UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

SUZETE JOSEIA GANDIN

Migração de Sistemas Legados

Dissertação apresentada como requisito parcial
para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Dr. Marcelo Soares Pimenta
Orientador

Prof. Dr. Roberto Tom Price
Co-orientador

Porto Alegre, maio de 2003.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Gandin, Suzete Joseia

Migração de Sistemas Legados / Suzete Joseia Gandin. – Porto Alegre: PPGC da UFRGS, 2003.
68 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós Graduação em Computação, Porto Alegre, BR - RS, 2003. Orientador: Marcelo Soares, Pimenta.

1. Sistemas Legados. 2. Engenharia reversa. 3. Reengenharia. 4.Perfis de Usuário. 5 Sites Institucionais. I. Pimenta, Marcelo Soares. II. Título.

SUMÁRIO

LISTA DE ABREVIATURAS

BNF	Backus Naur Form (Forma de Backus-Naur)
COBOL	Common Business Oriented Language
DAST	Draco Abstract Sintax Tree (Árvore Sintática Abstrata do Draco)
DER	Diagrama Entidade Relacionamento
EXL	Extraction Language (Linguagem de Extração)
KB	Knowledge Base (Base de Conhecimento)
LA	Linguagem de atributos
ML	Model Language (Linguagem de Modelo)
PCYACC	Portable Compiler Language ToolKit
PUC-RJ	Pontifícia Universidade Católica do Rio de Janeiro
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

RESUMO

Mesmo depois de todas as novidades tecnológicas nos últimos anos, ainda existem muitos sistemas desenvolvidos com tecnologias antigas, muitas vezes ultrapassadas e obsoletas denominados **sistemas legados**. O problema do bug do ano 2000 funcionou como um excelente despertador para o fato de que não podemos nos esquecer do grande número de sistemas ainda em produção, e que são importantes para a empresa. Não se pode simplesmente descartar estes sistemas e é muito difícil migrar sistemas legados rapidamente para novas plataformas. Mais ainda, as regras de negócio que regem qualquer empresa são muito complexas para poderem ser modeladas e remodeladas em poucos meses e em seguida automatizadas porque a maior dificuldade em desenvolver sistemas não é escrever código nesta ou naquela linguagem, mas entender o que o sistema deve fazer.

Este trabalho enfoca uma solução possível para o problema referente à migração de **sistemas legados**: a tradução destes **sistemas legados** da forma mais automatizada possível para que possam se beneficiar das novas tecnologias existentes deve ser o resultado final produzido.

Assim, o objetivo desta dissertação é a investigação do problema de migração de **sistemas legados** e suas soluções assim como o desenvolvimento de uma ferramenta que traduz um **sistema legado** escrito na linguagem COBOL para PROGRESS, visando o aproveitamento do código e principalmente o aproveitamento de soluções de análise e projeto, que exigiram bastante esforço para serem elaboradas e poderiam ser reutilizadas em novos desenvolvimentos.

Palavras-chave:
Reengenharia, Engenharia Reversa, Sistemas Legados

TITLE: “MIGRATION OF LEGACY SYSTEMS”

ABSTRACT

After all technological new features in the last years, there are still many systems developed with old technologies called **legacy systems**. The Y2K problem was an excellent alarm-clock for the following fact: we cannot forget a great number of systems actually in production, and they are very important for the company. We cannot simply discard these systems but it is very difficult to migrate **legacy systems** quickly for new platforms. They implicitely contain company business rules that are very complex to be shaped and remodelled in few months: the biggest difficulty in developing systems is not to write code in one or another language, but to understand what the system must do. This work focuses on a possible solution for the migration of **legacy systems**: the more automatically way as possible translation of these systems so that they can be benefited of some new existing technologies. Thus, the main objective of this dissertation is the investigation of the aspects of migration of **legacy systems** and some solutions as well as the development of a tool that translates a system from Cobol to Progress environment, aiming both code reuse and exploitation of analysis and design solutions.

Keywords:
Rengineering, Reverse Engineering, legacy systems

1 INTRODUÇÃO GERAL

A expressão, sistemas legados (tradução da expressão inglesa "*legacy systems*") , foi criada como se este nome empacotasse uma tecnologia ultrapassada e fixa.

Tipicamente, chamamos de "legados" os sistemas desenvolvidos em plataformas e linguagens obsoletas. Neste trabalho em particular, nos concentramos em sistemas desenvolvidos na linguagem COBOL em plataforma mainframe.

Acontece que, quando um novo sistema termina de ser desenvolvido e entra em produção, ele imediatamente se transforma em "legado" pois passa a necessitar de manutenção e a tecnologia na qual foi desenvolvido não é mais novidade.

Mesmo depois de todas novidades tecnológicas nos últimos anos, ainda existem muitos sistemas legados Não se pode simplesmente descartar estes sistemas: nenhuma empresa terá condições de migrar todos os seus sistemas rapidamente para novas plataformas. Além disso, com a velocidade em que as novidades aparecem, qualquer projeto de migração precisa ser continuamente revisto.

Mais ainda, as regras de negócio que regem qualquer empresa são muito complexas para poderem ser modeladas e remodeladas em poucos meses e em seguida automatizadas. Não importa a novidade tecnológica, sempre será impossível reimplantar os sistemas em poucos meses. Isto porque a maior dificuldade em desenvolver sistemas não é escrever código nesta ou naquela linguagem, mas entender o quê o sistema deve fazer.

Uma solução possível a este problema é a conversão destes sistemas legados - da forma mais automatizada possível - para que possam se beneficiar das novas tecnologias existentes.

A autora tem nos últimos anos trabalhado no desenvolvimento de sistemas em Cobol e mais recentemente em Progress. Em suas atividades, a migração de sistemas de um ambiente ou linguagem para outro tem sido um desafio constante e constitui a maior motivação para a realização desta pesquisa, de forma que os resultados serão necessariamente colocados em prática. Além desta motivação pessoal, acredita-se que este tipo de problema é bastante comum no mercado de informática nacional (embora infelizmente não se publiquem estatísticas no Brasil a respeito) aliado à falta de suporte metodológico e tecnológico para solucioná-lo. Desta forma, acreditamos que é um problema extremamente relevante e atual.

O objetivo desta dissertação é a investigação do problema de migração de sistemas legados e suas soluções assim como o desenvolvimento de uma ferramenta, que faça a conversão de sistemas legados desenvolvidos em COBOL para sistemas correspondentes em PROGRESS. Este processo de migração visa o aproveitamento do código, da modelagem da base de dados e principalmente de soluções de análise e

projeto, que exigiram bastante esforço para serem elaboradas e poderiam ser reutilizadas em novos desenvolvimentos.

1.1 Organização do Trabalho

Este trabalho está organizado da seguinte forma: o Capítulo 2 faz uma revisão da literatura especializada. São abordadas algumas definições e fundamentos de engenharia reversa, reengenharia de sistemas e também conceitos sobre sistemas legados, assim como uma revisão dos trabalhos, abordagens e ferramentas relacionadas.

O Capítulo 3 apresenta a proposta de solução para migração de sistemas legados, mais particularmente descrevendo a solução específica para migração de sistemas em COBOL para PROGRESS e as técnicas e ferramentas utilizadas neste processo. Além disto, um estudo de caso real, envolvendo a migração de Cobol para Progress de um sistema existente para suporte a vendas, compras e controle de estoque de peças no contexto de concessionárias e revendedoras autorizadas. Os exemplos descritos no capítulo para ilustrar e ajudar a explicar a solução são retirados deste estudo de caso.

Finalmente no capítulo 4 – Conclusões, resumem-se as contribuições, limitações e perspectivas futuras deste trabalho.

Algumas informações adicionais estão presentes nos anexos, a saber:

- O anexo 1 apresenta um exemplo de tela de cadastro de peças do sistema legado em Cobol;
- O anexo 2 apresenta a tela correspondente de cadastro de peças do sistema em Progress após a realização do processo de migração;
- O anexo 3 apresenta um exemplo de definição da base de dados em COBOL. (estoque de peças)
- O anexo 4 apresenta um exemplo de definição da base de dados em PROGRESS. (estoque de peças) – Após aplicação da engenharia reversa
- O anexo 5 apresenta o código fonte antes da reestruturação
- O anexo 6 apresenta o código fonte após reestruturação´.

2 SISTEMAS LEGADOS, ENGENHARIA REVERSA E RENGENHARIA: UMA REVISÃO BIBLIOGRÁFICA

São abordados neste capítulo os conceitos da literatura especializada que serão utilizados durante o desenvolvimento deste trabalho. Na Seção 2.1 resumem-se os conceitos e caracterísitcas de sistemas legados. Na seção 2.2 apresentamos definições e fundamentos de Engenharia Reveersa, destacando na seção 2.2.1 a necessidade de reuso de código e de ferramentas em Engenharia Reversa. A seção 2.3 resume os fundamentos da atividade de entendimento de programas, essencial para Engenharia Reversa, assim como alguns trabalhos relacionados a este tema. A seção 2.4 apresenta conceitos, definições e trabalhos relacionados a Reengenharia. A seção 2.5 apresenta algumas abordagens existentes na literatura para Engenharia Reversa, Reengenharia e Migração de sistemas. Finalmente, a seção 2.6 destaca algumas das idéias descritas e apresentadas neste capítulo de revisão bibliográfica que são consideradas relevantes na compreensão da proposta de migração apresentada e explicada no capítulo seguinte. Sistemas Legados: conceitos e características

Penteado, em (PENTEADO;GERMANO;MASIERO, 1995), declara que os sistemas legados são aqueles sistemas que estão em uso por muito tempo, que atendem aos requisitos dos usuários e são de difícil substituição ou porque a reimplementação de seu código é inviável financeiramente ou porque eles são imprescindíveis, ou seja, esses sistemas não podem ficar inativos por muito tempo. Na maioria das vezes, os sistemas legados foram desenvolvidos em linguagens procedurais, não implementam abstração de dados e não possuem documentação, exceto o código fonte. Tudo isso dificulta a adição de novas funcionalidades e a realização de manutenção no sistema.

Sistemas Legados (Legacy Systems), são sistemas em operação de longa data, e que são essenciais ao cumprimento da missão da organização. Além destas duas propriedades básicas, os sistemas legados são adicionalmente caracterizados por :

a) Serem sistemas de informação muito grandes, da ordem de milhões de linhas de código, tipicamente escritas em COBOL;

b) Usarem Sistemas de Gerência de Bancos de Dados do modelo hierárquico, quando usam SGBDs como apoio aos serviços de dados;

c) Estarem operacionais permanentemente.

Sistemas legados são encontrados em um número expressivo de organizações de médio e grande porte, estabelecidas há no mínimo uma década, e que têm suas atividades apoiadas por um certo nível de informatização operacional. Estas organizações pertencem a um dos diversos setores de nossa sociedade, tais como comércio, indústria, e serviços, tanto em nível de iniciativa privada como de administração pública, e em âmbito regional, nacional e internacional.

Os sistemas legados constituem-se em causadores de sérios problemas às organizações, pois, sendo essenciais às atividades das mesmas, devem estar em manutenção permanente, implicando altíssimos custos. Ademais, os sistemas legados não são facilmente adaptáveis aos novos requisitos das organizações, pois foram concebidos para uma realidade tecnológica que foi radicalmente tornada obsoleta face à fantástica evolução que a informática teve nos últimos anos. Espera-se que as plataformas tecnológicas e paradigmas de concepção de sistemas mais atuais possam contribuir, em muito, na flexibilidade para manutenção destes sistemas, a custos significativamente menores. Contudo, este processo de evolução é complexo, e enfrenta várias dificuldades, tais como:

a) dificilmente as empresas têm recursos financeiros e humanos para conduzir um processo de desenvolvimento dos seus sistemas de informação de acordo com a tecnologia mais avançada, e manter paralelamente os sistemas legados operacionais durante este processo ;

b) Há uma carência de métodos e ferramentas para fundamentar a escolha de tecnologias adequadas para a evolução de sistemas legados, considerando suas propriedades específicas, bem como para conduzir o processo de evolução propriamente dito de maneira sistemática e com custo eficiente;

c) Existe um desconhecimento do real impacto de novas tecnologias nos sistemas de informações resultantes da evolução dos sistemas legados para novas plataformas, em particular face a novos requisitos, e futuras inovações tecnológicas;

d) A parcial inadequação do estado atual da tecnologia empregada comercialmente, que ainda não contempla com soluções específicas a descrição e manipulação de informações avançadas ou ditas "não convencionais", tais como dados temporais, dados geográficos, dados multimeio, dados textuais, etc;

e) Freqüentemente há um desconhecimento sobre a integridade das funcionalidades oferecidas pelos sistemas legados por parte da própria corporação, devido ao conhecido problema de manter-se a documentação de um sistema de informação consistente e atualizada com a sua respectiva implementação, e à rotatividade dos membros da equipe de desenvolvimento e manutenção.

A evolução dos sistemas legados precisa levar em conta adicionalmente a adaptação dos dados pré-existentes às novas modelagens da aplicação, quer na forma de co-existência de diferentes modelagens a serem gerenciadas pelas aplicações, quer na conversão dos dados existentes para novas modelagens. Um fator complicante em sistemas legados é a baixa confiabilidade que a informação armazenada passa a apresentar, com o correr do tempo, após seguidas transformações. Com efeito, a informação na base de dados pode passar a ser incompleta, redundante, conflitante ou incerta, o que dificulta a manutenção de restrições de integridade dos dados, a compreensão dos dados para novas adaptações, e obviamente futuras evoluções. Mais grave ainda, as vezes existe mesmo um desconhecimento por parte da própria equipe de manutenção do significado dos dados, das restrições que deve atender, e do seu papel para atender as funcionalidades oferecidas pelos sistemas legados, como já destacado anteriormente.

Assim sendo, é parte integrante e fundamental do processo de evolução de sistemas legados uma análise cuidadosa dos dados para mensurar a qualidade, confiabilidade e mesmo o significado da informação.

Técnicas de mineração de dados são uma oportunidade de apoio a esta análise. Sistemas de mineração de dados, também chamados de sistemas de descoberta de

conhecimentos, têm por objetivo descobrir conhecimento implicitamente armazenado em bases com grandes volumes de dados, através da extração de padrões significativos apresentados por estes dados. Entre os padrões que podem ser detectados estão a dependência entre dados, a identificação e descrição de conceitos, e a detecção de comportamentos anômalos. É importante destacar que a utilidade da aplicação de técnicas de mineração vai além da análise da integridade da base de dados. A existência de informações históricas, em particular se relativas a uma porção significatica do tempo de vida de um sistema legado, constitui uma verdadeira mina de informações explícita ou implicitamente armazenadas, cujo valor pode revelar-se incalculável no apoio a tomadas de decisão de uma organização.

Portanto, a migração de sistemas legados para plataformas mais atuais requer um estudo cuidadoso da real contribuição que as novas tecnologias têm a oferecer para suporte à evolução para a solução final, e métodos e ferramentas voltados a assitir esta evolução.

2.1 Engenharia Reversa : Definições e Fundamentos

Em muitos sistemas de informação pode-se aproveitar a estrutura geral de um programa existente: por exemplo, os módulos para montagem de menus, manutenção, movimentação, relatórios, consultas e a estrutura de inserção, alteração e eliminação de dados. Esse reuso pode ser maior ou menor de acordo com o tipo de aplicação a ser desenvolvida e de acordo com os sistemas prontos disponíveis.

Ao longo de alguns anos de trabalho pode-se acumular sistemas completos com inúmeros módulos, que podem servir como base para elaboração de novos sistemas. Desenvolver sistemas dessa forma tem seus problemas: na ânsia de aproveitar algo já pronto, muitas vezes o sistema resultante não fica tão eficiente; existe a tendência de colar "remendo sobre remendo", produzindo sistemas difíceis de manter; as alterações feitas são tantas que não resta nada do sistema original (nesse caso, talvez tivesse sido melhor partir "do zero"); é também difícil saber qual dos sistemas prontos seria a melhor base para o novo sistema, devido à falta de rigor na documentação.

De fato, descartar o código antigo é a abordagem usual para conversão de sistemas quando se cogita de ampliar a funcionalidade de um sistema existente, com idade avançada e desatualizado.

O termo "*engenharia reversa*" tem sua origem em análise de hardware, na qual a prática de extrair projetos do produto final é trivial. A Engenharia Reversa é regularmente aplicada para melhorar o produto de partir da análise dos produtos de competidores. Por definição, "Engenharia Reversa é o processo de desenvolvimento de um conjunto de especificações para um sistema complexo por exame de espécimes daquele sistema, de forma ordenada.". Esse processo é, em geral, conduzido por outro desenvolvedor, sem o benefício de algum dos desenhos originais, com o propósito de fazer um *clone* do sistema de hardware original.

Em relação ao software, a engenharia reversa é o processo de análise de um sistema para identificar seus componentes e inter-relacionamentos e criar representações do mesmo em outra forma ou num nível mais alto de abstração. As informações extraídas do código fonte via engenharia reversa podem estar em diversos níveis de abstração. Por exemplo, num baixo nível de abstração têm-se representações de projeto procedimental, subindo para informações sobre a estrutura de dados e de programa, modelos de fluxo de controle e de dados e chegando a modelos entidade-relacionamento, que

constituiriam o nível de abstração mais alto em nível de dados. O ideal seria ter um nível de abstração o mais alto possível (PRESSMAN, 1995).

Existem diversas técnicas para Engenharia Reversa, tais como: diagrama de fluxo de dados, diagrama de entidades e relacionamentos ou de modelo de objetos do sistema, na maioria das vezes, a partir de inspeção do código fonte.

Como existe um certo conflito com relação à terminologia usada na abordagem de engenharia reversa se propõem a definição de alguns termos:

- *engenharia avante (forward enginering):* é o processo tradicional de partir de um nível de abstração alto, de requisitos do sistema, e chegar ao nível físico, de implementação do sistema.
- *engenharia reversa (reverse enginering):* processo de análise de um sistema existente, identifica seus componentes e os representa em um nível mais alto de abstração. Pode ser classificada como: *redocumentação* (criação ou revisão de uma representação da abstração semântica do sistema) e *recuperação do projeto* (adição de domínio do conhecimento e informações externas para identificar no sistema abstrações de alto nível, além daquelas obtidas diretamente pela análise do sistema); será melhor definida no capítulo 3.
- *reestruturação:* . Reestruturação é a transformação de uma forma de representação para outra no mesmo nível relativo de abstração, preservando o comportamento externo do sistema (funcionalidade e semântica). Como exemplos de reestruturação pode-se citar estruturação de programas e normalização de dados.
- *reengenharia:* consiste na análise e alteração do sistema, para reconstruí-lo em uma nova forma**.**

De acordo com Penteado (PENTEADO, 1996), a engenharia reversa é capaz de abstrair informações a partir do código existente e do conhecimento e experiência dos usuários, produzindo documentos consistentes com o código fonte que podem melhorar o desenvolvimento subseqüente, facilitar a manutenção e a reengenharia. Entretanto, reprojetar e documentar um sistema já existente é tarefa bem mais difícil do que realizar um projeto inicial. O processo de engenharia reversa é uma atividade de análise do sistema e não implica em sua mudança ou na criação de um novo sistema.

Os resultados das fases de levantamento de requisitos, quando se realiza a engenharia avante e engenharia reversa, são similares, isto é, ambos modelam os requisitos do sistema. Entretanto, a maneira de coletar as informações é diferente. No processo de engenharia reversa, em vez de estudar e observar requisitos de informações e o funcionamento das atividades do negócio, o analista estuda e observa a base de dados e o código fonte já existente.

A manutenção do software existente pode ser responsável por mais de 70% de todo o esforço dispendido por uma organização de software, conforme Pressman, (1995). Após um sistema ter sido liberado para uso, algumas atividades de manutenção podem ser definidas como segue:

- Manutenção Corretiva: atividade que inclui diagnóstico e correção de erros. Ocorre porque não é possível presumir a descoberta de todos os erros de um sistema através das atividades de testes de software;
- Manutenção Adaptativa: atividade que modifica o software para que ele tenha uma interface adequada com o ambiente em que é executado. Ocorre por causa da rápida mudança e evolução das plataformas de computação (hardware, sistemas operacionais, etc);

- Manutenção Perfectiva: atividade responsável pela maior parte de todo o esforço dispendido em manutenção de software. Ocorre à medida que o software é usado mediante solicitações de novas funcionalidades do sistema pelos usuários;
- Manutenção Preventiva: ocorre quando o software é modificado para melhorar a confiabilidade e manutenibilidade futura. Essa atividade é caracterizada pelos processos de engenharia reversa e reengenharia.

A fase de manutenção é a mais temida do ciclo de vida do software, pois é necessário entender e alterar o código fonte do sistema, na maioria das vezes, desenvolvido por pessoas com diferentes estilos de programação. Além disso, a maioria dos sistemas possui documentação desatualizada ou nenhuma documentação, dificultando ainda mais essa tarefa. Assim, o processo de engenharia reversa surgiu para recuperar o modelo de análise do sistema e amenizar os esforços empregados na fase de manutenção.

2.1.1 A necessidade de reuso de código e de ferramentas em engenharia reversa

O principal objetivo do estudo de engenharia reversa é o de se poder compreender artefatos de software já desenvolvidos. A motivação pode ser a manutenção do software, a migração, a reengenharia, ou qualquer outra tarefa que envolva conhecimento a respeito do comportamento do software, conforme apresentado por Leite (LEITE, 1996). Esses sistemas recebem o nome de sistemas legados. Assim a engenharia reversa é a forma pela qual, a partir do código existente, pode-se elaborar o modelo de análise do sistema legado para então serem feitas as modificações desejadas.

A reengenharia de sistemas pode ou não incluir a alteração da linguagem de implementação, a mudança de paradigma de desenvolvimento e a mudança parcial ou total da funcionalidade do sistema. Quando após a engenharia reversa decide-se pela total mudança da funcionalidade do sistema deve-se então utilizar a engenharia avante, que é a forma natural de desenvolvimento de sistemas.

O objetivo principal da engenharia reversa, a compreensão do software, só se dá completamente quando o ser humano é capaz de explicar o comportamento do programa, sua relação com o domínio de aplicação no qual este está inserido, e o seu efeito no ambiente da aplicação (BIGGERSTAFF;MITBANDER;WEBSTER, 1994). Para atingir tal compreensão em grandes sistemas de software é imprescindível que se utilize ferramentas que auxiliem a aquisição deste conhecimento.

Existem muitos tipos de informações a serem extraídas, em diferentes tipos de situações, dependendo principalmente da disponibilidade e da precisão de informações sobre o software. Para realizar tal tarefa são necessárias uma variedade de abordagens e habilidades, Wills, (1996). Portanto uma boa ferramenta de engenharia reversa não pode se limitar a um determinado tipo de informação.

Os principais requisitos de uma boa ferramenta de engenharia reversa seriam portanto:

- Ferramentas de engenharia reversa devem permitir seus usuários analisarem cada dimensão do software em separado mas também cruzando suas informações (SELFRIDGE; WALTERS; CHIKOFSKY, 1993).
- Deve ser possível coletar e armazenar informações sobre qualquer aspecto do software que se deseje.
- Deve-se possibilitar filtro das informações a serem analisadas a qualquer hora e em qualquer ponto.
- Deve-se ainda possibilitar a subdivisão do sistema de várias maneiras distintas.
- Por último deve-se facilitar a visualização dos resultados pelo usuário.

Para a realização deste processo, é fundamental a atividade de entendimento de programas, que será abordada na seção 2.3.

2.2 Entendimento de Programas para Engenharia Reversa: fundamentos

Entender um programa significa entender sua estrutura, seu comportamento, seus efeitos e seus relacionamentos com o domínio de aplicação, conforme Biggerstaff; mitbander; Webster, (1994). A compreensão ou entendimento de programas é a construção de uma estrutura semântica em diversos níveis de abstração, para representá-los. Para isso, o programador pode contar com o auxílio de seu conhecimento sintático da linguagem de programação e com seu conhecimento sobre o domínio da aplicação .

O nível de abstração em que um programa é entendido pode ser alto (por exemplo, a descoberta do quê o programa faz) ou baixo (por exemplo, o reconhecimento de seqüências familiares de comandos). Ressalta-se que é possível ter uma compreensão de alto nível mesmo sem ter-se a compreensão de baixo nível, e vice-versa .

A maneira mais elementar de obter conhecimento sobre um sistema é por meio da leitura do código fonte, o que pode ser efetivo mas muito difícil devido à grande quantidade de informação contida no código e à dificuldade de extrair o conhecimento necessário . Assim, é necessário um instrumental para apoiar o entendimento de programas, que compreende a criação de modelos, a construção de ferramentas para ajudar o entendimento e a realização de estudos empíricos sobre entendimento de programas.

Modelos cognitivos são criados para ajudar a entender como funciona o processo cognitivo, ou seja, o processo de aquisição de conhecimento. O conhecimento do "entendedor" de programas é um elemento de grande importância em um modelo de cognição, podendo ser geral (por exemplo, conhecimento sobre o domínio de aplicação ou sobre um algoritmo) ou específico (por exemplo, conhecimento sobre a linguagem de programação ou estrutura de dados). Além disso, o nível de experiência do especialista (no caso o programador ou analista de sistemas), a forma de organização do conhecimento (representação mental do entendimento) e a eficiência na decomposição do problema (direção *"top-down"* ou *"bottom-up"*) são fatores que influenciam diretamente no entendimento de programas.

Ferramentas para apoio ao entendimento de programas têm sido construídas, compreendendo desde "*parsers*" de programação, que fazem, por exemplo, a análise sintática do programa, produzindo informações sobre a hierarquia de chamadas, até ferramentas mais elaboradas, que se utilizam de bases de conhecimento e inteligência artificial para inferir possíveis resultados.

Estudos empíricos são uma forma de obter dados sobre o processo de cognição e validá-lo. Esses estudos podem ser conduzidos de três formas diferentes: observação, correlação e teste de hipóteses. A primeira observa o comportamento da maneira como ocorre no mundo real, por exemplo o comportamento de programadores diante de situações específicas. A segunda assume que uma teoria tenha sido construída a partir dos estudos observacionais e tenta explorar o relacionamento entre as variáveis em questão. A terceira investiga causas e efeitos entre variáveis, a fim de validar uma teoria existente. Os estudos empíricos levam em consideração, também, o tamanho do código fonte, linguagem de programação e o tipo de programador. Por exemplo, "pequeno" refere-se a programas de menos de 900 linhas, "médio" refere-se a programas entre 900 e 40.000 linhas e "grande" acima de 40.000 linhas. Programadores podem ser

"noviços", "estudantes de graduação" e "programadores profissionais" , (MAYRHAUSER, 1995) .

O principal motivo que leva à necessidade de entender um programa é que muitas regras de negócios estão embutidas nos programas e não estão documentadas de forma explícita e precisa em nenhum outro lugar. Essas regras de negócios são bens valiosos para as empresas e muito difíceis de serem captadas e redesenvolvidas, (Neighbors, 1989).

2.2.1 Entendimento de Programas: Trabalhos Relacionados

Em (PRADO, 1992), apresenta um modelo cognitivo do comportamento do programador, dividindo seu conhecimento em semântico e sintático. O conhecimento semântico consiste de conceitos gerais de programação independentes de linguagens específicas, enquanto que o conhecimento sintático é mais preciso, detalhado e fácil de esquecer. O programador cria representações internas na resolução de problemas tais como composição, compreensão, depuração, modificação e aprendizagem do programa.

Os componentes da memória humana são utilizados de maneira distinta na resolução de problemas. O conhecimento semântico e sintático do programador é armazenado em sua memória de longo prazo. Esse conhecimento é usado como base na resolução de problemas que envolvem novas informações contidas na memória de curto prazo. São relatados diversos experimentos empíricos, cujas conclusões levam à constatação de que a construção de uma estrutura semântica interna facilita a compreensão de programas.

A compreensão de programas por meio de experimentos empíricos, visa determinar como esta é afetada pela maior ou menor experiência de programação de quem tenta obtê-la. Quem tem maior experiência possui dois tipos de conhecimento ausentes em quem tem menos: de planos de programas e de regras de raciocínio de programas. Os primeiros são fragmentos de programas que constituem típicas seqüências de comandos que o programador tem experiência de uso em situações anteriores, as segundas são regras de bom senso comumente adotadas na prática de programação. Programadores experientes compreendem mais facilmente programas que tenham sido construídos com um determinado plano e que não infrinjam tais regras. Têm dificuldade de compreender programas que não tenham tais características. Já os novatos não são afetados por essas diferenças.

Biggerstaff, (1989), estabelece um processo básico de recuperação de projeto, composto de três passos. O primeiro ajuda no entendimento de programas, identificando estruturas de grande porte e associando-as com conceitos semânticos informais. O segundo ajuda na alimentação e de bibliotecas de reuso. O terceiro aplica os resultados obtidos na recuperação de projeto para descobrir candidatos a componentes de um sistema novo. Mostra a importância da utilização de informações informais, tais como comentários e nomes de variáveis mnemônicos, e de um modelo do domínio, para a construção de abstrações conceituais do projeto. Introduz as conexões associativas e os padrões estruturais como um modo de formalizar parcialmente as abstrações conceituais informais. As conexões associativas fazem a ligação entre um conceito abstrato e o respectivo código. Os padrões estruturais definem os tipos de estrutura de código fonte que representam cada abstração. Apresenta o primeiro protótipo do DESIRE ("*Design Recovery"),* um sistema recuperador de projetos baseado em modelos, que auxilia os engenheiros de software no entendimento de programas, mostrado na Figura 2.1. O sistema consiste de três partes principais: um "*parser"*, um conjunto de funções de pós-processamento e um sistema de hipertexto chamado "*PlaneText*". O *parser* processa

programas em linguagem "C" e gera as *"parse trees",* que são usadas pelos pós-processadores para produzir um dicionário contendo informações e relacionamentos entre funções e itens de dados, além de associá-las a informações informais (comentários, por exemplo). O sistema de hipertexto exibe os relacionamentos por meio de um "*browser"*, permitindo visões específicas em janelas separadas, de acordo com a necessidade do usuário. O sistema oferece, também, "*queries*" pré-definidas em Prolog, que ajudam na busca por padrões.

Prado (1992), reforça a idéia de que a compreensibilidade de programas é uma parte vital do processo de manutenção. Comenta algumas teorias de compreensão de programas, que modelam o processo de compreensão. Discute a dificuldade de entendimento de código apenas pela leitura, a qual é diretamente dependente do tamanho e complexidade do programa. Discute ainda outros fatores que afetam o entendimento de programas, tais como: estilo de programação, modularização, nomes de variáveis, tabulação e comentários. Descreve algumas das técnicas empregadas para automatizar a leitura de código, categorizando-as em estratégias estáticas e dinâmicas.

Considera-as como auxiliares de baixo nível ao processo de compreensão, pois ajudam na compressão de pequenas unidades do programa, mas oferecem muito pouca ajuda na determinação do projeto geral. As vantagens da automação são óbvias na manutenção de sistemas grandes nos quais nenhum dos mantenedores participou do desenvolvimento. A engenharia inversa ou engenharia reversa é considerada como auxiliar de alto nível, porém ainda precisa de muito trabalho para se tornar efetiva. As técnicas de inteligência artificial são consideradas uma área de pesquisa muito útil à compreensão de programas, porém pouco exploradas.

Biggerstaff (1994), relaciona o entendimento de programas com o problema de assimilação de conceito. Define como "Problema de assimilação de conceito" a descoberta de conceitos que envolvem um grande conhecimento do domínio de informação e sua associação com um programa ou contexto específico, defendendo a idéia de que não há algoritmo que permita o reconhecimento desses conceitos com plena confiança. É feita uma comparação entre conceitos orientados à programação, que podem ser derivados de forma dedutiva ou algorítmica, e conceitos orientados ao ser humano, que requerem um raciocínio razoável e exigem um conhecimento prévio do domínio em questão. Sugere a utilização de uma ferramenta para recuperação de projeto, o DESIRE, composto de um assistente "ingênuo", que oferece serviços simples para apoiar a inteligência humana e um assistente "inteligente", que oferece uma assistência inteligente, por meio de um motor de inferência baseado em Prolog e um reconhecedor de padrões baseado em conhecimento.

Propõe uma abordagem para recuperação de componentes reusáveis, na qual componentes funcionais de sistemas legados são reconhecidos, recuperados, adaptados e reutilizados no desenvolvimento de um novo sistema. Essa recuperação requer profunda análise e entendimento do código antigo. É apresentado um conjunto de ferramentas chamado Cobol/SRE, que ajuda no entendimento de programas por meio da segmentação de programas que, basicamente, divide o programa em partes mais fáceis de serem compreendidas por quem for utilizá-las. O Cobol/SRE oferece recursos para demarcar os segmentos e posteriormente empacotá-los em módulos independentes. Para a demarcação é possível englobar, automaticamente, as linhas de código afetadas por um "*perform*", uma condição, o valor de uma entrada ou saída. O empacotamento gera novos módulos de programas, baseando-se nos segmentos criados na demarcação, tornando o código antigo mais modularizado e portanto mais reusável.

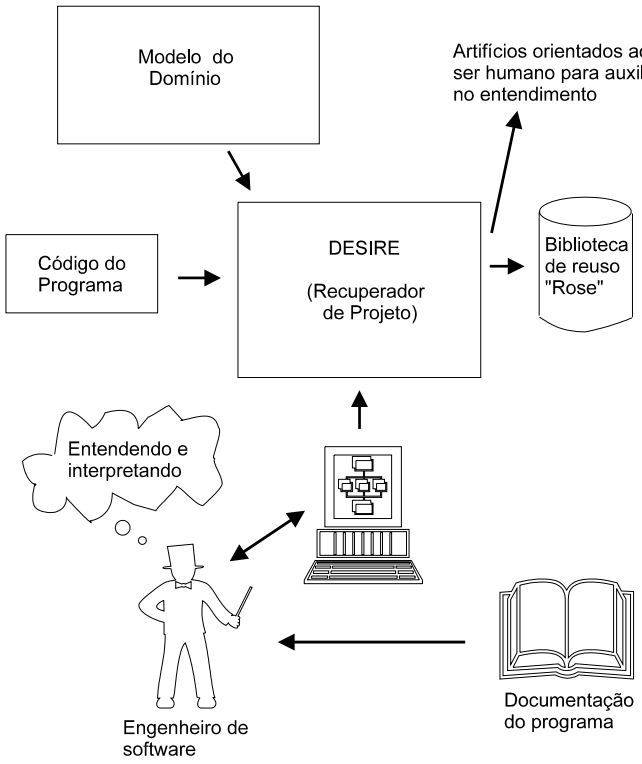


Figura 0.1: Sistema recuperador de projeto baseado em modelo *desire* apresentado por Biggerstaff (1989)

Leite (1995), discute a compreensão de programas durante a manutenção e evolução do software. Subdivide a manutenção em adaptativa, aperfeiçoativa e corretiva, detendo-se também no reuso e na alavancagem de código (*"code leverage")*.

Ressalta que, para que o processo de entendimento de programas possa ser otimizado, é necessário que se compreenda como os programadores entendem o código, o que pode ser apoiado pelos modelos de cognição. Identifica os elementos comuns aos modelos de cognição, a saber, o conhecimento, o modelo mental e as características do especialista. Descreve vários modelos de cognição de código, entre os quais: o modelo de Letovsky; o modelo de Shneiderman e Mayer , o modelo de Brooks; os modelos de fluxo de controle e o funcional, concebidos por Pennington; o modelo "*Top-down*", concebido por Soloway, Adelson e Ehrlich e o seu próprio modelo "integrado" (Figura 2.2). O modelo "integrado" considera três componentes: o modelo do programa, o modelo da situação e os planos de programação. No modelo do programa estão consolidados o conhecimento do domínio do programa, isto é, da estrutura de texto, dos planos e das regras de raciocínio. No modelo da situação estão reunidos os conhecimentos do domínio do problema, ou seja, o conhecimento funcional. Nos planos de programação constam os planos estratégicos, táticos e de implementação, bem como regras de raciocínio. A cada um dos três componentes corresponde um processo de modelagem e um setor integrado na base de conhecimento. No processo de entendimento há alternância entre os três modelos, de modo a conseguir novos conhecimentos que tornam a alimentar a base de conhecimento. Esse modelo permite tanto o entendimento "*top-down*" quanto o entendimento "*bottom-up"*, ou a mistura dos dois.

Mayhauser apresenta também uma análise comparativa quanto aos critérios estáticos: estruturas de conhecimento e representações mentais; dinâmicos e de

experimentação, indicando o grau de abstração e dimensão da escala dos experimentos, bem como se o objetivo é ganhar um entendimento geral superficial ou parcial e mais aprofundado. Em (1996), Mayrhauser reforça a parte experimental do trabalho, comprovando que os processos cognitivos atuam simultaneamente em todos os níveis de abstração, à medida que os programadores constróem o modelo mental do código.

Uma forma de facilitar o entendimento de programas, cujo domínio de aplicação seja totalmente desconhecido, é utilizar conhecimento geral, como por exemplo proximidade de comandos, separação por linhas em branco, similaridade entre símbolos formais e informais e acoplamento de definições via símbolos em comum. Porém, nem todo programador utiliza-se dos bons preceitos da programação. Isso tem que ser levado em conta quando se fazem estudos empíricos.

Entender um programa que contém implícitas regras de negócio valiosas e desconhecidas por qualquer pessoa na empresa, pode ser um estímulo incomparável a qualquer outro para o avanço da tecnologia de entendimento de programas. Muitas empresas que possuem programas nessa situação fazem constantemente investimentos nessa área.

A correspondência entre conceitos orientados ao ser humano e trechos de código nunca será totalmente automatizada, mas é possível apoio por computador, de utilidade para o engenheiro de software. Acrescentando-se elementos por ele fornecidos para completar os aspectos não atingidos pela automação, pode-se acelerar e simplificar de forma significativa o entendimento de programas, conforme Biggerstaff (1994). Essa idéia é unânime em todos os trabalhos estudados.

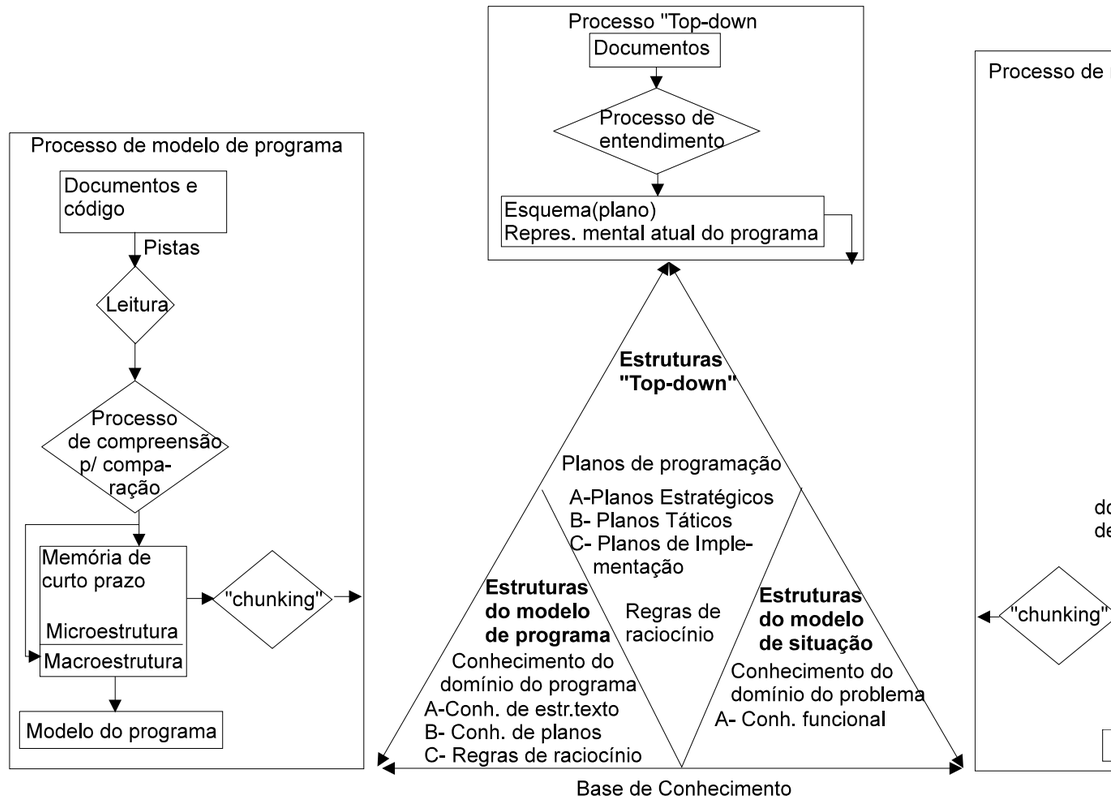


FIGURA0.2.*MetamodelointegradoMayhouser(1995)*

2.3 Reengenharia de Sistemas

Como vimos, a reengenharia consiste na análise e alteração do sistema, para reconstruí-lo em uma nova forma **e** para a sua realização é necessário primeiro empregar o processo de engenharia reversa sobre o sistema legado. A seguir, resumiremos alguns conceitos e trabalhos relacionados a reengenharia.

2.3.1 Reengenharia de sistemas: Conceitos e Trabalhos Relacionados

Freitas (Freitas, 1997), relata que um programador pode reconstruir a maior parte da hierarquia de um projeto de software através da explicitação das estruturas de dados e algoritmos comumente usados e conhecer como essas técnicas são implementadas em abstrações de nível mais alto; essas estruturas de programação são chamadas pelos autores de *clichés.* Com isso, os autores construíram um protótipo de uma ferramenta, chamada de *Recognizer,* que encontra automaticamente todas as ocorrências de um certo conjunto de *clichés* em um programa e constrói uma descrição hierárquica do programa a partir desses.

Do ponto de vista prático, o reconhecimento automático de *clichés* facilita muitas tarefas de engenharia de software como documentação, manutenção, otimização e depuração. Do ponto de vista teórico, esse reconhecimento é um problema em relação à extração dos conhecimentos e das experiências de programação.

Leite (1995), afirma que a transformação de programas procedurais em programas orientados a objetos é um processo importante para aumentar o potencial de reuso dos programas procedurais, mas que existem problemas difíceis de resolver no paradigma procedural, como é o caso da interconexão de módulos. Relatam que o reuso está assumindo papel relevante na produção industrial de software pois reduz os custos de desenvolvimento de software e melhora a qualidade do produto final. Os autores propõem um processo de transformação, conhecido como método COREM (*Capsule Oriented Reverse Engineering Method*) Leite; Prado; Santana (1992), que usa conhecimento do domínio da aplicação.

Esse método consiste de quatro passos principais: 1) recuperação do projeto: geração de diferentes documentos do projeto (diagramas de fluxo de dados, diagrama de entidades e relacionamentos) a partir do código fonte procedural e um modelo de aplicação orientado a objetos (RooAM); 2) modelagem da aplicação: a partir dos requisitos da análise do programa procedural é gerado um modelo de aplicação precoce orientado a objetos (FooAM); 3) mapeamento dos objetos: os elementos do RooAM são mapeados para os elementos do FooAM, resultando em um modelo de aplicação alvo (ooAM). Nessa fase algumas partes podem permanecer em um dos modelos de aplicação, sem correspondência, devido às disparidades dos paradigmas; 4) transformação do sistema: a partir do código fonte e dos resultados dos passos anteriores, é realizado o processo de transformação do programa resultando em três tipos diferentes de objetos: **objetos da aplicação** (definidos durante o processo de mapeamento entre RooAM e FooAM), **objetos de dados** (resultantes do encapsulamento dos dados globais) e **objetos de implementação** (resultantes das partes restantes do sistema procedural que não possuem correspondência com a implementação orientada a objetos). Embora a aplicação de ferramentas seja útil e necessária durante esse processo de transformação, é imprescindível a aquisição de conhecimento adicional de um analista. Portanto, a automação completa desse processo de transformação não é possível, mas muitos passos do processo podem ser auxiliados por ferramentas.

Gall e outros (GALL;KLOSCH, 1997) declaram que o método COREM auxilia na melhora da manutenibilidade de programas procedurais e na reusabilidade, pois permite a extração de componentes reusáveis dos softwares procedurais existentes. Esses componentes podem ser armazenados em uma biblioteca de componentes de software e ser integrados mais facilmente às novas arquiteturas em desenvolvimento. O software reusável reduz o custo de projeto e o tempo de codificação, pois o projeto e a implementação são reaproveitados.

Além disso, a reusabilidade produz softwares mais fáceis de serem entendidos e mais confiáveis, uma vez que a parte do código que está sendo reutilizada já foi testada em projetos anteriores. Para elevar a reusabilidade dos componentes em diversos projetos, é necessário que os desenvolvedores obedeçam às regras de estilo, como: manter a coerência dos métodos (cada método realiza uma única função ou um grupo de funções relacionadas), generalizar o método tanto quanto possível, manter baixo acoplamento e alta coesão. Os autores afirmam que o esforço para realizar esse processo de engenharia reversa é bem menor em relação ao esforço necessário para o desenvolvimento de um novo sistema orientado a objetos, pois o método de transformação COREM preserva a funcionalidade do sistema.

Leite (1995) apresenta um enfoque de extração automática de documentação de projeto orientada a objetos a partir de programas existentes em *mainframes*, escritos em linguagem COBOL. A desvantagem é que o código tem que ser reescrito e os dados de teste gerados novamente. Os objetos são deriváveis de quatro tipos de informações diferentes do programa: arquivos, visões do banco de dados, estruturas de dados e parâmetros de *linkagem* do programa, com isso sete tipos de objetos são identificados: objetos de interface de usuário, objetos de informação, objetos de arquivo, objetos de registro, objetos de visões, objetos de trabalho e objetos de *link*. Os métodos são extraídos do código através de análise de referência cruzada. Os parâmetros ou interfaces dos métodos são a base para determinar a conexão entre os objetos. Como passo final desse processo de recuperação do projeto, a seqüência em que os métodos são executados é capturada e documentada. Para auxiliar na recuperação do projeto, foi desenvolvida uma ferramenta, a OBJECT-REDOC, que não reconhece certas estruturas de programação, já que essas dependem exclusivamente do estilo de programação de cada desenvolvedor.

Também discute a importância da engenharia reversa para a realização de manutenção preventiva, corretiva e adaptativa nos sistemas legados das empresas; também existem outras aplicabilidades dessa abordagem como: diminuição dos custos de manutenção, melhorias na qualidade do sistema, vantagens competitivas ou facilidade no reuso de software. O autor cita a dificuldade que a equipe de engenharia reversa encontra na análise de grandes e complexos sistemas que foram criados sem técnica de desenvolvimento (programação estruturada, orientação a objetos, etc). Embora existam ferramentas que auxiliem na extração de informações do código fonte, essas são limitadas quanto ao tamanho dos programas ou quanto à qualidade de seus resultados, por isso é necessária a intervenção de um especialista.

Mostra os resultados referentes à análise de métricas de produtividade e qualidade dos dados durante um processo de engenharia reversa. Esse trabalho foi motivado pelas poucas informações existentes sobre o esforço requerido para realizar um processo de engenharia reversa. Os autores descrevem um estudo de caso real de engenharia reversa e mostram quais os dados considerados mais úteis para seu entendimento. Algumas das medições realizadas nos programas foram: contagem do número de linhas de código depois da realização de melhorias em sua estrutura; eliminação de módulos não utilizados, excluindo linhas de comentários; contagem das linhas de código da

Procedure Division, uma vez que o sistema estudado foi desenvolvido em Cobol; aplicação das métricas de complexidade de *McCabe* e volume de *Halstead,* conforme apresentado por Pressman (1995). Os dados obtidos mostraram que 1/4 do esforço é gasto em outras atividades além daquelas do processo de engenharia reversa, como é o caso da atividade de documentação; atividade de criação ou adaptação de ferramentas utilitárias para apoiar a análise realizada e atividade de validação realizada por especialistas do domínio da aplicação. Esse valor de esforço varia com as particularidades das atividades de engenharia reversa utilizadas, com o sistema de software específico, com a linguagem utilizada para implementá-lo e com o ambiente em que é executado. A produtividade foi avaliada através da medição do tempo gasto na reconstrução e recuperação do nível lógico e da comparação desse tempo com o número de linhas de código dos programas. Esse estudo auxilia o analista na verificação se um sistema pode ser passado para uma fase posterior à de engenharia reversa, isto é, a uma possível reengenharia.

Prado, (1992) sugere reengenharia de sistemas antigos para a arquitetura orientada a objetos, tendo como objetivo mostrar como um método orientado a objetos pode ser usado para modelar sistemas legados não desenvolvidos segundo esse paradigma.

Os autores definem reengenharia como sendo a somatória de engenharia reversa, e engenharia avante. Relatam que a fase de engenharia reversa é o primeiro passo a ser realizado e consiste em uma definição mais abstrata e mais fácil de ser compreendida da representação do sistema. O segundo passo, representa as possíveis modificações funcionais e de implementação do sistema. O terceiro e último passo, engenharia avante, refere-se ao desenvolvimento normal de sistemas, isto é, à implementação do sistema em alguma linguagem de programação. Nesse caso, uma linguagem que seja orientada a objetos. Nesse trabalho, os autores consideram os seguintes tipos existentes de reengenharia: 1) troca completa da implementação sem troca da funcionalidade; 2) troca parcial da implementação sem troca da funcionalidade e 3) troca total da funcionalidade.

O quadro 2-1 mostra esses tipos de reengenharia com seus respectivos passos.

Penteado (1996), propõe uma abordagem de engenharia reversa - Fusion/RE para obter modelos de análise orientados a objetos a partir de sistemas desenvolvidos sem utilizar técnicas de orientação a objetos. Os modelos criados com a utilização de Fusion/RE são baseados nos modelos da fase de análise do método Fusion [Col94].

Gall e Klosch (1995) declaram que devido às divergências existentes entre os paradigmas, procedural e orientado a objetos, não é possível mapear diretamente todos os elementos de uma aplicação procedural para uma orientada a objetos. Portanto, o sistema orientado a objetos resultante consiste de duas partes: a parte orientada a objetos e a parte restante procedural. Assim, para transformar a parte restante procedural em orientada a objeto é necessário realizar adaptações na implementação através da identificação de objetos, a partir de critérios funcionais. Os autores ressaltam que para transformar o código procedural em um código orientado a objetos, é necessário preocupar-se com as seguintes considerações: adaptações no programa com a transformação de variáveis e procedimentos em atributos e métodos, respectivamente; adaptações nas interfaces dos serviços, uma vez que o estado interno de um objeto é acessível para cada serviço; isolamento e encapsulamento dos ítens de dados globais em objetos separados (nomeados como objetos de dados, pois contêm somente atributos) para assegurar uma completa transformação orientada a objetos.

Tipo 1 - Troca completa da implementação sem troca da funcionalidade	Tipo 2 -Troca parcial da implementação sem troca da funcionalidade	Tipo 3 -Troca total da funcionalidade (engenharia avante)
PASSOS	PASSOS	PASSOS
1) preparar um modelo da análise;	1) identificar as partes do sistema que serão reimplementadas.	1) modificar o modelo de análise de acordo com os requisitos;
2) mapear cada objeto da análise para a implementação do sistema antigo;	2) preparar um modelo de análise da parte a ser trocada e seu ambiente;	2) projetar o sistema.
3) reprojetar o sistema.	3) mapear cada objeto para a implementação antiga do sistema;	
4) implementar o modelo de análise.	4) repetir os passos anteriores até que a interface entre a parte a ser trocada e a parte restante do sistema existente seja aceitável;	
	5) em paralelo: 5.1) projetar o novo subsistema e suas interfaces para o que falta do sistema antigo; 5.2) modificar o sistema antigo e adicionar uma interface ao novo subsistema;	
	6) integrar e testar o novo subsistema e o sistema antigo modificado.	

Quadro2.1: Tipos principais de reengenharia existentes e seus respectivos passos

Gall e Klösch (1995), relatam que identificar objetos em programas procedurais e tornar suas dependências explícitas, colaboram para o entendimento do projeto do sistema, evita degradação dos projetos originais durante a fase de manutenção e facilita o processo de reusabilidade. Além disso, essa identificação de objetos constitui base para uma completa rearquitetura do sistema, de um programa procedural convencional para um programa orientado a objetos. Esse processo de identificação de objetos não é baseado somente em informações deriváveis do código fonte do programa examinado, mas também integra conhecimento do domínio específico da aplicação para permitir resultados que representem objetos da aplicação semântica. Nesse trabalho foi também declarado que métodos automáticos de identificação de objetos são basicamente problemáticos porque eles podem retornar grupos de objetos irreais, por isso surgiu a necessidade de uma abordagem que integrasse conhecimento do domínio no processo de identificação dos objetos.

A engenharia reversa precisa ser capaz de extrair a informação necessária para sustentar o trabalho de análise de reengenharia.

Existem vários aspectos a serem considerados ao se iniciar a reengenharia. O primeiro ponto importante é que mesmo dentro de um dado sistema aplicativo pode haver problemas de diversas espécies. Um programa individual mostrará seus próprios problemas característicos, que poderão ser diferentes de qualquer outro sendo examinado.
Sem as ferramentas corretas para apoiar a fase de engenharia reversa, a eficiência do trabalho é significativamente reduzida.

Os resultados esperados de um trabalho de reengenharia podem ser expressos de duas formas básicas. Primeiro, eles devem ser medidos conforme os critérios estabelecidos no projeto piloto. Em segundo lugar, eles podem ser acessados pela equipe responsável pelo apoio ao produto e dos argumentos de custo/benefício em favor da reengenharia. A seguir temos uma lista de exemplos que mostram os possíveis resultados que devem ser alcançados após a reengenharia:

- O tamanho do código deve ser reduzido em alguns programas. Esta é uma melhoria considerável, principalmente para quem considera ser o tamanho do código uma boa medida da sua complexidade. Isto inclui a remoção de procedimentos e de dados redundantes.
- A complexidade McCabe deve ser reduzida. Diversos programas mostravam partição lógica. A reengenharia deve introduzir novas condições.
- A performance deve aumentar. Alguns autores estabelecem que a performance deve, no mínimo, ser a mesma para o novo código.
- código *´spaghetti´* deve ser removido, o que deve ser medido utilizando a análise dos nós.
- As violações de padrões devem ser removidas.
- Armazenagens internas múltiplas devem ser reduzidas para definições simples.
- A distribuição deve ser reduzida a níveis normais entre 5 e 9.
- Procedimentos longos devem ser divididos de forma a aumentar a coesão funcional.
- A profundidade excessiva deve ser removida pelo redesenho do controle e da lógica de processamento
- Ligações entre seções lógicas de processamento e seções de arquivo I/O devem ser reduzidas, permitindo melhor controle das funções I/O.
- Os nomes dos dados devem ser padronizados dentro dos programas. Com isto deve ocorrer um melhoramento nos dicionários de dados.

Tentar descrever o sucesso da reengenharia é difícil. Os melhoramentos são comumente medidos com a utilização de diversas métricas. E há desentendimentos freqüentes a respeito destes padrões. Por exemplo, consideramos a complexidade McCabe útil. Contudo, temos que usar diversas medidas de complexidade onde o padrão McCabe é totalmente inadequado.

Nossas experiências iniciais mostram a possibilidade da reengenharia ser comercialmente viável. Grande cuidado, entretanto, deve ser tomado na determinação de métodos e técnicas a serem utilizados antes da automação. Isto é necessário em virtude da falta de experiência aprofundada em reengenharia e por causa da grande diversidade de problemas que podem ser encontrados.

2.4 Abordagens para Engenharia Reversa, Reengenharia e Migração de Sistemas

Com base nos conhecimentos apresentados até agora, a seguir serão resumidas algumas abordagens para Engenharia Reversa, Reengenharia e tradução automática de código fonte.

A seção está estruturada como segue. A seção 2.5.1 resume a abordagem Fusion/RE enquanto a seção 2.5.2 resume as características do ambiente Draco-PUC. A seção 2.5.3 sintetiza algumas categorias de abordagens de Engenharia Reversa de Banco de Dados.

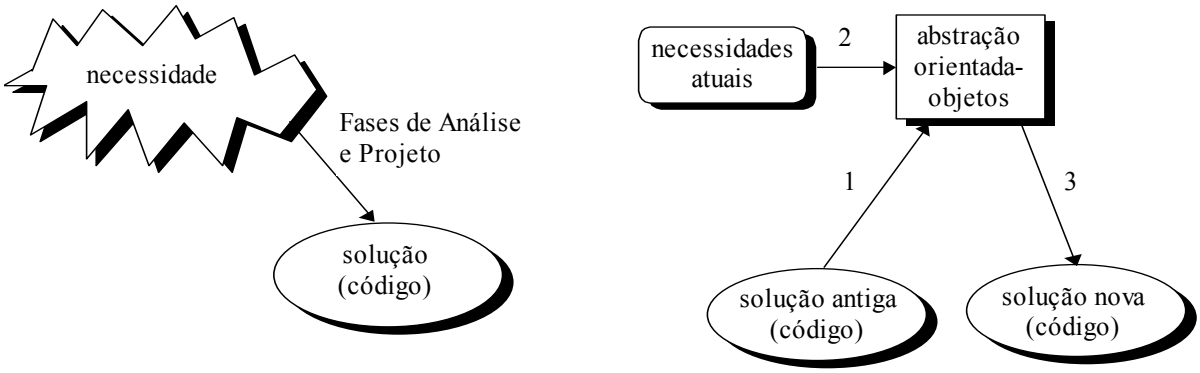
2.4.1 A Abordagem Fusion/RE

Fusion/RE é uma abordagem de engenharia reversa orientada a objetos e tem como objetivo a recuperação do projeto de sistemas legados. Para a realização do processo de engenharia reversa a partir dessa abordagem não há necessidade de se ter a documentação do sistema. A recuperação do projeto é possível apenas através do código fonte e de entrevistas com os usuários, quando possíveis.

Para implementar um sistema orientado a objetos a partir de um sistema legado orientado a procedimentos, são dadas duas razões: 1) os sistemas legados satisfazem a maioria das necessidades dos usuários mas a melhoria da programação de sua interface é necessária para estender ou padronizar as suas funções e 2) o código desses sistemas legados pode ser reutilizado para implementar totalmente ou parcialmente as funções de um sistema orientado a objetos, conforme Penteado, (1996). Assim, Fusion/RE foi criado para possibilitar a migração de um sistema procedural para um sistema orientado a objetos.

A Figura 2.3 mostra a diferença entre o desenvolvimento de software convencional (Figura 2.3 (a)) e o processo de engenharia reversa Fusion/RE (Figura 2.3 (b)). Para realizar um desenvolvimento de software convencional, os requisitos do sistema são obtidos e logo em seguida são realizadas as fases de análise, projeto e implementação. Esse processo tem como resultado o código fonte consistente com os requisitos solicitados, atendendo completamente às necessidades dos usuários.

Em um processo de engenharia reversa a recuperação do projeto inicia-se a partir do código fonte. Em Fusion/RE o modelo recuperado possui características de orientação a objetos. Essa documentação completa do sistema pode ter várias utilidades, como: auxiliar em futuras manutenções do sistema, dar suporte a um processo de reengenharia, podendo alterar a funcionalidade do sistema, realizar a troca total ou parcial da implementação. Os números indicados nas setas representam a ordem em que as operações devem ser realizadas. Note que a operação (3) é realizada quando se opta pela alteração do código fonte.



(a)
Esquema de um desenvolvimento de software convencional

(b)
Esquema da abordagem Fusion/RE

Figura 0.3: Comparação do desenvolvimento convencional com o Fusion/RE

O QUADRO 2-2, extraído de Penteado, (1996), mostra um breve resumo da abordagem Fusion/RE. Mais detalhes sobre essa abordagem de engenharia reversa orientada a objetos podem ser obtidos em Penteado (1996).

2.4.2 Reengenharia com mudança de linguagem: o projeto Draco-PUC

A reengenharia com mudança de linguagem de programação, além de atualizar o código para plataformas mais modernas, pode ser realizada para dar continuidade ao processo de engenharia reversa. Isso faz com que o código fique sincronizado com a análise e o projeto, melhorando, portanto, a manutenibilidade futura.

Após ter sido realizada a engenharia reversa, a reengenharia com mudança de linguagem pode ser feita manualmente ou auxiliada por uma ferramenta de transformação.

A transformação automática de uma linguagem para outra, se feita de forma direta, na maioria das vezes gera código difícil de compreender e manter. A execução prévia do processo de segmentação permite que essa transformação seja mais amena e o código gerado seja mais compreensível. Tal experimento foi feito usando a máquina Draco-Puc e é objeto de um artigo publicado em co-autoria com Penteado e outros (1996). Um resumo desse experimento é relatado nas subseções que se seguem.

A máquina Draco é baseada nas idéias de construção de software por transformação orientada a domínios. Um primeiro protótipo da máquina Draco foi construído por Neighbors (1984). Posteriormente, ela foi reconstruída na PUC-RJ, usando novas linguagens de plataformas modernas de hardware e de software. Essa versão foi denominada Draco-Puc.

Pela estratégia proposta por Prado (1992), é possível a reconstrução de um software pelo porte direto do código fonte para linguagens de outros domínios. Um domínio, de acordo com a máquina Draco-Puc, é constituído de três partes: um *parser*, um *pretty-printer* e um ou mais transformadores.

Para a reimplementação automática é necessária a construção dos domínios da 0linguagem origem e da linguagem destino, por intermédio de cinco atividades básicas: construir *parsers,* construir *pretty-printers*, construir um transformador da linguagem origem para a base de conhecimento, construir um transformador da linguagem origem para a linguagem destino e construir bibliotecas, conforme mostra a Figura 2.4.

Na atividade “construir *parsers*” obtém-se os *parsers* das gramáticas livres de contexto dos domínios origem e destino, a partir da definição dos seus analisadores léxicos e sintáticos. Essa atividade é auxiliada pelo subsistema gerador de *parsers* do Draco, denominado “Pargen”. As definições dos *parsers* são usadas como entrada nessa atividade.

Ao lado das regras gramaticais têm-se as ações semânticas (*makenode*, *makeleaf*, etc.) usadas para construção da DAST (Draco *Abstract Sintax Tree*), que é a linguagem interna usada pelo Draco nas transformações.

Passo	Objetivo	Produto
1. Revitalizar a Arquitetura do Sistema com base na documentação existente	Obter informações relacionadas à arquitetura do sistema para o seu entendimento	Lista de todos os procedimentos, sua descrição e a relação chama/chamado por
2. Recuperar o Modelo de Análise da Solução Atual	Obter um modelo de análise considerando somente os aspectos físicos	Documentos descritos nos passos 2.1., 2.2., 2.3., 2.4.
2.1. Definir Temas	Modelar em temas as informações armazenadas relativas às entradas, saídas, armazenamento permanente e temporário	Lista de Temas
2.2. Desenvolver oModelo de Objetos	Elaborar um modelo com as classes e seus relacionamentos, extraídos dos tipos abstratos de dados que compõem a base de dados do sistema	Modelo de Objetos, lista de atributos, procedimentos associados às classes, lista das anomalias existentes
2.3. Definir o Ciclo de Vida	Mostrar o comportamento global do sistema	Modelo de Ciclo de Vida
2.4. Abstrair e Desenvolver as Operações	Obter as operações realizadas pelo sistema	Modelo de Operações
3. Abstrair o Modelo de Análise do Sistema	Obter um modelo de análise do sistema considerando os aspectos do domínio da aplicação	Documentos descritos nos passos 3.1., 3.2., 3.3.
3.1. Desenvolver oModelo de Objetos	Elaborar um Modelo de Objetos considerando as classes e seus relacionamentos que devem ser tratados pelo sistema	Modelo de Objetos. Para cada classe: lista dos atributos e métodos
3.2. Elaborar o Modelo de Ciclo de Vida	Fornecer uma visão global do comportamento do sistema a partir da abstração realizada	Modelo de Ciclo de Vida
3.3. Especificar o Modelo de Operações	Descrever como as operações devem ser realizadas	Modelo de Operações
4. Mapear o Modelo de Análise do Sistema para o Modelo de Análise do Sistema Atual	Descrever a Relação entre os Modelos de Análise do Sistema atual e novo.	Mapeamento das Classes e dos Métodos do MAS para o MASA

Quadro 2.2: Abordagem de engenharia reversa orientada a objetos , Penteado (1996), p. 131.

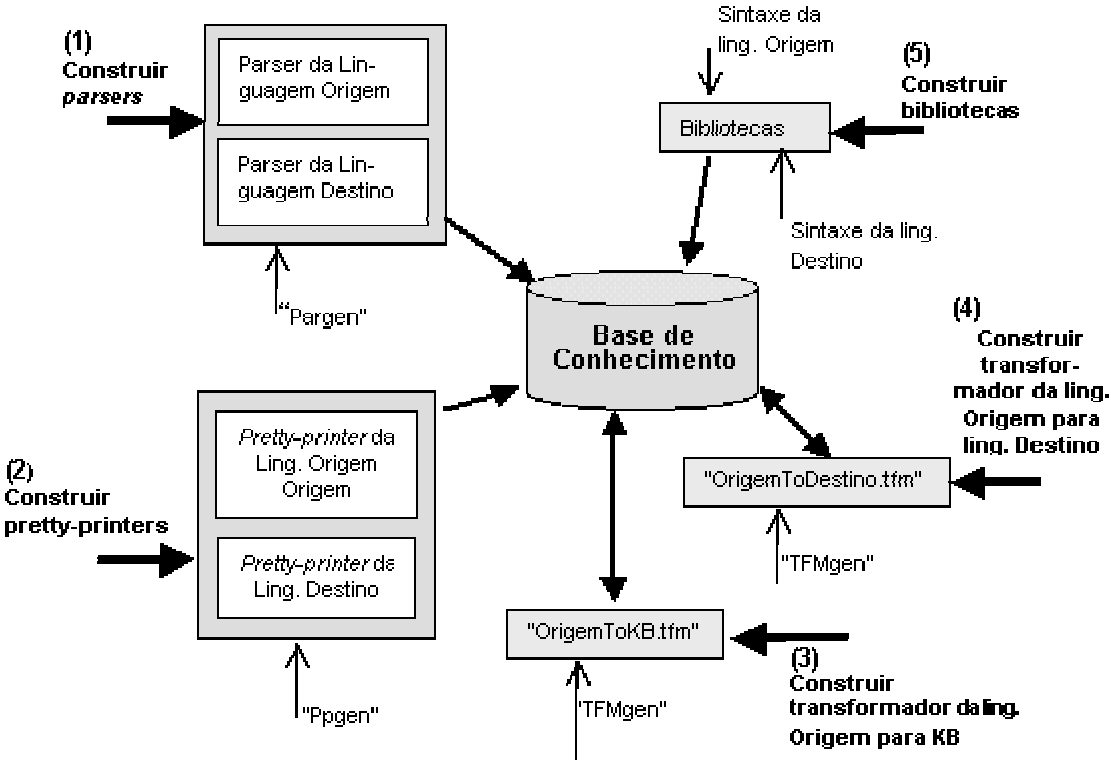


Figura 2.4: Atividades básicas para a construção dos domínios de origem e destino no Draco, de acordo com Prado (1992)

Na atividade "construir *pretty-printers*" o Draco gera automaticamente os *pretty-printers* das linguagens origem e destino, usando seu sub-sistema "Ppgen". Nessa geração, parte-se das definições das regras gramaticais das linguagens origem e destino e das primitivas de formatação (.sim, .sp, etc.) colocadas ao lado das regras. Um *pretty-printer* trabalha como *unparser* que exibe a DAST orientada pela sintaxe da linguagem, usando as primitivas de formatação. Exemplo dessas primitivas são: a que estabelece a margem esquerda na coluna corrente (.lm), a que exibe o n-ésimo símbolo da regra gramatical (.#n), etc.

A máquina Draco dispõe de uma base de conhecimento (Knowledge Base (KB)) semelhante à linguagem PROLOG para armazenar fatos e regras. Na atividade "construir o transformador da linguagem origem para a base de conhecimento" procura-se percorrer a DAST da linguagem de origem para reconhecer as suas diferentes estruturas, como os tipos de variáveis e funções, para gerar fatos na KB. O esquema KB que orienta a geração dos fatos é definido previamente. Nessa atividade, o engenheiro de software faz uso do subsistema "TFMgen" do Draco.

Na atividade "construir o transformador da linguagem origem para a linguagem destino", constrói-se o transformador inter-domínio que faz o mapeamento semântico da linguagem origem para a linguagem destino, utilizando os fatos armazenados na KB. O engenheiro de software usa o sub-sistema "TFMgen" do Draco para gerar esse transformador.

Para completar o processo de transformação são construídas bibliotecas com classes da linguagem origem e destino. Essas bibliotecas expressam, na linguagem destino, comandos da linguagem origem.

Após a construção dos domínios origem e destino, os programas escritos na linguagem origem são submetidos ao Draco, que depois de uma série de etapas produz o código fonte na linguagem destino. O código escrito na linguagem origem é inicialmente analisado pelo Draco usando o *parser* produzido na atividade 1 citada

anteriormente. Isso resulta na DAST com a representação interna do código original. Essa DAST é então submetida ao *unparser*, que a formata de acordo com a sintaxe da linguagem origem, fazendo uso do *pretty-printer* da linguagem origem, gerado na atividade 2. O código obtido é então submetido ao transformador construído na atividade 3, que extrai os fatos para a base de conhecimento.

Finalmente o código é submetido ao transformador construído na atividade 4 que, usando os fatos da base de conhecimento e as bibliotecas, transforma-o para a linguagem destino.

O paradigma Draco pode ser visto, e desta forma caracterizado, por vários pontos de vista, Freitas (1987). A motivação original para o seu desenvolvimento foi o de prover uma forma de construção de sistemas de software a partir de componentes preexistentes, ao contrário de construir o sistema através de métodos tradicionais. Um ponto de vista próximo seria o de construir sistemas através de um gerador de geradores, conseguindo, desta forma, gerar e manter facilmente uma classe de sistemas similares. Um terceiro ponto de vista é o de construir linguagens de especificação de alto nível para serem utilizadas na construção de sistemas, e por mecanismo de transformações produzirem os programas em linguagens executáveis.

O que está por traz destas definições é uma forma de organizar componentes de software para reuso diferente das disponíveis em linguagens de programação: bibliotecas de funções, ou bibliotecas de classes. No paradigma Draco, o elemento de reuso são linguagens formais que são denominadas domínios. Estas linguagens são construídas com intuito de encapsular os objetos e operadores de um determinado domínio, ou área de conhecimento. Os programas escritos nas linguagens dos domínios por sua vez devem sofrer um processo de refinamento até atingirem uma linguagem executável.

Para que se possa produzir software no ponto de vista do paradigma Draco, é necessário portanto que o processo seja dividido em duas etapas distintas: o encapsulamento do conhecimento do domínio, e a utilização do conhecimento encapsulado.

A máquina Draco-PUC, por sua vez, é um sistema de software que tem como objetivo principal implementar o paradigma Draco. O papel da máquina é o de com seu uso construir uma implementação real por tradução de todas as linguagens de domínio. Como a tradução ocorre entre os domínios (linguagens) surge a necessidade de domínios executáveis por exemplo: C, Pascal, Basic, Lisp entre outras. Estes são então os domínios alvos de nossas transformações.

O processo em que uma especificação escrita em um domínio fonte atinge um domínio alvo pode passar por inúmeros domínios intermediários. Esta propriedade diferencia o Draco-PUC de um Gerador de Geradores simples, pois além de construir o Gerador a máquina Draco-PUC permite que isto seja feito reutilizando outros domínios, o que acreditamos facilitar bastante o trabalho de construção das transformações de traduções.

Descrições mais detalhadas do paradigma Draco podem ser encontradas em versões anteriores do SBES, Leite (1992) e (1995), além de publicações internacionais Neighbors (1984) .

2.4.3 Engenharia Reversa de Bases de Dados

A estrutura de Banco de Dados talvez essa uma das mais difundidas e usadas técnicas de Engenharia Reversa. Com certeza, a maior necessidade de alteração dos sistemas legados diz respeito a forma como os dados são armazenados. Logo, é

necessário extrair dos dados os seus modelos conceituais para que eles possam ser acomodados aos novos paradigmas de gerenciamento de banco de dados.

Para representar um esquema de banco de dados em um outro esquema, é necessário a definição dos objetos de dados e a forma como eles se relacionam, ou seja, obter o modelo conceitual de dados. Com o modelo em mãos é possível alterar o modelo físico de um banco de dados de acordo com as necessidades apontadas pela Reengenharia.

Alguns métodos para recuperação dos modelos da base de dados podem ser agrupados nas seguintes categorias:

- Categoria 1: Essa categoria contém os métodos de engenharia reversa que classificam as relações. As relações são classificadas para depois serem transformadas em um esquema conceitual. O método de Chiang além de classificar as relações, classifica também os atributos. Os métodos de Chiang e de Batini e outros possuem características semelhantes pois, utilizam nomes consistentes para os atributos. O método de Chiang utiliza além do esquema de dados, também as instâncias de dados.

- Categoria 2: Essa categoria contém os métodos de engenharia reversa que tem como característica básica a utilização das dependências de inclusão para descobrir os tipos relacionamentos e também os tipos entidades. Os métodos dessa categoria são semelhantes apesar de terem algumas particularidades.

- Categoria 3: Essa categoria contém os métodos de engenharia reversa que utilizam as consultas SQL para achar os tipos entidades e os tipos relacionamentos. As instâncias de dados extraídas das consultas SQL são analisadas.

- Categoria 4: Essa categoria contém os métodos de engenharia reversa que utilizam um grafo para descobrir os tipos entidades e os tipos relacionamentos. Os método de Andersson utiliza as dependências de inclusão para construir o grafo. E utiliza ainda as consultas SQL para ajudar a construir o grafo.

- Categoria 5: Essa categoria contém o método de engenharia reversa que utiliza um conjunto de formulários para descobrir os tipos entidades e os tipos relacionamentos. O conjunto de formulários é gerado através de consultas SQL. A descoberta dos tipos relacionamentos entre os tipos entidades é feita através da observação das instâncias dos dados contidas no conjunto de formulários.

- Categoria 6: Essa categoria contém o método de engenharia reversa que utiliza apenas de nomes consistentes para os atributos para achar os tipos relacionamentos. Não são considerados nesse método de Davis as dependências de inclusão.

- Categoria 7: Essa categoria contém o método de engenharia reversa de Signore que não apresenta algoritmos para transformar o esquema relacional em um esquema conceitual. Esse método apresenta apenas informações (i.e. texto DDL, esquema relacional, dependências de inclusão, consultas SQL) que podem dizer qual tipo de objeto (i.e. tipo relacionamento ou tipo entidade) será extraído dessa informação.

2.5 Síntese

A abordagem "usual" para migração de sistemas é descartar o código antigo quando se cogita de ampliar a funcionalidade de um sistema existente, com idade avançada e desatualizado. O objetivo de um processo de migração com reengenharia é manter o conhecimento adquirido com os sistemas legados e utilizar estes conhecimentos como base para a evolução continua e estruturada do software . O código legado possui uma lógica de programação, decisões de projeto, requisitos do usuário e regras de negócio que podem ser recuperados e reconstruídos sem perda da semântica. O software é reconstruído com inovações tecnológicas e novos requisitos podem ser adicionados para atender prazos, custos, correções de erros e melhorias de desempenho.

A maioria dos sistemas legados não possui documentação, e quando possui, esta não está atualizada. Isto dificulta ainda mais o processo de reconstrução destes sistemas, sem garantia de recuperar o projeto original e manter as funcionalidades.

Neste capítulo, a partir do estudo da literatura, algumas idéias principais foram levantadas:

a) Engenharia Reversa: é o processo de análise de um sistema existente, identifica seus componentes e os representa em um nível mais alto de abstração.

b) O entendimento de programas é imprescindível para o sucesso de diversas atividades da engenharia de software, principalmente a manutenção, o reuso e a engenharia reversa. Um código só pode passar por alterações depois de entendido. Um componente só pode ser reusado se for entendido. Visões de um sistema em níveis de abstração mais altos que o próprio código só podem ser produzidas após o entendimento desse código.

c) Engenharia Reversa é tipicamente feita através de análise estática de código. No entanto, como Prado (1992), Leite (1995) e Penteado (1995), acreditamos que deve-se integrar o nível de *programming concepts* com o nível de *human concepts*, isto é, usar outros tipos de documentos e a experiência prática de pessoas no processo de engenharia reversa e reengenharia.

d) *Reengenharia:* consiste na análise e alteração do sistema, para reconstruí-lo em uma nova forma**.**

e) Como foi caracterizado acima, o processo de reengenharia carece de métodos e ferramentas de apoio tanto à escolha de tecnologias apropriadas para as especificidades das aplicações, bem como para a condução do processo de evolução e avaliação dos resultados. Sem este tipo de apoio, a reengenharia de sistemas legados continuará sendo um processo artesanal, envolvendo custos e prazos acima dos esperados/permitidos, com resultados muitas vezes não compensatórios e com chances muito grandes de resultar em um retumbante fracasso.

f) O processo de migração , isto é , de engenharia reversa seguida de reengenharia é um processo difícil e necessita de suporte;

f) A automação total do processo de migração do início ao fim é praticamente impossível devido às dificuldades de entendimento de programas mas a assistência (uso de ferramentas de software como auxiliar ao processo) é altamente recomendável nas atividades que seguem este entendimento. A automação pode ser feita após o entendimento e a reestruturação, neste caso, o problema de assistência a

migração fica reduzido a um problema de conversão de programas entre linguagens fontes com presumida equivalência semântica.

3 UMA ABORDAGEM PARA MIGRAÇÃO DE SISTEMAS DESENVOLVIDOS EM COBOL PARA PROGRESS

Neste capítulo, busca-se apresentar a nossa proposta de solução para migração de sistemas legados, descrevendo-a em particular para migração de sistemas em Cobol para Progress. A seção 3.1 apresenta uma visão geral do processo, com suas etapas e características principais. As seções seguintes detalham as etapas envolvidas neste processo. A seção 3.2 explica as atividades relacionadas à Engenharia Reversa, em particular detalhando a aplicação da Engenharia Reversa para a conversão das definições de bancos de dados do sistema original em Cobol para o sistema em Progress, usando a ferramenta ERWin. A seção 3.3 discute a Reestruturação do sistema. A seção 3.4 aborda a geração de parsers. A seção 3.5 aborda os detalhes para geração da base de conhecimento enquanto a seção 3.6 discute a utilização de transformadores. Todo o capítulo é permeado de exemplos retirados do estudo de caso realizado: a migração de um sistema real para suporte a vendas, compras e controle de estoque de peças no contexto de concessionárias e revendedoras autorizadas.

3.1 Processo de Migração: visão geral

Os sistemas implementados em COBOL levam muito tempo, cerca de quatro a cinco anos, para se estabilizarem e atenderem o mínimo de requisitos necessários. É grande a quantidade de sistemas implementados em versões antigas da linguagem COBOL, com interface orientada a caracter, cuja reengenharia é importante para atualizá-los para novas plataformas de hardware e software. A linguagem COBOL é uma linguagem estruturada de ampla utilização no mercado, por médias e grandes empresas que há muito tempo trabalham com sistemas aplicativos, por este motivo a linguagem de programação COBOL foi escolhida, como linguagem do código-fonte legado para validar a estratégia de reengenharia proposta.

Conforme mencionado no capítulo anterior, a reengenharia com mudança de linguagem de programação, além de atualizar o código para plataformas mais modernas, pode ser realizada para dar continuidade ao processo de engenharia reversa. Isso faz com que o código fique sincronizado com a análise e o projeto, melhorando, portanto, a manutenibilidade futura.

A transformação automática de uma linguagem para outra, se feita de forma direta, na maioria das vezes gera código difícil de compreender e manter. A execução prévia do processo de reestruturação permite que essa transformação seja mais amena e o código gerado seja mais compreensível.

Assim, este processo inicia-se com a reestruturação preliminar do código fonte, para fazer algumas melhorias no mesmo, como remoção de construções não estruturadas,

eliminação de variáveis não utilizadas, padronização do código (nome de variáveis) e tipos implícitos.

Com o trabalho de reestruturação, foi possível cumprir uma outra fase conhecida como *entendimento de programas* graças ao estudo detalhado do código-fonte para entende-lo em sua íntegra, possibilitando a reestruturação do código.

A finalidade dessa reestruturação preliminar é produzir um programa fonte mais fácil de analisar, entender e reestruturar. Em seguida, o código fonte produzido é analisado e são construídas representações do mesmo em níveis mais altos de abstração. Com base nisso, pode-se prosseguir com os passos de reestruturação, re-projeto e redocumentação, que são repetidos quantas vezes forem necessárias para se obter o sistema totalmente reestruturado.

Pode-se, então, gerar o programa na linguagem destino e dar seqüência aos testes que verificarão se a funcionalidade não foi afetada.

É importante salientar também que entre as diversas modificações que serão realizadas no código legado deverão ser aplicadas verificações para assegurar a manutenção de sua funcionalidade original.

Na literatura, existem diversos métodos para a realização de engenharia reversa e alguns desses já possuem ferramenta auxiliada por computador, mas essas ferramentas necessitam da intervenção de um especialista, uma vez que existem conceitos semânticos que estão implícitos no código fonte e que precisam ser recuperados para que o modelo de análise seja construído completamente.

Neste trabalho, o foco principal é o que Biggerstaff chama de *programming concepts*, ou seja ferramentas de engenharia reversa que tem como base de informações apenas o código fonte dos sistemas, ao contrário de outros trabalhos como Prado (1992) e Leite (1995) em que a fase de engenharia reversa é auxiliada por outros tipos de documentos e por experimentos práticos, *human concepts*, segundo Biggerstaff (1994).

Nesta etapa do trabalho, a proposta de reconstrução do software pela migração direta do código fonte da linguagem COBOL para a linguagem PROGRESS possui seis etapas (conforme mostra a Figura 3.1):

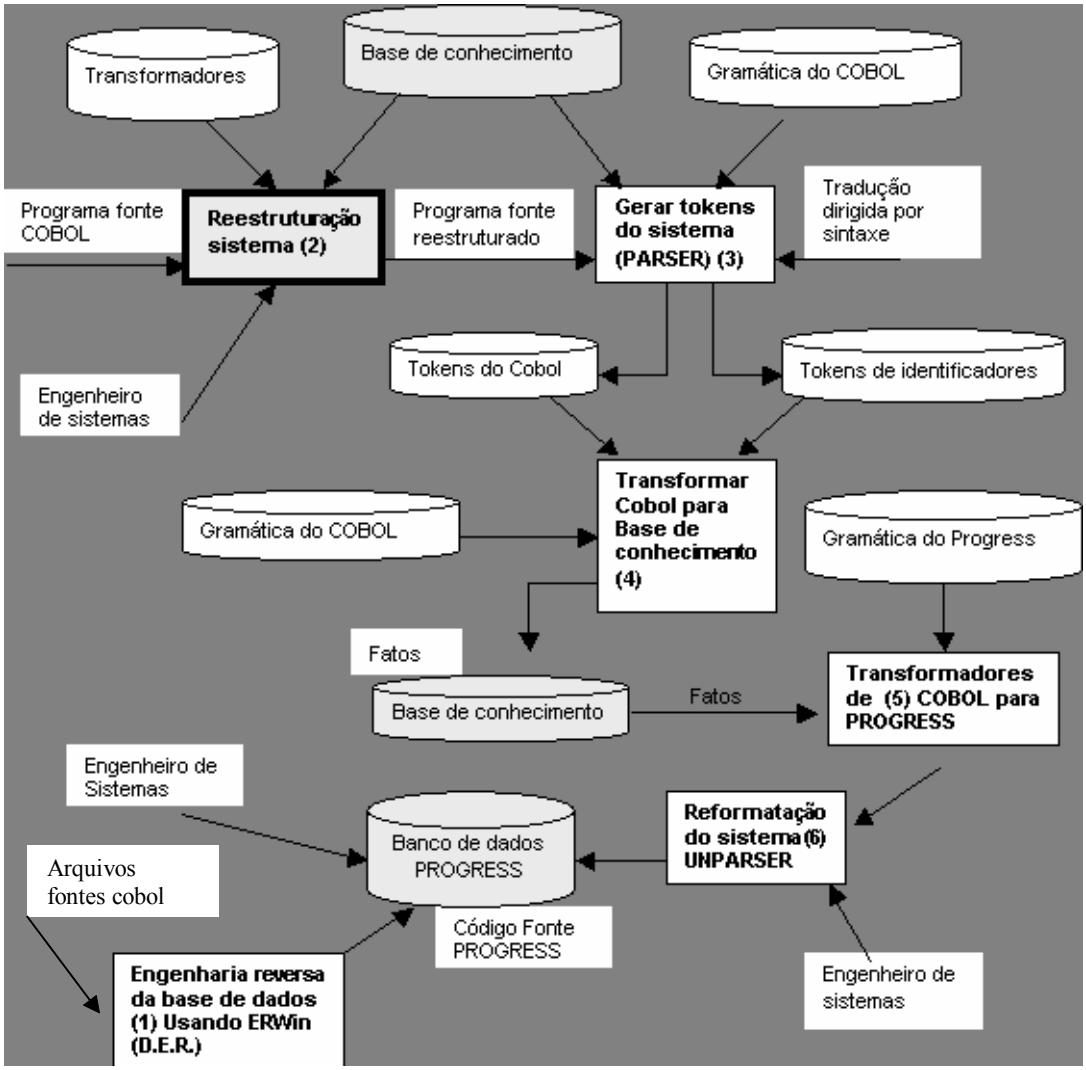


Figura 3.1: Processo de migração de um sistema legado Cobol para Progress

1) *Engenharia reversa da base de dados* – uso da ferramenta ERWin para obtenção do modelo conceitual da(s) base(s) de dados usada(s) pelo sistema. Esta fase foi inserida ao final da Figura porque, apesar de ser a primeira fase realizada, é apenas utilizada quando da execução da fase Reformatação do sistema (6). Nesta etapa, os programas fontes foram utilizados para verificação dos arquivos existentes e suas definições, a documentação do sistema não foi utilizada, pois não havia confiabilidade dos dados ali constantes. Com as definições dos arquivos, nos programas, estes foram redefinidos na base de dados do software ERWin que após o término das definições, foi executado o programa de engenharia reversa da própria ferramenta que gerou o diagrama de entidade e relacionamento (D.E.R), conforme ilustrado na Figura 3.2. Esta etapa foi realizada manualmente.

2) *reestruturação do sistema* - O código legado, escrito de forma procedural, é organizado sem prejuízo da sua lógica e semântica, de forma a facilitar sua transformação. A entrada para reengenharia deve ser o código-fonte original. Informações disponíveis sobre o código legado, caso existam, são também utilizadas neste passo para facilitar a reestruturação.

Nesta etapa deve-se então *reestruturar o código origem* para que fique com uma estrutura similar ao código-fonte destino, o que deve facilitar o trabalho de migração, solucionando diferenças sintáticas entre o código-fonte origem e destino. Para

realização da reengenharia deve-se organizar o código-fonte origem de forma a padronizá-lo de acordo com a estrutura do código-fonte destino. Para que esta tarefa seja realizada, o engenheiro deve conhecer o código origem e o código destino. Esta organização não deve prejudicar a lógica e a semântica dos programas fontes, de forma a facilitar sua transformação. No entanto, a base de conhecimento não foi totalmente implementada para o protótipo e desta forma o processo de reestruturação foi realizado manualmente. Esta base de conhecimento, quando implementada, deverá conter fatos e regras para auxiliar o processo de reestruturação .

3) *Gerar tokens do sistema (PARSER)*, responsável por analisar os programas da linguagem (COBOL), associado a sua gramática e gerar duas representações internas uma com os tokens da linguagem e outra com os identificadores correspondentes. Esta etapa foi realizada automaticamente.

4) *Transformar Cobol para base de conhecimento* – Esta etapa consiste em gerar fatos e regras em uma base de conhecimento com a finalidade de resolver problemas referentes a diferenças sintáticas entre Cobol e Progress. Foi realizada de forma automática, no entanto, algumas atividades foram feitas anteriormente, e de forma manual, para que esta etapa pudesse ser realizada com sucesso. Atividades estas como: definição das correspondências gramaticais entre o COBOL e PROGRESS.

5) *Transformadores de cobol para progress* - Esta etapa é utilizada para mapear estruturas da linguagem origem em estruturas da linguagem destino. Este mapeamento deverá ser realizado considerando os tokens da linguagem de origem e os tokens da linguagem destino. Estes transformadores também devem dispor de uma base de conhecimento, que permite gerar e consultar fatos. Etapa executada automaticamente.

6) *reformatação do sistema (também chamada UNPARSER)* - Etapa de migração que deverá fazer a formatação da linguagem destino tornando-a novamente textual, esta etapa foi executada automaticamente.

Na Figura, os retângulos representam as atividades (ou etapas). Retângulos com bordas finas - ver p.ex. etapa (4) Transformar Cobol para base de conhecimento - representam atividades realizadas automaticamente com uso de ferramentas; retângulos com bordas grossas - ver p.ex. etapa 2) Reestruturação do sistema – representam atividades a serem realizadas manualmente pelo analista responsável pela migração.

Deve-se notar que, conforme Figura 3.1, as gramáticas das linguagens Cobol e Progress devem ser definidas previamente.

3.1.1 Especificação da sintaxe: visão geral

Esta definição envolve uma especificação da sintaxe e da semântica destas linguagens.

Uma linguagem de programação pode ser definida pela forma de seus comandos e estruturas de dados (a sintaxe da linguagem) e pela expressão de que os mesmos significam (a semântica da linguagem). Para especificar a sintaxe de uma linguagem, existe uma notação amplamente aceita, chamada de gramática livre de contexto ou BNF (para Forma de Backus-Naur) (AHO, 1995). Com as notações corretamente disponíveis, a semântica é muito mais difícil de se descrever do que a sintaxe.

Além de especificar a sintaxe das linguagens , uma gramática livre de contexto pode ser usada como auxílio para guiar a tradução de programas. Uma técnica de compilação,

orientada por gramática , conhecida como *tradução dirigida pela sintaxe,* é de muita ajuda na organização das partes da análise léxica e gramatical do compilador.

Para realização desta etapa, deve-se conhecer toda a sintaxe da linguagem, que descreve naturalmente a estrutura hierárquica de muitas instruções das linguagens de programação.

Por exemplo, um comando *if-else*, em C, possui a forma:

if (expressão) comando **else** comando

Ou seja, o comando é uma concatenação da palavra-chave **if** , um parênteses à esquerda, um parênteses à direita, um comando, a palavra-chave **else** e um outro comando. (Em C não existe a palavra-chave **then**). Usando-se a variável *expr* a fim de denotar uma expressão e a variável *cmd* para um comando (ou enunciado) esta regra de estruturação pode ser expressa como

Cmd → **if** (*expr) cmd*
else cmd

Onde a seta deve ser lida com "pode ter a forma". Tal regra é chamada de uma produção. Numa produção, os elementos léxicos, como a palavra-chave **if** e os parênteses, são chamados de *tokens*. As variáveis como *expr* e *cmd* representam sequências de *tokens* e são chamadas de não-terminais.

Uma gramática livre de contexto possui quatro componentes:

b) Um conjunto de *tokens,* conhecidos como símbolos terminais

c) Um conjunto de não-terminais

d) Um conjunto de produções, onde uma produção consiste em um não-terminal, chamado de *lado esquerdo* da produção, uma seta e uma sequência de *tokens*/ou não terminais, chamados de *lado direito* da produção.

e) Uma designação a um dos não-terminais como o *símbolo de partida.*

3.1.2 Protótipo de uma ferramenta para migração

Propõe-se neste trabalho, resumidamente, além das etapas de reestruturação e definição da base de dados em Progress que foi executada manualmente, o desenvolvimento de um protótipo de uma ferramenta para migração de forma automática de programas escritos em COBOL para programas em PROGRESS, composto basicamente de três programas:

1) geraTokens.cbl - gera tokens através da gramática livre de contexto do COBOL, associada a um conjunto de variáveis aos símbolos gramaticais que possibilita o armazenamento futuro no processo de reconhecimento. Teoricamente este processo é conhecido como *tradução dirigida pela sintaxe,* ou seja a gramática e o conjunto de regras semânticas constituem a definição dirigida pela sintaxe, a tradução é um mapeamento de entrada e saída. O esquema de tradução é uma gramática livre de contexto, na qual, fragmentos de programas , chamados de *ações semânticas,* são inseridos nos lados direitos das produções. Um esquema de tradução é como uma definição dirigida pela sintaxe, exceto que a ordem de avaliação das regras semânticas é explicitamente mostrada. A tradução dirigida por sintaxe é uma técnica que permite realizar tradução concomitantemente com a análise sintática. Ações semânticas são associadas às regras de produção da gramática de modo que, quando uma dada produção é processada, essas ações são executadas (PRICE, 2001) ;

2) coboltoKB - transforma cobol para base de conhecimento, que durante o processo de reconhecimento de palavras chaves bem como lexemes gera a base de conhecimento, utilizando os símbolos gramaticais associados à gramática e;
3) CobolToProgress -transforma COBOL para PROGRESS, nada mais é do que um conjunto de transformadores, que através da base de conhecimento gerada e da gramática do PROGRESS, gera uma linguagem textual em seu banco de dados PROGRESS.

Analisando ainda a Figura 3.1, estes três programas correspondem às etapas 3, 4, 5 e 6. Nota-se que as etapas 5 e 6 são realizadas por apenas um programa(cobolToProgress).

3.2 Engenharia Reversa

Da literatura de engenharia reversa, quatro idéias marcaram predominante nossa proposta , Freitas (1997):

- Produzir representações gráficas de estruturas estáticas de programas através do código fonte (CROSS, 1995). Estas representações estão detalhas na seção 3.3 reestruturação do sistema;
- Criar um método, com auxílio de ferramentas, para realizar engenharia reversa de programas Cobol (EDWARDS ,1993), conforme mostrado na Figura 3.2;
- Construir uma biblioteca de transformações que reconhecem elementos de arquitetura em código fonte (HARRIS, 1996), conforme detalhado na seção 3.5 transformar cobol para base de conhecimento; e
- Propor uma ferramenta em que o resultado da extração da informação é armazenada em uma base de conhecimento, posteriormente utilizada para montar diagramas de desenho (JARZABEK, 1995), detalhado na seção 3.6 transformadores de cobol para progress.

Com base na literatura e, principalmente, nessas idéias acima, acredita-se que uma ferramenta de engenharia reversa (a partir do código fonte) deve atender a quatro pontos fundamentais, que seriam: como extrair as informações do código, como armazenar estas informações, como visualiza-las, e como visualizar apenas uma parte das informações.

A primeira decisão dessa proposta foi a de utilizar um sistema transformacional na extração das informações Harris (1996) e Jarzabek (1995).

A segunda decisão foi a utilização de um banco de conhecimentos para armazenar as informações extraídas do código fonte Jarzabek (1995). Esta decisão teve os seguintes argumentos:

- Necessidade de se trabalhar com um grande volume de dados, pois diferentes visões do código fonte devem estar disponíveis.
- Necessidade de não impor nenhum limite quanto ao tamanho dos sistemas analisados, uma vez que o problema de engenharia reversa ocorre geralmente em grandes sistemas.
- Necessidade de acessar várias vezes as informações do sistema sem que seja necessário re-análise do código fonte.

A estratégia a ser utilizada para a engenharia reversa, etapa (1) da fig. 3.1, é a utilização do software ERWin para modelagem dos dados, ou seja, usando o Diagrama de Entidade Relacionamento (D.E.R.). A modelagem ER é feita pelo analista a partir da definição da base de dados (e dos arquivos, se for o caso) definida em Cobol.

O software ERWin estabelece uma conexão ativa e nativa entre o modelo de dados e o banco de dados que permite a realização de engenharia progressiva e reversa. Através dessa conexão, o ERWin automaticamente gera tabelas, *views*, índices, normas de integridade referencial (chave primária, chave estrangeira), *defaults* e restrições de domínio/coluna. O ERwin inclui um conjunto de modelos de *triggers* de integridade referencial otimizados e uma linguagem abrangente de macros que permite a

personalização de suas próprias *triggers* e *stored procedures*. A partir do modelo gerado e do código fonte em COBOL, deverá ser definido o banco de dados a ser utilizado pelo PROGRESS.

Combinando as técnicas de reengenharia transformacional do sistema e engenharia reversa utilizando a ferramenta ERWin, foi definida uma estratégia para reengenharia do código legado COBOL.

O arquivo exemplo, mostrado na fig. 3.3, refere-se a uma parte do cadastro de estoque de peças de uma concessionária Volkswagen (completo no anexo 3), definido em Cobol.

Após o processo de engenharia reversa, onde se obteve o diagrama de entidade relacionamento (D.E.R.), mostrado na Figura 3.2, do sistema de controle de estoque de peças com controle da movimentação de entrada e saída, foi definido o esquema do dicionário de dados no banco de dados *PROGRESS*, diretamente pelo analista e o resultado está ilustrado na fig. 3.4 – definição do esquema da base de dados *PROGRESS* (completo no anexo 4). Utilizando o diagrama de E.R para definição do esquema da base de dados Progress, a identificação das chaves primárias e secundárias ficou extremamente facilitada.

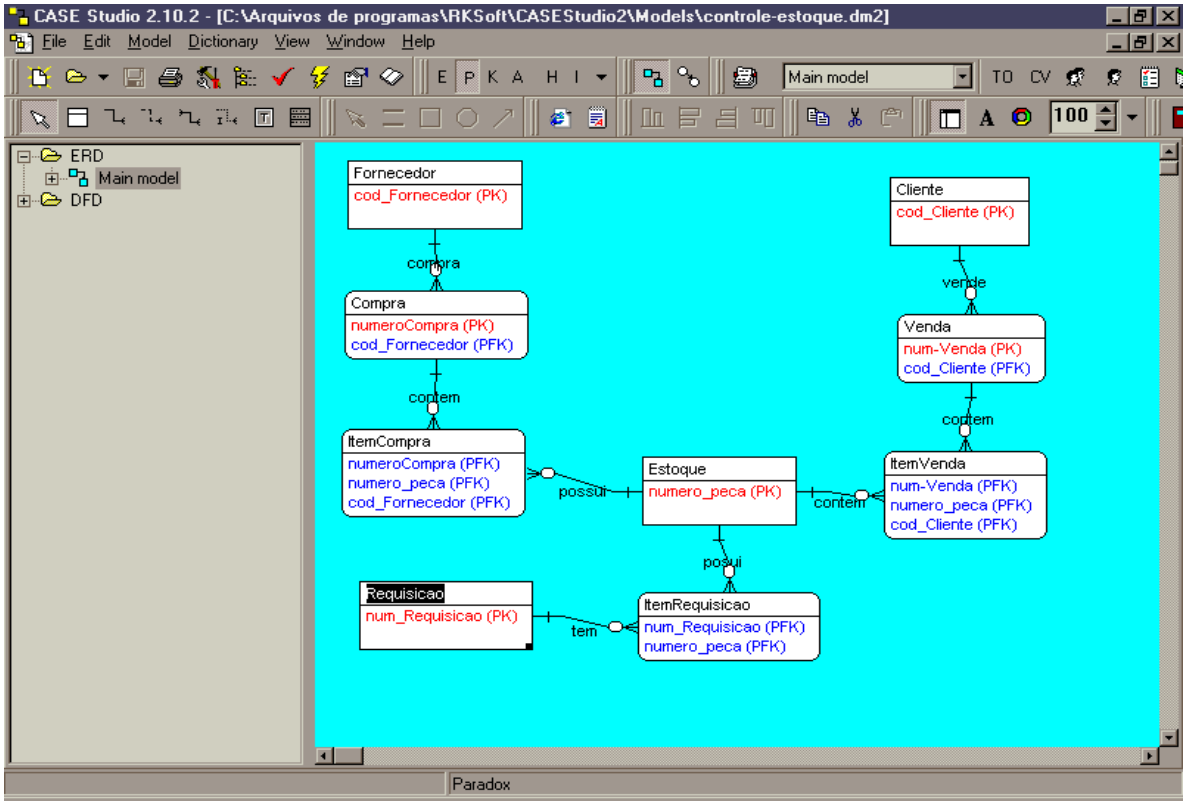


Figura 3.2: Diagrama de Entidade Relacionamento (DER) - exemplo

```
FD ARQ-ESTF001, RECORD 128.
    01 CADASTRO-GERAL-ESTOQUE-PECAS.
      02 F-NUMERO-PECA.              (Chave primária)
        03 FILLER          PIC X.
        03 F-FTIPO            PIC XXX.
        03 FILLER          PIC X(03).
        03 F-GRUPO-SUBGRUPO    PIC XXX.
        03 F-DESIGNACAO       PIC XXX.
        03 FILLER          PIC X.
        03 F-INDICE-MODIF      PIC XX.
        03 FILLER          PIC X.
        03 F-COR             PIC XXX.
      02 F-MODELO             PIC XXX.
      02 F-GRUPO             PIC XX.
      02 F-DESCRICAO          PIC X(30).
```

Figura 3.3: Exemplo de definição da base de dados em COBOL. (estoque de peças)

```
05/07/2002          PROGRESS Data Dictionary Report          Page   1
Database: teste                    estf001 File              (cadastro de pecas)
Frozen: no
Delete Validation
   Criterion:     Message:
   Field       Type  Ext Dec Format                    Init
   ----------- ----- --- --- ------------------------------ ----------
M     f-cod-fiscal int          9                          0
M     f-custo-med  dec          2 >>>,>>>,>>>,>>9.99             0
M     f-custo-rep  dec          2 >>>,>>>,>>>,>>9.99             0
M     f-descricao  char         x(30)
M     f-endereco   char         x(10)
      f-espaco     char         x(6)
      f-est-ant    dec          2 >>,>>9.9                       0
      f-est-max    int          99                         0
```

Figura 3.4: Exemplo de definição do esquema do dicionário de dados no banco de dados PROGRESS após aplicação da engenharia reversa

3.3 Reestruturação do sistema

A transformação automática de uma linguagem para outra, se feita de forma direta, na maioria das vezes gera código difícil de compreender e manter. A execução prévia do processo de reestruturação, etapa (2) da fig. 3.1, permite que essa transformação seja mais amena e o código gerado seja mais compreensível. Em primeiro lugar, é preciso especificar de maneira concisa a pseudo-linguagem a ser utilizada nas especificações das funções, e para isso é necessária a definição da gramática da linguagem.

Segue-se uma descrição mais detalhada das atividades realizadas neste passo, para a reestruturação do código legado.

```
PROCEDURE DIVISION.
    INICIO.
        DISPLAY SPACES.
        OPEN I-O ARQ-ESTF001.
        OPEN INPUT ARQ-ESTF006.
        OPEN I-O ARQ-ESTF003.
        OPEN INPUT ARQ-ESTF004.
    EXECUTA.
        READ ARQ-ESTF001 NEXT.
        IF STATUS-1 = "1"
                GO TO FIM.
        IF TIPO = "CTB"
                        GO TO EXECUTA.
    ATUALIZA-1.
        MOVE NUMERO-PECA TO NUMERO-P.
        READ ARQ-ESTF004 INVALID KEY
            GO TO EXECUTA2.
        MOVE QUANT-MINIMA-1    TO  UNIDADE-COMPRA.
        MOVE GRUPO-DESCONTO-1  TO  GRUPO-DESCONTO-COM.
    EXECUTA2.
        MOVE GRUPO-DESCONTO-COM TO CODIGO-DESCONTO.
        READ ARQ-ESTF006 INVALID KEY
                    GO TO GRAVA-F1.
```

Figura 3.5: Código origem antes da reestruturação

```
PROCEDURE DIVISION.
INICIO.
        DISPLAY SPACES.
        OPEN I-O ARQ-ESTF001.
    INICIO.
        PERFORM LEITURA-CADASTRO THRU FIM-LEITURA-CADASTRO
            UNTIL FINAL-LER = 1
          CLOSE ARQ-ESTF001 ARQ-ESTF003
              ARQ-ESTF006 ARQ-ESTF004
          STOP RUN.

   ***************************************************************
    LEITURA-CADASTRO.
        READ ARQ-ESTF001 NEXT AT END
            MOVE 1 TO FINAL-LER.
        IF TIPO NOT = "CTB"
            PERFORM VERIFICA-CAD-GERAL THRU FIM-VERIFICA-CAD-GERAL
            PERFORM VERIFICA-DESCONTO  THRU FIM-VERIFICA-DESCONTO
            PERFORM VERIFICA-PRECO     THRU FIM-VERIFICA-PRECO UNTIL
                NAO-DESCONTO = ZEROS
            PERFORM CALCULA-CUSTO      THRU FIM-CALCULA-CUSTO UNTIL
    EXIT.
```

Figura 3.6: Código origem após reestruturação

A fig. 3.5, mostra parte do código fonte antes da reestruturação. Nota-se, em negrito, os desvios não desejados através dos comandos " *go to"* . Logo após, a fig. 3.6 mostra parte do código reestruturado formando ninhos através da substituição dos comandos " *go to"* pelo comando "*perform"* , aninhando assim o conjunto de instruções para cada procedimento a ser adotado, também mostrado em negrito na Figura 3.6. A execução desta etapa foi possível através de um estudo detalhado do programa-fonte associado ao conhecimento do analista sobre a funcionalidade de cada programa. Estes códigos estão mais completos nos anexos 5 e 6, respectivamente às Figuras 3.5 e 3.6.

As alterações realizadas foram :

a) Eliminar os "*go to's*" , com a substituição por *PERFORM* , conseguindo desta forma encontrar os ninhos de procedimentos;

b) Tornar a linguagem mais próxima da linguagem destino, através de reestruturação do sistema ; e

c) Alterar nomes de variáveis, ou seja, quando a variável é correspondente a um arquivo, é alterada incluindo a letra ***"f-***", no início do nome da variável, significando uma variável de arquivo (*file*), conforme mostra a fig. 3.3, podendo desta forma tratar adequadamente as varáveis correspondentes a arquivos quando da transformação para PROGRESS.

Conforme mencionado na seção 3.2, produzir representações gráficas de estruturas estáticas de programas através do código-fonte, Cross (1995), isto foi possível porque quando se faz um trabalho de reestruturação do programa-fonte tornando-o estruturado, possibilita ao analista a visualização dos diagramas de fluxo de dados, porque os processos, os fluxos de informações e as bases de dados, aparecem nitidamente, conseguindo desta forma a representação gráfica conforme ilustra a Figura 3.7. Esta representação se refere ao programa ilustrado na fig 3.5 e 3.6. Por exemplo, este programa, Figuras 3.5 e 3.6 efetua o cálculo do custo de reposição, utilizando um arquivo de preços da fábrica Volkswagen (estf004), o arquivo de cadastro de peças (estf001), o arquivo de preços (estf003) e o arquivo de índices de descontos (estf006). Nnota-se que após o processo de reestruturação conseguimos visualizar a execução de 5 processos: leitura cadastro, verifica cadastro geral, verifica desconto, verifica preço e calcula custo. No entanto, este processo foi executado manualmente.

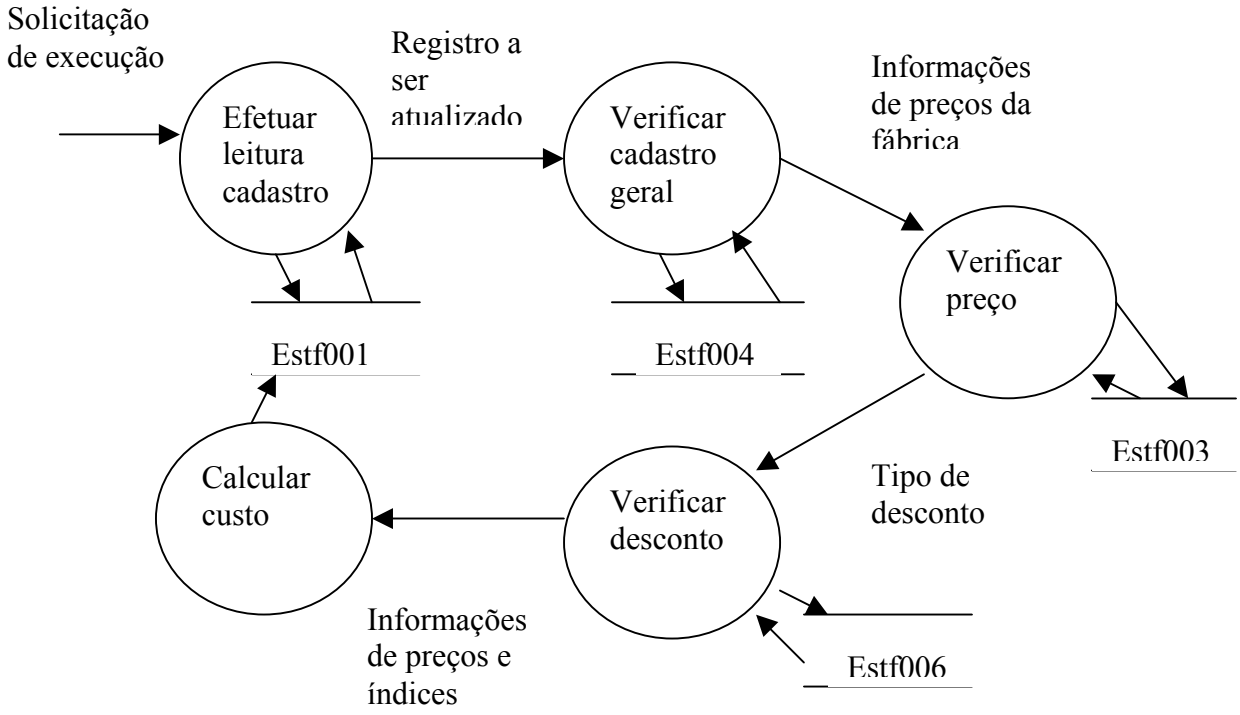


Figura 3.7: Diagrama de fluxo de dados do programa calcular custo de reposição

3.4 Gerar tokens do sistema

Um parser é responsável por analisar os programas da linguagem e gerar representações internas. A gramática é quem descreve as palavras válidas (lexemes) e suas funções dentro de sentenças de uma linguagem.

Uma sentença corresponde a uma sequência de lexemes que têm sentido na linguagem.

Uma gramática é composta de várias produções que representam de uma maneira geralmente recursiva a boa ordenação dos lexemes em uma sentença. A Figura 3.8 mostra partes da gramática da linguagem origem COBOL, e a Figura 3.9, mostra partes da gramática da linguagem destino PROGRESS, a correspondência entre as gramáticas está representada em negrito na Figura 3.8. Definida a gramática, pode-se então construir as principais partes do tradutor. Este terá dois analisadores: o Léxico e o Sintático.

O Analisador Léxico é a parte do tradutor responsável pelo reconhecimento dos lexemes, que são as menores partes de uma sentença de uma linguagem. Cada vez que o analisador Léxico reconhece um lexeme, ele envia um código, dito "token", a outra parte do tradutor, chamada de Analisador Sintático. Conforme ilustrado na etapa 3 da fig. 3.1, a próxima fase é a execução de um programa (`geraTokens.cbl`) que lê o programa fonte e a cada lexeme reconhecido, gera um token com o endereço correspondente e gera também um token correspondente para os identificadores com os mesmos endereços para que não haja perda dos mesmos.

A Figura 3.10 mostra a saída da execução deste programa com os tokens e endereços gerados. O Analisador Sintático é responsável por gerar a árvore sintática segundo as produções da gramática.

Os parsers do Cobol e Progress foram gerados através dos analisadores léxicos e das gramáticas livre de contexto através do uso do gerador de compiladores PCYACC (veja http://www.abraxas-software.com/ para maiores informações) As Figuras 3.8 e 3.9 mostram partes das gramáticas Cobol e Progress nas quais podem ser observadas, ao lado das regras, as ações semânticas (*READ file_name, INTO, etc).*

Aplicando o gerador de parsers (`geraTokens.cbl`), sobre o código-fonte Cobol e sua gramática obtemos um parser do Cobol. Nesta fase, também foi utilizado um gerador de fatos (CobolToKb) para armazenar em uma *base de conhecimento,* fatos e regras para solucionar os problemas de diferenças sintáticas, que também podemos chamar de *pontos de controle* este procedimento está detalhado na seção 3.5 transformar Cobol para base de conhecimento.

A Figura 3.10 mostra trechos do parser do programa origem Cobol gerado.

```
Add_statement              {source-computer}
  : add_statement_format1
  | add_statement_format2
  | add_statement_format3
  ;

add_statement_format1      {source-computer}
  : ADD  add_operand1s  TO  add_operand2s
  | ADD  add_operand1s  TO  add_operand2s  add_tail
  | ADD  error          TO  error
  | ADD  add_operand1s  TO  error          add_tail
  ;

add_operand1s    {object-computer}
  :                add_operand1
  | add_operand1s      add_operand1
  ;
/*
read_statement             {for-each-opt,$find-stat}  :
READ file_name
  | READ file_name           RECORD
  |  READ  file_name         NEXT        RECORD      |
read_statement     into_phrase
  | read_statement     end_phrase
  | read_statement     read_key_phrase
  | read_statement     invalid_key_phrase
  | read_statement     not_invalid_key_phrase
  | read_statement     END_READ
  ;

into_phrase        {corresponding}
  : INTO             Identifier
  ;

end_phrase         {end}
  : NOT   AT   END     statements
  |       AT   END     statements
  | NOT        END     statements
  |            END     statements
  ;

read_key_phrase    {find_stat where}
  : KEY  IS           Identifier
  | KEY               Identifier
  | KEY  IS           error
  | KEY               error
  ;

/*
```

Figura 3.8: Gramática do COBOL(trecho)

```
for_opt        : 'EACH'
               | 'FIRST'
               | 'LAST'
               | 'NEXT'
               | 'PREV'
               | 'CURRENT'
                        ;
open_opt       : tuning_phrase
               | 'INDEXED-REPOSITION'
               | 'MAX-ROWS' .sp NUM
                        ;
for_each_stat          :   'FOR'   .sp   record_item++','   (.nl
for_each_opt)*  ponto_ponto .nl
                 body_structs* .nl end_stat
                        ;
                        ;
for_each_opt    : tuning_phrase
               | 'BREAK'
               | 'BY' .sp name ('.sp DESCENDING')?
               | 'WHILE' .sp expr
               | 'TRANSACTION'
               | on_error_phrase
               | frame_phrase
                        ;
find_stat      : 'FIND' .sp record_item++',' '.'
                        ;
display_stat     : 'DISPLAY'  stream_opt?  unless_hide_opt?
display_list except_opt?
                 (.sp disp_options)*  '.'
                        ;
display_list           : (.sp constant display_field*)+
```

FIGURA 3.9: Gramática do PROGRESS(trecho)

```
token =   [ 411 ]          [IDENTIFICATION ]
token =   [ 352 ]            [DIVISION ]
token =   [  46 ]              '.' ]
token =   [ 498 ]            PROGRAM-ID ]
token =   [  46 ]              '.' ]
token =   [ 258 ]            Identifier ]
token =   [  46 ]              '.' ]
token =   [ 382 ]            ENVIRONMENT ]
token =   [ 352 ]            DIVISION ]
token =   [  46 ]              '.' ]
token =   [ 325 ]            DATA ]
token =   [ 352 ]            DIVISION ]
```

Figura 3.10: Trecho do parser gerado do programa fonte cobol

3.5 Transformar Cobol para base de conhecimento

O funcionamento do transformador Cobol para Progress é detalhado na Figura 3.14, nota-se que para cada comando, existe um transformador.
A *base de conhecimento* gerada, etapa 4 da fig. 3.1, para armazenar fatos e regras usa uma estrutura de fatos similar à da linguagem Prolog e está ilustrada na Figura 3.13. O esquema da base de conhecimento que orienta a geração dos fatos é definido previamente.

O código COBOL reestruturado do sistema, os tokens e a gramática do Cobol servem de entrada na atividade gerar base de conhecimento ("GERAR KB" na fig 3.14) que usa o transformador *cobolToKB* ("Transformar" na fig 3.14) para reconhecer variáveis , tipos de dados, performes e arquivos do COBOL, para instruir a criação destes em PROGRESS conforme ilustrado na Figura 3.14 .

Na saída deste passo tem-se fatos como: variável(nome, tipo, controle)., arquivo(nome, tipo)., variável_arquivo(nome, arquivo)., perform(nome_parag, controle)., .. e outros , chamado aqui neste trabalho de base de conhecimento, os quais serão usados no próximo passo de transformação de cobol para progress. A Figura 3.11 mostra parte do transformador CoboltoKB, em que a transformação Declara_Tipo_int, reconhece a declaração de uma variável do tipo inteiro, e armazena na base de conhecimento o fato variavel(nome, tipo, controle), usando o comando **assign** (**x**) no ponto de controle Ponto_Marca, onde **x** é uma string contendo o fato ou regra. Após analisar todo o programa COBOL segmentado, usa-se o comando "*assign*", para gravação da base de conhecimento denominada Utilitario.kb.

Do transaction: **Declara_Tipo_int**
Lhs:{{cobol.assgnment [[ID nome_var]] = 0 }}
Ponto_Marca: {{Progress. Compilacao-inicial
update ("variavel([[nome_var]],int,0)")
find Utilitario.kb where Uponto_marca = Ponto_marca no-error
if not available
Assign("variavel([[nome_var]],int,0)")
End.
End.
Figura 3.11 Transformador **cobolToKB**

A definição do esquema da base de conhecimento *utilitário.kb* está apresentada na Figura 3.12.

campo	tipo	tamanho	descrição
endereço	numérico	5	chave de acesso
comando	alfa	16	tipo de comando
comando-correspondente	alfanumerico	16	comando correspondente
nome	alfanumerico	16	nome da variável ou arquivo

Figura 3.12 Esquema da base de dados de **utilitário.kb**

A criação de uma *base de conhecimento,* contendo regras*,* conforme mostra a fig. 3.13, tornou-se necessária, porque a linguagem origem e destino possuem algumas diferenças sintáticas, solucionando assim este tipo de problema. Estas diferenças estão ilustradas nas tabelas 1 e 2 (comandos condicionais Cobol e comandos condicionais Progress respectivamente). Por exemplo, observando a Figura 3.13, a regra número 1 `Read_statement` (de acordo com a gramática do cobol) diz respeito a uma leitura seqüencial correspondente a um `for_each_opt` na gramática do progress e o argumento `$file_name` corresponde ao nome do arquivo lido.

Esta etapa de transformação visa extender as gramáticas do Cobol e Progress, associando às produções da gramática Cobol os símbolos gramaticais e ações semânticas correspondentes da gramática Progress, com o objetivo de instruir a transformação de Cobol para Progress.

Tabela 3.1: comandos de Condições em Cobol

:IF
[IF] <condição> <comando> **(1)**
[IF] <condição> <comando> [else] <comando> **(2)**
[IF] <condição> [next sentense] **(3)**
[IF] <condição> [next sentense] [else] <comando> **(4)**

Tabela 3.2: comandos de Condições em Progress

[IF] <condição> [then] [do:] **(1)**
[IF] <condição> [then] [do:] <comando> [end.][else] [do:] <comndo> [end] **(2)**
[IF] <condição> [then] [do:] [end] **(3)**
[IF] <condição> [then] [do:] [end] [else] [do:] <comando> [end] **(4)**

```
1   Read_statement(for_each_opt, $file_name:)

2   Read_key (find_stat, $file_name)

3   Perform ($nome_rotina,  $command)

4   Move_condition($var1, $var2, $var3,.. varn)

5   Compute_claus($var1, $var2, $computação)
```

Figura 3.13 Base de conhecimentos – exemplo de regras semânticas

A base de conhecimento gerada nesta etapa contém a ordem de execução das ações semânticas, as ações semânticas (Cobol), ações semânticas correspondentes (Progress) e um conjunto de regras de produção conforme ilustrado na Figura 3.13. Para tornar as ações semânticas mais efetivas, facilitando o processo de reconhecimento foi associado à gramática do Cobol as ações correspondentes em Progress em negrito conforme ilustrado na Figura 3.8, de modo que, quando uma dada produção é processada, essas ações são inseridas na base de conhecimento. O procedimento adotado para gerar esta base de conhecimento foi a utilização do transformador CoboltoKB, que faz parte da ferramenta implementada ilustrado na Figura 3.11, que reconhece tipos de dados e instruções do cobol e através da gramática correspondente, gera automaticamente os símbolos gramaticais e ações semânticas na base de conhecimento. Para cada token, verifica sua semântica, na gramática do cobol e associa a produção correspondente em progress na base de conhecimento. Toda vez que uma regra de produção é usada no processo de reconhecimento de uma sentença, os símbolos gramaticais dessa regra são "alocados" juntamente com os seus atributos,conforme Price (2001). E observando a Figura 3.11 (CobolToKb) nota-se que este programa efetua leituras seqüenciais aos tokens. Este reconhecimento é feito através do tratamento que o programa CobolToKb faz sobre os tokens gerados e os símbolos gramaticais associados (montado manualmente na gramática do Cobol), por exemplo; se o token for correspondente a um read , mais duas leituras aos tokens deve ser realizada, uma para identificar o nome do arquivo e outra para identificar o tipo de leitura (seqüencial ou indexada), possibilitando assim a pesquisa seqüencial a gramática do Cobol, e gerando na base de conhecimento a ordem da execução, o comando , no caso read_statement (leitura seqüência) ou read_key (leitura indexada), o nome do arquivo e o símbolo associado a gramática, no caso for_each_opt (leitura seqüencial) ou find_stat (leitura indexada) . Estes símbolos associados a gramática do COBOL servem para descrever o conjunto de lexemes que podem representar um token particular nos programas-fonte. E então associamos informações a uma contrução de linguagem de programação, mais precisamente o programa CobolToKb, atrelando os atributos aos símbolos gramaticais que representam a construção. O resultado deste exemplo está claramente representado nas Figuras 3.15 e 3.16 (a) em negrito, um read arq-estf001 next at end (leitura seqüencial), transformado em for each estf001.

3.6 Transformadores de Cobol para Progress – Reformatação do sistema

Um transformador implementa um conjunto de transformações. Uma transformação é composta basicamente por um padrão de reconhecimento que define a sintaxe da linguagem fonte origem para transformação e o padrão de substituição, que define a sintaxe da linguagem fonte destino para a transformação, etapa 5 da fig. 3.1. Além dos padrões de reconhecimento e substituição, uma transformação pode conter outros pontos de controle, aos quais pode-se associar o código para o desempenho das tarefas relacionadas com pré e pós-condições das transformações. A Figura 3.14 mostra o esquema de transformação de Cobol para Progress.

Na atividade **transformadores de Cobol para Progress** o código COBOL reestruturado do sistema é reimplantado usando o transformador externo COBOL_para_PROGRESS, que faz o mapeamento semântico da linguagem COBOL para a linguagem PROGRESS utilizando os fatos e regras que foram armazenados na base de conhecimento. A semântica do código COBOL segmentado é preservada a cada regra gramatical do código PROGRESS obtido. A garantia da semântica vem do tratamento com diferentes níveis de granularidade na definição das transformações, onde cada comando COBOL é transformado para o seu correspondente em PROGRESS.

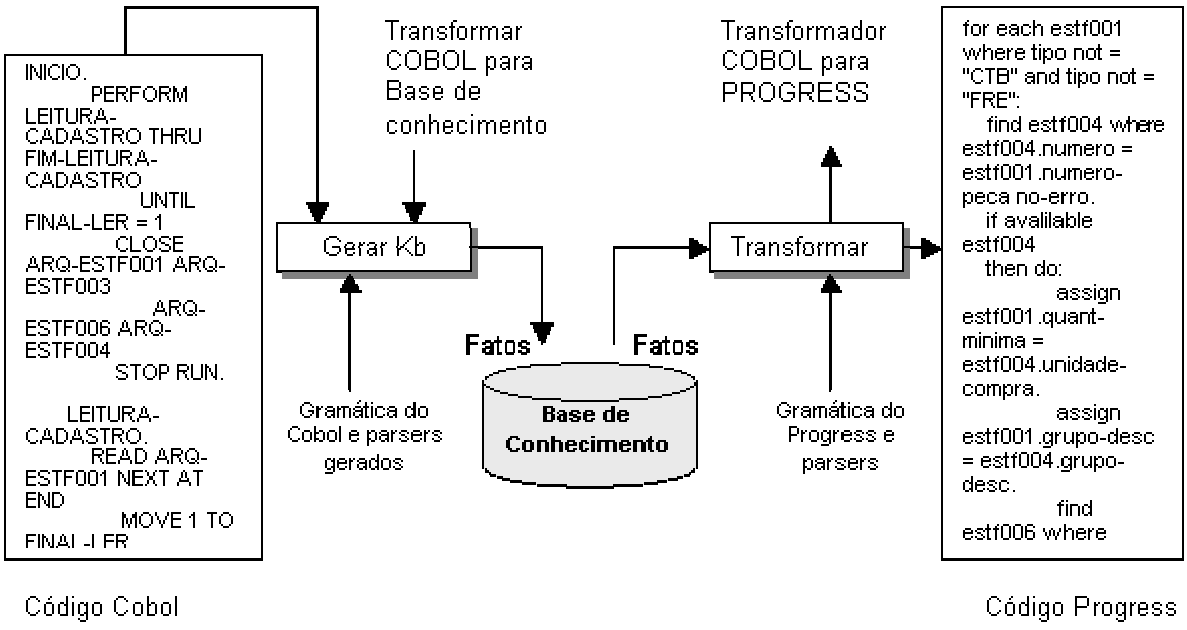


Figura 3.14: Transformador de Cobol para Progress

Para fazer as transformações automaticamente, podemos considerar o transformador `CobolToProgress` mostrado parcialmente na Figura 3.17. No ponto de controle `inicializacao_global`, o arquivo `Utilitario.kb`, gerado pelo transformador `CobolToKB`, é lido através do comando ***find*** (`area 1.1 da Figura 3.17`).

Num primeiro nível tem-se a transformação ***Output to nome-programa.Utilitario.kb*** (area 1.1 da Figura 3.17). para a criação do programa a ser transformado, em seguida temos os tratamentos de reconhecimentos e transformação da estrutura do programa COBOL para PROGRESS.

```
INICIO.
       PERFORM LEITURA-CADASTRO THRU FIM-LEITURA-CADASTRO  UNTIL FINAL-LER = 1
          CLOSE ARQ-ESTF001 ARQ-ESTF003 ARQ-ESTF006 ARQ-ESTF004
          STOP RUN.
    LEITURA-CADASTRO.
      READ ARQ-ESTF001 NEXT AT END       (a)
          MOVE 1 TO FINAL-LER.
      IF TIPO NOT = "CTB"
          PERFORM VERIFICA-CAD-GERAL THRU FIM-VERIFICA-CAD-GERAL
          PERFORM VERIFICA-DESCONTO  THRU FIM-VERIFICA-DESCONTO
          PERFORM VERIFICA-PRECO  THRU FIM-VERIFICA-PRECO NTIL  NAO-DESCONTO = ZEROS
          PERFORM CALCULA-CUSTO  THRU FIM-CALCULA-CUSTO  UNTIL NAO-PRECO = ZEROS.
    FIM-LEITURA-CADASTRO.
    EXIT.
    VERIFICA-CAD-GERAL.
      MOVE F-NUMERO-PECA TO F-NUMERO-P.
      READ ARQ-ESTF004 INVALID KEY
           MOVE ZEROS TO F-QUANT-MINIMA-1
           MOVE SPACES TO F-GRUPO-DESCONTO-1.
      MOVE F-QUANT-MINIMA-1 TO  UNIDADE-COMPRA.
      MOVE F-GRUPO-DESCONTO-1  TO  GRUPO-DESCONTO.
    FIM-VERIFICA-CAD-GERAL.
    EXIT.
```

Figura 3.15: Código fonte cobol reestruturado

```
for each estf001:                (a)
  if tipo not = "CTB":
  then do:
   find estf004 where estf004.numero = estf001.numero-peca no-erro.
   if avalilable estf004
   then do:
      assign   estf001.quant-minima = estf004.unidade-compra.
        assign   estf001.grupo-desc = estf004.grupo-desc.
        find estf006 where estf006.grupo-desconto =
                      estf004.grupo-desconto no-error.
        if avalilable estf006
        then do:
            find estf003 where estf003.peca = estf004.numero no-error
            if available estf004
            then do:
                assign estf003.preco-revendedor = estf003.preço-publico * estf006.índice.
                assign estf001.custo-reposicao  =
                       estf003.preco-revendedor.
           end.
       end.          end.
   end.
end.
```

Figura 3.16: Exemplo de Código fonte Progress após migração

O conjunto de transformações **export**, é utilizado para criar as linhas de comando correspondente em Progress, dentro do programa criado pela cláusula **output** (área 1.1 da Figura 3.17), através da transformação `if utilitário.comando = "read_statement"` (área 1.2 da Figura 3.17)**,** reconhece uma instrução de leitura seqüencial (ver fig.3.15 (a) *read arq-estf001 next at end*) de um arquivo e o transforma na leitura corresponde em progress, acessando na base de conhecimento

(utilitário.comando_corresponde) fig. 3.13 e gera a instrução correspondente em progress, conforme mostra o exemplo da fig.3.16 (a) `for-each.`
As Figuras 3.15 e 3.16 mostram os códigos Cobol reestruturado e Progress após migração respectivamente.

```
        Inicialização_global:
        Find utilitário.kb where enderecoUutilitário.kb = "1"
1.1             no-error.
        If available Utilitário.kb
          Output to  nome-programa.Utilitario.kb.
        for each Utilitário.kb where endereço.Utilitario.kb not = "1":
1.2             if commando.Utilitário = "read_next"
                then do:
                        export commando.Programa=
                                comando_corresponde.Utilitário.kb
                           file_name.Programa =
                                        file_name.Utilitário.kb
                            instrução.Programa = instrução.Utilitário.kb.
                end.
        End.
        For each Utilitario.kb  where endereco.Utilitario not = "1":
1.3             if commando.Utilitário = "move"
                then do:
                        export $var2.Programa= $var1.Programa.
                end.
        End.
```

Figura 3.17: Trecho de código do Transformador CobolToProgress

Observou-se que os sistemas obtidos pela reengenharia são mais fáceis de manter em relação ao código legado Cobol. O código transformado Progress possui uma boa legibilidade e a funcionalidade do sistema se manteve. O maior problema foi quanto ao custo de aquisição do Progress e a performance que caiu razoavelmente devido ao banco de dados PROGRESS ser mais robusto em relação ao COBOL.

4 CONCLUSÕES

Esta conclusão se articula em 4 eixos: as contribuições, o processo de migração, as limitações e as perspectivas futuras de continuidade deste trabalho.

Os trabalhos relacionados na revisão bibliográfica, nos auxiliaram no sentido de identificar e organizar as tarefas a serem executadas para o cumprimento deste trabalho.

Contribuições

Este trabalho contribui para solução do problema de migração de sistemas legados, particularmente nos aspectos seguintes:

a) proposta de um processo de migração de sistemas legados envolvendo atividades como Engenharia Reversa do código fonte associada a uma atividade de entendimento de programas, Engenharia Reversa de bancos de dados e Reengenharia do sistema incluindo uma tradução automática do código legado (reestruturado); o texto procurou mostrar como este processo realiza a reengenharia dos programas legados e o uso das ferramentas , com ênfase na tradução automática de códigos.
b) desenvolvimento (definição e implementação) de uma ferramenta para assistir este processo e que inclui suporte às atividades de Engenharia Reversa (integrando a utilização da ferramenta ERWIN para conversão da base de dados) e tradução automática de programas escritos na linguagem Cobol para programas que usam o ambiente Progress;
c) modelagem e população de uma base de conhecimentos e regras de um domínio orientado para as linguagens Cobol e Progres, que pode obviamente ser reusada em domínios e linguagens similares.

O estudo de caso realizado para o processo de migração e para a validação da ferramenta de tradução foi um sistema real existente para suporte a vendas, compras e controle de estoque de peças no contexto de concessionárias e revendedoras autorizadas. Vale lembrar que uma grande vantagem do uso da transformação automática foi a facilidade de manutenção e do reuso das transformações do sistema escrito em Cobol. A reengenharia de sistemas para novas plataformas como Progress teve êxito em atualizar o software sem perda de sua funcionalidade. O estudo de caso demonstrou a viabilidade do processo e do uso da ferramenta construída.

O processo de migração

O processo de migração foi realizado com 50% das atividades de forma manual e 50% de forma automática.

A correspondência gramatical das linguagens COBOL e PROGRESS foram feitas em sua totalidade e de forma manual.

O processo de migração ocorreu da seguinte forma:

- Definição das tabelas do sistema no Banco de dados PROGRESS

- Identificação dos módulos iniciais a serem migrados, Os módulos iniciais identificados foram os correspondentes aos cadastros do sistema de controle de estoque
- Inclusão da base de conhecimento no banco de dados, contendo a correspondência gramatical das linguagens COBOL e PROGRESS , para facilitar a migração
- Execução da ferramenta de migração
- Avaliação dos resultados obtidos
- Acertos no código de migração

Após a migração dos módulos de cadastro do sistema de controle de estoque, os próximos módulos migrados foram correspondentes a movimentação do sistema, entrada e saída de produtos e finalmente os relatórios e consultas. Foram realizados vários testes do sistema para verificar sua integridade antes de liberá-lo aos usuários. Durante um período de 45 dias foi feito um acompanhamento do sistema com objetivo de identificar sua usabilidade corretitude, manutenibilidade, confiabilidade , integridade, eficiência e segurança. Os resultados foram satisfatórios tanto a nível de atendimento dos requisitos do sistema quanto a satisfação do usuário.

Limitações

O trabalho desenvolvido apresenta limitações quanto à flexibilidade para utilização de outras linguagens de programação origem e destino. Para mudanças destas linguagens seria necessária a redefinição das gramáticas, o que não é uma tarefa trivial embora seja possível.

A reestruturação do código Cobol também é um fator restritivo pois esta atividade não possui uma ferramenta associada desenvolvida. Esta reestruturação do código COBOL envolveu atividades como: a) padronização das variáveis correspondentes a arquivos com o objetivo de auxiliar a identificação na aplicação da ferramenta, b) eliminação de "GO TO's" e inclusão de PERFORMS com o objetivo de deixar o código origem estruturado também para facilitar a etapa de migração.

Uma limitação da ferramenta é sua integração com uma ferramenta específica para Engenharia Reversa de banco de dados, no caso ERWin.

Além disto também não houve um tratamento metodológico de migração de projeto de interfaces com usuário, porque o ambiente Progress faz a formatação das telas automaticamente a partir do modelo do banco de dados.

Perspectivas de trabalhos futuros

Como sugestão para futuros trabalhos e extensões, está a possibilidade de desenvolvimento de uma ferramenta genérica, podendo utilizar o processo aqui proposto, mas permitindo mais flexibilidade na escolha da linguagem a ser utilizada como origem e destino, Além disto, estudos da capacidade reflexiva de linguagens (CAMPO; PRICE, 1996), mostram que visualizações da dinâmica de um programa podem aumentar a compreensão desses programas. Dentro da atividade de engenharia reversa acreditamos, que conjuntamente com a abordagem aqui proposta, pode-se também investigar o uso da capacidade de reflexão para registro de comportamento como forma de ajuda à compreensão do software.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiladores** princípios, técnicas e Ferramentas**.** Rio de Janeiro, LTC, 1995

BERGMANN, U.; PRADO, A. F.; LEITE, J.C.S.P. Desenvolvimento de Sistemas Orientados a Objetos Utilizando o Sistema Transformacional Draco-PUC, Em **Anais do X Simpósio Brasileiro de Engenharia de Software**, Sociedade Brasileira de Engenharia de Software, José Carlos Maldonado. São Carlos: UFSCAR, 1996, p. 173-188.

BIGGERSTAFF, T. J.; MITBANDER, B.G.; WEBSTER, D.E. Program understanding and the Concept Assignment Problem. **Communications of The ACM**, New York, v.37, n.5, May 1994.

CAMPO, M.R.; PRICE, R.T. Um Framework Reflexivo para Ferramentas de Visualização de de Software**,** Em **Anais do X Simpósio Brasileiro de Engenharia de Software,** Sociedade Brasileira de Engenharia de Software, José Carlos Maldonado, ed., São Carlos, 1996, pp. 153 - -172.

CROSS, J. H.; HENDRIX T. D. Using Generalized Markup and SGML for Reverse Engineering Graphical Representation of Software. In: Second Working Conference on Reverse Engineering, **Proceedings** [S.l.:s.n.], 1995.

EDWARDS, H.E.; MUNRO, M.. R. Reverse Engineering from COBOL to SSADM. Working Conference on Reverse Engineering, **Proceedings** [S.l:s.n], Baltimore, May 1993.

FREEMAN, P. A Conceptual Analysis of the Draco Approach to Constructing Software Systems**.** IEEE Transactions on Software Enginnering**. Communications of the ACM.** New York, v.13, n.7, July 1987.

FREITAS, F.G.. **A Geração de Parsers da Máquina DRACO-PUC**, 1994. Trabalho Final de Curso – Engenharia de Computação; Departamento de Informática; Puc-Rio, Rio de Janeiro.

FREITAS, F.G. **VG Um visualizador gráfico genérico,** 1996. Projeto Final de Programação de Informática, Puc-Rio, Rio de Janeiro.

FREITAS, F.G. **Aplicando técnicas de reuso de software na construção de ferramentas de engenharia reversa**, 1997. Dissertação (Mestrado) – Departamento de Informática, Rio de Janeiro

GALL, M. J.; Klosch, G. T. evolution of legacy and emerging systems. In Second Working Conference On Reverse Engineering, Italy, 1995. **Proceedings** [S.l.:s.n].

HARRIS, D.R.A.S.Y.; REUBENSTEIN, H.B. Extracting Architectural Features from Source Code. Automated Software Engineering, **Communications of the ACM** .New York, v.3 n.1/2, June 1996.

JARZABEK, S.; KEAM, T. P. Design of a Generic Reverse Engineering Assistant Tool. In: Second Working Conference on Reverse Engineering, 1995.

Proceedings, [S.l.;s.n],1995.

LEITE, J.C. Working Results on Software Re-Engineering. Software Engineering Notes. **Communications of the ACM**, New York, v.21 n.2, March 1996.

LEITE, J.C.S.P.; PRADO, A.F.; SANT'ANNA, M. Draco-PUC, Experiências e Resultados de Re-Engenharia de Software, Em **Anais do VI Simpósio Brasileiro de Engenharia de Software,** Sociedade Brasileira de Engenharia de Software, Daltro Nunes. Gramado, 1992, p. 115-128.

LEITE, J.C.; SANT'ANNA, M.; FREITAS, F.G. Draco-PUC: a Technology Assembly for Domain Oriented Software Development. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 3., 1994, Rio de Janeiro. **Advances in software reusability.** Los Alamitos: IEEE Computer Socyety Press, 1994.

LEITE, J.C.; SANT'ANNA, M.; FREITAS, F.G. O Uso do Paradigma Transformacional no Porte de Programas Cobol, In: Simpósio Brasileiro de Engenharia de Software, SBES, 10., 1995. Recife. **Anais**, Jaelson Castro, 1995, p.397-414.

MAYHAUSER, A.V. Program Undaerstanding Behavior During Enhancement of large Scale Software**,** Jornal of software Maintenance: Research and Practice, **Communications of the ACM,** New York, v. 9, 1995

MAYHAUSER, A. VON. Program Undaerstanding Processes During Corretive

Mantenaence of large Scale Software**,** In**:** International Conference on Software mantenance, 1996. **Proccedings** [S.l:s.n.], Italy 1996.

NEIGHBORS, J. M. The Draco Approach to Constructing Software from Reusable Components. IEEE Transactions on Software Enginnering, **Communications of the ACM.** New York, v.10 n.5:564-574, Sep. 1984.

PCYACC. Disponível em: <http://www.abraxas-software.com/pcyacc.htm>. acesso em: mar. 2002.

PENTEADO, R.A.D.; GERMANO, F.S.R.; MASIERO, P.C. Engenharia Reversa Orientada a Objetos do Ambiente StatSim, In: IX Simpósio Brasileiro de Engenharia de Software, Sociedade Brasileira de Engenharia de Software, Recife, **Anais,** Jaelson Castro, Recife 1995, p. 345-362.

PENTEADO, R.A.D., GERMANO, F.S.R., MASIERO, P.C. Engenharia Reversa Orientada a Objetos do Ambiente StatSim, In: IX Simpósio Brasileiro de Engenharia de software, Sociedade Brasileira de Engenharia de Software, **Anais,** Jaelson Castro, Recife 1996.

PRADO, A. F. **Estratégia de Re-Engenharia de Software Orientada a Domínios.** 1992. Tese (Doutorado em engenharia de software) – Faculdade (Pontifícia Universidade Católica do Rio de Janeiro), PUC-Rio, Rio de Janeiro, 1992.

PRESSMAN, R.S. **Engenharia de Software**. São Paulo: Makron books, 1995.

PRICE, A.M. DE A. **Implementação de linguagens de programação:** Compiladores. 2. ed. Porto Alegre: Sagra Luzatto, 2001.

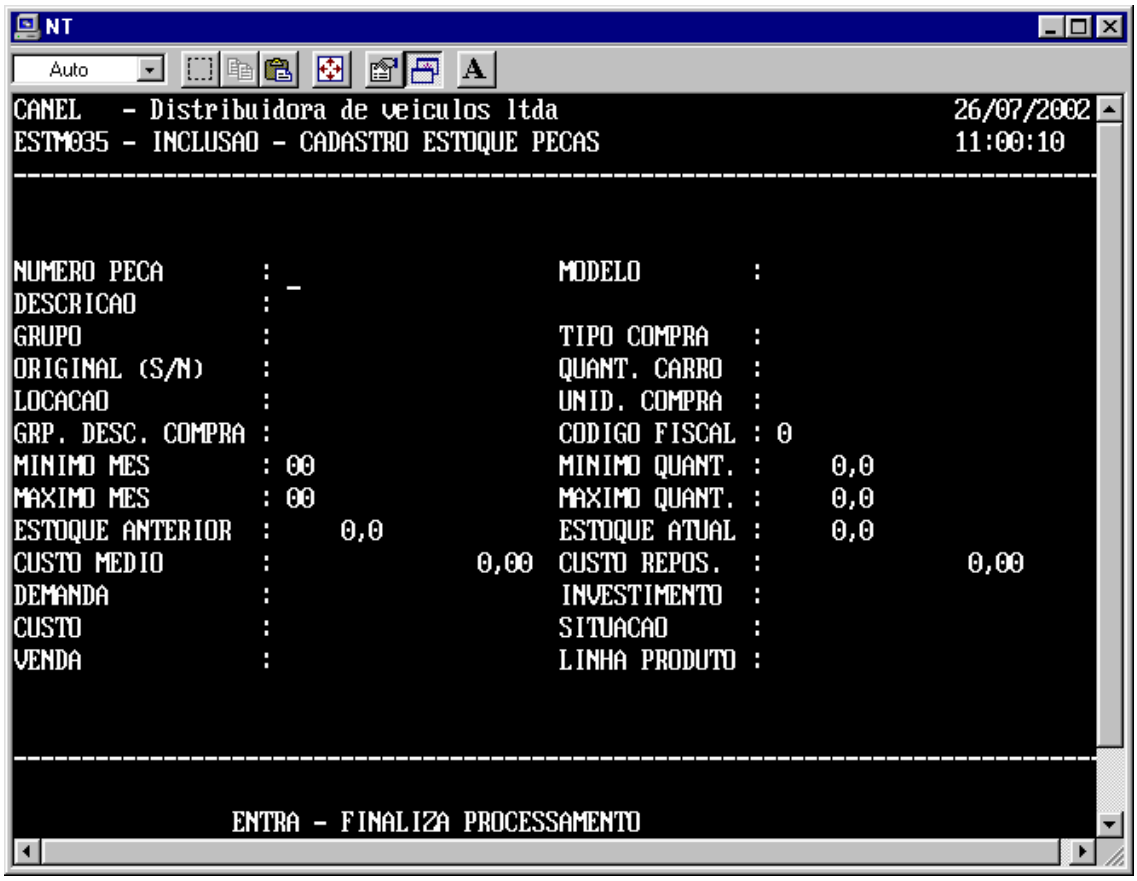
RUMBAUGH, J. et al. **Object Oriented Modeling and Design**. Englewood chffs: Prentice Hall, 1991

SELFRIDGE, P.G.; WALTERS, R.C.; CHIKOFSKY, E.J. Challenges to the Field of Reverse Engineering. In: Working Conference on Reverse Engineering, 1993. **Proceedings,** [S.l.:s.n], 1993.

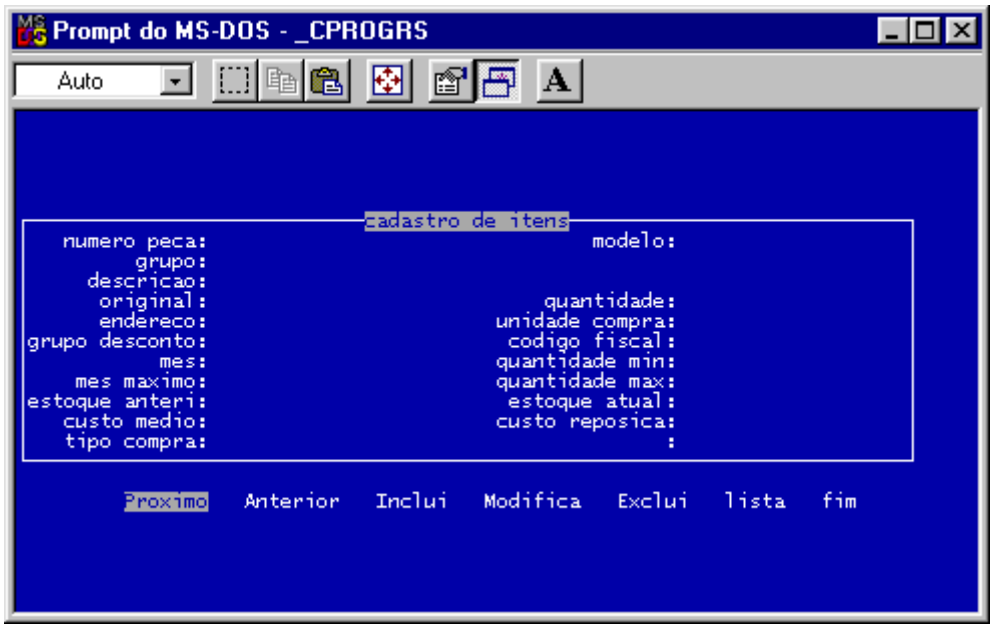
SMART, J. wxWindows User Manual. Edinburgh: University of Edinburgh, Artificial Applications Institute, .1995.

WILLS, L. M.; CROSS, J. H. II. Recent Trends and Open Issues in Reverse Engineering**.** Automated Software Engineering, **Communications of the ACM,** New York, v. 3 n.1/2, June 1996.

ANEXO A TELA DE CADASTRO DE PEÇAS EM COBOL



ANEXO B TELA DE CADASTRO DE PEÇAS EM PROGRESS



ANEXO C EXEMPLO DE DEFINIÇÃO DA BASE DE DADOS EM COBOL. (ESTOQUE DE PEÇAS)

```
FD ARQ-ESTF001, RECORD 128.
     01 CADASTRO-GERAL-ESTOQUE-PECAS.
       02 F-NUMERO-PECA.              (Chave primária)
         03 FILLER          PIC X.
         03 F-FTIPO           PIC XXX.
         03 FILLER          PIC X(03).
         03 F-GRUPO-SUBGRUPO    PIC XXX.
         03 F-DESIGNACAO       PIC XXX.
         03 FILLER          PIC X.
         03 F-INDICE-MODIF      PIC XX.
         03 FILLER          PIC X.
         03 F-COR             PIC XXX.
       02 F-MODELO              PIC XXX.
       02 F-GRUPO             PIC XX.
       02 F-DESCRICAO           PIC X(30).
       02 F-ORIGINAL-S-N        PIC X.
       02 F-QUANT-CARRO         PIC 999.
       02 F-ENDERECO            PIC X(10).
       02 F-UNIDADE-COMPRA      PIC 9(5).
       02 F-GRUPO-DESCONTO-COM  PIC X.
       02 F-CODIGO-FISCAL       PIC 9.
       02 F-ESTOQUE-MINIMO.
         03 F-MES-A           PIC 99.
         03 F-QUANTIDADE-A      PIC 9(5)V9.
       02 F-ESTOQUE-MAXIMO.
         03 F-MES-B           PIC 99.
         03 F-QUANTIDADE-B      PIC 9(5)V9.
       02 F-ESTOQUE-ANTERIOR    PIC 9(5)V9.
       02 F-ESTOQUE-ATUAL       PIC 9(5)V9.
       02 F-CUSTO-MEDIO         PIC 9(12)V99 COMP-3.
       02 F-CUSTO-REPOSICAO     PIC 9(12)V99 COMP-3.
       02 F-TIPO-COMPRA         PIC XX.
       02 FILLER            PIC X(06).
```

ANEXO D EXEMPLO DE DEFINIÇÃO DA BASE DE DADOS EM PROGRESS. (ESTOQUE DE PEÇAS) – APÓS APLICAÇÃO DA ENGENHARIA REVERSA

```
05/07/2002          PROGRESS Data Dictionary Report          Page   1
Database: teste                  estf001 File                (cadastro de pecas)
Frozen: no
Delete Validation
   Criterion:     Message:
   Field       Type  Ext Dec Format                    Init
   ----------- ----- --- --- ------------------------------ ----------
   M    f-cod-fiscal int         9                          0
   M    f-custo-med  dec         2 >>>,>>>,>>>,>>9.99       0
   M    f-custo-rep  dec         2 >>>,>>>,>>>,>>9.99       0
   M    f-descricao  char        x(30)
   M    f-endereco   char        x(10)
        f-espaco     char        x(6)
        f-est-ant    dec         2 >>,>>9.9                 0
        f-est-max    int         99                         0
   M    f-estoque-at dec         2 >>,>>9.9                 0
   M    f-gr-desc    char        x(1)
   M    f-grupo      char        x(2)
   M    f-mes-est-mi int         99                         0
   M    f-modelo     char        x(3)
*M    f-numero     char        x(20)
   M    f-original   logic       sim/nao                    no
   M    f-quant-car  int         999                        0
        f-quant-max  dec         2 >>,>>9.9                 0
        f-quant-min  dec         2 >>,>>9.9                 0
   M    f-tipo-compr char        x(2)
   M    f-un-compra  int         >>>>9                      0
   Field-Name   Label                    Col-label
   ----------- ------------------------- ---------------------------
   f-cod-fiscal codigo fiscal             codigo!fiscal
   f-custo-med  custo medio
   f-custo-rep  custo reposicao
   f-descricao  descricao
   f-endereco   endereco
   f-espaco
   f-est-ant    estoque anterior          estoque!anterior
   f-est-max    mes maximo                mes!estoque!maximo
   f-estoque-at estoque atual             estoque!atual
   f-gr-desc    grupo desconto            grupo!desconto
   f-grupo      grupo
   f-mes-est-mi mes                       mes!estoque!minimo
   f-modelo     modelo
   f-numero     numero peca               numero!peca
   f-original   original
   f-quant-car  quantidade
   f-quant-max  quantidade max            quantidade!maxima
   f-quant-min  quantidade minima         quantidade!minima
   f-tipo-compr tipo compra               tipo!compra
   f-un-compra  unidade compra            unidade!compra
   Database: teste   estf001 File   (cadastro de pecas)
     Index Name   Unique Field Name   Seq Ascending abbreviate
     ------------ ------ ------------ --- --------- ----------
   # numro              yes   f-numero
   Help Messages
   ------------------------------------------------------------------------------
   f-cod-fiscal :  digitar o codigo fiscal
   f-custo-med  :  digitar o custo medio
   f-custo-rep  :  digitar o custo reposicao
   f-descricao  :  digitar a descricao do produto
   f-endereco   :  digitar o endereco da peca na prateleira
   f-espaco     :  reservado
   f-est-ant    :  digitar o estoque anterior (mes)
   f-est-max    :  digitar o mes para estoque maximo
   f-estoque-at :  digitar o estoque atual
   f-gr-desc    :  digitar o grupo de desconto
   f-grupo      :  digitar o grupo "PM", "AC", "CL" ou "OM"
   f-mes-est-mi :  digitar o mes para estoque minimo
   f-modelo     :  digitar o codigo do modelo
   f-numero     :  digitar o codigo da peca
   f-original   :  digitar "s" para pecas originais e "n" para nao originais
   f-quant-car  :  digitar a quantidade minima para instalar no veiculo
   f-quant-max  :  digitar a quantidade maxima para o mes
   f-quant-min  :  digitar a quantidade minima para o mes
   f-tipo-compr :  digitar o tipo de compra
   f-un-compra  :  digitar a unidade de compra
              * - Indicates that a field participates in an index
              # - Indicates the primary index for a database file
              M - Indicates that a field is mandatory
```

ANEXO E CÓDIGO FONTE ANTES DA REESTRUTURAÇÃO

```
PROCEDURE DIVISION.
    INICIO.
        DISPLAY SPACES.
        OPEN I-O ARQ-ESTF001.
        IF FILE-STATUS NOT = "00"
                DISPLAY FILE-STATUS AT 1001
                DISPLAY "TERMINO ANORMAL ESTF001" AT 1101
                CLOSE ARQ-ESTF001
                STOP RUN.
        OPEN INPUT ARQ-ESTF006.
        IF FILE-STATUS NOT = "00"
                DISPLAY FILE-STATUS AT 2001
                DISPLAY "TERMINO ANORMAL ESTF006" AT 2101
                CLOSE ARQ-ESTF001
                STOP RUN.
        OPEN I-O ARQ-ESTF003.
        IF FILE-STATUS NOT = "00"
                DISPLAY FILE-STATUS AT 1501
                DISPLAY "TERMINO ANORMAL ESTF003" AT 1601
                CLOSE ARQ-ESTF003
                STOP RUN.
       OPEN INPUT ARQ-ESTF004.
       IF FILE-STATUS NOT = "00"
                  DISPLAY FILE-STATUS AT 1401
                  DISPLAY "TERMINO ANORMAL F4" AT 1501
                  CLOSE ARQ-ESTF003 ARQ-ESTF004
                  STOP RUN.
    ACCEPT DATA-W FROM DATE.
    ADD ANO-W TO ANO-C
    MOVE MES-W TO MES-C.
    MOVE DIA-W TO DIA-C.
    ******************************************************************
    EXECUTA.
       READ ARQ-ESTF001 NEXT.
       IF STATUS-1 = "1"
                GO TO FIM.
       IF TIPO = "CTB"
                     GO TO EXECUTA.
    ATUALIZA-1.
       MOVE NUMERO-PECA TO NUMERO-P.
       READ ARQ-ESTF004 INVALID KEY
          GO TO EXECUTA2.
       MOVE QUANT-MINIMA-1    TO  UNIDADE-COMPRA.
       MOVE GRUPO-DESCONTO-1  TO  GRUPO-DESCONTO-COM.
    EXECUTA2.
       MOVE GRUPO-DESCONTO-COM TO CODIGO-DESCONTO.
       READ ARQ-ESTF006 INVALID KEY
                 GO TO GRAVA-F1.
    EXECUTA3.
       MOVE NUMERO-PECA TO NUM-PECA.
       READ ARQ-ESTF003 INVALID KEY
                GO TO EXECUTA.
    CALCULA.
       MOVE PRECO-PUBLICO TO PRECO-PUBLICO-AUX.
       COMPUTE PRECO-AUX =  PRECO-PUBLICO-AUX * INDICE.
       MOVE PRECO-AUX TO CUSTO-REPOSICAO PRECO-REVENDEDOR.
       REWRITE LISTA-PRECOS-PECAS INVALID KEY
          DISPLAY "PROBL. RE-GRAV ESTF001" AT 2201.
    GRAVA-F1.
       REWRITE CADASTRO-GERAL-ESTOQUE-PECAS INVALID KEY
          DISPLAY "PROBL. RE-GRAV ESTF001" AT 2201.
       GO TO EXECUTA.
    FIM.
       CLOSE ARQ-ESTF001 ARQ-ESTF003 ARQ-ESTF006 ARQ-ESTF004. STOP RUN.
```

ANEXO F CÓDIGO FONTE APÓS REESTRUTURAÇÃO

```
PROCEDURE DIVISION.
INICIO.
        DISPLAY SPACES.
        OPEN I-O ARQ-ESTF001.
        IF FILE-STATUS NOT = "00"
                DISPLAY FILE-STATUS AT 1001
                DISPLAY "TERMINO ANORMAL ESTF001" AT 1101
                CLOSE ARQ-ESTF001
                STOP RUN.
        OPEN INPUT ARQ-ESTF006.
        IF FILE-STATUS NOT = "00"
                DISPLAY FILE-STATUS AT 2001
                DISPLAY "TERMINO ANORMAL ESTF006" AT 2101
                CLOSE ARQ-ESTF001
                STOP RUN.
        OPEN I-O ARQ-ESTF003.
        IF FILE-STATUS NOT = "00"
                DISPLAY FILE-STATUS AT 1501
                DISPLAY "TERMINO ANORMAL ESTF003" AT 1601
                CLOSE ARQ-ESTF003
                STOP RUN.
       OPEN INPUT ARQ-ESTF004.
       IF FILE-STATUS NOT = "00"
                  DISPLAY FILE-STATUS AT 1401
                  DISPLAY "TERMINO ANORMAL F4" AT 1501
                  CLOSE ARQ-ESTF003 ARQ-ESTF004
                  STOP RUN.
       ACCEPT DATA-W FROM DATE.
       ADD ANO-W TO ANO-C
       MOVE MES-W TO MÊS-C.
       MOVE DIA-W TO DIA-C.
    INICIO.
      PERFORM LEITURA-CADASTRO THRU FIM-LEITURA-CADASTRO
           UNTIL FINAL-LER = 1
        CLOSE ARQ-ESTF001 ARQ-ESTF003
            ARQ-ESTF006 ARQ-ESTF004
        STOP RUN.
    ****************************************************************
    LEITURA-CADASTRO.
       READ ARQ-ESTF001 NEXT AT END
          MOVE 1 TO FINAL-LER.
       IF TIPO NOT = "CTB"
          PERFORM VERIFICA-CAD-GERAL THRU FIM-VERIFICA-CAD-GERAL
          PERFORM VERIFICA-DESCONTO  THRU FIM-VERIFICA-DESCONTO
          PERFORM VERIFICA-PRECO     THRU FIM-VERIFICA-PRECO UNTIL
               NAO-DESCONTO = ZEROS
          PERFORM CALCULA-CUSTO      THRU FIM-CALCULA-CUSTO UNTIL
               NAO-PRECO = ZEROS.
    FIM-LEITURA-CADASTRO.
    EXIT.
    VERIFICA-CAD-GERAL.
       MOVE F-NUMERO-PECA TO F-NUMERO-P.
       READ ARQ-ESTF004 INVALID KEY
            MOVE ZEROS TO F-QUANT-MINIMA-1
            MOVE SPACES TO F-GRUPO-DESCONTO-1.
       MOVE F-QUANT-MINIMA-1    TO  UNIDADE-COMPRA.
       MOVE F-GRUPO-DESCONTO-1  TO  GRUPO-DESCONTO-COM.
    FIM-VERIFICA-CAD-GERAL.
    EXIT.
    VERIFICA-DESCONTO.
       MOVE ZEROS TO NÃO-DESCONTO.
       MOVE F-GRUPO-DESCONTO-COM TO CODIGO-DESCONTO.
       READ ARQ-ESTF006 INVALID KEY
                MOVE 1 TO NAO-DESCONTO.
```

ANEXO F CÓDIGO FONTE APÓS REESTRUTURAÇÃO (CONTINUAÇÃO)

```
FIM-VERIFICA-DESCONTO.
EXIT.
VERIFICA-PRECO.
   MOVE ZEROS TO NÃO-PRECO.
   MOVE F-NUMERO-PECA TO F-NUM-PECA.
   READ ARQ-ESTF003 INVALID KEY
            MOVE 1 TO NÃO-PRECO.
FIM-VERIFICA-PRECO.
EXIT.
CALCULA-CUSTO.
   MOVE F-PRECO-PUBLICO TO PRECO-PUBLICO-AUX.
   COMPUTE PRECO-AUX =  PRECO-PUBLICO-AUX * INDICE.
   MOVE PRECO-AUX TO F-CUSTO-REPOSICAO F-PRECO-REVENDEDOR.
   REWRITE LISTA-PRECOS-PECAS INVALID KEY
      DISPLAY "PROBL. RE-GRAV ESTF001" AT 2201.
   REWRITE CADASTRO-GERAL-ESTOQUE-PECAS INVALID KEY
      DISPLAY "PROBL. RE-GRAV ESTF001" AT 2201.
FIM-CALCULA-CUSTO.
 EXIT.
```