Reengenharia de Software Orientada a Componentes Distribuídos

Edimilson Ricardo Azevedo Novais - novais@dc.ufscar.br
Antonio Francisco do Prado - prado@dc.ufscar.br
Universidade Federal de São Carlos

Resumo

Este artigo apresenta uma estratégia de Reengenharia de Software Orientada a Componentes Distribuídos para reconstrução de sistemas legados. Componentes são utilizados para facilitar a manutenção do sistema reconstruído e permitir sua execução em plataformas heterogêneas e distribuídas. A estratégia é realizada em 5 passos. No passo **Organizar Código Legado**, 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. No passo **Gerar Especificações Orientadas a Objetos**, o código legado procedural é transformado para especificações do primeiro nível da linguagem de modelagem *Catalysis*. No passo **Especificar Componentes,** o engenheiro de software parte do projeto original do código legado, obtido das especificações do primeiro nível de *Catalysis*, e especifica os componentes em uma ferramenta *CASE*, conforme o segundo nível de *Catalysis*. No passo **Projetar Componentes**, o engenheiro de software faz o projeto interno dos componentes, seguindo o terceiro nível de *Catalysis*. Finalmente, no passo **Reimplementar Sistema**, as especificações em *Catalysis* são transformadas para uma linguagem de programação orientada a objetos, obtendo-se a implementação final do sistema.

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

Abstract

*This paper presents a strategy for Distributed Components Oriented Software Reengineering for reconstruction of legacy systems. Components are used to facilitate the maintenance of the rebuilt system and to allow its execution in heterogeneous and distributed platforms. The strategy is executed in 5 steps. In the **Organize Legacy Code** step, the legacy source code is organized according to the principles of the object orientation. It's a preparatory step to facilitate the transformation of a procedural code into an object oriented. In the **Generate Objects Oriented Specifications** step, the procedural code is transformed into first level specifications of the modeling language Catalysis. In the **Specify Components** step, the software engineer starts from the original project of legacy code, obtained of the first level specifications of Catalysis and specifies the components in a CASE tool, according to the second level of Catalysis. In the **Project Components** step, the software engineer does the components inner project, following the third level of Catalysis. Finally, in the **Reimplement System** step, the specifications in Catalysis are transformed into an object oriented programming language, getting the final implementation of system.*

***Key Words**: Software Reengineering, Transformation System, Reverse Engineering, Components, Reuse and Object Oriented.*

1. Introdução

A dificuldade em atualizar os softwares para a utilização de novas tecnologias tem motivado os pesquisadores a investigar soluções que diminuam os custos de desenvolvimento, garantam um tempo de vida maior para o sistema e facilitem a manutenção [1, 2, 4].

O conhecimento adquirido com os sistemas legados é utilizado como base para a evolução contínua e estruturada do software. O código legado possui lógica de programação, decisões de projeto, requisitos de usuário e regras de negócio que podem ser recuperados sem perda da semântica, facilitando sua reconstrução.

As técnicas atuais utilizam basicamente a Orientação a Objetos e a Computação Distribuída como principais alicerces. A simples adoção da tecnologia de objetos, sem a existência de um padrão de reuso explícito e de um processo de desenvolvimento de software orientado ao reuso, não fornece o sucesso esperado para produção de software em larga escala [7]. Como evolução destas técnicas, têm-se os componentes de software distribuídos que facilitam o reuso e a manutenção do software. Os componentes de software devem ser reutilizados em todos os níveis de abstração do processo de desenvolvimento e não simplesmente no nível de código.

A reengenharia de software é também uma forma de reuso que permite obter o entendimento do domínio da aplicação, recuperando as informações das etapas de análise e projeto e organizando-as de forma coerente e reutilizável.

Motivados por estas idéias pesquisou-se uma estratégia de reengenharia cujo objetivo é recuperar o projeto orientado a objetos de sistemas legados, reconstruindo-os, com técnicas de componentes distribuídos, para serem executados em novas plataformas de hardware e software. A estratégia suporta a manutenção do sistema, inclusive com alteração da sua estrutura original, garantindo sua evolução através do uso de componentes de software distribuídos.

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 de Software Orientada a Componentes Distribuídos, 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 da Reengenharia proposta é a da reconstrução de software usando transformações. Um importante sistema de transformação de software que vem sendo utilizado nesta área é o *Draco-PUC*, conforme pode ser constatado em [9, 16, 14, 1, 5, 15]. Os transformadores são responsáveis pela automatização ou semi-automatização do processo de reconstrução de software.

Outra técnica que se destaca em trabalhos de reengenharia é a de engenharia reversa *Fusion-RE* [11], utilizada para obter o entendimento e revitalizar a estrutura do código legado segundo o paradigma da orientação a objetos, visando reutilizar 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, unidades de programa 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 das unidades de programa.

Além do *Fusion-RE*, outras técnicas de Engenharia Reversa são utilizadas para obtenção dos Casos de Uso do código legado e de seus Diagramas de Seqüência, que representam os fluxos de execução dos cursos normal e alternativos de cada de cenário de uso do código legado.

As diferentes técnicas empregadas na engenharia reversa do sistema legado permitem recuperar os modelos do primeiro nível do método *Catalysis*. *Catalysis* é um método de desenvolvimento baseado em componentes, que cobre todas as fases do ciclo de vida do sistema, desde a especificação até a implementação. *Catalysis* suporta desde a análise e especificação do domínio do problema, até a especificação e o projeto interno dos componentes, que constituem os elementos reutilizados nas diferentes aplicações. O processo de desenvolvimento de software em *Catalysis* é dividido em níveis lógicos: **Domínio do Problema, Especificação dos Componentes e Projeto Interno dos Componentes** [3].

As hierarquias de comportamentos herdados de um sistema Orientado a Objetos podem se tornar estruturas pesadas, difíceis de serem entendidas e cheias de interdependências, o que as tornam difíceis de serem modificadas [19]. Por isso é necessário estabelecer uma relação formal entre os componentes e a aplicação que os utiliza, por meio de interfaces bem definidas. Para atender a este requisito a estratégia proposta utiliza a técnica *Enterprise JavaBeans*, *EJB* [18], na implementação dos componentes distribuídos.

A técnica *EJB* permite separar a interface, as regras de negócio e o banco de dados, criando aplicações multicamadas. Esta técnica emprega componentes orientados à transação que são executados em servidores *EJB*. Com a técnica *EJB*, os componentes que implementam e encapsulam as regras lógicas do negócio são disponibilizados no servidor *EJB*. Esta técnica é ideal para a implementação de sistemas distribuídos e portáveis.

Além destas técnicas, vem ganhando destaque o uso de ferramentas *CASE* no projeto ou reprojeto de sistemas a serem reconstruídos. Uma ferramenta *CASE* que possui características importantes para esta estratégia é a *MVCASE*, resultado de outro projeto de pesquisa [10], que além de suportar a especificação do sistema em linguagens de modelagem orientadas a objetos, gera código automaticamente em uma linguagem de programação orientada a objetos, a partir das especificações em alto nível, usando componentes distribuídos.

Combinando as idéias do sistema de transformação *Draco-PUC*, as técnicas de engenharia reversa do método *Fusion-RE* e das experiências em trabalhos de reengenharia [1, 5, 14, 15], o método de desenvolvimento de componentes *Catalysis*, a técnica para implementação de sistemas distribuídos *Enterprise JavaBeans* e a ferramenta *CASE MVCASE*, definiu-se uma estratégia para a Reengenharia de Software Orientada a Componentes Distribuídos, que será apresentada a seguir.

3. Reengenharia de Software Orientada a Componentes Distribuídos

A estratégia de Reengenharia de Software Orientada a Componentes Distribuídos é realizada em cinco passos: **Organizar Código Legado, Gerar Especificações Orientadas a Objetos, Especificar Componentes, Projetar Componentes e Reimplementar Sistema**, como mostra a Figura 1.

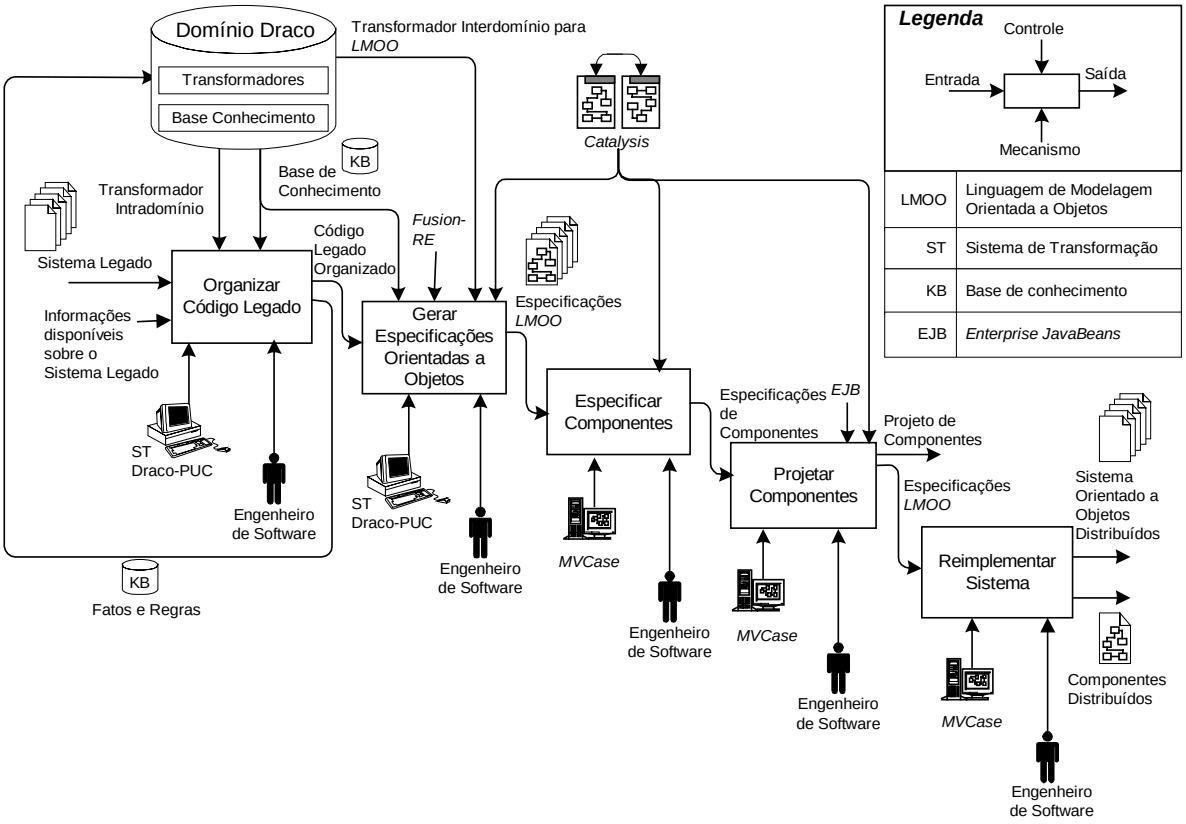

Figura 1 – Estratégia de Reengenharia de Software Orientada a Componentes Distribuídos

Segue-se uma apresentação de cada um dos passos da estratégia.

3.1 Organizar Código Legado

Neste primeiro passo, faz-se a organização do código legado segundo os princípios da orientação a objetos. O código legado, escrito de forma procedural, é organizado, 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. Informações disponíveis sobre o código legado, caso existam, são também utilizadas neste passo para facilitar a organização.

O principal mecanismo de execução deste passo é o sistema de transformação *Draco-PUC*. Para automatizar grande parte das transformações foi construído um transformador intradomínio que mapeia o código legado no código legado organizado [14].

Para auxiliar o processo das transformações utiliza-se uma Base de Conhecimento, que permite armazenar e consultar fatos e regras com informações sobre o código legado [14]. Neste passo, a Base de Conhecimento armazena fatos relacionados com as supostas classes, atributos, métodos, relacionamentos, hierarquia de chamadas e particionamento do sistema em Casos de Uso.

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

3.1.1 Definir hierarquia de chamadas de supostos métodos. Inicialmente o código legado é percorrido pelo transformador intradomínio, do sistema de transformação *Draco-PUC*, para definir a hierarquia de chamadas dos supostos métodos do código legado. Para realizar esta atividade deve-se:

a) Para cada unidade de programa, que pode ser: um programa, subprograma, arquivo *include*, procedimento ou função, criar um suposto método da seguinte forma:

a. Criar um arquivo de programa. O nome do arquivo de programa é formado pela concatenação do nome do programa fonte e o nome da unidade de programa;
b. Mover o código do suposto método do código legado para o arquivo de programa criado;

b) Para cada suposto método, determinar seus parâmetros, baseado na ocorrência de variáveis e estruturas de dados locais e globais do código legado, analisando escopo, dependência, visibilidade e tempo de vida; e
c) Reorganizar o código legado através de chamadas aos supostos métodos.

Nesta etapa, o código legado correspondente a cada unidade de programa é transformado em um suposto método. O código legado é substituído por uma chamada ao suposto método criado, mantendo o fluxo de execução original e criando uma hierarquia de chamadas dos supostos métodos.

A Figura 2 mostra, à esquerda, na parte superior, o código legado original, em seguida, o código legado reorganizado através de chamadas aos supostos métodos, e à direita os supostos métodos criados.

```
/* File: GEDPRAZO.W */
ON START-SEARCH OF BROWSE-CADASTRO IN FRAME DEFAULT-FRAME DO:
    /* Chama o filtro para classificar o browse */
    {\PROG\FONT\INCLUDE\VISNFILT.I}  (1)
    CASE COLUNA:
        WHEN GEAPRAZO.PRAZO:HANDLE IN BROWSE
            BROWSE-CADASTRO THEN
            OPEN QUERY BROWSE-CADASTRO FOR EACH GEAPRAZO NO-LOCK
                WHERE GEAPRAZO.PRAZO >= INTEGER(INICIAL) AND
                      GEAPRAZO.PRAZO <= INTEGER(FINAL)  (2)
                BY GEAPRAZO.PRAZO.
    /* Muda o ponteiro do mouse */
    T_METODO_RETORNO    =   BROWSE_CADASTRO:LOAD-MOUSE-
POINTER("GLOVE").
END.
ON CHOOSE OF BTN_OK IN FRAME Dialog-Frame /* OK */ DO:
    DEF VAR T_RESP AS LOGICAL INITIAL NO NO-UNDO.

    RUN TESTA_CAMPOS (OUTPUT T_RESP). (3)
...
```

```
/* GEDPRAZO_VISNFILT.P */(1)
DEF INPUT PARAMETER T_CURRENT-
COLUMN AS HANDLE NO-UNDO.
DEF VAR T_METODO_RETORNO
      AS LOGICAL NO-UNDO.
ASSIGN COLUNA T_CURRENT-COLUMN.
IF COLUNA:LABEL-FONT <> 15
                              THEN
RETURN.
CASE COLUNA:DATA-TYPE:
  WHEN    "CHARACTER" THEN
     ASSIGN  T_TIPO    =   1.
  WHEN    "DATE"      THEN
...
```

```
/* GEDPRAZO_ABRE_BROWSE */ (2)
CASE COLUNA:
    WHEN GEAPRAZO.PRAZO:HANDLE
IN BROWSE
  OPEN QUERY BROWSE-CADASTRO FOR
EACH GEAPRAZO NO-LOCK
...
```

```
/* File: GEDPRAZO.W – Reorganizado */
ON START-SEARCH OF BROWSE-CADASTRO IN FRAME DEFAULT-FRAME DO:
    /* Chama o filtro para classificar o browse */
 RUN GEDPRAZO_VISNFILT.P (INPUT BROWSE_CADASTRO:CURRENT-COLUMN) (1)
    CASE COLUNA:
        WHEN GEAPRAZO.PRAZO:HANDLE IN BROWSE
            BROWSE-CADASTRO THEN
            RUN GEDPRAZO_ABRE_BROWSE (INPUT INICIAL, FINAL). (2)
    /* Muda o ponteiro do mouse */
    T_METODO_RETORNO    =   BROWSE_CADASTRO:LOAD-MOUSE-POINTER("GLOVE").
END.
ON CHOOSE OF BTN_OK IN FRAME Dialog-Frame /* OK */ DO:
    DEF VAR T_RESP AS LOGICAL INITIAL NO NO-UNDO.

    RUN GEDPRAZO_TESTA_CAMPOS.P (INPUT GEAPRAZO.DESC_PRAZO:SCREEN-VALUE,
OUTPUT T_RESP).  (3)
...
```

```
/* GEDPRAZO_TESTA_CAMPOS */ (3)
DEF INPUT PARAMETER P_DESC_PRAZO
AS CHARACTER NO-UNDO.
DEF OUTPUT PARAMETER P_RESP AS
LOGICAL NO-UNDO.

IF P_DESC_PRAZO = “” THEN DO:
  MESSAGE "O prazo deve ser
    informado" VIEW-AS ALERT-BOX
         ERROR TITLE "Erro".
  RETURN NO-APPLY.
END.

ASSIGN P_RESP = YES.
...
```

Figura 2 –Particionamento do código legado em supostos métodos

3.1.2 Criar supostas classes. Nesta atividade o código legado é organizado em supostas classes, com seus supostos atributos e supostos relacionamentos. Para organizar o código legado são definidos padrões de reconhecimento que são transformados em novos comandos, na mesma linguagem do código legado, porém com as características da orientação a objetos.

O particionamento do sistema em supostas classes considera o Modelo Entidade-Relacionamento (MER) que deu origem ao banco de dados do sistema, normalmente relacional, as estruturas de dados globais ou locais, e as interfaces de interação do sistema

com o ambiente externo. Parte-se do princípio que o *MER* está consistente e que o banco de dados do sistema está normalizado. Destacam-se os seguintes padrões de reconhecimento [15]:

- Supostas Classes: reconhecidas pelos comandos de abertura e criação de tabelas no banco de dados, estruturas de dados, e interfaces de interação;
- Supostos Atributos: reconhecidos pelos comandos de acesso ao banco de dados, campos de estruturas de dados, e elementos de interação das interfaces; e
- Supostos Relacionamentos: reconhecidos pelos campos que formam um índice único em uma tabela A, e também são atributos de uma tabela B. Para tabelas que são relacionadas, mas não possuem os mesmos atributos, são também analisados os comandos de acesso ao banco de dados.

Procurando organizar o código legado segundo uma arquitetura de 3 camadas: Interface, Regras de Negócios, e Banco de Dados, os Componentes Transformacionais que criam as supostas classes devem:

a) Para cada tabela do banco de dados, criar uma suposta classe persistente da camada Banco de Dados. Para cada campo da tabela, criar um suposto atributo da suposta classe. Baseado nos relacionamentos da tabela, determinar os relacionamentos da suposta classe. Gerar fatos na base de conhecimento sobre as interfaces da suposta classe de Banco de Dados, para disponibilizar métodos de acesso ao banco de dados. Estas supostas interfaces fazem parte da camada Regras de Negócio;
b) Para cada estrutura de dados, criar uma suposta classe da camada Regras de Negócio. Para cada campo da estrutura de dados, criar um suposto atributo da suposta classe; e
c) Para cada tela de interação, criar uma suposta classe da camada Interface. Para cada elemento da tela de interação, criar um suposto atributo da suposta classe, e para cada evento de um elemento da tela de interação, criar um suposto método da suposta classe.

As supostas classes do sistema são organizadas em três camadas: *Interface*, *Regras de Negócio* e *Banco de Dados*. As supostas classes responsáveis pela interface do sistema são incluídas no pacote *Interface*. As supostas classes responsáveis pelas regras de negócio, que definem a lógica e a funcionalidade do sistema, são incluídas no pacote *Regras de Negócio*. As supostas classes de acesso e atualização do banco de dados são incluídas no pacote *Banco de Dados*.

A separação do sistema em 3 camadas facilita o reuso, aumenta a independência e a portabilidade do sistema reconstruído. Dessa forma, pode-se, por exemplo, ter aplicações na camada de interface escritas para serem executadas na *Web* e implementadas em diferentes linguagens como *HTML*, *JSP* [8], *PHP* [12], *Java* [8] ou *Delphi*. Essas diferentes aplicações podem reutilizar as mesmas regras da camada de negócios, executadas em outras linguagens diferentes da interface. Esta independência facilita a manutenção dos sistemas. Por sua vez a persistência dos dados, *Banco de Dados*, pode ser feita em um Banco de Dados: *Oracle*, *Progress*, *MySQL* ou em qualquer outro sistema gerenciador de banco de dados. Esta organização em camadas torna o sistema mais flexível para suportar as mudanças tecnológicas sem comprometer sua estrutura, aumentando assim seu ciclo de vida.

Com o objetivo de facilitar o processo de transformação, principalmente das classes que tratam funções de acesso ao banco de dados, foi criada uma biblioteca de componentes que utiliza padrões de projeto [6]. Esta biblioteca fica na camada *Banco de Dados* e usa comandos SQL para acessar um banco de dados.

3.1.3 Alocar supostos métodos. Nesta atividade, os supostos métodos são alocados nas supostas classes, corrigindo-se anomalias, conforme o método *Fusion-RE*. Os supostos

métodos são classificados em construtores **(c)**, quando alteram a estrutura de dados, e observadores **(o)**, quando somente consultam as estruturas de dados. Por exemplo, quando um suposto método refere-se a mais de uma suposta classe ele deve ser alocado usando os seguintes critérios:

- Se construtor de uma suposta classe e observador de outra **(oc)**, será alocado na suposta classe que faz referência como construtor;
- Se observador de uma suposta classe e construtor de várias **(oc+)**, será alocado na primeira suposta classe que faz referência como construtor**;** e
- Se observador de mais de uma suposta classe e construtor de apenas uma **(o+c)**, será alocado na primeira suposta classe que faz referência como observador.

3.1.4 Criar casos de uso. Nesta atividade, o engenheiro de software define o nome de cada caso de uso do código legado e o respectivo suposto método que dá início as suas execuções. Esta definição é realizada com base na hierarquia de chamadas dos supostos métodos. Como resultado, tem-se o particionamento da hierarquia de chamadas em torno de cada caso de uso. Cada curso, normal e alternativo, do caso de uso representa um cenário de uso do código legado.

Para cada caso de uso deve-se determinar, na hierarquia das chamadas dos supostos métodos, o suposto método que dá início ao caso de uso e definir o nome do caso de uso.

A Figura 3 mostra, por exemplo, o caso de uso *AtenderPedido* com seu curso normal e alternativo.

Caso de Uso	Chamador	Chamados	
Atender Pedido	VDVPEDID.W (Curso Normal)	1, VDDPEDID.W 2, GECCLIEN.W 3, VDDPEDID_TESTA_CAMPOS.P 4, VDDPEDID_GRAVA.P 5, VDDPEDID_INCLUI_ITENS.P	6, VDDITEMP.W 7, MTCPRODU.W 8, VDDITEMP_TESTA_CAMPOS.P 9, VDDPEDID_CALCULA_TOTAL.P
	VDVPEDID.W (Curso Alternativo)	1, VDDPEDID.W 2, GECCLIEN.W 3, VDDPEDID_TESTA_CAMPOS.P 4, VDDPEDID_GRAVA.P 5, VDDPEDID_ALTERA_ITENS.P	6, VDDITEMP.W 7, MTCPRODU.W 8, VDDITEMP_TESTA_CAMPOS.P 9, VDDPEDID_CALCULA_TOTAL.P

Figura 3 – Seqüência de chamadas de um Caso de Uso

3.1.5 Criar suposta classe principal. Os supostos métodos, que não fazem parte da seqüência de execução de nenhum caso de uso, são alocados em uma suposta classe principal que descreve o comportamento global do sistema, considerando sua comunicação com o ambiente.

O engenheiro de software, baseado em informações sobre o código legado, pode consultar e alterar a base de conhecimento durante as transformações de organização do código. As possíveis alterações são:

- Mudar os nomes das classes, elementos de dados e procedimentos para nomes mais significativos;
- Mudar a classificação de um procedimento: construtor, observador ou interface;
- Mudar um procedimento de uma classe, ou substituí-lo por outro; e
- Mudar o tipo de um relacionamento.

A organização do código fonte legado, segundo os princípios da Orientação a Objetos, facilita a transformação para especificações orientadas a objetos.

3.2 Gerar Especificações Orientadas a Objetos

Neste passo, o engenheiro de software também utiliza o sistema de transformação *Draco-PUC* para gerar automaticamente as especificações orientadas a objetos.

A base de conhecimento obtida no passo anterior, Organizar Código Legado, é consultada neste passo para a geração das classes, atributos, protótipos de métodos e relacionamentos entre as classes. O fluxo de controle do código legado organizado, definido pela seqüência de chamadas dos supostos métodos, é substituído pelas conexões de mensagem entre os objetos.

O código legado organizado segundo classes, com seus atributos e protótipos de métodos, é transformado para especificações do primeiro nível, **Domínio do Problema**, da linguagem de modelagem *Catalysis*,

As especificações dos corpos dos métodos são geradas diretamente na linguagem de programação *Java*. Dado o conhecimento que grande parte dos desenvolvedores têm sobre linguagens de programação, decidiu-se pelo "porte" direto do código legado dos corpos dos supostos métodos para a linguagem *Java*. O código *Java* gerado fica embutido nas especificações *Catalysis* das classes, na parte semântica de cada método.

Para realizar este passo são executadas as seguintes atividades.

3.2.1 Gerar Especificações de Métodos. Para cada suposto método, transformar seu código legado organizado em um corpo de método na linguagem de programação *Java*, mantendo a mesma semântica.

A Figura 4 apresenta um exemplo de transformação do código legado organizado para a linguagem *Java*. Do lado esquerdo, tem-se em destaque o padrão de reconhecimento *Display* que mostra uma mensagem na tela, no código legado *COBOL*. Do lado direito, tem-se o correspondente padrão *expression* que chama um método para mostrar uma mensagem na tela, na linguagem alvo *Java*.

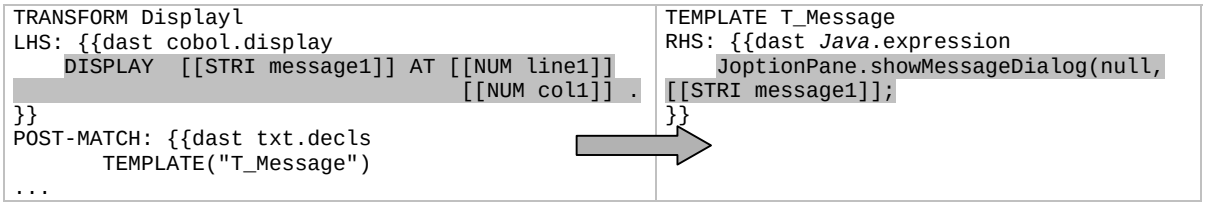

Figura 4 –Transformação de Código Legado para *Java*

3.2.2. Gerar Especificações de Classes. Nesta atividade criam-se as classes com seus protótipos de métodos e atributos. Para realizar esta atividade deve-se:

a) Para cada suposta classe, criar a especificação correspondente na linguagem de modelagem *Catalysis.* Para cada suposto método da suposta classe, criar a especificação correspondente ao seu protótipo em *Catalysis*.
b) Incluir o código do corpo do método em *Java*, na especificação na linguagem de modelagem *Catalysis*.
c) Para cada suposto atributo da suposta classe, criar a especificação correspondente na linguagem de modelagem *Catalysis*.

3.2.3 Gerar Especificações de Casos de Uso. Nesta atividade, para cada caso de uso criar a especificação correspondente em *Catalysis* e para cada seqüência de chamadas do caso de uso criar a correspondente especificação do diagrama de seqüência em *Catalysis*.

A Figura 5 apresenta um exemplo de transformação do código organizado para *Catalysis*. Do lado esquerdo, tem-se em destaque o padrão de reconhecimento *Function*, que define uma função na linguagem de programação *Clipper*. Do lado direito, tem-se o correspondente padrão de substituição *Operations*, que define o protótipo de método na linguagem de modelagem *Catalysis*.

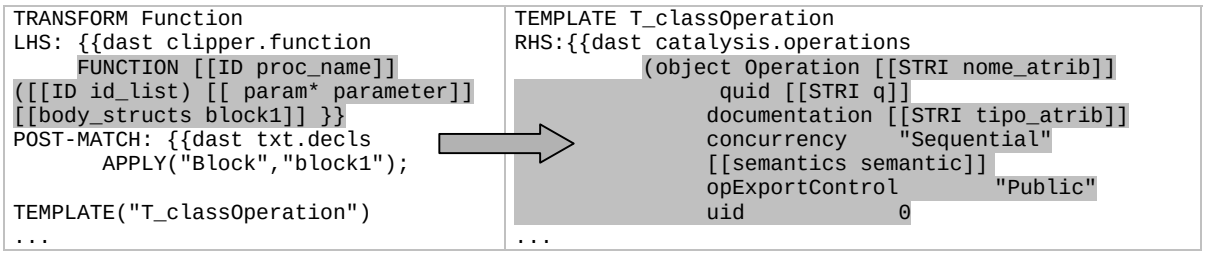

Figura 5 –Transformação do código legado para *Catalysis*

Da mesma forma, foram escritas as demais transformações que mapeiam o código legado organizado para as especificações orientadas a objetos.

3.3 Especificar componentes

Neste passo, o engenheiro de software refina as especificações geradas na linguagem de modelagem orientada a objetos, em uma ferramenta *CASE*, para obter as especificações dos componentes.

Os modelos gerados, representados pelas técnicas de Caso de Uso, Diagrama de Seqüência e Modelo de Tipos do primeiro nível de *Catalysis* são refinados para modelar os componentes do sistema. Novos modelos podem ser especificados na ferramenta *CASE*.

No nível de **Especificação dos Componentes** em *Catalysis* é dada ênfase na identificação, comportamento e responsabilidade dos componentes. Entre as técnicas *Catalysis* para especificar componentes, destacam-se os Modelos de Colaboração, de Casos de Uso e de Tipos. Na estratégia proposta estes modelos são obtidos do refinamento dos modelos gerados nas especificações na linguagem de modelagem *Catalysis*.

As atividades deste passo são realizadas pelo engenheiro de software, na ferramenta *CASE*, e compreendem:

a) Importar as especificações geradas em *Catalysis*, para obter o projeto orientado a objetos do sistema atual;
b) Especificar o Modelo de Colaboração, baseado nos Casos de Uso;
c) Reespecificar o Modelo de Tipos; e
d) Reespecificar o Diagrama de Seqüência.

Estes modelos são utilizados no próximo passo da estratégia para projeto interno dos componentes.

3.4 Projetar Componentes

Neste passo o engenheiro de software, também usando a ferramenta *CASE*, faz o projeto interno dos componentes. Este passo corresponde ao nível **Projeto Interno dos Componentes** em *Catalysis*, responsável pela parte física dos componentes, preocupando-se com a distribuição na plataforma de hardware e software definida para reimplementação do sistema.

Neste nível refinam-se os Modelos de Tipos para Diagrama de Classes, onde são modelados os componentes com suas interfaces que facilitam o reuso. Outro modelo deste passo é o Diagrama de Interação entre objetos, obtido do refinamento do Diagrama de Seqüência.

Na ferramenta *MVCASE*, a técnica de implementação *EJB* é utilizada para representar a arquitetura física e lógica dos componentes. Estes componentes, denominados *Beans*, podem ser do tipo *Session*, disponíveis durante a sessão de execução, ou *Entity*, persistentes em um banco de dados usando o protocolo JDBC. Ainda neste passo são especificadas as interfaces do componente: a *Interface Home*, que possui métodos para referenciar e manter o objeto remoto, e a *Interface Remote* que possui métodos responsáveis pelas regras de negócio, acessados pela aplicação. A *MVCASE* gera automaticamente as interfaces e métodos para dar suporte ao componente *EJB*.

Com base no projeto dos componentes, a *MVCASE* também gera automaticamente um arquivo de configuração *XML* [20], denominado *Deployment Descriptor*, que contem as propriedades dos componentes. O engenheiro de software pode alterar as propriedades do *Deployment Descriptor* para definir a arquitetura distribuída do componente.

Em resumo, as atividades deste passo são:

a) Criar o Diagrama de Classes dos componentes;
b) Criar o Diagrama de Interação dos componentes;
c) Gerar os componentes *EJB*, com suas interfaces e métodos; e
d) Gerar o *Deployment Descriptor* dos componentes.

Com o projeto do sistema recuperado é possível compreendê-lo e reprojetá-lo usando técnicas de alto nível de abstração, e a partir daí gerar novamente seu código em uma linguagem de programação orientada a objetos. Tanto o projeto como a implementação dos métodos estão disponíveis na ferramenta *CASE* para o reprojeto, caso seja necessário alguma manutenção.

3.5 Reimplementar Sistema

Finalmente, no último passo da estratégia, faz-se a reimplementação final do sistema numa linguagem orientada a objetos, no caso *Java*. Neste passo a estratégia usa a ferramenta *MVCASE* como mecanismo de geração de código, a partir dos Diagramas de Classes, com seus atributos e protótipos de métodos especificados em *Catalysis*. O código *Java* gerado neste passo é integrado com o código dos métodos, que foram inseridos na parte semântica das especificações *Catalysis*, obtendo-se a implementação final do sistema.

Os componentes implementados em *Java*, pela ferramenta *CASE*, são baseados na arquitetura *Java*/*EJB* e podem ser chamados em outras máquinas virtuais *Java* de forma remota, permitindo a execução do sistema distribuído em múltiplas plataformas. O pacote *Remote Method Interface*, (*RMI)*, que suporta a chamada de métodos remotos na arquitetura Cliente e Servidor, é utilizado pelos componentes *EJB* para tornar mais simples e transparente a distribuição de objetos em um programa *Java*. Um Cliente *RMI* interage com objetos no Servidor por meio da Interface Remota do objeto servidor. Utiliza-se um *stub*, gerado automaticamente pelo *EJB*, para chamar um método da classe remota [5].

Para validar a estratégia de Reengenharia foram reconstruídos diferentes sistemas legados. Segue-se a apresentação de um estudo de caso que mostra a aplicação da estratégia de Reengenharia proposta.

4. Estudo de Caso

Trata-se de um sistema de controle de distribuição de livros. O sistema se divide em dois grandes módulos: o primeiro é responsável por Controlar e Atender os Pedidos, com as seguintes funcionalidades: cadastrar o cliente, aprovar o crédito, cadastrar o produto, controlar o estoque de produtos, disparar o processo de compras de produtos de terceiros, registrar os pedidos do cliente e controlar a cobrança. O segundo módulo é responsável em fornecer informações ao setor de Atendimento a Clientes, como por exemplo: situação atual dos pedidos, situação dos clientes, saldo disponível em estoque, informações de faturamento, informações financeiras e histórico dos clientes. Este sistema possui cerca de 1600 programas fontes, totalizando 160.000 linhas de código escritas em *Progress 4GL* versão 7.3, com interface caractere.

Segue-se uma apresentação de cada passo da estratégia proposta para reengenharia deste sistema.

4.1 Organizar Código Legado

A aplicação da estratégia tem início com o código fonte do sistema legado. Neste

passo faz-se a organização do código legado segundo os princípios da orientação a objetos, utilizando os transformadores intradomínio *ProgressToKB* e *ProgressKBToUML* do sistema de transformação *Draco-PUC*.

A Figura 6 mostra, na parte superior esquerda, o código legado de entrada, na parte superior à direita, o código após a organização, e abaixo a seqüência de cada uma das atividades deste passo. Numa primeira atividade foi definida a hierarquia de chamadas das unidades de programa. Baseado na hierarquia de chamadas, criam-se os supostos métodos. Em seguida, são recuperadas as supostas classes do sistema legado, com seus supostos atributos e supostos relacionamentos. Cada suposto método é alocado em uma suposta classe. Na quarta atividade, o engenheiro de software define o início e o nome de cada caso de uso do sistema. Finalmente, na última atividade, os supostos métodos que não fazem parte de nenhuma seqüência de execução de Caso de Uso, são alocados em uma suposta classe principal.

O engenheiro de software, baseado em informações sobre o sistema legado, pode alterar a base de conhecimento para melhorar o significado dos nomes das supostas classes, supostos atributos e supostos métodos recuperados.

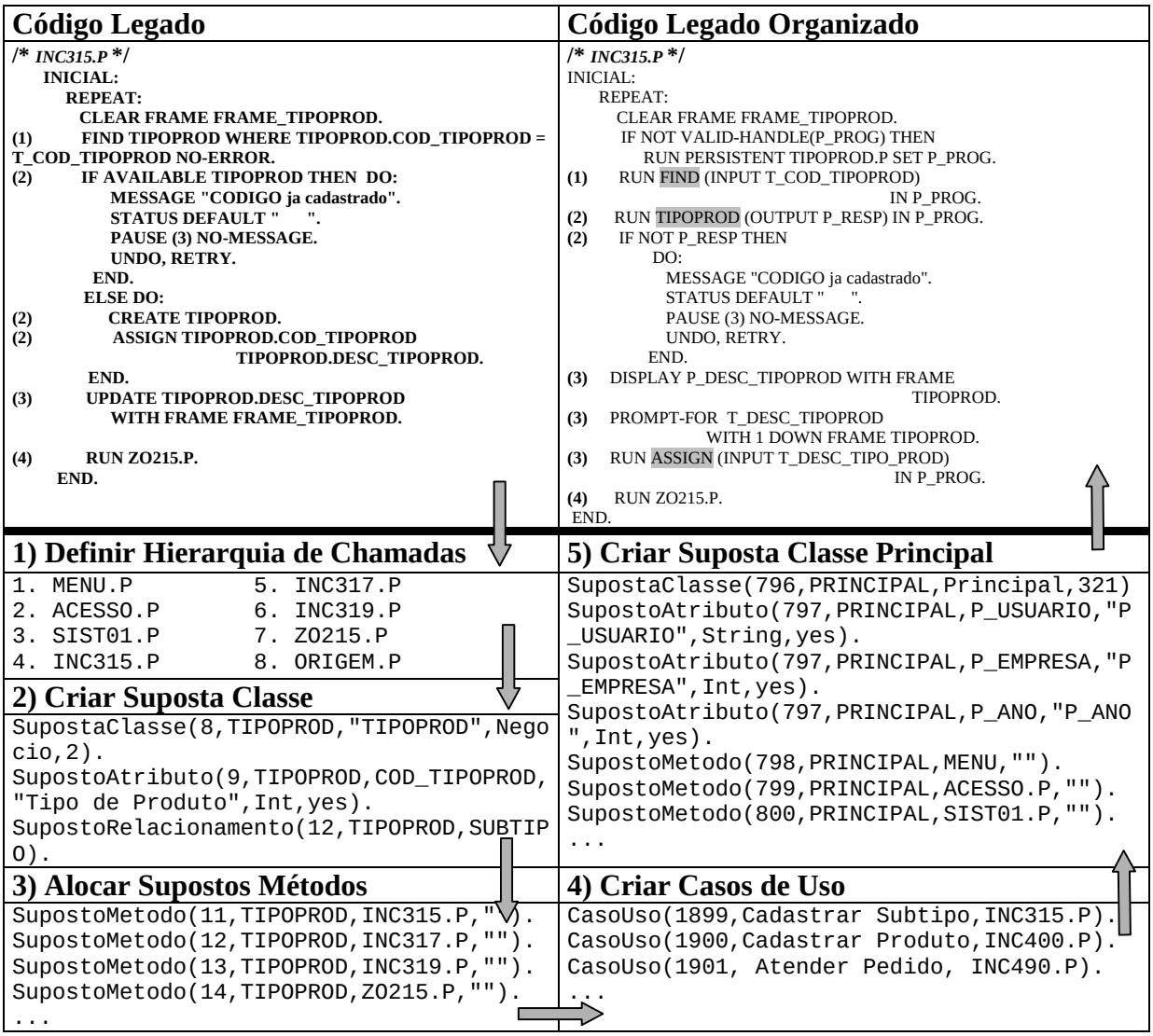

Figura 6 – Atividades de organização do código legado *Progress 4GL*

Depois de concluídas as transformações, tem-se o código legado organizado segundo os princípios da orientação a objetos.

4.2 Gerar Especificações Orientadas a Objetos

Neste passo, aplica-se o transformador interdomínio *ProgressToCatalysisJava* para gerar as especificações em *Catalysis* e *Java*. As supostas classes, supostos métodos, supostos relacionamentos, casos de uso e diagramas de seqüência, recuperados do código, legado são transformados para especificações em *Catalysis*. O código *Progress 4GL* do corpo de cada suposto método é transformado diretamente para *Java*. Dessa forma, as especificações em *Catalysis* das classes geradas têm embutidas na parte semântica dos métodos o respectivo código *Java*.

Para melhor entendimento deste passo apresenta-se, na Figura 7, um exemplo de transformação onde, à esquerda, tem-se o código legado organizado e à direita, a correspondente descrição *Catalysis*. A figura destaca, o trecho de código organizado que é transformado diretamente para *Java*. Este código fica embutido na parte semântica do método construtor *TIPOPROD*.

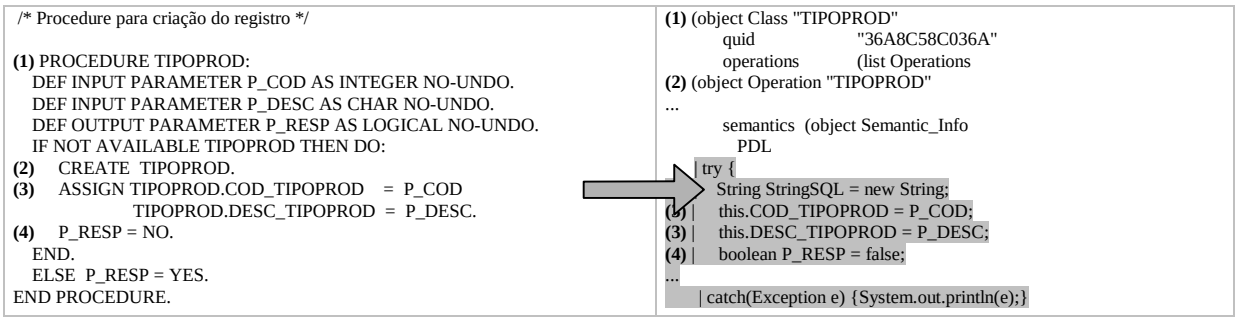

Figura 7 – Transformação do código legado organizado para especificações *Catalysis*

As especificações na linguagem de modelagem *Catalysis*, geradas neste passo, permitem obter o projeto orientado a objetos do sistema na ferramenta *MVCASE*, conforme apresenta o próximo passo da estratégia.

4.3 Especificar Componentes

Neste passo, inicialmente, o engenheiro de software, usando a *MVCASE*, importa as especificações na linguagem de modelagem *Catalysis* para obter o projeto orientado a objetos do sistema. Este projeto é representado pelos modelos do primeiro nível de *Catalysis*. Em seguida, o engenheiro de software reprojeta estes modelos para especificar os componentes do sistema.

Assim, refina-se os Casos de Uso, e obtém-se o Modelo de Colaboração. Um novo Modelo de Tipos é especificado para representar os componentes do sistema. Do refinamento do Diagrama de Seqüência, obtém-se um novo Diagrama de Seqüência com as conexões de mensagens dos componentes.

A Figura 8 mostra, à esquerda, um Caso de Uso do sistema e à direita, o correspondente Diagrama de Seqüência, com as interações entre o ator Vendedor e os componentes especificados.

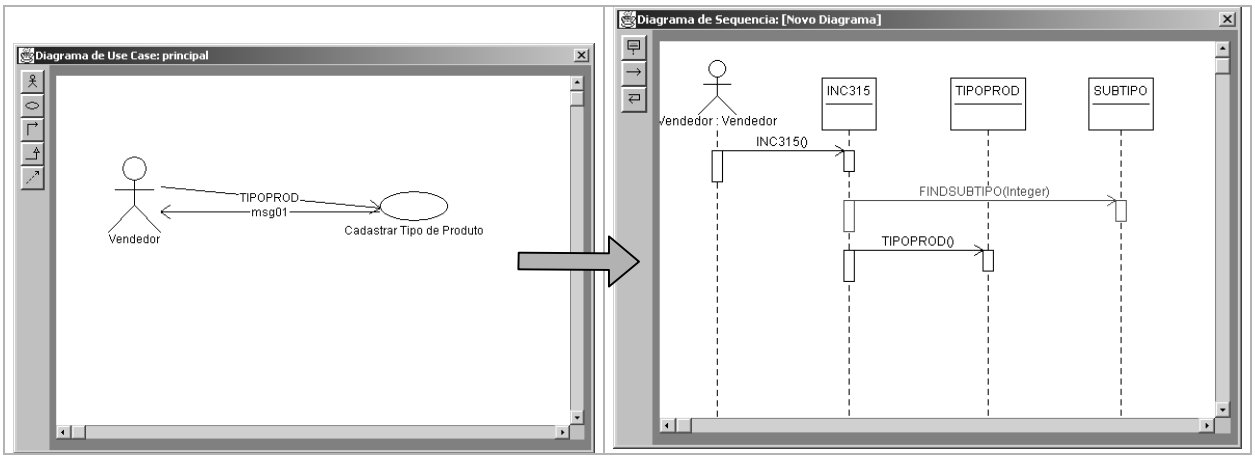

Figura 8 –Especificação de um componente na *MVCASE*

4.4 Projetar Componentes

Neste passo, o engenheiro de software usa a ferramenta *MVCASE* para projetar os componentes especificados em *Catalysis*, refinando os modelos do passo anterior. Assim, do refinamento do Modelo de Tipos, obtém-se o Diagrama de Classes dos componentes e do refinamento do Diagrama de Seqüência, obtém-se o Diagrama de Interação dos componentes.

A Figura 9 mostra, à esquerda, um modelo de tipos e à direita, o projeto de um componente *EJB*. Ao selecionar a opção *Gerar Entity Bean*, a *MVCASE* gera automaticamente as interfaces e métodos de suporte para o componente.

A *MVCASE* também gera o *Deployment Descriptor* dos componentes, criando os scripts *XML* de configuração dos componentes que poderão ser alterados pelo engenheiro de software para distribuí-los.

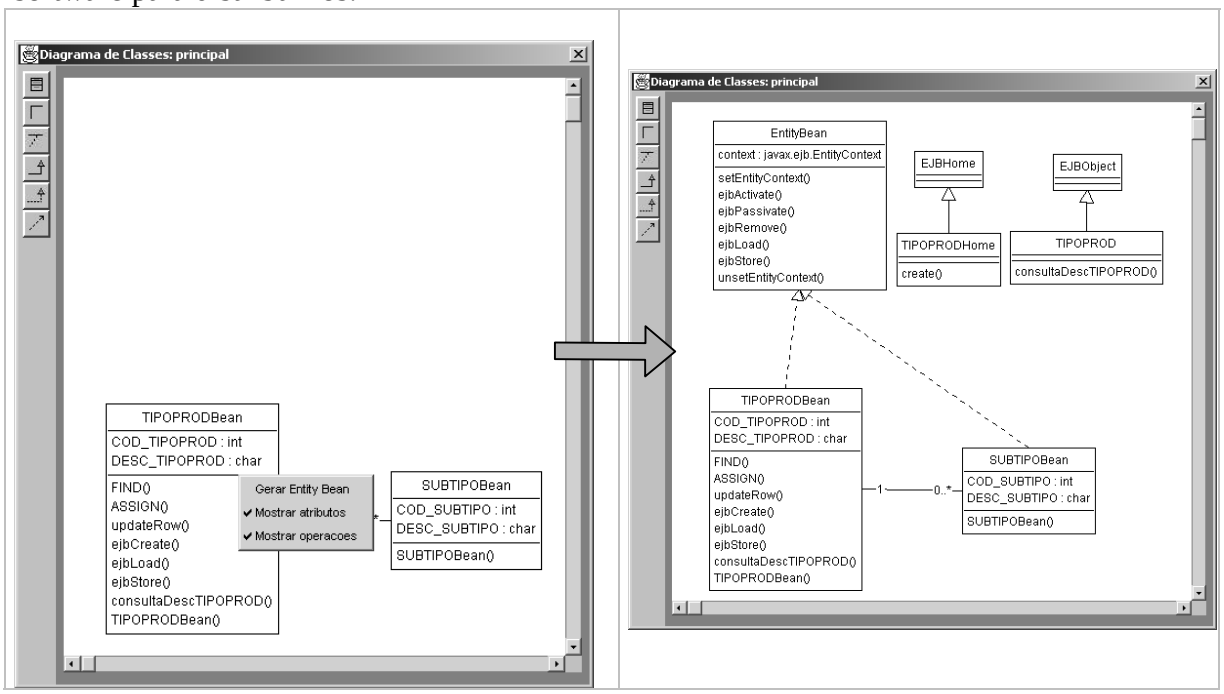

Figura 9 – Projeto dos componentes *EJB*

Com o projeto do sistema em *Catalysis*, pode-se compreendê-lo e reprojetá-lo para atender novos requisitos.

4.5 Reimplementar Sistema

No último passo da estratégia, são aplicadas as transformações da linguagem de modelagem *Catalysis* para a linguagem de programação *Java*. A *MVCASE* reimplementa em

Java os Diagramas de classes do sistema reprojetado Na reimplementação, a geração de código *Java* da *MVCASE* trata os métodos das classes, cujos comportamentos já estão especificados em *Java* e embutidos na descrição *Catalysis*. O código *Java* gerado a partir das especificações *Catalysis* é integrado com o código *Java* dos métodos para obter a implementação final do sistema. A Figura 10 mostra, à esquerda, parte do código *Java* gerado e à direita, a correspondente interface, após a reengenharia.

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 o sistema.

A funcionalidade do sistema legado é mantida no sistema reimplementado com a vantagem de facilitar a manutenção, uma vez que o código está organizado e encapsulado em componentes. Como 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.

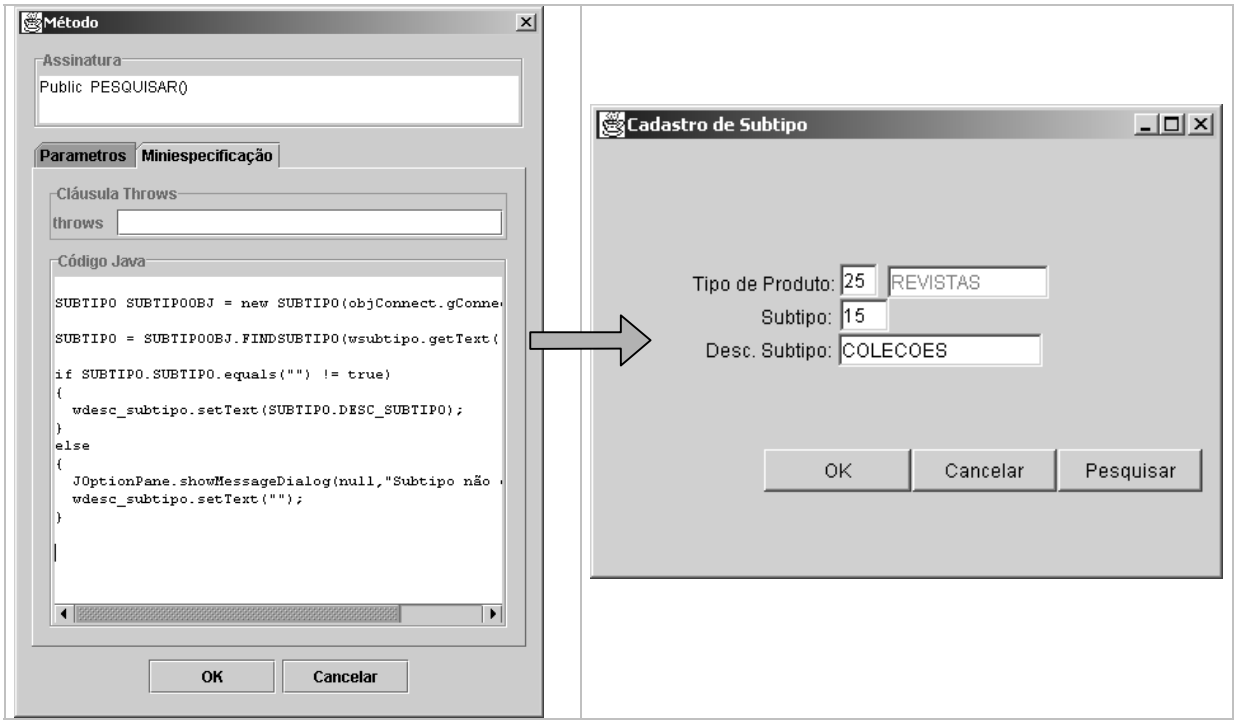

Figura 10 – Reimplementação final Orientada a Componentes Distribuídos

5. Conclusão

Este artigo apresentou uma estratégia para a reconstrução de sistemas legados orientados a procedimentos para sistemas orientados a componentes distribuídos, sem perda da funcionalidade e com a possibilidade de reprojeto. A estratégia objetiva reconstruir sistemas legados, escritos em linguagem procedural, para serem executados em plataformas mais modernas de hardware e software, facilitando a manutenção através do reuso de Componentes Distribuídos.

Esta estratégia cobre todo o ciclo da reengenharia de um software não orientado a objetos, destacando-se: recuperação do projeto do sistema atual e posterior reprojeto orientado a componentes, em uma ferramenta *CASE*, com reimplementação automática em uma linguagem orientada a objetos. Diferentes técnicas, destacando-se o método *Fusion-RE*, o método *Catalysis*, uma ferramenta *CASE*, *Enterprise JavaBeans* (*EJB*) e um Sistema de Transformação foram integrados para suportar a Reengenharia.

Dada a capacidade do Sistema de Transformação *Draco-PUC* de trabalhar com uma rede de domínios com diferentes linguagens, a estratégia pode ser aplicada a sistemas legados implementados em diferentes linguagens, diferentes de *Progress 4GL*, como por exemplo, *Clipper* e *Cobol*. A linguagem alvo da implementação também pode ser outra, diferente de *Java*, como por exemplo, *Object Pascal* e *C++*.

A aplicação sistemática desta estratégia em diferentes empresas e áreas de negócios irá tornar a Reengenharia de Software Orientada a Componentes Distribuídos um importante fator de evolução tecnológica, diminuindo a defasagem dos sistemas de informação em relação aos seus requisitos. Sistemas legados de diferentes domínios de aplicação podem ser reconstruídos usando a estratégia proposta, criando componentes reutilizáveis em novas aplicações destes domínios.

Referências

[1] 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.

[2] 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.

[3] D'SOUZA, D.; WILLS, A. *Objects, Components and Frameworks with UML – The Catalysis Approach.* USA:Addison Wesley, 1998.

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

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

[6] GAMMA, E. et al. *Design Patterns*. Elements of Reusable Object-Oriented Software. Ed. Addison-Wesley. USA.1995.

[7] JACOBSON, I. et al. Software Reuse - architecture process and organization for business success. Ed. Addison-Wesley, New York, 1997.

[8] Sun Microsystems. Tutoriais *Java*, www.*Java*soft.com, 2001.

[9] 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.

[10] Barrere, T.S., Prado, A.F., Bonafé V.C., CASE Orientada a Objetos com Múltiplas Visões e Implementação Automática de Sistemas – MVCASE, 1999

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

[12] PHP:Hypertext Preprocessor, www.php.net, 2001.

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

[14] 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.

[15] PRADO, A. F., NOVAIS, E. R. A. Reengenharia Orientada a Objetos de Código Legado Progress 4GL, *XIV Simpósio Brasileiro de Engenharia de Software*, pg. 21-36 - Outubro de 2000.

[16] 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.

[17] BOOCH, GRADY, *UML* Guia do Usuário, Editora Campus, 2000.

[18] Valeskey, T., Enterprise *Java*Beans, - Developing Component-Based Distributed Applications. ADDISON-WESLEY, 1999.

[19] WERNER, C.M.L., BRAGA, R. M.M. Desenvolvimento Baseado em Componentes. *XIV Simpósio Brasileiro de Engenharia de Software* SBES2000 – Minicursos e Tutoriais – pg.

297-329 – 2-6 de Outubro, 2000.

[20] www.w3.org/xml