CHIKOFISKY, 1994).

2.1.2 Visões de software

A partir da engenharia reversa e com base nos diferentes níveis e graus de abstração, o software pode ser visualizado de diferentes maneiras (HARANDI e NING, 1990):

⇒ Visão em nível implementacional: abstrai características da linguagem de programação e características específicas 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;

⇒ 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);

⇒ Visão em nível de domínio: abstrai o contexto em que o sistema está operando, ou seja, o porquê do sistema a ser desenvolvido.

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 idiosincrasia da representação do código muito mais do que a representação original, que reflete a compreensão do problema pelo analista ou projetista.

A Figura 2 mostra a correspondência entre as categorias de visualização do software e as diferentes atividades do ciclo de desenvolvimento de software.

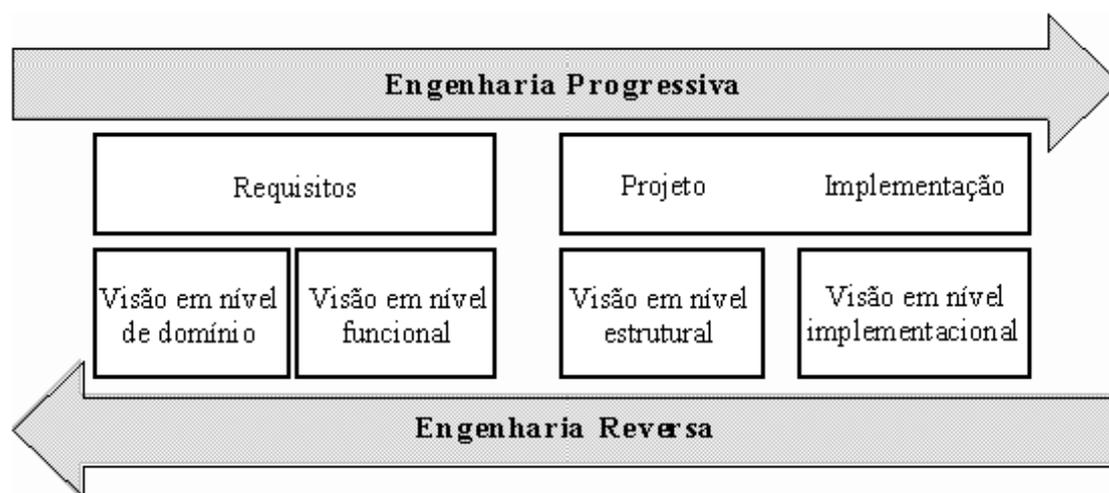


Figura 2: Visualizações de Software no Ciclo de Desenvolvimento (COSTA, 1997)

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.

2.1.3 Categorias

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** (CHIKOFFSKY e CROSS, 1990).

Visualização de código

Também denominada **redocumentação**, a visualização de código é a criação ou revisão de representações semanticamente equivalentes num mesmo nível de abstração (CHIKOFFSKY e 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 compreensão 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 [CHIKOFFSKY e CROSS, 1990].

Segundo Biggerstaff (1989), o **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, deve produzir todas as informações necessárias para se entender completamente o **que, como, e por que** o sistema faz.

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.

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

A Figura 3 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).

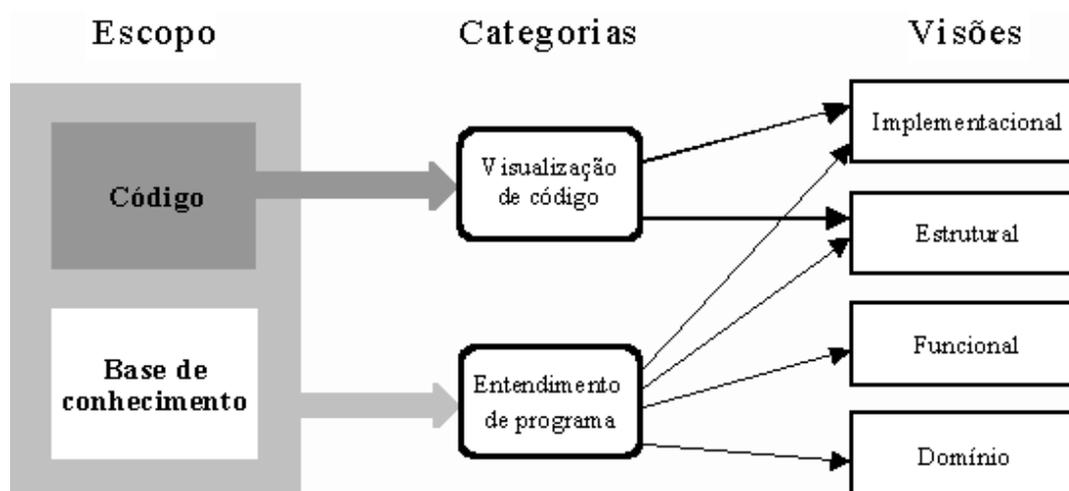


Figura 3: Categorias da engenharia reversa e suas visões

2.2 Reengenharia

O termo **reengenharia** está relacionado com a reconstrução de algo do mundo real, e independentemente de sua aplicação, 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.

2.2.1 Definições

Sendo a reengenharia de sistemas de software o foco deste artigo, a seguir são incluídas definições de diversos autores para que um nível de compreensão mais amplo seja obtido.

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 subsequente 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.

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.

Das diversas definições elencadas, 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.

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 com maior facilidade de manutenção e a engenharia reversa é usada como parte do processo de reengenharia, pois fornece o entendimento do sistema a ser reconstruído.

2.2.2 Categorias

Sage (1995) indica diversas categorias de melhorias relacionadas à reengenharia, entre elas:

⇒ Reengenharia de processos administrativos: é direcionada para alterações potenciais em todos os negócios ou processos organizacionais;

⇒ Reengenharia de processos produtivos: consiste em modificar qualquer ciclo de processos padrão, que esteja em uso em uma dada organização, a fim de melhor acomodar as tecnologias novas e emergentes, bem como os requisitos dos clientes para um produto ou sistema;

⇒ 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 ao 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).

3 Por que realizar a reengenharia

A tendência de reengenharia dos processos das empresas são influenciados por fatores tais como a necessidade de melhoria da qualidade dos serviços e produtos oferecidos, a compressão das margens de lucro, a redução do ciclo de vida dos produtos, a diminuição da interferência dos governos e dos subsídios, a explosão tecnológica, o rápido crescimento do conhecimento humano, a maturidade dos mercados de consumo e a globalização da economia.

Outros fatores são relacionados com a complexidade das atividades empresariais, tais como a busca por produtividade; a flexibilidade frente às constantes mudanças; a concentração no ramo de negócio; os relacionamentos com clientes, com o meio ambiente e com os governos; o apoio de consultores; a parcerização; a gestão e a remuneração dos recursos humanos por resultados. Todos esses pontos influenciam diretamente na reengenharia dos softwares existentes em empresas (HAMMER e CHAMPY, 1994).

Além disso, todos os sistemas têm um tempo de vida limitado, sendo que cada alteração efetuada 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 e LINDSTRÖM, 1991).

3.1 Reengenharia de software legado

Um 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 e BENNETT, 1995). A quantidade de código em sistemas legados é imensa. Em 1990, estimava-se (ULRICH, 1993) que existiam 120 bilhões de linhas de código fonte de sistemas desse tipo, a maioria em COBOL ou FORTRAN, linguagens com facilidades limitadas de estruturação de programas e 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 à limitação imposta pelo custo do hardware, tinha que ser reduzido, atualmente não é crucial (devido ao barateamento dos componentes de hardware). 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 embutidas 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. Conseqüentemente, 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 (sofrer 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 prever 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 e TSAI, 1996). Pesquisas revelam que

mais de 50% do custo de um produto de software está relacionado com as atividades de manutenção, havendo casos de este percentual chegar até a 85% (LAYZELL *et al.*, 1995).

Em alguns negócios, estima-se que 80% de todos os gastos com software são consumidos pelas manutenções e evoluções dos sistemas (YOURDON, 1990). O número de sistemas que precisa ser modificado está aumentando gradualmente. Existe fila de espera para pedidos de manutenção. Isto significa que, algumas vezes, é impossível para as organizações investirem em novos sistemas para melhorar a eficiência organizacional.

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 outros diferentes programas que, de alguma forma, compartilham dados. Também ocorre que esses programas foram desenvolvidos por diferentes pessoas ao longo dos anos, as quais não estão mais na organização. Muitas vezes foi usado um sistema de administração de bases de dados que pode estar obsoleto, ou depende 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. Essa 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 um software mais fácil de ser mantido (BENNETT, 1991).

Aplicando-se a reengenharia de software, 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 às 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 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 objetos, reusabilidade, etc. (RUGABER e WILLS, 1996; SNEED e NYÁRY, 1995). Também o desenvolvimento de novos projetos de software deverão se deparar, cada vez mais, com processos de reengenharia (COLEMAN *et al.*, 1994).

3.2 Migração do paradigma procedimental para o de orientação a objetos

Atualmente, a 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, o que implica, muitas vezes, o mover de uma arquitetura orientada a ação (paradigma procedimental) para uma arquitetura orientada a objetos (paradigma de orientação a objetos) (SNEED, 1992; MITTRA, 1995).

Para Newcomb e Kotik (1995), a programação orientada a objetos tem muitas vantagens sobre a programação procedimental, pois 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 objetos, 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 objetos:

⇒ 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 objetos 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.

4 Como realizar a reengenharia

O processo de reengenharia de software é constituído de duas fases distintas. Na primeira, o software objeto de reconstrução é “desmontado”, visando seu entendimento. Na segunda, o software é reconstruído, na forma desejada, a partir do produto da primeira fase, sendo incluídos os ajustes que se fizerem necessários.

O processo de Reengenharia pode ser traduzido como (JACOBSON e LINDSTRÖM, 1991):

$$\text{Reengenharia} = \text{Engenharia Reversa} + \Delta \text{Engenharia Progressiva}$$

onde Δ pode ser de dois tipos:

⇒ **Alterações parciais de funcionalidade** (Alteram parcialmente o objetivo principal do sistema): ocorrem devido a mudanças nos negócios, ou necessidade do usuário;

⇒ **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):

⇒ Deve ser executado somente se existir um argumento aceitável de custo/benefício;

⇒ Implica melhoria através de projeto;

⇒ Deve remover projetos ruins, mas reconhecer e manter projetos bons e simples, mesmo que eles sejam desestruturados;

⇒ A Engenharia Reversa é dirigida por tipos de problemas, os quais necessitam ser identificados;

⇒ Os 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.

4.1 O processo de reengenharia

Jacobson e Lindström (1991) declaram que, para executar um processo de reengenharia de um sistema, é necessário:

⇒ **Realizar a 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, que 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;

⇒ **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”);

⇒ **Reprojetar o sistema:** parte-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, devem-se considerar questões sobre a integração/comunicação entre as partes velhas e novas do sistema.

Gall e Klosch (1995) relatam um processo para se 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) descrevem uma abordagem para extrair automaticamente documentação de projeto, orientada a objetos 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 objetos sem alteração de funcionalidade. O processo de transformação em forma orientada a objetos localiza dados e procedimentos redundantes, duplicados e similares, e os abstrai em classes e métodos.

Segundo Ramamoorthy e Tsai (1996), os métodos de engenharia reversa existentes até o momento não recuperam de modo automático todas as visões do software. Isso 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 e CHIAROLLA, 1995).

4.2 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.

As 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 1 apresenta várias dessas ferramentas, mostrando de onde provêm as informações de entrada e o que cada uma delas produz como resultado.

Ferramenta	Escopo das informações utilizadas		Visões produzidas				Outras saídas produzidas	Referências bibliográficas
	Código	Base de conhecimento a partir de	Implementacional	Estrutural	Funcional	Domínio		
Art	C		✓	✓				[Tonella <i>et al.</i> , 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 <i>et al.</i> , 1995]
Macs	C	Repositório com conhecimentos do domínio de aplicação e implementação. Experiência do programador	✓	✓	✓			[Bennett, 1991]
Mandrake	Assembly		✓	✓				[Morris & Filman, 1996]
Newcomb	Cobol		✓	✓	✓			[Newcomb & Kotik, 1995]
Pat	C++		✓	✓			Padrões de código (clichês)	[Krämer & Prechelt, 1996]
Re ²	Genérico		✓	✓				[Canfora <i>et al.</i> , 1994]
Redo	Cobol e Fortran	Biblioteca de componentes padrões de software	✓	✓	✓			[Bennett, 1991]
Rigi	Cobol, C		✓	✓	✓			[Wong <i>et al.</i> , 1995]
Sneed	Cobol	Base de conhecimento do programador	✓	✓	✓		Classes e métodos (COBOL-OO)	[Sneed, 1996]

Quadro 1: Escopo das informações utilizadas por ferramentas de reengenharia, as respectivas visões e outras saídas produzidas.

O escopo das informações que cada ferramenta utiliza são provenientes:

⇒ do código fonte do software: cada ferramenta trabalha com código em determinada linguagem (pré-definida);

⇒ da base de conhecimento do sistema: constituída por documentação existente, informações de usuários ou projetistas do software e bibliotecas de componentes, entre outras.

Em relação aos resultados produzidos, quando apenas o código fonte é utilizado como entrada, as visões fornecidas são, em geral, em nível implementacional e estrutural. Para se obter visões mais abstratas (funcional e de domínio), bem como outras saídas, é necessário utilizar como entrada, além do código fonte, bases de conhecimento sobre o sistema.

5 Considerações finais

Este artigo não aborda exhaustivamente todo o conteúdo relacionado à reengenharia de software. Sendo destinado ao apoio pedagógico, o enfoque maior está nas definições – “o que” é a reengenharia e outros termos relacionados a manutenção de software – e na aplicabilidade dessa atividade – “por que” realizar a reengenharia de um software.

O “como”, ou seja, a forma de realizar a reengenharia em um software depende de muitos fatores, relacionados, por exemplo, com a forma de trabalho da empresa, a origem do software em questão e com o desenvolvimento de tecnologias para apoiar essa atividade. Algumas ferramentas disponíveis foram incluídas, com o intuito de fornecer uma visão geral de como executar a reengenharia de um software.

A importância da reengenharia está em possibilitar que todo conhecimento agregado ao software legado não seja perdido, o que aconteceria se a opção fosse pelo desenvolvimento de um novo software.

O processo de extração do conhecimento de um software legado é realizado pela engenharia reversa, que fornece visões mais abstratas do que o código do sistema e/ou alguma outra documentação possivelmente existente. Com essas visões é possível compreender o software e o sistema em que está inserido, possibilitando a produção de modelos de análise que servirão à reengenharia.

Da reengenharia de um software resulta uma nova versão, que agrega toda a informação do software legado, as inovações tecnológicas e novas funcionalidades desejadas. Dessa forma, a reengenharia de software não trata apenas de modernizá-lo, mas também adaptá-lo de acordo com as novas necessidades do processo em que está inserido.

Referências

- BENEDUSI, P.; CIMITILE, A.; CARLINI, U. Reverse Engineering Processes, Design Document Production, and Structure Charts. *Journal Systems and Software*, v. 19, 1992, p. 225-245.
- BENNETT, K. H. Automated Support of Software Maintenance. *Information and Software Technology*, v. 33, n.1, 1991, p. 74-85.
- BENNETT, K. H. Legacy Systems: coping with success. *IEEE Software*, v. 12, n. 1, 1995, p. 19-23.
- BIGGERSTAFF, T. J. Design Recovery for Maintenance and Reuse. *IEEE Computer*, v. 22, n. 7, 1989, p. 36-49.

- CANFORA, G.; CIMITILE, A.; MUNRO, M. RE2: Reverse-engineering and Reuse Re-engineering. *Journal of Software Maintenance: Research and Practice*, v. 6, n. 2, 1994, p. 53-72.
- CHIKOFFSKY, E. J. e CROSS II, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, v. 7, n. 1, 1990, p. 13-17.
- CHIN, D. N.; QUILICI, A. DECODE: A Co-operative Program Understanding Environment. *Journal of Software Maintenance: Research and Practice*, v. 8, n. 1, 1996, p. 3-33.
- COLEMAN, D.; ARNOLD, P.; BODOFF, S.; DOLLIN, C.; GILCHRIST, H.; HAYES, F. e JEREMAES, P. *Object-Oriented Development: THE FUSION METHOD*. Englewood Cliffs, New Jersey: Prentice Hall, 1994.
- COSTA, R. M. *Aplicação do Método de Engenharia Reversa FUSION-RE/I na Recuperação da Funcionalidade da Ferramenta de Teste PROTEUM*. São Carlos: ICMSC-USP, 1997. Dissertação (mestrado).
- GALL, H. e KLOSCH, R. *Finding Objects in Procedural Programs: An alternative Approach*. In: Proceedings of Second Working Conference on Reverse Engineering. Monterey, EUA: 1995, p. 208-216.
- 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 em função dos clientes, da concorrência e das grandes mudanças da gerência*. Rio de Janeiro: Campus, 1994.
- HARANDI, M. T.; NING, J. Q. Knowledge-Based Program Analysis. *IEEE Software* v. 7, n. 1, 1990, p. 74-81.
- 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, 1991, p. 340-350.
- KRÄMER, C.; PRECHELT, L. *Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software*. In: Proceedings of Third Working Conference on Reverse Engineering, Monterey, EUA, 1996, p. 208-215.
- 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, 1995, p. 279-299.
- LEHMAN, M. M. Programs, Life-Cycles, and the Laws of program Evolution. *Proc. IEEE*, 1980, p. 1060-1076.
- MITTRA, Sitansu S. A Road Map for Migrating Legacy Systems to Client/Server. *Journal of Software Maintenance: Research and Practice*, v. 8, n. 2, 1995, p. 117-130.

- MORRIS, P.; FILMAN, R. *Mandrake: A Tool for Reverse-Engineering IBM Assembly Code*. In: Proceedings of Third Working Conference on Reverse Engineering, Monterey, EUA, 1996, p. 57-66.
- NEWCOMB, P. e KOTIK G. *Reengineering Procedural Into Object-Oriented Systems*. In: Proceedings of Second Conference on Reverse Engineering, Toronto, Canada, 1995, p. 237-249.
- OMAN, P.W. Maintenance Tools. *IEEE Software*, v. 7, n. 3, 1990, p. 59-65.
- OSBORNE, W.M.; CHIKOFFSKY, E.J. Fitting Pieces to the Maintenance Puzzle. *IEEE Software*, v. 7, n. 1, 1990, p. 11-12.
- PREMERLANI, W. J. e BLAHA, M. R. An Approach for Reverse Engineering of Relational Databases. *Communications of the ACM*, v. 37, n. 5, 1994, p. 42-49.
- PRESSMAN, R. S. *Engenharia de Software*. São Paulo: Makron Books, 1995.
- RAMAMOORTHY, C. V.; TSAI, W. Advances in Software Engineering. *IEEE Computer*, v. 29, n. 10, 1996, p. 47-58.
- REKOFF Jr., M. G. On Reverse Engineering. *IEEE Transaction on Systems, Man, and Cybernetics*, v. 15, n. 2, março/abril, 1985.
- RUGABER, S. e WILLS, L. M. *Creating a Research Infrastructure for Reengineering*. In: Proceedings of Third Working Conference on Reverse Engineering, Monterey, EUA, 1996, p. 98-102.
- RUGABER, S. *et al.* Recognizing Design Decision in Programs. *IEEE Software*, v. 7, n. 1, 1990, p. 46-54.
- SAGE, A.P. Systems Engineering and Systems Management for Reengineering. *Journal Systems and Software*, v. 30, n. 1, 1995, p. 3-25.
- SAMUELSON, P. Reverse-Engineering Someone Else's Software: Is It Legal? *IEEE Software*, v.1, n. 1, 1990, p. 90-96.
- SNEED, H. M. e NYÁRY, E. *Extracting Object-Oriented Specification from Procedurally Oriented Programs*. In: Proceedings of Second Conference on Reverse Engineering. Toronto, Canada, 1995, p. 217-226.
- SNEED, H. M. *Object-Oriented COBOL Recycling*. In: Proceedings of Third Working Conference on Reverse Engineering. Monterey, EUA, 1996, p. 169-178.
- SNEED, H. M. *Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture*. In: Proceedings of Conference on Software Maintenance, Orlando, EUA, 1992, p. 105-116.
- SOMMERVILLE, I. *Software Engineering (International Computer Science Series)*. 5ª Edição. Reading: Addison-Wesley, 1995.
- TANGORRA, F.; CHIAROLLA, D. A methodology for reverse engineering hierarchical databases. *Information and Software Technology*, v. 37, n. 4, 1995, p. 225-231.

- 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, EUA, 1996, p. 198-208.
- ULRICH, W. M. *Re-engineering: Defining an integrated migration framework*. In: Software Reengineering (R. S. Arnold, ed.). Los Altos, California: IEEE Computer Society Press, 1993, p. 108-118.
- WARD, M. P.; BENNETT, K. H. Formal Methods for Legacy Systems. *Journal of Software Maintenance: Research and Practice*, v. 7, n. 3, 1995, p. 203-219.
- 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.; CHIKOFFSKY, E. J. Reverse Engineering: Progress Along Many Dimensions. *Communications of the ACM*, v. 37, n. 5, 1994, p. 23-24.
- WONG, K. *et al.* Structural Redocumentation: A Case Study. *IEEE Software*, v. 12, n. 1, 1995, p. 46-54.
- 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, 1995, p. 227-236.
- YOURDON, E. *Análise Estruturada Moderna*. Rio de Janeiro: Editora Campus, 1990.