UNIVERSIDADE FEDERAL DE SÃO CARLOS
Departamento de Ciência da Computação
PPG/CC – Programa de Pós-Graduação em Ciência da Computação

Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um *Framework*

Adail Roberto Nogueira

SÃO CARLOS - SP
2002

UNIVERSIDADE FEDERAL DE SÃO CARLOS
Departamento de Ciência da Computação
PPG/CC – Programa de Pós-Graduação em Ciência da Computação

Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um *Framework*

Adail Roberto Nogueira

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.

Orientador:
Dr. Antonio Francisco do Prado

SÃO CARLOS – SP
Janeiro de 2002

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

N778td

Nogueira, Adail Roberto.
Transformação de dataflex procedural para visual dataflex orientado a objetos reusando um framework / Adail Roberto Nogueira. -- São Carlos : UFSCar, 2004.
138 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2002.

1. Tradutores de linguagem de programação. 2. Reengenharia orientada a objetos. 3. Sistema de transformação de software. 4. Framework (programa de computador). 5. Ferramenta de transformação de software I. Título.

CDD: 005.45 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

TRANSFORMAÇÃO DE DATAFLEX PROCEDURAL PARA VISUAL DATAFLEX ORIENTADO A OBJETOS REUSANDO UM FRAMEWORK

Adail Roberto Nogueira

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Antonio Francisco do Prado
(Orientador – DC/UFSCar)

Prof. Dr. Mauro Biajiz
(DC/UFSCar)

Prof. Dr. Julio Cesar Sampaio do Prado Leite
(DI/PUC – Rio de Janeiro)

São Carlos
Fevereiro/2002

Orientador
Prof. Dr. Antonio Francisco do Prado

À minha esposa Luciani e minha filha Lorena,
ao meu pai Adail, aos meus irmãos:
Maria Inêz, Celso, Sônia e Lúcia,
ao meu cunhado Ariovaldo.

Agradecimentos

Agradeço primeiramente a Deus, pois sem ele nada somos e nada podemos.

Agradeço aos meus pais, irmãos, cunhados, meu sogro e minha sogra por terem me apoiado e auxiliado em todos os momentos de minha vida.

Agradeço a minha esposa Luciani e minha filha Lorena... Vocês foram a força em meu coração, a palavra amiga, o amor e o carinho... Sem vocês não conseguiria superar todos os obstáculos e resistir aos momentos difíceis.

Meu orientador Antonio Francisco do Prado, sua orientação, seus ensinamentos, sua amizade, e principalmente seu exemplo de dedicação marcaram de forma permanente minha vida... Nunca terei palavras suficientes para lhe agradecer.

Ao professor Mauro pela participação em minha qualificação e pela palavra amiga no momento exato que revitalizou minhas esperanças de produzir um bom trabalho.

À professora Rosangela, também pela qualificação, pelas críticas construtivas e contribuições.

Aos demais professores por todo o conhecimento que generosamente me ofereceram no decorrer dos créditos, e funcionários do DC pelo apoio.

Ao meu amigo Lupércio, pela amizade e contribuições nas publicações.

Aos meus amigos, Lucio, Sergio, Ademir, Arnaldo, Joseval, Simone e Patrícia, pelo apoio e troca de idéias que enriqueceram este trabalho.

Ao Thiago e Renato companheiros de trabalho e de pesquisa.

Ao *Draco Team* da UFSCar, Ricardo,Vinícius, Valdirene, Ângela e Moraes, a ajuda de vocês sempre foi crucial.

Aos meus companheiros Iolanda, Calebe, Diogo, Elisângela, Eduardo, Daniel, Christmas, João Noivo, Eduardo Cotrin, Eduardo Leal e Góis, que me acompanharam e ajudaram nessa caminhada.

A CONSYSTEM e UNIFIL, pelo apoio e compreensão.

A todos que de forma direta ou indireta contribuíram para que este trabalho fosse desenvolvido.

Muito Obrigado.

Sumário

Lista de Figuras

Resumo

Este trabalho apresenta uma estratégia para transformação de sistemas legados, construídos em DataFlex Procedural (DFP), para sistemas em Visual DataFlex Orientado a Objetos (VDFOO), reusando um *Framework*, denominado DataFlex Application *Framework* (DAF), com arquitetura em três camadas: interface, regras de negócio e banco de dados. A transformação é realizada em três passos: "Organizar Código Legado", quando ocorre a organização do código legado em DFP; "Reimplementar Código DFP Organizado", quando é realizada a reimplementação do código DFP Organizado para VDFOO; e Executar Código VDFOO, quando é executado o código em VDFOO com as mesmas funcionalidades do código legado em DFP.

São apresentadas as tecnologias utilizadas para a elaboração deste trabalho, envolvendo Técnicas para Reengenharia, Sistemas Transformacionais, Linguagens DataFlex Procedural (DFP) e Visual DataFlex Orientado a Objetos (VDFOO) e o DataFlex Application *Framework* (DAF).

O Sistema Transformacional Draco-PUC é o principal mecanismo para automação das transformações. Para auxiliar na definição dos domínios no Draco-PUC foi construído um editor de domínios denominado Draco Domain Editor (DDE), além do *parser* e *prettyprinter* DFP e VDFOO. Para automatizar o passo "Organizar Código Legado", foi construído o Transformador de DFP para DFP Organizado, e para o passo "Reimplementar Código DFP Organizado", foi construído o Transformador de DFP Organizado para VDFOO. O DDE também é usado para definir o *script* de execução dos transformadores.

A Transformação de DFP para VDFOO reusa um *Framework* em três camadas: interface, regras de negócio e banco de dados.

São apresentadas as transformações de dois sistemas: Controle de Mandados, com cerca de 20 mil linhas de código legado; e o Sistema Integrado para Revendas e Concessionárias (SIRC-X), com 5.3 milhões de linhas de código.

Abstract

This work presents a strategy for transformation of legacies systems write in Procedural DataFlex (DFP), for systems in Objects Oriented Visual DataFlex (VDFOO) reusing a Framework, denominated DataFlex Application Framework (DAF), with architecture in three layers: interface, business rules and database. The transformation is accomplished in three steps: To "Organize Legacy Code", when it happens the organization of the legacy code in DFP; "Reemployments Organized Code" DFP, when the reemployment of the Organized Code DFP for VDFOO; and to “Execute the VDFOO Code”, when the code is executed in VDFOO with the same functionalities of the legacy code in DFP.

The technologies used for the elaboration of this work are presented, involving Techniques for Reengineer, Transformations Systems, Languages Procedural DataFlex (DFP) and Objects Oriented Visual DataFlex (VDFOO) and DataFlex Application Framework (DAF).

The Draco-PUC Transformation System is the main mechanism for automation of the transformations. To aid in the definition of the domains in Draco-PUC an editor of denominated domains it was built Draco Domain Editor (DDE), besides the parser and prettyprinter DFP and VDFOO. To automate the step "Organize Legacy Code", the Transformer of DFP was built for Organized DFP, and for the step "Reemployments Organized Code" DFP, the Transformer of Organized DFP was built for VDFOO. DDE is also used to define the script of execution of the transformers.

The Transformation of DFP for VDFOO reuse a Framework in three layers: interface, business rules and database.

The transformations of two systems are presented: Control of Orders, with about 20 thousand lines of legacy code; and the Integrated System for stores of sale’s of cars (SIRC-X), with 5.3 million code lines.

Capitulo 1 Introdução

A evolução das técnicas e linguagens de programação, objetivando o ganho de produtividade e qualidade no produto de *software* é cada vez mais rápida e efetiva. As empresas responsáveis pela construção de *software* e ferramentas de desenvolvimento mostram-se impelidas a lançar novas versões de seus produtos, que atendam aos anseios do mercado. Dentre estes anseios destacam-se o de atender ao paradigma da orientação a objetos e dispor de ambientes gráficos de desenvolvimento, que suportem a construção de sistemas com interface gráfica. [Rea92, Cor93, Bax97, Faq98].

Devido a complexidade de manter uma linguagem híbrida, procedural e orientada a objetos, produtores de linguagens, como a DataAccess Corporation, decidiram romper o elo com o reconhecimento dos códigos legados em detrimento dos novos recursos que podem ser explorados na concepção orientada a objetos. Esse problema, das empresas responsáveis pelas linguagens, dificulta o aproveitamento do conhecimento contido em sistemas legados. Assim, para possibilitar o aproveitamento e a evolução dos sistemas legados, diversas técnicas de Reengenharia de Software são propostas, incluindo técnicas de reconstrução do software usando transformações [Icsr4].

Diferentes Sistemas Transformacionais têm sido usados para a automatização, na Reengenharia de Software, destacando-se o Draco-PUC, que implementa as idéias de transformação de software orientada a domínios [Nei84, Lei91a, Lei91b, Lei94, Pra92, San93, Pra98, Abr99a, Abr99b]. Seguindo esse enfoque, este trabalho apresenta uma forma para a transformação de sistemas legados, construídos em DataFlex Procedural (DFP), para sistemas em Visual DataFlex Orientado a Objetos (VDFOO), reusando um *Framework* denominado DataFlex Application *Framework*, com arquitetura em três camadas: interface, regras de negócio e banco de dados.

A motivação deste projeto vem da existência de um grande número de sistemas construídos em DFP, usando uma interface em caracter e da necessidade de evoluí-los para o paradigma orientado a objetos com uma interface gráfica. Segundo o levantamento realizado pela DataAccess Corporation, empresa responsável pela linguagem, cerca de 30.000 de seus clientes ainda trabalham com DFP, totalizando nove milhões de programas, com cerca de 13,5 bilhões de linhas de código fonte. Conforme [Fle95, Fle98, Fle99], 80% desses clientes têm

interesse que suas aplicações possam ser executadas usando os recursos apresentados pelo VDFOO*.

O reuso de um *Framework*, para auxiliar na reimplementação do sistema com a arquitetura em três camadas [Gam95], proporciona maior facilidade na readequação do sistema gerado, para inserção de novos requisitos, após o processo de transformação. A inserção de novos requisitos posterior à transformação permite uma evolução gradual do sistema, preservando o controle de fluxo de execução do sistema legado. Este trabalho está organizado da seguinte forma:

- O capítulo 2 apresenta as principais tecnologias utilizadas no desenvolvimento deste trabalho, que inclui a Engenharia Reversa, Reengenharia, Sistemas Transformacionais, Sistema Transformacional Draco-PUC, Linguagens DataFlex Procedural e Visual DataFlex Orientado a Objetos, e o DataFlex Application *Framework*;
- O capítulo 3 apresenta um editor de domínios para o Draco-PUC, chamado Draco Domain Editor (DDE). São apresentados os editores de gramáticas, de transformadores, de *scripts* de execução dos transformadores e de *scripts* de transformação;
- O capítulo 4 apresenta a construção dos domínios DataFlex Procedural (DFP) e Visual DataFlex Orientado a Objetos (VDFOO), detalhando a construção das gramáticas DFP e VDFOO e dos transformadores de DFP para DFP Organizado e de DFP Organizado para VDFOO além do *script* de execução das transformações;
- O capítulo 5 apresenta a estratégia para o uso da Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um *Framework*. São apresentados estudos de caso que mostram a execução da estratégia para transformação de DFP para VDFOO reusando um *Framework*; e
- Finalmente, o capítulo 6 apresenta as conclusões sobre este projeto de pesquisa.

* Levantamento realizado pela DataAccess Corporation, para direcionamento de seus produtos, resumindo tendências mostradas nas revistas Flex Magazine de 1995 a 1999, apresentado na Convenção Internacional sobre DataFlex, realizada em São Paulo, em abril de 2000.

Capitulo 2 Tecnologias Utilizadas

Para suportar a transformação de códigos escritos em DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um *Framework*, foram integradas diferentes tecnologias, como Engenharia Reversa de Software, Reengenharia de Software, Sistemas Transformacionais, o Draco Doman Editor (DDE), as linguagens DataFlex Procedural e Visual DataFlex Orientado a Objetos, e *Framework*s.

A terminologia empregada na Engenharia de Software para referenciar as tecnologias de análise e entendimento de sistemas legados é apresentada por Chikofsky [Chi90], com o objetivo de racionalizar termos que já estão em uso. Os principais termos definidos e relacionados são: Engenharia Avante, Engenharia Reversa, Redocumentação, Recuperação de Projeto, Reestruturação e Reengenharia. O relacionamento entre esses termos é mostrado na Figura 1, considerando-se que o ciclo de vida do *software* possui três grandes etapas: Requisitos, Projeto e Implementação, com claras diferenças no nível de abstração. Os Requisitos tratam da especificação do problema, incluindo objetivos, restrições e regras de negócio. O Projeto trata da especificação da solução. A Implementação trata da codificação, teste e entrega do sistema em operação.

A Figura 1 mostra a direção seguida pela Engenharia Avante, do nível de abstração mais alto para o mais baixo. Mostra também, que a Engenharia Reversa percorre o caminho inverso, podendo utilizar-se da Recuperação de Projeto para melhorar o nível de abstração. A Reengenharia geralmente inclui uma Engenharia Reversa, seguida de alguma forma de Engenharia Avante ou Reestruturação.

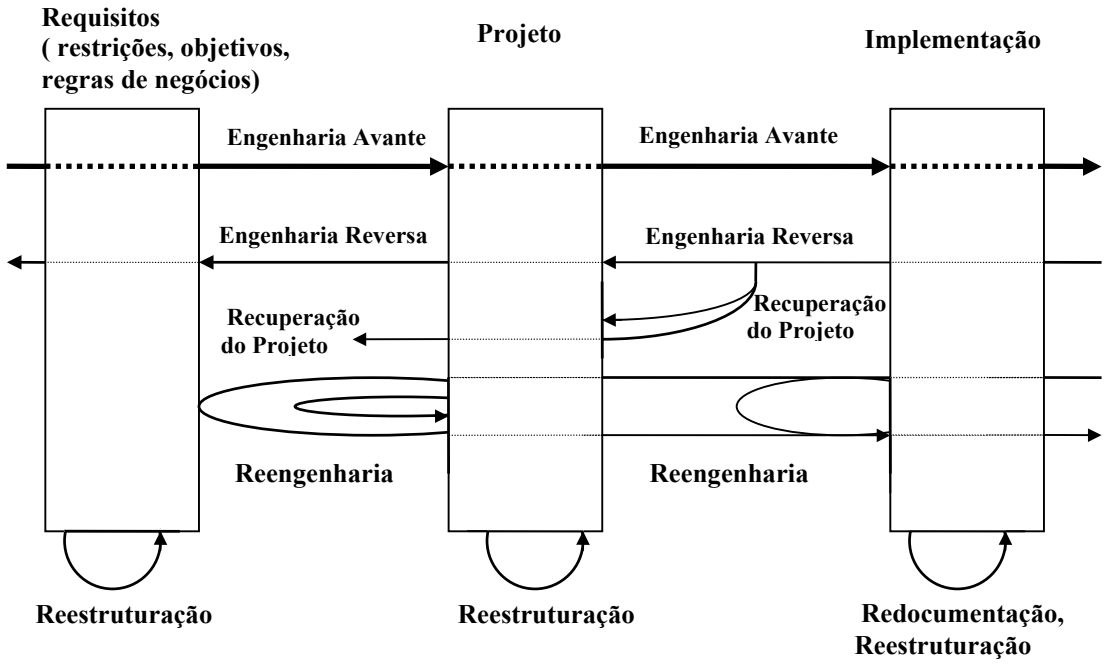


Figura 1. Ciclo de vida de *software* [Chi90].

2.1 Engenharia Reversa

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 normalmente aplicada para melhorar o produto de uma empresa a partir da análise dos produtos do adversário. "*Engenharia Reversa é o processo de desenvolvimento de um conjunto de especificações para um sistema de hardware complexo por exame de espécimes daquele sistema, de forma ordenada.*" [Rek85]. Esse processo é, em geral, conduzido por outro desenvolvedor, sem o benefício de alguns dos modelos originais, com o propósito de fazer um *clone* do sistema de *hardware* original.

No *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 [Chi90]. 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, depois informações sobre a estrutura de dados e de programa, modelos de controle de fluxo e de dados, chegando a modelos entidade-relacionamento, que constituem o nível de abstração mais alto. O ideal é ter um nível de abstração mais alto possível [Pre95].

A utilização de Engenharia Reversa proporciona uma melhoria significativa do reuso e da manutenção de *software*. O reuso de *software* pode ser apoiado pela Engenharia Reversa tanto na identificação e composição de componentes a partir de partes reutilizáveis de sistemas legados quanto na elaboração da documentação dos novos sistemas. A manutenção de *software* é uma tarefa difícil e por isso deve contar com a ajuda de uma documentação completa de todo o processo de desenvolvimento. Nesse ponto, a Engenharia Reversa pode fornecer as visões em diversos níveis de abstração, permitindo a localização dos componentes a serem mantidos, além de melhorar a compreensibilidade do *software*.

Existem ferramentas que usam o código fonte de um programa como entrada e extraem a arquitetura do programa, estrutura de controle, fluxo lógico, estrutura de dados e fluxo de dados; e ferramentas que monitoram o *software* em execução e usam as informações para construir um modelo comportamental do programa [Pre95].

Diversas ferramentas e métodos têm sido propostos para apoiar a Engenharia Reversa, entre os quais podemos destacar:

Benedusi [Ben92], define um paradigma de referência, chamado "*Goals/Models/Tools"*, para estabelecer processos de Engenharia Reversa capazes de gerar, automaticamente, informações e documentos de projeto, a partir do código fonte;

Wong [Won95], ressalta que para sistemas grandes, o entendimento de aspectos estruturais da arquitetura do sistema é mais importante do que o entendimento de um componente isolado qualquer;

Sneed [Sne95], apresenta uma abordagem para extrair documentação de projeto orientado a objetos, automaticamente, a partir de programas existentes em COBOL;

Hainaut [Hai96], propõe um processo para recuperação da estrutura na Engenharia Reversa de bases de dados;

Finnigan [Fin97], propõe um ambiente de apoio à migração de *software*, chamado "*Software bookshelf"*, que provê meios de captar, organizar e gerenciar informações sobre sistemas legados;

Armstrong [Arm98], compara cinco ferramentas para recuperação da arquitetura de sistemas legados: *Rigi*, *Dali workbench*, *Software Bookshelf*, *CIA* e *SniFF+*; e

Penteado [Pen95, Pen96], apresenta um método, posteriormente denominado Fusion/RE, para Engenharia Reversa de sistemas legados implementados sem usar a tecnologia da orientação a objetos, com o objetivo de produzir o modelo de análise orientado a objetos. Baseia-se no método Fusion [Col94] para desenvolvimento orientado a objetos. Para a Engenharia Reversa, o método Fusion/RE propõe quatro passos: Revitalizar a Arquitetura, Recuperar o Modelo de Análise do Sistema Atual (MASA), Criar o Modelo de Análise do Sistema (MAS) e Mapear o Modelo de Análise do Sistema para o Modelo de Analise do Sistema Atual.

Das técnicas apresentadas pelo Fusion/RE, a hierarquia de chamadas, descritas no passo Revitalizar a Arquitetura, é usada para auxiliar o tratamento do fluxo de execução pelos transformadores. O passo MASA, aborda diversas técnicas como as de identificação de supostas classes, supostos métodos e atributos. Também são abordadas técnicas que possibilitam definir o encapsulamento dos supostos métodos em supostas classes. As técnicas de elaboração de cenários para criação dos Modelos de Ciclo de Vida e de Operações, são utilizadas para auxiliar o particionamento do sistema em casos de uso, e reconhecimento de eventos, que auxiliam no tratamento do fluxo de controle. No passo MAS, as técnicas para tratamento de generalização e desmembramento de métodos que possuem anomalias, são usadas para auxiliar a definir o encapsulamento dos métodos, a extensão de classes e tratamento das conexões de mensagens.

A Engenharia Reversa produz excelentes resultados, de grande utilidade para o Engenheiro de Software responsável por alterações, reuso e evolução do sistema. Quando os modelos por ela obtidos seguem o paradigma da orientação a objetos, ainda mais vantagens são oferecidas, principalmente quanto à facilidade de Reengenharia com mudança de paradigma.

No tópico seguinte são apresentados conceitos sobre Reengenharia.

2.2 Reengenharia

A Reengenharia [Pre95], também chamada renovação ou recuperação, utiliza as informações recuperadas de um sistema para alterá-lo ou reconstituí-lo, procurando melhorar sua qualidade global e reduzir custos com manutenção. Na maioria dos casos a Reengenharia reimplementa as mesmas funcionalidades do sistema legado. Ao mesmo tempo, o desenvolvedor de sistemas pode também adicionar novas funções e melhorar o desempenho global do sistema.

Uma das grandes motivações para aplicação da Reengenharia é a diminuição dos altos custos de manutenção de sistemas, que se devem a diversos fatores, discutidos por Wilkening em [Wilk95].

A manutenção contínua de sistemas, sem uma adequada atualização da documentação de seu projeto, faz com que a implementação fique inconsistente com o projeto original, tornando o código difícil de ser entendido e sujeito a erros, e a documentação desatualizada. As linguagens em que esses sistemas foram implementados normalmente estão ultrapassadas, não havendo suporte por parte dos fabricantes nem programadores que as dominem. A manutenção quase sempre é manual e não dispõe de ferramentas *CASE*. Esses sistemas muitas vezes foram desenvolvidos sem seguir os princípios da Engenharia de Software e tem um código desestruturado e difícil de entender. A alta rotatividade do pessoal que realiza a manutenção do sistema, faz com que muita informação fique perdida, diminuindo o conhecimento existente sobre o sistema.

Assim, o objetivo da aplicação de técnicas de Reengenharia em sistemas é facilitar sua evolução disciplinada, desde o estado corrente até o novo estado desejado. Isso é possível por meio de operações que agem nos diferentes níveis de abstração do ciclo de vida do *software*.

Para organizar o processo de reengenharia, diversas técnicas são propostas, dentre as quais podem-se destacar:

Jacobson [Jac91] apresenta uma técnica para efetuar a Reengenharia de sistemas legados, implementados em uma linguagem procedimental como C ou Cobol, obtendo sistemas orientados a objetos. Mostra como fazer essa Reengenharia de forma

gradual, pois considera impraticável substituir um sistema antigo por um completamente novo. Considera três cenários diferentes: no primeiro faz-se a mudança de implementação sem mudança de funcionalidade, no segundo faz-se a mudança parcial da implementação sem mudança de funcionalidade, e no terceiro faz-se alguma mudança na funcionalidade. As técnicas propostas para Reengenharia de forma gradual foram adaptadas neste trabalho para a divisão das atividades de transformação em etapas. De forma semelhante às apresentadas por Jacobson, as etapas são evolutivas, sempre se baseando no resultado da etapa anterior;

Markosian [Mar94] reclama da falta de apoio computadorizado para a Reengenharia de sistemas, em contraposição à grande proliferação de ferramentas CASE para desenvolvimento de software novo. Diz que as ferramentas de transformação atuais são muito limitadas, sendo difícil suas adaptações a um projeto em particular. Aborda uma nova tecnologia para Reengenharia, que chama de "tecnologia facilitadora" (*enabling technology*), relatando resultados práticos bastante animadores quanto à produtividade da sua aplicação. A "tecnologia facilitadora" consiste no rápido desenvolvimento de ferramentas para analisar e modificar sistemas legados. Deve ser usada em tarefas complexas de Reengenharia, que estejam sendo feitas de forma manual ou semi-automatizada. As sugestões de Markosian para melhoria nas ferramentas de análise e modificação dos sistemas legados foram uma das inspirações para a construção de uma ferramenta para auxiliar na implementação das transformações a serem realizadas;

Sage [Sag95] discute os possíveis tipos de Reengenharia que podem ser considerados: Reengenharia de Produto, de Processo e de Gestão de Sistemas. A Reengenharia de Produto é definida como sendo o exame, estudo, captação e modificação dos mecanismos internos ou da funcionalidade de um produto ou sistema existente, para reconstituí-lo em uma nova forma e com novas características, mas sem maiores mudanças na funcionalidade e propósitos do sistema. A Reengenharia de Processo é o exame, estudo, captação e modificação dos mecanismos internos ou da funcionalidade de um processo existente, ou ciclo de vida de engenharia de sistemas, para reconstituí-lo em uma nova forma e com novas características funcionais e não-funcionais, mas sem mudar o propósito inerente ao processo. A Reengenharia de Gestão de Sistemas é o exame, estudo, captação e modificação dos mecanismos internos ou da funcionalidade de processos e práticas de gestão de sistemas legados em uma organização, para reconstituí-los em uma nova forma e com novas características, mas sem mudar o propósito inerente à organização. Embora Sage não

tenha apresentado técnicas usadas diretamente neste trabalho, sua visão abrangente sobre a reengenharia proporcionou uma visão mais crítica no processo de transformação;

Wilkening [Wilk95] apresenta um processo para efetuar a Reengenharia de sistemas legados, aproveitando partes de sua implementação e projeto. Esse processo inicia-se com a reestruturação preliminar do código fonte, para introduzir algumas melhorias, como remoção de construções não-estruturadas, código "morto" e tipos implícitos. 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 suas representações em níveis mais altos de abstração. Com base nessas representações, pode-se prosseguir com os passos de reestruturação, reprojeto e redocumentação, que são repetidos quantas vezes forem necessárias para se obter o sistema totalmente reestruturado. Pode-se, então, implementar o programa na linguagem destino e dar seqüência aos testes que verificarão se a funcionalidade não foi afetada. Wilkening apresenta a ferramenta RET, que automatiza parcialmente esse processo, sendo, portanto, uma Reengenharia assistida por computador. As técnicas de organização do código legado, apresentadas por Wilkening foram muito úteis na sua preparação para a transformação. Apesar de não tratar da mudança de paradigmas, o código organizado, sem construções não estruturadas, redundâncias e código "morto", facilita a transformação;

Klösh [Klo96] discute uma abordagem para Reengenharia com mudança de orientação, na qual aplica primeiramente a Engenharia Reversa, para depois mudar para uma linguagem orientada a objetos. Esse trabalho é uma continuação do trabalho de Gall [Gal95], no qual é apresentado o método COREM para transformação de programas, que refaz sistemas de arquitetura procedimental, tornando-os orientados a objetos. Klösh justifica o uso da orientação a objetos como forma de melhorar a manutenibilidade futura, devido a conceitos como abstração, encapsulamento e conexão de mensagens. É contrário à utilização de herança e polimorfismo, por acreditar que esses conceitos complicam potencialmente as operações de manutenção. A abordagem proposta por Klösh possui quatro passos: recuperação de projeto, modelagem da aplicação, mapeamento dos objetos e adaptação do código fonte. Klösh concorda com a automatização de partes do processo de Reengenharia, mas acredita que seja melhor uma abordagem híbrida, que use a assistência automática de uma ferramenta computadorizada, mas que, quando ela alcançar seus limites, conte com a intervenção da sabedoria humana para superá-los. Foram usadas as técnicas apresentadas por

Klösh, para o mapeamento dos objetos e adaptação do código fonte, com algumas alterações para mapeamento de classes; e

Sneed [Sne96] descreve um processo de reengenharia apoiado por uma ferramenta para extrair objetos a partir de programas existentes em COBOL. Ressalta a predominância da tecnologia de objetos nos dias de hoje, principalmente em aplicações distribuídas com interfaces gráficas, questionando a necessidade de migração de sistemas legados para essa nova tecnologia. Identifica obstáculos à Reengenharia orientada a objetos como, por exemplo, a identificação dos objetos, a natureza procedimental da maioria dos sistemas legados, que leva a blocos de código processando muitos objetos de dados, a existência de código redundante e a utilização arbitrária de nomes. Assume como pré-requisitos para a Reengenharia orientada a objetos a estruturação e modularização dos programas, além da existência de uma árvore de chamadas do sistema. É apresentado um processo de Reengenharia orientada a objetos, composto de cinco passos: seleção de objetos, extração de operações, herança de características, eliminação de redundâncias e conversão de sintaxe. A seleção de objetos é feita com apoio da ferramenta, mas o responsável pela determinação dos objetos é o Engenheiro de Software. A extração de operações particiona o programa em sub-rotinas, removendo os segmentos referentes a um determinado objeto e substituindo-os por envio de mensagens ao objeto no qual os dados locais estão encapsulados. Foram usadas as técnicas de particionamento do programa em sub-rotinas para obtenção dos métodos e as de substituição destas sub-rotinas por mensagens de acionamento das operações.

Como mecanismos para sistematizar a Reengenharia de Software foram pesquisados sistemas transformacionais, abordados no próximo tópico.

2.3 Sistemas Transformacionais

Sistemas transformacionais são ferramentas de apoio ao Engenheiro de Software que permitem a manipulação estrutural e semântica de sistemas. Essas ferramentas têm sido aplicadas a diversos tipos de tarefas tais como síntese de programas e Reengenharia, alcançando resultados positivos [Nei84, Lei91a, Lei91b, Lei94, Pra92, San93, Pra98, Abr99a, Abr99b, Fuk99, Jes99, Nog01a, Nog01b].

Sistemas transformacionais podem trabalhar com apenas um conjunto restrito

de linguagens de descrição de artefatos e de operadores, ou então, podem ser caracterizados por uma arquitetura que permite ampla configuração para diversas situações de uso. A primeira categoria é chamada de sistemas transformacionais específicos e a segunda de genéricos.

Nesta forma de classificação, compiladores tradicionais são vistos como exemplos de sistemas transformacionais específicos, já que não podem ser configurados para trabalhar com diversas linguagens e também possuem um conjunto pré-configurado e restrito de operadores de transformação. Os principais esforços realizados em pesquisa e desenvolvimento de sistemas transformacionais concentram-se no aprimoramento de sistemas genéricos.

De maneira abstrata, sistemas transformacionais genéricos podem ser caracterizados por cinco módulos principais: um módulo para a configuração de linguagens de descrição de artefatos, um módulo de descrição de transformadores, um módulo de importação de descrições de artefatos, um Motor Transformacional (MT) e um módulo de exportação de descrições de artefatos. O módulo de configuração de linguagens permite descrever no ambiente diversas linguagens de representação de artefatos, de tal forma que o módulo importador possa capturar estas descrições e submetê-las ao MT. O MT aplica os transformadores que foram disponibilizados através do módulo de descrição de transformadores. O MT pode ter sua lógica de aplicação configurada através de estratégias de uso dos transformadores, de acordo com o grau de automatização suportada pelo ambiente na interação com o Engenheiro de Software que controla a atividade. Uma vez produzida, a descrição resultante é exportada para que o Engenheiro de Software a utilize. O módulo de exportação realiza essa tarefa, mapeando a forma interna manipulada pelo MT para o formato de apresentação desejado pelo Engenheiro de Software. O funcionamento dos módulos de importação e exportação está vinculado às linguagens de representação disponibilizadas pelo módulo de descrição de linguagens de representação.

A seguir são descritos alguns dos principais sistemas transformacionais existentes.

ANTLR/DLG [Par91, Ant95] ANTLR é uma ferramenta para reconhecimento de idioma e DLG é uma abreviação para gerador de analisador léxico. São partes de um

conjunto de ferramentas de construção de compiladores (PCCTS-*Purdue Compiler Construction Tool Set*) e têm sido usadas na área de Reengenharia para definir gramáticas.

ASF+SDF [Ber89, Hee89], são formalismos de especificação algébrica modular para a definição de sintaxe e semântica de programação. É uma combinação de dois formalismos. ASF (*Algebraic Specification Formalism*) é um formalismo para especificação algébrica, e SDF (*Syntax Definition Formalism*) é um formalismo para definição de sintaxe. ASF e SDF são integrados com um meta-ambiente de programação interativa. Este sistema suporta todo o ciclo de desenvolvimento de *software*. Trata-se de um gerador de *parser* genérico, baseado em modificações do algoritmo de Tomita [Tom86]. Foi usado para definir dialetos de COBOL com o objetivo de suportar a Reengenharia, gerando especificações de componentes para fábricas de renovação de *software* independentes de linguagem.

CobolTransformer [Sib97, Blo97, Bra97, Bra97a] é um produto designado para transformação de doze dialetos de COBOL em um dialeto padronizado. Para descrever a gramática COBOL é utilizado o BtYacc (*Back Tracking Yacc*) criado em Berkeley, por Chris Dodd que modificou a definição original do Yacc. O projeto teve a participação de Vadim Maslov da Siber Systems. Na versão 1.1.4 o CobolTransformer inclui suporte à modularidade que ajuda ao tratar dialetos diferentes de uma mesma linguagem.

COSMOS [Hal96, Tec97] é uma ferramenta para auxiliar na solução automatizada do *bug* do ano 2000 (Y2K). Pode analisar gramaticalmente um grande número de linguagens e dialetos diferentes. Concebido por Fred Hirdes, utiliza as técnicas Flex e Bison em seus *parsers*.

DMS [Bax97] é um sistema desenvolvido pela *Semantic Designs Inc.*, e utiliza a tecnologia de transformação de sistemas para revisar as informações formais do projeto. A infra-estrutura do DMS é descrita na forma de linguagens de domínio e de componentes de *software*.

Popart [Wile00] é um sistema com uma linguagem de definição para analisar e reescrever regras de produção. Possui um ambiente para definição de gramáticas livres de contexto. Este ambiente gera o *parser* e um *PrettyPrinter*. O projeto foi concebido como uma forma de possibilitar a criação de gramáticas mais simples para os projetistas e

programadores, principalmente linguagens formais para descrição da análise e dos requisitos, para submetê-las à transformação e gerar um código em Lisp. Foram desenvolvidos projetos criando gramáticas de linguagens existentes, com o intuito de possibilitar a Reengenharia para Lisp. Existem projetos em andamento para suportar a geração em C++ e Java.

Refine [Rea92] é um ambiente de Reengenharia comercial produzido pela *Reasoning Systems* (US), baseado em pesquisa desenvolvida em *Kestrel*. O principal ponto no Refine é uma biblioteca de *Common Lisp* que encapsula as diversas bases de um sistema de transformação. A ênfase atual da *Reasoning* é o uso do Refine como uma ferramenta de Engenharia Reversa. Já existem pacotes usando o Refine com suporte para as linguagens Cobol, Fortran, C e Ada.

RescueWare [Faq98, Rel00] é um produto comercial produzido pela *Relativity Technologies*, fundada em fevereiro de 1997, no *Research Triangle Park*, Carolina do Norte, cuja missão é procurar ser o principal fornecedor de soluções para a transformação de software legado. O RescueWare propõe-se a resolver o problema de compreensão de sistemas legados e a mover seletivamente somente as partes relevantes da aplicação para uma plataforma de *software* mais moderna. Essas plataformas estão baseadas na Internet e na arquitetura cliente e servidor, suportando as linguagens Java, C++ e Visual Basic.

Revolve/2000 [Bil89, Mic00] é uma ferramenta de *parser* da Micro Focus que pode entender dialetos alemães específicos do COBOL, criado por Joachim Blomi. Pouco se sabe sobre a forma de sua implementação interna. É utilizado para reconhecer os diversos dialetos de COBOL e transformá-los em um padrão definido ainda em COBOL.

SES GmbH [Sne97] não é uma ferramenta, mas sim uma empresa especializada em serviços de Reengenharia de Software. A empresa possui *parsers* para COBOL, PL/1, Assembler, NATURAL, CICS, DLI e SQL. Os *parsers* para CICS, DLI e SQL são chamados pelos *parsers* do COBOL, PL/1 e Assembler, através de comando EXEC CICS, EXEC DLI, ou EXEC SQL.

Tampr [Bir00] é um sistema inicialmente construído em Lisp e posteriormente portado para Fortran. Usa a linguagem Poly para a definição de *parsers* e transformações. A maioria das aplicações de *Tampr* têm sido de Lisp para Fortran, e o código

resultante é relatado como muito eficiente. Também são relatados trabalhos com Pascal e C;

TXL [Cor91, Cor93, Com00] tem uma elegante linguagem de descrição de transformação e uma documentação bem melhor que as demais ferramentas aqui apresentadas. Sua disponibilidade na Internet tem difundido seu uso e diversas aplicações de sucesso têm sido relatadas. Porém, o TXL trabalha com análise gramatical recursiva, através de um algoritmo que varre o código em profundidade para depois regressar, e não utiliza meios de armazenamento auxiliar, o que gera um comportamento exponencial de consumo de recursos.

Outro importante sistema transformacional é o **Draco-PUC**, que foi usado como principal ferramenta para a automação realizada neste trabalho.

2.3.1 Sistema Transformacional Draco-PUC

O paradigma Draco [Nei80, Nei84] está fundamentado no reuso de componentes de software. Esses componentes são organizados em domínios, representados por uma linguagem com sintaxe e semântica bem definida.

A máquina Draco-PUC foi construída com o objetivo de testar, desenvolver e colocar em prática o paradigma Draco [Lei91a, Pra92]. O Draco-PUC é um sistema transformacional genérico baseado em um Motor Transformacional (MT) para manipulação de Árvores de Sintaxe Abstratas Draco (DAST - *Draco Abstract Syntax Trees*). Seguindo o modelo abstrato de sistemas transformacionais genéricos, este motor funciona acoplado a módulos de descrição de linguagens, e de transformadores, que representam um domínio no Draco-PUC. Um domínio no Draco-PUC, conforme mostra a Figura 2, é constituído de três partes [Fuk99, Jes99]:

1. **Linguagem:** Através do módulo de descrição de linguagens, linguagens de representação são especificadas segundo seus elementos léxicos e sintáticos. A partir destas especificações são gerados analisadores léxicos e sintáticos. Os analisadores léxicos trabalham com autômatos finitos determinísticos, derivados a partir das expressões regulares que descrevem os símbolos da linguagem de representação. Os analisadores sintáticos trabalham com autômatos dirigidos por tabelas. O módulo de importação

trabalha com os analisadores léxicos e sintáticos gerados pelo módulo de descrição de linguagens, ou através da leitura de arquivos textuais que contêm Árvores de Sintaxe Abstratas representadas sob a forma de expressões simbólicas, para carregar o código do domínio especifico e representá-lo internamente através de uma Árvore de Sintaxe Abstrata, que no Draco-PUC é denominada *Draco Syntax Abstract Tree* (DAST);

2. ***PrettyPrinter:*** O módulo de exportação pode exportar DAST´s sob a forma de expressões simbólicas ou, então, pode usar os formatadores de layout gerados pelo módulo de descrição de linguagens. Os formatadores de layout, também chamados de *PrettyPrinters*, trabalham com um conjunto restrito de operações de exibição de informação textual que permitem exibição de símbolos, espaçamento, tabulação e apresentação de seqüências de repetição de elementos. Estas operações básicas são organizadas em procedimentos de navegação sobre DASTs para que as informações contidas em seus nodos sejam apresentadas textualmente. Esta apresentação textual segue a sintaxe concreta da linguagem de representação às quais estão associados os nodos de uma DAST; e

3. **Transformadores**: O módulo de descrição de transformadores é baseado em regras de reescrita. Uma regra é composta por um padrão de busca, chamado de Lado Esquerdo da Regra (LHS - *Left Hand Side*) e por um padrão de substituição, chamado de Lado Direito da Regra (RHS - *Right Hand Side*). Estes padrões são padrões sintáticos, descritos através de parametrização de fragmentos de DASTs, ou trechos de programas nas linguagens de representação disponíveis. A parametrização de DASTs ou programas é feita através de variáveis-padrão (*pattern-variables*) que generalizam as estruturas a serem encontradas e substituídas. Nomes simbólicos são associados a estas variáveis para que possam ser referenciadas. As diversas regras de rescrita são organizadas em conjuntos, os quais por sua vez podem ser agrupados em módulos. Estes módulos são os blocos básicos usados pelo engenheiro de software que controla o processo de transformação. Estes módulos de conjuntos de regras de reescrita são chamados de Transformadores.

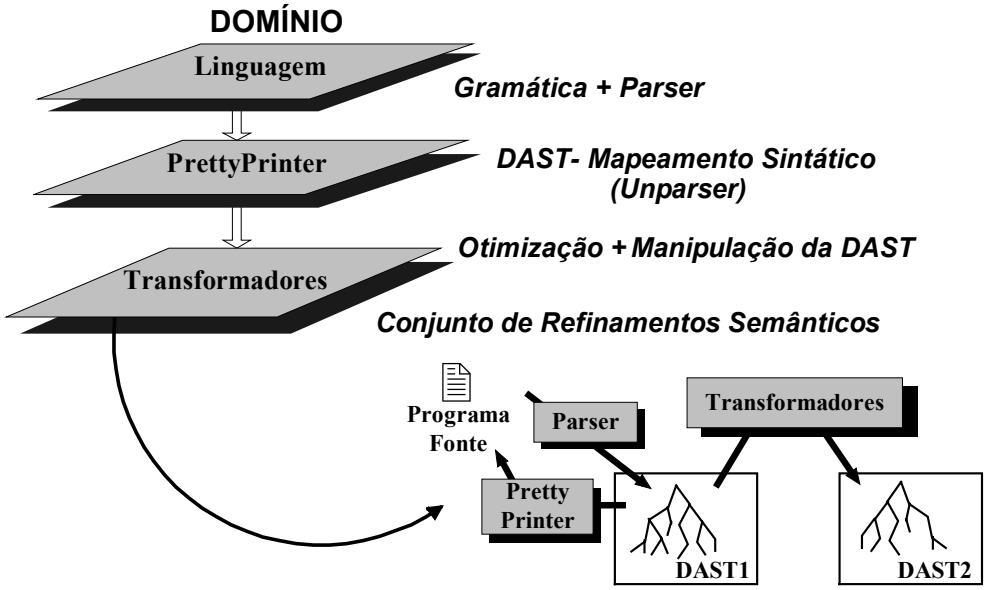


Figura 2. Partes de um domínio na máquina Draco-PUC [Fuk99].

O Motor Transformacional (MT) do Draco-PUC trabalha de forma análoga a um sistema de inferência clássico. Ao invés de operar com uma memória de trabalho baseada em fatos lógicos, a memória de trabalho do MT opera com DASTs. Ao invés de trabalhar com regras de inferência, deflagradas a partir de um mecanismo de unificação para termos de primeira ordem, o MT usa um mecanismo de casamento de padrões em árvores que permite parametrização por variáveis do tipo seqüência. Ao invés de operar com um mecanismo de controle baseado em resolução, o MT trabalha com um mecanismo de controle que combina navegação sobre árvores para definição de um filtro de contexto e um filtro de regras de reescrita baseada em nodos-chave.

O MT itera sobre os nodos da DAST em busca de novos localizadores que permitam a aplicação de regras. Uma vez posicionado um localizador, é extraído o nodo-chave do mesmo e este é usado como elemento de busca para regras candidatas dentro do conjunto de regras ativo. Por meio do mecanismo de casamento de padrões, as regras candidatas são testadas segundo seus padrões de busca. Caso o casamento seja realizado com sucesso, um ambiente de amarração de variáveis é produzido e sobre este é aplicado o padrão de reescrita da regra. Uma instância de substituição é produzida e o conteúdo do localizador é totalmente substituído pelo conteúdo desta instância. Segundo o mecanismo de controle definido para a navegação do MT sobre as árvores e escolha de regras, este continua seu trabalho até que as possibilidades de transformação terminem.

No que diz respeito aos mecanismos de navegação, estes são parametrizados por conjuntos de transformação segundo estratégias de reposicionamento de localizadores durante as atividades de busca e substituição. Para reposicionamento de localizadores em atividade de busca de transformações, estão disponíveis as estratégias de navegação ascendente, descendente e seqüencial. Para reposicionamento de localizadores após atividades de substituição, estão disponíveis as estratégias de passo único e exaustiva. Na estratégia de passo único, após a substituição do conteúdo de um localizador pelo conteúdo de uma nova instância de substituição, este novo conteúdo não é mais visitado durante as próximas atividades de aplicação de transformações do conjunto de transformações. Na estratégia exaustiva, logo após a substituição, o conteúdo da nova instância é selecionado como contexto candidato para aplicação de novas transformações. Estas estratégias podem ser aplicadas tanto para conjuntos de transformações horizontais como para verticais.

O Draco-PUC suporta o uso de pré-condições e pós-condições em suas transformações, através de pontos de controle genéricos sobre regras, conjuntos e transformadores. Cada ponto de decisão do MT possui um mecanismo de escape que permite a comunicação com elementos externos. A comunicação é feita através de comandos descritos na linguagem C++. A organização destes pontos de controle é representada na Figura 3, onde cada um dos elementos de uma solução transformacional em Draco-PUC está representado, juntamente com seus pontos de controle.

Para a fase de tentativa de aplicação de um padrão de busca (fase de *match* de um *transform*), estão disponíveis os pontos de controle *pre-match*, *match-constraint* e *post-match*. Para a fase de aplicação de um padrão de substituição (fase de *apply* de um *transform*), estão disponíveis os pontos de controle *pre-apply* e *post-apply*. Para um conjunto de transformações (*Set Of Transforms*), estão disponíveis os pontos de controle *initialization* e *end*. Para um transformador (*transformer*) estão disponíveis os pontos de controle *declaration*, *initialization* e *end*. Todos os pontos de controle recebem um valor correspondente ao estado do MT e um ambiente de variáveis.

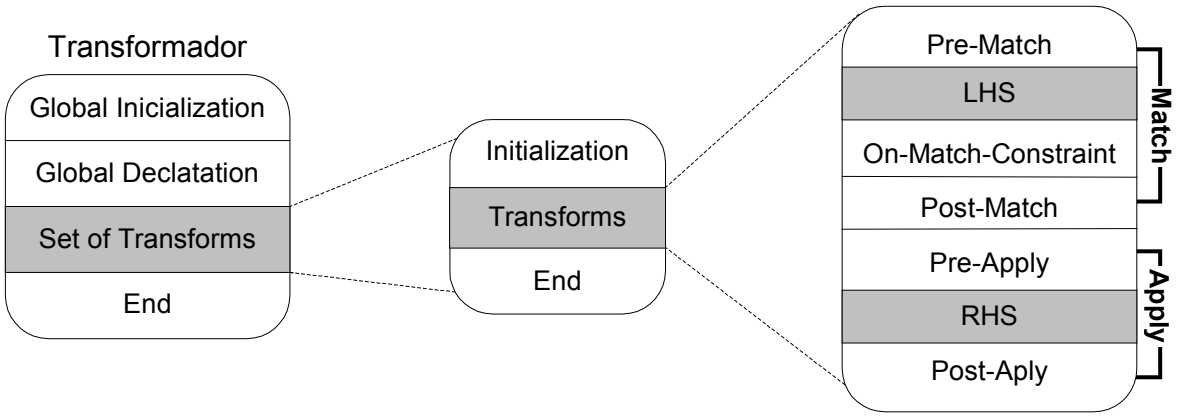


Figura 3. Estrutura de pontos de controle de um transformador Draco-PUC

Na construção dos transformadores também estão disponíveis comandos de acesso à base de conhecimento, acesso à área de trabalho compartilhada e de fluxo de controle entre transformações. Para uso das bases de conhecimento, existem comandos de inserção e remoção de cláusulas, limpeza total do conteúdo e consultas. Para acesso a áreas de trabalho compartilhadas, existem comandos para criação, remoção, limpeza total além da transferência e cópia de conteúdos entre áreas. Para fluxo de controle entre transformações existem comandos para chamar um conjunto de transformações sobre o conteúdo de uma variável padrão ou sobre o conteúdo de uma área de trabalho.

Foram realizadas diversas experiências na utilização do Draco-PUC [Nov01, Nog01a, Nog01b, Mor01, Pra00, Bra00, Jes99a, Jes99b, Fuk99a, Fuk99b, Pra98], nas quais os pesquisadores comprovam a eficiência de seu uso e relatam algumas dificuldades encontradas durante a realização dos trabalhos. Sant'Anna em [San99] apresenta um estudo sobre as características do Draco-PUC, relata as experiências realizadas e dificuldades encontradas pelos pesquisadores. Dentre as dificuldades encontradas no relato de Sant'Anna, estão a edição e depuração das gramáticas livres de contexto e do código dos transformadores. Outro problema citado é a organização do uso das transformações, tanto para validação e testes, quanto para o uso dos transformadores criados. Sant'Anna propõe a construção de um novo sistema denominado SpinOff para solução dos problemas apresentados. O SpinOff possui características de orientação a componentes que deverá facilitar sua utilização. Contudo, por tratar-se de uma nova proposta, demandará um grande esforço até adquirir a atual estabilidade do Draco-PUC.

Nas próximas seções são abordadas as linguagens envolvidas neste trabalho.

2.4 DataFlex Procedural (DFP)

A linguagem DataFlex é de autoria da DataAccess Corporation, cuja primeira versão comercial foi lançada em 1987, com o DataFlex 2.2, sendo apresentada como uma solução de custos baixos, facilidade de desenvolvimento e boa performance, na construção de aplicações para diversas plataformas, como UNIX, XENIX, VAX/VMS, DOS, Netware e OS/2, mantendo um código fonte totalmente portável entre elas.

Com esta característica chegou a se tornar líder em ambientes UNIX em empresas de médio porte no Brasil. A Figura 4 mostra o percentual de utilização da linguagem DataFlex no UNIX, comparado com similares. A pesquisa foi realizada em empresas de médio porte em 1994 pelo IDC Brasil Pesquisa [Fle95]. Na pesquisa, "*Progress*", "*Oracle*" e "*Informix*" estão caracterizados como linguagens de programação, provavelmente devido à formulação do questionário com enfoque comercial.

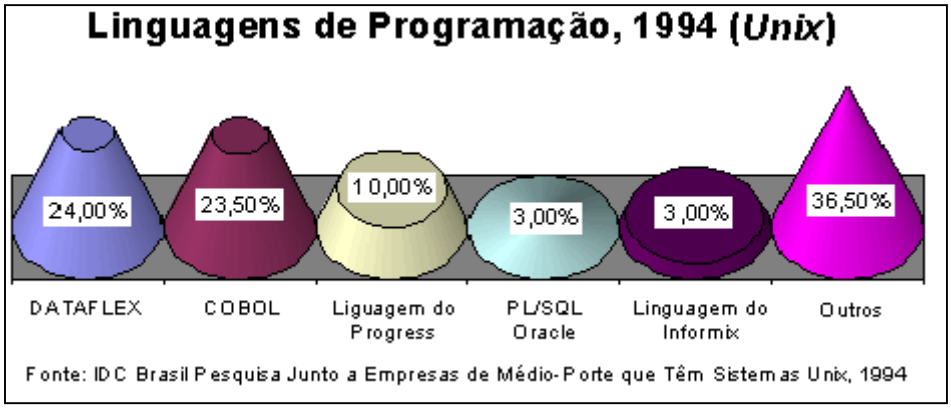


Figura 4. Comparação da utilização de linguagens em ambiente UNIX [Fle95]

A primeira versão do DataFlex que alcançou uma grande aceitação no mercado, foi a 2.3b lançada em 1988. Em 1989 o DataFlex na versão 2.3b se torna líder no mercado Brasileiro e Australiano de linguagems de programação para sistemas operacionais UNIX. Esta vesão ficou também entre as mais utilizadas para UNIX, nos EUA e Europa, no ano de 1989, mantendo-se nesta posição durante 4 anos. Das características do DataFlex na versão 2.3b, que justificaram a grande aceitação, destaca-se a de ter uma grande integração com seu SGBD. Outro fator relevante é a alta produtividade no desenvolvimento, principalmente de aplicações comerciais, devido à existência de macro-comandos e uma grande facilidade no desenho das telas e relatórios.

Em 1990 foi lançada a versão 3.0 do DataFlex com características de orientação a objetos, que se propunha manter a compatibilidade com os códigos gerados em DataFlex 2.3b, permitindo até mesclar em um mesmo programa trechos de código procedural e orientado a objetos. Devido à instabilidade do produto, a grande maioria do mercado continuou a trabalhar com DataFlex 2.3b, ignorando os benefícios da orientação a objetos. Diversas versões foram lançadas em seguida. Finalmente a versão 3.01b demonstrou recuperar estabilidade, mas devido a algumas incompatibilidades nos critérios de tratamento do código, em se comparando com a versão 2.3b, desenvolvedores continuaram trabalhando com a versão antiga. Na versão 3.05 o DataFlex passou a suportar o tratamento de *"console mode"* em DOS e Windows, otimizando o uso de memória, e construção de interfaces gráficas. Esta versão não utilizava as API´s do Windows, uma vez que todos os componentes foram construídos pela DataAccess para melhor portabilidade a outros ambientes gráficos. Por força do mercado, e necessidade de lançar um produto que competisse com as demais linguagens de desenvolvimento em ambientes Windows, a DataAccess reviu seu posicionamento de não utilizar as API´s, e lançou o Visual DataFlex, com o qual recuperou a competitividade no mercado Windows, mas perdeu a possibilidade de porte entre ambientes, e quebrou a compatibilidade de código com as versões anteriores.

Para este trabalho está sendo caracterizado como ***DataFlex Procedural***, o conjunto de códigos compiláveis com compatibilidade entre as versões DataFlex 2.3b, DataFlex 3.0, DataFlex 3.05 e DataFlex 3.1. Apesar das versões superiores a 3.0 reconhecerem uma forma de orientação a objetos, a versão 2.3b reconhece apenas o paradigma de programação procedural.

A linguagem DataFlex Procedural tem forte estrutura de checagem de tipos. É compilada e exige um ambiente *run-time* para ser executada. Os comandos são formados a partir de declarações, funções, frases, *handles* e operadores. A linguagem é macroexpansível, propiciando a criação de novos macro-comandos, usando os comandos básicos da linguagem ou estendendo os macro-comandos.

Possuem comandos para repetições, atribuições, seleções, cálculos básicos, e macro-comandos para instruções complexas, escritas em linha de programação, que são expandidos em tempo de compilação. Macro-comandos são normalmente usados no acesso e manutenção do banco de dados e na geração de relatórios.

O ambiente DataFlex tem gerenciadores de banco de dados, programas de manutenção, relatórios e menus. Tem editor de texto próprio, gerenciador de teclas de função, recuperador de banco de dados além do *Run-Time* de execução. O ambiente é portável para os seguintes ambientes operacionais: MS-DOS e compatíveis, PC-Net, CP/M e compatíveis, CP/M – 86 e compatíveis, Concurrent DOS, Novell, Amplinet, Net-MB, Multilink, Lanlink, Tapestry, 3COM+, Xenix, Unix e Vax.

Espaços em branco, tabulações, caracteres de controle de fim de linha e comentários não são significativos, exceto como conteúdo de *strings* ou para separar símbolos terminais (*tokens)*. O caracter de fim de linha marca o fim de uma declaração simples. Declarações longas podem ser continuadas na linha seguinte colocando um ponto-e-vírgula (;) no fim da linha anterior. Uma declaração não deve ultrapassar 255 caracteres.

2.4.1 Anatomia de um programa em DFP

A Figura 5 mostra um programa típico em DFP, onde em (1) tem-se exemplos de telas, compostas por informações textuais e janelas de interação. Em (2) exemplos de comandos básicos da linguagem, como a declaração de variáveis “string aux_str” e abertura de tabelas do Banco de Dados (BD) “open autosde”. Em (3) é apresentado um exemplo de macro-comando, no caso o macro-comando “Enter”, que controla a entrada de dados para uma ou mais tabelas do BD através de uma ou mais telas de interação. Em (4) é apresentado um exemplo de tratamento de eventos neste caso, é definido um bloco de código a ser executado quando for pressionada a tecla “escape” durante uma interação com o usuário.

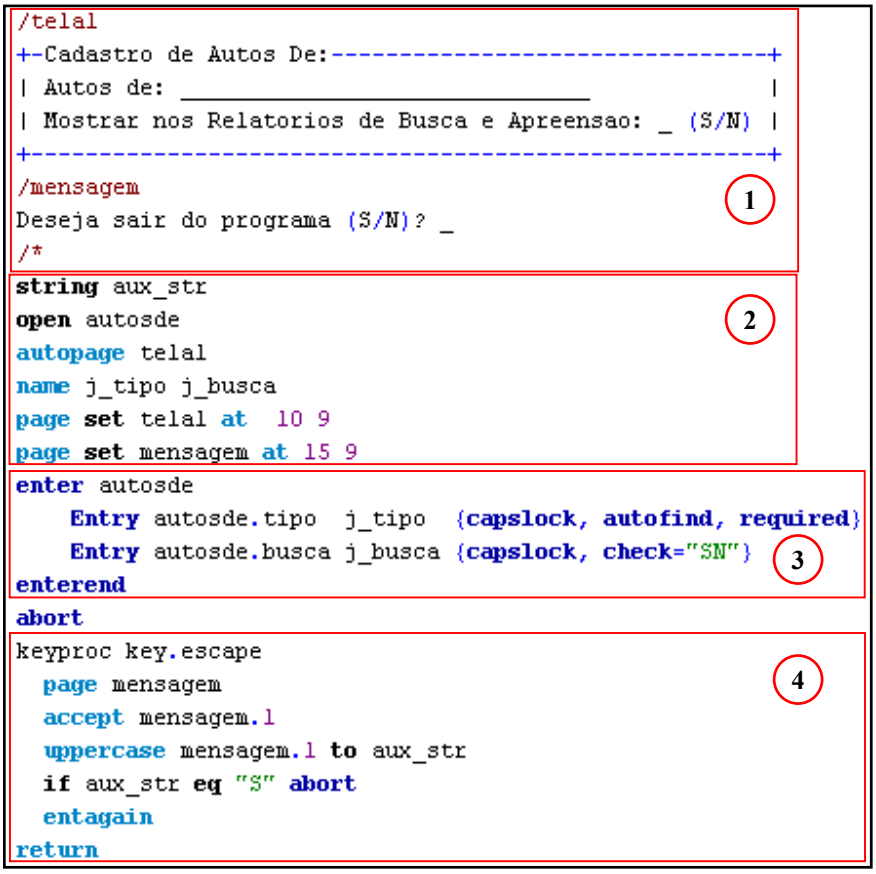


Figura 5. Exemplo de um programa em DFP

2.4.2 Operadores

Os operadores são classificados em Binário Aritmético ("+", "-", "*" e "/"), Unário Aritmético ("-"), Relacionais ("EQ", "NE", "LT", "GT", "LE", "GE", "IN" e "Matches") e Booleanos ("Not" e "~"). A precedência de operadores é redefinida usando "(" e ")" parênteses. Uma expressão é avaliada da esquerda para a direita.

2.4.3 Tipos e Estruturas de Dados

Na declaração de uma variável é especificado seu tipo. O tipo de uma variável determina o conjunto de valores e operações que a variável vai aceitar. Os tipos reconhecidos são "Integer", "Number", "Real", "Date" e "String".

Arquivos de dados: Um arquivo de dados é declarado pelo comando "OPEN", e este comando importa a definição contida no arquivo com a extensão ".FD" ex: "Open cliente". Nesse caso, a definição do arquivo encontra-se em "cliente.fd", com a relação de

campos que compõem o arquivo de dados. A criação, tratamento e manutenção dos arquivos de dados é feita através de um utilitário chamado "DFFile".

2.4.4 Telas e Janelas

O DataFlex Procedural desenha as telas em formato textual, com campos de interação que permitem a entrada de dados chamados janelas. As telas são definidas pelo prefixo “/” na primeira coluna de uma linha, seguido do nome da tela. Para finalizar a definição da tela, usa-se “/*” na primeira coluna de uma linha ou inicia-se a definição de outra tela.

As janelas são desenhadas dentro das telas, usando o caracter “_” (underline). Podem ser tratadas como variáveis dentro de um programa e possuem definição de tipo de dado. Esta definição é caracterizada ao desenhar a janela, através de caracteres de controle. Para janelas do tipo "string", cada "_" é equivalente a uma letra. O caracter "@" colocado em qualquer ponto na janela indica que espaços em branco no final da seqüência devem ser desprezados. Para janelas do tipo "inteiro" e "numérico", cada "_" é equivalente a um dígito numérico, e o "." (ponto) indica a posição do separador decimal. Para janelas do tipo data, utiliza-se a "/" (barra) para formatação, como por exemplo: "__/__/__" ou "__/__/____".

2.4.5 Visibilidade e Tempo de Vida

Uma variável é visível posterior a sua declaração e durante o tempo de vida do programa que a declarou. Não existe variável local ou restrita.

2.4.6 Procedimentos

O tratamento de procedimentos é implementado na linguagem através da criação de rotinas. Uma rotina é um bloco nomeado de código incluindo seqüências de declarações e comandos. Uma rotina é identificada por um rótulo caracterizado pelo caracter “**:**” no final de uma linha, e termina com a declaração Return.

2.4.7 Passagem de Parâmetros

Não existe passagem de parâmetros na chamada de procedimentos. Existe

passagem de parâmetros por valor na chamada de subprogramas, usando o comando "Chain".

2.4.8 Estruturas de Controle e Fluxo de Execução

A linguagem DataFlex Procedural tem estruturas de seqüência, seleção e repetição para controle do fluxo de execução.

Seqüência: Um programa consiste em uma lista de instruções e blocos de instrução que são executados seqüencialmente. Um bloco agrupa várias declarações e comandos dentro de um módulo simples de processamento.

Os comandos que iniciam blocos são: "Begin – End", "For – Loop", "Repeat – Until", "While – End", "Enter – EnterEnd", "EnterGroup – EndEnterGroup" e "Report - ReportEnd".

Seleção: A forma mais básica de seleção é o comando "If". Se o valor da expressão seguinte ao comando "If" for verdadeiro, as instruções seguintes são executadas. Caso contrário, são executadas as instruções seguintes à opção "Else". A linguagem aceita até nove níveis de aninhamento de comandos "IF".

O comando "Case Begin" é usado para decidir, entre várias alternativas, qual será o fluxo do programa. As alternativas são definidas pela cláusula "Case" se nenhuma condição for válida é executado o bloco definido pelo "Case Else".

Repetição: A linguagem DataFlex Procedural possui três comandos de repetição: "While", "Repeat" e "For". O comando "Repeat" inicia um bloco para ser processado repetidas vezes até ser encerrado pelo comando "Until". O comando "While" inicia um bloco para ser processado repetidas vezes até ser encerrado pelo comando "End". A diferença entre estes dois comandos é que o "Repeat" executa o teste posterior, na cláusula "Until", e o "While" executa o teste anterior. O comando "For" inicia um bloco de comandos, controlado por uma variável, e repetido pelo comando "Loop".

2.4.9 Gerenciamento de Memória

O próprio ambiente de *run-time* se encarrega da coleta de lixo. No acionamento

de subprogramas, com a utilização do comando "Chain", a área de memória, destinada ao gerenciamento de arquivos, é acumulada em uma pilha. Porém, se o comando for utilizado com a cláusula "Export_Files", a área de memória, destinada ao tratamento de arquivos, passa a ser compartilhada pelo subprograma.

2.4.10 Tratamento de Eventos

Em DataFlex Procedural, é possível definir a execução de eventos baseando-se no acionamento de teclas de função pré-definidas (F1-F10, Return, Escape e Tab). Estes eventos são caracterizados pela utilização da declaração "KeyProc Key....".

Mais detalhes sobre DataFlex Procedural são encontrados em [Dat91a, Dat91c, Dat92 e Dat95e, Gol92]. Na próxima seção é abordada a linguagem VDFOO.

2.5 Visual DataFlex Orientado a Objetos (VDFOO)

O Visual DataFlex Orientado a Objetos (VDFOO) é pré-compilada para o ambiente de desenvolvimento gráfico, em que se utiliza a orientação a objetos como paradigma de desenvolvimento, exigindo um ambiente *run-time* para ser executada. Porém, ao contrário da DataFlex Procedural, o *run-time* tem distribuição livre. Os comandos são formados a partir de declarações, funções, frases, *handles* e operadores. Os comandos da linguagem podem ser expandidos com a criação de macro-comandos [Dat98a, Dat98b, Dat98c, Dat91b, Dat95a, Dat95b, Dat95d].

O tratamento de extensões é livre, porém normalmente utiliza-se “.src” para programas fonte, “.PKG” ou “.INC” para pacotes de funções ou bibliotecas de classes, “.VW” para visões (telas de integração), “.DG” para telas de diálogo, “.SL” para listas de seleção, “.DD” para dicionários de dados (regras de negócio) e “.RL” para relatórios. Após submeter o código fonte à compilação, é criado o código objeto, com a extensão “.VDF” (versão 4 e 5), “.VD6” (versão 6) e “.VD7” (versão 7), executável pelo *runtime* (DFRUN32.EXE).

Existe uma forte ligação sintática entre o VDF e o DataFlex Procedural tratado na seção anterior. Cerca de 80% das características sintáticas do DataFlex Procedural são suportadas pelo Visual DataFlex Orientado a Objetos. Porém como se trata de uma linguagem

Orientada a Objetos, o código seqüencial utilizado no DataFlex Procedural só é reconhecido dentro de classes e objetos, conforme os princípios da orientação a objetos.

Segue-se uma apresentação do VDFOO segundo os princípios da orientação a objetos.

2.5.1 Classes

Uma classe é um tipo estruturado consistindo em um número fixo de componentes. Os componentes possíveis são métodos, propriedades e objetos privados. Os componentes de classe são às vezes também chamados membros. É declarada com "Class <corpodaclasse> End_Class". Cada classe herda todos os componentes de sua superclasse. Cada método herdado poder ser estendido, anulado ou cancelado dentro da nova declaração de classe. Pode-se definir novas propriedades, métodos e objetos privados dentro do método construtor da classe. A Figura 6 mostra um exemplo de declaração de uma classe, onde é definida uma nova classe "cStorageForm", que herda da classe "Form" contendo um objeto privado do tipo vetor chamado "oStorageArray". Dois métodos são definidos como interface para "oStorageArray". DoAppendValue que recupera o valor do objeto "Form" e anexa no vetor de "oStorageArray" e "StorageValue" que retorna um valor indexado do vetor.

```
Class cStorageForm  is a Form
   Procedure Construct_Object
       Forward Send Construct_Object
       Object oStorageArray  is an Array  // O objeto oStorageArray é componente
       End_Object                         // da classe cStorageForm
   End_Procedure
   Procedure DoAppendValue
       Integer iItemCount
       String  sFormValue
       Get Value       of Self  0        To sFormValue
       Get Item_Count  of oStorageArray  To iItemCount
       Set Array_Value of oStorageArray  iItemCount  To sFormValue
   End_Procedure
   Function StorageValue  Integer iItem  Returns String
       String sReturnValue
       Get  String_Value  of oStorageArray  iItem  To sReturnValue
       Function_Return sReturnValue
   End_Function
End_Class
```

Figura 6. Sintaxe de declaração de uma Classe

2.5.2 Declaração de Instâncias (Objetos)

Um objeto é uma instância de uma classe. Para utilizar as funcionalidades de uma classe, esta deve ser instanciada em um ou mais objetos.

O VDFOO suporta a criação de novas propriedades dentro de uma declaração de objeto. Estas propriedades são chamadas de propriedades de instância ou de objeto. As propriedades de objeto são diferentes para cada objeto declarado e não são compartilhadas por outros objetos da mesma classe.

Além de criar propriedades de objeto, o VDFOO suporta a criação de novos métodos dentro de uma declaração de objeto. Esses métodos são diferentes para cada objeto declarado e não são compartilhados por outros objetos da mesma classe.

2.5.3 Operações (Procedimentos e Funções)

São suportados ambos os conceitos através de declarações "Procedure <corpodometodo> End_Procedure" ou "Function <corpodometodo> End_Function". A diferença entre as duas construções é que apenas a função tem um valor de retorno.

Uma declaração de "Procedure" associa um identificador com um bloco de código como um procedimento. Pode ser chamado dentro do programa por "Send".

Uma declaração "Function" associa um identificador com um bloco de código como uma função. Pode ser chamada no programa dentro de uma expressão.

As operações, podem receber zero ou mais parâmetros. Cada parâmetro se comporta como uma variável local que é inicializada pela declaração que chama a operação. Os tipos de dados de cada parâmetro e o valor de retorno podem ser qualquer dos tipos de dados reconhecidos pela linguagem.

Os parâmetros são passados por valor. Não existe passagem de parâmetros por referência. A passagem de parâmetros por valor significa que o parâmetro real é avaliado e o valor de seu resultado é copiado para o parâmetro formal correspondente na unidade de programa chamada.

2.5.3.1 Tipos e Estruturas de Dados e Operadores

Na declaração de uma variável, é especificado seu tipo. O tipo de uma variável determina o conjunto de valores e operações que a variável vai aceitar. Os tipos reconhecidos são ("Integer", "Number", "Real", "Date", "String", "Pointer", "Handle", "Dword" e tipos complexos através da declaração "Type ... FIELD ...FIELD... End_Type").

Um arquivo de dados é declarado pelos comandos "Declare_DataFile", ou "OPEN", sendo que estes comandos importam a definição do arquivo contida em uma extensão ".FD". Os arquivos de dados utilizados por DataFlex Procedural são reconhecidos integralmente pelo VDFOO, mas o inverso nem sempre é verdadeiro, pois VDFOO reconhece um conjunto maior de tipos de dados.

Os operadores em VDFOO são classificados em Binário Aritmético ("+", "-", "*", "/", "Max" e "Min"), Unário Aritmético ("-"), Relacionais ("=", "<>", "<", ">", "<=", ">=", "IN" e "Matches"), Booleanos ("Not", "~", "AND" e "OR"), Operadores de String ("+", "-" e "*") e Operadores de Bits ("Iand" e "Ior"). Uma expressão é avaliada da esquerda para a direita. Para assegurar a ordem correta de avaliação de uma expressão, deve-se utilizar “(“ e “)” parênteses.

2.5.4 Encapsulamento

A visibilidade das variáveis depende da sua declaração e pode ser "local" ou "global". Variáveis são locais quando definidas dentro de métodos (*procedure* ou *function*). Variáveis locais possuem visibilidade restrita ao método que as definiu e seu tempo de vida é igual ao tempo de execução deste método. Variáveis globais são visíveis em todo o programa, e são definidas fora de um método, ou identificada por "global", seu tempo de vida é igual ao tempo de vida do programa.

2.5.5 Herança

O VDFOO suporta o conceito de herança simples. Na declaração de uma classe é definida sua superclasse com “is a” ou “is an”, como por exemplo “Class advogado is a DataSet”. Neste exemplo a classe “advogado” herda todos os atributos, métodos e associações da classe “DataSet”.

A herança múltipla não é suportada, porém existem os conceitos de importação de métodos e atributos de mais do que uma classe. Uma declaração "Import_Class_Protocol" é utilizada para importar os métodos e atributos de uma classe, como por exemplo, tem-se a classe A, que estende a classe B, por herança. A classe A pode ainda importar os protocolos das classes C e D. Assim a classe A possui todas as funcionalidades definidas em B, sua super classe e em C e D, classes importadas. O VDFOO trata os métodos e atributos de C e D como se fossem criados diretamente na classe A e não como métodos herdados como os da classe B.

Propriedades e métodos declarados como privados "private" na classe que é importada, não ficam disponíveis na classe que a importou.

2.5.6 Composição (Todo-Parte) e Associações

As classes em VDFOO podem conter objetos de qualquer outra classe, denominados objetos privados. Toda vez que uma instância da classe é criada, seus objetos privados também são criados. Os objetos privados são ferramentas úteis para definir classes que têm requisitos de armazenamento e de comportamento e dados sofisticados.

Os objetos privados que são parte da definição de uma classe tornam-se atributos privados da classe, ou seja, são parte do projeto interno da classe. Sempre que uma instância da classe é destruída, qualquer objeto privado que ela contenha também é automaticamente destruído. Devem ser declarados no método de construtor da classe "Construct_Object".

2.5.7 Eventos

Os eventos são uma implementação especializadas dos métodos normais. Geralmente estão relacionados a interações do usuário, como o controle das ações realizadas com os dispositivos de entrada e saída, como o mouse e o teclado. Também podem ser definidos gatilhos ou chamadas predefinidas de eventos, durante a elaboração dos métodos de uma classe. Outros exemplos de eventos são os acionados por operações com o banco de dados.

Cada classe na biblioteca de classe VDFOO executa métodos de eventos predefinidos durante as operações importantes. Por exemplo, a classe "TreeView" possui

suporte a eventos como por exemplo: "OnCreateTree", executado quando um objeto de "TreeView" é criado; e "OnItemChanged", executado sempre que for alterado o objeto em foco atual da treeview.

O tratamento de eventos facilita o projeto de um comportamento especializado para a classe durante as operações importantes. Podem criar vínculos entre o acionamento de eventos e a execução de operações, através de uma declaração "On_Key", que permite vincular o acionamento de qualquer seqüência de teclas à execução de um método.

2.5.8 Conexão de Mensagens

A linguagem VDFOO tem estruturas de seqüência, seleção e repetição com a mesma sintaxe das apresentadas para o DataFlex Procedural. Porém, para controle do fluxo de execução são acrescentadas as declarações de acionamento e conexão de mensagens, "Set", "Get", "Send", "Forward" e "BroadCast".

Outra característica do VDFOO é sua integração com um *Framework* denominado DataFlex Application *Framework*, detalhado na próxima seção.

2.6 DataFlex Application *Framework* (DAF)

Um *Framework* é um projeto genérico em um domínio que pode ser adaptado para aplicações específicas, servindo como um molde para a construção de aplicações no domínio. *Frameworks* são desenvolvidos para atender aplicações de um domínio. A reutilização de classes de um *Framework* é diferente da reutilização de classes de uma biblioteca. No *Framework* é prevista a reutilização de um conjunto de classes inter-relacionadas. Na biblioteca, as classes são utilizadas isoladamente, cabendo ao desenvolvedor estabelecer suas interligações.

Um *Framework* pode ser visto como uma arquitetura de software semidefinida que consiste de um conjunto de componentes e suas interconexões de tal forma a criar uma infra-estrutura de suporte pré-fabricada para o desenvolvimento de aplicações de um domínio específico [Lew95].

Uma vantagem de usar *Frameworks* é que o processo de criação de uma aplicação torna-se fácil e demanda menos conhecimento específico do domínio da aplicação. O desenvolvedor necessita apenas saber como usar o *Framework*, não requerendo conhecimento detalhado da sua complexidade, e reduzindo a implementação. Será necessário apenas o código adicional que corresponde às particularidades da solução procurada.

Segundo [Rog97] existem duas categorias de *Frameworks*:

- Horizontal: é mais abrangente e pode ser usado por mais tipos de aplicações. ToolKits GUI são exemplos de Frameworks horizontais, pois podem ser usados para construir interfaces de usuários para um grande domínio de aplicações; e
- Vertical: é mais específico, para um particular domínio. Por exemplo, um Framework para executar análise estatística de dados econômicos é específico para aplicações financeiras.

Existem três variáveis que podem ser usadas para distinguir um *Framework* horizontal de um vertical:

- O nível de generalidade que é a medida de quantas aplicações podem usar as potencialidades de um Framework;
- A porção média do Framework que é usada pela aplicação; e
- A porção média de código em uma aplicação que é implementada a partir do Framework.

Em [Tal96] é encontrada uma terceira categoria de *Frameworks,* denominada **Suporte**, que fornece serviços em nível mais básico, para acesso a arquivos ou *drivers* de dispositivos, que de certa forma são também horizontais. A distinção entre os tipos de *Framework*s não é radical. Pode-se combinar o que as classes têm de melhor nestes dois conjuntos, chegando-se a uma boa solução para grande parte das aplicações.

O DataFlex Application *Framework* (DAF) usado na estratégia é horizontal [Dat94]. Utiliza o padrão de desenvolvimento em camadas: interface, regras de negócio e

banco de dados [Dat95d, Dat98a, Dat98b, Her91]. Fornece adaptações para utilizar suas classes por herança e instanciação. Em qualquer uma das camadas pode-se estender as classes existentes e agregar as classes estendidas nas funcionalidades do *Framework* como, por exemplo, criar tipos especiais de componentes de interação com o usuário.

As funções pré-definidas também podem ser estendidas ou alteradas para o desenvolvimento específico da aplicação.

O DAF assume o controle de fluxo de execução, respeitando as regras de inter-relacionamentos das classes. Por exemplo, uma vez definido um conjunto determinado de valores possíveis para um atributo de uma classe persistente, na camada de regras de negócio, automaticamente é criado um objeto de seleção na camada de interação com o usuário.

Pode-se utilizar o DAF com Visual DataFlex Orientado a Objetos (VDFOO), DataFlex Caracter Mode Orientado a Objetos (DFCOO), que é uma versão da linguagem também orientada a objetos, porém com a construção de interface em formato caracter, ou ainda com o WebApp Server que possibilita o desenvolvimento para a internet através de tecnologia *Active Server Page* (ASP) ou *.Java Server Page* (JSP). Suporta também a integração com diversos sistemas de gerenciamento de banco de dados (SGBD) relacionais.

As três camadas do DAF, possibilitam uma grande flexibilidade no desenvolvimento, pois uma vez implementadas as regras de negócio, usando-se extensões da classe DataDictionary, pode-se simplificar a interface com o usuário.

A Figura 7 mostra a arquitetura do DAF, onde na primeira camada são apresentados os pacotes de componentes que podem ser utilizados para construção de interfaces com o usuário. Na segunda camada tem-se o pacote de componentes que gerencia a construção e utilização das regras de negócio. Finalmente na terceira camada encontram-se os pacotes de componentes para acesso ao banco de dados.

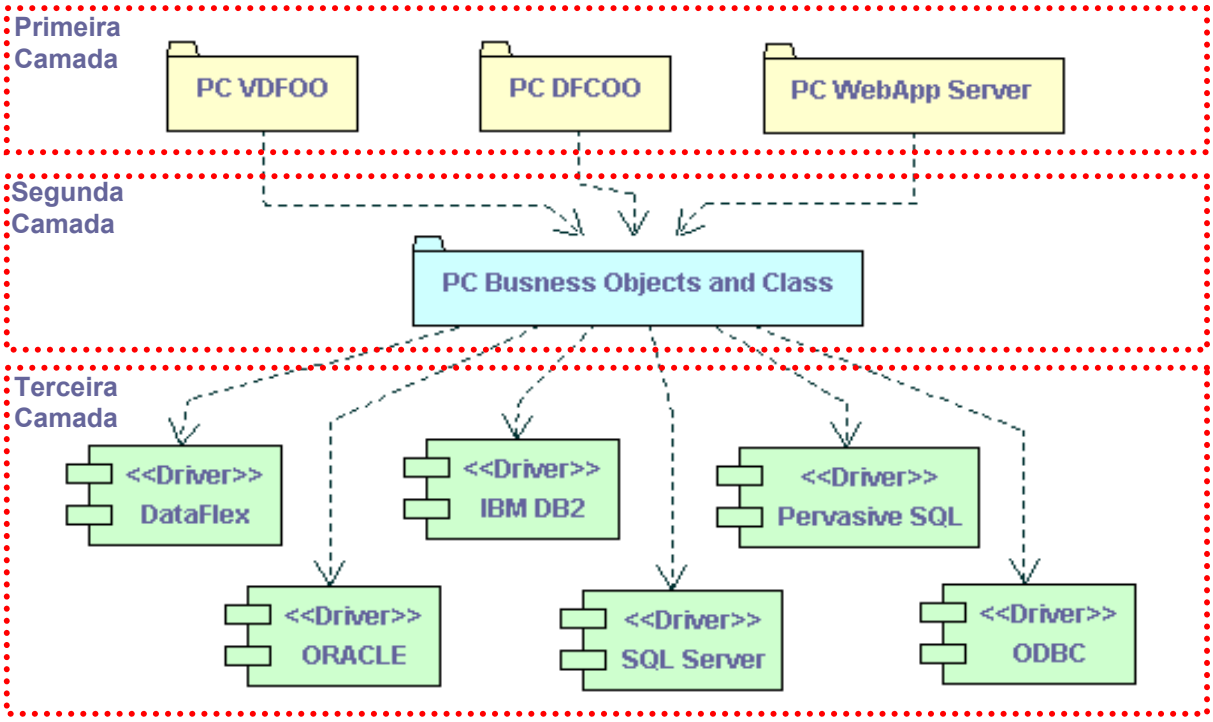


Figura 7. Arquitetura do DataFlex Application *Framework*

Na **Primeira Camada** o DataFlex Application *Framework* apresenta diversos Pacotes de Componentes (PC) de interface para diferentes plataformas de software. Em todos eles são disponibilizadas classes equivalentes, que auxiliam na construção da interface com o usuário, usando as regras de negócio da segunda camada.

O PC DFCOO é uma opção para construção da interface com o usuário em formato caracter, portável para diversas plataformas, como UNIX, LINUX, DOS e NetWare, integrado a linguagem DFCOO.

O PC VDFOO, é uma opção para o desenvolvimento com interface gráfica, em ambiente Windows 95/98/NT/2000 e XP, integrado com a linguagem VDFOO.

O PC WebApp Server destina-se ao desenvolvimento de aplicações que são executadas na internet e intranet, através de um browser, como Internet Explorer ou Netscape. O servidor de aplicações deve estar em um equipamento com sistema operacional NT ou Linux, mas os clientes podem estar em qualquer plataforma que possua um browser padronizado, que reconheça HTML, XML e Java. Permite também acesso a equipamentos que utiliza tecnologia WAP para construção de sua interface, como agendas com Windows CE, telefones celulares, coletores de dados, entre outros. Trabalha de forma semelhante ao

DataFlex AppServer, e pode ser instalado no mesmo servidor de aplicativos.

O DataFlex Application *Framework* possibilita que um sistema seja desenvolvido usando um ou mais componentes integrados na primeira camada. Assim, uma aplicação pode ser desenvolvida parte em Visual DataFlex, parte em DataFlex Console Mode e parte com DataFlex AppServer, compartilhando as mesmas regras de negócio e banco de dados das demais camadas.

A **Segunda Camada** é formada pelo PC *Busness Objects and Class* (BOC), que contêm componentes de tratamento de Dicionário de Dados (*DataDictionary*) e utilitários de gerenciamento. Os componentes de Dicionário de Dados suportam a definição das regras de negócio em seus métodos. Assim, é possível definir a essência do aplicativo em classes da camada regras de negócio, que possuem inter-relacionamentos com as demais camadas. Trabalhando com os requisitos do ambiente de negócio codifica-se no Dicionário de Dados as regras que controlarão o sistema. Pode-se definir regras para: criação, alteração e exclusão de objetos, relacionamento com outras classes, tratamento das propriedades de atributos e suas validações, formatação de dados para o usuário e regras de integridade. Esta implementação torna mais simples a manutenção nas camadas de interface e banco de dados.

A **Terceira Camada** é formada por um conjunto de componentes responsáveis pela conexão e acesso a diversos bancos de dados. O fornecimento de serviços padronizados para acesso ao banco de dados, assegurado pela implementação de interfaces predefinidas na classe *DataSet*, torna possível que as regras de negócio, definidas na segunda camada, utilizem diversos bancos de dados de forma transparente durante a implementação.

Após a definição das principais tecnologias envolvidas neste projeto, o capítulo seguinte mostra o Draco Domain Editor (DDE), construido para auxiliar elaboração e execução desta pesquisa.

Capitulo 3 Draco Domain Editor (DDE)

Com base nos relatos sobre as experiências realizadas com o Sistema Transformacional Draco-PUC [Nov01, Nog01a, Nog01b, Mor01, Pra00, Bra00, Jes99a, Jes99b, Fuk99a, Fuk99b, Pra98] e considerando as propostas realizadas por Sant´Anna, na construção do SpinOff [San99], foi construído o Draco Domain Editor (DDE).

Nos relatos de trabalhos realizados, a maioria das criticas na utilização do Draco-PUC, relacionam-se com a dificuldade de edição e execução dos domínios. São raros os relatos de falhas ou restrições na máquina transformacional Draco-PUC. De fato, a maioria dos autores elogia a estabilidade e flexibilidade da máquina transformacional do Draco-PUC, mas critica a forma de construção das gramáticas e transformadores, além de reportarem dificuldades o uso das transformações criadas.

Desta forma, a construção do DDE busca suprir uma das principais dificuldades relatadas, que é a edição de domínios e transformadores no Draco-PUC. O DDE também propõe uma forma de organização para a realização das transformações através de *scripts*.

O DDE possui uma interface gráfica contendo Editores de Domínios, de Transformadores, de Scripts de Execução dos Transformadores e de Scripts de Transformação. Conta ainda com um editor de códigos genérico, que trata a sintaxe de diversas linguagens de programação como C++, Java, DataFlex e Delphi.

A Figura 8 mostra a interface geral do DDE, onde em (1) estão os menus de e de acesso rápido às funcionalidades do DDE. Em (2) tem-se um *browser* que permite o acesso rápido às gramáticas existentes, transformadores e scripts. Em (3) tem-se o editor gramática, e em (4) a visualização gráfica da gramática que está sendo editada.

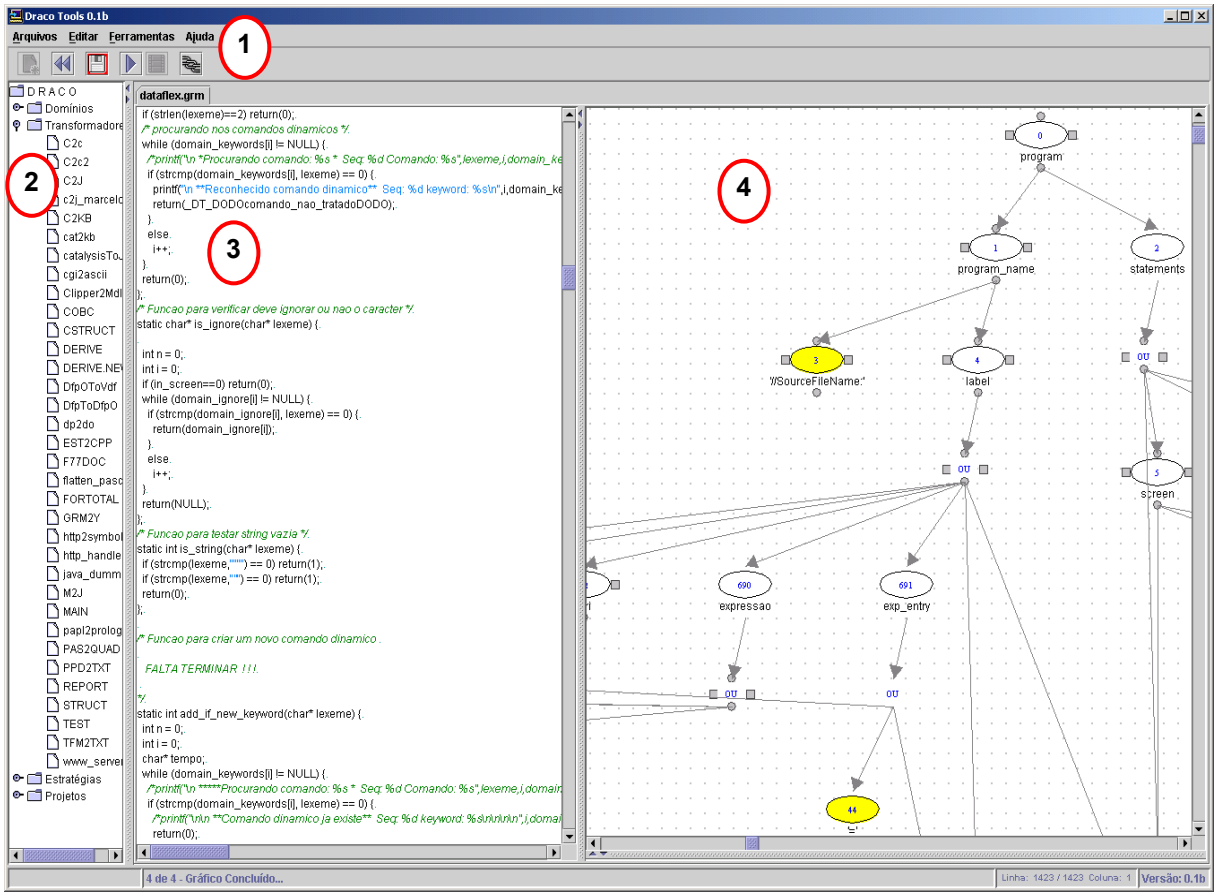


Figura 8. Interface do DDE

3.1 Editor de Gramáticas

O Editor de Gramáticas suporta a edição de gramáticas livres de contexto, sinalizando através de cores os símbolos terminais e as regras de produção da gramática. A gramática é apresentada na forma gráfica, através de um diagrama semelhante a uma árvore gramatical, com as regras de produção e os símbolos terminais sinalizados por cores.

A Figura 9 mostra a área de edição textual, com um trecho da gramática DataFlex Procedural. São sinalizados os símbolos terminais, os comandos de formatação para o *PrettyPrinter*, caracteres especiais relevantes para descrição de gramáticas no Draco-PUC, além dos comentários.

O reconhecimento e comparação de texto usam a interface da biblioteca *Document* do Java [Sun01]. Assim por exemplo, a classe *AbstractDocument* manipula e divide o código para o tratamento da sinalização de cores.

Figura 9. Área de edição textual da gramática

A representação gráfica da gramática é gerada com base na sua descrição, e possibilita uma navegação mais fácil entre as regras de produção, além de facilitar a visualização dos elementos da gramática.

Para construção do gráfico são filtrados alguns elementos que não são apresentados como comentários, espaços em branco, e comandos de formatação do *PrettyPrinter*. O filtro serve para reduzir a quantidade de nós da árvore gramatical apresentada e simplificar sua visualização. Os elementos restantes após a execução do filtro darão origem aos nós da árvore gramatical que é apresentada.

Para o posicionamento dos nós no gráfico, é calculada sua posição e verificada a existência de um nó apresentado que ocupe o mesmo perímetro. Caso seja encontrado um nó que ocupe parte do perímetro, a posição é redimensionada para um ponto abaixo do atual e novamente é feita verificação até que uma área livre seja encontrada. O perímetro testado consiste em um raio de valor "*r*" em relação ao centro do nó a ser apresentado, sendo que "*r*" é calculado com base no diâmetro horizontal do nome do nó.

A Figura 10 mostra a sinalização por cores na árvore gramatical, onde:

- Branco – Nó intermediário, sem problemas na derivação;
- Amarelo – Nó folha representando um token;

- Cinza – Nó intermediário cuja derivação leva a uma regra de expressão regular no analisador léxico. Por exemplo, a regra "ID : [A-Za-z]+;" pode retornar tokens formados pelo agrupamento de um ou mais caracteres de A a Z, maiúsculo ou minúsculo;
- Vermelho – Nó intermediário cujos filhos não foram definidos; e
- Verde – Denota uma ação semântica associada a uma regra de produção. Esta ação semântica é executada quando ocorre uma redução pela regra.

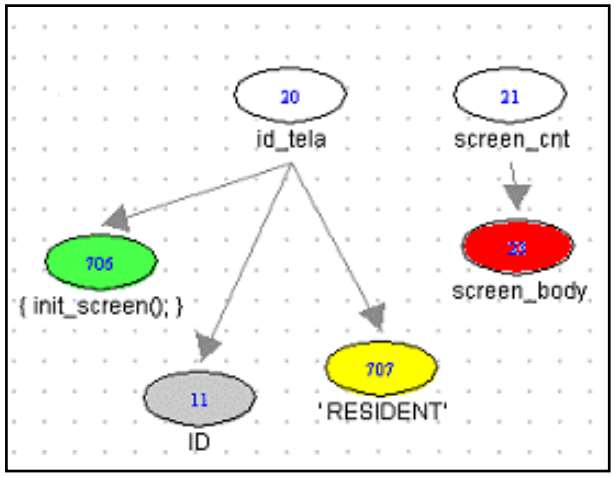


Figura 10. Sinalização de cores na árvore gramatical.

O DDE não trata problemas de conflitos do tipo "*shift reduce*" e "*reduce reduce*". Estes erros são tratados pelo PARGEN do Draco-PUC.

Navegando na árvore gramatical, o Engenheiro de Software, ao clicar em qualquer nó pode editar a regra de produção ou a correspondente ação semântica.

Depois de criada a gramática é submetida ao PARGER para geração do *parser* e do PPGEN para geração do *PrettyPrinter* [Fre96].

3.2 Editor de Transformadores

O Editor de Transformadores suporta a edição dos transformadores, sinalizando através de cores os elementos relevantes para a descrição de transformadores no Draco-PUC. É criada uma representação gráfica do transformador, adotando uma notação adaptada da proposta por Sant´Anna para Circuitos Transformacionais [San99].

A descrição de transformadores é baseada em regras de reescrita. Uma regra é composta por um padrão de busca, chamado de lado-esquerdo da regra (LHS - *left hand side*) e por um padrão de substituição, chamado de lado-direito da regra (RHS - *right hand side*).

Estes padrões são padrões sintáticos, descritos através de parametrização de fragmentos de DASTs, ou trechos de programas nas linguagens de representação disponíveis. A parametrização de DASTs ou programas é feita através de variáveis-padrão (*pattern-variables*) que generalizam as estruturas a serem encontradas e substituídas.

Nomes simbólicos são associados a estas variáveis para que possam ser referenciadas. As diversas regras de reescrita são organizadas em conjuntos, os quais por sua vez podem ser agrupados em módulos. Estes módulos são os blocos básicos utilizados pelo engenheiro de software que controla o processo de transformação. Para a codificação dos transformadores, é utilizada uma linguagem própria de descrição, baseada em C++.

Como mostra a Figura 11, o Editor de Transformadores do DDE reconhece os tokens da linguagem C++ e os comandos específicos da linguagem de definição dos transformadores.

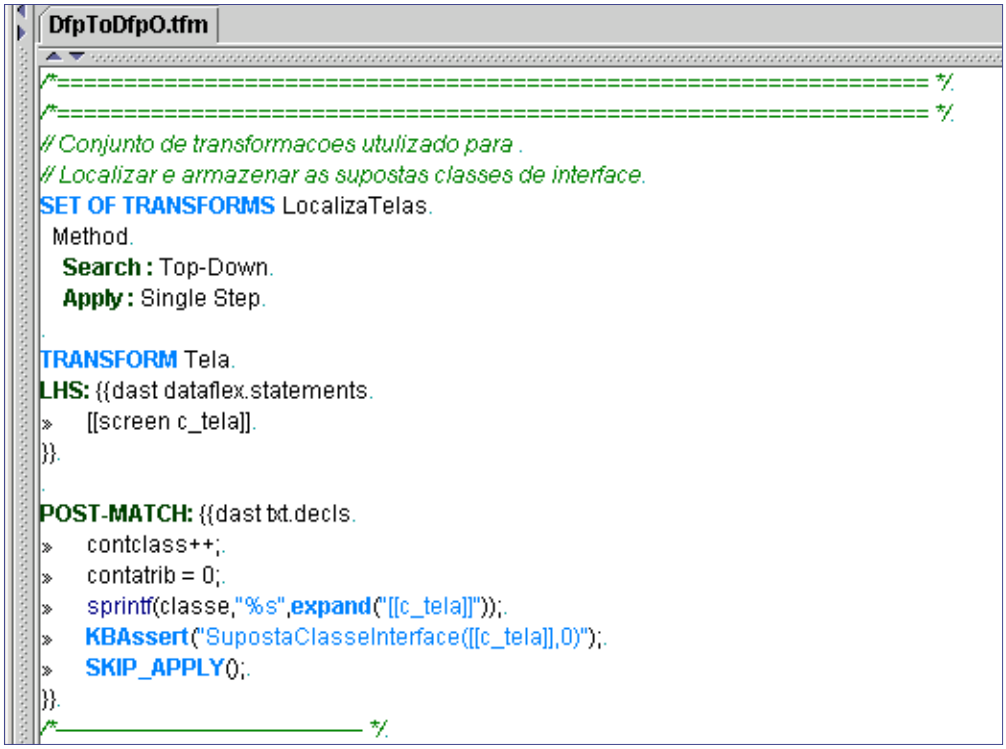


Figura 11. Área de edição textual do transformador no DDE

A geração de uma representação gráfica para o transformador auxilia em sua modelagem, implementação e manutenção. Como os transformadores têm que mapear cada regra gramatical do domínio original, dar o tratamento adequado e gerando outras regras gramaticais, a tendência é que os transformadores tornem-se estruturas de codificação complexas. A representação desta estrutura em um gráfico permite um entendimento mais fácil de sua lógica e facilita a navegação no código. Para a representação gráfica dos transformadores, são identificadas as estruturas, "Global Declaration", "Global Inicialization", "Global End", "Set Of Transformer", "Transform", "Template" e "KB". Estas estruturas são representadas e interligadas conforme os pontos de conexões reconhecidos no código do transformador. Este tratamento é equivalente ao reconhecimento descrito anteriormente para a representação da árvore sintática das gramáticas. A Figura 12 mostra a estrutura da árvore de representação do transformador, onde cada estrutura é apresentada através de símbolos que representam o conjunto de componentes transformacionais utilizados para implementação de um transformador.

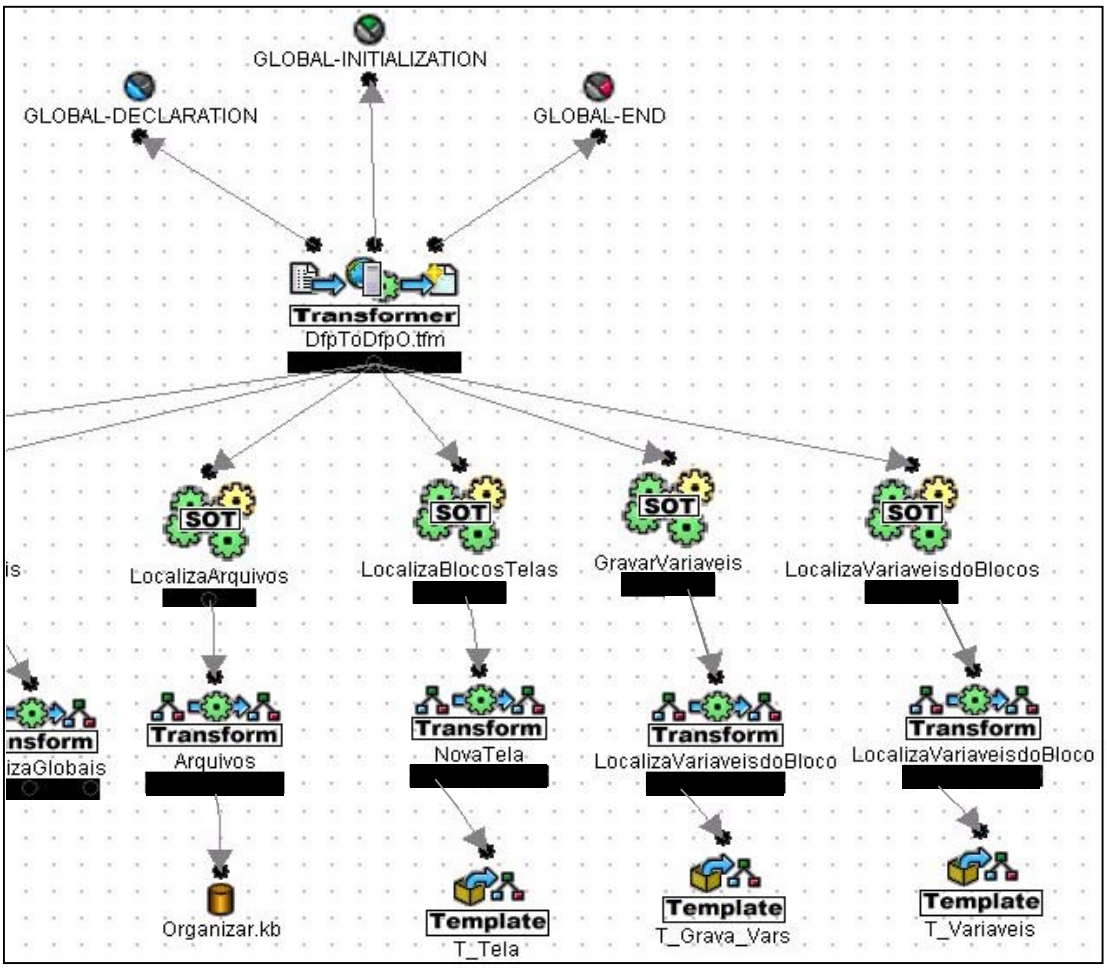


Figura 12. presentação gráfica da arquitetura de um transformador

O objeto *Transformer* é a identificação do arquivo que possui a lógica de transformação, os símbolos de *GLOBAL-DECLARATION*, *GLOBAL-INITIALIZATION* e *GLOBAL-END* são ligados ao objeto *Transformer*, servindo respectivamente para a declaração das estruturas auxiliares utilizadas durante a transformação. A inicialização destas estruturas e a definição das rotinas de finalização devem ser executadas no encerramento do processo de transformação. O transformador pode conter vários *SOT*s (*Set Of Transforms*), que são conjuntos de componentes *Transform*, que executam alguma ação específica. Um *SOT*, pode ser acionado na seqüência principal do transformador ou através de clamadas em componentes do tipo *Transform*, que neste caso contém uma cláusula "*Trigger:External*". O acionamento dos SOTs em componentes *Transform* possibilita o tratamento gradativo do código. Como mostra a Figura 13, um componente *SOT* pode acionar um conjunto de componentes *Transform*. Cada componente *Transform* é responsável pelo reconhecimento e tratamento de uma regra de produção da gramática descrita no Draco-PUC. Os componentes *Transform* podem acionar outros SOT's, e utilizar recursos disponíveis para a transformação, como componentes do tipo *Template* ou Bases de Conhecimento (KB). As conexões com o componente *Transform* tem posições específicas: objetos *Template* à esquerda, *KB* ao centro e *SOT*'s à direita.

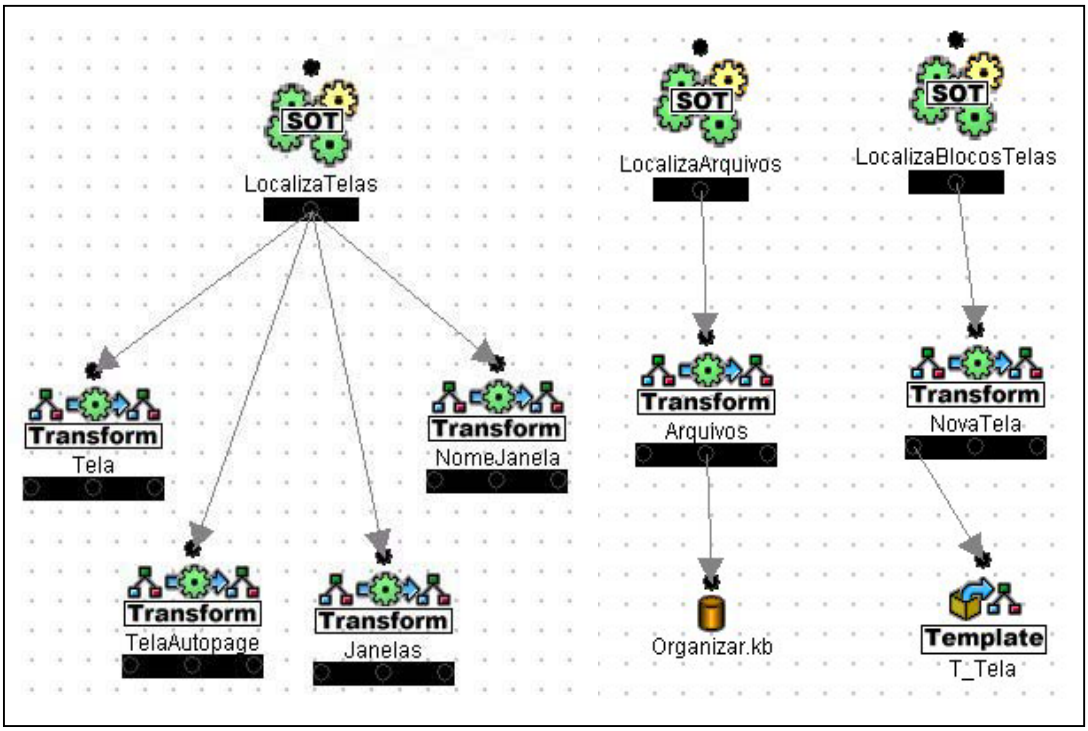


Figura 13. Ligações entre os componentes do transformador no DDE

3.3 Editor de *Scripts* de Execução dos Transformadores

Antes da implementação do DDE, para automatizar o uso do Draco-PUC, na aplicação dos transformadores, eram utilizados *scripts* do sistema operacional, para preparação do ambiente, cópia de arquivos, criação de diretórios entre outras operações necessárias a cada execução dos transformadores. Também eram utilizados *scripts* do Draco-PUC, em arquivos ".dsf", com instruções para execução de seus comandos.

O Editor de *Scripts* de Execução, permite a edição destes arquivos e propõe uma melhor organização das estruturas auxiliares ao uso das transformações. Com o DDE passam a ser incorporados aos domínios Draco-PUC, os *scripts* de execução dos transformadores.

Como o Draco-PUC trabalha com um conjunto de códigos, que são submetidos a um ou mais transformadores, cria-se um arquivo de *Script* de Execução contendo a seqüência de comandos que deve ser realizada com estes códigos. Uma vez criado o *script* de execução, são criados um ou mais *Scripts* de Transformação, onde são definidos os códigos que serão submetidos aos *Scripts* de Execução dos Transformadores.

Para a codificação dos *Scripts* de Execução dos Transformadores, foi definida uma meta-linguagem, na qual podem ser utilizados todos os comandos reconhecidos pelo sistema operacional, além dos comandos do Draco-PUC. Foram definidas estruturas como blocos de comandos e laços de execução (loop), para tratar os códigos.

A Figura 14 mostra um exemplo de *script* de execução, onde em (1) é apresentado o bloco principal de execução, delimitado por "BEGIN_MAIN" e "END_MAIN". Em (2) é apresentado um exemplo de inclusão de um bloco de *script* do sistema operacional, delimitado por "BEGIN_BAT" e "END_BAT". A chamada dos *scripts* do sistema operacional é feita pelo comando "CALL". Em (3) é apresentado um exemplo de *script* do Draco-PUC, delimitado pelo "BEGIN_DSF" e "END_DSF", chamado no código principal pelo comando "DRACO".

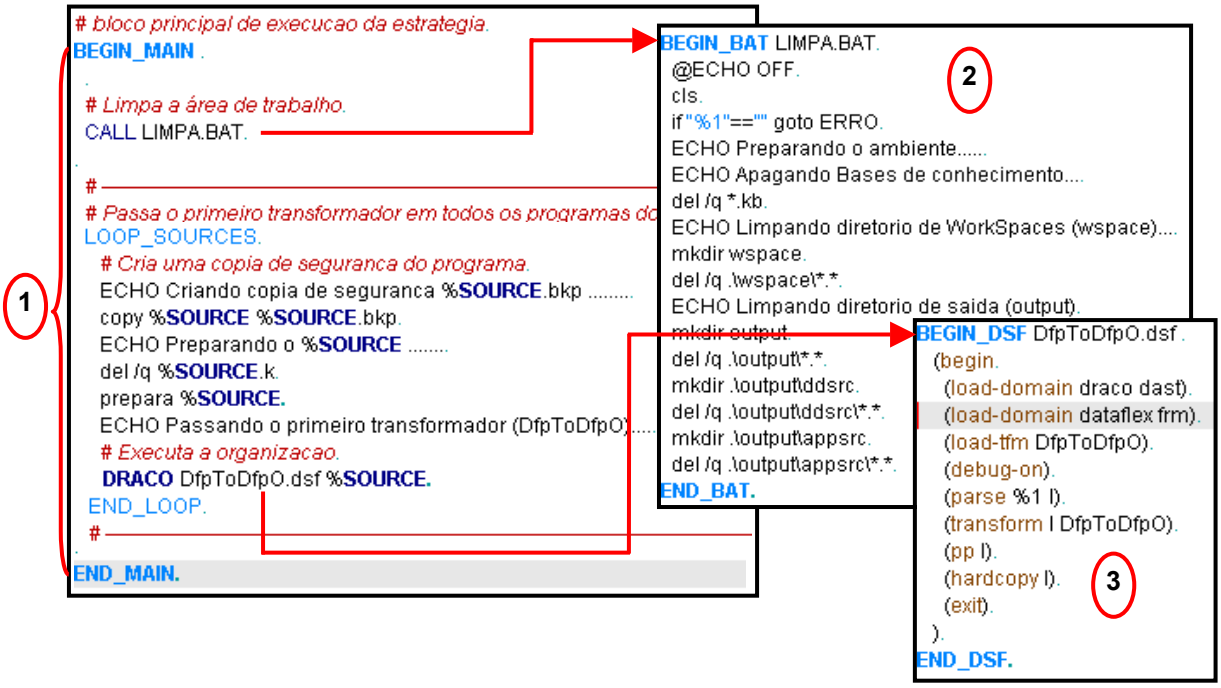


Figura 14. Estrutura do *Script* de Execução dos Transformadores

Outra estrutura apresentada ainda na Figura 14, é o "LOOP_SOURCES" e "END_LOOP", com o parâmetro "%SOURCE". Esta estrutura pode ser inserida dentro dos blocos "BEGIN_MAIN", "BEGIN_BAT" ou "BEGIN_DSF", e serve para permitir que um bloco de comandos seja executado para cada código fonte pertencente ao *Script* de Transformação através de macro substituição. Podem ainda ser inseridos pontos de interação do Engenheiro de Software durante a execução da estratégia com o comando "PAUSE_PROJECT {mensagem}". Este comando paralisa a execução até que o projeto seja novamente liberado, possibilitando a utilização de ferramentas auxiliares na transformação.

3.4 Editor de *Scripts* de Transformação

O Editor de Scripts de Transformação permite organizar uma seqüência de códigos e submetê-los a um ou mais *Scripts* de Execução.

A Figura 15 mostra um exemplo do *Script* de Transformação, onde o bloco "BEGIN_SOURCES" e "END_SOURCES" indica o conjunto de códigos a serem transformados e o bloco "BEGIN_EXECUTE" e "END_EXECUTE", o conjunto de *Scripts* de Execução que serão utilizados.

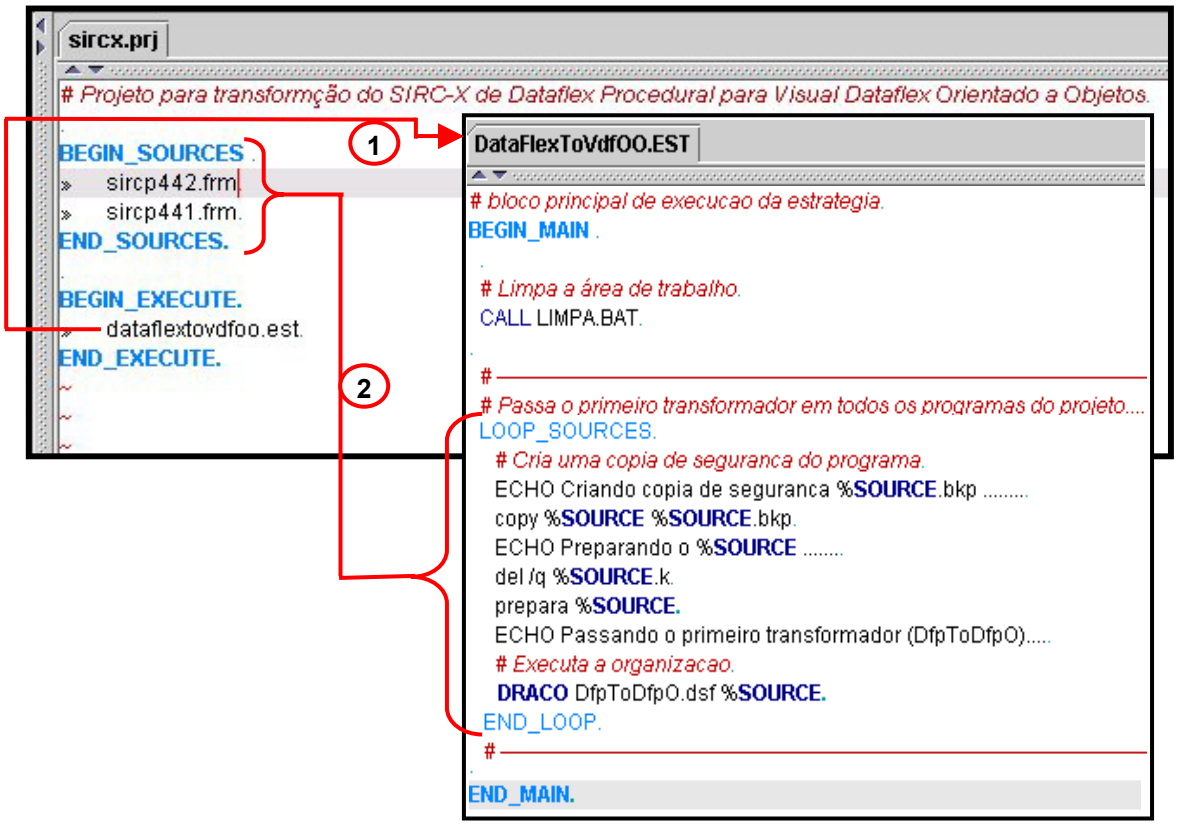


Figura 15. Uso do *script* de execução dos transformadores no *script* de transformação

Ainda na Figura 15, conforme indicado em (1), o nome do *Script* de Execução dentro do bloco "BEGIN_EXECUTE" será executado para todos os códigos de “BEGIN_SOURCES”. Conforme indicado em (2), dentro dos blocos "LOOP_SOURCES" e "END_LOOP", os parâmetros "%SOURCE" serão substituídos pelo nome do código constante no bloco "BEGIN_SOURCES" do *Script* de Transformação. Todo o conteúdo do bloco é executado para cada código fonte

A validação do DDE foi feita utilizando-o como ferramenta auxiliar ao Draco-PUC, na Construção da Transformação de DFP para VDFOO reusando o DAF, detalhada no próximo capítulo.

Capitulo 4 Construção da Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um *Framework*

A transformação de sistemas legados em DataFlex Procedural (DFP) para Visual DataFlex Orientado a Objetos (VDFOO), reusando o DataFlex Application *Framework* (DAF), é realizada em três passos: ***Organizar Código Legado,*** quando o código legado em DataFlex Procedural é organizado para facilitar a transformação para o paradigma orientado a objetos; ***Reimplementar Código Organizado,*** quando o código DataFlex Procedural organizado, obtido no passo anterior, é transformado para Visual DataFlex Orientado a Objetos reusando o DataFlex Application *Framework*; e ***Executar Código VDFOO*** quando o código Visual DataFlex Orientado a Objetos, reusando o DataFlex Application *Framework*, é executado para verificação dos resultados. A Figura 16 mostra os passos para a transformação com os controles, entradas, saídas e mecanismos de cada passo.

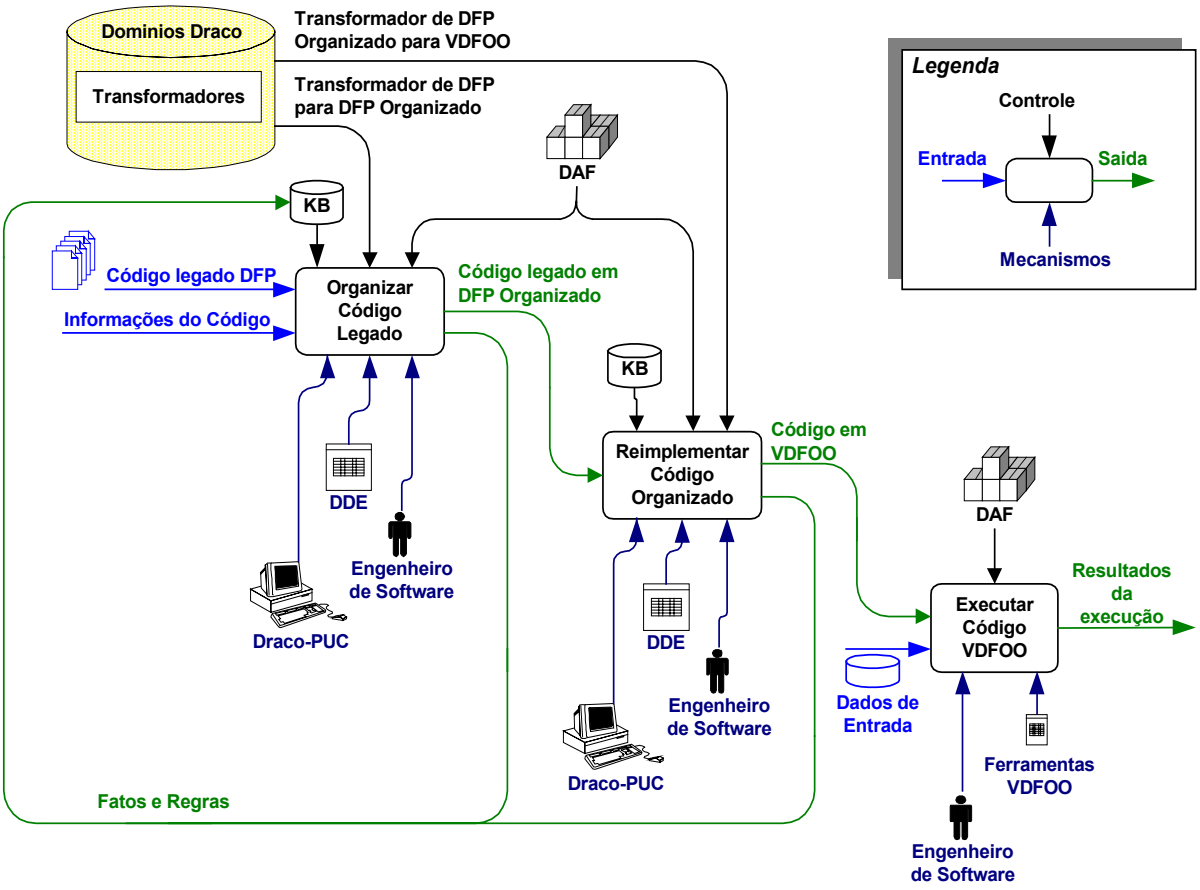


Figura 16. Estratégia de Transformação de DFP para VDFOO

No passo "**Organizar Código Legado**", entra-se com o código legado DFP e, aplicando transformações, obtém-se o código, ainda em DFP, porém organizado com uma estrutura que atende melhor aos princípios da orientação a objetos. O código legado DFP Organizado é particionado em supostas classes, com seus supostos métodos e atributos. São geradas regras sobre o controle de fluxo de execução do código legado, sobre a hierarquia de chamadas das Unidades de Programa (UP), sobre as supostas classes, atributos, métodos e associações em uma base de conhecimento (KB). Estas regras definem as conexões de mensagens, a seqüência de execução do código, o particionamento em supostas classes e o encapsulamento dos supostos métodos e atributos.

No passo "**Reimplementar Código Organizado**", parte-se do código DFP Organizado, obtido no passo anterior, e usando transformações, obtém-se um código VDFOO. A Base de Conhecimento (KB) é consultada durante a transformação para obter detalhes sobre a organização do código. As classes são reimplementadas nas camadas de Interface, Regras de Negócio e Acesso ao Banco de Dados. As regras sobre a hierarquia de chamadas das Unidades de Programa (UP), sobre as conexões de mensagens e eventos, são utilizadas na identificação e extensão das classes do DAF. A classe principal é criada e trata os eventos que disparam cada caso de uso do sistema.

Finalmente, no passo "**Executar Código VDFOO**", registra-se o código gerado nas ferramentas de desenvolvimento do VDFOO. Pré-compila-se o novo código, gerando o código para ser executado através do *runtime* VDFOO. Usando os mesmos dados de entrada usados pelo sistema legado, verifica-se o resultado obtido comparando os resultados de cada caso de uso do sistema gerado. Nesta etapa o Engenheiro de Software pode realizar adequações na interface para o ambiente gráfico ou acrescentar novas funcionalidades ao sistema gerado.

Para automação dos dois primeiros passos da estratégia, foram construídos dois transformadores no Draco-PUC, o Transformador de DFP para DFP Organizado e o Transformador de DFP Organizado para VDFOO. Para a construção dos transformadores foi necessária a construção do *parser* e *PrettyPrinter* DFP e VDFOO. O Draco Domain Editor (DDE) foi utilizado para edição e validação das gramáticas e transformadores.

4.1 Construção do *parser* e *PrettyPrinter* DFP

Para que o Draco-PUC reconheça os códigos legados escritos em DFP, e seja possível o mapeamento de suas regras de produção através dos transformadores, é necessário que esta linguagem seja definida através de seu *parser* e *prettyprinter*. Como a linguagem DFP é uma linguagem comercial e não é fornecida sua gramática, foi necessário construí-la com base na análise de códigos legados escritos em DFP, manuais e livros sobre DFP. A Figura 17 mostra os passos para construção do *parser* e *prettyprinter* DFP: Editar Gramática DFP, Gerar *Parser* e *PrettyPrinter* DFP, e Verificar *Parser* e *PrettyPrinter* DFP.

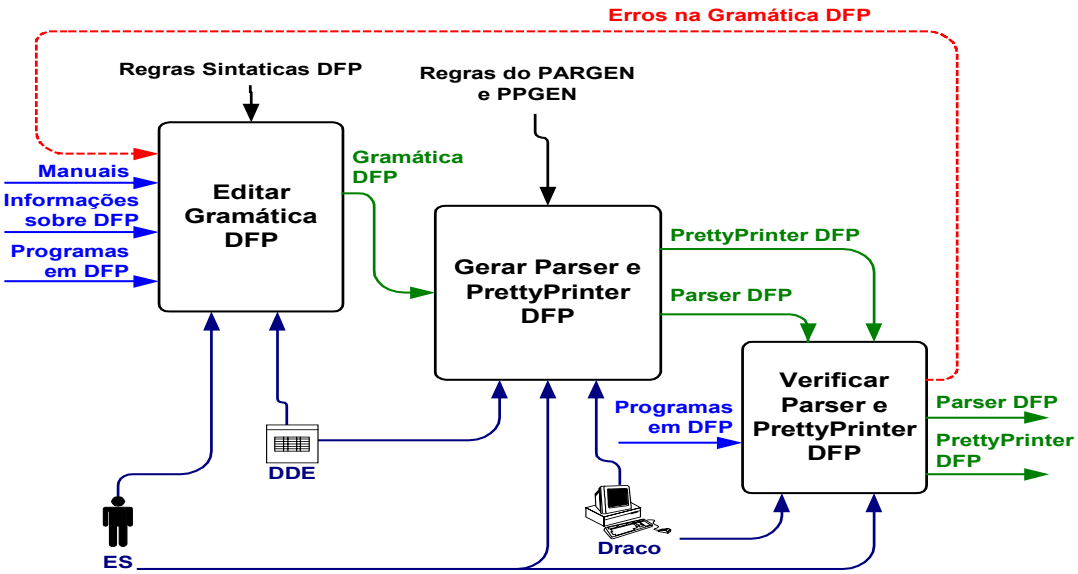


Figura 17. Passos para construção do *Parser* e *PrettyPrinter* DFP

Segue uma apresentação de cada passo para a construção do *parser* e *prettyprinter* DFP.

No passo **Editar Gramática DFP**, partiu-se dos manuais da linguagem, informações coletadas em livros e outras publicações especializadas em DFP e de códigos em DFP, para obter uma gramática que reconheça programas válidos para o pré-compilador DFP. As regras sintáticas descritas em manuais e documentos direcionaram a construção das regras de produção da gramática DFP. O DDE foi utilizado para escrever as regras de produção da gramática agrupadas de forma a facilitar seu uso nos transformadores.

A Figura 18 mostra regras de produção da gramática DFP. À esquerda da figura tem-se a descrição das regras de produção e à direita a árvore gramatical. A raiz da árvore gramatical é definida pela regra de produção "program", da qual deriva a regra de produção "statements". As regras de produção derivadas de "statements" são: "screen", que reconhece as telas de interação; "com_state", que reconhece a maior parte das estruturas sintáticas do DFP; "procedure", que reconhece os procedimentos; e "function", que reconhece as funções. Todas são representadas como nós intermediários na árvore gramatical.

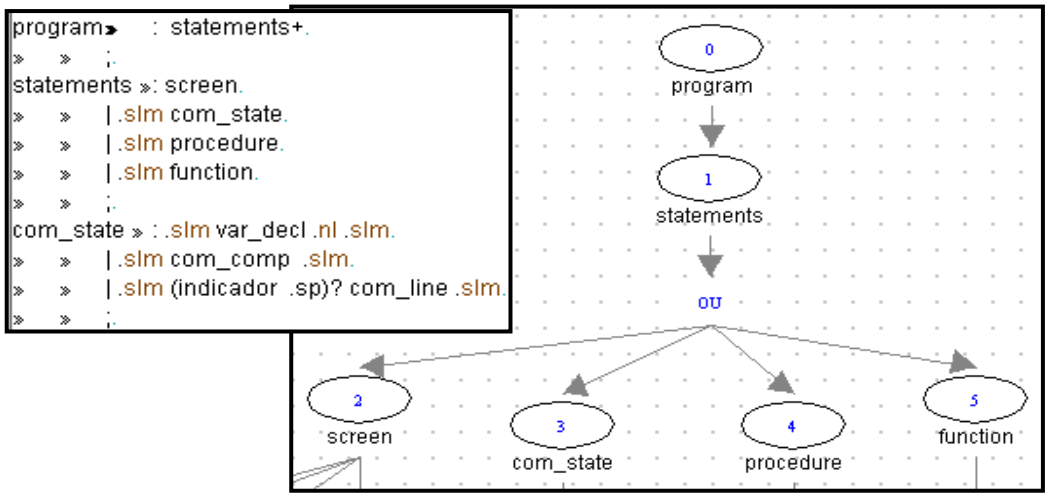


Figura 18. Regras de produção iniciais da Gramática DFP no DDE

A regra de produção "screen" reconhece as telas de interação com o usuário. Para o DFP uma tela de interação é composta de informações textuais e campos de interação com o usuário, que são referenciados nos manuais DFP como janelas. Assim, uma tela de interação é um conjunto de janelas (campos de interação) e texto. A informação textual de uma tela pode conter diversas linhas com espaços e outros caracteres que são ignorados pelo parser DFP nas demais regras de produção. Para que o *parser* DFP tratasse adequadamente a estrutura das telas de interação, foram definidas ações semânticas executadas na redução da regra de produção "screen".

A Figura 19 mostra detalhes sobre a derivação da regra de produção "screen", com as ações semânticas "init_screen()" e "end_screen()". Estas ações semânticas tratam caracteres especiais na redução da regra de produção "screen". À esquerda da figura têm-se as ações semânticas acionadas pela regra "screen" e pela expressão regular "IGNORE". A expressão regular "IGNORE" é utilizada no tratamento de caracteres que são ignorados pelo parser, como espaços em branco e caracteres de tabulação. Estas ações semânticas permitem que estes caracteres não sejam ignorados ao tratar a regra de produção "screen".

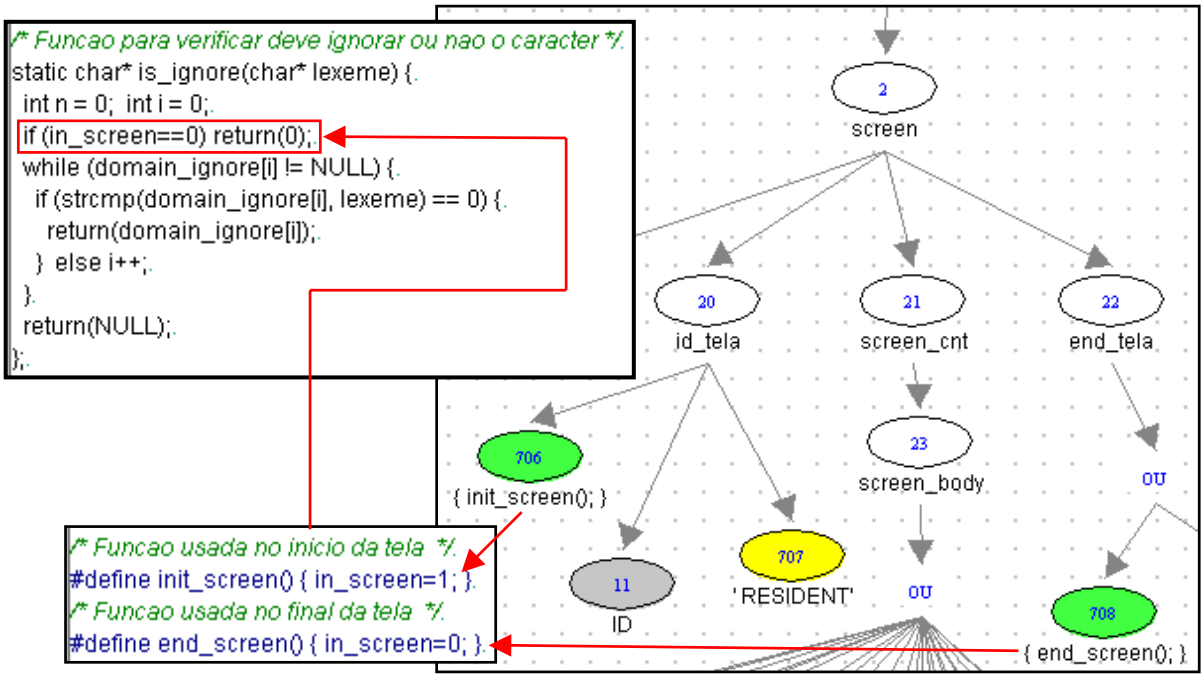


Figura 19. Ações semânticas da regra de produção "screen" na Gramática DFP

A Figura 20 mostra as derivações da regra de produção "com_state", que reconhece a maior parte das estruturas tratadas pelo *parser* DFP. As regras de produção derivadas de "com_state" são: "indicador", que reconhece variáveis booleanas (referenciados nos manuais DFP como indicadores); "var_decl", que reconhece a declaração de variáveis; "com_line", que reconhece os comandos gerais da linguagem, escritos em linhas de comando; e "com_comp", que reconhece os comandos do pré-compilador DFP,.

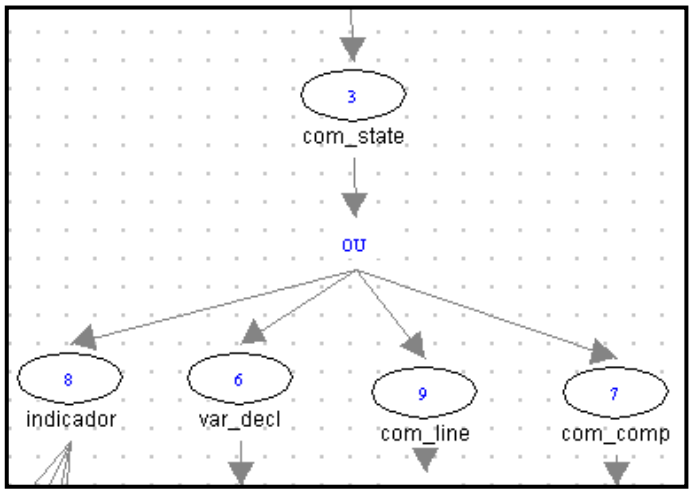


Figura 20. Derivações da regra de produção "com_state" na Gramática DFP

As regras de produção derivadas de "com_state" derivam outras regras de produção que por sua vez derivam outras regras de produção, agrupadas conforme as características do DFP.

Também foram definidas ações semânticas para a regra "new_command", derivada de "com_comp". Estas ações semânticas tratam a criação de novos macro-comandos no código DFP. A Figura 21 mostra as ações semânticas disparadas pela regra "new_command", e a árvore gramatical com as derivações desta regra. Estas ações semânticas reconhecem a criação de um novo macro-comando no código DFP, inserindo-o na tabela de *tokens* válidos da gramática.

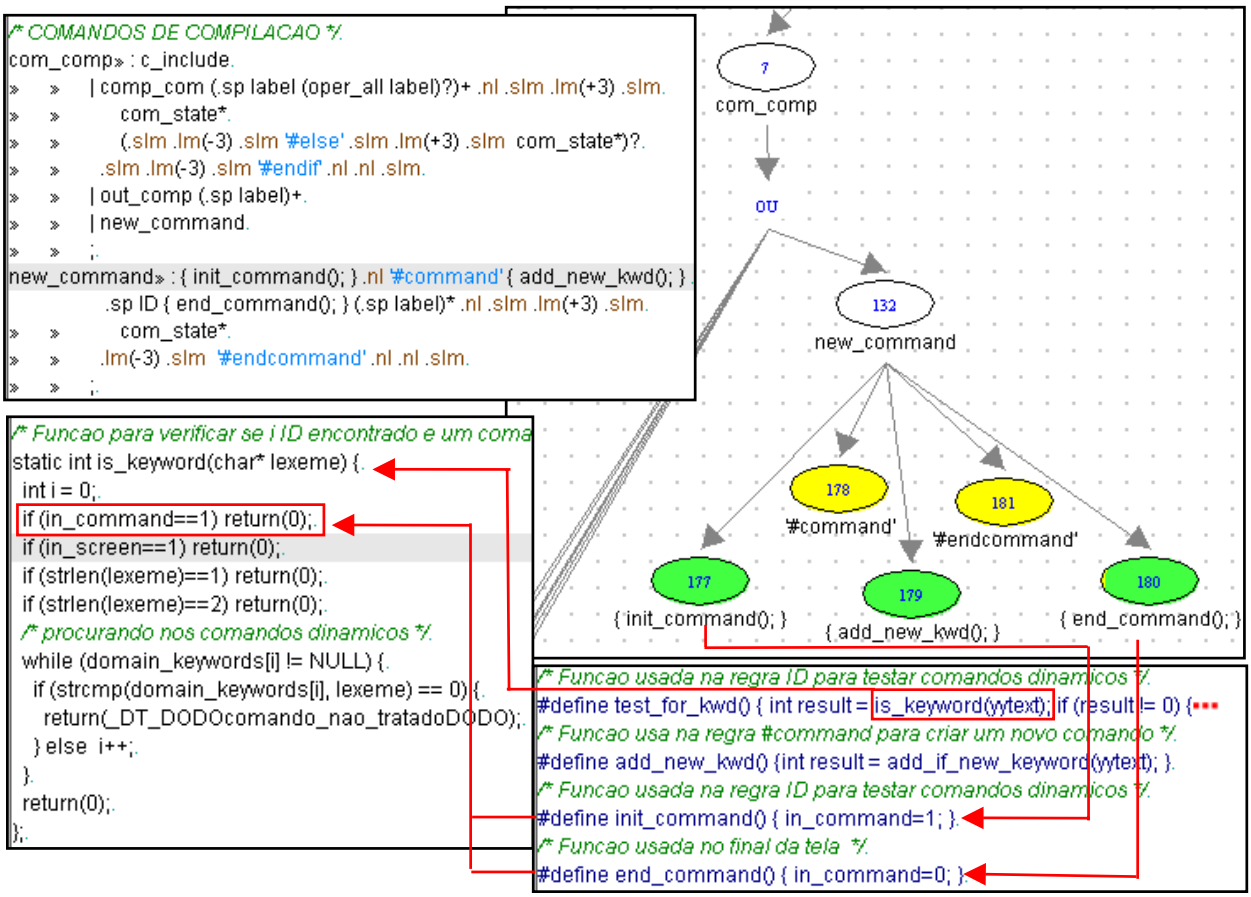


Figura 21. Ações semânticas da regra de produção "new_command"

Os novos *tokens* inseridos através destas ações semânticas são tratados por uma outra ação semântica definida na expressão regular "ID". A expressão "ID" reconhece palavras no código DFP que não foram reconhecidas como *tokens*. O acionamento da ação semântica nesta expressão permite ao *parser* tratar os novos comandos inseridos. A gramática completa encontra-se no ANEXO I.

No passo **Gerar *Parser* e *PrettyPrinter* DFP**, o Engenheiro de Software, no DDE, submete a gramática DFP ao Draco-PUC que emprega o PARGEN para gerar o *parser* e o PPGEN para gerar o *PrettyPrinter* DFP. O *parser* DFP baseia-se nas regras de produção

da gramática LALR com tratamento de *back-track* [Fre96]. O *PrettyPrinter* baseia-se nos símbolos de formatação colocados ao lado das regras de produção para reescrever o código orientado pela sintaxe da linguagem DFP.

No último passo, **Verificar *Parser* e *PrettyPrinter* DFP**, programas previamente analisados pelo pré-compilador DFP, de diversas fontes e funcionalidades, são usados para validar o *parser* e *prettyprinter*. A Figura 22 mostra um código em DFP (1), que é submetido ao parser, gerando a DAST no Draco-PUC (2). Através do *PrettyPrinter* é novamente obtido o código textual em DFP (3).

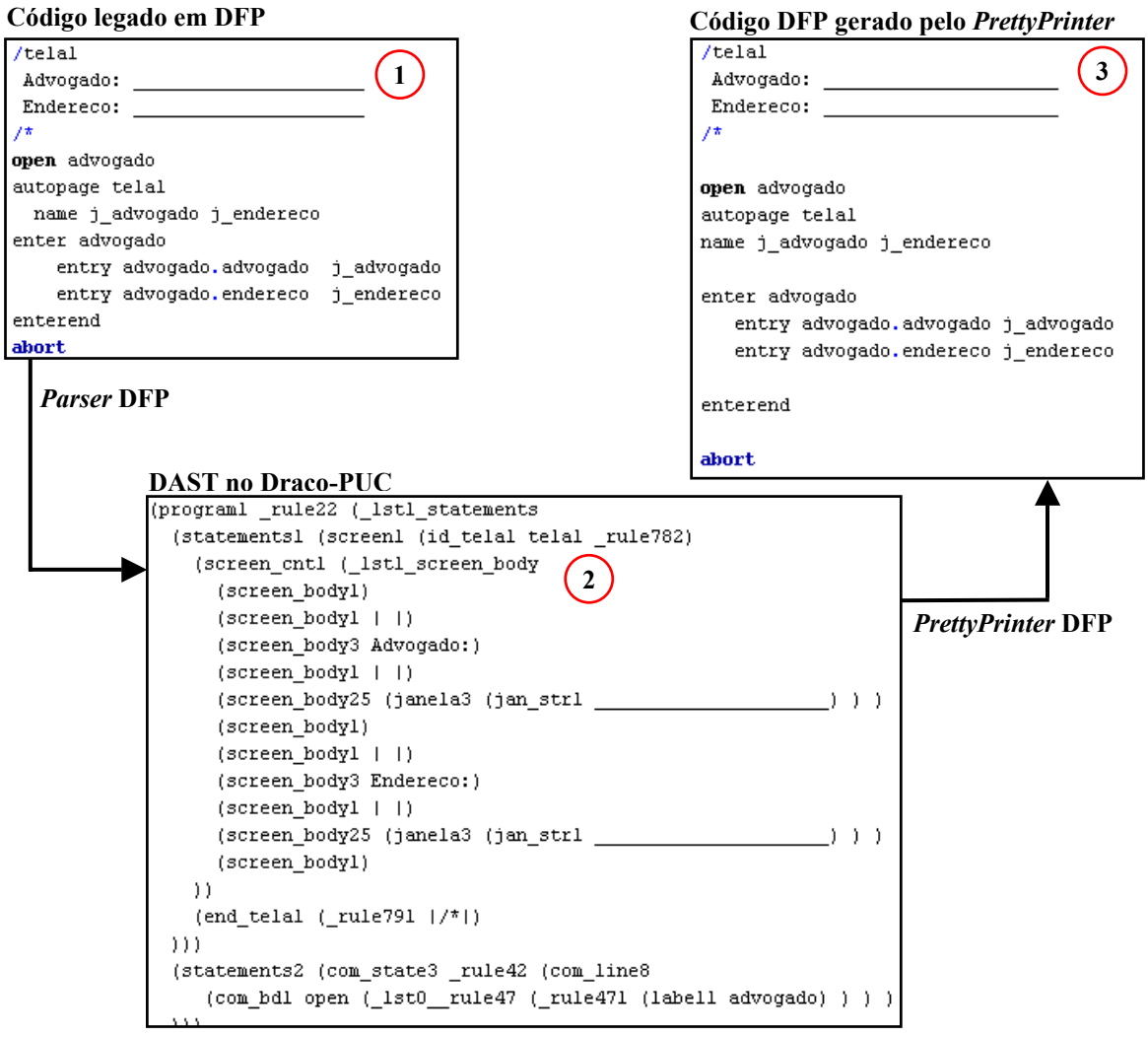


Figura 22. Verificação do *parser* e *PrettyPrinter* DFP

Para a validação do *parser*, é analisada a DAST gerada no Draco-PUC para o código DFP, identificando as regras utilizadas para a redução. A validação do PrettyPrinter é feita com base no código gerado novamente em formato textual, que comparado ao código legado permite identificar falhas na formatação do código.

Na identificação de falhas, seja na execução do *parser*, do *PrettyPrinter* ou no reconhecimento das regras de produção adequadas, retorna-se a edição da gramática para realizar as correções e repetir novamente os passos.

Para validar o *parser* e *prettyprinter* DFP foram utilizados os códigos de 5 sistemas diferentes, em um total de 2870 programas, que totalizam aproximadamente 6 milhões de linhas de código. Estes sistemas também foram utilizados como estudos de caso para validar a transformação. Destes programas, 20 foram analisados detalhadamente, através do seu mapeamento com as regras de produção da gramática DFP. Os demais foram comparados com o auxilio de um comparador de arquivos "FC - File Compare - Norton". Este utilitário comparou os códigos legados com os obtidos após a passagem do *parser* e do *PrettyPrinter*. Nesta comparação foram identificadas apenas diferenças de formatação do código, devido às regras de formatação padronizadas para o *PrettyPrinter* DFP.

Apesar do grande volume de programas reconhecidos pela gramática DFP, podem ocorrer situações não tratadas pelas suas regras de produção, requerendo alterações e conseqüentemente gerando novos *parser* e *PrettyPrinter*.

4.2 Construção do *parser* e *PrettyPrinter* VDFOO

Para que o Draco-PUC possa gerar os códigos em VDFOO através de seus transformadores foi necessário construir este domínio com seu *parser* e *prettyprinter*. Como a linguagem VDFOO também é comercial e sua gramática não estava disponível, foi necessário construí-la da mesma forma que para o DFP. Sua construção baseou-se nos manuais e livros sobre a linguagem, além das informações sobre o DataFlex Application *Framework* (DAF) e sua integração com o VDFOO. A Figura 23 mostra os passos para construção do *parser* e *prettyprinter* VDFOO.

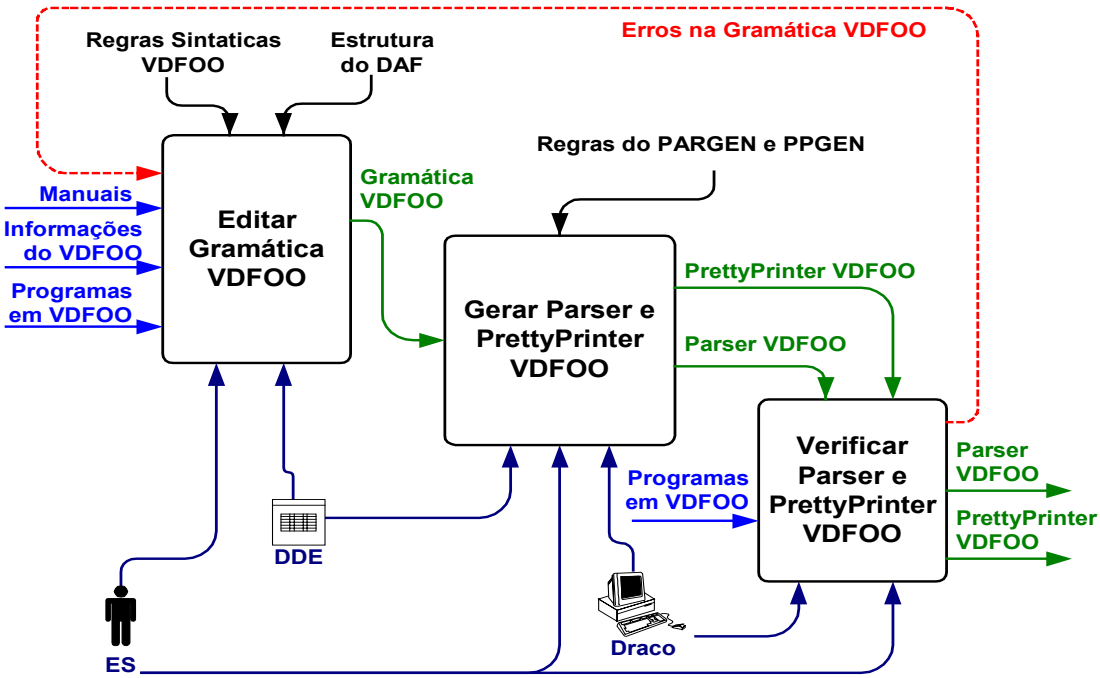


Figura 23. Passos para construção do *Parser* e *PrettyPrinter* VDFOO

No passo **Editar Gramática VDFOO**, partiu-se dos manuais da linguagem, informações coletadas em livros e outras publicações especializadas em VDFOO e de programas em VDFOO, que foram previamente analisados pelo pré-compilador VDFOO. As regras sintáticas descritas em manuais e documentos, e as informações sobre a estrutura do DAF direcionaram a construção das regras de produção da gramática VDFOO. O DDE foi utilizado para escrever as regras de produção da gramática agrupadas de forma a facilitar seu uso nos transformadores.

A Figura 24 mostra as regras de produção iniciais da gramática VDFOO. À esquerda da figura tem-se a descrição das regras de produção e à direita a árvore gramatical. A raiz da árvore gramatical é definida pela regra de produção “program”, da qual deriva a regra de produção “statements”. As regras de produção derivadas de “statements” são: “oo_class”, que reconhece a estrutura sintática de classes; e “oo_body”, que reconhece as demais estruturas sintáticas do VDFOO, incluindo objetos, procedures, functions e linhas de comandos. Conforme se pode ver na gramática a regra “oo_class” também deriva a regra “oo_body”. Isto permite que dentro de uma classe em VDFOO tenha objetos, *procedures*, *functions* e outras estruturas gramaticais.

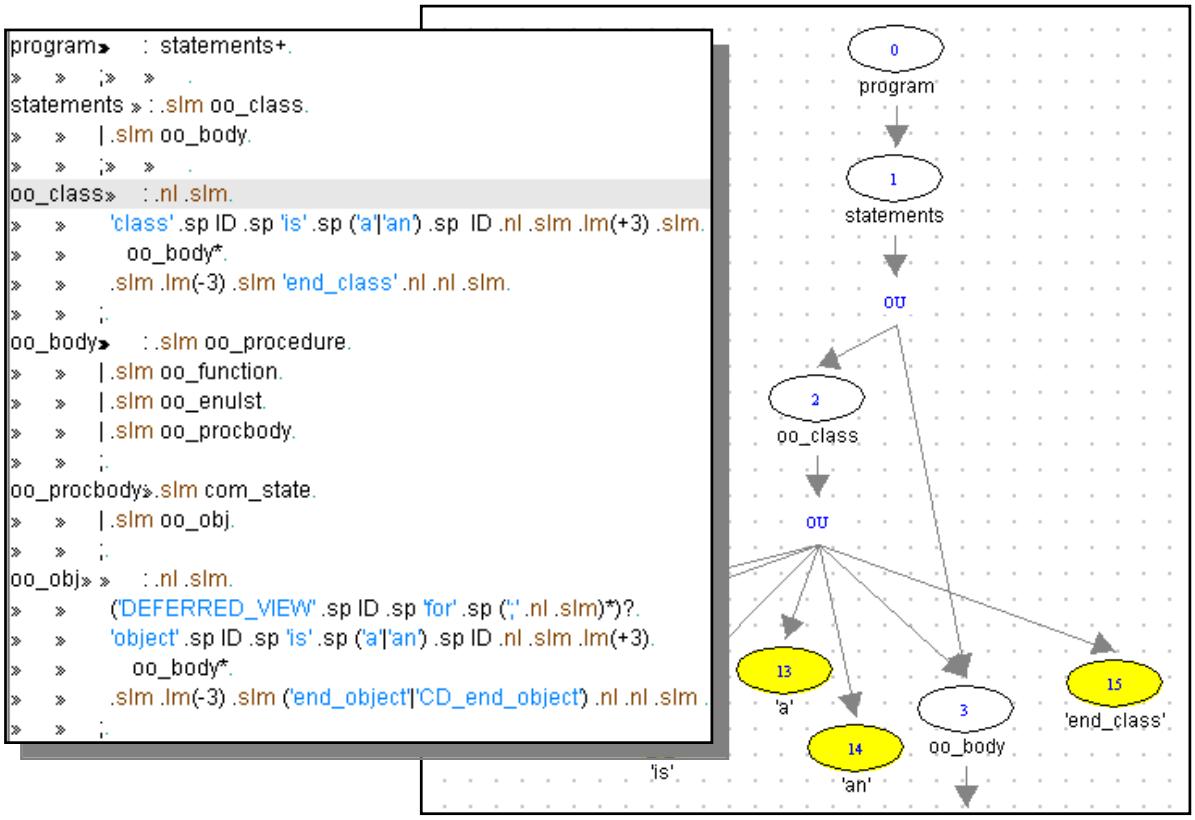


Figura 24. Regras de produção iniciais da Gramática VDFOO

A regra de produção "oo_body" deriva regras para tratamento de procedimentos "oo_procedure", funções "oo_function", listas enumeradas "oo_enulst" e corpo de estruturas orientadas a objetos "oo_proc_body". A regra "oo_procbody" deriva regras para o tratamento de comandos da linguagem VDFOO "com_state" e para tratamento de objetos "oo_obj".

Da mesma forma procedeu-se com as demais regras da gramática VDFOO. A gramática completa encontra-se no ANEXO II. De uma forma geral a linguagem suporta tratamento de entrada e saída, processamento de argumentos, como cálculos e atribuições, acesso ao banco de dados, controle do fluxo de execução, tratamento de formulários de entrada de dados, macro-comandos, teclas de função, cálculos matemáticos, construção de relatórios, controle multi-usuário, tratamento de *strings*, entrada e saída seqüencial, estruturas de bloco, interação com o sistema operacional, e tratamento de eventos. Também são definidas ações semânticas em algumas regras de produção, para suportar a criação de novos macro-comandos, de forma semelhante à descrita para DFP.

No passo **Gerar *Parser* e *PrettyPrinter* VDFOO**, o Engenheiro de Software, no DDE, submete a gramática VDFOO ao Draco-PUC que utiliza o PARGEN para gerar o *parser* VDFOO e o PPGEN para gerar o *PrettyPrinter.*

No último passo, **Verificar *Parser* e *PrettyPrinter* VDFOO,** utiliza-se programas previamente analisados pelo pré-compilador do VDFOO para verificar o *parser* e *prettyprinter* construídos. Da mesma forma que o DFP é utilizado o comando "*display*" do Draco-PUC para identificar os reconhecimentos de cada regra pelo *parser* VDFOO. O prettyprinter gera o código novamente em formato textual conforme suas regras de formatação. Na identificação de falhas, seja na execução do *parser* ou do *PrettyPrinter* retorna-se a edição da gramática realizando as correções e repetir os passos da geração e verificação.

Para validar os *parser* e *prettyprinter* VDFOO foram utilizados os códigos de 3 sistemas diferentes. Os sistemas possuem em média 25 classes de tratamento de regras de negócio, 7 classes de interação de entrada de dados, com 210 componentes e 7 classes de relatório com 350 componentes, perfazendo cerca de 950 mil linhas de código.

A gramática VDFOO criada atendeu plenamente as necessidades da estratégia, contudo poderão ocorrer casos não tratados na reengenharia que requeiram modificações na gramática, gerando novos *parser* e *prettyprinter*.

Finalizada a construção dos *parsers* e *PrettyPrinters*, foi feita a implementação dos transformadores descritos a seguir.

4.3 Construção do Transformador de DFP para DFP Organizado

O Transformador de DFP para DFP Organizado (DFPO) foi construído para automatizar o passo "Organizar Código Legado". A construção do transformador foi feita em dois passos, como mostra a Figura 25.

No passo **Criar Transformador de DFP para DFPO**, parte-se de informações sobre a linguagem DFP, principalmente sua gramática, e das descrições das transformações de DFP para DFPO. Este passo é orientado pelas técnicas de organização do

código legado, pelos princípios da orientação a objetos, pelas regras de escrita das transformações do TFMGEN do Draco-PUC e pela estrutura do DAF.

O DDE auxilia o Engenheiro de Software (ES) na edição do transformador. O Draco-PUC é usado para gerar o transformador a partir das descrições das transformações.

No passo **Verificar Transformador de DFP para DFPO**, aplica-se o transformador construído sobre os programas em DFP, obtendo os programas em DFP Organizado. A estrutura do programa gerado é analisada, buscando identificar falhas na sua organização. Ao localizar uma falha, deve ser identificada a origem desta falha, que pode ser causada pelas definições erradas das transformações ou da gramática DFP.

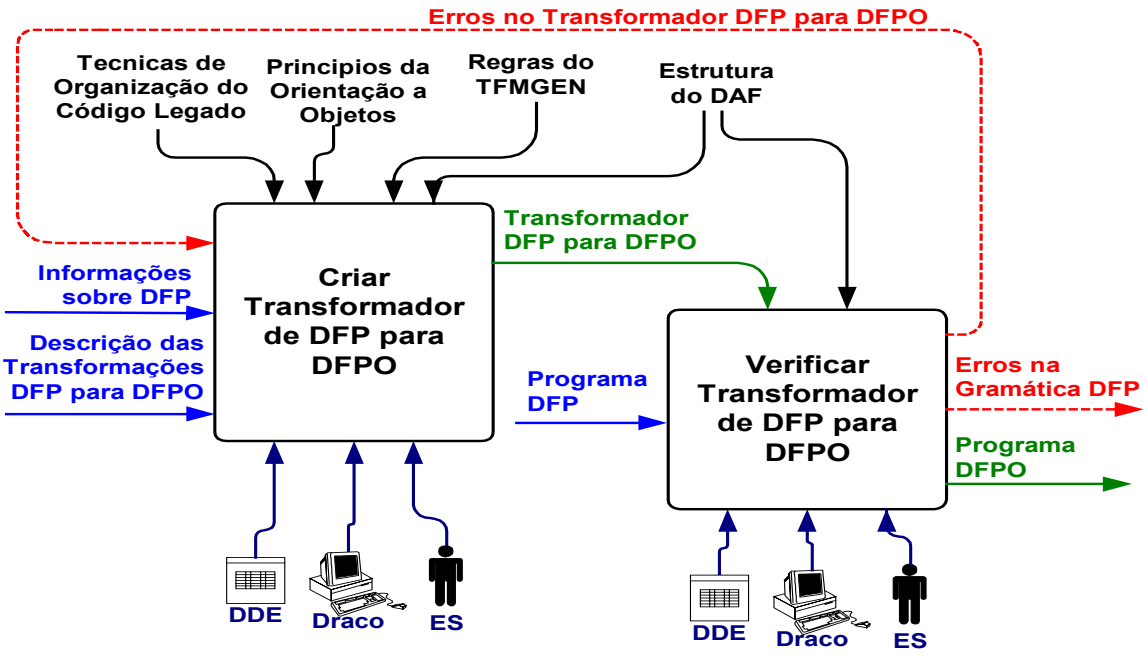


Figura 25. Construção do Transformador de DFP para DFP Organizado

As transformações para organização do código legado, mapeiam a sintaxe e a semântica dos comandos em DFP, para comandos na mesma linguagem DFP, porém organizados segundo os princípios da orientação a objetos. O objetivo é preparar o código do sistema legado de maneira a facilitar a transformação para VDFOO.

A Figura 26 mostra exemplos das transformações para organização do código legado que foram implementadas no transformador de DFP para DFP Organizado. À esquerda tem-se o código legado em DFP e à direita o correspondente em DFP Organizado. Conforme

indicado em (**1**), os comandos da tela dão origem a supostas classes de interface. Comandos de abertura de arquivos (**2**) dão origem a supostas classes de banco de dados. Comandos que referenciam uma suposta classe são nela incluídos, como por exemplo, os comandos de posicionamento da tela (**3**). Rotinas e macro-comandos (**4**), que referenciam supostos atributos são incluídos como supostos métodos da suposta classe. Comandos de inclusão de código (**5**) são utilizados para manter a consistência do código organizado. São criados desvios incondicionais no código da suposta classe (**6**), para garantir a mesma seqüência de execução do código legado. Como resultados deste passo têm-se o código DFP Organizado e uma Base de Conhecimento no Draco-PUC (KB) com fatos e regras sobre o código organizado que serão usados no próximo passo da transformação.

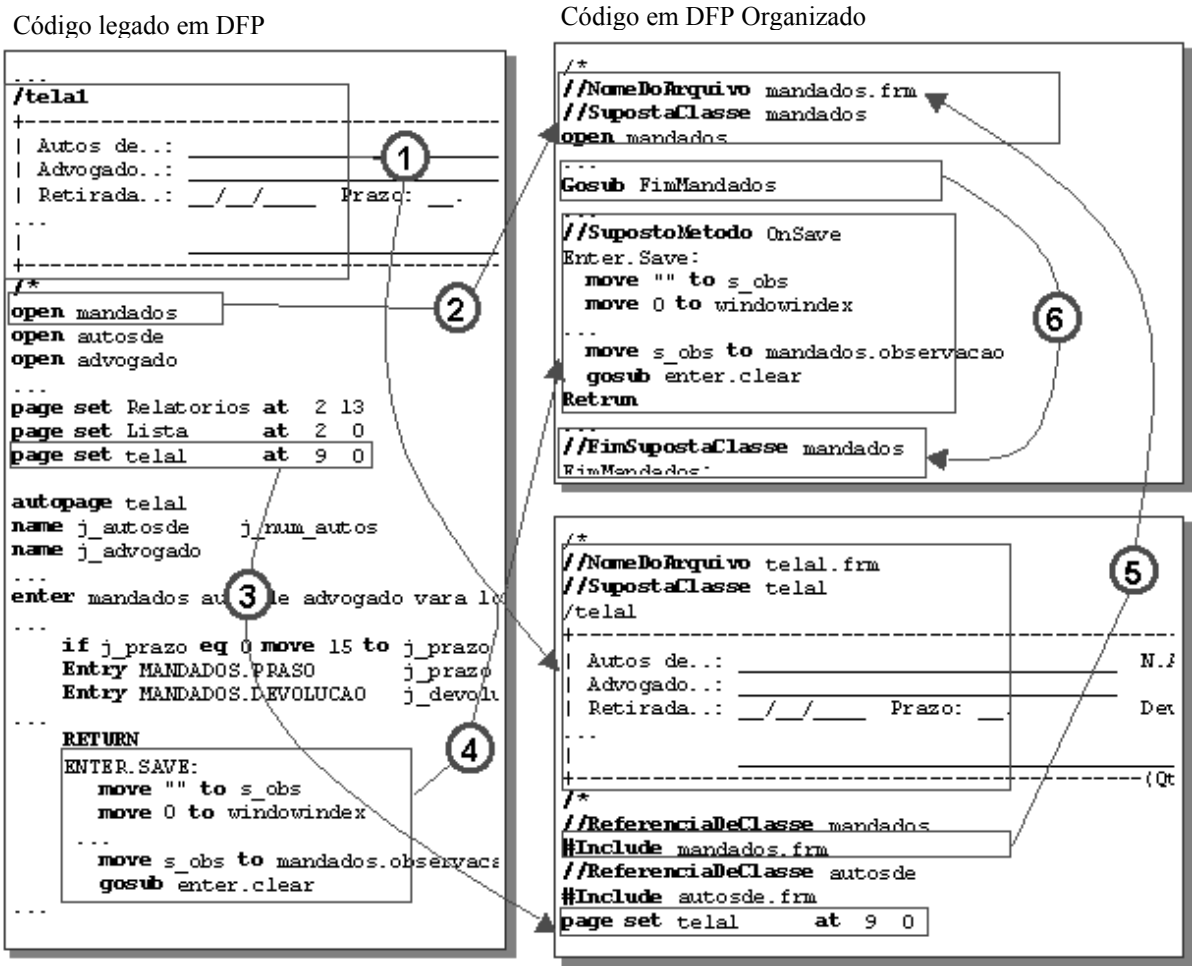


Figura 26. Exemplos da organização do código legado

Foram adicionados meta-símbolos, no código organizado, na forma de comentários, para que o próximo transformador reconheça as supostas classes e seus membros. Os meta-símbolos criados são apresentados na Figura 27.

Meta-símbolos	Descrição
//NomeDoArquivo <nome_do_arquivo>	Define o nome do arquivo que contem a suposta classe.
//SupostaClasse <nome_da_classe>	Indica o inicio da definição da suposta classe.
//FimSupostaClasse <nome_da_classe>	Indica o fim da definição da suposta classe.
//SupostoMetodo <nome_do_método>	Indica o inicio de um suposto método.
//SupostoParametro <nome_da_variáve>	Define as variáveis que serão alocadas como parâmetros do suposto método.
//SupostoRetornoDeMetodo <tipo_de_retorno>	Define tipo de retorno
//ControleDeFluxo <nome_do_método>	Identifica desvios de fluxo para acionamento de supostos métodos.
//ReferenciaDeClasse <nome_da_classe>	Indica a referência a uma suposta classe.
//ReferenciaDeMetodo <nome_do_método>	Indica a referência a um suposto método

Figura 27. Meta-símbolos adicionados na geração do código DFO Organizado

Para facilitar o entendimento deste transformador, a Figura 28, apresenta uma visão global dos principais conjuntos de transformação (SOT – *Set Of Transforms*) que compõem o Transformador de DFP para DFP Organizado. O transformador está agrupado em 14 conjuntos principais de transformação. Cada um dos SOT´s é composto de um ou mais transformadores, que podem acionar a execução de SOT´s secundários.

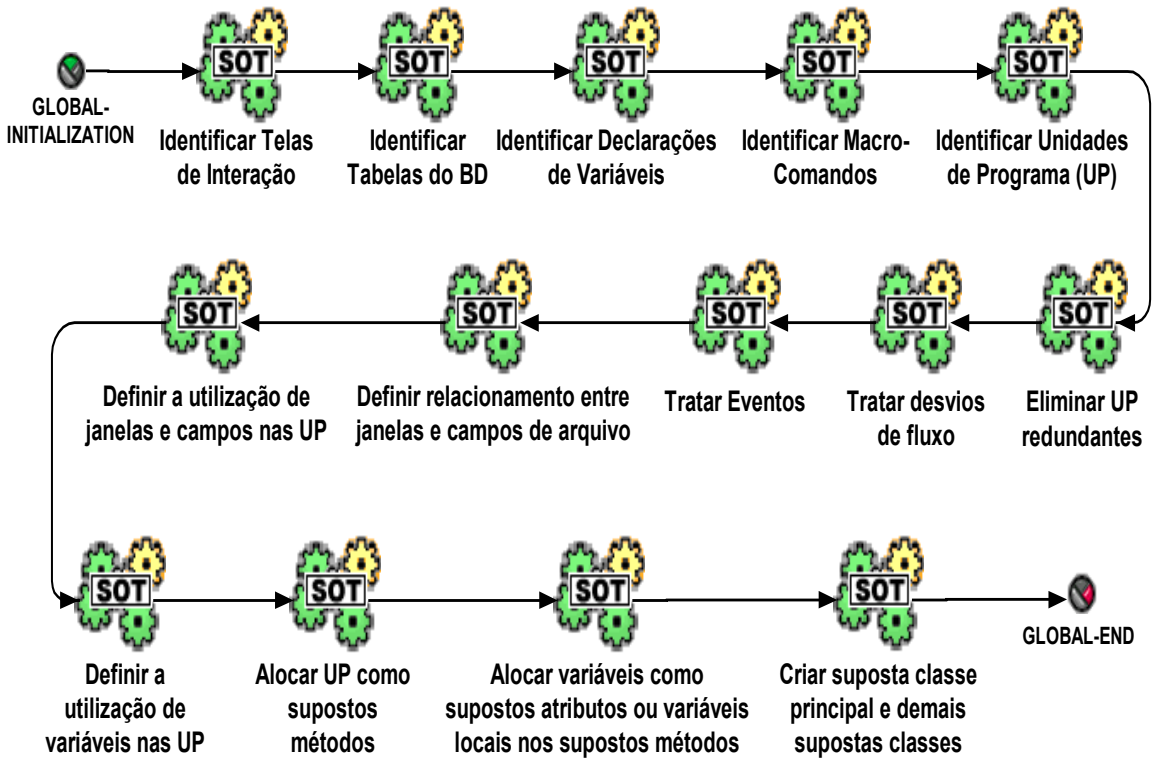


Figura 28. Principais SOT´s do Transformador de DFP para DFP Organizado

O transformador de DFP para DFP Organizado identifica comandos relacionados com: interface, declaração de variáveis, declaração de sub-rotinas, acesso à banco de dados, pontos de desvio do fluxo de execução, comandos ou atribuições seqüenciais, relacionamento entre as variáveis e os outros elementos do código legado, definição de macros e configuração das funcionalidades para interação do usuário.

Durante a execução do transformador, cada comando identificado tem seu conteúdo armazenado em *workspaces* [Fuk99]. Na Base de Conhecimentos do Draco (KB) são armazenados fatos e regras sobre estes comandos, contendo a referência do seu *workspace* e suas características. Todas as regras tratadas na KB, como as *workspaces*, utilizam o nome do programa como identificador para possibilitar o tratamento dos vários programas de um sistema.

Segue uma apresentação detalhada de cada SOT da Figura 28.

4.3.1 Identificar Telas de Interação

A identificação das telas de interação é importante para a organização do código legado, pois permite o particionamento do código, segundo os casos de uso do sistema. Cada tela identificada é considerada como uma suposta classe de interface do sistema. A identificação das telas facilita a sua reconstrução na linguagem alvo VDFOO.

A identificação das telas no DFP, é feita pela base à regra de produção "screen", e suas derivações. Cada tela identificada, é armazenada como um fato na KB. O conteúdo da tela é consultado pelos transformadores que reconhecem suas janelas de interação.

Ao identificar uma tela, é chamado um SOT auxiliar que percorre o código, com transformações que buscam comandos de tratamento das telas, para verificar se estas são apresentadas do usuário, tratadas apenas como áreas de trabalho ou fazem parte de código não referenciado, como os comandos "page", "accept" e "display".

No caso das telas apresentadas para o usuário, tem-se um SOT auxiliar que reconhece sua estrutura armazenando informações na KB. A estrutura da tela é usada para sua reconstrução no ambiente visual. A Figura 29 mostra o trecho do código da transformação que

identifica a estrutura da tela com suas janelas de interação e informações textuais, armazenando na KB a estrutura e formato da tela.

```
TRANSFORM NovaTela.
LHS: {{dast dataflex.statements.
»    [[screen c_tela]].
}}.
POST-MATCH: {{dast txt.decls.
»    contatrib = 0;.
»    char linha[1000], *coluna = new char, *contjanela = new
»    int contjan = 0;.
»    SET_LEAF_VALUE("prog_nome",prog_nome);.
»    sprintf(classe,"%s",expand("[[c_tela]]"));.
»    sprintf(blocotela,"./WorkTemp/%s_%s",prog_nome,clas
»    printf("\n Escrevendo %s no bloco: %s \n",classe,blocote
»    SET_LEAF_VALUE("classe",classe);.
»    CREATE_WORKSPACE(blocotela,"dataflex");.
»    TEMPLATE("T_Tela").
..   TRANSPORT_VALUE("c_tela");
»    »    SET_LEAF_VALUE("t_janela",linhacopia);.
»    »    SET_LEAF_VALUE("t_linha",linha);.
»    »    SET_LEAF_VALUE("t_coluna",coluna);.
»    »    SET_LEAF_VALUE("prog_nome",prog_nome);.
»    »    if (atoi(linha) > 0) {.
»    »     KBAssert("SupostoAtributoInterface([[prog_nome]]
»    »    }.
»      }.
»      memset(linhatela,0x00,1000);.
»      memset(linhatelaaux,0x00,1000);.
»     }.
»     if (ejanela) {.
»      for (int a = i; a < tamanhotela; a++) {.
»       if (!(corpotela[a] == '_' || corpotela[a] == ',' || corpotela[a]
»       corpotela[a] == '@' || corpotela[a] == 'e' || corpotela[a] ==
»       else {.
»        char *letra = " ";.
»        sprintf(letra,"%c",corpotela[a]);.
»        strcat(linhatela,letra);.
»        colunatela++;.
»        i++;.
»       }.
»      }.
»      linhatela[i+1] = '\0';.
»      sprintf(linha,"%d",contatrib);.
»      sprintf(coluna,"%d",colunatela-strlen(linhatela));.
»      sprintf(linhatelaaux,"%c%s%c",34,linhatela,34);.
»      SET_LEAF_VALUE("t_janela","Janela");.
»      SET_LEAF_VALUE("t_linha",linha);.
»      SET_LEAF_VALUE("t_coluna",coluna);.
»      SET_LEAF_VALUE("prog_nome",prog_nome);.
..     sprintf(contjanela,"%d",++contjan);
»      memset(linhatela,0x00,1000);.
»      memset(linhatelaaux,0x00,1000);.
»      contatrib++;.
»      colunatela = 0;.
»     } else {.
»      char *letra = " ";.
»      sprintf(letra,"%c",corpotela[i]);.
»      strcat(linhatela,letra);.
»      colunatela++;.
»     }.
»    }.
»    //delete(corpotela);.
»    SKIP_APPLY();.
}}.
```

Figura 29. Trechos do código da transformação que trata a estrutura das telas

Como resultado deste SOT têm-se os fatos sobre as telas como mostra a Figura 30. A regra "screen" do DFP, que reconhece uma tela, dispara uma ação que cria o fato "SupostaClasseInterface" (1), que guarda o nome do programa, o nome da tela e um indicador de utilização da tela no programa. As janelas de interação (2) são armazenadas na KB através do fato "SupostoAtributo", que contém os nomes do programa e da tela, a seqüência da janela, o nome da janela, o tamanho da janela, e informações sobre relacionamentos com banco de dados. Informações sobre a utilização da janela, identificadas pelo SOT "**Definir relacionamento entre janelas e campos de arquivo**" descrito a seguir, como por exemplo se a janela é utilizada para entrada de dados ou apenas para a apresentação de dados. São armazenadas fatos do tipo "SupostoParametro" (3), para identificar características de validação para a janela de entrada de dados, como o "capslock", que faz com que as letras digitadas na janela de interação, sejam convertidas para maiúsculas.

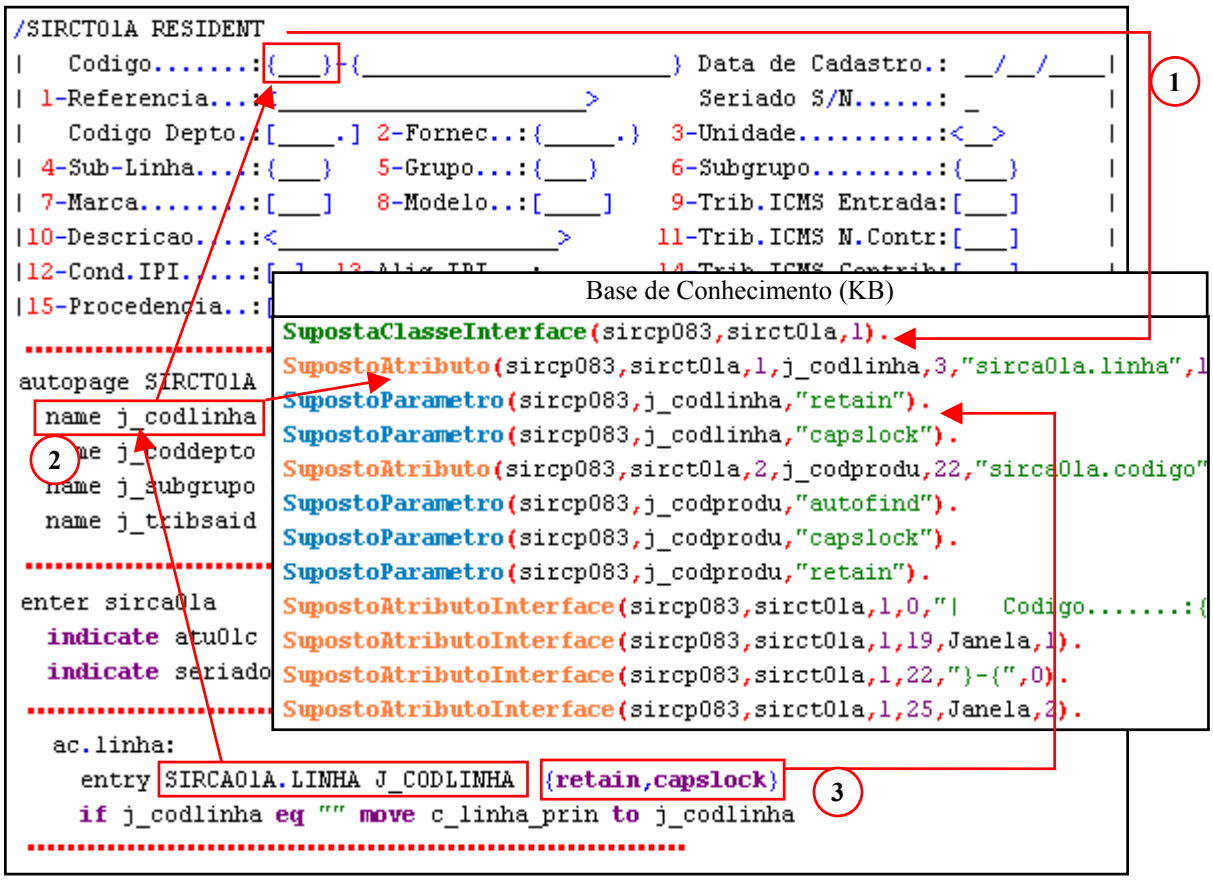


Figura 30. Regras na KB sobre as telas de interação

As supostas classes de interface identificadas e armazenadas neste SOT são atualizadas pelos demais SOTs, quando são identificadas regras de produção no código legado que afetam o comportamento dos supostos atributos de interface.

4.3.2 Identificar Tabelas do BD

Normalmente códigos legados, principalmente relacionados a sistemas comerciais, utilizam banco de dados para o armazenamento das informações. Desta forma, é necessário identificar as tabelas utilizadas em cada código, para criação de supostas classes de acesso ao banco de dados com seus supostos atributos. Cada tabela do banco de dados é identificada na KB pelo fato "SupostaClasseBD".

Ao identificar a utilização de uma tabela do banco de dados, um SOT é chamado para coletar informações sobre o tipo de utilização da tabela "R-leitura", "W-somente gravação" e "RW-Leitura e Gravação". Este SOT identifica também os campos da tabela como supostos atributos.

A Figura 31 mostra um trecho do código do transformador que identifica os campos da tabela do banco de dados e cria fatos "SupostoAtributoDB" na KB.

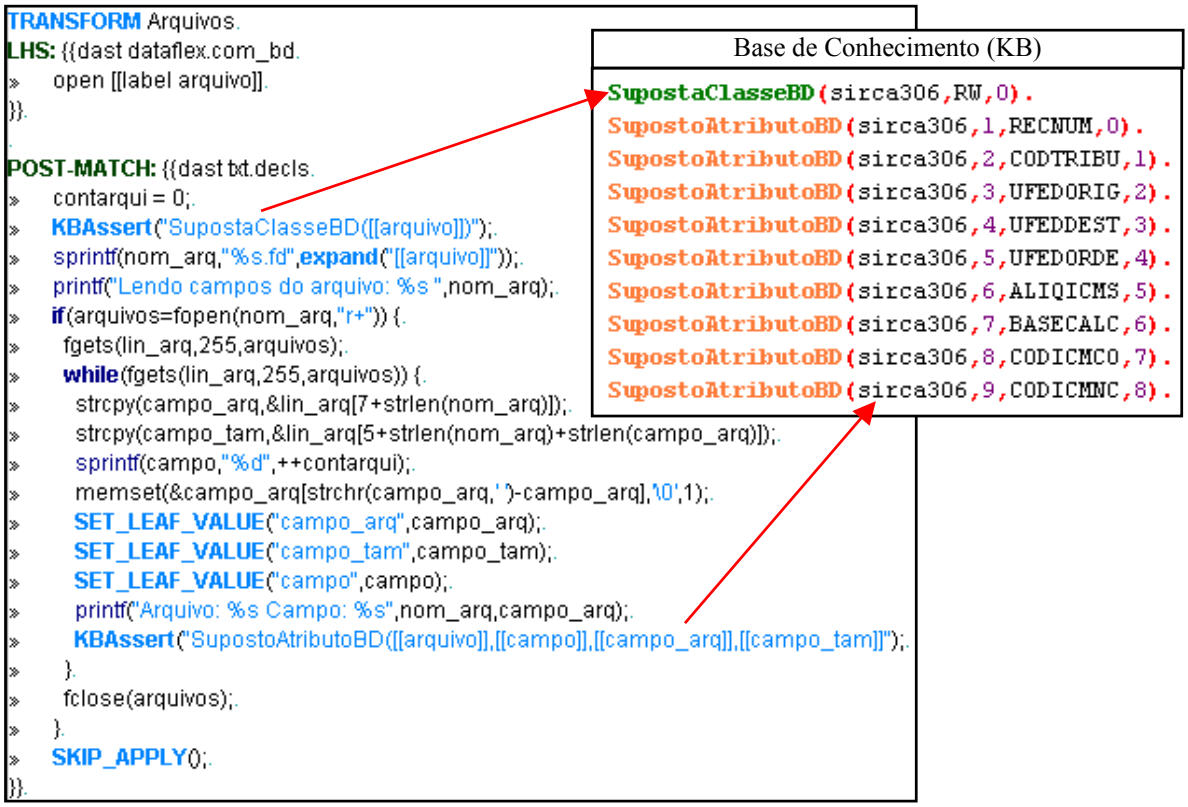


Figura 31. Código do transformador que identifica os supostos atributos do BD

Os fatos na KB sobre as supostas classes de banco de dados e seus supostos atributos, são atualizados pelos demais SOT´s, em relação ao tipo de utilização "R, W ou RW". Também são atualizadas informações sobre o relacionamento com outras supostas classes.

4.3.3 Identificar Declarações de Variáveis

As variáveis são reconhecidas pela sua declaração no código legado. São criados fatos na KB, contendo informações sobre o nome da variável, seu tipo e tamanho. Um vetor relaciona a variável com as Unidades de Programa (UP) que as declaram. Este vetor é atualizado por outros SOTs, com informações sobre o tipo de uso das variáveis nas UPs.

4.3.4 Identificar Macro-Comandos

Na gramática DFP são encontrados macro-comandos que executam funções complexas, os seguintes macro-comandos são tratados por este SOT:

- "Enter", macro-comando que realiza a entrada de dados para uma ou mais tabelas do BD. Possui todas as funcionalidades comuns para telas de entrada de dados como inclusão, alteração, pesquisa e exclusão. O relacionamento entre janelas (campos de entrada de dados) e os campos do arquivo são feitos através de comandos "Entry". Este macro-comando possui ainda um conjunto de sub-rotinas pré-definidas, acionados por gatilhos ou por teclas de função. As sub-rotinas são destinadas à implementação de códigos executados durante a leitura, gravação, exclusão, criação de registros, limpeza das informações da tela e validações para gravação e exclusão. Este macro-comando sinaliza a definição de uma suposta classe de regras de negócio, para conter os métodos acionados pelos gatilhos e teclas de função, além de controlar as validações da entrada de dados;
- "EnterGroup", este macro-comando tem funcionalidades semelhantes às descritas para o macro-comando "Enter". Destina-se também ao tratamento de entrada de dados através de uma tela de interação. Porém, o macro-comando "EnterGroup" não vincula diretamente as janelas da tela de interação com campos das tabelas do BD, mas sim com variáveis. Este macro-comando também sinaliza a definição de uma suposta classe de regras de negócio;
- "ScanEnter", este macro-comando cria um controle de entrada de dados multi-linhas, de forma semelhante ao macro-comando "Enter". Todas as funcionalidades do macro-comando "Enter" estão disponíveis neste macro-comando, com a diferença que o macro-comando "Enter" realiza entrada de dados simples, no estilo de um formulário, e o macro-comando "ScanEnter" realiza a entrada de dados na forma de uma tabela. Este macro-comando sinaliza a definição de uma suposta classe de regras de negócio, para inserir o tratamento das validações e demais sub-rotinas, e direciona a definição de supostas classes de interface que tratem a entrada de dados na forma de tabela;
- "Listar", macro-comando que constrói uma lista para pesquisa em uma ou mais tabelas do BD. Este macro comando apenas lista as informações um uma tabela, não permite a entrada de dados como o macro-comando "ScanEnter". Podem ser inseridas sub-rotinas para tratar o filtro das informações das tabelas do BD que serão apresentadas. Este macro-comando sinaliza a criação de uma suposta classe de regras de para tratar os filtros e

uma suposta classe de interface para apresentação da lista;

- "ListarPnts", este macro-comando tem o funcionamento semelhante ao macro-comando "Listar", porém permite inserir eventos para acionamento de telas auxiliares, destinadas a manutenção das informações ou apresentação de detalhes. Também pode ser usado em conjunto do o macro-comando "Point_Select", para criação de listas de pesquisa onde pode ser selecionado um registro. Este macro-comando sinaliza a criação de supostas classes de regras de negócio e de interface para possibilitar as criações de objetos normalmente conhecidos como "LookUps" ou "Prompts"; e
- "Report", este macro-comando controla a criação de relatórios. Possui tratamento para pesquisa e seleção de informações em uma ou mais tabelas do BD, sub-rotinas pré-definidas para a criação de quebras em até nove níveis, além de controlar a quebra de página, cabeçalho, rodapé e numeração de página. Este macro-comando sinaliza a criação de supostas classes de regras de negócio para tratamento dos filtros e estrutura de pesquisa do relatório, e supostas classes de interface geração do relatório.

Para cada macro-comando tratado foi criado um SOT auxiliar, com transformações específicas para o reconhecimento e tratamento de cada macro-comando. A KB é alimentada com informações sobre as supostas classes identificadas.

4.3.5 Identificar Unidades de Programa (UP)

A identificação das Unidades de Programa (UP) no código legado permite definir supostos métodos das supostas classes. As UPs podem ser: procedimentos, funções, sub-rotinas e blocos de código com execução seqüencial e sem desvios de fluxo ou interação com o usuário. São usadas as técnicas de particionamento de supostos métodos no Fusion/RE [Pen96], experiências de transformação no Draco-PUC [Fuk99, Jes99] e as técnicas de formação de Hiper-Blocos propostas por Scott Mahlke [Mah96], para identificar as UPs.

As UP são identificadas e enumeradas seqüencialmente. O SOT possui um conjunto de transformações que percorre o código legado inicialmente para localizar procedimentos, funções e sub-rotinas, que são caracterizados como UP. O corpo principal do código também é caracterizado como uma UP. Cada UP identificada é rotulada com um padrão de nomenclatura que inclui o prefixo "Bloco_Comandos", seguido do nome do programa, e da seqüência de reconhecimento da UP. A UP é submetida a um SOT auxiliar

que seu corpo, identificando a ocorrência de comandos de desvio de fluxo ou de interação com o usuário. Ao identificar um comando que quebre a seqüencialidade do código, a UP é subdividida em novas UPs, que são novamente submetidas ao SOT para verificar a necessidade de novas subdivisões. Na KB são gravadas informações sobre as UP identificadas e sua hierarquia de chamadas.

A Figura 32 mostra um exemplo de transformação que localiza a quebra de UPs por comandos de interação. À esquerda da figura é apresentado o código da transformação "LocalizaQuebraPorInteracao", que é uma das transformações do auxiliar SOT "EncontarQuebradeBloco". Este SOT auxiliar é acionado pelas transformações que identificam UPs, para verificar a necessidade de subdividi-la devido aos comandos de interação, ou desvios de fluxo. À direita da figura é apresentado o gráfico de um trecho do SOT "Identifica_UP", gerado pelo DDE.

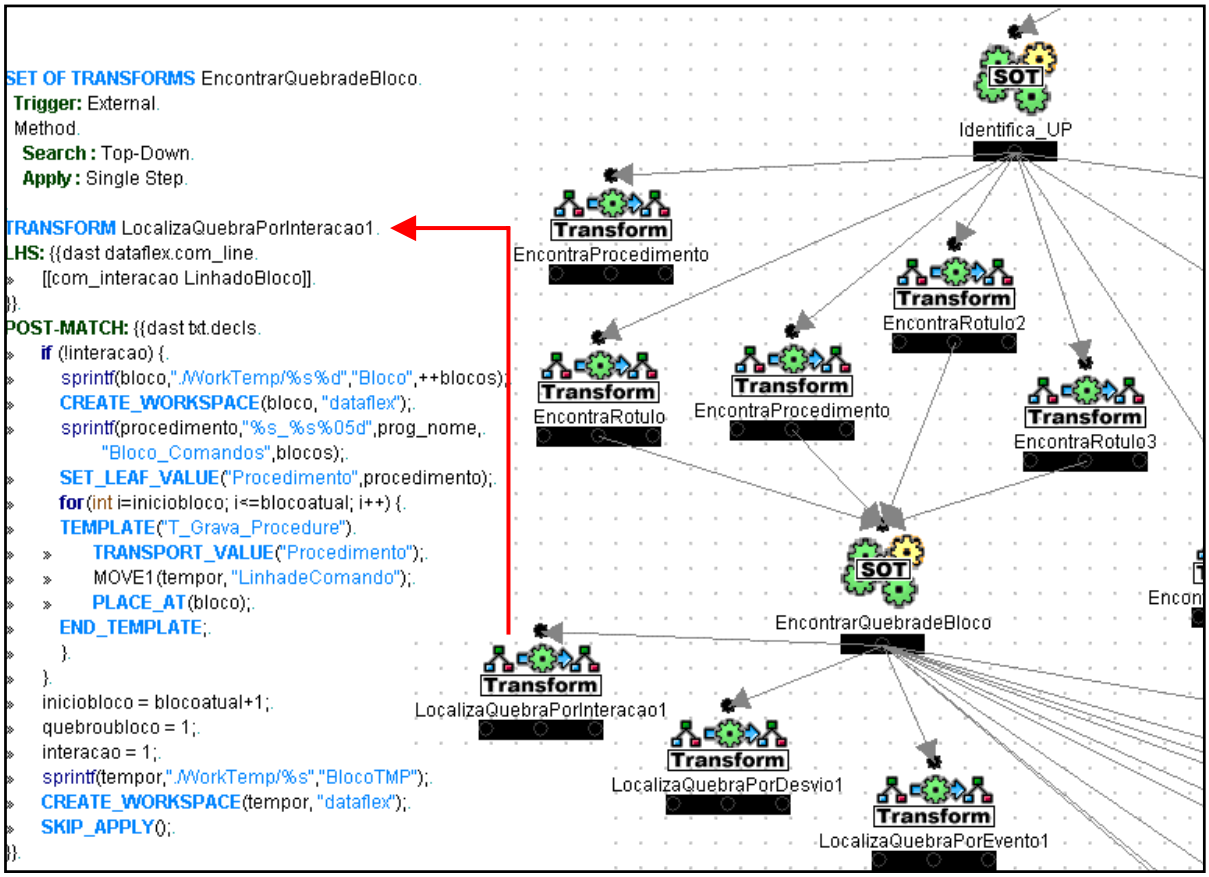


Figura 32. Transformação que localiza quebra de UP por comandos de interação.

Um SOT auxiliar foi criado para tratar as UP que são definidas dentro das estruturas de macro-comandos. As UP identificadas dentro dos macro-comandos recebem uma identificação diferenciada na KB, indicando a origem do macro-comando e a

funcionalidade ou *trigger* que a inicia. Como por exemplo, dentro do macro-comando "Enter", normalmente são identificadas sub-rotinas, rotuladas com o nome "Enter.Save:" que são acionadas no momento que as informações da tela de interação são gravadas nas tabelas do BD.

Os comandos de desvio de fluxo, como "GoTo" e "GoSub", que podem ser precedidos por declarações condicionais, como "if" e "case", são utilizados para sinalizar a quebra de uma UP, objetivando supostos métodos com o corpo mais simples, facilitando o tratamento e alocação em supostas classes.

Comandos de interação com o usuário, como o "Entry", "EntryItem", "Accept" e "InKey" também definem a quebra das UP, pois podem indicar um desvio de fluxo, causado pelo acionamento de eventos por parte do usuário através de teclas de função definidas pelo comando "KeyProc". Os comandos de interação são mantidos em uma única UP, quando são precedidos pelo comando "KeyProc Off", até a ocorrência de um comando "KeyProc On". O comando "KeyProc Off" inibe a execução de eventos durante a interação, impedindo a ocorrência de desvios de fluxo.

4.3.6 Eliminar Unidades de Programa redundantes

Depois de identificadas, este SOT compara as UP, identificando redundâncias. A comparação é realizada através da estrutura da árvore sintática de UP, ignorando diferenças de formatação. Ao identificar UP idênticas, a segunda identificada é eliminada, a hierarquia de chamadas atualizada substituindo as referências da UP eliminada pela compatível. Os comandos que realizam desvios para a UP eliminada também são atualizados. Este SOT também identifica e elimina as UP que não fazem parte do fluxo de execução do programa considerada como código morto. A eliminação de redundâncias e de código morto reduz a quantidade de supostos métodos, simplificando o posterior tratamento das conexões de mensagem.

4.3.7 Tratar desvios de fluxo

Este SOT trata os comandos de desvios de fluxo, visando reduzir sua quantidade. O método *peephole* para otimização do fluxo de controle [Aho95, Dav84], mostrado na Figura 33, é utilizado para reduzir UPs nos seguintes casos: Sejam A, B e C UPs

referenciadas entre si e não referenciadas por outras UPs. (1) Se A possui um desvio incondicional para B e a primeira instrução de B é um desvio incondicional para C, então pode-se substituír o desvio de A para B por um desvio de A para C, eliminando B; (2). Se A possui um desvio condicional para B e a primeira instrução de B é um desvio incondicional para C, então pode-se substituír o desvio condicional de A para B por um desvio condicional de A para C, eliminando B.

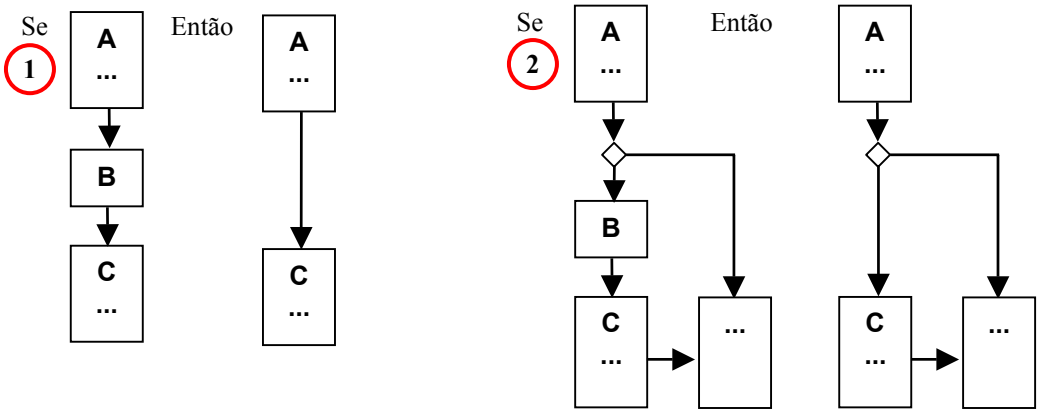


Figura 33. Tratamento do método de otimização de fluxo de controle (*peephole)*

Também são identificados os casos onde é possível a substituição de desvios do tipo "GoTo", por chamadas de sub-rotinas tipo "GoSub". Nestes casos pode ocorrer a quebra de UP em outras UPs. A Figura 34 mostra o tratamento nos casos: Sejam A, B e C UPs referenciadas entre si e não referenciadas por outras UPs.

Caso (**a**) – Se A possui em sua última instrução um desvio condicional ou incondicional para C (1), a última instrução de C é um desvio incondicional para B (2), B segue A e em sua última instrução possui um desvio incondicional para a próxima UP depois de C (3), então a instrução de desvio de A para C pode ser substituída por uma chamada de sub-rotina (4), e o desvio incondicional de C para B pode ser substituído pela instrução de retorno da sub-rotina (5);

Caso (**b**) – Se A possui em sua última instrução um desvio condicional ou incondicional para C (6), C tem um desvio condicional para B (7), B segue A e em sua última instrução possui um desvio incondicional para a próxima UP depois de C (8), então C deve ser subdividida em duas UPs. A primeira parte de C, C1, vai até o comando de desvio condicional, e a segunda, C2, depois do desvio condicional (9). É feita a inversão do teste

condicional ao final de C1, para realizar o desvio para C2 (10). A instrução de desvio de A para C pode ser substituída por uma chamada de sub-rotina para C1 (11). O desvio incondicional de C para B é substituído pela instrução de retorno da sub-rotina C1 (12);

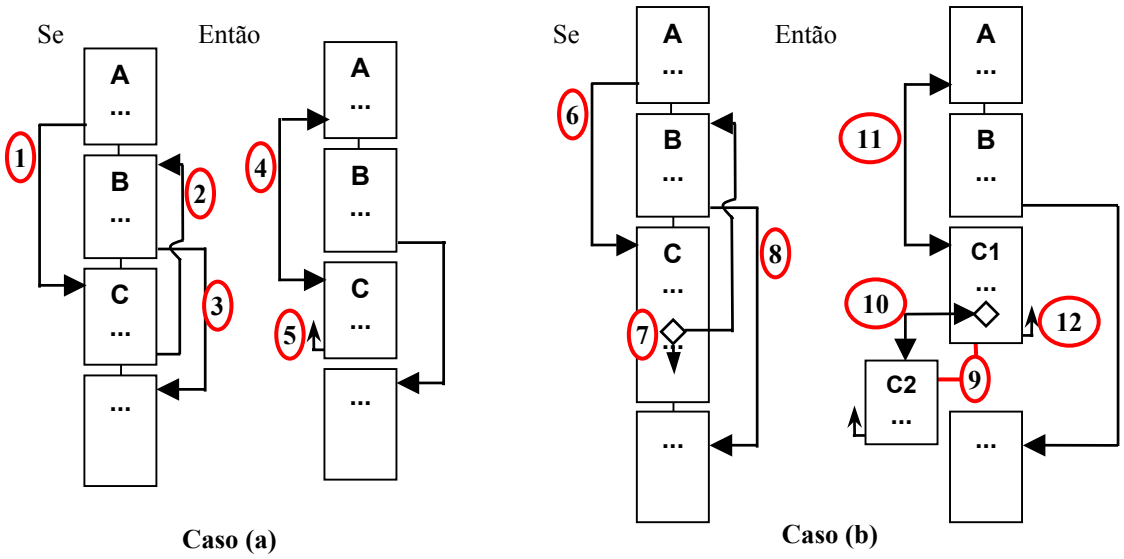
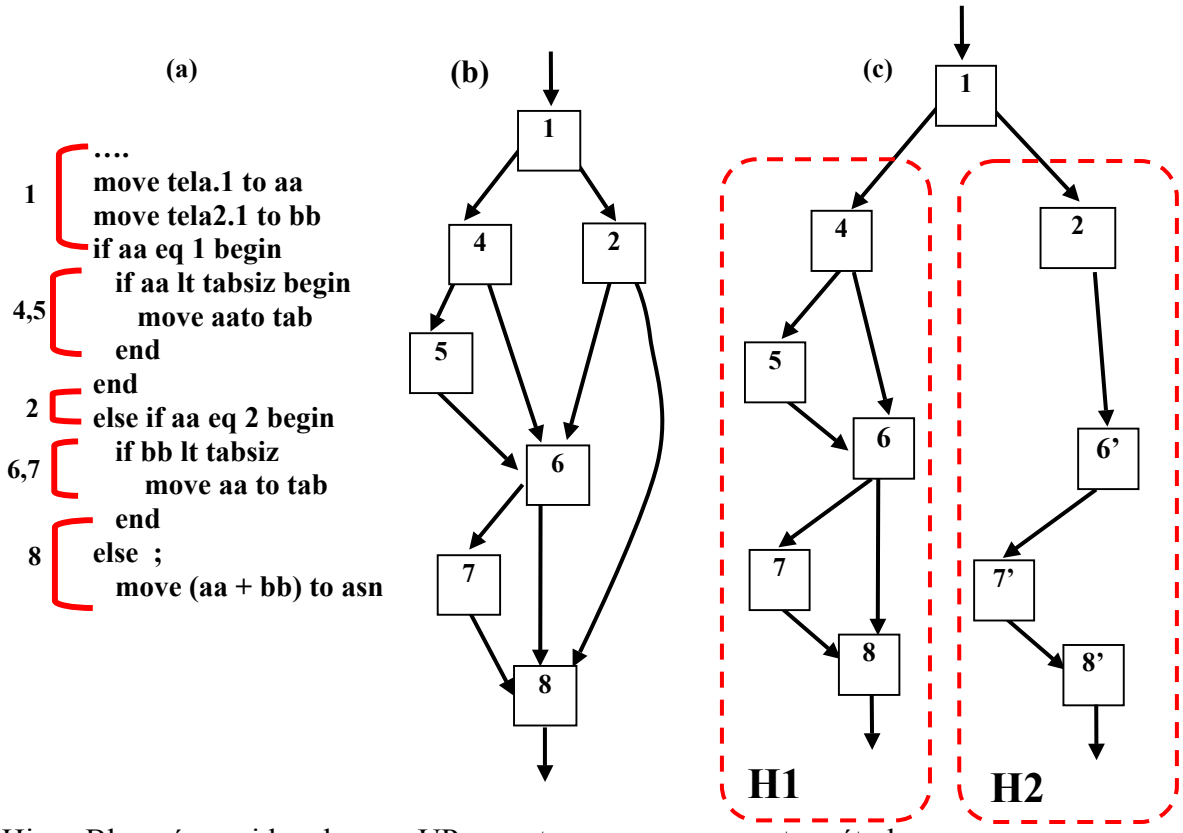


Figura 34. Substituição de desvios "GoTo" por "GoSub"

Em alguns casos é necessária a duplicação de UP para facilitar o controle dos blocos de código. Com base nas técnicas de formação Hiper-Blocos, propostas por Scott Mahlke [Mah96], e implementadas por Nelson Amaral no projeto do "Gerador de Código do Pro64" [Ama01], foi utilizada a técnica de Duplicação Agressiva para o tratamento das UPs. Os Hiper-Blocos são formados por regiões de fluxo de execução com única-entrada e múltiplas-saídas. Geralmente envolvendo estruturas de repetição e seleção.

A Figura 35 mostra um exemplo de tratamento de Hiper-Blocos com Duplicação Agressiva, onde em (a) está um trecho de código DFP, com um encadeamento de comandos if. Em (b) o grafo do fluxo de execução deste trecho de código e em (c), o grafo do fluxo de execução após o tratamento para formação dos Hiper-Blocos. Os blocos estão numerados de 1 a 8, e foram divididos com base na seqüencialidade de execução do código. O algorítmo de Duplicação Agressiva busca agrupar a maior quantidade possível de blocos em um Hiper-Bloco, de forma que cada Hiper-Bloco possua um único fluxo de entrada e saída. Neste caso, partido do bloco (1), o algorítmo percorre a seqüência de blocos até o final formando o H1 como os blocos (4,5,6,7 e 8). Depois segundo fluxo do bloco (1) é percorrido formando o H2. Ao encontrar um bloco que já foi alocado para H1, este é duplicado. Assim H2 é formado pelos blocos (2, 6', 7' e 8'), com a duplicação dos blocos (6, 7 e 8). Cada



Hiper-Bloco é considerado uma UP, para tornar-se um suposto método.

Figura 35. Formação de Hiper-Blocos com Duplicação Agressiva no código DFP

Durante o tratamento do fluxo de execução, a hierarquia de chamadas na KB é atualizada. Os tratamentos realizados nos desvios de fluxo, apesar de utilizarem técnicas de otimização do código não visam a redução de custos de execução, mas sim a simplificação do tratamento das UPs, para facilitar sua alocação em supostos métodos.

4.3.8 Tratar Eventos

O DFP suporta o tratamento de eventos através de teclas de função, ou de gatilhos em macro-comandos. Para que seja possível alocar as UP em supostos métodos e depois realizar o encapsulamento em supostas classes, devem ser identificados todos os pontos de acionamento destes métodos, sejam através de comandos de desvio de fluxo, tratado no SOT anterior, ou através de eventos. O tratamento dos eventos também direciona a definição dos eventos acionados por teclas de função para os objetos da camada de interface.

O SOT construído para tratar eventos possui um conjunto de transformações

que localizam as diversas formas de gatilhos para acionamento de eventos no DFP, como o comando “KeyProc”, ou caso de gatinhos acionados em macro-comandos, os eventos são localizados através de rótulos pré-definidos em cada macro-comando. Por exemplo, o macro-comando “Enter”, aciona através de gatilhos uma sub-rotina rotulada por “Enter.Save:”, antes de salvar as informações no BD. O macro-comando “Report” aciona a sub-rotina rotulada por “Section Footer” toda vez que termina uma página de impressão.

São armazenas informações na KB sobre os eventos encontrados para os macro-comandos, relacionando a suposta classe do macro-comando, com as UP, indicando o tipo de evento pré-definido nos macro-comandos.

Para os eventos acionados por teclas de função, através da declaração “KeyProc”, é acionado um SOT auxiliar, com um conjunto de transformações que reconhece os vínculos entre os eventos, os comandos de interação nas UP e as janelas das telas de interação com o usuário. As teclas de função podem ser acionadas pelo usuário durante uma interação em uma janela. Ao acionar uma tecla de função, o usuário faz com que seja realizado um desvio de fluxo da UP que contém o comando de interação para a UP vinculada à tecla de função pressionada. Assim, cada janela de interação pode ser considerada como um desvio condicional no fluxo de execução, sendo que a condição para ocorrência do desvio é o acionamento da tecla de função. Desta forma, a hierarquia de chamadas na KB é atualizada, para considerar também os desvios condicionais do fluxo de execução, ocasionados por eventos em todos os comandos de interação.

O conteúdo da UP acionado pelo evento, pode ser subdividido, quando são identificados trechos que são executados apenas para determinadas janelas de interação, através de comandos condicionais para teste de telas e janelas como “IfImage” e “IfWindow”, ou pela utilização de variáveis pré-definidas para tratamento das janelas, como “WindowIndex”, “CurrentWindow” e “CurrentImage”. A subdivisão das UP permite simplificar o tratamento dos eventos em cada ponto de interação, e posteriormente guiar a criação dos eventos e conexões de mensagens no código VDFOO.

4.3.9 Definir relacionamento entre janelas e campos de arquivo

Em DFP, comumente são utilizadas telas de interação para criação de entradas

de dados para tabelas do BD. Estas entradas de dados normalmente são feitas utilizando janelas vinculadas aos campos da tabela, através dos macro-comandos "Enter", "EnterGroup" e "ScanEnter". Dentro do corpo destes macro-comandos, são encontrados comandos como o "Entry" e o "ScanEntry", que relacionam diretamente uma janela de interação com um campo específico da tabela do BD. Neste caso, as unidades de programa acionadas para validação das informações digitadas, refletem as validações das informações do campo da tabela do banco de dados, e podem ser direcionadas para a classe de regras de negócio criada para tratar as validações da tabela.

Também são identificados vínculos indiretos entre a janela de interação e o campo de arquivo que utilizam uma variável como meio intermediário. A identificação desta situação pode ser feita quando um comando de interação vincula uma janela de uma tela com uma variável e não com um campo de uma tabela do BD. Neste caso, é utilizado um SOT auxiliar, que possui transformações que localizam o tratamento de leitura e gravação de valores na variável, considerando o fluxo de execução. Por exemplo, Seja "`vv`" uma variável, "`jj`" uma janela de interação pertencente a uma tela, e "`cc`" um campo de uma tabela no BD. Seguindo o fluxo de execução do código DFP, é localizado um comando atribuindo o valor de "`cc`" para "`vv`" ("`Move cc to vv`"). Seguindo novamente o fluxo de controle, caso o valor de "`vv`" não sofra novas atribuições, é localizado um comando de interação, vinculando a variável "`vv`" a janela "`jj`" ("`Accept jj to vv`", ou "`Accept jj`" e "`Move jj to vv`"). Continuando o fluxo de execução, caso o valor de "`vv`" não seja alterado, é encontrado um comando de atribuição de "`vv`" para "`cc`" ("`Move vv to cc`"). Desta forma, fica estabelecido o vínculo indireto entre a janela "`jj`" e o campo "`cc`".

Ao estabelecer a relação entre uma janela e um campo, as regras sobre os supostos atributos de interface na KB são atualizadas para refletir esta relação. Também são criadas regras na KB sobre características de formatação ou validações simples da entrada de dados, como aceitação apenas de letras maiúsculas, através da cláusula "capslock" ou validação de faixas de valores através da cláusula "Range" no comando "Entry".

4.3.10 Definir a utilização de janelas e campos nas Unidades de Programa

Este SOT possui um conjunto de transformações que varrem o código de cada UP identificada, para localizar comandos que manipulem janelas de interação e campos de

arquivo. São criadas informações estatísticas das janelas e campos de arquivo utilizado por cada UP, e sobre o tipo de tratamento dado para cada uma das janelas ou campos, se a UP realiza apenas a leitura, a gravação ou ambos. Estas informações serão importantes para possibilitar a alocação das UPs como supostos métodos nas supostas classes.

As transformações desse SOT partem de cada UP identificada e acionam um SOT auxiliar que trata o corpo da UP, primeiro para identificar a utilização das janelas e depois para localizar a utilização dos campos. Ao localizar um comando que trata uma janela ou campo, é identificado o tipo de tratamento dados, leitura ou gravação, então é realizada uma pesquisa na KB, para verificar se a janela ou campo já está vinculado a UP, e são atualizadas as informações sobre o tipo de utilização.

4.3.11 Definir a utilização de variáveis nas Unidades de Programa

A definição da utilização de variáveis nas UPs, é feita de forma semelhante a realizada para definição da utilização de janelas ou campos. Porém, para as variáveis, também é feita a identificação de informações relativas a atribuição de valores para cada variável, seguindo o fluxo de controle. Esta identificação é essencial para possibilitar a definição de quais variáveis podem ser passadas como parâmetros para as UPs quando forem alocadas como supostos métodos. Também possibilita identificar as variáveis que devem ser alocadas como supostos atributos nas supostas classes e quais devem ser alocadas como supostos atributos da suposta classe principal. Para possibilitar este tratamento, são criados vetores para cada variável, resumindo sua utilização nas UP seguindo o fluxo de execução.

4.3.12 Alocar Unidades de Programa como supostos métodos.

Partindo das UPs de programa e supostas classes identificadas, das informações na KB sobre a utilização de janelas e campos nas UP, é definida a responsabilidade sobre a UP para uma suposta classe, encapsulando a UP como um suposto método da suposta classe.

A definição de responsabilidade utiliza os princípios do método Fusion/RE [Pen96] e de experiências anteriores na transformação de sistemas no Draco-PUC [Fuk99, Jes99]. Foi estabelecida uma fórmula de cálculo com base na estatística de utilização das janelas e campos de arquivo pela UP. Para cada UP, são realizados os cálculos de pontuação

da utilização de atributos para cada suposta classe identificada. No cálculo são considerados os supostos atributos de cada suposta classe, e o tipo de tratamento feito com os supostos atributos na UP que está sendo tratada. A Figura 36 mostra a fórmula utilizada, sendo que o resultado do cálculo "vu" é armazenado em um vetor com uma linha para cada classe. Depois de realizado o cálculo para todas as classes, a UP é alocada como suposto método para a suposta classe com maior valor de "vu". Caso o valor máximo de "vu" não seja exclusivo, é adotada a prioridade para supostas classes de regras de negócio, seguida pelas de supostas classes banco de dados e finalmente pelas supostas classes de interface. Caso ainda existam

```
Seja:  vu = Valor de utilização da classe para UP.
        x = Quantidade de utilizações de supostos atributos
            para leitura.
        y = Quantidade de utilização de supostos atributos
            para gravação.
        z = Quantidade de utilização de supostos atributos de
            outras classes.
        t = Tipo de classe.
            t=1 para classes de interface.
            t=2 para classes de banco de dados .
            t=3 para classes de regras de negocio.

Tem-se que: vu = t.( 4.u + 8.y – 2.z )
```

duas ou mais supostas classes na mesma camada, com o valor máximo de "vu", a UP é alocada como suposto método para a primeira identificada no vetor.

Figura 36. Formula para calculo do valor de utilização da classe pela UP

A KB é atualizada para indicar o encapsulamento do suposto método. Também são atualizadas as informações sobre a localização do suposto método na hierarquia de chamadas.

4.3.13 Alocar variáveis como supostos atributos ou variáveis locais

Com os supostos métodos alocados nas supostas classes, as variáveis podem ser alocadas conforme sua utilização. Uma variável pode ser alocada como local em supostos métodos, pode ser alocada como parâmetro dos supostos métodos ou como supostos atributos das supostas classes.

Variáveis locais são definidas quando são utilizadas dentro de apenas um suposto método, ou quanto a primeira referencia referência da variável em todos os supostos métodos que a utilizam é um comando de atribuição de valor.

Parâmetros dos supostos métodos são definidos quando o valor da variável é atribuído no nível de hierarquia de chamadas anterior ao do suposto método, e dentro do corpo do método o valor da variável não é alterado.

Supostos atributos são definidos quando o tratamento da variável não se enquadrar nos casos anteriores. Neste caso, são adicionadas instruções para leitura do valor do suposto atributo no início do suposto método, e instruções para atualização do valor do atributo antes da finalização do suposto método.

4.3.14 Criar suposta classe principal e demais supostas classes

Para a geração do código DFP Organizado, que contém o código particionado da suposta classe principal e demais supostas classes, são utilizadas as informações da KB. Inicialmente é gerado o código da suposta classe principal, com o método principal de execução. O método principal é criado com base nos primeiros níveis da hierarquia de chamadas, para que sejam acionados os supostos métodos das demais supostas classes, respeitando o fluxo de execução do código legado.

Após a geração do código da suposta classe principal, são geradas as demais supostas classes, utilizando os meta-símbolos para identificar cada elemento das supostas classes, como seus supostos atributos, e supostos métodos.

4.4 Construção do Transformador de DFP Organizado para VDFOO

O Transformador de DFP Organizado (DFPO) para VDFOO foi construído para automatizar o passo “Reimplementar Código Organizado”. A construção do transformador foi feita em dois passos, como mostra a Figura 37.

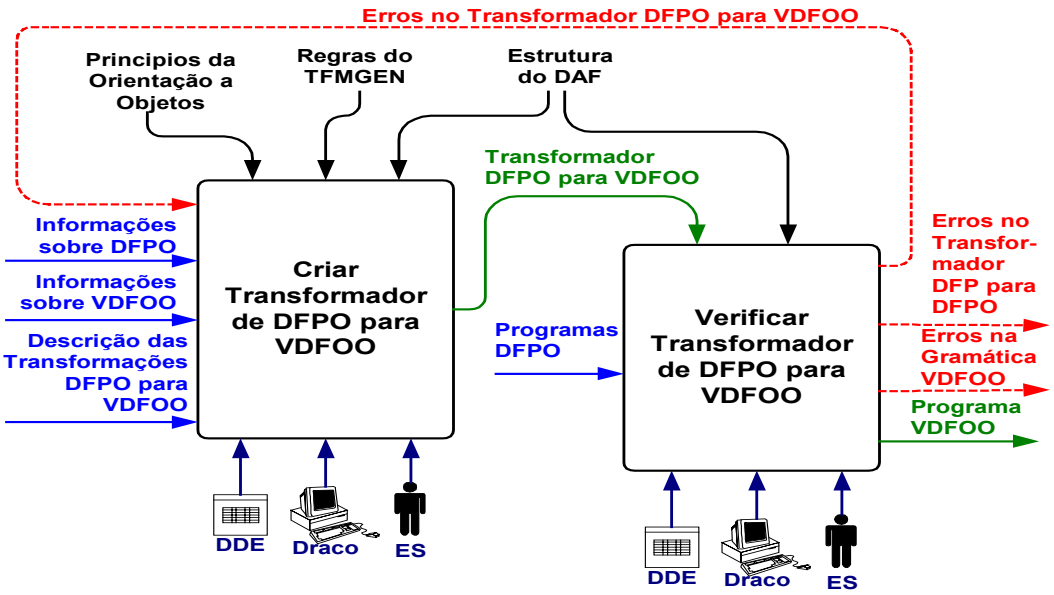


Figura 37. Construção do Transformador de DFP Organizado para VDFOO

No passo **Criar Transformador de DFPO para VDFOO**, parte-se de informações a estrutura do código em DFPO, de informações sobre o VDFOO, principalmente sua gramática, e da descrição das transformações de DFPO para VDFOO. Este passo é orientado pelos princípios da orientação a objetos, pelas regras de escrita do TFMGEN do Draco-PUC e pela estrutura do DAF.

O DDE auxilia o Engenheiro de Software (ES) na edição do transformador. O Draco-PUC é usado para gerar o transformador a partir das descrições das transformações.

No passo **Verificar Transformador de DFPO para VDFOO**, aplica-se o transformador construído sobre os programas em DFPO, obtidos de programas DFP submetidos ao Transformador de DFP para DFPO, reimplementando programas em VDFOO. A estrutura do programa gerado é analisada, buscando identificar falhas na reimplementação para VDFOO. Ao localizar uma falha, deve ser identificada a origem desta falha, que pode ser causada pelas definições erradas das transformações de DFP para DFPO, ou nas transformações de DFPO para VDFOO, ou da gramática VDFOO.

As transformações para reimplementação do código em VDFOO percorrem o código DFPO para reimplementar cada classe, com base nas camadas do DAF. O transformador encontra classes com definições de telas e gera classes de interface, com seus métodos e atributos. Classes da camada de interface do DAF são estendidas, conforme regras

definidas na KB, durante o passo Organizar Código Legado.

Para gerar as classes de acesso ao banco de dados, são identificadas as supostas classes que utilizam o banco de dados, gerando extensão da classe *DataSet* do DAF, para suprir os serviços de armazenamento e recuperação de informações no banco de dados.

Concluída a reimplementação das classes de interface e acesso ao banco de dados, faz-se a reimplementação das supostas classes da camada de regras de negócio.

As regras sobre a hierarquia de chamadas dos supostos métodos, tratamento das conexões de mensagens e eventos definem a classe do DAF a ser estendida. No caso particular de supostas classes baseadas em macro-comandos, o reuso do DAF na reimplementação da classe é definido conforme o macro-comando. Por exemplo, o macro-comando *report* identifica o reuso da classe *WinReport*.

A suposta classe para a qual não for identificada, qualquer conexão de reuso do DAF, é reimplementada estendendo a classe *UI_Object* que propicia os serviços básicos para conexão de mensagens, sendo considerada como uma classe auxiliar, é alocada na camada de regras de negócio.

As conexões de mensagem e eventos, entre os objetos, que estabelece o fluxo de execução do código, são reimplementadas em cada método de uma classe. A classe principal do sistema, com base na suposta classe principal do código legado, contém o tratamento de eventos que sinalizam cada caso de uso do sistema, nas interações com o ambiente externo.

Casos particulares em que o fluxo de execução não é sinalizado por eventos externos, são tratadas diretamente na geração das classes, em seus métodos, com base na hierarquia de chamadas, conexões de mensagem e eventos.

A Figura 38 mostra exemplos das transformações realizadas neste passo, onde à esquerda tem-se o código DFP Organizado, obtido no passo anterior e à direita o código gerado em VDFOO. Conforme indicado em (**1**), as supostas classes de interface dão origem a classes de interface. A estrutura de uma tela (**2**) é usada para criar objetos de interação que estendem classes de interface do DAF. O posicionamento da tela (**3**) é convertido em

unidades de medida do ambiente gráfico na construção da interface. Variáveis do código DFP organizado (**4**), que não foram encapsuladas em supostas classes, são encapsuladas na classe principal do sistema. A ligação entre as classes de interface e as de regras de negócio, baseia-se em comandos de interface do código legado que referenciam atributos das classes de regras de negócio (**5**) e (**6**). As supostas classes que acessam o banco de dados (**7**) são transformadas em classes da camada de banco de dados, que estendem a classe *DataSet* do DAF.

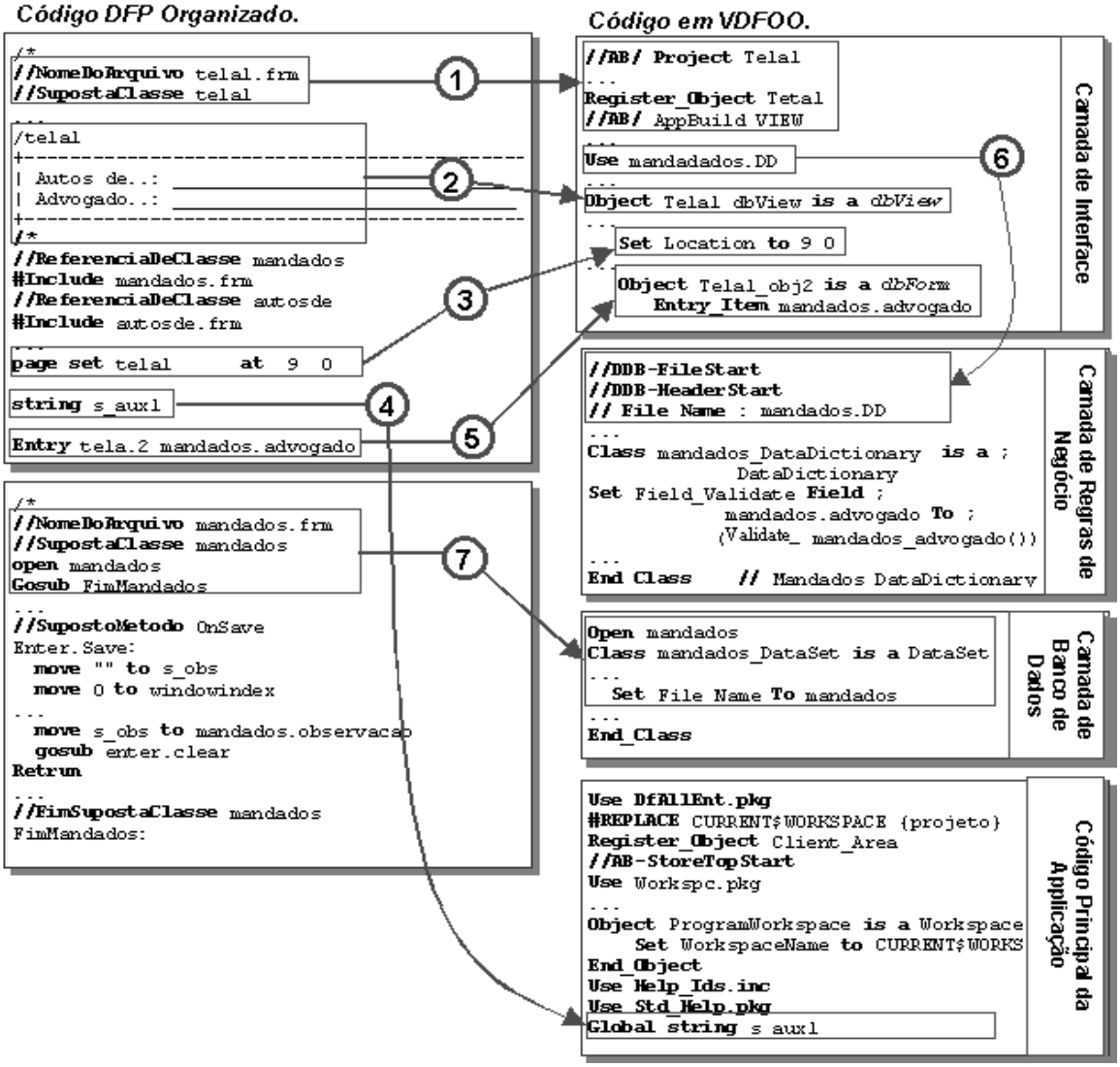


Figura 38. Exemplos da reimplementação do código organizado

Para facilitar o entendimento deste transformador, a Figura 39, apresenta uma visão global dos principais conjuntos de transformação (SOT – *Set Of Transforms*) que compõem o Transformador de DFPO para VDFOO. O transformador está agrupado em 8 conjuntos principais de transformação. Cada um dos SOT´s é composto de um ou mais transformadores, que podem acionar a execução de SOT´s secundários.

O transformador de DFPO para VDFOO baseia-se nas informações contidas na KB, carregadas pelo transformador anterior, nos meta-símbolos acrescentados no código DFPO. Durante a execução do transformador, cada elemento identificado pelos meta-símbolos tem seu conteúdo armazenado em *workspaces*. As referências sobre a localização das *workspaces* que contêm as supostas classes, atributos e métodos são atualizadas sobrepondo as localizações indicadas pelo transformador anterior.

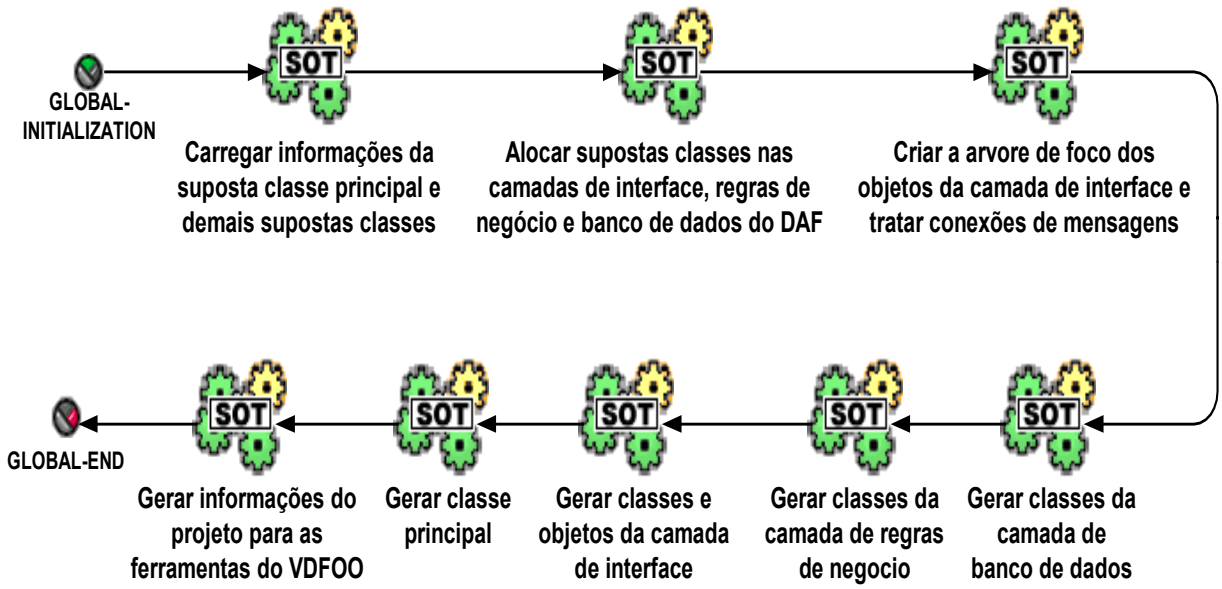


Figura 39. Principais SOT´s do Transformador de DFPO para VDFOO

Segue uma apresentação detalhada de cada SOT da Figura 39.

4.4.1 Carregar informações da suposta classe principal e demais supostas classes

Este SOT varre o código DFPO e reconhece os meta-símbolos inseridos pelo transformador anterior, carregando as informações sobre as supostas classes e seus elementos em *workspaces* para possibilitar o tratamento pelo transformador.

Durante a carga das informações, as regras da KB, que contêm a referência da *workspaces* de cada suposta classe, método e atribulo, são atualizadas para refletir a nova workspace utilizada.

Depois de carregadas as informações das supostas classes e seus elementos, este SOT carrega as informações sobre as associações das supostas, armazenadas na KB, e

reconstrói as estruturas de vetores internos sobre estas associações. As informações sobre as dependências são carregadas com base na KB e no reconhecimento dos meta-símbolos "`//ReferenciaDeClasse`" e "`//ReferenciaDeMetodo`" que indicam a chamada de métodos entre as classes.

4.4.2 Alocar supostas classes nas camadas de interface, regras de negócio e banco de dados do DataFlex Application *Framework*

Com as informações sobre as supostas classes, métodos, atributos, associações e dependências carregadas, este SOT realiza a distribuição das classes nas camadas do DAF. Para realizar esta alocação, é utilizado um SOT auxiliar que varre o código das supostas classes, separando as que foram identificadas com base em tabelas do banco de dados, como classes da camada de banco de dados. As classes que foram identificadas através de telas de interação, são alocadas na camada de interface e as demais ficam inicialmente alocadas na camada de regras de negócio.

As classes de acesso ao banco de dados são definidas para estender a classe "DataSet", do DAF, que supre os serviços de conexão com o BD. Desta forma, as classes da camada de banco de dados herdam os métodos de acesso ao banco, possibilitando que as classes das demais camadas utilizem seus serviços para acessar o banco de dados.

O código dos métodos de cada classe da camada de regras de negócio, é tratado para substituir chamadas de subrotinas por conexões de mensagens que realizam o acionamento dos métodos das classes da camada de banco de dados. Este SOT possui ainda transformadores que reconhecem os comandos de interação direta com o BD nos métodos das classes de regras de negócio, substituindo por conexões de mensagens para execução dos métodos estendidos da classe "DataSet" do DAF. Desta forma, a conexão com o BD é sempre gerenciada pela camada de Banco de Dados do DAF. Por exemplo, ao encontrar um comando do tipo "**`Save clientes`**", é feita a substituição pela mensagem "**`Send Request_Save to (clientes_DD(self))`**". O restante do corpo dos métodos é transformado para VDFOO através de um SOT com um grande conjunto de transformações locais.

Para definir a possibilidade de extensão das classes da camada de regras de negócio do DAF, são usadas as informações sobre a funcionalidade das classes na KB. Para cada classe de acesso ao BD, é criada uma classe estendendo a classe "DataDictionary" da

camada de regras de negócio do DAF. As classes que foram reconhecidas através de macro-comandos, estendem classe com funcionalidade equivalentes no DAF, como por exemplo, o macro-comando "Report" indica a extensão da classe "WinReport". As classes cuja funcionalidade não pode ser definida são tratadas como classes auxiliares ou classes utilitárias e estendem a classe "Ui_Object" do DAF, permanecendo na camada de regras de negócio.

Para o tratamento das classes da camada de interface é feito o mapeamento da utilização dos supostos atributos das supostas classes de interface, por parte dos métodos das classes da camada de regras de negócio e banco de dados. Durante o mapeamento, são identificadas características sobre o tipo de utilização feita, como por exemplo a utilização através do comandos "ScanEntry" ou listas de validações como a cláusula "check' do comando "Entry". Estes tipos de utilização servem para direcionar o uso das classes de interface do DAF, na criação dos objetos de interface em VDFOO.

As supostas classes de interface são alocadas na camada de interface como Objetos, sendo que para cada tela em DFPO é criado um objeto do tipo "DbView", caso a tela seja utilizada em macro-comandos tipo "Enter" ou "EnterGroup", ou do tipo "DbModalPanel", caso a tela seja utilizada através de comandos do tipo "Accept". Para telas que possuem apenas uma janela de interação cujos valores de entrada são validados como "S" ou "N", ou variações desta validação, são criados objetos do tipo "ConfirmationBox", como botões para cada tipo de opção no lugar da janela. Comandos do tipo "Pause" e "Error" indicam a criação de objetos do tipo "InfoBox" ou "StopBox".

São criados objetos para cada suposto atributo da suposta classe. São criados objetos do tipo "TextBox", "DbEdit", "DbComboBox", "DbGrid" e "DbSelectionList", todos usando classes da camada de interface do DAF. O posicionamento dos objetos é feito utilizando as informações sobre a estrutura da tela na KB, e o tipo de objeto é definido conforme o tipo de utilização. Sendo que para as informações textuais da tela, sempre são utilizados objetos do tipo "TextBox", com a fonte "Courier New" objetivando manter o formato original da tela. Para as janelas é utilizado como padrão a adoção de objetos do tipo "DbEdit", salvo nos casos onde for possível a identificação para utilização de outros tipos de objetos.

A Figura 40 mostra um trecho de uma das transformações que tratam as classes

de interface. O tratamento é realizado, pesquisando todas as classes de interface da KB (1), e para cada classe encontrada, todos os seus atributos (2). O tipo do atributo define o tratamento a ser realizado. Para os atributos que devem se tornar objetos de entrada de dados (3), são realizadas novas consultas na KB, para determinar o tipo de objeto a ser usado. Para os atributos com informações textuais (4), é feito o tratamento para formatação das informações. Ao final, as informações sobre a classe e seus atributos são armazenadas em *workspaces* (5).

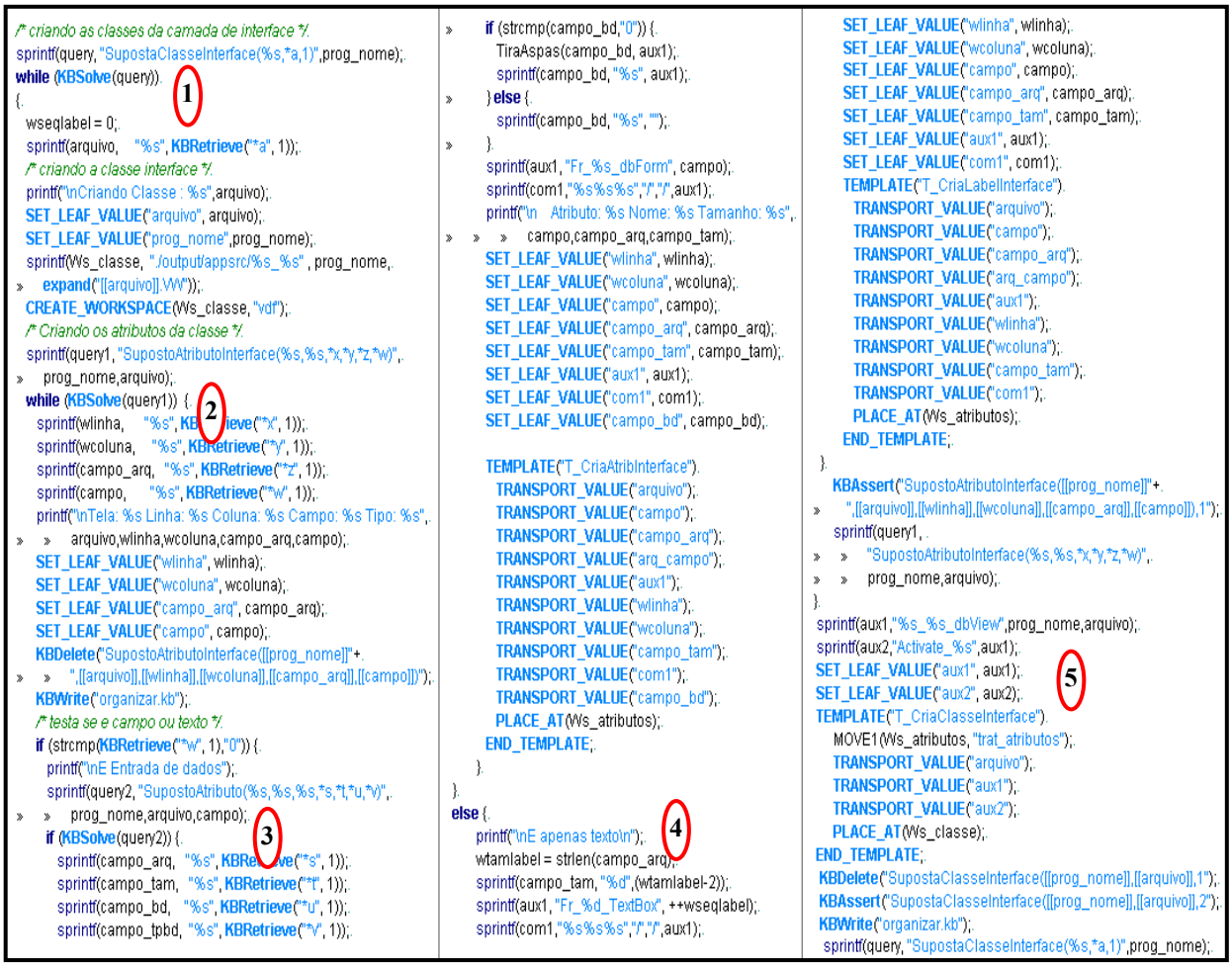


Figura 40. Trecho da transformação para tratamento das classes de interface

As informações sobre a extensão ou uso de classes do DAF são armazenadas na KB, para uso posterior durante a geração do código.

Este SOT também utiliza as informações sobre o tratamento de eventos da KB, para adicionar gatilhos para os métodos das classes de interface através de mensagens do tipo "On_key", que vinculam o acionamento de uma tecla de função com um método.

4.4.3 Criar a árvore de foco dos objetos da camada de interface

Este SOT utiliza as informações sobre a hierarquia de chamadas e sobre o fluxo de controle para estabelecer uma árvore de foco a ser seguida pelos objetos da camada de interface de forma a manter a mesma seqüência nas interações com o usuário. A árvore de foco no VDFOO é criada através do agrupamento de objetos e pela seqüência de definição no código.

A Figura 41 mostra o funcionamento da árvore de foco em relação às conexões de mensagem. Neste exemplo, o método "wrx" do objeto "D" chama o método "xyz" do objeto "C", porém para que a conexão funcione, é necessário estipular o caminho a ser percorrido pela mensagem na árvore de foco.

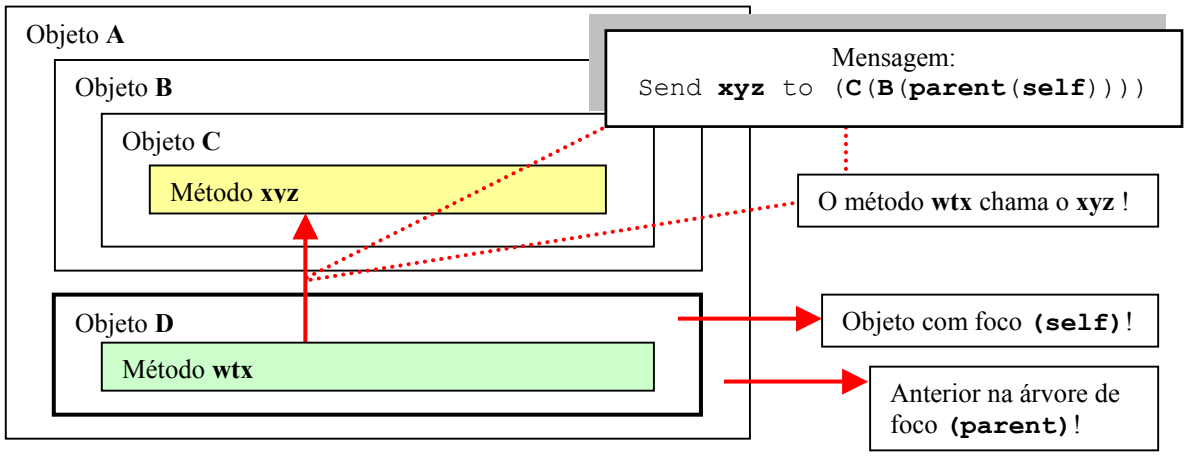


Figura 41. Funcionamento da árvore de foco dos objetos da Interface

Depois de estabelecida a árvore de foco, é acionado um SOT auxiliar que varre o código de todos os métodos que referenciam objetos da camada de interface e atualiza as conexões de mensagens com informações sobre a árvore de foco.

4.4.4 Gerar classes da camada de banco de dados

Este SOT possui um conjunto de transformações que lê as informações da KB sobre as classes alocadas na camada de interface e gera o código em VDFOO para estas classes. Cada classe é gerada em um arquivo separado, com o nome definido pelo nome da tabela do banco de dados seguido de ".ds".

A Figura 42 mostra exemplos dos *Templates* usados para gerar as classes da

camada de banco de dados. O *Template* "T_CriaClasseBD" cria uma classe, estendendo a classe "DataSet"do DAF, para cada tabela do banco de dados usada no código legado. Esta classe permite que o acesso ao banco de dados seja feito através de seus serviços, possibilitando a conexão com diversos gerenciadores de banco de dados de forma transparente. Os *Templates* "T_CriaFuncion" e "T_CriaProcedure" são usados para criar os métodos dentro da classe de banco de dados criada.

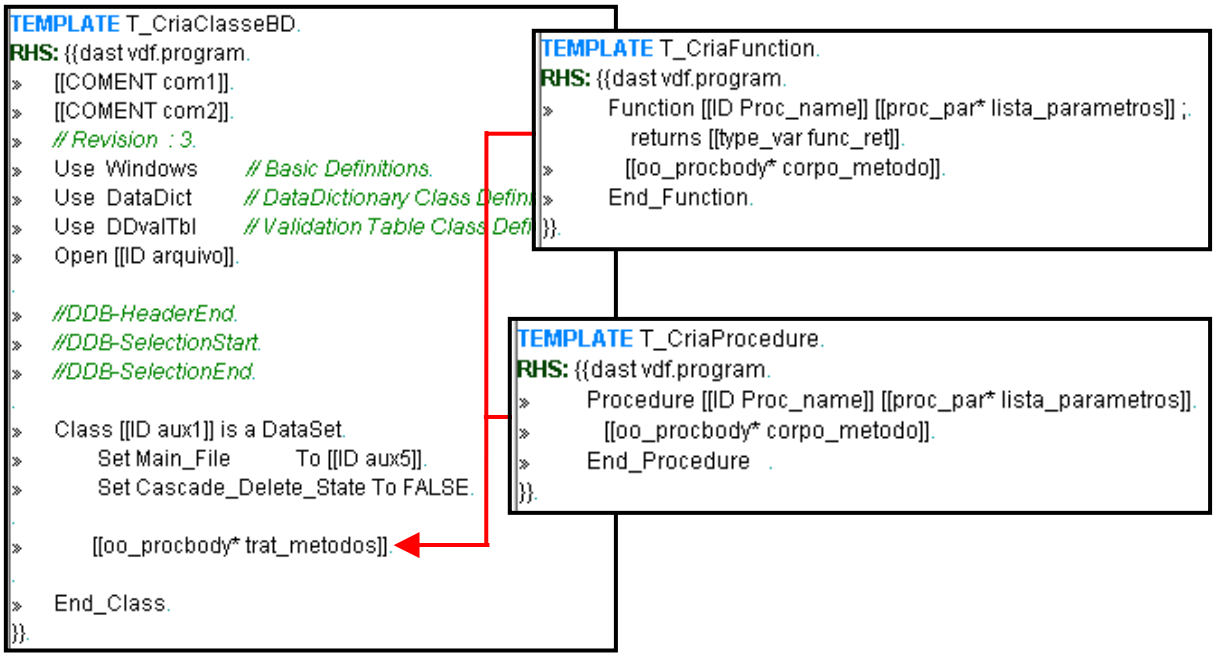


Figura 42. Exemplos de *Templates* para gerar classes de Banco de Dados.

As classes de banco de dados são utilizadas para gerenciar o acesso às tabelas do banco de dados. Para cada classe de banco de dados é criada também uma classe de regras de negócio que usa os serviços da classe de banco de dados adicionando métodos de validação e cálculos.

4.4.5 Gerar classes da camada de regras de negócio

Este SOT possui um conjunto de transformações que lê as informações da KB sobre as classes alocadas na camada de regras de negócio, e gera o código em VDFOO para estas classes. Cada classe é gerada em um arquivo separado, com o nome definido dependendo do tipo de classe que está sendo gerada, seguida de ".dd".

São utilizadas as informações sobre as associações e dependências entre as classes da camada de regras de negócio e de banco de dados, para que o código da classe de

regras de negócio possua comandos para inclusão do código das classes de banco de dados da qual dependa.

A Figura 43 mostra exemplos dos *Templates* usados para gerar as classes da camada de regras de negócio relacionadas às classes da camada de banco de dados. O *Template* "T_CriaClasseDD" cria uma classe, estendendo a classe "DataDictionary"do DAF, relacionada com a classe "DataSet" criada anteriormente (1). O *Template* cria "T_CriaAtribDD" é usado para tratar os atributos da classe de regras de negócio (2). Os *Templates* "T_CriaFuncion" e "T_CriaProcedure" também são usados nesta camada.

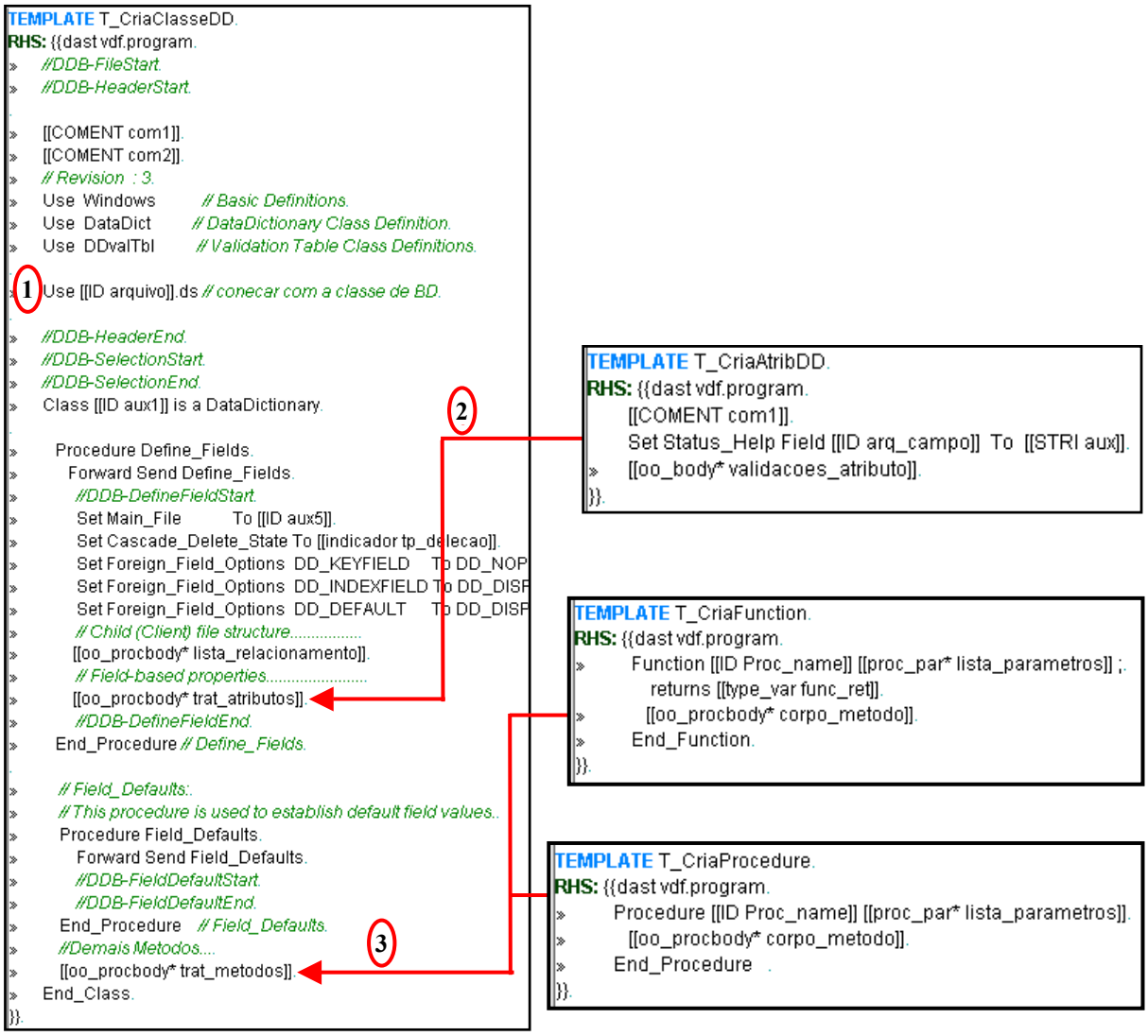


Figura 43. Exemplos de *Templates* para gerar classes de Regras de Negócio.

Outros tipos de classes além das estendidas da "DataDictionary" estão presentes na camada de regras de negócio, como por exemplo, classes que estendem a "BusinessProcess", utilizadas para execução de cálculos ou atualizações em grande

quantidade. Classes de suporte para geração de relatórios, estendidas da "WinPrint" e "WinReport". Classes utilitárias como as estendidas da classe "Array", para o tratamento de vetores, e da classe "ImageList" para tratar listas de imagens. Na camada de regras de negócio também ficam alocadas as classes para as quais não foi possível definir uma extensão apropriada das classes do DAF.

4.4.6 Gerar classes e objetos da camada de interface

A interface do sistema em VDFOO é gerada utilizando como base a árvore de foco, que contém a seqüência e agrupamento dos objetos. Primeiro são criados os arquivos de código para conter os objetos auxiliares da interface, como caixas de diálogo, de mensagens e telas secundárias. Depois são criados os arquivos de código para as telas principais e relatórios, que refletem os casos de uso do sistema.

A Figura 44 mostra o *Template* "T_CriaClasseInterface", que cria os objetos para as telas principais, usando a classe dbView do DAF. A árvore de foco é usada para seqüência de criação dos objetos usando os *Templates* "T_CriaLabelInterface" e "T_CriaAtribInterface", que criam os objetos componentes da dbView.

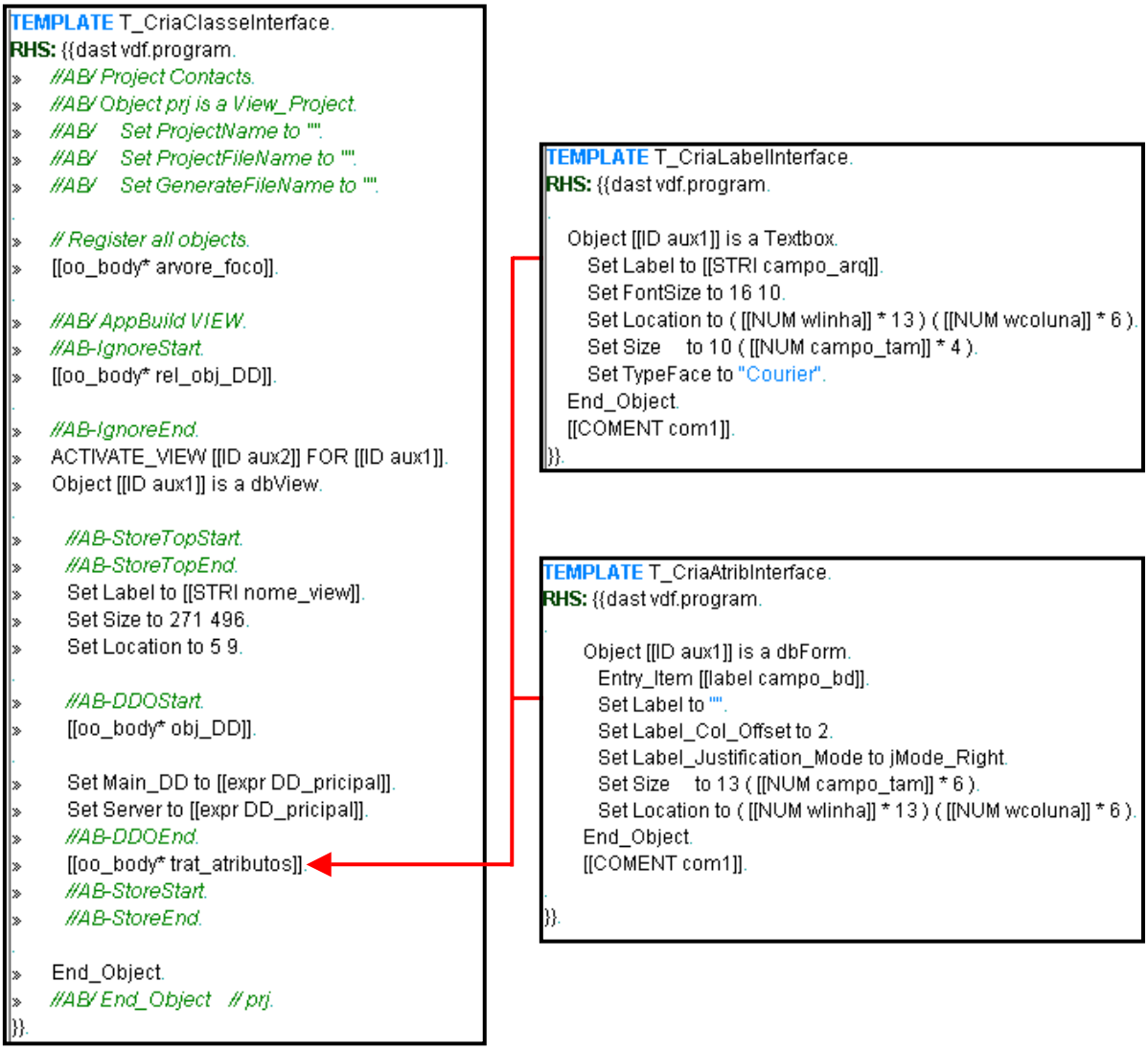


Figura 44. Exemplos de *Templates* para gerar classes de Interface

4.4.7 Gerar classe principal

Este SOT gera o código da classe principal, com o método principal de acionamento das demais classes e objetos. As informações sobre o primeiro nível da hierarquia de chamadas são usadas para guiar a geração das conexões de mensagem para acionamento dos casos de uso do sistema.

A Figura 45 mostra o *Template* para geração da Classe Principal. A hierarquia de chamadas estabelece a seqüência de acionamento dos casos de uso (1). A árvore de foco estabelece a seqüência de inclusão dos códigos dos casos de uso (2). O tratamento de eventos e o fluxo de execução estabelecem o acionamento de eventos e mensagens (3).

```
TEMPLATE T_CriaClassePrincipal.
RHS: {{dast vdf.program.
    Use DfAllEnt.pkg.
    #REPLACE CURRENT$WORKSPACE  [[ID codigo_prog]].
    Register_Object Client_Area.
    //AB-StoreTopStart.
    Use Workspc.pkg.
    //  Set date attributes as needed.
    Set_Date_Attribute sysdate4_State  to (TRUE).
    Set_Date_Attribute Date4_State     to (TRUE).
    Set_Date_Attribute epoch_value     to 30.
    Object ProgramWorkspace is a Workspace.
        Set WorkspaceName to CURRENT$WORKSPACE.
    End_Object.
    Use Help_Ids.inc // Developer should provide this file of help context links..
    Use Std_Help.pkg.
    //AB-StoreTopEnd.
    Object Main is a Panel.
      Set Label To [[STRI nome_prog]].
      DFCreate_Menu Main_Menu.
        #include File_PM.inc    // file pulldown menu.
        DFCreate_Menu "&View" ViewPopupMenu is a ViewPopupMenu.
            [[oo_body* lista_casos_uso]]. (1)
        End_Pull_down.
        Set Status_Help To "Available Views".
        #include Navi_pm.inc    // Naviation pulldown menu.
        #include win_PM.inc     // Window of available views.
        #include Helpa_PM.inc   // help pulldown menu w/ About.
      End_Menu.
      Use DFStdBtn.pkg // Tool-Bar object..

      Object Client_Area is a AppClientArea.
        // Include all views.
            [[oo_body* codigos_casos_uso]]. (2)
      End_Object // Client_Area.

      [[oo_body* acionamento_eventos]]. (3)

      Use DFStdSbr.pkg // Status-Bar object..
      //AB-PanelStoreStart.
      Use StdAbout.pkg // Standard "About" package..
      //AB-PanelStoreEnd.
    End_Object.
    //AB-StoreStart.
    Start_UI.
    //AB-StoreEnd.
}}.
```

Figura 45. *Templates* para gerar a classe principal

4.4.8 Gerar informações do projeto para as ferramentas do VDFOO

O último SOT gera o código do projeto para a ferramenta de desenvolvimento IDE do VDFOO. O código do projeto tem uma estrutura semelhante a da classe principal, porém é gravado com a extensão ".prj".

Este SOT gera também uma relação de todas as classes e objetos criados para que possam ser registradas nas ferramentas de desenvolvimento do VDFOO.

4.5 Definição do *Script* de Execução dos Transformadores no DDE

Depois de construídos os transformadores, estes devem ser verificados em conjunto para avaliar a necessidade de realizar correções ou ajustes no conjunto de transformadores. Estas validações são feitas através da transformação de sistemas que apresentem características de construção diferentes e uma grande quantidade de código.

Para facilitar a aplicação dos transformadores e realizar a transformação de sistemas, submetendo diversos códigos para transformação de forma automatizada, deve ser criado um *script* de execução dos transformadores para que a seqüência de operações necessárias à transformação seja respeitada, evitando que erros ocorram devido falta de método na aplicação dos transformadores.

Desta forma, foi definido um *script* de execução dos transformadores no DDE, que deve ser usado na transformação dos sistemas através dos transformadores apresentados.

A Figura 46 mostra o *script* de transformação no DDE, onde em (1) é apresentado o bloco principal do *script*, que chama um bloco auxiliar para iniciar o ambiente de transformação (2), que limpa os arquivos de diretórios de trabalho. Ainda no bloco principal, em (3), é apresentada a estrutura "LOOP_SOURCE", que define uma repetição do seu conteúdo para cada código legado a ser transformado. Dentro desta estrutura, é acionado o *script* do Draco-PUC "DfpToDfpo.dsf" (4) , que aplica o primeiro transformador em todos os códigos. Continuando na estrutura principal, (5) mostra a segunda estrutura de repetição, que aplica o segundo transformador em todos os códigos DFPO resultantes do primeiro transformador, através do script Draco-PUC "DfpOToVdf.dsf" (6).

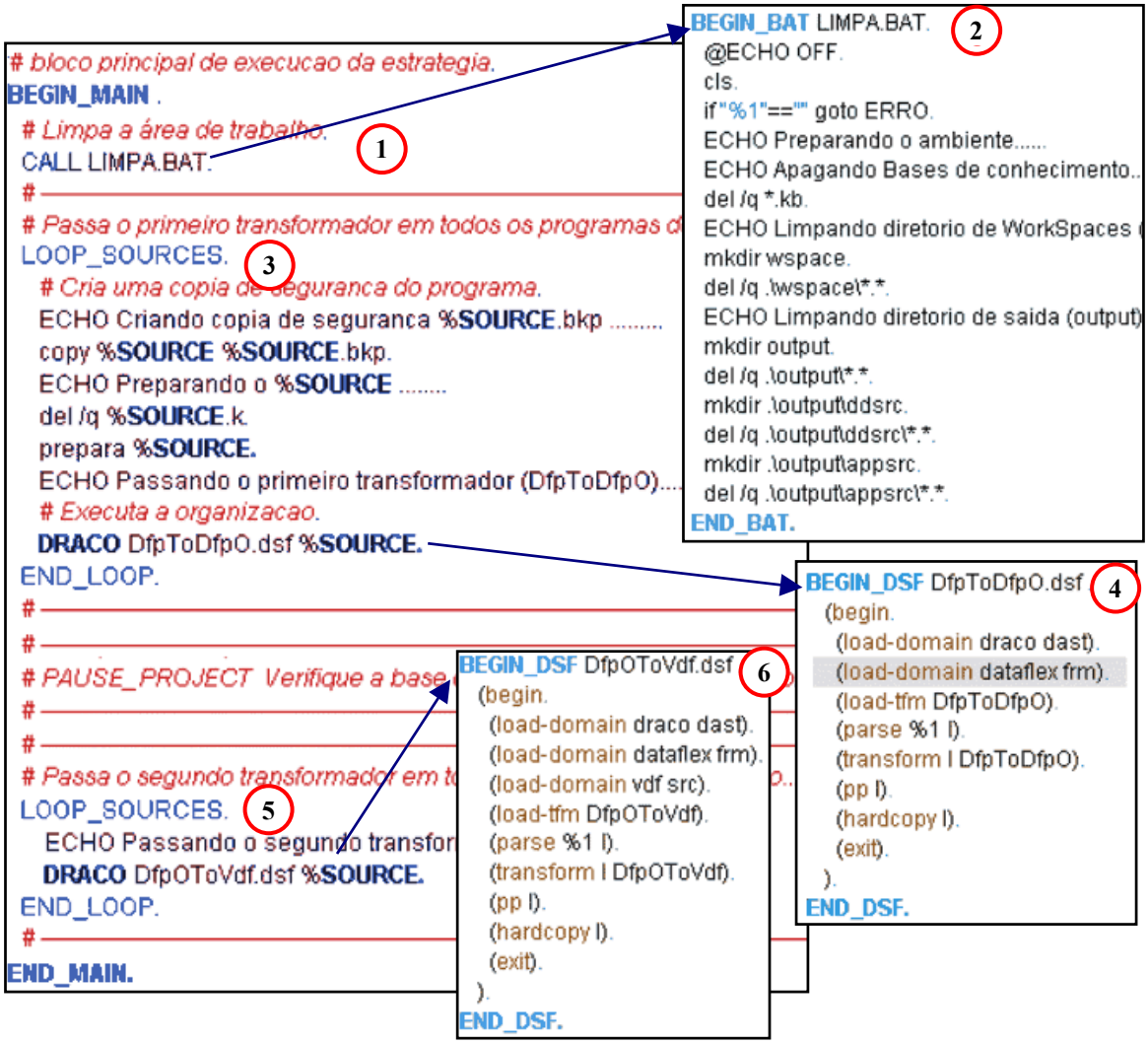


Figura 46. Script de execução dos transformadores no DDE

Construídos os transformadores e seu script de execução, é definida uma estratégia para uso da Transformação de DFP para VDFOO reusando um Framework, apresentada no próximo capítulo.

Capitulo 5 Uso da Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um *Framework*

O uso da Transformação de DataFlex Procedural (DFP) para Visual DataFlex Orientado a Objetos (VDFOO), reusando o DataFlex Application *Framework* (DAF), é realizada em quatro passos: **Preparar *script* de transformação no DDE**, quando usando o DDE, são relacionados os códigos legados a serem transformados e o script de execução das transformações que será usado; **Executar o script de transformação**, quando o DDE aciona o Draco-PUC, submetendo o conjunto de códigos para os transformadores, através do script de execução das transformações, gerando o código VDFOO e as informações para as ferramentas do VDFOO; **Registrar o código gerado nas ferramentas do VDFOO**, quando o Engenheiro de Software (ES), registra a relação componentes nas ferramentas de desenvolvimento do VDFOO, e **Compilar e executar o código gerado**, quando o código VDFOO gerado, é pré-compilado e executado através das ferramentas de desenvolvimento, conforme apresentado na Figura 47.

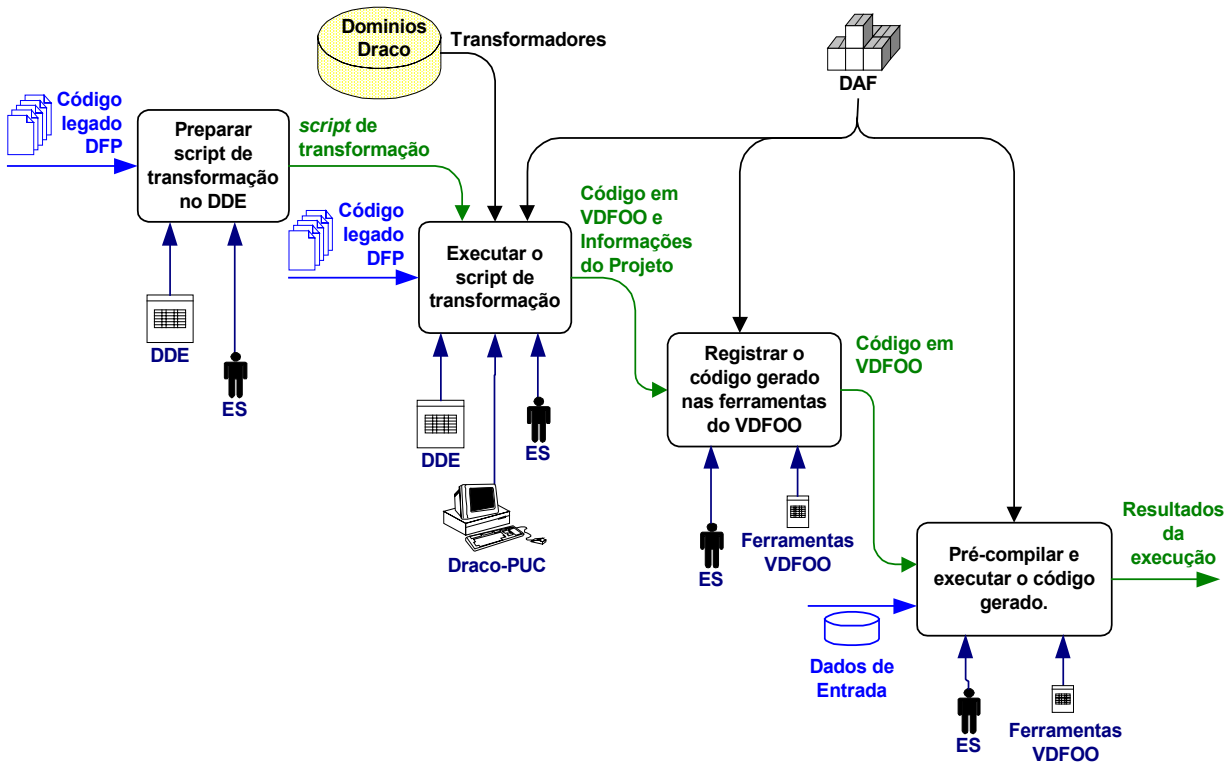


Figura 47. Estratégia de Uso da Transformação de DFP para VDFOO reusando um Framework

Para mostrar o uso da estratégia são apresentadas as transformações de dois sistemas: **Controle de Mandados**, com cerca de 40 mil linhas de código legado; e o **Sistema Integrado para Revendas e Concessionárias (SIRC-X)**, com cerca de 5.3 milhões de linhas de código.

5.1 Estudo de Caso I - Transformação do Controle de Mandados

O sistema de Controle de Mandados Judiciais para Oficiais de Justiça, está em DFP, possuindo cerca de 40.000 linhas de código. Foi construído em 1994, sendo utilizado por diversos oficiais de justiça na região de Londrina – PR, para controlar a carga de mandados do oficial, prazos de devolução, e os documentos do mandado. No sistema, o ator Oficial de Justiça pode acompanhar a situação dos mandados, acessar sua documentação, identificar sua localização física, e controlar sua agenda.

Segue a apresentação dos passos utilizados na transformação.

5.1.1 Preparar *Script* de Transformação no DDE

Para a preparação do script de transformação do Sistema de Controle de Mandados, foi criado um único script de execução, relacionando os códigos legados do sistema, como mostra a Figura 48. Em (1) é apresentada a relação dos códigos legados que compõe o sistema apresentado. Em (2) a chamada do *script* de execução das transformações criada no DDE.

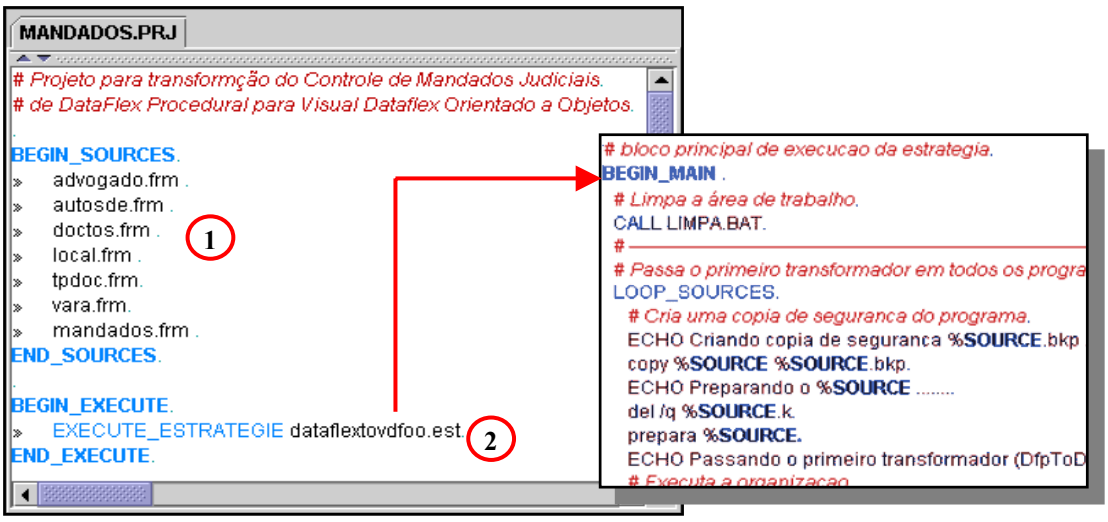


Figura 48. Script de Transformação para o Controle de Mandados "mandados.prj"

Não é necessário seguir uma ordem pré-definida para relacionar códigos legados no *script* de transformação, mas no caso do Controle de Mandados, foram relacionados primeiro os programas auxiliares e no final o programa principal "mandados.frm".

5.1.2 Executar o *Script* de Transformação

Para executar o script de transformação, basta clicar no botão de execução do *script*, para que o DDE acione o Draco-PUC, para a realização das transformações, seguindo as definições do *script* de execução das transformações, como mostra a Figura 49.

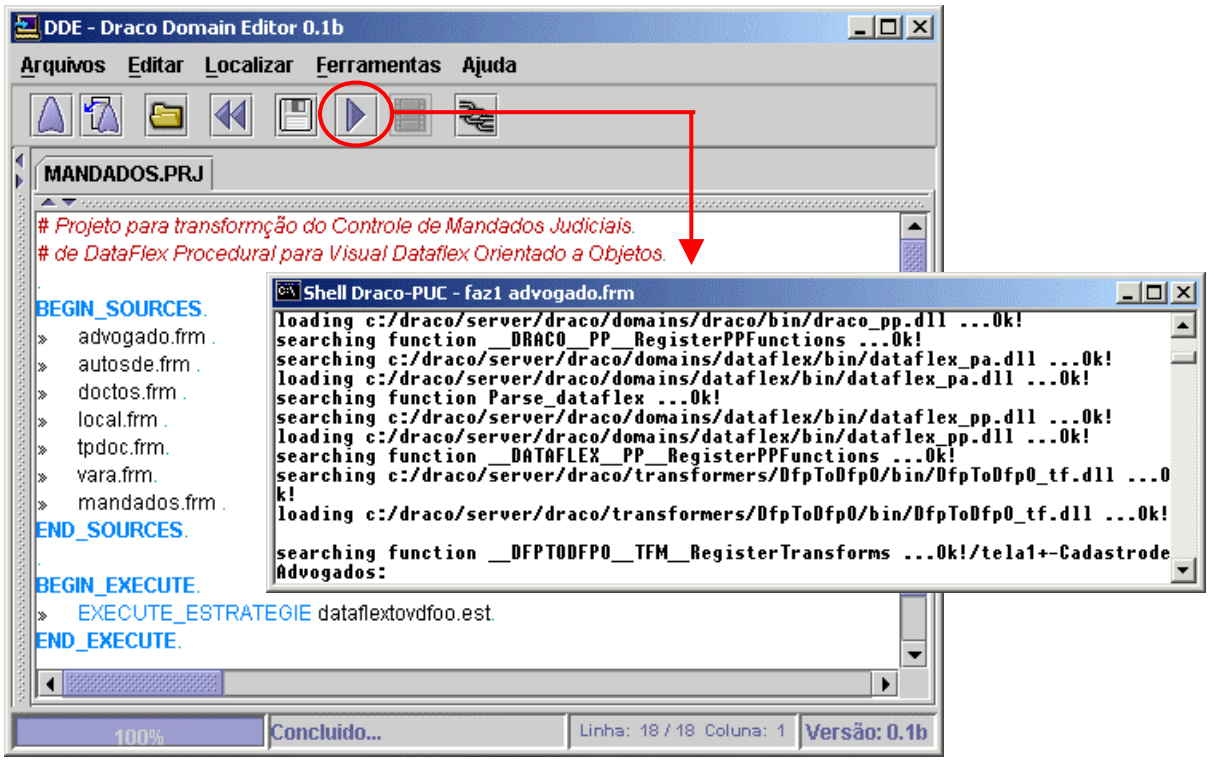


Figura 49. Execução do script de transformação para o Controle de Mandados

Ao acionar o botão de execução do script de transformação, o DDE interpreta as instruções do script de execução das transformações, gerando um script para execução do Draco-PUC, fornecendo os programas relacionados para transformação.

A execução da transformação do Controle de Mandados levou cerca de duas horas, conforme mostra a Figura 50.

Código Legado	Tempo do 1º Transformador	Tempo do 2º Transformador
advogado.frm	0:15:16	0:03:01
Autosde.frm	0:10:23	0:02:05
doctos.frm	0:16:14	0:05:23
local.frm	0:08:46	0:02:27
tpdoc.frm	0:12:30	0:03:00
vara.frm	0:09:35	0:02:01
mandados.frm	0:22:03	0:10:02
TOTAL	1:34:47	0:27:59
TEMPO TOTAL DA TRANSFORMAÇÃO		**2:02:46**

Figura 50. Tempos de transformação do Controle de Mandados

Como resultado da execução da transformação, foram gerados diversos códigos em VDFOO, e relacionados através do arquivo "transf.txt", que contém a relação de códigos gerados para possibilitar o registro destes códigos nas ferramentas do VDFOO.

5.1.3 Registrar o código gerado nas ferramentas do VDFOO

Com base na lista de códigos gerada pelo transformador, os códigos são registrados na ferramenta IDE do VDFOO, através da opção "Register External Component", como mostra a Figura 51. Na tela de registro, pode ser usado o botão "..." para localizar o arquivo de código. Ao carregar o arquivo de código, as informações sobre a descrição e nome do objeto são sugeridas e podem ser alteradas se necessário.

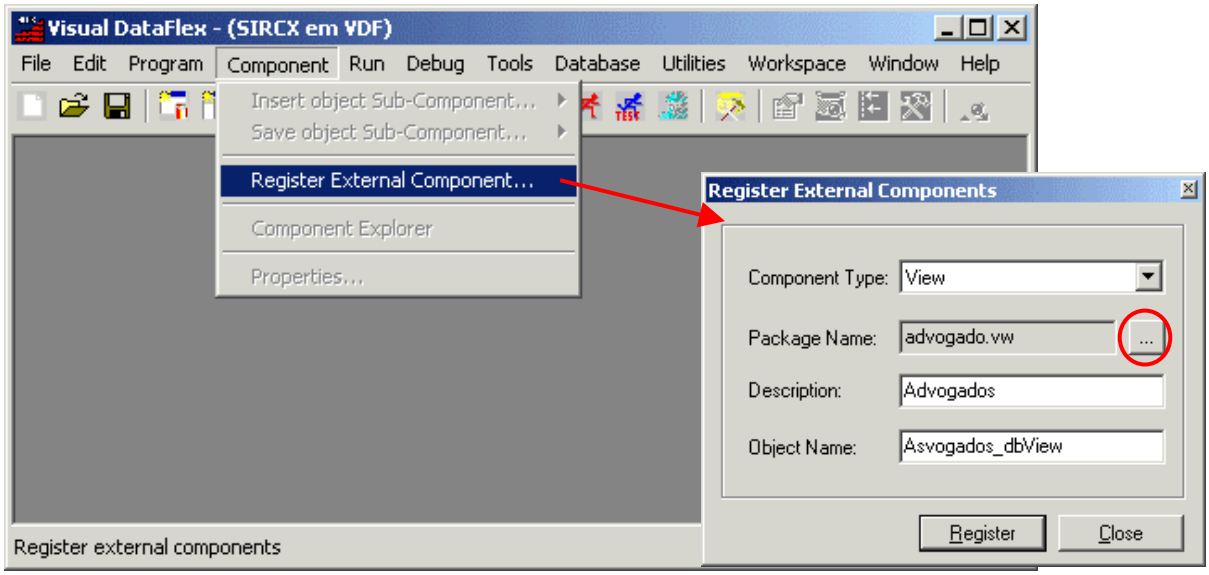


Figura 51. Registro de código na Ferramenta IDE do VDFOO.

Este registro deve ser feito para todos os códigos gerados.

5.1.4 Pré-compilar e executar o código gerado.

Depois de registrados os códigos, o sistema é pré-compilado para possibilitar a execução através do *runtime* VDFOO. A execução serve também para verificar as funcionalidades do sistema, permitindo que o sistema legado e o gerado sejam executados utilizando a mesma base de dados. Isso facilita a avaliação das funcionalidades, comparando os resultados dos tratamentos realizados em cada caso de uso do sistema. O sistema gerado pode então sofrer manutenções para adaptações ou inclusões de novos requisitos. A Figura 52 apresenta acima a tela principal do Controle de Mandados em DFP e abaixo a sua correspondente em VDFOO após a reimplementação.

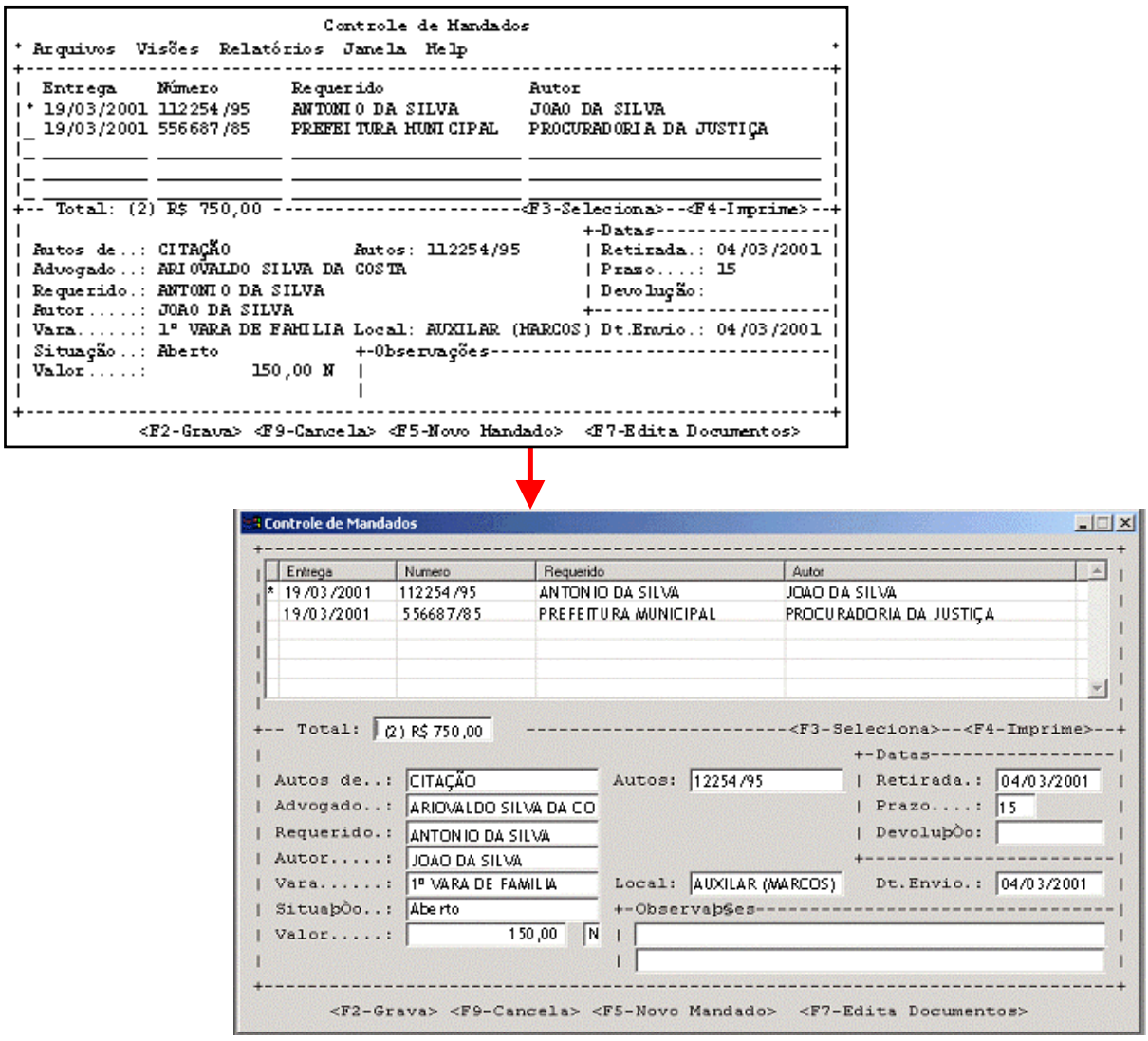


Figura 52. Execução do Controle de Mandados em VDFOO

As interfaces geradas mantêm a mesma formatação do código legado, para evitar que informações dependentes desta formação sejam perdidas na transformação.

Depois de transformado, o sistema gerado pode sofre manutenções como mostra a Figura 53, na qual foram realizadas algumas alterações na formatação da tela principal do Controle de Mandados, utilizando alguns recursos disponíveis no VDFOO, como: Troca das molduras textuais, criando a aparência 3D (1). Troca do "*" por um ícone (2). Alteração na aparência da *Grid* retirando as linhas (3). Substituição dos textos que indicavam acionamento de funções por botões (4). Adequação no tamanho, formato e posição de alguns campos (5). Não foram realizadas inclusões de métodos. Os botões inseridos acionam os métodos criados na transformação e executados através de teclas de função.

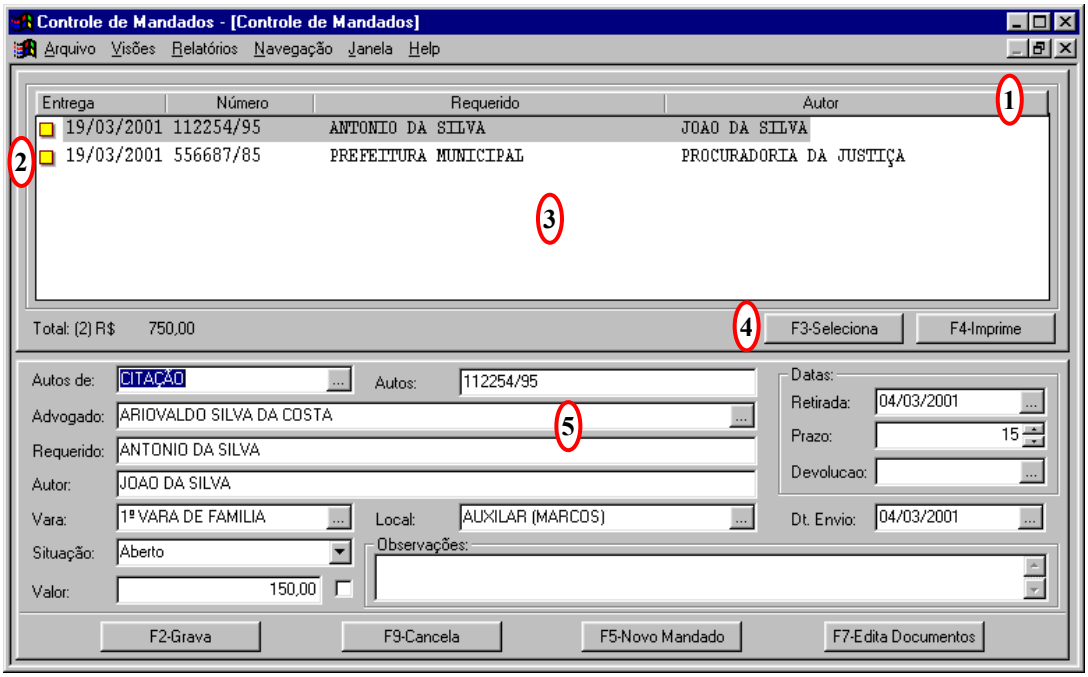


Figura 53. Execução do Controle de Mandados após alterações

O sistema transformado foi implantado em cinco dos usuários do sistema legado. Foi realizado um acompanhamento durante um período de dois meses. No período de acompanhamento, foram analisadas as solicitações de suporte dos usuários, e comparadas com o histórico de chamadas. Este levantamento possibilitou identificar que a maioria das solicitações de suporte destinavam-se à retirada de dúvidas sobre a operação do sistema em ambiente Windows ou a configuração do Windows conforme os requisitos do *runtime* VDFOO. Foram realizadas algumas adaptações nos relatórios devido a diferença de tratamento de impressoras entre o DFP e o VDFOO. A produtividade para realização de alterações aumentou no ambiente de desenvolvimento VDFOO.

5.2 Estudo de Caso II - Transformação do SIRC-X

O Sistema Integrado para Revendas e Concessionárias (SIRC-X) está em DFP, possuindo cerca 1.300 programas gerais e 600 programas específicos para atender a requisições de clientes ou fábricas, totalizando aproximadamente 5.3 milhões de linhas de código. O sistema é de propriedade da CONSYSTEM – Consultoria e Sistemas S/C Ltda, está em operação desde 1992, e tem 180 cópias instaladas em revendas e concessionárias de veículos de médio e grande porte, em vários estados brasileiros. O SIRC-X possui módulos para controle de **Contas a Pagar e Receber**, **Caixa e Bancos**, **Contabilidade**, **Escrituração Fiscal**, **Faturamento**, **Gestão de Peças**, **Gestão de Veículos**, **Gestão de Oficina**, **Controle de Garantia**, **Comunicação entre Filiais**, **Comunicação com Fábricas** (Case, Fiat, Ford, GM, Hyster, VW, Renault, Peugeot, Kia, MF, Subaru, Mitsubishi, Audi, Ásia Motors e Volvo), **Controle Gerencial** e um módulo de **Controle de Acesso ao Sistema**.

A CONSYSTEM apresenta informações detalhadas sobre o SIRC-X em seu *site*: "www.consystem.com.br". Apesar do enfoque do *site* ser comercial, podem ser identificadas as funcionalidades principais de cada módulo do sistema. Também estão disponíveis informações sobre o resultado das transformações de alguns módulos, que por questões comerciais foram caracterizados como "Módulos Visuais". A empresa decidiu tratar os módulos transformados como adicionais ao produto, dando ao cliente a possibilidade de optar por alguns dos módulos em DFP e outros em VDFOO. Isso é possível pois na transformação, a mesma base de dados do sistema legado é usada pelo sistema gerado.

Com a transformação, a empresa pretende expandir sua área de atuação para revendas de pequeno porte. O principal empecilho na comercialização do SIRC-X para revendas de pequeno porte é o alto custo do *runtime* DFP. Tendo em vista esta perspectiva, a CONSYSTEM alocou quatro pessoas da equipe de desenvolvimento para acompanhar a transformação.

Foi realizado um treinamento junto à equipe de desenvolvimento, que aprendeu a usar o DDE e o Draco-PUC. Seguindo a estratégia de uso apresentada, o SIRC-X foi transformado, dividido em módulos. Isso mostrou a viabilidade de utilização do Draco-PUC, com o auxílio dos scripts de transformação do DDE, por pessoas que não participaram do desenvolvimento das gramáticas e transformadores.

O primeiro módulo a ser transformado foi o **Controle de Acesso ao Sistema**, realizado com o acompanhamento de toda a equipe. Este módulo foi transformado primeiro, pois serviu de base para o uso dos demais módulos. A participação de toda a equipe serviu como reforço do treinamento. Para transformação dos demais módulos a equipe foi subdividida em duas. Alguns módulos, depois de transformados sofreram manutenções relacionadas ao formato das telas de interação e inclusão de novos requisitos.

5.2.1 Preparar script de transformação no DDE

Foram preparados 28 *scripts* para a transformação do SIRC-X, como mostra a Figura 54. Em (1) é apresentada a relação dos scripts criados no DDE, e em (2) é apresentado um trecho do *script* "Controle_de_Accesso.prj", que foi o primeiro *script* executado com o acompanhamento de toda a equipe.

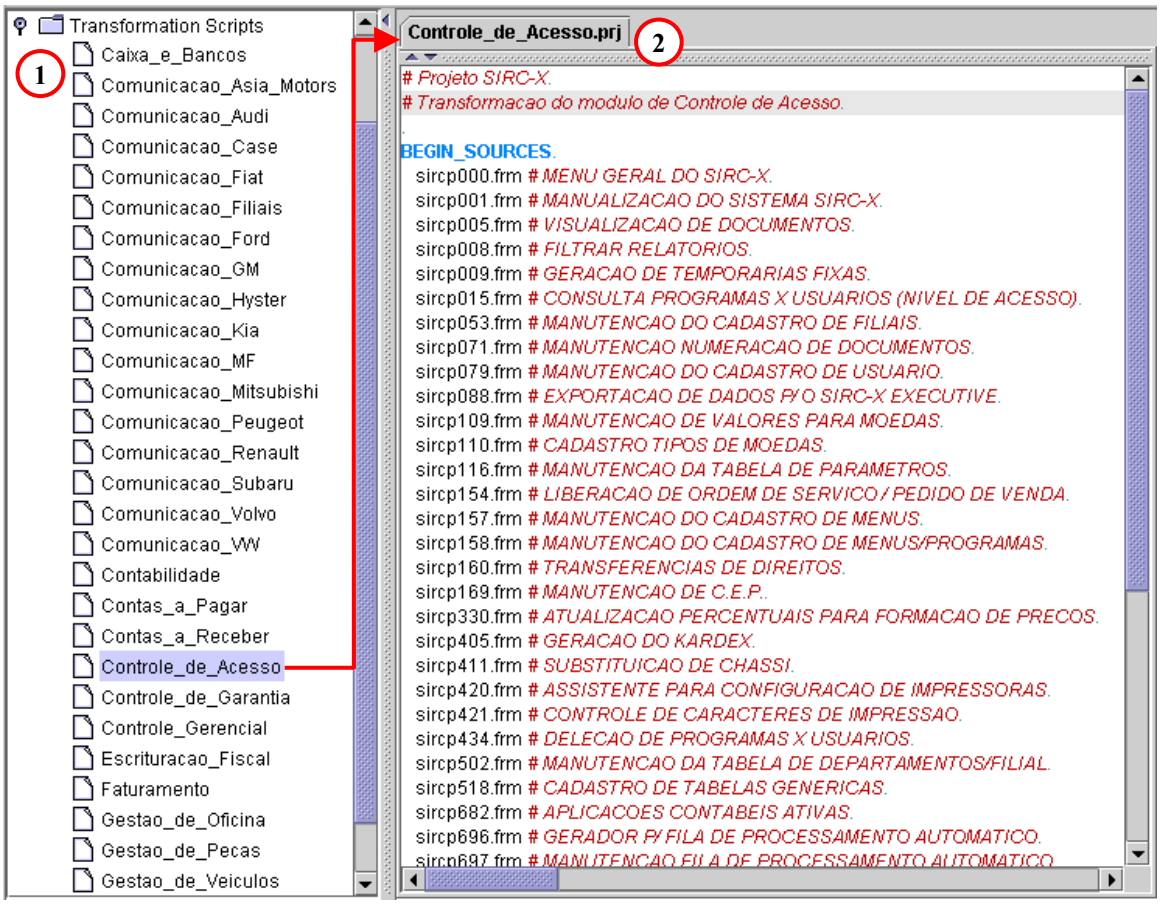


Figura 54. Scripts para a transformação do SIRC-X

A grande quantidade de código legado a ser transformado exigiu alguns cuidados na preparação dos *scripts*, como por exemplo, identificar a função de cada programa no *script* para facilitar o controle e divisão dos programas. A identificação e separação dos programas e a criação dos *scripts* que foram realizadas manualmente e levaram 40 horas.

5.2.2 Executar o script de transformação

As execuções dos *scripts* de transformação foram realizadas em várias etapas. A cada *script* executado, as etapas seguintes eram cumpridas, antes que o próximo *script* fosse executado. No total o Draco-PUC levou 406 horas, para executar todos os scripts de transformação do sistema. Este tempo de processamento está considerando apenas o tempo de execução dos *scripts*, sem considerar o tempo gasto nas etapas seguintes.

Foram geradas ao todo 296 classes na camada de banco de dados, 3.598 classes na camada de regras de negócio e 7.116 classes e objetos na camada de interface, totalizando 11.010 códigos gerados em VDFOO.

5.2.3 Registrar o código gerado nas ferramentas do VDFOO

De forma semelhante à execução dos scripts, os registros dos códigos VDFOO foram feitos a cada módulo transformado, com base nas listas de códigos geradas pelo segundo transformador.

Como mostra a Figura 55, a equipe desenvolveu um programa auxiliar em VDFOO, que partindo da lista de códigos gerada pelo segundo transformador, registra-os nas ferramentas do VDFOO. Esta ferramenta foi construída para tratar a nomenclatura gerada para programas do SIRC-X, mas pode ser adaptada no caso da transformação de outros sistemas de grande porte.

Apesar de simples, a ferramenta economizou cerca de 60 horas de trabalho no registro dos códigos, considerando que é possível registrar cerca de 3 códigos a cada minuto de trabalho nas ferramentas do VDFOO.

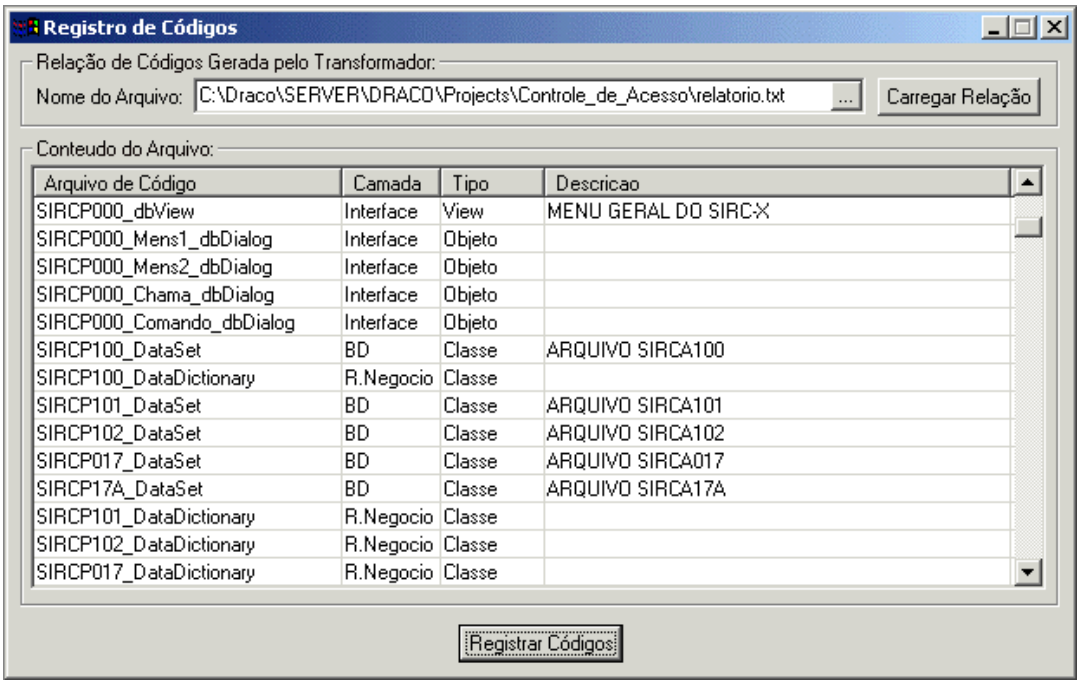


Figura 55. Ferramenta para registro dos códigos gerados em VDFOO

5.2.4 Pré-compilar e executar o código gerado.

A pré-compilação dos códigos gerados possibilita sua execução através do *runtime* VDFOO. A execução serve para verificar as funcionalidades do sistema pela execução paralela dos sistemas legado e o gerado, usando a mesma base de dados. Isso facilita a avaliação das funcionalidades, comparando os resultados dos tratamentos realizados em cada caso de uso do sistema.

A Figura 56 mostra um resumo dos tempos gastos para a transformação. A primeira coluna contém a relação de scripts. A segunda contém a quantidade de programas do *script*. A terceira contém soma das linhas dos programas. A quarta contém o total de horas gastas para a preparação do *script*. A quinta e sexta contém a soma do tempo de processamento do primeiro e segundo transformadores. A última contém o tempo gasto em dias para realizar a verificação e manutenção dos programas gerados.

Como as equipes trabalham em paralelo para a transformação, o projeto foi realizado em 4 meses, com uma equipe de quatro pessoas.

***Script* de Transformação**	**Prog.**	**Linhas** *(Mil)*	**Preparação** *(horas)*	**1º Transf.** *(horas)*	**2º Transf.** *(horas)*	**Finalização** *(dias)*
Controle de Acesso	38	95.8	1:00	04:50:02	02:25:01	2
Gestão de Veículos	205	774.9	5:00	15:06:59	19:33:30	20
Gestão de Pecas	194	611.1	4:00	06:50:53	15:25:26	15
Controle de Garantia	58	146.2	1:30	07:22:41	03:41:21	3
Gestão de Oficina	193	729.5	5:00	12:49:36	18:24:48	20
Contas a Pagar	96	302.4	2:00	15:15:54	07:37:57	8
Contas e Receber	105	330.8	2:10	16:41:46	08:20:53	10
Caixa e Bancos	134	506.5	3:40	01:34:08	12:47:04	12
Contabilidade	95	359.1	2:40	18:07:38	09:03:49	8
Escrituracão Fiscal	47	118.4	1:20	05:58:44	02:59:22	2
Comunicacão entre Filiais	22	41.6	0:30	02:05:56	01:02:58	1
Faturamento	47	177.7	1:10	08:58:05	04:29:03	6
Controle Gerencial	98	432.2	2:30	21:48:58	10:54:29	10
Comunicacão Case	15	37.8	0:25	01:54:29	00:57:15	1
Comunicacão Fiat	8	20.2	0:15	01:01:04	00:30:32	1
Comunicacão Ford	35	88.2	0:50	04:27:08	02:13:34	2
Comunicacão GM	22	55.4	0:30	02:47:55	01:23:57	1
Comunicacão Hyster	19	47.9	0:30	02:25:01	01:12:31	1
Comunicacão VW	34	85.7	0:50	04:19:30	02:09:45	1
Comunicacão Renault	24	60.5	0:40	03:03:11	01:31:35	1
Comunicacão Peugeot	18	45.4	0:30	02:17:23	01:08:42	1
Comunicacão Kia	15	37.8	0:25	01:54:29	00:57:15	1
Comunicacão MF	20	50.4	0:30	02:32:39	01:16:19	1
Comunicacão Subaru	37	93.2	1:00	04:42:24	02:21:12	2
Comunicacão Mitsubishi	7	17.6	0:10	00:53:26	00:26:43	1
Comunicacão Audi	22	55.4	0:30	02:47:55	01:23:57	1
Comunicacão Ásia Motors	9	22.7	0:15	01:08:42	00:34:21	1
Comunicacão Volvo	7	17.6	0:10	00:53:26	00:26:43	1
TOTAL	**1.624**	**5.361,9**	**40hs**	**270hs e 40mm**	**135hs e 20mm**	**134 dias**

Figura 56. Tempos de transformação do SIRC-X

São apresentadas a seguir alguns dos resultados da execução do código legado DFP e do código gerado em VDFOO usando a mesma base de dados.

A Figura 57 mostra a cima a tela controle de acesso ao sistema em DFP (1), e a baixo a sua correspondente em VDFOO gerada pelos transformadores (2) e a tela após algumas adaptações de formatação(3). As funções da tela em VDFOO após a manutenção são as geradas pelo transformador. Objetos foram apenas movimentados para pontos diferentes na tela. Foram, inseridos objetos para agrupamento das informações e para apresentação a imagem. Os botões “OK” e “Cancela” acionam as operações da tecla “Enter” e “Escape”.

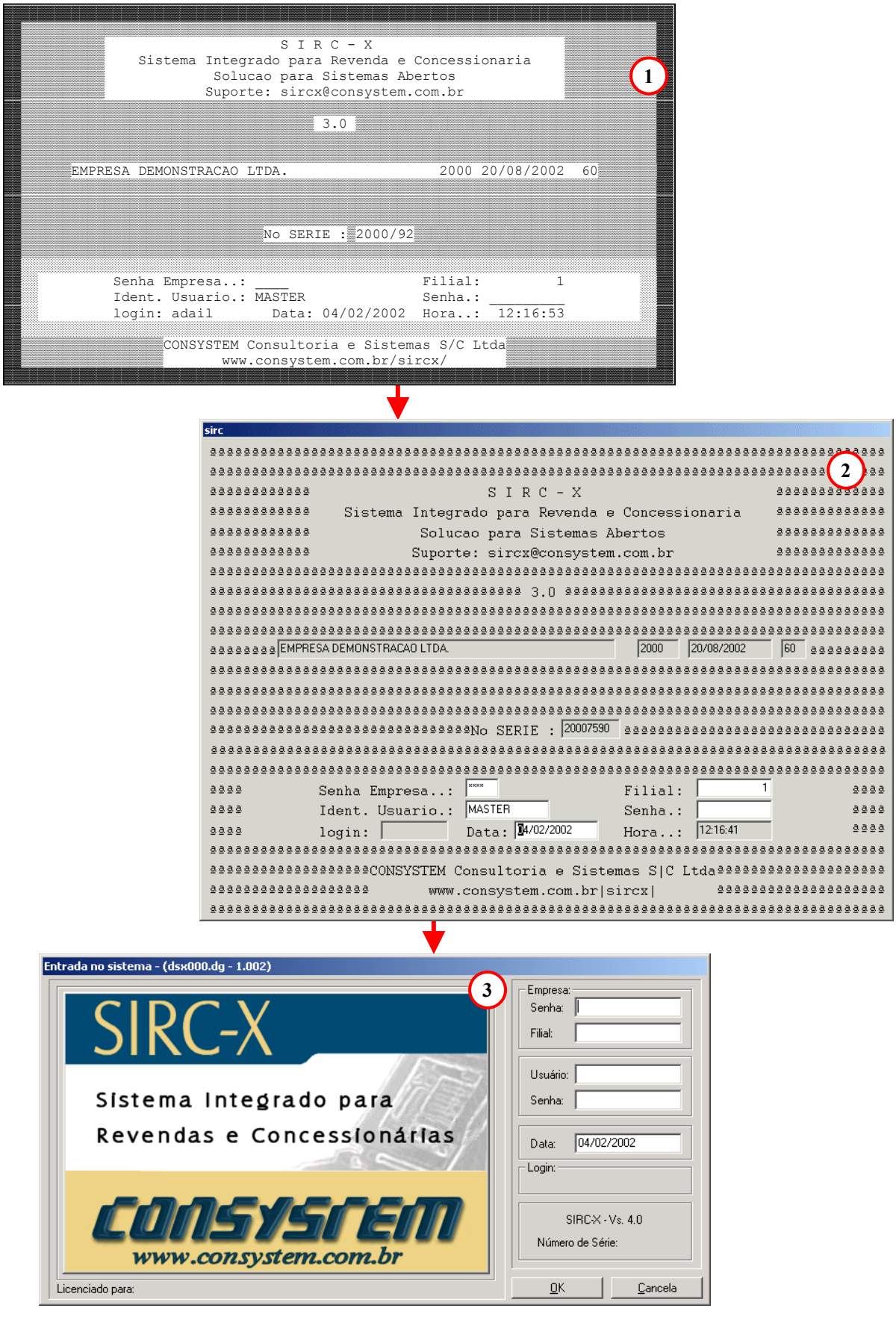


Figura 57. Execução do SIRC-X "Controle de Acesso" em VDFOO

A Figura 58 mostra a tela do Menu Principal do SIRC-X, executando a função de chamada rápida de programas, acionada pela tecla "F8". A baixo na figura, é apresentada a tela gerada em VDFOO, executando a mesma função através do "F8".

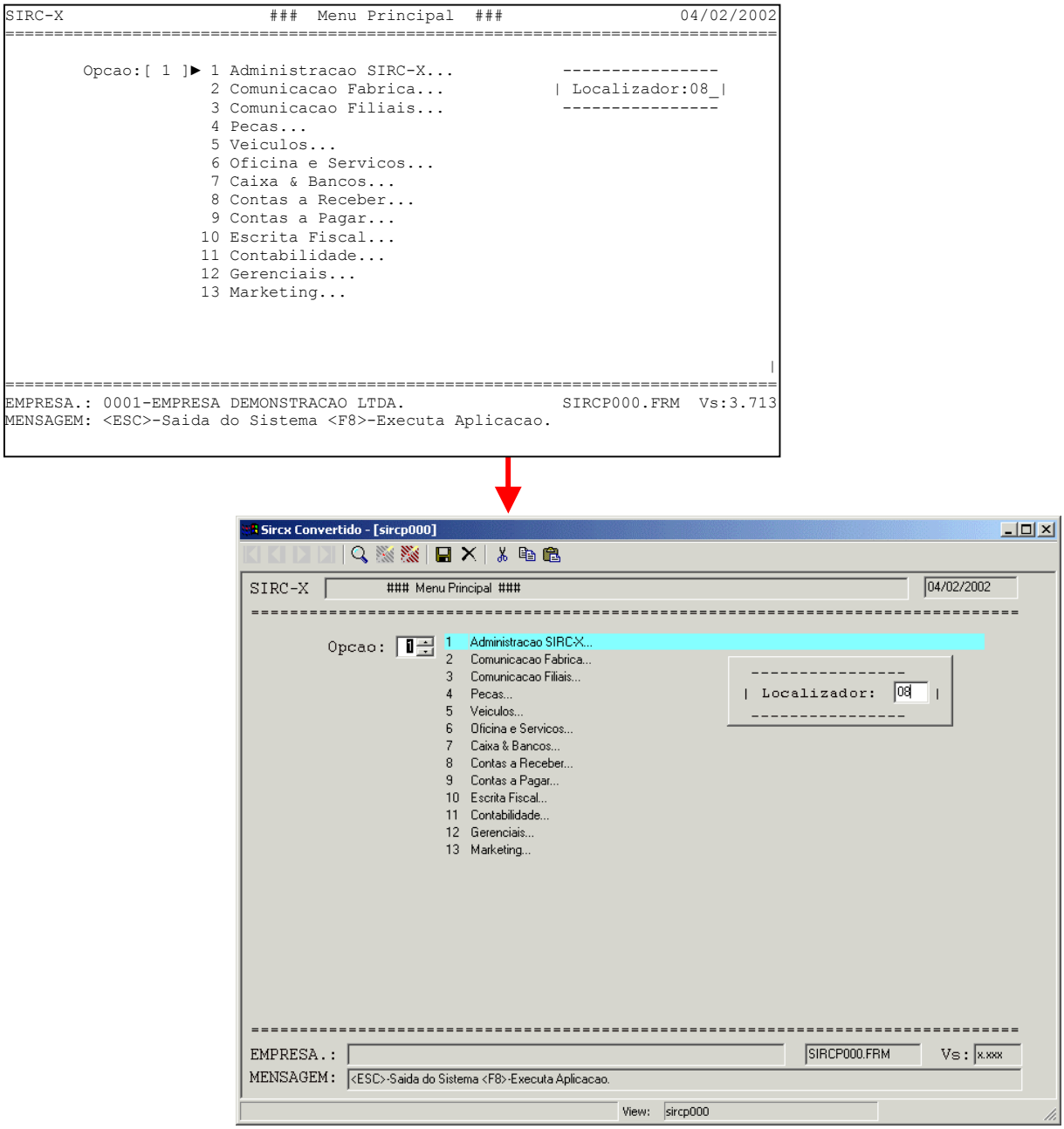


Figura 58. Execução do SIRC-X "Menu Principal" em VDFOO

A Figura 59 mostra outra tela do SIRC-X, usada pelo Gestão de Peças para manutenção dos produtos e pela Gestão de Veículos para manutenção de modelos. A parte superior da figura mostra a tela em DFP, e a tela auxiliar acionada ao pressionar "F5" no código do produto. A parte inferior da figura mostra a tela gerada em VDFOO, e a tela auxiliar acionada pela tecla "F5" no campo de edição do código do produto.

As telas auxiliares destinam-se à criação de uma pesquisa de produtos cadastrados, possibilitando a seleção de um produto para a manutenção, tanto em DFP quanto em VDFOO. As classes do DAF inserem algumas funcionalidades adicionais nas telas de pesquisa, como a facilidade de navegação e a procura posicional automática. Porém, estas inclusões de funcionalidades não afetam o objetivo da tela de consulta.

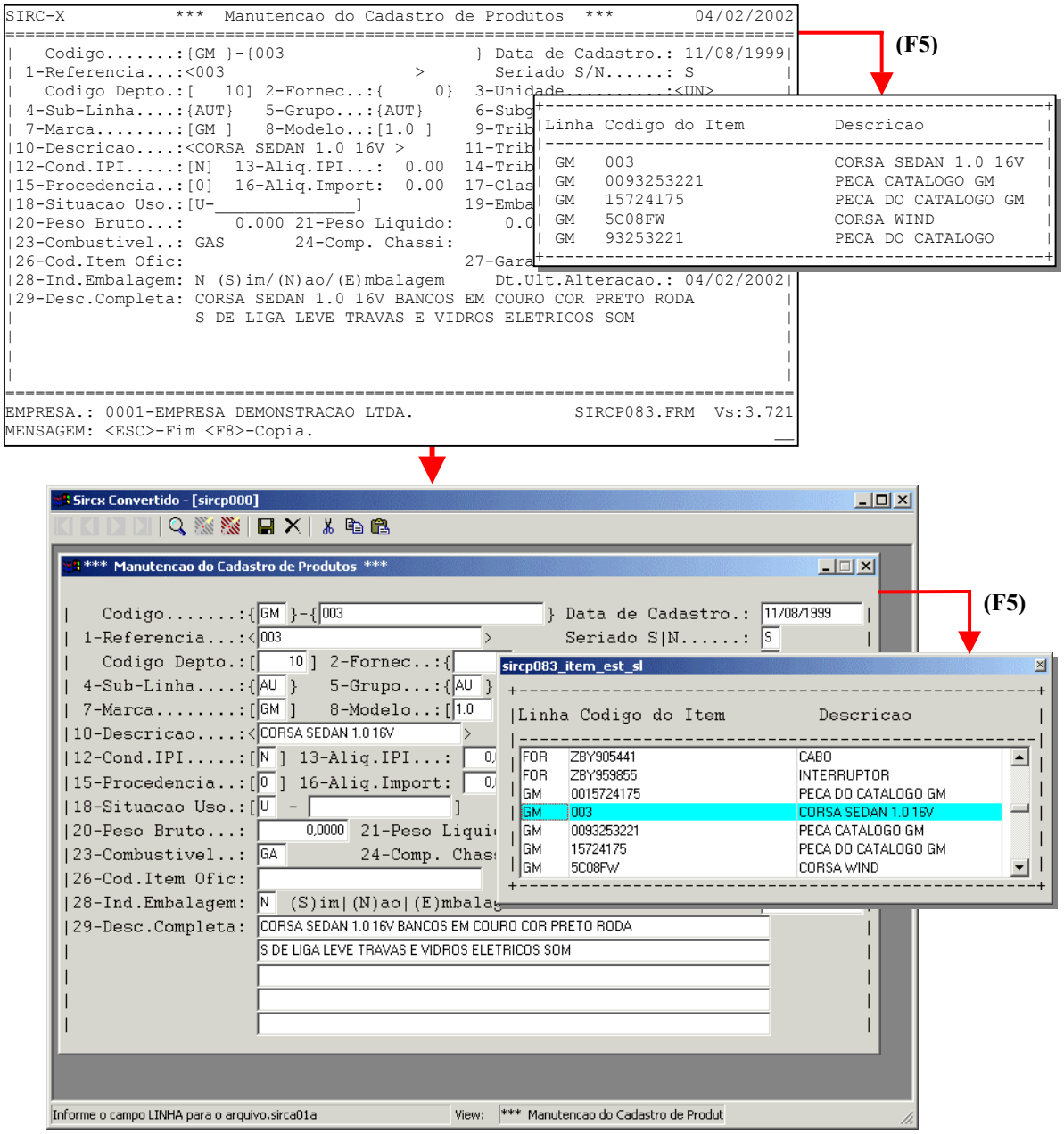


Figura 59. Execução do SIRC-X "Manutenção de Produtos" em VDFOO

Após a transformação do SIRC-X, alguns módulos gerados em VDFOO foram trabalhados para inclusão de novos requisitos, e disponibilizados como adicionais ao produto.

Os módulos de Veículos e Oficina estão funcionando em conjunto com o SIRC-X em 10 clientes. A intenção é realizar a liberação do SIRC-X completo em VDFOO assim que uma estratégia de *marketing* for definida.

A CONSYSTEM mostrou-se satisfeita com os resultados da transformação do software. A empresa já passou por um processo de reengenharia manual, para o porte do sistema SIRC/400, que foi escrito em PL/SQL, rodando em plataforma IBM AS/400, para obter a primeira versão do SIRC-X em DFP, rodando em UNIX. Esta reengenharia levou um ano e meio para ser realizada, com uma equipe de 20 pessoas. A experiência de reengenharia, do SIRC/400 para o SIRC-X, foi feita de forma manual, assistida por algumas ferramentas produzidas para acompanhar o processo. Não foi realizada mudança de paradigma, e o SIRC/400 possuia cerca de 60% das funcionalidades atuais do SIRC-X.

Em uma estimativa calculada, com base nas experiências anteriores de reengenharia da empresa e no acompanhamento das transformações no Draco-PUC, chegou-se a possibilidade de redução de custos de até 90% com a estratégia. O processo realizado em quatro meses com uma equipe de quatro pessoas gastaria um ano e meio com uma equipe de quinze pessoas.

O capitulo seguinte apresenta as conclusões deste trabalho, resumindo suas contribuições e sugestões para trabalhos futuros.

Capitulo 6 Conclusões

Este trabalho apresentou uma estratégia para Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos Reusando um *Framework* em três camadas.

Foram apresentados os três passos para transformação de um sistema legado: Organizar Código Legado, Reimplementar Código Organizado e Executar Código VDFOO. No primeiro passo tem-se a organização do código legado em DataFlex Procedural, através da transformação. No segundo passo tem-se a reimplementação através de transformação do código DFP Organizado para Visual DataFlex Orientado a Objetos. No terceiro passo executa-se o código Visual DataFlex Orientado a Objetos na plataforma com sistema operacional Windows 95/98/NT.

Depois de transformado o sistema pode evoluir e utilizar os novos recursos disponíveis na linguagem Visual DataFlex Orientada a Objetos, como o tratamento de multimídia, e construção de uma interface mais amigável com o usuário.

A utilização do DataFlex Application *Framework* com arquitetura em três camadas, interface, regras de negócio e banco de dados, sustenta a evolução do sistema dando maior flexibilidade para alterar os componentes em cada camada.

Foram transformados sistemas legados, onde a estratégia apresentada permitiu a redução de até 90% no tempo necessário para transformação dos sistemas, se comparado com o tempo de transformação manual.

Em um projeto como o SIRC-X, com 5,3 milhões de linhas de código, a redução do tempo de transformação de um ano e meio, com uma equipe de 15 pessoas, para 4 meses, com uma equipe de 4 pessoas representa uma economia de 500 mil reais, só em custo direto com mão de obra.

Existe o interesse de uso dessa estratégia por parte de empresas que possuem códigos legados em DataFlex Procedural, como é o caso da empresa a qual o autor deste

projeto mantém um vínculo de pesquisa. Esta empresa possui sistemas comerciais integrados em DataFlex Procedural e tem o interesse de transformá-los em Visual DataFlex Orientado a Objetos.

6.1 Principais Contribuições

A principal contribuição desta pesquisa vem da definição de uma estratégia sistemática para reengenharia de códigos legados em DataFlex Procedural para códigos em Visual DataFlex Orientado a Objetos com o reuso do DataFlex Application *Framework*. Esta reengenharia possibilita o reuso do código legado em novas plataformas de hardware e software, economizando tempo e recursos.

A construção do editor de domínios para o Draco-PUC, denominado DDE - Draco Domain Editor, facilitou a criação dos domínios e o uso da estratégia de transformação por uma equipe de trabalho. Desta forma o DDE mostrou-se uma importante contribuição, pois aumentou a produtifididade na implementação da estratégia, e facilitou seu uso, possibilitando uma maior organização na transformação de sistemas de grande porte.

Outras contribuições deste projeto vêm da criação dos domínios DataFlex Procedural e Visual DataFlex Orientado a Objetos, com a elaboração de suas gramáticas no Sistema Transformacional Draco-PUC. A elaboração dos Transformadores DFP para DFP Organizado e de DFP Organizado para VDFOO. Na construção destes transformadores foi dada uma grande ênfase na organização do código, onde foram utilizadas técnicas de otimização, normalmente aplicadas a compiladores, que possibilitaram elevar o grau de automação, reduzindo a necessidade de interações durante os processos de transformação. Ainda na implementação dos transformadores, foi proposta uma formula para automatizar o tratamento o encapsulamento de métodos que referenciam mais do que uma classe. Este trabalho também mostra a possibilidade de combinar a tecnologia de transformação de software com a tecnologia de *frameworks*.

Finalmente, temos a validação da estratégia de transformação em um sistema legado de grande porte, como mais de 5 milhões linhas de código. Isso mostra que a estratégia apresentada, aliada ao Sistema Transformacional Draco-PUC, e o DDE, são fortes aliados para redução de custos para transformação de sistema de grande porte, proporcionando um

resultado confiável em um prazo bem menor, se comparado ao desenvolvimento do mesmo trabalho sem o auxilio de Sistemas Transformacionais.

6.2 Desmembramentos

Os desmembramentos deste trabalho vêm com a realização da estratégia em outras linguagens que tenham as mesmas características, ou a criação de um *framework* equivalente ao DataFlex Application Framework em outras linguagens, facilitando a construção e transformação de sistemas segundo o padrão de desenvolvimento em três camadas. Estes novos trabalhos ficam facilitados, dada a capacidade do Sistema Transformacional Draco-PUC de trabalhar com múltiplos domínios.

O uso de técnicas de otimização de códigos na implementação de transformadores pode ser mais explorada, proporcionando maior grau de automação e gerando resultados mais estruturados. Estas técnicas foram aplicadas neste trabalho, objetivando maior eficiência na identificação dos métodos e classes, mas podem ser aplicadas também para reduzir a dependência entre classes, diminuindo a quantidade de conexões de mensagens, para o recolhecimento e geração de níveis mais apurados de herança, substituição da assinatira dos métodos através da análise de seu conteúdo, contribuindo para melhorar o polimorfismo e o encapsulamento no sistema.

A formula apresentada para o tratamento do encapsulamento dos métodos que referenciam mais do que uma classe, também chamados de métodos com anomalias [Pen96], pode ser mais bem depurada para tornar-se genérica, aplicável a outras linguagens e paradigmas.

Também pode ser explorada a evolução da ferramenta DDE, abrangendo funcionalidades mais avançadas, como funções de depuração, acompanhamento e testes das transformações.

Referências Bibliográficas

[Abr99a] ABRAHÃO, S.M., PRADO, A.F. **Web-Enabling Legacy Systems Through Software Transformations**. *IEEE International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems.* In Proceedings, pp, 149-152. Santa Clara – USA. April, 08-09, 1999.

[Abr99b] ABRAHÃO, S.M., PRADO, A.F; SANT'ANNA, M. – **A Semi-Automatic Approach for Building Web-Enabled Aplications from Legacy** – Submitted on 4 IEEE international softwaer engineering Standards Symposium – Curitiba, Brasil, May 1999.

[Aho95] AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compiladores - Princípios, Técnicas e Ferramentas**, (Compilers Principles, Techniques, and Tools), Ed. Guanabara Koogan S.A., 1995, Tradução de: Daniel de Ariosto Pinto.

[Ama01] AMARAL, J. N., BARTON, C. , MACDONELL, A. C., and Mcnaughton, M., **"Using the SGI Pro64 Open Source Compiler Infra-Structure for Teaching and Research**," 13th Symposium on Computer Architecture and High Performance Computing, Pirenopolis, GO, Brazil, September, 2001.

[Ant95] ANTLR: **A predicated- LL(k) *parser* generator** - *Software--Practice and Experience*, 25(7):789-810, 1995.

[Arm98] ARMSTRONG, M. N.; TRUDEAU, C. **Evaluating Architectural Extractors**. In: IEEE Working Conference on Reverse Engineering, 5., Honolulu, Hawai, EUA, outubro de 1998. *Anais.* LosAlamitos-CA, EUA, IEEE Computer Society, p. 30-39.

[Bax97] BAXTER I., PIDGEON, C.W. **Software Change Through Design Maintenance**. International Conference on Software Maintenance – ICSM'97. In Proceedings. Bari, Italy. October 1st –3rd, 1997.

[Ben92] BENEDUSI, P.; CIMITILE, A; CARLINI, U. **Reverse Engineering Processes, Design Document Production and Structure Charts**. J. Systems Software, V.19, p. 225-245, 1992.

[Ber89] BERGSTRA J.A.,HEERING J., e KLINT P.- **The algebraic specification formalism ASF**. - In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series - pg 1-66. The ACM Press in co-operation with Addison-Wesley, 1989. Cap 1.

[Ber96] **Desenvolvimento de Sistemas Orientados a Objetos Utilizando o Sistema Transformacional Draco-PUC**
Bergmann, Ulf e Prado, Antonio F., Leite, Julio C.L.S.P.
X Simpósio Brasileiro de Engenharia de Software, pg 173-188, São Carlos, SP, Out 96

[Bil89] BILLOT S. e LANG B. - **The structure of shared forests in ambiguous parsing**. - Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, Vancouver, pag 143-151, 1989.

[Bir00] Birkhauser - **Modern Software Tools for Scientific Com – (Tampr)**, http://www.birkhauser.com/cgi-win/ISBN/0-8176-3974-8, 2000

[Blo97] BLOMI J. - **Metamorphosen des Datumsfeldes** – Micro Focus GmbH, München, Germany, 1997. http://www.microfocus.de/y2000/articles/y2k-info.htm

[Boy89] BOYLE, J. **Abstract Programming and Program Transformation – An Approach to Reusing Programs, in Software Reusability.** Vol.1, pp. 361-413. Ed Ted Biggerstaff. ACM Press, 1989.

[Bra00] BRANCO, L. H. C., PRADO, A. F., SOUZA, W. L., SANT'ANNA, M. - **Automatic Implementation of Distributed Systems Formal Specifications** -International Parallel and distributed Processing Symposium - Workshop on Formal Methods for Parallel Programming. Parallel and Distributed Processing - 15 IPDPS 2000 Workshops Lecture Notes in Computer Science, pg. 1019-1026, vol. 1800. Proceedings, ISBN: 3-540-67442-X Cancun, Mexico, May 1-5, 2000.

[Bra97] BRAND M.G.J. van den, SELLINK M.P.A., e VERHOEF C. - **Control flow normalization for COBOL/CICS legacy systems**. - Technical Report P9714, University of Amsterdam, Programming Research Group, 1997. http://adam.wins.uva.nl/~x/cfn/cfn.html

[Bra97a] BRAND M.G.J. van den, KLINT P., e VERHOEF C. - **Re-engineering needs generic programming language technology**. *ACM SIGPLAN Notices*, 32(2):54-61, 1997.

[Chi90] CHIKOFSKY, Elliot J.; CROSS, James H. **Reverse Engineering and Design Recovery: a Taxonomy**. IEEE Software, p. 13-17, janeiro 1990.

[Col94] COLEMAN D. AT ALL, **Object Orientend Developement – The Fusion Method**, Pretice Hall, 1994.

[Com00] Comp.compilers: **ANSI C Portable TXL 5.2**, http://compilers.iecc.com/comparch/article/91-05-087,

[Cor91] CORDY J.R., HALPERN Hamu C.D., e PROMISLOW E. - TXL: **A rapid prototyping system for programming language dialects**. Computer Languages, 16(1):97-107, 1991.

[Cor93] CORDY, J., CARMICHAEL, I., **The TXL Programming Language Syntax and Informal Semantics.** Technical Report. Vol.7.Queen's University at Kingston – Canada. June, 1993. - http://www.queis.queensu.ca/STLab/TXL

[Dat91a] Data Access Corporation, **Enciclopédia DataFlex, Vs. 3.0**, São Paulo - SP, 1991

[Dat91b] Data Access Corporation, **Enciclopédia DataFlex OOP/UIMS**, Vs. 3.0, São Paulo - SP, 1991

[Dat91c] Data Access Corporation, **Guia do Usuário, Vs. 3.0**, São Paulo – SP, 1991

[Dat92] Data Access Corporation, **DataFlex Debugger**, Miami, Florida, USA, 1992

[Dat94] Data Access Corporation, **DataFlex Application *Framework*, Miami, Florida**, USA, 1995

[Dat95a] Data Access Corporation, **Developing Applications With DataFlex, Vs. 3.1**, Miami, Florida, USA, 1995

[Dat95b] Data Access Corporation, **UIMS Handbook, Vs. 3.1**, Miami, Florida, USA, 1995

[Dat95d] Data Access Corporation, **UIMS Refecence, Vs. 3.1**, Miami, Florida, USA, 1995

[Dat95e] Data Access Corporation, **Encyclopedia, Vl. 1 e 2, Vs. 3.1**, Miami, Florida, USA, 1995

[Dat98a] Data Access Corporation, **Database Guide, Vs. VDF 6.0**, Miami, Florida, USA, 1998

[Dat98b] Data Access Corporation, **Developing Applications, Vs. VDF 6.0**, Miami, Florida, USA, 1998

[Dat98c] Data Access Corporation, **Language Guide, Vs. VDF 6.0**, Miami, Florida, USA, 1998

[Dav84] DAVIDSON, Jack W. and FRASER, Christopher W., **Register Allocation and Exhaustive Peephole Optimization, Software-Practice and Experience**, September, 1984, 14, 9, 8857-865.

[Dso99] D'SOUZA, D.; WILLS, A. **Objects, Components and *Framework*s with UML - The Catalysis Approach.** USA: Addison-Wesley, 1999.

[Faq98] FAQs – **RescueWare**. http://www.relativity.com/products/faqs/index.html

[Fin97] FINNIGAN, P. J. et al. **The Software bookshelf**. IBM Systems Journal, V. 36, nº 4, p. 564-593, 1997.

[Fle95] Intercomp, **FlexPress, Ano 1** Numero 6, Abril/Maio 95, São Paulo- SP, 1995

[Fle98] Data Access do Brasil, **FlexPress, Ano 3** Numero 7, Junho/Julho 98, São Paulo- SP, 1998

[Fle99] Data Access do Brasil, **FlexPress, Ano 4** Numero 1, Janeiro/Fevereiro/Março 99, São Paulo- SP, 1999

[Fre87] FREEMAN, P., A **Conceptual Analysis of The DRACO Approach to Constructing Softwaer Systems**, IEEE Transactions on Softwaer Engineering, 13,7. Jul/1987.

[Fre96] FREITAS, F.G., LEITE J.C.S., SANT'ANNA M., **Aspectos Implementacíonais de um Gerador de Analisadores Sintáticos para Suporte a sistemas Transformacionais**. I Simpósio Brasileiro de Linguagens de programação, Belo Horizonte, 1996, pp. 115-127.

[Fuk99] FUKUDA, A. P. **Refinamento Automático de Sistemas Orientados a Objetos Distribuídos***, Qualificação de Mestrado*, UFSCar, 1999.

[Fuk99a] Fukuda, A.P., Prado A.F. - **Refinamento Automático de Sistemas Orientados a Objetos Distribuídos** - IV Workshop de Teses em Engenharia de Software - WTES'99, pg. 64-68. Florianópolis–SC, Brasil. 13 – 15 de Outubro, 1999.

[Fuk99b] FUKUDA, A.P., JESUS, E.S., PRADO A.F. -**Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos** - XXV Conferência Latino Americana de Informática – CLEI'99, pg. 471-482. Assunção-Paraguai. 30 de Agosto – 3 de Setembro, 1999.

[Gal95] GALL, Harald C., KLÖSH, René R.; MITTERMEIR, Roland T. **Architectural Transformation of Legacy Systems**. International Conference on Software Engineering, 11., Abril 1995. (Technical Report nº CS95-418)

[Gam95] GAMMA, E. et al. **Design Patterns.** Elements of Reusable Object-Oriented Software. Massachusetts: Addison-Wesley, 1995.

[Gol92] GLODNER, DOUG - **Convertendo Aplicações de DataFlex 2.3 para DataFlex 3.0**, Miami, Florida, USA, 1992

[Hai96] HAINAUT, J-L et al. **Structure Elicitation in Database Reverse Engineering**. In: Working Conference on Reverse Engineering (WCRE), 3., Monterey-California, 1996. *Anais*. IEEE, 1996, p. 131-140.

[Hal96] HALL B. - **Year 2000 tools and services**. - Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century, page 14. 1996.

[Hee89] HEERING J., HENDRIKS P.R.H., KLINT P., e REKERS J. – **The syntax definition formalism SDF** - *Reference manual*. SIGPLAN Notices, 24(11):43-75, 1989.

[Her91] HERRING, SCOTT - **3.0 DataFlex Concepts**, Miami, Florida, USA, 1991

[Icsr4] **ICSR4 Tutorial: Transformation Systems**, http://vtopus.cs.vt.edu/~edwards/icsr5/icsr4/tut-baxter.html,

[Jac91] JACOBSON, Ivar e LINDSTROM, Fredrik. **Re-engineering of old systems to an object-oriented architecture**. In: Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA). *Anais. 1991,* p. 340-350.

[Jes99] Jesus, E. S., Fukuda, A.P., Prado A.F. **Reengenharia de Software para Plataformas Distribuídas Orientadas a Objetos**, *XIII Simpósio Brasileiro de Engenharia de Software*, Outubro de 1999.

[Jes99a] JESUS, E.S., FUKUDA, A.P., PRADO A.F. - **Reengenharia de Software para Platafomas Distribuídas Orientadas a Objetos** - XIII Simpósio Brasileiro de Engenharia de Software - SBES'99, pg. 289-304. Florianópolis–SC, Brasil. 13 – 15 de Outubro, 1999.

[Jes99b] JESUS, E.S., PRADO A.F. - **Reengenharia de Sistemas de Software** - IV Workshop de Teses em Engenharia de Software - WTES'99, pg. 74-78. Florianópolis–SC, Brasil. 13 – 15 de Outubro, 1999.

[Klo96] KLÖSH, René R. **Reverse Engineering: Why and How to Reverse Engineer Software**. In: California Software Symposium (CSS), California, EUA, abril de 1996. *Anais*. University of Southern California, 1996, p. 92-99.

[Lei91a] LEITE, J.C.S, PRADO, A.F. **- Desing Recovery - A Multi-Paradigm Approach.** *First International Workshop on Software Reusability***.** In proceedings, pp.161-169. Dourtmund, Germany. July, 1991.

[Lei91b] LEITE, J.C.S., FREITAS, F.G., SANT'ANNA, M. – **Máquina Draco-PUC: A Technology Assembly for Domain Oriented Software Development**, 3rd IEEE International Conference of Software Reuse, Rio de Janeiro – RJ, 1994.

[Lei94] LEITE, J.C.S., FREITAS, F.G., SANT'ANNA M. **Draco-PUC Machine: A Technology Assembly for Domain Oriented Software Development**. *3rd International Conference of Software Reuse*. IEEE Computer Society Press. In proceedings, pp. 94-100. Rio de Janeiro, 1994.

[Lew95] LEWIS, T. et al. **Object Oriented Aplication *Frameworks*.** USA: Manning, 1995.

[Mah96] MAHLKE, S. and NATARAJAN, B. **"Compiler synthesized dynamic branch prediction,"** in Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture, pp. 153--164, December 1996. http://citeseer.nj.nec.com/mahlke96compiler.html

[Mar94] MARKOSIAN, Lawrence, et al. **Using na Enabling Technology to Reengineer Legacy Systems**. Communications of the ACM, V.37, nº5, p. 58-70, maio 1994.

[Mic00] **Microman Examples and Download Links for Lex & Yacc**, http://www.uman.com/lexyacc.shtml, 2000

[Mor01] MORAES, J. L. C., PRADO, A. F **- Geração Automática de Código Delphi a partir de Especificações em Catalysis** - XXVII Conferência Latino Americano de Informática – CLEI'2001. ISBN: 980-11-0527-5. Mèrida, México. 24-28 de Setembro, 2001. Documentação em CD-ROM.

[Nei80] NEIGHBORS, J.M., **Software Construction Using Components**, Doctoral dissertation, Information and Computer Science Dept. University of California, Irvine, 1980.

[Nei84] NEIGHBORS, J.M. **The Draco approach to Constructing Software from Reusable Components**. *IEEE Transactions on Software Engineering.* v.se-10, n.5, pp.564-574, September, 1984.

[Nog01a] NOGUEIRA, A. R., PRADO, A. F. - **Transformation of Procedural Dataflex to Object Oriented Visual Dataflex Applying the Reuse of a Framework** - 2nd International Conference on Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing – SNPD'2001. Pág. 856-863. ISBN: 0-9700776-1-0. Nagoya, Japan. 20-22 de Agosto, 2001.

[Nog01b] NOGUEIRA, A. R., PRADO, A. F.- **Transformação de Dataflex Procedural para Visual Dataflex Orientado a Objetos Reutilizando um Framework** -Workshop de Teses – XV Simpósio Brasileiro de Engenharia de Software – SBES'2001. Rio de Janeiro–RJ, Brasil. 03-05 de Outubro, 2001. Documentação em CD-ROM.

[Nov01] NOVAIS, R. E. A., PRADO, A. F. - **Reengenharia de Software Orientada a Componentes Distribuídos** XV Simpósio Brasileiro de Engenharia de Software – SBES'2001. Pág. 224-239. CDU: 681.31:061.68. Rio de Janeiro–RJ, Brasil. 03-05 de Outubro, 2001.

[Par91] PARR T.J., DIETZ H.G., e COHEN W.E. - **PCCTS Reference Manual**, 1.00 edition, 1991.

[Pen95] PENTEADO, R.D., GERMANO, F.; MASIERO, P.C. **Engenharia Reversa Orientada a Objetos do Ambiente StatSim: Método Utilizado e Resultados Obtidos**, In: Simpósio Brasileiro de Engenharia de Software, 9., Recife-PE, 1995. *Anais.* Recife, UFPE, 1995. p. 345-351.

[Pen96] PENTEADO, R.D. **Um Método para Engenharia Reversa Orientada a Objetos**. São Carlos, 1996. *Tese de Doutorado*. Universidade de São Paulo. 251p.

[Pen98] PENTEADO, R. A. D., MASIERO, P. C., PRADO, A. F., BRAGA, R. T. V - **Reengineering of Legacy Systems Based on Transformation Using the Oriented Object Paradigm** - 5th IEEE Working Conference on Reverse Engineering , pg 144 - 153, October 12th - October 14th, 1998 Honolulu, Hawaii.

[Pra00] PRADO, A. F., NOVAIS, E. R. A. - **Reengenharia Orientada a Objetos de Código Legado Progress 4GL** - XIV Simpósio Brasileiro de Engenharia de Software - SBES'2000. Pág. 21 – 36. CDU 681.31:519.683.2. João Pessoa–PB, Brasil. 4 – 6 de Outubro, 2000.

[Pra92] PRADO, A.F. **Estratégia de Engenharia de Software Orientada a Domínios.** Rio de Janeiro, 1992. Tese de Doutorado. Pontífica Universidade Católica. 333p.

[Pra98] PRADO, A.F., PENTEADO, R.A.D., ABRAHÃO, S.M., FUKUDA, A. P. **Reengenharia de Programas Clipper para Java**. *XXIV Conferência Latino Americana de Informática - CLEI 98.* pg. 383-394. - 19-23 de Outubro, 1998.

[Pra98] PRADO, A. F.; PENTEADO, R. A. D.; ABRAHÃO, S. M. e FUKUDA, A. P. - **Reengenharia de Programas Clipper para Java** - XXIV Conferência Latino Americana de Informática - CLEI 98, pg. 383-394, 19-23 Outubro, Quito-Equador.

[Pre95] PREE, W. **Design Patterns for Object – Oriented Software Development**. USA: Addison – Wesley, 1995.

[Rea92] REASONING SYSTEMS INCORPORATED. **Refine User's Guide, Reasoning Systems Incorporated.** Palo Alto, 1992.

[Rek85] REKOFF, M. G. **On Reverse Engineering**. IEEE Transactions on Systems, Man and Cybernetics, V.SMC-15, nº 2, p. 244-252, Março-Abril 1985.

[Rel00] **Relativity Technologies**, http://www.relativity.com/products/RescueWare/Methodology.htm, 2000

[Rog97] ROGERS, G. F. ***Framework* – Based Software Development in C++**. New Jersey: Prentice – Hall, 1997.

[Sag95] SAGE, Andrew P. **Systems Engineering and Systems Management for Reengineering**, Journal Systems Software, V.30, p. 3-25, 1995.

[San93] SANT'ANNA, M. Lavoisier: **Uma Abordagem Prática do Paradigma Transformacional.** Monografia de Graduação. Rio de Janeiro. PUC-Rio - Pontífica Universidade Católica do Rio de Janeiro. 1993. 100p.

[San99] SANT´ANNA, M. **Circuitos Transformacionais**. Rio de Janeiro, 99. Tese (Doutorado) – Departamento de Informática, Pontifica Universidade Católica.

[Sib97] Siber Systems Inc. - **CobolTransformer--Peek Under the Hood***: Technical White Paper*, 1997. - http://www.siber.com/sct/tech-paper.html

[Sne95] SNEED, Harry M.; NVÁRY, Erika. **Extracting Object-Oriented Specification from Procedurally Oriented Programs**, In: Working Conference on Reverse Engineering (WCRE), 2., Toronto, Canada, 1995. *Anais.* IEEE, 1995, p. 217-226.

[Sne96] SNEED, Harry M. **Object-Oriented COBOL Recycling**. In: Working Conference on Reverse Engineering (WCRE), 3., Monterey-CA, EUA, 1996. *Anais.* IEEE, 1996, p. 169-178.

[Sne97] SNEED H.M. (*Director*) - **SES Software-Engineering Service**, GmbH, *News Letter* - December 1997.

[Sun01] SUN Microsystems - **Java(TM) 2 Platform, Standard Edition, v1.2.2 API Specification: Package javax.swing.text** – disponivel em http://java.sun.com/products/jdk/1.2/docs/api/javax/swing/text/package-summary.html - Consultado em Junho de 2001.

[Tal96] TALIGENT White Paper. **Building Object-Oriented *Framework*s.** Taligent, Inc,Apple Computer, Inc, IBM Corporation and Hewlett-Packard Company, 1996, http://www.taligent.com.

[Tec97] TechForce B.V., P.O. Box 3108, 2130 KC Hoofddorp, **The Netherlands. COSMOS 2000** *White paper*, 1997.- http://www.cosmos2000.com/whitepap.pdf

[Tom86] TOMITA, M. **Efficient Parsing for Natural Languages--A Fast Algorithm for Practical Systems**. Kluwer Academic Publishers, 1986.

[Wile00] WILE David S. **POPART: Producer of Papers and Related Tools System Builders Manual.** Technical Report. USC/Information Sciences Institute, http://www.isi.edu/software-sciences/wile/Popart/popart.html, 2000

[Wilk95] WILKENING, D. E.; LOYALL, J. P.; PITARYS, M.J. e LITTLEJOHN, K. **A Reuse Approach to Software Reengineering**. Journal Systems Software, V.30, p. 117-125, 1995.

[Won95] WONG, Kenny; et al. **Structural Redocumentation: A case Study**. IEEE Software, p. 46-54, janeiro 1995.

ANEXO I - Gramática DFP

```
%left   'OR' 'AND' 'NOT' '~'
%left   '='   '<>' '<'  '<=' '>='
%left   'EQ'  'NE' 'LE' 'LT' 'GE' 'GT' 'IN'
%left   '*'   '/'  '+'   '-' '^'

%{
/* Variavel para controle da interpretacao das telas */
static int in_screen = 0;
static int in_command = 0;
/* Lista de comandos dinamicos
   Esta lista e incrmentada pelo #command */
static char* domain_keywords[] = {
    "abortcheck",
    "accept_seg",
    "accept_text",
    "acessa",
    "acha$numdc",
    "acha$refer",
    "aguarda",
    "arredonda",
    "atualiza_ind_disp",
    "atualiza_saldo",
    "atualiza_saldo_cre",
    "atualiza_saldo_deb",
    "aviso",
    "aviso$disp",
    "bad",
    "beep",
    "begin_dfini",
    "bin_dec",
    "binario",
    "blink",
    "breakline",
    "busca_autorizacao",
    "busca_icms",
    "busca_numcpv",
    "busca_par_pagto",
    "busca_tipoprnf",
    "busca_unimed",
    "buscacli",
    "buscadep",
    "buscafil",
    "buscafil_rec_pag",
    "calcage",
    "calcgiro",
    "calcula_taxa",
    "calcula_valor",
    "cb90a",
    "cb92a",
    "cb93a",
    "cb94a",
    "cb_apaga_texto",
    "cb_data_valida",
    "cb_remove_arq",
    "centra",
    "central",
    "chain_sirc",
    "checkexit",
    "chopline",
    "clear01c",
    "cliabc",
    "code39",
    "colocabarra",
    "completa",
    "cpage",
    "dayofweek",
    "deleta_hist",
    "desbloqueio",
    "display_text",
```

```
"editaloc",
"editaref",
"encontra_servico",
"end_dfini",
"endfind519",
"endlineloop",
"endlist",
"endscan",
"entryfind",
"estoque_est",
"exclui_lanc",
"exist?",
"existe_arq",
"extend",
"f10f9f4",
"f10f9f6f",
"field_def_",
"field_name",
"file_def",
"filenam",
"filetyp",
"find01d",
"find01p",
"find515",
"find518ato",
"find519",
"findaccept",
"findequal",
"finds01a",
"finds180",
"flegafind519",
"fmoeda",
"formfedesc",
"fsysdate",
"gera_dat",
"gera_def",
"gera_lanc",
"gerasitu01c",
"getenv",
"getesc",
"getimp_dos",
"getnunot",
"good",
"graf",
"grava_lancto",
"gravacon",
"gravaconf",
"gravahist",
"help$img",
"help_def",
"hex_bin",
"hexa",
"hexa_dec",
"highlight",
"hrcentes",
"hrnormal",
"index_def_x",
"indica$situacao",
"inicia_icms",
"inkey_seg",
"invalid_date",
"invcodigo",
"inverdata",
"invp_aammdd",
"lancamento$",
"lancsimples1",
"lancsimples2",
"libnunot",
"limpa_mens",
"lineloop",
"linha_topo",
"list",
"lowercase",
"lowlight",
"lpad",
"m_open",
"makeaddress",
```

```
"makename",
"makephone",
"makestr",
"makestr$help",
"makestri",
"mensagem",
"menserr",
"mes_demanda",
"minimenu",
"minput",
"modelimp",
"modelimp_nota",
"monta_doccpv",
"monthofdate",
"move$find",
"move$proc",
"move_deletados",
"moveall",
"movec",
"movesit",7
"obtem_portador",
"old_gerasitu01c",
"old_validavei",
"opecxa",
"open_as",
"open_ct",
"open_status",
"outfesc",
"outpesc",
"overlay",
"paccept",
"parse",
"parsedate",
"pause",
"pesq_602",
"poprec",
"preco_pma",
"preco_pv2",
"printfile",
"prockey",
"procsis",
"procura_conta",
"prog_acesso",
"purge",
"pushrec",
"range?",
"rdisplay",
"read_dfini",
"read_reg",
"regravacon",
"rev",
"rightjustify",
"roundcalc",
"rpad",
"rptcheck",
"rptprms",
"saldodev",
"saldopag",
"saldorec",
"scan",
"sdoclifor",
"setaecf",
"setaesc",
"situacao$dst",
"somalcto",
"sumr",
"sweda_def",
"sweda_envia_recebe",
"sweda_esc",
"sweda_esc.01",
"sweda_esc.02",
"sweda_esc.03",
"sweda_esc.05",
"sweda_esc.07",
"sweda_esc.08",
"sweda_esc.09",
"sweda_esc.10",
```

```
    "sweda_esc.11",
    "sweda_esc.12",
    "sweda_esc.13",
    "sweda_esc.14",
    "sweda_esc.17",
    "sweda_esc.18",
    "sweda_esc.19",
    "sweda_esc.20",
    "sweda_esc.23",
    "sweda_esc.26",
    "sweda_esc.27",
    "sweda_esc.28",
    "sweda_esc.29",
    "sweda_esc.30",
    "sweda_esc.31",
    "sweda_esc.33",
    "sweda_esc.35",
    "sweda_esc.39",
    "sweda_esc.40",
    "sweda_esc.50",
    "sysdthr",
    "terminate",
    "timestr",
    "tipopen",
    "title",
    "tp_trs_cxa",
    "ubox",
    "udrawc",
    "valida_serienf",
    "validavei",
    "valprese",
    "vencto_fixo",
    "ver$dados$cta",
    "ver$juncao",
    "ver$periodo",
    "ver_doctotp",
    "ver_ult_mov_sai",
    "verif_aut_pagto",
    "verifica_bloqueio",
    "verifica_moeda",
    "verifica_senha",
    "voltadata",
    "write_reg",
    "writelsc",
    "writesc",
    NULL,
};
/* Lista de caracteres ignorados no conteudo do programa
   e que nao devem ser ignorados nas telas */
static char* domain_ignore[] = {
    " ",
    "\n",
    "\t",
    NULL,
};

/* Funcao para verificar se i ID encontrado e um comando dinamico */
static int is_keyword(char* lexeme) {
  int i = 0;
  /*printf(" * ID: %s *",lexeme); */
  if (in_command==1) return(0);
  if (in_screen==1) return(0);
  /*if (strcmp(lexeme,"!A") == 0) return(IDBASE);
  if (strcmp(lexeme,"!a") == 0) return(ID);
  */
  if (strlen(lexeme)==1) return(0);
  if (strlen(lexeme)==2) return(0);
  /* procurando nos comandos dinamicos */
  while (domain_keywords[i] != NULL) {
    /*printf("\n *Procurando comando: %s *  Seq: %d Comando:
%s",lexeme,i,domain_keywords[i]);*/
    if (strcmp(domain_keywords[i], lexeme) == 0) {
      printf("\n **Reconhecido comando dinamico**  Seq: %d keyword:
%s\n",i,domain_keywords[i]);
      return(_DT_DODOcomando_nao_tratadoDODO);
    }
    else
```

```
      i++;
  }
  return(0);
};
/* Funcao para verificar deve ignorar ou nao o caracter */
static char* is_ignore(char* lexeme) {

  int n = 0;
  int i = 0;
  if (in_screen==0) return(0);
  while (domain_ignore[i] != NULL) {
    if (strcmp(domain_ignore[i], lexeme) == 0) {
      return(domain_ignore[i]);
    }
    else
      i++;
  }
  return(NULL);
};
/* Funcao para testar string vazia */
static int is_string(char* lexeme) {
  if (strcmp(lexeme,"""") == 0) return(1);
  if (strcmp(lexeme,"''") == 0) return(1);
  return(0);
};

/* Funcao para criar um novo comando dinamico

   FALTA TERMINAR  ! ! !

*/
static int add_if_new_keyword(char* lexeme) {
  int n = 0;
  int i = 0;
  char* tempo;
  while (domain_keywords[i] != NULL) {
    /*printf("\n *****Procurando comando: %s *  Seq: %d Comando:
%s",lexeme,i,domain_keywords[i]);*/
    if (strcmp(domain_keywords[i], lexeme) == 0) {
      /*printf("\n\n **Comando dinamico ja existe**  Seq: %d keyword:
%s\n\n\n\n",i,domain_keywords[i]);*/
      return(0);
    }
    else {
      i++;
    }
  }
  domain_keywords[i] = new char*;
  strcpy(domain_keywords[i], "MYACCEPT");
  printf("\n\n\n ** Comando dinamico criado **  %s Seq: %d\n\n\n",lexeme,i);
  //sprintf(tempo,"%s",lexeme);
  //printf("\n\n\n ** Comando dinamico criado **  %s Seq: %d\n\n\n",tempo,i);
  i++;

  domain_keywords[i]=NULL;
  return(0);
};
/*
   ATENCAO
   As regras que contem acoes semanticas devem ficar no final
   senao gera erro nos transformadores
*/
/* Funcao usada na regra ID para testar comandos dinamicos */
#define test_for_kwd() { int result = is_keyword(yytext); if (result != 0) {
yylval.lc=PutDelimiters(yytext); tokenpos +=yyleng; return(result); }}
/* Funcao usa na regra #command para criar um novo comando */
#define add_new_kwd() {int result = add_if_new_keyword(yytext); }
/* Funcao usada no inicio da tela  */
#define init_screen() { in_screen=1; }
/* Funcao usada no final da tela  */
#define end_screen() { in_screen=0; }
/* Funcao usada na regra ID para testar comandos dinamicos */
#define init_command() { in_command=1; printf("\n inicio do comando \n"); }
/* Funcao usada no final da tela  */
#define end_command() { in_command=0; printf("\n final do comando \n"); }
/* Funcao para testar os caractere ignorados (tela) */
```

```
#define test_for_ignore() { char* result = is_ignore(yytext); if (result != NULL) {
yylval.lc=PutDelimiters(result); tokenpos +=yyleng; return(IDTUDO); } }
%}

%%
program     :  statements+
        ;
//program_name  : '//SourceFileName:' .sp p_name
//      ;
//p_name        : label
//      ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/* TIPOS DE LINHAS ENCONTRADAS */
statements  : screen
        | .slm com_state
        | .slm procedure
        | .slm function
        ;
com_state   : .slm var_decl .nl .slm
        | .slm com_comp  .slm
        | .slm (indicador  .sp)? com_line .slm
        ;
procedure   : .nl .slm 'procedure' .sp ID (.sp ('global'|('for' .sp ID)))? (.sp type_var .sp
ID)* (.sp 'returns' .sp type_var)? .nl .slm .lm(+3) .slm
              com_state*
          .lm(-3) .slm 'end_Procedure' .nl .nl .slm
        ;
function    : .nl .slm 'function' .sp ID .sp (.sp ('global'|('for' .sp ID)))? (.sp type_var
.sp ID)* (.sp 'returns' .sp type_var) .nl .slm .lm(+3) .slm
             com_state*
          .lm(-3) .slm 'end_function' .nl .nl  .slm
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/* TELAS */
/* As telas nao estao tratando corretamente os espacos e
   retornos de linhas pois estao gerando conflito             */
screen      : '/' id_tela screen_cnt end_tela
        ;
/* Os comando que acionam funcoes foram para o final
id_tela     : { init_screen(); } ID
        ;
end_tela    : { end_screen(); } (FIMTEL|screen)
        ;
*/
screen_cnt  : screen_body+
        ;
screen_body     : IDTUDO
        | ID
        | LBROT
        | NUM
        | IDPONT
        | stri
        | oper_rel
        | '(' | ')' | ']' | '[' | '}' | '{' | ',' | '-' | '+' | '*'
        | '=' | '<' | '>' | '>=' | '<=' | '<>' | '|'
        | janela
        ;

/*
screen_body     : IDTUDO
        | ID
        | LBROT
        | NUM
        | IDPONT
        | janela
        | { printf("\n\n\n\n\ com_state na tela %s \n\n\n",yytext); } com_state

|'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'
x'|'y'|'z'

|'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'
X'|'Y'|'Z'
        |'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
        |'!'|'|'|'@'|'#'|'$'|'%'|'&'|'?'|'.'|':'
```

```
        | LBROT
        | janela
        | ':' | '|' | '+' | '-' | '.' | '(' | ')' | '*' | '>' | '<'
        ;
*/
janela      : jan_num
        | jan_dat
        | jan_str
        ;
/* estas regras estao gerando um shift/reduce
   por causa do ponto existente em ambas--------*/
jan_num     : JIDNUM
        ;
/*-----------------------------------------------------*/
/*   a janela do tipo real esta dando conflito por causa do 'e'  */
/*
        | jan_rea
jan_rea     : JID 'e' JID
        ;
*/
jan_dat     : JIDDAT
        ;
/*
retirado por causa de dois shift/reduce
        | ('_')+ ('/') ('_')+ ('/')? ('_')+
*/
jan_str     : JID
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/* DECLARACAO DE VARIAVEIS */
var_decl    : 'GLOBAL'? .sp type_var (.sp var_id (.sp num)?)+
        | 'LOCAL'   .sp type_var (.sp var_id)+
        ;
type_var    : 'DATE'
        | 'INDICATOR'
        | 'INTEGER'
        | 'NUMBER'
        | 'REAL'
        | 'STRING'
        ;
var_id      : ID
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/* GRUPOS DE LINHAS DE COMANDO */
com_line    : com_macro .slm
        | com_bloco .slm
        | com_interacao  (.sp label)* .slm
        | com_eventos  (.sp label)* .slm
        | com_desvio (.sp label)* .slm
        | com_atribuicao (.sp (indicador|label))* .slm
        | com_form .slm
        | com_bd .slm
        | linha_simples (.sp label)* .slm
        | linha_basica
        | com_out_esp .slm
        | nao_tratado .slm
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
com_macro   : com_enter
        | com_rel
        ;
/*----Macro Enter-----*/
com_enter   : .nl .slm 'entergroup' (.sp label)+ .nl .slm .lm(+3) .slm
           com_state*
          .slm .lm(-3) .slm 'endgroup' .nl .nl .slm
        | .nl .slm 'enter' (.sp ID)+ .nl .slm .lm(+3) .slm
           com_state*
          .slm .lm(-3) .slm 'enterend' .nl .nl .slm
        | .slm 'entdisplay' (.sp label)*
        | .slm 'entry' (.sp label)+
        ;
/*-----------------------------------------------------------------*/
/*----Reletorios-----*/
com_rel     : .nl .slm 'report' (.sp label)+ .nl .slm .lm(+3) .slm
```

```
           com_state*
         .slm .lm(-3) .slm 'reportend' .nl .nl .slm
       ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/*
       | 'for' (.sp label)+ .slm .lm(+3) .slm
           com_state*
         .slm .lm(-3) .slm c_loop .nl .nl .slm
*/
com_bloco   : 'begin' .slm .lm(+3) .slm
           com_state*
         .slm .lm(-3) .slm 'end' .nl .nl .slm
       | 'for' (.sp label)+ .slm
       | 'while' (.sp (label|indicador))+ .slm .lm(+3) .slm
           com_state*
         .slm .lm(-3) .slm 'end' .nl .nl .slm
       | 'repeat' .slm .lm(+3) .slm
           com_state*
          .slm .lm(-3) .slm 'until'(.sp (label|indicador))+
       | c_if (.sp (label|indicador))+ .sp com_line
       | 'ifchange' (.sp label) .sp com_state
       | 'else' .sp com_line
       | 'auto' .sp com_state
       | 'auto2$' .sp com_state
       | 'section' .sp label (.sp label)?
       | .nl .slm c_rotina .slm .lm(+3) .slm
       ;
c_if        : 'if'
       | 'if_'
       | 'ifnot'
       | 'ifnot_'
       ;
c_loop      : 'loop'
       | indicador .sp 'loop'
       ;
/* se colocar a subrotina normal esta gerandoi muita ambiguidade */
/* o formato correto da sub rotina deveria tratar o return*/
c_rotina    : LBROT
       ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
com_interacao   : 'accept'
       | 'keycheck'
       | 'inkey'
       | 'inkey$'
       | 'input'
       ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
com_eventos : 'backfield'
       | 'entagain'
       | 'keyproc'
       ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/* O loop ja esta definido na regra do for mas deve estar aqui tambem */
com_desvio  : c_loop
       | 'pagecheck'
       | 'abort'
       | 'call'
       | 'chain'
       | 'runprogram'
       | 'gosub'
       | 'on_gosub'
       | 'on_goto'
       | 'return'
       | 'system'
       ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
com_atribuicao  : 'blankform'
       | 'indicate'
       | 'clearform'
       | 'display'
       | 'print'
       | 'print_wrap'
```

```
        | 'subtotal'
        | 'read'
        | 'read_block'
        | 'readln'
        | 'calculate'
        | 'calc'
        | 'move'
        | 'movedate'
        | 'moveint'
        | 'movenum'
        | 'movereal'
        | 'movestr'
        | 'cmdline'
        | 'memory'
        | 'field_def'
        | 'file_size'
        | 'get_argument_size'
        | 'get_current_directory'
        | 'get_environment'
        | 'index_def'
        | 'sysdate'
        | com_mat
        | com_stri
        ;
/*----Matematica-----*/
com_mat     : 'abs'
        | 'acos'
        | 'asin'
        | 'atan'
        | 'cos'
        | 'increment'
        | 'decrement'
        | 'exp'
        | 'hi'
        | 'log'
        | 'low'
        | 'mod'
        | 'random'
        | 'round'
        | 'sin'
        | 'sqrt'
        | 'tan'
        ;
/*----String-----*/
com_stri    : 'append'
        | 'ascii'
        | 'character'
        | 'eval'
        | 'insert'
        | 'left'
        | 'length'
        | 'lowercase'
        | 'ltrim'
        | 'mid'
        | 'overstrike'
        | 'pad'
        | 'pos'
        | 'remove'
        | 'replace'
        | 'replaces'
        | 'right'
        | 'rtrim'
        | 'trim'
        | 'uppercase'
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
com_form    : 'autopage' .sp ID (.sp num)?
        | 'name' (.sp c_name)*
        ;
c_name      : label
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
com_bd      : 'open' (.sp label)*
        | 'relate' (.sp label)*
        ;
```

```
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
com_out_esp : 'findkey' .sp oper_rel .sp label .sp label .sp label .sp 'for' (.sp label .sp
oper_rel .sp label)+
        | 'page' (.sp 'set')? (.sp label)+
    ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
linha_simples   : com_graf
        | com_mu
        | com_seq
        | com_io_out
        | com_acessobd
        | com_sist
        | com_uims
        | com_outros
        | com_sircx
        ;
/*----Elementos graficos-----*/
com_graf    :  'arc'
        |  'box'
        |  'chart'
        |  'chart_init'
        |  'circle'
        |  'display_cut'
        |  'display_graphic'
        |  'ellipse'
        |  'flood'
        |  'get_grxy'
        |  'getdrange'
        |  'gr_print'
        |  'graph_getfirst'
        |  'graph_getnext'
        |  'graph_init'
        |  'graph_stat'
        |  'graph_sum'
        |  'graph_value'
        |  'graphic'
        |  'grxy'
        |  'line'
        |  'palette'
        |  'pie'
        |  'plot'
        |  'save_cut'
        |  'save_graphic'
        |  'line_style'
        |  'set_text'
        |  'setworld'
        ;
/*----Controle muiti-usuario-----*/
com_mu      : 'despool'
        | 'reread'
        | 'unlock'
        ;
/*----Entrada e Saida Sequencial-----*/
com_seq     : 'append_output'
        | 'close_input'
        | 'close_output'
        | 'direct_input'
        | 'direct_output'
        | 'get_channel_position'
        | 'set_channel_position'
        | 'write'
        | 'writeln'
        ;
/*----Entrada e saida padrao-----*/
com_io_out  : 'clearscreen'
        | 'clearxy'
        | 'gotoxy'
        | 'screenmode'
        | 'show'
        | 'showln'
        ;
/*----Banco de Dados-----*/
com_acessobd    : 'attach'
        | 'clear'
        | 'close'
```

```
        | 'constrained_clear'
        | 'constrained_find'
        | 'constraint'
        | 'constraint_set'
        | 'constraint_validate'
        | 'delete'
        | 'find'
        | 'save'
        | 'saverecord'
        | 'set_relate'
        | 'vfind'
        | 'zerofile'
        ;
/*----Sistema-----*/
com_sist    : 'copyfile'
        | 'diretory'
        | 'erasefile'
        | 'file_mode'
        | 'filelist'
        | 'make_file'
        | 'read_dfini'
        | 'registration'
        | 'renamefile'
        | 'set_argument_size'
        | 'set_dfpath'
        | 'set_termlist'
        | 'sleep'
        | 'use'
        ;
/*----Orientacao a Objetos-----*/
com_uims    : 'broadcast'
        | 'define'
        | 'define_arguments'
        | 'define_symbol'
        | 'delegate'
        | 'delete_db'
        | 'delete_field'
        | 'delete_index'
        | 'delete_permission'
        | 'dependent_item'
        | 'diskfree'
        | 'end_class'
        | 'end_enumeration_list'
        | 'end_item_group'
        | 'end_item_list'
        | 'end_object'
        | 'end_procedure'
        | 'end_pull_down'
        | 'end_transaction'
        | 'ent$key'
        | 'entermode'
        | 'entfind'
        | 'entry_item'
        | 'entry_name_item'
        | 'entsfind'
        | 'entupdate'
        | 'enumeration_list'
        | 'error_report'
        | 'field_map'
        | 'file$mode'
        | 'file$mode$help'
        | 'file_exist'
        | 'filelist$put'
        | 'fill_field'
        | 'find$page'
        | 'find_'
        | 'flex$init'
        | 'for_'
        | 'format$range'
        | 'forward'
        | 'forward_end_construct'
        | 'function_return'
        | 'get'
        | 'get$field$value$help'
        | 'get$obj$image'
        | 'get$set'
        | 'get_attribute'
```

```
| 'get_channel_size'
| 'get_current_colors'
| 'get_current_lockcount'
| 'get_cursor_visible'
| 'get_date_attribute'
| 'get_date_format'
| 'get_directory'
| 'get_effective_user_id'
| 'get_field_value'
| 'get_file_mod_time'
| 'get_file_owner'
| 'get_filelist'
| 'get_group_id'
| 'get_number_format'
| 'get_permission'
| 'get_resource_name'
| 'get_resource_type'
| 'get_transaction_retry'
| 'get_user_id'
| 'get_video_mode'
| 'getxy'
| 'goto'
| 'grshow'
| 'grshowln'
| 'handle$message'
| 'hard$exit'
| 'help'
| 'if$help'
| 'import_class_protocol'
| 'include_resource'
| 'indct$$0'
| 'indct$$1'
| 'indct$as'
| 'indct$group'
| 'indct$status'
| 'inherit_screen'
| 'initialize_interface'
| 'is_file_included'
| 'item_group'
| 'item_list'
| 'join'
| 'join_if'
| 'joinend'
| 'joinmove'
| 'load_def'
| 'load_driver'
| 'lock'
| 'login'
| 'logout'
| 'make_directory'
| 'make_temp_file'
| 'makedef$'
| 'menu_required'
| 'mes'
| 'message$address'
| 'move$find'
| 'move_sub_page'
| 'multi$'
| 'multiback$'
| 'name_column'
| 'name_item'
| 'name_object'
| 'name_property'
| 'netware_get_available'
| 'netware_get_tts_flag'
| 'netware_set_tts_flag'
| 'object'
| 'object$properties'
| 'odd'
| 'old_attach'
| 'old_clear'
| 'old_close'
| 'old_delete'
| 'old_open'
| 'old_refresh'
| 'old_relate'
| 'old_reread'
```

```
        | 'old_save'
        | 'old_saverecord'
        | 'old_zerofile'
        | 'on'
        | 'on_item'
        | 'on_key'
        | 'on_name_item'
        | 'os$call'
        | 'outclose'
        | 'outfile'
        | 'output_aux_file'
        | 'output_aux_file_help'
        | 'output_def_file'
        | 'output_fd_file'
        | 'pause'
        | 'pop'
        | 'property'
        | 'property$help'
        | 'read_hex'
        | 'register_function'
        | 'register_object'
        | 'register_procedure'
        | 'register_resource'
        | 'remove_char'
        | 'remove_directory'
        | 'repeat_item'
        | 'replaceall'
        | 'runprogram_pipe'
        | 'screen_optimize'
        | 'searchkey'
        | 'select$find'
        | 'semana'
        | 'send'
        | 'send$cmd'
        | 'send$help'
        | 'set'
        | 'set$field$value$help'
        | 'set_attribute'
        | 'set_cursor_visible'
        | 'set_date_attribute'
        | 'set_deferred'
        | 'set_directory'
        | 'set_field_value'
        | 'set_file_field'
        | 'set_file_index'
        | 'set_file_mod_time'
        | 'set_file_owner'
        | 'set_filelist'
        | 'set_group_id'
        | 'set_length_offset'
        | 'set_line_style'
        | 'set_permission'
        | 'set_resource_library'
        | 'set_segment_misc'
        | 'set_transaction_retry'
        | 'set_user_id'
        | 'set_video_mode'
        | 'sort'
        | 'sound'
        | 'start_ui'
        | 'status$'
        | 'stop'
        | 'stop_here'
        | 'structure_abort'
        | 'structure_copy'
        | 'structure_end'
        | 'structure_start'
        | 'sub$image$define'
        | 'sub_page'
        | 'subtract'
        | 'sum'
        | 'sysdate$help'
        | 'sysdate4'
        | 'ui_accept'
        | 'unload_driver'
        | 'valid_drive'
        | 'variable_file_command'
```

```
        | 'vconstrain'
        | 'vconstrain_as'
        | 'vconstrain_compare'
        | 'vconstrain_relate'
        | 'write$hex$help'
        | 'write$str'
        | 'write_hex'
        | 'forward_begin_construct'
        | 'set$field$value$help$2'
        | 'get_licensed_max_users'
        | 'get_effective_group_id'
        | 'get_current_output_channel'
        | 'get_current_user_count'
        | 'get_current_input_channel'
        ;
/*----outros nao classificados-----*/
com_outros  : 'clearwarning'
        | 'debug'
        | 'error'
        | 'formfeed'
        | 'clear_option'
        | 'format'
        | 'output'
        | 'output_wrap'
        | 'set_option'
        | 'setchange'
        | 'abort_transaction'
        | 'add'
        | 'add$image$args'
        | 'add_permission'
        | 'api_attach'
        | 'api_clear'
        | 'api_close'
        | 'api_delete'
        | 'api_open'
        | 'api_refresh'
        | 'api_relate'
        | 'api_reread'
        | 'api_save'
        | 'api_saverecord'
        | 'api_zerofile'
        | 'aspect'
        | 'auto3$'
        | 'auto4$'
        | 'bar'
        | 'base_create_class'
        | 'begin_transaction'
        | 'break'
        | 'break$'
        | 'break$down'
        | 'break$prt'
        | 'break$t3'
        | 'break$up'
        | 'breakinit'
        | 'breakpoint'
        | 'broadcast_focus'
        | 'call$driver$1'
        | 'call$driver$2'
        | 'call_driver'
        | 'case'
        | 'caseelse'
        | 'caseend'
        | 'cgc_check'
        | 'chain$help'
        | 'cic_check'
        | 'class'
        | 'clearstack'
        | 'compute$crc$help'
        | 'compute_screen_crc'
        | 'conditional'
        | 'conditionalend'
        | 'constraint_init'
        | 'constraint_save_init'
        | 'copy$db$callback'
        | 'copy$db$compression'
        | 'copy$record$help'
        | 'copy_db'
```

```
        | 'copy_records'
        | 'count'
        | 'create_field'
        | 'create_index'
        | 'declare_datafile'
        | 'default_end_object'
        ;
/*----comando do SIRCX-----*/
com_sircx   : 'beep'
        | 'atimage'
        | 'arredonda'
        | 'accept_date'
        | 'accept_segundo'
        | '@listar_aux'
        | '@listar_pntsaux'
        | '@listar_pntsaux2'
        | 'break$ln'
        | 'breakln'
        | 'calc_time'
        | 'clrbuf'
        | 'completa'
        | 'clearmessage'
        | 'checa_seculo'
        | 'check_number'
        | 'entwait'
        | 'entskip'
        | 'diasmes'
        | 'endpoint'
        | 'endscanenter'
        | 'display_date'
        | 'encript'
        | 'exit'
        | 'fim_da_lista'
        | 'fim_pnts'
        | 'findfirst'
        | 'findnext'
        | 'findprior'
        | 'foot'
        | 'get$choice'
        | 'getenv'
        | 'highvalue_date'
        | 'ifcolun'
        | 'ifimage'
        | 'ifnumeral'
        | 'ifwindow'
        | 'loop_time'
        | 'lowvalue_date'
        | 'menu'
        | 'message'
        | 'modulo_10'
        | 'modulo_11'
        | 'listar'
        | 'listar_pnts'
        | 'inkey_segundo'
        | 'inkey_time'
        | 'invert_date'
        | 'invp_aammdd'
        | 'invp_ddmmaa'
        | 'inicia$variaveis'
        | 'move_default'
        | 'moveall'
        | 'moveano'
        | 'movedata'
        | 'moveimage'
        | 'movereg'
        | 'movewindow'
        | 'norequir'
        | 'ncgc_check'
        | 'ncic_check'
        | 'point$paint'
        | 'point_select'
        | 'pos_char'
        | 'read_valor'
        | 'scanbreak'
        | 'selec_and$find'
        | 'scanenter'
        | 'scanentry'
```

```
        | 'reverse'
        | 'show_file'
        | 'split_date'
        | 'stopskip'
        | 'sub'
        | 'showxy'
        | 'sys$chk'
        | 'sysdata'
        | 'testa_tecla'
        | 'tira_ponto'
        | 'wait'
        | 'x_report'
        | 'yesno'
        | 'center'
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
linha_basica    : IDBASE .sp indicador .sp label (.sp ('|')? label)*
        ;
/*-----------------------------------------------------------------*/
nao_tratado : '$$comando_nao_tratado$$' (.sp label)*
        ;
/*-----------------------------------------------------------------*/
/* COMANDOS DE COMPILACAO */
com_comp    : c_include
        | comp_com (.sp label (oper_all label)?)+ .nl .slm .lm(+3) .slm
             com_state*
             (.slm .lm(-3) .slm '#else' .slm .lm(+3) .slm  com_state*)?
          .slm .lm(-3) .slm '#endif' .nl .nl .slm
        | out_comp (.sp label)+
        | new_command
        ;
comp_com    : '#if'
        | '#ifclass'
        | '#ifdef'
        | '#iftype'
        | '#ift2ype'
        | '#ifsame'
        | '#ifind'
        | '#ifsub'
        ;
out_comp    : '#check'
        | '#chksubn'
        | '#replace'
        | '#set'
        | '#noisy'
        | '#keyproc'
        | '#fref'
        | '#revision'
        | '#compat'
        | '#num'
        | '#real'
        | '#str'
        | '#data'
        | '#pop'
        | '#push'
        | '#xpop'
        | '#xpush'
        | '#dfpush'
        | '#stksym'
        | '#spush'
        | '#spop'
        | '#error'
        | '#freg'
        | '#entopt'
        | '!set'
        | '!$'
        ;
c_include   : '#include' .sp ID
        ;
/*-----------------------------------------------------------------*/
/* ESTRUTURAS DE APOIO */
oper_calc   : '+' | '-' | '*' | '/' | '^'
        ;
label       : ID
        | NUM
        | stri
```

```
        | expressao
        | exp_entry
        | oper_rel
        | oper_log
        | IDVACOMP
        ;
label2      : ID
        | NUM
        | stri
        | pre_func '(' (expressao|label2) (.sp ',' .sp (expressao|label2))* ')'
        ;
stri        : STRI
        ;
oper_rel    : 'EQ' | 'NE' | 'LE' | 'LT' | 'GE' | 'GT' | 'IN'
        ;
oper_exp    : OPER_EXP
        ;
oper_log    : 'AND'
        | 'OR'
        | 'NOT'
        | '~'
        ;
indicador   : ('[' (oper_log? .sp label .sp)* ']')+
        ;
/* Se a funcao tem o mesmo nome de um comando deve ser colocada aqui */
pre_func    : 'length'
        | 'mid'
        | 'trim'
        ;
oper_all    : oper_exp
        | oper_calc
        | oper_log
        ;
expressao   : '(' label2  ')'
        | '(' (oper_all (expressao|label2))+ ')'
        | '(' ((expressao|label2) oper_all?)+ (expressao|label2) ')'
        | '(' expressao ')'
        ;
/*
        | '(' (ID|NUM) (oper_exp|oper_calc|oper_log)? expressao+ ')'
        | '(' expressao+ (oper_exp|oper_calc|oper_log)? (ID|NUM) ')'
*/
exp_entry   : '{' .sp label (.sp (','|'=') .sp label)* .sp '}'
        ;
num     : NUM
        ;
/* o tipo real esta dando conflito
        | 'E' NUM
*/
/*
   ATENCAO
   As regras que acionam funcoes devem ficar no final
   senao gera erro nos transformadores
*/
id_tela     : { init_screen(); } ID (' RESIDENT')?
        ;
end_tela    : { end_screen(); } (FIMTEL|screen)
        ;
new_command : { init_command(); } .nl '#command' { add_new_kwd(); } .sp ID { end_command(); }
(.sp label)* .nl .slm .lm(+3) .slm
             com_state*
          .lm(-3) .slm  '#endcommand' .nl .nl .slm
        ;
/*
new_command : '#command' { printf("\n CRIANDO COMANDO DINAMICO %s \n",yytext); add_new_kwd();
} .sp ID (.sp label)*
        ;
*/
/* ESTRUTURAS TERMINAIS */
IGNORE      : "//"[^\n]*
        ;
IGNORE      : [ \t\n]       { test_for_ignore(); }
        ;
IGNORE      : "#rem"[^\n]*
        ;
STRI        : \'([^\']|\'\')*\'
        ;
```

```
STRI        : \"([^\"]|\"\")*\"
        ;
FIMTEL      : "/*"[ ]*\n
        ;
IDBASE      : [\!][A]
        ;
NUM     : [\!\+\-\*\/\^]?[0-9]+(\.[0-9]+)?
        ;
JIDNUM      : [\_]+\.[_]*
        ;
JIDDAT      : [\_]+\/[_]+(\/[_]+)?
        ;
JID     : [\_]+(\@[\_]+)?
        ;
IDTUDO      : [^ A-Za-z_\@\#\$\%\&\?\t\n\)\(\}\{]+
        ;
LBROT       : [A-Za-z_\@\#\$\%\&\?]+[A-Za-z_0-9\-\@\#\$\%\&\?\.]*[A-Za-z0-9]+\:
        ;
ID      : [\!]?[A-Za-z_\@\#\$\%\&\?]+[A-Za-z_0-9\-\@\#\$\%\&\?\.]*[A-Za-z_0-9\$\&]*     {
test_for_kwd(); }
        ;
IDVACOMP    : ["|"][VvCc][IiSsNnRr][0-9]+
        ;
OPER_EXP    : ([\=\<\>]|["<="]|["<>"]|["=>"])
        ;
IDPONT      : [\@\#\$\%\&\?\-\.]+
        ;
%%
```

ANEXO II - Gramática VDFOO

```
%left   'OR'
%left   'AND'
%left   'NOT'  '~'
%left   '='   '<>' '<'  '<=' '>='
%left   'EQ'  'NE' 'LE' 'LT' 'GE' 'GT'
%left   'IN'
%left   '+'   '-'
%left   '*'   '/'
%left   '^'

%%
program     : (program_name .nl)? statements+
        ;
program_name    : '//SourceFileName:' .sp label
        ;
/* TIPOS DE LINHAS ENCONTRADAS */
statements  : screen
        | .slm oo_class
        | .slm oo_body
        ;
/* COMANDOS DA ORIENTACAO AO OBJETOS */
oo_body     : .slm oo_procedure
        | .slm oo_function
        | .slm oo_enulst
        | .slm oo_procbody
        ;
oo_procbody : .slm com_state
        | .slm oo_obj
        ;
oo_class    : .nl .slm
          'class' .sp ID .sp 'is' .sp ('a'|'an') .sp  ID .nl .slm .lm(+3) .slm
             oo_body*
          .slm .lm(-3) .slm 'end_class' .nl .nl .slm
        ;
oo_obj      : .nl .slm
          ('DEFERRED_VIEW' .sp ID .sp 'for' .sp (';' .nl .slm)*)?
          'object' .sp ID .sp 'is' .sp ('a'|'an') .sp ID .nl .slm .lm(+3)
              oo_body*
          .slm .lm(-3) .slm ('end_object'|'CD_end_object') .nl .nl .slm
        ;
oo_enulst   : .nl .slm
          ('enum_list'|'Enumeration_List') .nl .slm .lm(+3)
             (.slm 'define' (.sp label)+ .slm)+
          .slm .lm(-3) .slm ('End_Enumeration_List'|'End_Enum_List') .nl .nl .slm
        ;
oo_procedure    : .nl .slm 'procedure' (.sp 'set')? .sp ID (.sp ('global'|('for' .sp ID)))?
(.sp type_var .sp ID)* (.sp 'returns' .sp type_var)? .nl .slm .lm(+3) .slm
              oo_procbody*
          .slm .lm(-3) .slm 'end_Procedure' .nl .nl .slm
        ;
oo_function : .nl .slm 'function' (.sp 'set')? .sp ID .sp (.sp ('global'|('for' .sp ID)))?
(.sp type_var .sp ID)* (.sp 'returns' .sp type_var) .nl .slm .lm(+3) .slm
             oo_procbody*
          .slm  .lm(-3) .slm 'end_function' .nl .nl  .slm
        ;
com_state   : .slm var_decl  .slm
        | .slm com_comp  .slm
        | .slm (indicador  .sp)? com_line .slm
        | .slm oo_com  .slm
        | .slm coment
;
/* OO */
oo_com      : oo_property
        | oo_mensage
        ;
oo_property : 'property' .sp type_var .sp ID ( .sp ('public'|'private'))?  ( .sp label)?
        ;
oo_mensage  : 'Forward' .sp oo_mens
        | 'Broadcast' ( .sp ('Recursive'|'Recursive_Up'))? ( .sp 'No_Stop')? .sp oo_mens
```

```
        | 'Delegate' .sp oo_mens
        | oo_mens
        ;
/* falta tratar corretamente as mesnsagens */
oo_mens     : 'Get' (.sp label)+
        | 'set' (.sp label)+
        | 'Send'(.sp label)+
        ;
/*----Orientacao a Objetos-----*/
com_uims    : 'Register_Object' (.sp label)+
        | 'Register_Procedure' (.sp label)+
        | 'Register_Function' (.sp label)+
        | 'Set_Date_Attribute' (.sp label)+
        | 'start_ui'
        | 'ui_accept'
        | 'on_key' .sp f_key .sp 'Send' (.sp label)+
        | 'Procedure_Return' (.sp label)+
        | 'Function_Return' (.sp label)+
        | 'activate_view' .sp ID .sp 'for' .sp ID
        | 'entry_item' (.sp label)+
        ;
f_key       : label
        | (label '+' label)
        ;
/* TELAS */
screen      : '/' ID  screen_body+ (FIMTEL|screen)
        ;
screen_body     : IDTUDO
        | ID
        | LBROT
        | janela
        ;

/*      | LBROT
        | janela
        | ':' | '|' | '+' | '-' | '.' | '(' | ')' | '*' | '>' | '<'
        ;
*/
janela      : jan_int
        | jan_num
        | jan_rea
        | jan_dat
        | jan_str
        ;
/* estas regras estao gerando um shift/reduce
   por causa do ponto existente em ambas--------*/
jan_int     : JID '.' .sp
        ;
jan_num     : JID '.' JID .sp
        ;
/*-----------------------------------------------------*/
jan_rea     : JID 'e' JID .sp
        ;
jan_dat     : '__/__/__' '__'? .sp
        ;
/*
retirado por causa de dois shift/reduce
        | ('__/__/____') .sp
        | ('_')+ ('/') ('_')+ ('/')? ('_')+
*/
jan_str     : JID
        | JID '@' JID
        ;
/* DECLARACAO DE VARIAVEIS */
var_decl    : 'GLOBAL'? .sp type_var (.sp var_id (.sp num)?)+ .nl
        | 'LOCAL'   .sp type_var (.sp var_id)+ .nl
        ;
type_var    : 'DATE'
        | 'INDICATOR'
        | 'INTEGER'
        | 'NUMBER'
        | 'REAL'
        | 'STRING'
        ;
var_id      : ID
        ;
/* COMANDOS DE COMPILACAO */
```

```
com_comp    : c_include
        | '#chksubn' .sp label
        | '#replace' .sp label .sp label
        | '#command' .sp label
        | '#rem' .sp label
        | '#set' .sp label
        | '#if' .sp label
        | '#ifdef' label .nl (.sp statements .nl)* ('#else' .nl (.sp statements .nl)*)?
'#endif' .nl
        | '#iftype' .sp label
        | '#if2type' .sp label
        | '#ifclass' .sp label
        | '#ifsame' .sp label
        | '#ifind' .sp label
        | '#check' .sp label
        | '#noisy' .sp label
        | '#keyproc' .sp label
        | '#fref' .sp label
        | '#revision' .sp label
        | '#compat' .sp label
        | '#num' .sp label
        | '#real' .sp label
        | '#str' .sp label
        | '#data' .sp label
        | '#pop' .sp label
        | '#push' .sp label
        | '#xpop' .sp label
        | '#xpush' .sp label
        | '#dfpush' .sp label
        | '#stksym' .sp label
        | '#spush' .sp label
        | '#spop' .sp label
        | '#error' .sp label
        | '#freg' .sp label
        | '#entopt' .sp label
        | '#ifsub' .sp label
        | '!set' .sp label
        | '!$' .sp label
        ;
c_include   : '#include' .sp ID (('.')? ID)?
        ;
/* GRUPOS DE LINHAS DE COMANDO */
com_line    : com_procarg
        | com_io
        | com_cotrole
        | com_bd
        | com_enter
        | com_form
        | com_indic
        | com_tecla
        | com_mat
        | com_mu
        | com_rel
        | com_seq
        | com_stri
        | com_estr
        | com_sist
        | nao_tratado
        | com_uims
        ;
/*-----------------------------------------------------------------*/
nao_tratado : new_com .sp (label .sp)+
        ;
new_com     : '$$comando_nao_tratado$$'
        ;
/*----Processamento de argumentos-----*/
com_procarg     : 'calculate' (.sp label)*
        | 'calc' (.sp label)*
        | 'move' .sp label .sp 'to' .sp label
        | 'movedate' .sp label .sp 'to' .sp label
        | 'moveint' .sp label .sp 'to' .sp label
        | 'movenum' .sp label .sp 'to' .sp label
        | 'movereal' .sp label .sp 'to' .sp label
        | 'movestr' .sp label .sp 'to' .sp label
        ;
/*-----------------------------------------------------------------*/
/*----Entrada e saida padrao-----*/
```

```
com_io      : 'clearscreen' (.sp label)*
        | 'clearxy' (.sp label)*
        | 'gotoxy' (.sp label)*
        | 'inkey' (.sp label)*
        | 'inkey$' (.sp label)*
        | 'input' (.sp label)*
        | 'keycheck' (.sp label)*
        | 'screenmode' (.sp label)*
        | 'show' (.sp label)*
        | 'showln' (.sp label)*
        ;
/*-----------------------------------------------------------------*/
/*----Controle-----*/
com_cotrole     : 'abort' (.sp label)*
        | 'call' (.sp label)*
        | 'chain' (.sp label)*
        | 'clearwarning' (.sp label)*
        | 'debug' (.sp label)*
        | 'error' (.sp label)*
        | 'gosub' (.sp label)*
        | 'on_gosub' (.sp label)*
        | 'on_goto' (.sp label)*
        | 'return' (.sp label)*
        | 'system' (.sp label)*
        ;
/*-----------------------------------------------------------------*/
/*----Banco de Dados-----*/
com_bd      : 'attach' (.sp label)*
        | 'clear' (.sp label)*
        | 'close' (.sp label)*
        | 'constrained_clear' (.sp label)*
        | 'constrained_find' (.sp label)*
        | 'constraint' (.sp label)*
        | 'constraint_set' (.sp label)*
        | 'constraint_validate' (.sp label)*
        | 'delete' (.sp label)*
        | 'find' (.sp label)*
        | 'open' (.sp label)*
        | 'relate' (.sp label)*
        | 'save' (.sp label)*
        | 'saverecord' (.sp label)*
        | 'set_relate' (.sp label)*
        | 'vfind' (.sp label)*
        | 'zerofile' (.sp label)*
        ;
/*-----------------------------------------------------------------*/
/*----Macro Enter-----*/
com_enter   : 'entergroup' (.sp label)+
        | 'endgroup'
        | 'enter' (.sp ID)+
        | 'enterend'
        | 'entdisplay' (.sp label)*
        | 'entry' (.sp label)+
        ;
/*-----------------------------------------------------------------*/
/*----Formularios-----*/
com_form    : 'accept' (.sp label)*
        | 'autopage' .sp ID (.sp num)?
        | 'blankform' (.sp label)*
        | 'clear_option' (.sp label)*
        | 'clearform' (.sp label)*
        | 'display' (.sp label)*
        | 'format' (.sp label)*
        | 'ifchange' (.sp label)*
        | 'output' (.sp label)*
        | 'output_wrap' (.sp label)*
        | 'page' (.sp label)*
        | 'print' (.sp label)*
        | 'print_wrap' (.sp label)*
        | 'set_option' (.sp label)*
        | 'setchange' (.sp label)*
        | 'subtotal' (.sp label)*
        | 'name' (.sp c_name)*
        ;
c_name      : label
        ;
/*-----------------------------------------------------------------*/
```

```
/*----indicadores de fluxo-----*/
com_indic   :  'if' .sp label (.sp oper_rel .sp label)? .sp  (';' .nl .slm)* com_state
        |  'indicate' (.sp label)*
        |  'else' .sp (';' .nl .slm)* com_state
        ;
/*---------------------------------------------------------------*/
/*----Tecla-----*/
com_tecla   : 'backfield' (.sp label)*
        | 'entagain' (.sp label)*
        | 'keyproc' (.sp label)*
        ;
/*---------------------------------------------------------------*/
/*----Matematica-----*/
com_mat     : 'abs' (.sp label)*
        | 'acos' (.sp label)*
        | 'asin' (.sp label)*
        | 'atan' (.sp label)*
        | 'cos' (.sp label)*
        | 'decrement' (.sp label)*
        | 'exp' (.sp label)*
        | 'hi' (.sp label)*
        | 'increment' (.sp label)*
        | 'log' (.sp label)*
        | 'low' (.sp label)*
        | 'mod' (.sp label)*
        | 'random' (.sp label)*
        | 'round' (.sp label)*
        | 'sin' (.sp label)*
        | 'sqrt' (.sp label)*
        | 'tan' (.sp label)*
        ;
/*---------------------------------------------------------------*/
/*----Controle muiti-usuario-----*/
com_mu      : 'despool' (.sp label)*
        | 'reread' (.sp label)*
        | 'unlock' (.sp label)*
        ;
/*---------------------------------------------------------------*/
/*----Reletorios-----*/
com_rel     : 'formfeed' (.sp label)*
        | 'pagecheck' (.sp label)*
        | 'report' (.sp label)+
        | 'section' (.sp label)*
        | 'reportend'
        ;
/*---------------------------------------------------------------*/
/*----Entrada e Saida Sequencial-----*/
com_seq     : 'append_output' (.sp label)*
        | 'close_input' (.sp label)*
        | 'close_output' (.sp label)*
        | 'direct_input' (.sp label)*
        | 'direct_output' (.sp label)*
        | 'get_channel_position' (.sp label)*
        | 'read' (.sp label)*
        | 'read_block' (.sp label)*
        | 'readln' (.sp label)*
        | 'set_channel_position' (.sp label)*
        | 'write' (.sp label)*
        | 'writeln' (.sp label)*
        ;
/*---------------------------------------------------------------*/
/*----String-----*/
com_stri    : 'append' (.sp label)+
        | 'ascii' (.sp label)*
        | 'center' (.sp label)*
        | 'character' (.sp label)*
        | 'eval' (.sp label)*
        | 'insert' (.sp label)*
        | 'left' (.sp label)*
        | 'legth' (.sp label)*
        | 'lowercase' (.sp label)*
        | 'ltrim' (.sp label)*
        | 'mid' .sp label .sp 'to' .sp label .sp label .sp label
        | 'overstrike' (.sp label)*
        | 'pad' (.sp label)*
        | 'pos' (.sp label)*
        | 'remove' (.sp label)*
```

```
        | 'replace' (.sp label)*
        | 'replaces' (.sp label)*
        | 'right' (.sp label)*
        | 'rtrim' (.sp label)*
        | 'trim' (.sp label)*
        | 'uppercase' (.sp label)*
        ;
/*-----------------------------------------------------------------*/
/*----Controle Estruturado-----*/
com_estr    : 'begin'
        | 'for' .sp label .sp 'from' .sp label .sp 'to' .sp label
        | 'while' .sp label (.sp oper_rel .sp label)?
        | 'repeat'
        | 'loop'
        | 'end'
        | 'until'.sp label .sp oper_rel .sp label
        | c_rotina
        ;
c_rotina    : LBROT
        ;
/* se colocar a subrotina normal esta gerandoi muita ambiguidade */
/* este seria o formato correto da sub rotina*/
/*c_sub_rot : label ':'*/
/*-----------------------------------------------------------------*/
/*----Sistema-----*/
com_sist    : 'cmdline' (.sp label)*
        | 'copyfile' (.sp label)*
        | 'diretory' (.sp label)*
        | 'erasefile' (.sp label)*
        | 'field_def' (.sp label)*
        | 'file_mode' (.sp label)*
        | 'file_size' (.sp label)*
        | 'filelist' (.sp label)*
        | 'get_argument_size' (.sp label)*
        | 'get_current_diretory' (.sp label)*
        | 'get_environment' (.sp label)*
        | 'index_def' (.sp label)*
        | 'make_file' (.sp label)*
        | 'memory' (.sp label)*
        | 'read_dfini' (.sp label)*
        | 'registration' (.sp label)*
        | 'renamefile' (.sp label)*
        | 'runprogram' (.sp label)*
        | 'set_argument_size' (.sp label)*
        | 'set_dfpath' (.sp label)*
        | 'set_termlist' (.sp label)*
        | 'sleep' (.sp label)*
        | 'sysdate' (.sp label)*
        | 'sysdate4' (.sp label)*
        | 'use' (.sp label)*
        ;
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/* ESTRUTURAS DE APOIO */
oper_rel    : 'EQ' | 'NE' | 'LE' | 'LT' | 'GE' | 'GT' | 'IN'
        ;
oper_exp    : .sp '=' .sp  | .sp '<>' .sp | .sp '<=' .sp | .sp '<' .sp | .sp '>=' .sp | .sp
'>' .sp
        ;
oper_calc   : '+' | '-' | '*' | '/' | '^'
        ;
oper_log    : 'AND'
        | 'OR'
        | 'NOT'
        | '~'
        ;
indicador   : ('[' (('~'|'NOT')? .sp label .sp)+ ']')+
        ;
label       : ID
        | NUM
        | STRI
        | camp_arq
        | expressao
        | exp_entry
        | 'to'
        ;
label2      : ID
```

```
        | NUM
        | STRI
        | camp_arq
        ;
oper_all    : oper_exp
        | oper_calc
        | oper_log+
        ;
expressao   : '(' label2  ')'
        | '(' (oper_all (expressao|label2))+ ')'
        | '(' ((expressao|label2) oper_all?)+ (expressao|label2) ')'
        | '(' oper_all? expressao ')'
        ;
/*
        | '(' (ID|NUM) (oper_exp|oper_calc|oper_log)? expressao+ ')'
        | '(' expressao+ (oper_exp|oper_calc|oper_log)? (ID|NUM) ')'
*/
exp_entry   : '{' label (',' label)* '}'
        ;
camp_arq    : ID '.' ID
        ;
num     : NUM
        | 'E' NUM
        ;
coment      :  COMENT
     ;
/* ESTRUTURAS TERMINAIS */
COMENT      : "//"[^\n]*\n
        ;
STRI        : \'([^\']|\'\')*\'
        ;
STRI        : \"([^\"]|\"\")*\"
        ;
NUM     : [\+\-]?[0-9]+(\.[0-9]+)?
        ;
JID     : [\_]+
        ;
ID      : [A-Za-z_\@\#\$\%\&\?]+[A-Za-z\-\@\#\$\%\&\?\.]*[A-Za-z0-9\$\&]*
        ;
IDTUDO      : [^ A-Za-z_\@\#\$\%\&\?\_\t\n\(\)]+
        ;
LBROT       : [A-Za-z_\@\#\$\%\&\?]+[A-Za-z\-\@\#\$\%\&\?\.]*[A-Za-z0-9]+\:
        ;
FIMTEL      : "/*"[ ]*\n
        ;
IGNORE      : [ \t\n]
        ;
%%
```