Reengenharia Orientada a Objetos de Código Legado *Progress 4GL*

Antonio Francisco do Prado - prado@dc.ufscar.br
Edimilson Ricardo Azevedo Novais - novais@dc.ufscar.br
Universidade Federal de São Carlos
Rodovia Washington Luís (SP-310), Km. 235
São Carlos – São Paulo – Brasil – CEP: 13565-905

Resumo

Este artigo apresenta uma estratégia para reconstrução de sistemas legados escritos na linguagem *Progress 4GL*. A estratégia tem 4 passos. No passo **Organizar Sistema**, organiza-se o código legado segundo os princípios da Orientação a Objetos. É um passo preparatório para facilitar a transformação de um código procedural para orientado a objetos. O código procedural organizado permanece na mesma linguagem *Progress 4GL*. No passo **Transformar Sistema**, parte do código que está organizado segundo as idéias de classes, atributos e protótipos de métodos é transformado para a linguagem de modelagem UML. Para a linguagem Java são transformadas as partes do código organizado, correspondentes aos procedimentos, candidatos a métodos de uma classe. No passo seguinte, **Reprojetar Sistema**, usando uma ferramenta CASE, obtêm-se o projeto atual do sistema, modelado segundo os princípios da orientação a objetos. Ainda neste passo pode-se reespecificar o sistema gerando um novo projeto orientado a objetos atualizado. Finalmente, no passo **Reimplementar Sistema**, as especificações em UML, do sistema reprojetado, são também transformadas para Java e integradas com o código Java dos métodos, obtendo-se a implementação final do sistema.

Palavras chaves: Reengenharia de Software, Engenharia Reversa, Orientação a Objetos e Sistema de Transformação.

Abstract

This paper presents a strategy for reconstruction of legacy systems written in Progress 4GL language. The strategy has 4 steps. In the step **to Organize System**, the legacy code is organized according to the principles of the object orientation. It is a preparatory step to facilitate the transformation of a procedural code into an object-oriented. The organized procedural code stay in the same Progress 4GL language. In the step **to Transform System**, part of the code that is organized according to the ideas of classes, attributes and methods prototypes is transformed into modeling UML language. For Java language the parts of organized code are transformed, corresponding to the procedures, candidates to methods of a class. In the next step, **to Re-project System**, using a CASE tool, obtain the current system's project, modeled according to the principles of the object orientation. Still in this step the system can be redesign generating an up to date object-oriented project. Finally, in the step **to Re-implement System**, the specifications in UML of the re-projected system, are also transformed into Java and integrated with the Java code of the methods, getting the final implementation of system.

Key Words: Software Re-engineering, Reverse Engineering, Object Oriented and Transformation System.

1. Introdução

Uma das áreas da Engenharia de Software que vem se destacando é a Reengenharia de código legado. A dificuldade em atualizar os softwares para atenderem a novos requisitos e serem executados em novas plataformas de hardware e software, tem motivado os pequisadores a investigar novas soluções para diminuir os custos e facilitar a manutenção [Rea92, Cor93, Wid93, Bax97, Faq98, Boy99].

O objetivo da reengenharia de software é manter o conhecimento adquirido com os sistemas legados e utilizar estes conhecimentos como base para a evolução contínua e estruturada do software. O código legado possui uma lógica de programação, decisões de projeto, requisitos do usuário e regras de negócio que podem ser recuperados e reconstruídos sem perda da semântica. O software é reconstruído com inovações tecnológicas e novos requisitos podem ser adicionados para atender prazos, custos, correções de erros e melhorias de desempenho.

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

A indústria de software sente necessidade de aperfeiçoar o ciclo de vida de seus sistemas. O processo de desenvolvimento e operacionalização de um produto passa por 3 fases básicas: criação, estabilização e manutenção. Na fase de criação o sistema é analisado, projetado e implementado; na fase de estabilização podem ser incluídas novas funcionalidades para atender os requisitos não funcionais do sistema e finalmente na fase de manutenção o sistema pode ser alterado para corrigir falhas, atender novos requisitos ou melhorar seu desempenho. Nesta última fase, muitas vezes o sistema fica comprometido por manutenções estruturais não previstas no projeto original.

Um sistema comercial, como é o caso da maioria dos sistemas implementados em *Progress 4GL*, leva de um a dois anos para se estabilizar e atender o mínimo dos requisitos necessários. É grande a quantidade de sistemas implementados, inclusive em versões antigas da linguagem *Progress 4GL*, com interface caracter, cuja reengenharia é importante para atualizá-los para novas plataformas de hardware e software. A Linguagem *Progress 4GL* é uma linguagem de 4ª geração orientada a procedimentos, desenvolvida pela *Progress Software Corporation* (PSC), intimamente ligada ao Banco de Dados *Progress*, com grande uso em aplicações no mercado mundial. Possui os comandos básicos de uma linguagem de consulta a banco de dados com transações atômicas e *triggers* de eventos. Por ser orientada a procedimentos e pela grande quantidade de sistemas implementados, que são utilizados por médias e grandes empresas, a linguagem *Progress 4GL* foi escolhida, como linguagem do código fonte legado, para validar a estratégia de reengenharia proposta.

Motivados por estas idéias pesquisou-se uma estratégia de reengenharia cujo objetivo é recuperar o modelo de análise de sistemas legados e reconstruí-los para serem executados em novas plataformas de hardware e software. Partindo do código legado *Progress 4GL*, o desenvolvedor pode organizar o código segundo os princípios da Orientação a Objetos e obter o projeto atual do sistema em UML. Em seguida pode-se reprojetar o sistema para atender novos requisitos e finalmente pode-se reimplementar o sistema numa linguagem orientada a objetos como Java. A estratégia suporta a manutenção do sistema, inclusive com alteração da sua estrutura original, garantindo sua evolução.

O artigo está organizado da seguinte forma: a seção 2 apresenta as principais técnicas usadas na estratégia de reengenharia, a seção 3 descreve a estratégia de Reengenharia Orientada a Objetos de Código Legado *Progress 4GL*, a seção 4 apresenta um estudo de caso como exemplo de uso da estratégia e finalmente, a seção 5 apresenta as conclusões desta pesquisa.

2. Principais Técnicas da Reengenharia de Software

Uma das principais técnicas usadas na Reengenharia de Software é de reconstrução do software, usando transformações. Diferentes Sistemas de Transformação têm sido usados destacando-se o Tampr [Boy89], Refine [Rea92], Popart [Wid93], TXL [Cor93], DMS [Bax97] e RescueWare [Faq98]. Outro importante sistema de transformação que vem sendo utilizado nesta área é o Draco-PUC, como pode ser constatado em [Lei91, Lei94, Pra92, San93, Pra98, Abr99].

O Draco-PUC [Nei84, Lei91, Pra92, Lei94] implementa as idéias de transformação de software orientada a domínios. Pela estratégia proposta por Prado [Pra92], é possível a reconstrução de um software pelo "porte" direto do código fonte de uma linguagem para linguagens de outros domínios. Um domínio, de acordo com o paradigma Draco, é constituído de três partes: uma **linguagem**, definida por um *parser* responsável por analisar os programas da linguagem e gerar a representação interna do Draco-PUC, uma *Abstract Syntax Tree (AST)*, denominada no Draco-PUC de DAST; um ***prettyprinter*** ou *unparser*, que faz a formatação da DAST, tornando-a novamente textual na linguagem do domínio; e um ou mais **Transformadores**, que mapeiam estruturas de uma linguagem para estruturas na mesma linguagem do domínio, chamados de transformadores Intra-Domínio, e os que mapeiam as aplicações descritas na linguagem de um domínio para descrições de uma linguagem de outro domínio, chamados transformadores Inter-Domínios. Os transformadores são responsáveis pela automatização ou semi-automatização do processo de construção de software.

Um transformador possui um conjunto de transformações. Uma transformação é composta basicamente pelos pontos de controle LHS (*Left Hand Side*) e RHS (*Right Hand Side*). O LHS, ou padrão de reconhecimento, define a sintaxe da linguagem fonte para a transformação e o RHS (*Right Hand Side*), ou padrão de substituição, define a sintaxe da linguagem alvo para a transformação. Além dos padrões de reconhecimento e de substituição, uma transformação pode conter outros pontos de controle, aos quais pode-se associar código para o desempenho de tarefas relacionadas com pré e pós-condições da transformação. As substituições utilizam *Templates*, padrões formatados de uma dada categoria sintática, para padronizar a escrita do código alvo [Jes99].

A técnica de engenharia reversa Fusion/RE [Pen96] é utilizada para obter o entendimento e revitalizar a estrutura do código legado segundo as idéias do paradigma de orientação a objetos, visando reutilizar toda a funcionalidade do código legado na reconstrução do sistema. A estratégia apresentada neste artigo usa apenas parte da abordagem Fusion/RE para recuperar o modelo de objetos do Modelo de Análise do Sistema Atual (MASA), com suas classes, elementos de dados, procedimentos e relacionamentos, que originalmente não estão orientado a objetos, e a partir daí, elaborar o modelo de objetos do Modelo de Análise do Sistema (MAS), corrigindo as anomalias dos procedimentos. Para recuperar o projeto, consulta-se o banco de dados do sistema para obter todas as entidades, relacionamentos e chaves das relações, que definem o Modelo de Entidades e Relacionamentos (MER) do sistema. Como no modelo relacional os dados são armazenados em tabelas, a duplicação dos dados de uma coluna é um indicativo para identificação da chave de um relacionamento. Uma chave candidata pode ser usada para obter as informações para recuperar as entidades e seus relacionamentos. A partir do MER, pode-se obter parte do modelo de objetos, com as cardinalidades e respectivos papéis de objetos. Do código, levantam-se as estruturas de dados relevantes, globais ou locais. Baseado no modelo de objetos inicial recuperam-se os procedimentos que atuam sobre as estruturas de dados lógicas e documentam-se os fatos obtidos. Os procedimentos do código legado são organizados em torno de possíveis classes, corrigindo-se anomalias, como por exemplo, a de um

procedimento pertencer a mais de uma classe candidata. Os procedimentos são candidatos a métodos das classes e são classificados em construtores, quando alteram a estrutura de dados, e observadores, quando somente consultam as estruturas de dados. Existem também procedimentos dependentes da implementação, como por exemplo, os responsáveis pela interface, pelo acesso ao banco de dados e de uma forma geral os que implementam os requisitos não funcionais do sistema.

Além destas técnicas, vem ganhando destaque o uso de ferramentas CASE no projeto ou reprojeto de sistemas a serem reconstruídos. Uma das ferramentas CASE conhecida é a *Rational Rose* [Rat98] que suporta a especificação do sistema em UML [UML97, UML98] e a geração parcial de código em uma linguagem executável.

Combinando estas diferentes técnicas de Engenharia Reversa Fusion/RE, o sistema de transformação Draco-PUC e a ferramenta CASE *Rational Rose*, definiu-se uma estratégia para a Reengenharia Orientada a Objetos de Código Legado *Progress 4GL*, que será apresentada a seguir.

3. Reengenharia Orientada a Objetos de Código Legado *Progress 4GL*

A estratégia de Reengenharia Orientada a Objetos de Código Legado *Progress 4GL* é realizada em quatro passos: **Organizar, Transformar, Reprojetar** e **Reimplementar Sistema**, como mostra a Figura 1 [Pra92].

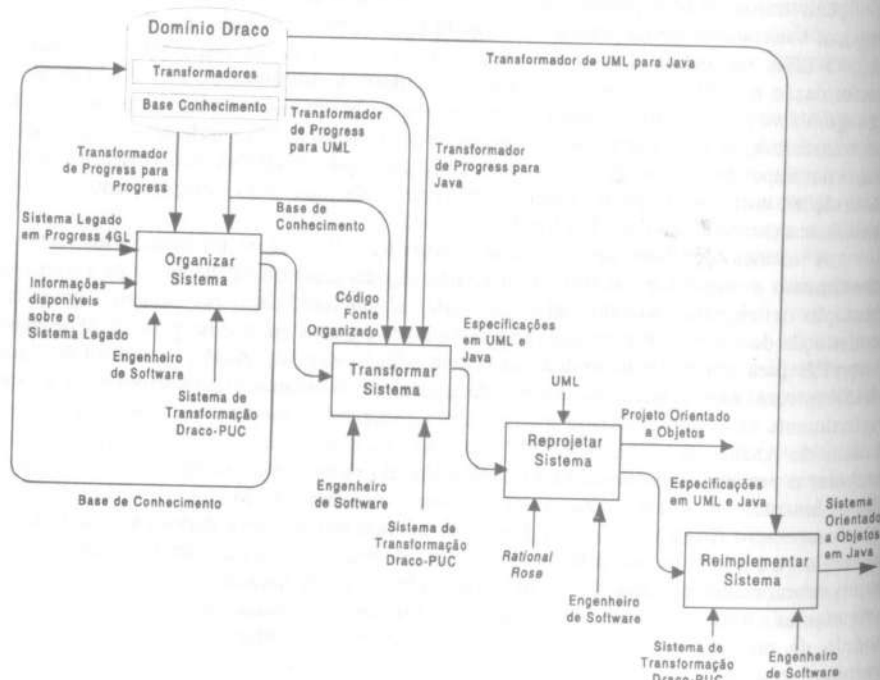

Figura 1 – Estratégia de Reengenharia Orientada a Objetos de Código Legado *Progress 4GL*

A entrada desta estratégia de Reengenharia é o código legado do sistema em *Progress 4GL*. As informações disponíveis sobre o sistema legado são armazenadas em uma base de conhecimento para auxiliar a organização do código. No passo **Organizar Sistema**, o código legado é organizado segundo os princípios da orientação a objetos. No passo **Transformar Sistema**, o código organizado segundo as idéias de classes e objetos, com seus atributos e protótipos de métodos, é transformado para os Diagramas de Classes da UML. Ainda neste passo, o comportamento dos métodos das classes, obtidos dos procedimentos do código legado, são transformados diretamente em Java. No passo **Reprojetar Sistema**, o engenheiro de software pode alterar o projeto original, obtido do código fonte organizado, utilizando uma ferramenta CASE, no caso a *Rational Rose*. Finalmente, no passo **Reimplementar Sistema**, as especificações em UML, do sistema reprojetado, são transformadas para Java, integrando com o código Java dos métodos. Obtém-se assim a implementação orientada a objetos do sistema reconstruído. A seguir são apresentados cada um dos passos desta estratégia de Reengenharia.

3.1 Organizar Sistema

Neste passo faz-se a organização do código legado *Progress 4GL* segundo os princípios da orientação a objetos. O código legado, escrito de forma procedural, possui comandos e declarações que podem ser organizados, sem prejuízo da sua lógica e semântica, de forma a facilitar sua transformação para o paradigma orientado a objetos, que tem a classe como a unidade básica.

Para suportar este passo foi construído o domínio *Progress* no sistema de transformação Draco-PUC. Baseado na gramática livre de contexto do *Progress 4GL* gerou-se o *parser* e o *prettyprinter*. O *parser* foi gerado a partir da definição das regras da gramática *Progress 4GL*, utilizando o gerador *Pargen* do Draco-PUC. O *prettyprinter* foi gerado automaticamente pelo subsistema *Ppgen* do Draco-PUC, a partir das definições da gramática. A figura 2 mostra parte da gramática *Progress 4GL*.

```
foreach_stat : 'FOR EACH' .sp iden .sp (expr .sp)? lock_opt?  ':'
                statements* end_proc                              ;
find_stat    :  'FIND' .sp iden .sp select_opt?
                lock_opt? (.sp 'NO-ERROR')?  end_proc              ;
display_stat : 'DISPLAY' (preprocessor | object_set) .sp frame_opt?
                end_proc                                          ;
object_set   : .sp object+                                        ;
create_stat  : 'CREATE' .sp iden end_proc                          ;
atrib_stat   : expr end_proc                                      ;
assign_stat  : 'ASSIGN' (preprocessor | (.sp expr)+) frame_opt?
               end_proc                                           ;
...
```

Figura 2 – Parte da gramática *Progress 4GL*

As transformações são escritas no Draco-PUC, com base nas regras gramaticais das linguagens fonte e alvo das transformações. O sistema de transformação Draco-PUC dispõe de uma linguagem própria para facilitar a escrita das transformações.

Para auxiliar o processo das transformações que necessitam concentrar ou espalhar comandos ao longo do código fonte organizado, utilizou-se uma Base de Conhecimento (*Knowledge Base*), que permite armazenar fatos e regras para consultas posteriores. Este é um recurso disponível no Draco-PUC, utilizado quando se faz necessário adiar decisões durante a aplicação das transformações. Os principais comandos para atualização da Base de Conhecimento são: *KBSolve*, que busca por um fato; *KBRetrieve*, que recupera o fato;

KBDelete, que elimina o fato; *KBAssertifNew*, que armazena o fato; e *KBWrite*, que grava as modificações. Neste passo, a Base de Conhecimento armazena fatos relacionados com as supostas classes, atributos, métodos e relacionamentos, como mostra a Figura 3. Estes fatos são extraídos da estrutura do banco de dados *Progress* por um programa auxiliar escrito em *Progress 4GL*.

```
SupostaClasse(8,CSACTCOM,"CSACTCOM",2).
SupostoAtributo(9,CSACTCOM,CUSTO_COMERCIAL,"CUSTO_COMERCIAL",Int,yes).
SupostoAtributo(10,CSACTCOM,DESC_ITEM_CUSTO,"DESC_ITEM_CUSTO",String,yes).
SupostoAtributo(11,CSACTCOM,PORCENTAGEM,"PORCENTAGEM",Float,yes).
SupostoAtributo(12,CSACTCOM,SEQUENCIA,"SEQUENCIA",Int,yes).
SupostoMetodo(13,CSACTCOM,CSACTCOM,"Construtor de CSACTCOM","").
...
```

Figura 3 – Parte da base de conhecimento com fatos sobre a estrutura de um banco de dados *Progress*

O *Workspace* é outra técnica auxiliar usada no processo de transformação. Trata-se de um espaço de trabalho temporário usado durante a organização dos procedimentos, para solucionar as diferenças semânticas entre o código procedural e o orientado a objetos [Jes99]. Por exemplo, pode-se compor procedimentos a partir dos trechos de comandos espalhados ao longo do código, como no caso dos procedimentos para acesso ao Banco de Dados. No início das transformações é criado um *Workspace* para cada suposta classe onde são colocados todos os seus procedimentos candidatos a métodos.

O transformador intra-domínio, que organiza o código legado, utiliza nos pontos de controle de reconhecimento (LHS) e de substituição (RHS), os padrões sintáticos da mesma linguagem, no caso *Progress 4GL*. Os demais pontos de controle são utilizados para armazenar e recuperar fatos da base de conhecimento que auxiliam na solução de problemas de organização do código legado.

Em resumo, a biblioteca *ProgressToProgress* do domínio *Progress 4GL*, construída no Draco-PUC para organizar o código legado, contêm transformações para:

- Agrupar as variáveis globais;
- Obter o comportamento dos procedimentos relacionados com a interface;
- Reutilizar os procedimentos comuns em várias partes do código fonte;
- Separar o código fonte em procedimentos atômicos, associados a uma única classe candidata, respeitando o escopo, dependência de dados e visibilidade do código legado;
- Reunir em um único programa fonte, todos os procedimentos candidatos a métodos de uma suposta classe; e
- Armazenar na Base de Conhecimento os supostos atributos e relacionamentos, para serem consultados no próximo passo da estratégia.

O particionamento do sistema em classes e objetos considera o Modelo Entidade-Relacionamento (MER) que deu origem ao banco de dados do sistema. Parte-se do princípio que o MER está consistente e que o banco de dados está normalizado.

Para organizar o código legado foram definidos padrões de reconhecimento do código *Progress 4GL* que são transformados em novos comandos, na mesma linguagem *Progress 4GL*, porém com as características da orientação a objetos. Destacam-se os seguintes padrões de reconhecimento:

- Suposta Classe: usado no reconhecimento dos comandos de criação de registros no banco de dados, comparando o nome da tabela do banco de dados com as supostas classes armazenadas na base de conhecimento, criada no início da organização a partir da estrutura das tabelas do banco de dados do sistema legado. As interfaces do usuário e seus procedimentos para validação de informações são reconhecidos como supostas classes de projeto;
- Suposto Atributo: usado no reconhecimento dos campos declarados nos comandos de acesso ao banco de dados, comparando-os com os supostos atributos armazenados na base de conhecimento, criada a partir da estrutura das tabelas do banco de dados do sistema legado;
- Suposto Método: usado no reconhecimento dos comandos que armazenam e recuperam informações de uma tabela do banco de dados, candidata a classe. Neste caso deve-se observar a visibilidade, o escopo e a dependência de dados. Na análise dos procedimentos que atualizam ou recuperam informações de mais de uma tabela do banco de dados, é eleita a tabela que tem mais dados alterados ou recuperados como possível classe do correspondente método deste procedimento. Os disparadores de eventos do banco de dados, *"triggers"*, armazenados como fatos na base de conhecimento, são transformados em procedimentos candidatos a métodos para criação, gravação e exclusão de objetos da classe. Os comandos de recuperação de dados, *FOR EACH e FIND*, são encapsulados em um único procedimento de leitura para cada tabela, recebendo como parâmetros as informações usadas no seu comportamento. O comando para criação de um registro na base de dados *CREATE*, gera um procedimento candidato a método construtor de uma suposta classe. Os comandos de atualização de dados *ASSIGN*, espalhados pelo código legado, são encapsulados dentro de um único procedimento de atualização de dados para cada tabela do banco de dados, candidata a uma classe.
- Suposto Relacionamento: usado no reconhecimento dos relacionamentos entre as supostas classes. Baseia-se nos relacionamentos entre as tabelas do banco de dados. Uma tabela se relaciona com outra quando os atributos que formam um índice único na tabela, também são atributos em outra tabela. Para tabelas que são relacionadas mas não possuem os mesmos atributos, é necessário analisar os comandos de leitura de informações no banco de dados para reconhecer o relacionamento.

As transformações para organização do código são executadas em duas etapas: na primeira etapa, o transformador *ProgressToProgress* reconhece o modelo de objetos MASA e tenta corrigir automaticamente as anomalias para elaborar o modelo de objetos MAS. Todas estas informações são armazenadas em uma base de conhecimento que pode ser consultada e alterada, usando uma interface amigável desenvolvida em *Progress 4GL*, pelo engenheiro de software no final da primeira etapa de transformação. Na segunda etapa usando as informações da base de conhecimento atualizada pelo engenheiro de software, o transformador finaliza as transformações para obtenção do código organizado segundo os princípios da orientação a objetos. As alterações que podem ser feitas pelo engenheiro de software visam melhorar a distribuição do comportamento do sistema procedimental, entre elas estão:

- Mudar os nomes das classes, elementos de dados e procedimentos para nomes mais significativos;
- Mudar a classificação de um procedimento: construtor, observador ou interface;
- Corrigir anomalias nos procedimentos que não foram identificadas pelo transformador;
- Mudar um procedimento de classe ou substituí-lo por outro;
- Excluir uma classe, elemento de dado, procedimento ou relacionamento;
- Mudar o tipo de um relacionamento e
- Criar uma nova classe e selecionar elementos de dados, procedimentos e

relacionamentos para esta classe;

A figura 4 mostra um exemplo de transformação intra-domínio usada na organização do código fonte legado. Do lado esquerdo tem-se em destaque o padrão de reconhecimento do comando *FIND*, que faz a leitura de um registro no banco de dados. Do lado direito tem-se o correspondente padrão de substituição, no caso *PROCEDURE FIND*. Este padrão de substituição gera um procedimento de leitura que recebe como parâmetros os atributos declarados nos comandos *FIND*, fazendo com que todos os comandos de leitura de um registro sejam substituídos por um único procedimento de leitura para cada tabela do banco de dados. Da mesma forma foram escritas as demais transformações que organizam o código legado.

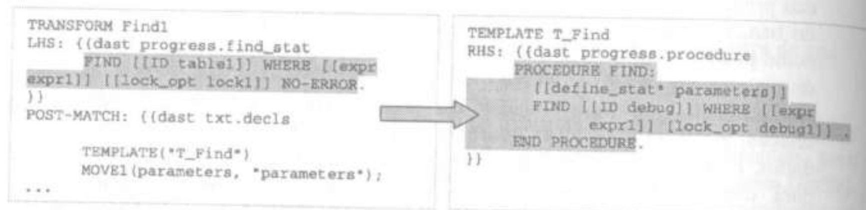

Figura 4 – Exemplo de um Padrão de Reconhecimento e Substituição para organização do código legado

A organização do código fonte legado, segundo os princípios da Orientação a Objetos, facilita a transformação para UML e Java, que é o objetivo do segundo passo da estratégia.

3.2 Transformar Sistema

Neste passo o engenheiro de software também utiliza o sistema de transformação Draco-PUC para gerar automaticamente as especificações nas linguagens UML e Java.

Para suportar este passo foram construídas as bibliotecas de transformação *ProgressToUML* e *ProgressToJava* do domínio *Progress 4GL*. Estas bibliotecas contêm as transformações que reconhecem padrões sintáticos e semânticos do código fonte legado e substituem pelo seus correspondentes padrões sintáticos e semânticos, definidos na linguagem alvo da transformação, no caso UML e Java.

A base de conhecimento obtida no passo anterior, Organizar Sistema, é também consultada neste passo para a criação das classes, atributos, protótipos de métodos e relacionamentos entre as classes. O fluxo de controle do sistema legado é mantido no código orientado a objeto por meio de conexões de mensagem entre os objetos.

Parte do código *Progress 4GL* organizado segundo classes, com seus atributos e protótipos de métodos, é transformado para especificações na linguagem de modelagem UML. Para persistência das especificações UML usa-se a mesma linguagem da ferramenta CASE *Rational Rose*, que armazena as especificações em arquivos com extensão ".MDL".

Em UML as miniespecificações dos métodos das classes podem ser descritas através de pré e pós-condições e semântica. Dado o conhecimento que grande parte dos desenvolvedores têm sobre linguagens de programação, decidiu-se pelo "porte" direto do código legado dos corpos dos procedimentos *Progress 4GL* para a linguagem Java. Assim, o transformador *ProgressToJava*, foi construído para transformar os códigos dos procedimentos *Progress4GL* em código Java. O código Java gerado fica embutido na UML, na especificação da semântica de cada método de uma classe. A integração das duas linguagens, UML e Java,

foi possível porque o Draco-PUC suporta descrições em múltiplas linguagens.

A figura 5 apresenta um exemplo de transformação da biblioteca *ProgressToUML*. Do lado esquerdo, tem-se em destaque o padrão de reconhecimento *Procedure*, que define um procedimento na linguagem origem *Progress 4GL*. Do lado direito, tem-se o correspondente padrão de substituição *Operations*, que define um método na linguagem alvo UML.

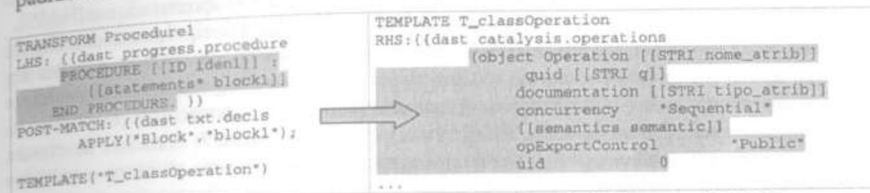

Figura 5 –Transformação Inter-Domínio *ProgressToUML*

A figura 6 apresenta um exemplo de transformação da biblioteca *ProgressToJava*. Do lado esquerdo tem-se em destaque o padrão de reconhecimento *Run* que executa um procedimento na linguagem origem *Progress 4GL*. Do lado direito, tem-se o correspondente padrão *Expression*, que descreve a chamada a um método na linguagem alvo Java.

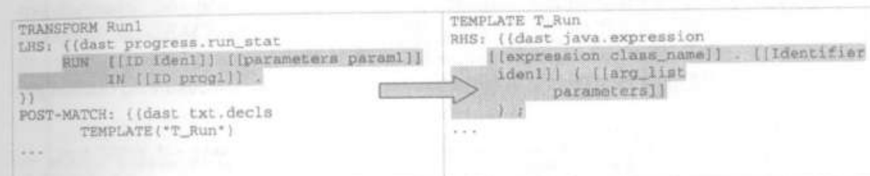

Figura 6 –Transformação Inter-domínio *ProgressToJava*

Em resumo, neste passo as especificações UML são usadas para representar a parte estática das classes, com seus atributos e protótipos de métodos. Os procedimentos em *Progress 4GL*, candidatos a métodos das classes, são portados diretamente para a linguagem Java.

3.3 Reprojetar Sistema

No terceiro passo da estratégia de reengenharia, Reprojetar Sistema, o Engenheiro de Software, usando a ferramenta CASE *Rational Rose*, importa as especificações na linguagem UML para obter o projeto do sistema atual. Nas especificações UML, a semântica dos métodos em cada classe está descrita diretamente em Java. Dessa forma o código Java que implementa os corpos dos métodos fica embutido na especificação UML. Com o projeto do sistema recuperado em UML, é possível compreendê-lo e reprojetá-lo. Tanto o projeto como a implementação dos métodos, estão disponíveis na ferramenta CASE para o reprojeto, possibilitando a alteração do sistema para atender novas especificações de requisitos, caso seja necessário.

3.4 Reimplementar Sistema

Finalmente, no último passo da estratégia, faz-se a reimplementação final do sistema numa linguagem orientada a objetos, no caso Java. Este passo usa um Transformador Inter-

Domínios já existente no sistema de transformação Draco-PUC, objeto de outra pesquisa [Fuk99]. O sistema de transformação Draco-PUC é utilizado pelo engenheiro de software para transformar automaticamente para Java, os Diagramas de Classes do sistema projetado, com seus atributos e protótipos de métodos especificados em UML. O código Java gerado neste passo é integrado com o código Java dos métodos, inseridos na parte semântica das especificações UML, obtendo-se a implementação final do sistema, conforme apresentado em [Fuk99].

Com o objetivo de facilitar o processo de transformação, principalmente das classes que tratam funções de entrada e saída do *Progress 4GL*, foram criadas classes padrões em Java cujos métodos expressam a mesma semântica dos comandos *Progress 4GL*. Estas classes ficam em um pacote denominado *Progress* e usam comandos SQL para acessar um banco de dados conectado pelo protocolo JDBC/OBDC, como mostra a figura 7. Trata-se de classes abstratas que não são instanciadas, servindo apenas para reuso dos seus atributos e métodos através do princípio da herança.

```
progress

Package progress;
import java.sql.*;
import java.util.Vector;

abstract public class DataBase (
public static void initDataBase( String nameODBC, String user, String pwd ) (
        String url    = "jdbc:odbc:"+nameODBC;
        try (   Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
                //DriverManager.setLogStream(System.out);
                con = DriverManager.getConnection ( url, user, pwd );
                stmt = com.createStatement ();
                isConect = true;
...
public void find (String exp) (
        try  ( rs = Progress.stmt.executeQuery("SELECT * FROM " + table + " WHERE " +
                        currentIndexLabel + "=" + exp);
...
```

Figura 7 – Package *Progress* para persistência de objetos

4. Estudo de Caso

A estratégia proposta foi aplicada em diferentes estudos de caso, destacando-se um sistema verídico do tipo *Enterprise Resource Planning* (ERP), para gestão empresarial de indústrias. Os seguintes módulos fazem parte do sistema:

- Básico: que contém a estrutura básica do sistema e as informações comuns aos demais componentes de segurança e controle;
- Financeiro, composto por: Contas a Pagar, Contas a Receber, Fluxo de Caixa e Conta Corrente de Caixa e Bancos;
- Materiais, composto por: Compras e Estoque;
- Vendas, composto por: Vendas, Faturamento, Mix, Cotas e Conta Corrente de Representantes;
- Produção, composto por: MRP II, PCP e Controle de Chão de Fábrica;
- Recursos Humanos, composto por: Folha de Pagamento, Ponto Eletrônico, Benefícios e Recursos Humanos;
- Custos, composto por: Planejamento Orçamentário, Custo Contábil e Custo Industrial; e
- Contabilidade, composto por: Controle de Patrimônio, Contabilidade e Livros Fiscais.

Este sistema possui mais de 1500 programas fontes, totalizando em torno de 1.500.000 linhas de código em *Progress 4GL*.

Segue-se uma apresentação de cada passo da estratégia usada na reengenharia deste sistema, com exemplos reais da aplicação da estratégia.

4.1 Organizar Sistema

A aplicação da estratégia tem início com o código fonte *Progress 4GL* do sistema legado. Neste passo faz-se a organização do código legado segundo os princípios da orientação a objetos, utilizando o transformador intra-domínio *ProgressToProgress*, no sistema de transformação Draco-PUC.

A figura 8 mostra, à esquerda acima, um trecho do código legado *Progress 4GL*, à esquerda abaixo, o correspondente código organizado segundo os princípios da orientação a objetos e à direita, os procedimentos da tabela *GEACLIEN*, candidatos a métodos, criados durante a organização. Por exemplo, a partir do comando *FIND GEACLIEN WHERE GEACLIEN.CLIENTE = W_CLIENTE*, que faz a leitura de um registro da tabela de clientes no banco de dados, cria-se o arquivo de procedimentos *GEACLIEN.P*, contendo os procedimentos de leitura relacionados com a tabela *GEACLIEN*, como *FIND*, que recebe como parâmetro os campos que compõem a cláusula de seleção *WHERE*. O comando *CREATE*, que cria um registro no banco de dados, é transformado no procedimento *GEACLIEN* e o comando *ASSIGN*, que grava as informações no registro criado, é transformado no procedimento *ASSIGN*, ambos armazenados em *GEACLIEN.P*. O procedimento *GEACLIEN* é responsável pela inclusão de um registro da tabela no banco de dados e o procedimento *ASSIGN* pela alteração. Como mostra a figura 8, o código é organizado para executar chamadas aos procedimentos armazenados em *GEACLIEN.P*.

```
/* código legado Progress 4GL */
/* GEVCLIEN.W */
(1) ON CHOOSE BTN_OK DO:
(2)     FIND GEACLIEN WHERE GEACLIEN.CLIENTE =
                          W_CLIENTE NO-LOCK NO-ERROR.
(3)     IF NOT AVAILABLE GEACLIEN THEN
(3)       CREATE GEACLIEN.
(4)     ASSIGN GEACLIEN.CLIENTE = W_CLIENTE.
(4)     ASSIGN GEACLIEN.NOME GEACLIEN.TELEFONE
                 WITH FRAME (&FRAME-NAME).
(1) END.
```

```
/* código organizado Progress 4GL */
/* GEVCLIEN.W */
(1) ON CHOOSE OF BTN_OK DO:

      IF NOT VALID-HANDLE(P_PROG) THEN
         RUN PERSISTENT GEACLIEN.P SET P_PROG.

(2)  RUN FIND (INPUT W_CLIENTE)      IN P_PROG.
(3)  RUN GEACLIEN IN P_PROG.
(4)  RUN ASSIGN (INPUT W_CLIENTE,
                       W_NOME,
                       W_TELEFONE)  IN P_PROG.
(1) END.
```

```
/*  procedimentos da tabela GEACLIEN */
/* GEACLIEN.P */
/* Procedure para leitura de um registro */
 PROCEDURE FIND:
     DEF INPUT PARAMAMETER P_CLIENTE LIKE
                           GEACLIEN.CLIENTE NO-UNDO.
 (2) FIND GEACLIEN WHERE GEACLIEN.CLIENTE =
                         P_CLIENTE NO-LOCK NO-ERROR.
END PROCEDURE.

 /* Procedure para criação do registro */
PROCEDURE GEACLIEN:
    IF NOT AVAILABLE GEACLIEN THEN
 (3)    CREATE GEACLIEN.
END PROCEDURE.

/* Procedure para alteração do registro */
PROCEDURE ASSIGN:
   DEF INPUT PARAMETER P_CLIENTE  LIKE
                       GEACLIEN.CLIENTE  NO-UNDO.
   DEF INPUT PARAMETER P_NOME     LIKE
                       GEACLIEN.NOME     NO-UNDO.
   DEF INPUT PARAMETER P_TELEFONE LIKE
                       GEACLIEN.TELEFONE NO-UNDO.
(4) ASSIGN GEACLIEN.CLIENTE  = P_CLIENTE
           GEACLIEN.NOME     = P_NOME
           GEACLIEN.TELEFONE = P_TELEFONE.
END PROCEDURE.
```

Figura 8 – Organização do código legado *Progress 4GL*

Outras transformações são, da mesma forma, aplicadas para organizar o código fonte, segundo as idéias do paradigma da orientação a objetos. Este código organizado é usado no próximo passo da estratégia.

4.2 Transformar Sistema

Neste passo, utilizam-se os Transformadores Inter-Domínios *ProgressToUML* e *ProgressToJava*. O código legado, organizado em blocos de comandos e declarações, que atuam sobre objetos comuns a determinado tipo, é transformado para descrições em UML. Classes encapsulando as declarações e os protótipos dos métodos são geradas para cada bloco de código organizado. Os códigos de cada procedimento *Progress 4GL* são transformados diretamente em Java. Dessa forma, as descrições UML das classes geradas têm embutidas na parte semântica dos métodos os respectivos códigos Java.

Para melhor entendimento deste passo apresenta-se na figura 9 um exemplo de transformação, onde à esquerda, tem-se o código legado organizado e à direita, a correspondente descrição UML. Por exemplo, o reconhecimento do comando *CREATE*, seguido de comandos *ASSIGN* no código fonte organizado, dá origem à especificação de uma classe. O comando *CREATE GEACLIEN* dentro do procedimento *GEACLIEN*, dá origem à especificação da classe *Geaclien*, descrita com *Object Class "Geaclien"* em UML. O procedimento *GEACLIEN*, no código fonte organizado, dá origem à especificação do protótipo do método *Object Operation "GEACLIEN"* em UML. O comando de atribuição *ASSIGN GEACLIEN.NOME = W_NOME*, no código fonte organizado, é transformado para *Object Class Attribute "Nome"*, em UML. Na figura 9 pode-se ver ainda, em destaque, o trecho de código organizado que foi transformado diretamente para Java. Este código fica embutido na parte semântica do método *BTN_OK_Clicked*, cujo protótipo está declarado na classe *GEVCLIEN*.

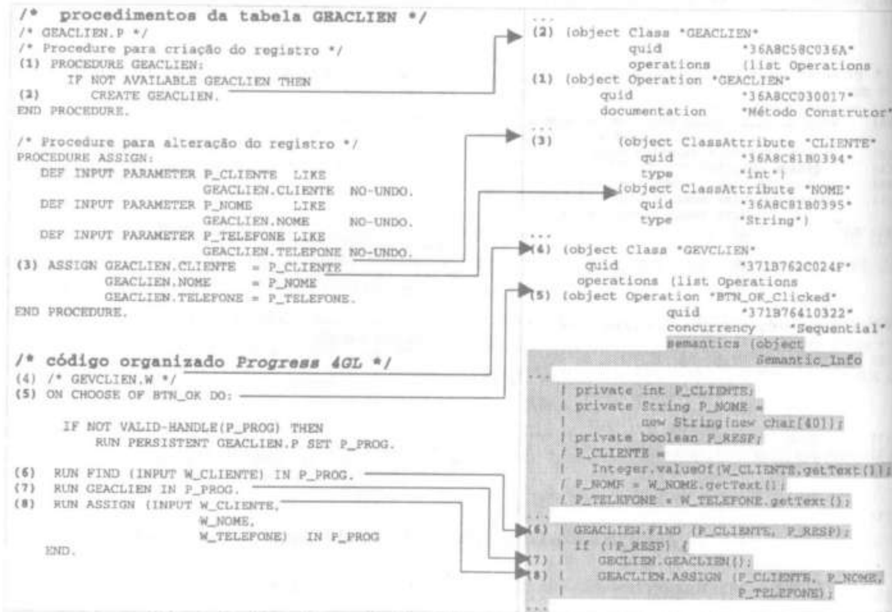

Figura 9 – Transformação do código organizado para especificações UML

As especificações na linguagem de modelagem UML, geradas neste passo, permitem recuperar o projeto orientado a objetos do sistema na ferramenta CASE *Rational Rose*, conforme apresenta o próximo passo da estratégia.

4.3 Reprojetar Sistema

Neste passo, o engenheiro de software usa a ferramenta *Rational Rose* para importar as especificações em UML e recuperar o Projeto Orientado a Objetos do sistema atual. A figura 10 mostra à esquerda parte o projeto recuperado na ferramenta CASE *Rational Rose* e a direita os correspondentes códigos Java dos métodos em cada classe do projeto recuperado.

Com o projeto do sistema recuperado em UML, torna-se possível compreendê-lo e reprojetá-lo para atender os requisitos. Pode-se também editar e alterar a semântica dos métodos, "portados" diretamente para Java.

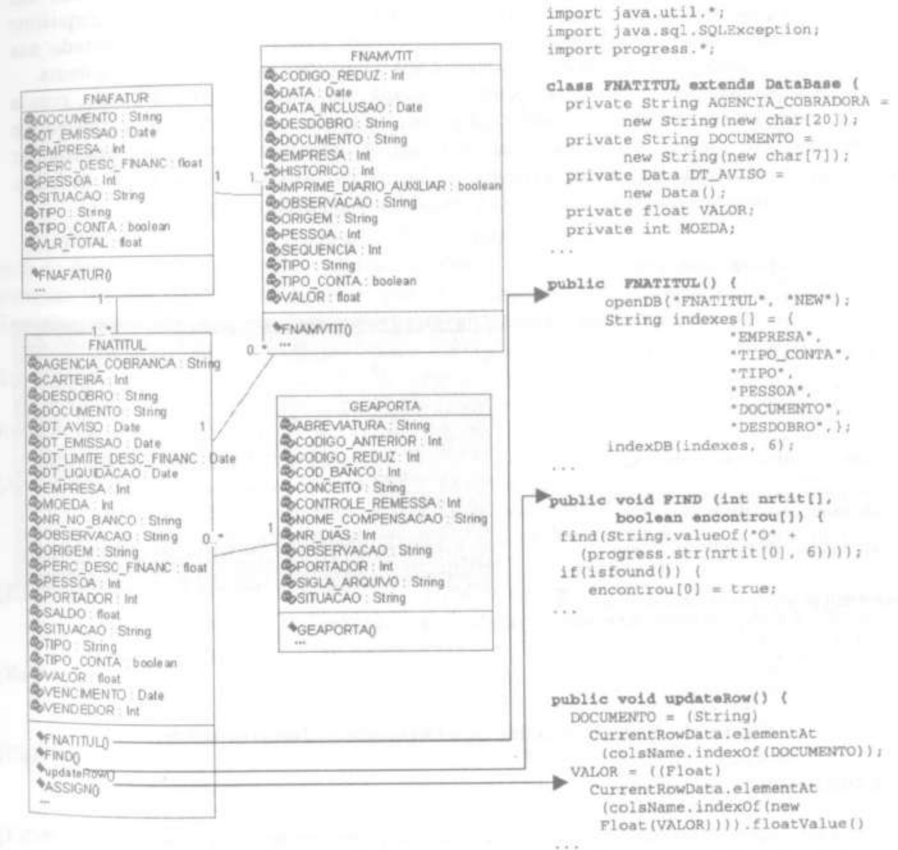

Figura 10 – Projeto orientado a objetos recuperado pela ferramenta CASE e código Java dos métodos

4.4 Reimplementar Sistema

No último passo da estratégia, são aplicadas as transformações da linguagem de

modelagem UML para a linguagem orientada a objetos Java. Os Diagramas de classes do sistema reprojetado são reimplementados em Java pelo transformador *CatalysisToJava* do domínio *Catalysis*. Na reimplementação, o sistema de transformação Draco-PUC, trata os métodos das classes, cujos comportamentos já estão especificados em Java, como uma linguagem embutida na linguagem UML.

O código Java gerado a partir das especificações UML é integrado com o código Java dos métodos para obter a implementação final do sistema. A figura 11, mostra à esquerda o código Java gerado e à direita, a sua correspondente interface, após a reengenharia.

Como resultado da reconstrução do sistema, foram geradas 362 classes de análise e 1020 classes de projeto, responsáveis pela interface. Cada classe, em média, possui 10 métodos e se relaciona com outras 3 classes.

O código gerado pode ser executado por diferentes CPUs e os resultados das execuções são analisados pelo engenheiro de software para verificar se atendem aos requisitos do sistema. Estes resultados fornecem um *feedback* aos passos anteriores, orientando nas correções para validar se a implementação atende aos requisitos especificados para sistema.

A funcionalidade do sistema legado é mantida no sistema reimplementado, com a grande vantagem de facilitar a manutenção, uma vez que o código está organizado e encapsulado em classes e objetos. Uma vez que as transformações foram construídas preservando a semântica da linguagem origem na linguagem alvo da implementação, as falhas podem ser do próprio código legado ou das alterações introduzidas no reprojeto.

```
import java.util.*;
import java.sql.SQLException;
import java.awt.*;
import progress.*;
import com.sun.java.swing.UIManager;

package MEGAADM;

public class GEVCIDAD {
  boolean packFrame = false;

public GEVCIDAD {
  Frame1 frame = new Frame1();
...
  ButtonControl OK = new ButtonControl();
  ButtonControl CANCELAR  = new
                            ButtonControl();
  ButtonControl PESQUISAR = new
                            ButtonControl();
...
void PESQUISAR_actionPerformed(ActionEvent e) {
    GEATRNSP.FIND(P_TRANSP, P_RESP);
    while (!P_RESP[0]) {
        GEATRNSP.GET(P_TRANSP);
        GEATRNSP.NOME ();
        Utilitary.LIMPA();
...
```

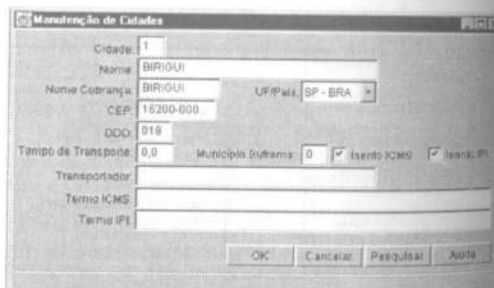

Figura 11 – Reimplementação final Orientada a Objetos do sistema

5. Conclusão

Este artigo apresentou uma estratégia para reengenharia de sistemas legados orientados a procedimentos, escritos em *Progress 4GL*, para sistemas orientados a objetos, sem perda da funcionalidade e com a possibilidade de reprojeto do sistema antes da reimplementação. A estratégia objetiva reconstruir sistemas legados escritos em linguagem procedural, para serem executados em plataformas mais modernas de hardware e software.

A estratégia pode ser aplicada a sistemas legados em outras linguagens diferentes de *Progress 4GL*, como por exemplo, Clipper. A linguagem alvo também pode ser outra, diferente de Java, como por exemplo, *Object Pascal* e C++. O desempenho do sistema orientado a objetos, após a reengenharia, depende da linguagem alvo da transformação, usando Java ganha-se em portabilidade e perde-se em desempenho.

As seguintes idéias comprovam a originalidade desta estratégia:

- Sistemas legados descritos em *Progress 4GL* são reimplementados, segundo os princípios da orientação a objetos, em uma linguagem de 4ª Geração, incluindo comandos específicos para consulta e manutenção de informações em um banco de dados relacional;
- O Domínio *Progress 4GL*, com 3 bibliotecas *ProgressToProgress*, *ProgressToUML* e *ProgressToJava*, foi construído no Sistema de Transformação Draco-PUC;
- Uma biblioteca com classes que tratam as entradas e saídas do Banco de Dados *Progress* em Java foi construída para facilitar o processo de transformação;
- Diferentes técnicas, destacando-se o método Fusion/RE, uma ferramenta CASE e um Sistema de Transformação foram integrados para suportar a Reengenharia;
- Uma estratégia que cobre todo o ciclo da reengenharia de um software não orientado a objetos, desde a recuperação do projeto original do sistema até a reimplementação automática numa linguagem orientada a objetos, foi definida e testada em vários sistemas legados, incluindo um sistema com aproximadamente 1.500.000 linhas de código.

Dada a capacidade do Sistema de Transformação Draco-PUC de trabalhar com uma rede de domínios com diferentes linguagens, pode-se aplicar a estratégia de reengenharia em sistemas legados escritos em outras linguagens, diferentes de *Progress 4GL*. Também as linguagens alvo da reimplementação podem ser diferentes de UML e Java.

Outra contribuição desta pesquisa é que a estratégia torna sistemático o uso de Sistemas de Transformação na Reengenharia de Software.

Referências

[Abr99] 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.

[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.

[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.

[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. (or http://www.queis.queensu.ca/STLab/TXL)

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

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

[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.

[Lei91] 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.

[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.

[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.

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

[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.* Memorias, pg. 383-394. Quito-Equador. 19-23 de Outubro, 1998.

[Rat98] RATIONAL SOFTWARE CORPORATRION, *Rational Rose 98 Rose Extensibility User's Guide*, 1998.

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

[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.

[Uml97] FOWLER, M., SCOTT, K. **UML Distilled – Applying the Standard Object Modeling Language.** Addison-Wesley, 1997.

[Uml98] URL: http://www.rational.com/uml/references, 1998.

[Wid93] WILE D. **POPART: Producer of Papers and Related Tools System Builders Manual.** Technical Report. USC/Information Sciences Institute. November, 1993.